Ministère de l'enseignement Supérieur et de la recherche Scientifique
وزارة التعليم العالي والبحث العلمي

Badji Mokhtar Annaba University
Université Badji Mokhtar – Annaba
Faculté de Technologie
Département Informatique

جامعة باجي مختار – عنابـــــــــة
كلية التكـنولوجـيا
قســـــــم الإعلام الآلي

Thèse

Présentée pour obtenir le diplôme de

Doctorat En-Sciences

Spécialité : Informatique

Par :
MENASRIA Ahcene

Thème :

Abstraction de la communication dans les architectures logicielles

Thèse soutenue le 20/06/2022 devant le jury composé de :

N°	Nom et prénom	Grade	Etablissement	Qualité
01	Souici-Meslati Labiba	Prof.	Université Badji Mokhtar -Annaba	Président
02	Bahi Halima	Prof.	Université Badji Mokhtar -Annaba	Rapporteur
03	Amirat Abdelkrim	Prof.	Université de Souk Ahras	Co-rapporteur
06	Chikhi Salim	Prof.	Université Constantine 2	Examinateur
05	Chaoui Allaoua	Prof.	Université Constantine 2	Examinateur
04	Atil Fadhila	Prof.	Université Badji Mokhtar -Annaba	Examinateur
07	Oussalah Mourad	Prof.	Université de Nantes- France	Invité

Dédicaces

Je dédie cette thèse à :

La mémoire de mon père, qui avec le peu qu'il avait, sa sagesse avait fait l'essentiel,

Mon premier et éternel supporter scientifique : Ma mère, qui n'a jamais été à l'école,

Mes enfants : Med Aniss, Med Amir et Takou pour qui je dis que le synonyme de la réussite c'est l'effort. Par la même, j'essaye de vous donner « une éducation par l'exemple ».

Remerciements,

Au terme de cet effort, j'aimerai bien exprimer toute ma gratitude, mon profond respect et ma sincère reconnaissance à tout un chacun de mes encadrants : Pr Mourad Oussalah, Pr Halima Bahi et Pr Abdelkrim Amirat. Aucun parmi eux, et en quelque situation que se soit, n'a jamais retenu aucun effort, conseil, soutien et apport scientifique pour que ce travail aboutisse. J'ai eu la chance d'avoir été si bien entouré.

Je tiens à remercier le Pr Mourad Oussalah en particulier pour m'avoir accueilli dans son laboratoire (LS2N de l'Université de Nantes) et pour m'avoir ouvert toutes les portes de la France scientifique. Merci pour toute votre patience, tous vos encouragements, toutes les grandes idées et l'apport scientifique certain. Merci pour m'avoir accordé les longues discussions scientifiques hebdomadaires. Merci d'avoir supporté mes errances et mes maladies. Je regrette juste une chose, c'est d'avoir quitté sans vous dire au revoir. Nous nous sommes quittés le 15/03/2020 pour nous retrouver dans 15 jours après la première fermeture liée au Coronavirus, hélas nous nous sommes plus revus depuis. Ceci dit, je retiens la leçon à jamais: « L'effort finit toujours par payer ».

Je tiens également à remercier ma sœur Pr Halima Bahi pour sa bonté et sa sincérité. Merci d'avoir été là pour débloquer toutes les situations. Merci pour ton soutien indéfectible, tes encouragements. Merci pour l'accompagnement scientifique et administratif que tu m'as fraternellement accordé tout le long de cette thèse. Merci de m'avoir donné la force d'aller jusqu'au bout. Prière de croire à l'estime éternelle que j'ai à votre égard.

Je remercie aussi mon vieil ami, qui rentre sans frapper, le Pr Abdelkrim Amirat pour tout ce qu'il a fait pour moi durant cette thèse. Merci d'avoir eu les clés à toutes les situations bloquantes. Merci pour les longues discussions scientifiques que nous avons eu autour d'une tasse de café. Durant ces sept années, le Pr Amirat a été la seule personne au courant dans le détail de toutes les douleurs qui ont accompagné cette thèse. Merci de m'avoir écouté, de m'avoir conseillé et de m'avoir des fois brandit le bâton. J'espère seulement que notre cap de réussite scientifique de M'Daourouch ne se termine jamais.

Mes remerciements vont également aux membres du jury qui m'ont accordé l'honneur et le privilège de lire et d'évaluer cette thèse. Je pense particulièrement au :

- Pr Souici-Meslati Labiba pour avoir accepté sans hésitation ni réserve aucune de présider le jury de soutenance.
- Pr Atil Fadila que j'ai rencontré à plusieurs reprises dans la littérature du domaine.
- Pr Allaoua Chaoui, « Osthadhi » qui m'a toujours appris l'essentiel du génie logiciel. Il reste pour moi une grande école et une grande référence du domaine.
- Pr Salim Chikhi, qui m'a accordé un privilège particulier, celui de me juger deux fois dans ma carrière scientifique.

A vous tous, je reste à vie extrêmement reconnaissant.

Je ne peux oublier également tout le personnel administratif et scientifique du laboratoire LS2N de l'université de Nantes. Je cite :

- Le directeur, Pr Claude Jare

- L’équipe AeLos « Architecture et Logiciels sûrs » pour tous les séminaires hebdomadaires et la qualité des discussions et interventions. Je veux citer en particulier Pr Jérémie Christian ATTIOGBE, Pr Mourad Oussalah, les docteurs Benoit DELAHAYE, Pascal ANDRE, Gilles ARDOUREL, Arnaud LANOIX. Sans oublier les doctorants qui m’ont accompagné durant ce long séjour, je cite en particulier : Kadi, Bah, Dimitri, Faouzi, Eva et Abderhmane.
- Le personnel administratif et en particulier : Mme Annie Boillot pour avoir assuré les nombreuses et longues tâches administratives, Mr Charlerie pour le support informatique et Mme Christine Brunet pour la logistique.

Je cherche par la présente à remercier les membres de l’association d’intersection des chercheurs étrangers à Nantes (Euraxess) ainsi que la maison des chercheurs étrangers à Nantes (MCE) pour leurs précieux concours durant mon séjour à Nantes. Je veux remercier en particulier Mmes Morgane, Soline et Marry. Merci à vous toutes.

Je ne saurais également pas oublier les services de Mme Patricia Bossart, la responsable de la résidence Wangari Maathai.

Je tiens vivement à remercier mes étudiants de génie logiciel à l’université de SoukAhras devenus sans surprise des ingénieurs très courus par les entreprises de développement à Paris et à Dijon. Je pense à Mohammed Tebib et Khaled Amirat qui ont eu un apport considérable dans cette thèse.

Je remercie aussi mes amis Dr Nourdine Gasmallah pour le retour d’expérience à chaque fois et Dr Ammar Ladjailia pour m’avoir alimenté en documentations rares et inaccessibles.

Aussi, je ne peux aussi oublier mes médecins traitants qui m’ont « réparé » à chaque fois. Je pense en particulier à : Dr Anne Quillard et Dr Maeva Batista ainsi que tout le personnel médical des urgences du CHU de Nantes et de la PASS.

Je remercie sincèrement aussi les services services français pour toutes les aides et facilitations qu’ils m’ont accordées. Je cite : Préfecture et Mairie de Nantes, la CAF, Ameli, mesServices, Tan et izly.

Je remercie ma femme Sana, mes enfants Med Aniss, Med Amir et Takou pour avoir su supporté et géré ma longue absence.

Je remercie toute ma famille pour son soutien et ses encouragements de toujours: ma mère el hadja Badra bent youcef, mes sœurs et frères Souad, Dalila, Atika, Salah et Nacer. Toutes mes nièces et tous mes neveux, en particulier Imad. Mes belles sœurs et beaux frères.

Je remercie mes amis à Nantes qui m’ont reproduit l’atmosphère de SoukAhras à Nantes surtout durant le Ramadan. Je pense en particulier à Mourad, Hassen et Mohammed.

A vous tous, MERCI

Résumé

Le monde actuel tend vers une orientation extrêmement interactive. Les applications modernes sont de plus en plus communicantes entre elles ainsi qu'avec leur environnement ; les voitures contemporaines en sont l'exemple typique. Cette fonction d'interaction atteint souvent des proportions équivalentes par rapport à leurs fonctions intrinsèques de calcul et de traitement. De ce fait, la communication devient un facteur important qui participe à l'accroissement de la complexité et de l'hétérogénéité et par conséquent elle devient un réel défi d'interopérabilité. Ceci dit, la communication peut ne pas être un simple transfert d'information de point à point mais un processus complexe. Dans cette thèse, il est donc question de réfléchir et de proposer un modèle conceptuel pour décrire la communication dans les architectures logicielles. Ce modèle doit être assez générique pour traverser les architectures logicielles à travers les différents paradigmes de développement et traduire fidèlement le monde réel de la communication. Aussi, nous ne considérons plus la communication un artéfact ou un produit mais plutôt comme un processus logiciel entier.

Pour cette fin, nous nous sommes intéressés aux architectures logicielles à base de composants où l'abstraction de la communication est la plus aboutie via le concept de connecteur. Après une étude exhaustive des différentes approches qui considèrent le connecteur comme une entité de première classe, nous avons proposé notre propre connecteur (CaP) à base de processus logiciel. En utilisant les techniques de méta-modélisation, nous avons proposé un connecteur sur étagère sous forme d'un composant de communication ayant des interfaces et des services de communication particuliers. Au même titre qu'un composant, le connecteur est doté d'une nouvelle structure et d'un comportement. Nous avons intégré les éléments connus dans la communauté processus logiciel au sein même du nouveau connecteur et ce à travers les concepts et les extensions du standard ou profil SPEM (*Software & Systems Process Engineering Méta-model*). Cette proposition se veut être une fusion entre les domaines d'architectures et de processus logiciels. Nous avons également proposé notre propre langage de description d'architecture (capADL) à base de processus qui montre la faisabilité de notre contribution. À partir de notre ADL, nous avons proposé aussi une transformation de modèle à travers une grammaire de graphe pour générer le squelette de l'application en code java à partir de l'architecture initiale. L'ensemble de la proposition est validé par deux exemples pris chez les deux communautés à savoir un modèle de cycle de vie en « V » et le patron bien connu Pipe&Filtre.

Mots clés : Architecture Logicielle, Processus Logiciel, Méta-Modélisation, Connecteur Logiciel, Profil SPEM, ADL, Transformation de Modèle, Grammaire de graphe.

Abstract

Today's world tends towards an extremely interactive orientation. Modern applications are increasingly communicating with each other as well as with their environment; contemporary cars are a case in point. This interaction function often reaches equivalent proportions with respect to their intrinsic functions of calculation and processing. As a result, communication becomes an important factor that contributes to the increase in complexity and heterogeneity, and conséquently, it becomes a real challenge for interoperability. However, communication may not be a simple point-to-point transfer of information but a complex process. In this thesis, it is therefore a question of thinking and proposing a conceptual model to describe communication in software architectures. This model must be generic enough to cross software architectures across the different development paradigms and faithfully translate the real world of communication. Also, we no longer see communication as an artifact or a product, but rather as an entire software process.

To this end, we were interested in component-based software architectures where the abstraction of communication is most successful through the concept of a connector. After an exhaustive study of the different approaches which consider the connector as a first-class entity, we have proposed our own connector (CaP) based on software processes. Using méta-modeling techniques, we proposed an on-the-shelf connector as a communication component with particular communication interfaces and services. Like a component, the connector has a new structure and behavior. We have integrated the éléments known in the software process community within the new connector itself, through the concepts and extensions of the SPEM (Software & Systems Process Engineering Métamodel) standard or profile. This proposal is intended to be a merger between software architecture and software process domains. Also, we have proposed our own process-based Architecture Description Language (capADL) which shows the feasibility of our contribution. From our ADL, we also proposed a model transformation through a graph grammar that generates a skeleton of the application in java code from the initial architecture. The whole proposition is validated by two examples taken from the two communities, namely a "V" life cycle model and the well-known Pipe & Filter pattern.

Key words: Software Architecture, Software Process, Méta-Modeling, Software Connector, SPEM Profile, ADL, Model Transformation, Graph Grammar.

الملخص

يميل عالم اليوم نحو اتجاه تفاعلي للغاية. تتواصل التطبيقات الحديثة بشكل متزايد مع بعضها البعض وكذلك مع بيئتها ؛ السيارات المعاصرة هي مثال على ذلك. غالبًا ما تصل وظيفة التفاعل هذه إلى نسب مكافئة فيما يتعلق بوظائفها الجوهرية في الحساب والمعالجة. نتيجة لذلك ، يصبح عنده الاتصال عاملاً مهمًا يساهم في زيادة التعقيد وعدم التجانس ، وبالتالي ، يصبح تحديًا حقيقيًا للتشغيل البيني. ومع ذلك ، قد لا يكون الاتصال مجرد نقل للمعلومات من نقطة إلى نقطة ولكنه عملية متكاملة و معقدة. المطلوب في هذه الأطروحة إذا هي مسألة تفكير واقتراح نموذج مفاهيمي لوصف الاتصال في معماريات البرمجيات. يجب أن يكون هذا النموذج عامًا بما يكفي لعبور هياكل البرامج عبر نماذج التطوير المختلفة وترجمة عالم الاتصال الحقيقي بأمانة. أيضًا، لم نعد نرى الاتصال على أنه عبارة عن منتج برمجي ، بل هو عملية برمجية كاملة.

تحقيقًا لهذه الغاية، كنا مهتمين بهياكل البرامج القائمة على المكونات حيث يكون تجريد الاتصال أكثر نجاحًا من خلال مفهوم الموصل. بعد دراسة شاملة للطرق المختلفة التي تعتبر الموصل كيانًا من الدرجة الأولى ، اقترحنا الموصل الخاص بنا (CaP) استنادًا إلى عمليات البرامج. باستخدام تقنيات النمذجة الوصفية ، اقترحنا موصلًا على الرف كمكون اتصال مع واجهات وخدمات اتصال معينة. مثل المكون ، الموصل له هيكل وسلوك جديدان. لقد قمنا بدمج العناصر المعروفة في مجتمع عملية البرمجيات داخل الموصل الجديد نفسه ، من خلال مفاهيم وامتدادات معيار أو ملف تعريف SPEM (Software & Systems Process Engineering Métamodel). يهدف هذا الاقتراح إلى أن يكون دمجًا بين هندسة البرمجيات ومجالات عمليات البرمجيات. أيضًا ، اقترحنا لغة وصف المعمارة القائمة على العمليات (capADL) والتي توضح جدوى مساهمتنا. من ADL الخاص بنا، اقترحنا أيضًا تحويلًا للنموذج من خلال قواعد تحويلات الرسم البياني التي تولد هيكلًا للتطبيق في كود java إنطلاقا من البنية الأولية. تم التحقق من صحة الاقتراح بأكمله من خلال مثالين مأخوذين من المجتمعين ، وهما نموذج دورة الحياة "V" ونمط الأنابيب والفلتر المعروف.

كلمات مفتاحية: هندسة البرمجيات ، عملية البرمجيات ، النمذجة الوصفية، موصل البرامج، ملف تعريف SPEM، ADL، تحويل النموذج، قواعد الرسم البياني

Table des Figures

Liste des tableaux

Table des Matières

Avant propos

Introduction

Dans le monde du développement de logiciels, les architectures logicielles représentent une ligne médiane entre deux mondes distincts : un espace métier et un espace technologique (Figure 1). Cette ligne est loin de décrire une séparation entre ces deux mondes mais plutôt une jonction qui, au-delà de représenter des assises structurelles pour l'application, reflète et dessine un certain nombre de concepts du mode métier.

Reste à dire qu'il est difficile voire impossible de distinguer une architecture dans un système en exécution comme dans un cahier des charges formulé par un client ou un maître d'ouvrage profane en la matière. Le concept « architecture » est donc décelable dans un stade précoce du cycle de développement. Il émerge dans la phase du processus comme un pont entre l'expression des besoins et le codage.

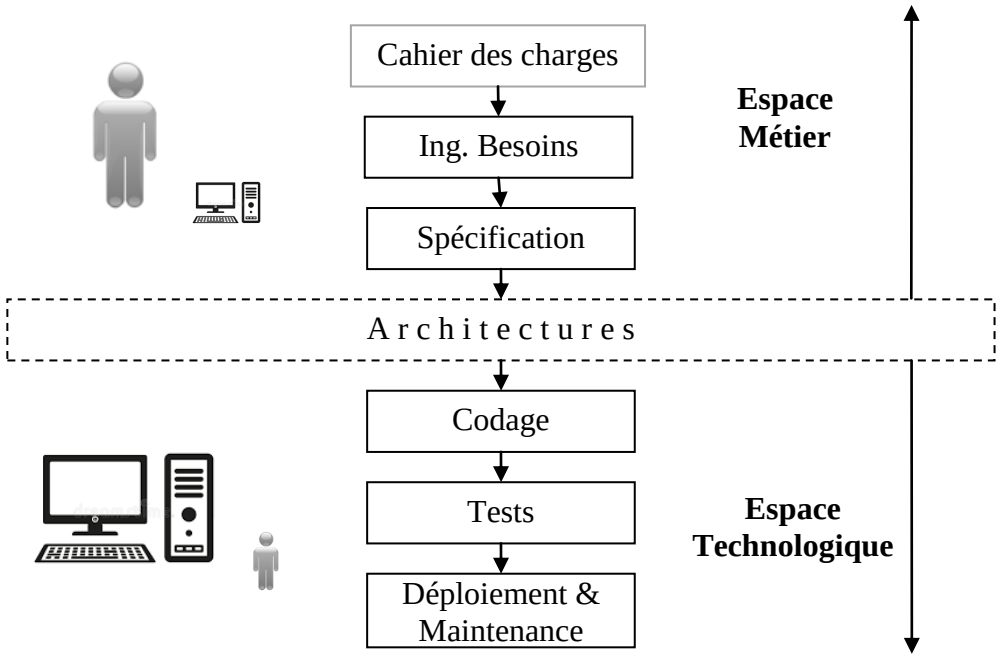


Figure 1.1- Position des architectures logicielles.

Pour autant, les architectures logicielles demeurent comme un niveau fédérateur qui permet de pallier aux aléas de fonctionnement ultérieurs ainsi qu'aux évolutions futures tant autour des fonctions, des critères de qualité et des technologies de mise en œuvre.

Au fil des années, et compte tenu des exigences et des complexités en perpétuelle croissance, on a vu naître une cascade de paradigmes de développement non seulement pour remédier aux insuffisances antérieures mais surtout pour supporter les aspirations humaines sans cesse grandissantes. En informatique, le rêve démesuré n'existe pas dès lors qu'à chaque défi les paradigmes de développement parviennent à avaler et produire des solutions compactes et sûres.

Ainsi, un long chemin sépare le monde procédural de celui du micro-service ou encore le monde de la machine de Turing de celui du « *Cloud Computing* ». Ce chemin à le caractère d'être itératif et incrémental afin d'éviter les sursauts d'orgueil inexpliqués. Créer à partir du néant est une qualité divine et loin d'être humaine. En effet, nous autres ingénieurs, ne savons jamais rien faire sans exemples antécédents.

Cette chaîne d'évolution : Procédure, Objet, Composant, Service, Agent, Model, Cloud et Internet des Objets (IoT) ou encore Micro-Services s'appuient sur un seul et éternel constat

celui de la capitalisation des valeurs ou propriétés antérieures. Chaque paradigme reprend l'essentiel utile de son prédécesseur pour ouvrir de nouveaux horizons afin de faire face aux nouveaux défis. Sans la procédure, ses avantages et inconvénients, on n'aurait guèrre songé à l'objet. Toutefois, un certain nombre d'éléments et de concepts traversent la longueur de cette chaîne et demeurent inchangés le long de ses maillons. L'exemple phare doit, certainement et sans égal aucun, être la communication.

Plus encore, les systèmes actuels tendent à devenir fortement interactifs. La voiture en est le parfait exemple. En plus de sa fonction (avancer, tourner, reculer), de sa structure (carrosserie, roues, moteur, transmission) et de son comportement (rotation moteur, embrayage, changement de vitesse, rotation roues), la voiture moderne est extrêmement communicante et fortement connectée avec son environnement.

Théoriquement, la communication est donc l'ensemble des interactions avec autrui qui transmettent une quelconque information. Il s'agit donc aussi de l'ensemble des moyens et techniques permettant la diffusion d'un message auprès d'une certaine audience. Elle concerne aussi bien l'être humain (communication interpersonnelle, groupale…), l'animal, la plante (communication intra- ou inter- espèces) ou la machine (télécommunications, nouvelles technologies…), ainsi que leurs hybrides : homme-animal, hommes-technologies.

Le domaine de « sciences de l'information et de la communication », propose une approche de la communication centrée sur la transmission d'informations. Le concept de communication est un élément central pour chaque paradigme. Sans cet élément, aucune interaction entre les constituants d'un système ou ceux de l'environnement n'est possible. Ce qui donne des systèmes inertes, oisifs et inutiles.

Ainsi, la communication est un dénominateur commun à tous les paradigmes de développement de par sa nécessité en premier lieu que par ses éléments. Cependant, il existe des distinctions significatives qui peuvent être formulées sous deux angles : Les moyens et les techniques de mise en œuvre ainsi que le degré de « *communicabilité* » où l'on passe du simple appel de procédure dans le procédural aux langages et services de communication à part entière « CaaS » dans le « *cloud computing* » qu'avec l'internet des objets.

La problématique initiale discutée dans le cadre de cette thèse est l'abstraction de la communication dans les architectures logicielles. De ce fait, deux concepts principaux apparaissent comme étant directement impliqués dans cette étude à savoir les « architectures logicielles » et la « communication ». Aussi, le concept « abstraction » peut être considéré comme une notion qu'il faut développer également à partir du moment qu'il représente l'outil ou l'accessoire pour la production du modèle de communication attendu.

L'abstraction de la communication étant par définition un sujet épineux et en même temps enthousiaste et passionnant. Dans son livre « *Art as a social systems* », le célèbre sociologue Niklas Luhman (2000) avance une motivation de taille à ce sujet ; il dit : *« Une théorie de la communication doit être développée dans le domaine de l'abstraction. Etant donné que la physique a franchi ce pas dans la théorie de la relativité et de la mécanique quantique, l'abstraction ne devrait pas être en soi un obstacle ».*

Aux premiers abords de cette thèse, nous avons considéré les architectures logicielles au sens large. Nous avons essayé de considérer les modèles de communication dans chacun des paradigmes de développements avec l'intention de proposer un modèle unificateur à la fin du processus. Or, le niveau de détails ainsi que la distance sémantique entre les paradigmes ont rendu cette démarche peu fructueuse. Nous avons donc décidé de procéder par l'inverse, soit de partir d'un modèle de communication riche et explicite pour essayer de l'enrichir afin de servir et valoir de moyen de communication inter-paradigmes.

Pour cela, nous avons donc opté pour l'étude du modèle de communication dans le paradigme de développement orienté composant. En génie logiciel, ce paradigme de développement en particulier est connu pour avoir une maturité avérée en termes de séparation des préoccupations entre la partie fonctionnelle et la partie interactionnelle. Ainsi, le paradigme à base de composants considère une configuration ou un système architectural comme un ensemble de « composants » autonomes dont la fonctionnalité globale émerge de leurs interactions. Cette interaction est, elle aussi, assurée par des entités autonomes dédiées qui ont pris une importance croissante dans le développement des systèmes complexes.

Aussi, dans la pratique, la notion d'architecture logicielle est par définition souvent confondue au développement orienté composant qu'au reste des paradigmes. A ce niveau, une question légitime peut se poser, « Mais pourquoi le concept d'architecture logicielle est aussi intimement lié au paradigme composant ? ». Comparé à l'objet, nous pensons que les architectures logicielles sont un synonyme idoine des développements orientées composants pour deux raisons essentielles :

(i) la granularité : les objets par exemple sont assez fins pour former un élément architectural qui peut être impliqué directement dans une architecture. Un parpaing ne peut être un élément architectural alors que les murs construits de parpaing peuvent l'être.

(ii) la représentation : les diagrammes de classes ne peuvent pas représenter tous les aspects de l'application. Ils ne représentent qu'une vue structurelle du système, quant à la vue interactionnelle elle est déléguée à d'autres diagrammes séparés. Le diagramme de composant en revanche représente conjointement la partie structure et la partie interaction. Ce qui fait de ce diagramme une représentation complète, précoce et fidèle de l'application entière en cours de développement.

Pour assurer sa finalité essentielle de réutilisation, le composant en tant que boîte noire –par définition- ne peut être modifié en fonction de l'environnement particulier ou des fonctionnalités globales des systèmes auxquels il appartient. Il ne peut qu'afficher ses services à travers ses interfaces. Pour construire un système à base de composants, la solution est désormais assurée par une opération de composition utilisant des connexions particulières entre les composants impliqués. Ces connexions sont réalisées par des éléments indépendants de l'application ; communément appelés : « connecteurs ».

A travers son historique, le connecteur a évolué donc en fonction des demandes en interaction plus ou moins complexe ainsi que les besoins d'abstraction et de réification. De ce fait, on vu passé le connecteur d'un statut implicite à un statut explicite pour devenir enfin un élément architectural au même titre que le composant. Ce statut doit être désormais conservé tout le long du cycle développement du système c'est-à-dire de la conception au déploiement.

Comme tout élément architectural, un connecteur logiciel à une structure et un comportement. La structure représente les constituants élémentaires et leur organisation alors que le comportement décrit toujours l'aspect fonctionnel et dynamique du connecteur.

Le rôle déterminant du connecteur dans les architectures à base de composants lui a conféré un impact critique, gagnant ainsi en importance au fil du temps pour devenir une entité de premier ordre dans les différentes phases du cycle de vie du développement à base de composants (Mary Shaw, 1993). A ce titre, le concept de connecteur est intégré comme un élément architectural à part entière dans les standards ainsi que dans les langages dédiés principalement les langages de description d'architecture (ADLs) (Medvidovic & Taylor, 2000).

Pour plus de détails, le connecteur est un lien entre les interfaces des composants en interaction et peut en outre gérer la communication (transfert de données) et la coordination (transfert de contrôle) du composant (Fujita & Herrera-Viedma, 2018). De plus, les connecteurs peuvent représenter des fonctionnalités plus complexes pour agir en tant qu'adaptateurs, distributeurs et parfois arbitres pour garantir la qualité de l'interaction en assurant certaines propriétés telles que la performance et la sécurité (Mehta et al., 2000).

En pratique, l'intérêt dans la recherche comme dans l'industrie s'est souvent porté sur les composants ou la partie fonctionnelle. Dans des environnements fortement interactifs, les applications récentes sont de plus en plus communicantes pour atteindre des proportions souvent équivalentes entre la partie interaction par rapport à la partie fonctionnelle. Richard Taylor (2019) dit : « Les connecteurs méritent particulièrement d'être étudiés, car ils sont très variés et puissants, et ils sont largement sous-estimés dans la recherche et dans l'enseignement informatique typique ».

La communauté des architectures logicielles (Fujita & Herrera-Viedma, 2018; Mehta et al., 2000; Shaw & Clements, 2006; Taylor, 2019) considère que tous les connecteurs, quelle que soit leur complexité, ne sont autres que des mécanismes de transfert de données et de transfert de contrôle via une conduite ou un canal. Ce même constat est partagé par la communauté des processus logiciels (Aoussat et al., 2014; Dai et al., 2008; Dami et al., 1998; Ismail et al., 2017) qui définit les connecteurs de transfert de données et les connecteurs de transfert de contrôle séparément pour réussir sa quête de réutilisation des processus logiciels.

Compte tenu de ce qui précède, notre approche est un type intégral d'approches centrée sur le connecteur. Partant du constat que la communication est un processus dynamique et un phénomène de plus en plus complexe, elle ne peut donc plus être modélisée par un produit logiciel ou un artefact. Comme palliatif, nous estimons que la notion de processus logiciel peut être imaginée comme modèle pour la communication capable d'absorber les disparités syntaxiques et sémantiques entre les entités en interactions. La notion de processus nous semble comme le concept qui traduit le plus fidèlement la réalité courante de la communication.

En général, les théories et modèles de communication (Lasswell & Tréanton, 1952; Shannon & Weaver, 1963) identifient certains éléments et mécanismes communs dans le processus de communication. Ils peuvent être résumés comme suit : i) (Quoi) Produit communiqué entre deux ou plusieurs entités en entrée/sortie. ii) (COMMENT) Activités pour réaliser cette communication. iii) (QUI) Contrôleur dont le rôle est d'assurer le succès de cette communication.

Dans la communauté des processus logiciels, nous rencontrons exactement la même structure des modèles de communication avec des éléments ayant la même sémantique dans le concept de processus logiciel ("WorkProduct", "WorkUnit", et "Role"). Réunir et fusionner les deux concepts semble être un chemin de recherche prometteur à explorer.

Ainsi, il nous paraît évident que modéliser le monde réel de la communication à travers un processus logiciel est une voie naturelle et prometteuse pour surmonter les multiples disparités conceptuelles et techniques. De plus, « WorkUnit » peut être une tâche complexe et, par conséquent, elle peut être décomposée en plusieurs étapes. En parallèle, modéliser la communication dans les architectures logicielles comme un produit logiciel ou artefact large et fermé (connecteur classique) est assez limitatif et ne traduit pas fidèlement la véritable nature du processus de communication.

L'utilisation du processus logiciel pour modéliser les connecteurs nous permet en outre d'analyser et vérifier facilement certaines propriétés de processus logiciel connues. Aussi,

cette orientation peut nous permettre d'appliquer certaines guidances et d'utiliser la boîte à outils et les profils disponibles. Ainsi, tous les atouts des processus logiciels peuvent être explorés pour la communication et l'interaction dans les architectures logicielles.

Pour toutes ces raisons, la principale contribution que nous suggérons dans le cadre de cette thèse est l'intégration du concept de processus logiciel dans le connecteur afin de modéliser sa structure et son comportement. Nous proposons une nouvelle structure qui accorde au connecteur un statut égal à celui du composant avec des ports qui exportent des services dans son interface. Une telle interface est formée de ports ou points d'interaction utilisés en entrée/sortie par ses services. Le comportement est considéré comme le service de communication exposé par l'interface. Les services listés dans l'interface donnent l'identité et le rôle du connecteur. Le connecteur dans son ensemble est décrit avec les éléments et la sémantique du processus logiciel. Nous baptisons notre proposition : (CaP) pour '*Connector as Process*'.

De manière plus pratique, le processus logiciel est un ensemble de trois éléments liés : « Unité de travail », « Produit de travail » et « Rôle ». C'est une séquence d'activités ou « unité de travail », chaque activité consomme des produits « input work product » pour fournir un « output work product ». Le produit est sous la responsabilité d'un « Rôle » qui est en mesure de lancer et de contrôler l'exécution de l' « Activité » qui le produit.

En plus des notions de communication et d'architectures logicielles, il nous reste d'évoquer la notion d' « abstraction ». L'étude et la réalisation de ce concept revient à faire un appel trivial au concept de méta-modélisation. Ces concepts sont souvent vus comme des synonymes. Abstraire ou méta-modéliser revient à se décharger des détails en passant à un niveau supérieur plus fin et plus simple. À partir de ce niveau, on peut facilement définir la sémantique et les langages des niveaux inférieurs.

Pour décrire la sémantique et la structure du connecteur proposé, nous présentons donc un méta-modèle UML qui définit les différents éléments structurels et comportementaux et leurs relations. Ces relations représentent les différentes dérivations et spécialisations ainsi que l'intégration de la structure relative aux processus logiciels. Afin de doter ce modèle d'un pouvoir plus expressif, il à été augmenté par des stéréotypes du profile SPEM pour lui donner une dimension plus pratique. Aussi, ces stéréotypes représentent une forme d'extension de notre proposition pour un domaine particulier et une démarche à suivre pour représenter un nouveau paradigme.

Deux exemples assez distincts sont utilisés pour montrer la puissance du modèle en termes de généricité à travers un mécanisme de conformité visuelle. Ces deux exemples issus de deux communautés différentes et très distantes sémantiquement. Avec les mêmes moyens, le même connecteur est capable de décrire l'interaction du modèle « Pipe&Filer » connu dans la communauté architecture logicielle et le modèle « Cycle de vie en V » connu dans la communauté processus logiciel.

Pour pouvoir prendre en charge les architectures logicielles à base de connecteur orienté processus « CaP », nous avons également proposé un langage de description d'architecture (ADL) pour cette fin. Nous avons baptisé cet ADL « capADL » capable de supporter la charge de sémantique du connecteur « CaP ». Nous avons pu modéliser les mêmes exemples avec le même outil (capADL).

Etant donné que la génération de code est souvent une fonction usuelle de tout ADL, nous avons doté capADl de cette faculté. Nous avons pu modéliser et exécuter une transformation de modèles de du type « modèle vers texte » qui génère le code Java à partir de l'architecture de l'application. Nous avons proposé une grammaire de graphe de 32 règles

pour cette fin. L'ensemble de la partie pratique est réalisé avec l'outil AToM3.

Problématique et motivation

Dans des environnements distribués et hétérogènes, la communication est une problème fondamentale qui peut conduire à un ensemble de risques importants. Ces risques et préoccupations doivent être pris en compte dès les premières étapes du processus de développement de systèmes complexes.

Les systèmes actuels tendent à devenir fortement interactifs. La voiture en est le parfait exemple. En plus de sa fonction (avancer, tourner, reculer), de sa structure (carrosserie, roues, moteur, transmission) et de son comportement (rotation moteur, embrayage, changement de vitesse, rotation roues), la voiture moderne est extrêmement communicante et fortement connectée avec son environnement.

Par conséquent, en plus des axes usuels du développement des systèmes informatiques (fonction, structure et comportement), nous avons l'ambition d'aborder la proposition d'un nouvel axe celui de la communication. Cet axe doit être désormais pris en compte dans les phases précoces du processus de développement au même titre que les autres axes.

Dans ces conditions, la communication devient un processus complexe et non plus des liaisons point à point. Ainsi, la communication devient une source principale d'hétérogénéité et un réel défi d'interopérabilité internes et externes. Prendre en charge le problème d'interopérabilité à un stade précoce dans le cycle ou le processus de développement peut être un atout majeur. Généralement, face au problème d'interopérabilité, on a toujours apporté des solutions ad-hoc et technologique (Corba CCM, framework .Net) qui restent non réutilisables et non généralisables. Les solutions conceptuelles, réutilisables par définition, en revanche sont limitées voire inexistantes en dehors de certains contextes dédiés.

Le principal objet de cette thèse est la proposition d'un modèle conceptuel pour l'abstraction de la communication dans les architectures logicielles. Il s'agit de réfléchir, d'explorer et de proposer un modèle qui soit générique assez pour être en mesure d'absorber la disparité syntaxique et sémantique entre les paradigmes ainsi que de favoriser l'interopérabilité.

Loin de parvenir encore à cette fin en termes de $4^{ème}$ axe de modélisation, la problématique discutée dans le cadre de cette thèse repose sur les constats suivants :

- Lors de développent des systèmes à base de composants, la partie communication est généralement déléguée à un second rang ou classe. Dans la pratique, l'accent est souvent mis sur la partie calcul plus que la partie interaction.
- Dans des environnements fortement communicants, les applications récentes sont de plus en plus communicantes pour atteindre des proportions souvent équivalentes ou des statuts égaux entre la partie interaction et la partie fonctionnelle
- La disparité des paradigmes et la croissance constante de la complexité font que des travaux de recherche se sont consacrés à étudier et passer en revue ces paradigmes dans leur ensemble où l'on retrouve à chaque fois la communication comme un critère de comparaison parmi d'autres. En revanche, aucune étude n'a été consacrée à la communication proprement dite comme une entité de première classe qui aura l'ambition de traverser les paradigmes de développement.

- Le monde actuel et futur évoluent sans cesse vers une forte interactivité (« cloud » et « pervasive computing », internet des objets, ..) (Garlan, 2014). Les différents équipements et espaces interactifs émanant de plusieurs constructeurs, dans différents paradigmes et avec différentes technologies posent de réels défis d'interopérabilité où la communication reste le facteur essentiel pour de tels aléas.
- Dans de pareilles conditions, une interaction doit être vue comme un processus complexe et non plus une liaison point à point.
- Dans l'équipe d'architecture logicielle du Pr Mourad Oussalah du laboratoire LS2N (ex LINA), beaucoup de travaux et thèses ont abordé plusieurs aspects des architectures logicielles sous différents angles. On a donc voulu également consacrer une thèse à l'étude exclusive de l'aspect communication dans les architectures logicielles.

Ainsi, pour apporter des réponses aux soucis susmentionnés, nous pensons qu'un modèle conceptuel qui soit capables d'absorber toutes les disparités syntaxiques, hétérogénéités, complexités, qui favorise l'interopérabilité et qui soit générique assez pour traverser les paradigmes de développement est fortement recommandé. Ce modèle doit aussi être préservé en tant qu'entité réutilisable, autonome et de première classe tout au long du processus de développement.

Contributions

Comme contributions majeures, il est question d'apporter des réponses et des suggestions quant aux problèmes posés précédemment en:

- précisant que la principale contribution de cette thèse est l'abstraction de la communication dans les architectures logicielles par la notion de connecteur vu comme un processus logiciel. Ceci étant fait en utilisant les techniques de méta-modélisation particulièrement.
- proposant un connecteur qui considère la communication comme un processus complexe. Il est vu comme un composant de communication réutilisable au même titre qu'un composant sur étagères de calcul. Le connecteur préserve ce statut de la phase de conception à la phase de déploiement lors son développement.
- étudiant de manière détaillée l'état du domaine avec toute sa disparité. Ainsi, nous avons étudié les architectures et les connecteurs logiciels, les processus logiciels ainsi que la méta-modélisation et l'ingénierie des modèles. Des comparatifs et synthèses ont été à chaque fois élaborés. Cette étude est faite de sorte que toute notion évoquée dans les chapitres (4 et 5) soit évoquée et définie au préalable dans cette partie.
- proposant un nouveau connecteur orienté processus (CaP) dans les architectures à base de composants. Une structure, un comportement et une riche sémantique ont été définis à travers les techniques de méta-modélisation.
- proposant les différents stéréotypes du profil SPEM pour lier les concepts de processus logiciel et concept d'architecture logiciel afin de prendre en charge notre connecteur (CaP) assortie d'une phase de validation qui montre comment surmonter les problèmes d'interopérabilité. Une validation visuelle est proposée à travers deux exemples très distincts.

- proposant également un langage de description d'architecture (capADL) pour prendre en charge la nouvelle charge sémantique de notre approche en utilisant l'Outil de modélisation AToM3 et le langage Python pour l'expression des contraintes.
- En utilisant capADL, nous avons pu instancier les modèles architecturaux pour le patron « Pipe&Filter » et le modèle de cycle de vie «V».
- Avec le même outil, nous avons pu faire une transformation automatique de ces modèles pour obtenir le pseudo code Java des deux modèles. Cette transformation de modèles est réalisée par une approche de transformation de graphes. Une grammaire de graphes de 32 règles a été proposée en utilisant l'outil AToM3. La grammaire a été validée sur les mêmes les mêmes cas d'études définis précédemment.

Le choix des deux exemples de validation, soit le « Pipe&Filter » et le « Cycle_V » qui n'ont aucun rapport sémantique entre eux, montre la force de notre contribution. On peut dire que le connecteur (CaP) est assez générique pour modéliser et prendre en charge des domaines ou paradigmes très distants sémantiquement. Dans notre cas présentement, ces domaines sont les architectures logicielles à base de composants et les processus logiciels. Le reste des paradigmes est certainement compris entre ces deux bornes.

Plan de la thèse

Suite à cette introduction, le contenu de la thèse est organisé globalement en deux grandes parties, l'état de l'art du domaine (trois chapitres) et l'approche proposée (deux chapitres). Toutes les notions développées dans les chapitres (4 et 5) sont détaillées dans les chapitres (1, 2 et 3). La thèse est organisée comme suit:

- **Le chapitre 1**: consacré aux architectures logicielles. Il concerne tous les éléments inhérents à cette discipline comme première partie. La seconde partie est réservée exclusivement aux connecteurs qui sont vus comme des instances de première classe ainsi qu'aux langages de descriptions d'architectures qui les prennent en charge. Les sections importantes de ce chapitre sont concernées par une synthèse et un comparatif. Ce chapitre décrit les assises théoriques, conceptuelles et pratiques nécessaires à notre proposition.
- **Le chapitre 2**: consacré au concept de processus logiciel avec tous les éléments associés. Aussi, un intérêt particulier est accordé au méta-modèle SPEM qui représente un élément central de notre approche. Aussi, d'autres langages dédiés à la prise en charge des processus logiciels pour des buts comparatifs. Ce chapitre décrit les concepts qui forment notre principale contribution ou réponse à la problématique discutée dans cette thèse.
- **Le chapitre 3**: concerne les principaux aspects liés aux concepts souvent inséparables de méta-modélisation et d'ingénierie de modèles assortis de quelques comparatifs également. Les transformations de graphes sont aussi sommairement discutées. Ces deux concepts représentent les moyens et les outils utilisés pour la mise en œuvre de nos propositions.
- **Le chapitre 4**: concerne l'approche proposée en présentant le modèle conceptuel du connecteur orienté processus (CaP) et de ses différents méta-

modèles. Une partie est réservée aux différents stéréotypes SPEM qui permettent de fusionner les concepts d'architectures logicielles et processus logiciels. Cette migration est validée sur des modèles ou exemples de deux communautés différentes : architecture logicielle « Pipe&Filter » et processus logiciel « Cycle en V ».

- **Le chapitre 5**: représente la réalisation en utilisant l'outil multi-paradigme (MPM) de méta modélisation et de transformation de modèles AToM3. Elles sont décrites les différentes étapes pour définir et produire un langage de description d'architecture (capADL) permettant de supporter les architectures logicielles à base de connecteur orienté processus (CaP). Une deuxième partie est consacrée à la génération de code de l'application à partir de son architecture orientée processus à partir du même ADL. Pour cette transformation de modèles, une grammaire de réécriture de graphes est discutée dans ce même chapitre. Une étude de cas avec les mêmes exemples (Pipe&Fileter et Cycle en V) est présentée.
- Enfin, nous terminons avec une conclusion et une synthèse suivies de propositions et ambitions futures.

CHAPITRE

1 Architecture Logicielle et Connecteur Software

1.1 Introduction

Le petit robert (Rey, 2018) définit l'Architecture comme « l'art de construire des édifices ». Il en ressort que l'architecture est d'abord et avant tout un art. Le terme architecture a toujours été lié au domaine de génie civil: on pense à l'architecture d'un monument, une maison ou encore d'une route, tunnel et pont que l'on appel communément ouvrages d'art. L'architecture à toujours été l'élément visible et éternel qui traverse le temps et les ères comme un témoin de passage et de liaison entre les civilisations humaines.

On n'imagine pas se lancer dans la construction d'un bâtiment sans avoir prévu son apparence, ses fonctionnalités, étudié ses fondations et son équilibre, choisir les matériaux de construction, etc. Dans le cas contraire, on va au devant de graves désillusions.

Au même titre que le génie civil, on a associé également la notion d'architecture au domaine de génie logiciel qui, à sa naissance, était plutôt une sensation plus qu'un fait visible. II parait que les domaines d'ingénierie aussi lointains soient-ils partagent au moins ce même concept : l'architecture.

Par analogie donc à l'architecture bâtiment, on a vu naître l'architecture logicielle. C'est une discipline récente du génie logiciel qui s'intéresse à la structure, les comportements ainsi que les propriétés globales d'un système. Elle s'adresse plus particulièrement à la conception de systèmes de grande taille et ce à un stade précoce de leurs cycles de vie.

Durant de nombreuses années, l'architecture logicielle était décrite en termes de « *boîtes-et-lignes* ». Ce n'est qu'au début des années 90 que les concepteurs de logiciels se sont rendu compte du rôle déterminant que joue l'architecture logicielle dans la réussite d'un projet de développement, de la maintenance et de l'évolution de leurs systèmes logiciels. Une bonne conception d'une architecture logicielle doit donner un produit performant qui répond aux besoins initiaux des clients et de leurs évolutions dans le temps. Une architecture inappropriée peut avoir des conséquences désastreuses allant jusqu'à l'arrêt du projet (Garlan, 2000).

Pour des raisons de granularité et de réutilisabilité, les architectures logicielles ont toujours été synonymes de développement orienté composant ou « *architecture logicielle à base de composants* » (Bass et al., 2003; M Shaw & Garlan, 1993; Mary Shaw & Garlan, 1996; Taylor et al., 2009). Or, dans la pratique, on rencontre une panoplie d'architectures logicielles associées aux différents paradigmes de développement. La plus répandue est celle lié à l'objet, où l'on parle « *architecture logicielle à base d'objets* » (Booch et al., 2008;

Jacobson, 1993) ainsi que dans le développement orienté service (Chu, 2005; Koubaa, 2019) comme dans les systèmes à base d'agents (Mustapha & Frayret, 2016; Vu et al., 2020). Aussi, on retrouve celles liées aux domaines récents tels que le cloud computing (Shroff, 2010), l'internet des objets (Muccini et al., 2019), les micro-services (Bakshi, 2017) et les systèmes cyber-physiques (Ahmed et al., 2013) ...etc.

Dans ce chapitre, ils seront évoqués les aspects liés aux architectures logicielles, allant des définitions et certains éléments historiques jusqu'aux langages de descriptions d'architectures. Il sera dressé également un petit comparatif entre différentes architectures logicielles à travers quelques paradigmes de développement. Aussi, il sera question des éléments architecturaux dans les systèmes à base de composants, les propriétés et les motivations ainsi que les styles et les différents modes d'évolution d'une architecture. La deuxième partie de ce chapitre sera entièrement dédiée à la notion de connecteur qui représente l'élément d'interaction entre composants. Ce chapitre se termine par la discussion et la comparaison d'un certain nombre de langages de description d'architecture (*Architecture Description Language* ADLs).

Tout au long de ce chapitre, les différents concepts évoqués seront à chaque fois discutés par analogie avec les architectures en génie civil afin de rapprocher et simplifier la compréhension.

1.2 Historique et définitions

Comme évoqué précédemment, les architectures logicielles étaient à l'origine une sensation. On cite souvent Ian Sharp (Randell & Buxton, 1970) pour ses commentaires faits en 1969, en relation avec les travaux précédents de Dijkstra lors de la conférence de l'OTAN de 1968 (*Software Engineering Techniques*) (Naur, 1968):

Sharp: "I think that we have something in addition to software engineering: something that we have talked about in small ways but which should be brought out into the open and have attention focused on it. This is the subject of software architecture. Architecture is different from engineering". C'est sûr que le plan d'une maison (*architecture*) est différent en nature de la maison elle-même (*Ingénierie*).

Les concepts de bases des architectures logicielles, tels qu'ils sont connus aujourd'hui, remontent aux débuts de la discipline du génie logiciel. Les travaux pionniers de Dijkstra en 1968 sur la matière ont proposé une structuration impérative d'un système avant même de se lancer à écrire des lignes de code (Dijkstra, 1968). Aussi, ils mettent en évidence le besoin de la notion de « *niveau d'abstraction* » dans la conception de systèmes à grande taille. Bien que Dijkstra n'utilise pas le terme « *architecture* » dans l'ensemble de ses travaux, les résultats de ses travaux par contre représentent une base pour la définition des architectures logicielles contemporaines. Pour en faire référence, on les appel tout de même « *Dijkstra School of Architecture*» (Randell & Buxton, 1970).

Au début des années 70, les travaux de David Parnas ont introduit le concept de « *modularité* » qui propose d'améliorer la souplesse et le contrôle conceptuel du logiciel en réduisant le temps de développement des systèmes. Il a montré l'importance d'une bonne structuration des systèmes dans les étapes de conception dont il propose certaines idées afin d'atteindre un niveau de structuration adéquat (Parnas, 1972).

C'est incontestablement dans les années 90 que les architectures logicielles commencent à avoir un essor important. Les idées de base, telles que donner de l'importance aux décisions prises dans les premières étapes du développement du système ainsi qu'à la définition correcte de l'architecture logicielle, ont joué un rôle déterminant dans l'évolution de la discipline.

A ce jour, il ya toujours une variété de définitions qui représentent des interprétations plutôt que des normes à suivre pour les architectures logicielles. Dans la section qui suit nous présentons ces interprétations et définitions par ordre chronologique.

Après avoir examiné les architectures dans d'autres disciplines (bâtiments, réseaux, mécanique), Perry et Wolf décrivent l'architecture logicielle comme étant un modèle constitué de trois composants : les éléments, la forme et le raisonnement. Cette définition est restée solide jusqu'à ce jour, le reste des définitions ne sont que des variations de celle-ci.

1.2.1 Perry et Wolf (1992)

L'une des premières définitions formelles d'architectures logicielles, Perry et Wolf (1992), est restée l'une des plus perspicaces. Les auteurs décrivent l'architecture logicielle comme étant un modèle constitué de trois composants : *les éléments, la forme et le raisonnement*.

Les *éléments* capturent les blocs de construction du système, qui peuvent être de trois types : éléments de traitement, éléments de données et éléments de connexion.

La *forme* capture la manière dont les éléments (architecturaux) sont organisés dans l'architecture, au moyen de propriétés et de relations pondérées. C'est-à-dire que la forme capture la composition des éléments (la configuration), les caractéristiques de leurs interactions et leur relation avec leur environnement d'exploitation ainsi que les contraintes associées

Le *raisonnement* capture les motivations pour le choix d'un style architectural, le choix des éléments et la forme. C'est-à-dire l'intention du concepteur du système, les hypothèses, les choix, les contraintes externes, les modèles de conception sélectionnés et d'autres informations qui ne sont pas facilement observables à partir de l'architecture.

Perry et Wolf identifient trois éléments de construction architectural : (1) les éléments de traitements (2) les éléments de stockage (3) les éléments de connexion. Ces éléments vont former plus tard les éléments architecturaux de base.

1.2.2 Garlan et Shaw (1993)

En 1996, Shaw et Garlan (1996) ont soutenu que l'architecture logicielle représentait un problème de conception et de spécification de la structure globale du système qui dépassait les algorithmes et les structures de données du calcul. Ils proposent une définition plus restrictive que celle de Perry et Wolf pour donner une définition populaire à nos jours :

« *un ensemble de composants de calcul – ou tout simplement de composants – accompagné d'une description des interactions entre ces composants – les connecteurs* ».

Sur la base de cette définition, les auteurs ont utilisé le terme de style architectural pour désigner une famille de systèmes (c'est-à-dire d'architectures applicatives) qui partagent un vocabulaire commun de composants et de connecteurs, et qui répondent à un ensemble de contraintes pour ce style. Les contraintes peuvent porter sur une variété de choses, notamment sur la topologie des connecteurs et des composants, ou sur la sémantique de leur exécution.

1.2.3 Garlan et Perry (1995) / Kruchten (1995)

Parmi les définitions largement admises et acceptées, on retrouve celle proposée par Garlan et Perry pour leur éditorial invité dans l'IEEE Transactions on Software Engineering d'avril 1995 (Garlan & Perry, 1995), consacré à l'architecture logicielle où l'on retrouve un début de standardisation :

« L'architecture logicielle est la structure des composants d'un programme/système, leurs interrelations et les principes et directives régissant leur conception et leur évolution dans le temps »

Dans la même année, une autre définition plus étendue et en même temps plus précise a été proposée par Phillipe Kruchten (1995):

« L'architecture logicielle traite de la conception et de la mise en œuvre de la structure de haut niveau du logiciel. C'est le résultat de l'assemblage d'un certain nombre d'éléments architecturaux sous des formes bien choisies pour satisfaire les principales exigences de fonctionnalité et de performance du système, ainsi que d'autres exigences non fonctionnelles telles que la fiabilité, l'évolutivité, la portabilité et la disponibilité. .

L'architecture logicielle traite de ***l'abstraction****, de la* ***décomposition*** *et de la* ***composition****, du* ***style*** *et de* ***l'esthétique*** *».*

1.2.4 Bass, Clements et Kazman (1998)

Bass, Clements et Kazman (1998), ont proposé une définition de l'architecture logicielle par laquelle ils ont insisté sur les propriétés visibles de l'extérieur des éléments définissant leur comportement attendu. Ces propriétés traduisent les hypothèses que d'autres composants peuvent faire sur ce composant (par exemple, les services qu'il fournit, les ressources requises, ses performances, ses mécanismes de synchronisation).

« *L'architecture logicielle d'un programme ou d'un système informatique est la ou les structures du système qui comprennent des éléments logiciels, les propriétés visibles de l'extérieur de ces éléments et les relations qui les relient*».

Un composant est donc une unité d'abstraction dont la nature dépend de la structure considérée dans le processus de conception architecturale. Il peut représenter un service, un module, une bibliothèque, un processus, une procédure, un objet, une application, etc. Mais le comportement observable de chaque composant fait partie de l'architecture puisqu'il détermine ses interactions possibles avec d'autres composants.

1.2.5 Taylor, Medvidovic, and Dashofy (2009)

L'une des formes d'extension les plus intéressantes de Perry et Wolf est fournie par Taylor, Medvidovic et Dashofy (2009). Elle embrasse le sens des questions « Quoi », « Comment » et « Pourquoi » :

Les *éléments* aident à répondre aux questions « Quels » sur l'architecture : Quels sont les éléments d'un système ? Quel est leur objectif principal et les services qu'ils fournissent?

La *forme* permet de répondre aux questions « Comment » sur l'architecture : Comment l'architecture est-elle organisée ? Comment les éléments sont-ils composés pour accomplir la tâche clé du système ? Comment sont répartis les éléments ?

Le *raisonnement* aide à répondre aux questions « Pourquoi » sur l'architecture : Pourquoi des éléments particuliers sont-ils utilisés ? Pourquoi sont-ils combinés d'une manière particulière? Pourquoi le système est-il distribué d'une manière donnée ?

La même année, une autre définition est donnée par les mêmes auteurs. Ils déclarent que l'architecture du système représente l'ensemble des principales décisions de conception qui dépendent des objectifs du système définis par les parties intervenantes (du genre les exigences pilotent l'architecture). La définition place l'architecture logicielle comme un artefact central dans le cycle de vie du logiciel qui est spécifié dans les premières étapes du développement logiciel et qui constitue le modèle de processus.

« L'architecture d'un système logiciel est l'ensemble des principales décisions de conception concernant le système. Les décisions de conception englobent tous les aspects du système en cours de développement, y compris : la structure du système, le comportement fonctionnel, l'interaction, les propriétés non fonctionnelles et la mise en œuvre. ».

En guise de synthèse, il n'y a aucun conflit entre ces définitions. Elles ne se diffèrent que par la vue qu'elles ont du système. Elles sont similaires en ce sens qu'elles sont toutes concernées par la structure et le comportement du système :

- La « structure » décrit comment le système est constitué d'éléments interconnectés appelés composants.
- Le « comportement » fait référence à l'interaction visible de ces composants afin d'obtenir la fonctionnalité globale du système.
- La structure et le comportement de l'architecture logicielle sont formellement décrits à l'aide des langages de description d'architecture (ADL).

1.2.6 La norme ANSI/IEEE Std 1471 (2000)

L'IEEE a défini une norme pour la description architecturale des systèmes logiciels, l'IEEE 1471 (2000). Cette norme a été mise à jour en 2008 et adoptée comme standard ISO/IEC en 2007. Elle a sciemment proposé une définition généraliste capable d'englober plusieurs interprétations. La norme IEEE 1471 établit un cadre conceptuel (Figure 1.1) et un vocabulaire pour désigner des problématiques architecturales des systèmes. Elle comprend l'utilisation de plusieurs vues, de modèles réutilisables au sein des vues et la relation qu'entretient l'architecture avec le contexte du système (appelé aussi environnement). En se basant sur ce cadre conceptuel, l'idée est d'identifier et d'édicter des pratiques architecturales pertinentes, et de les incuber.

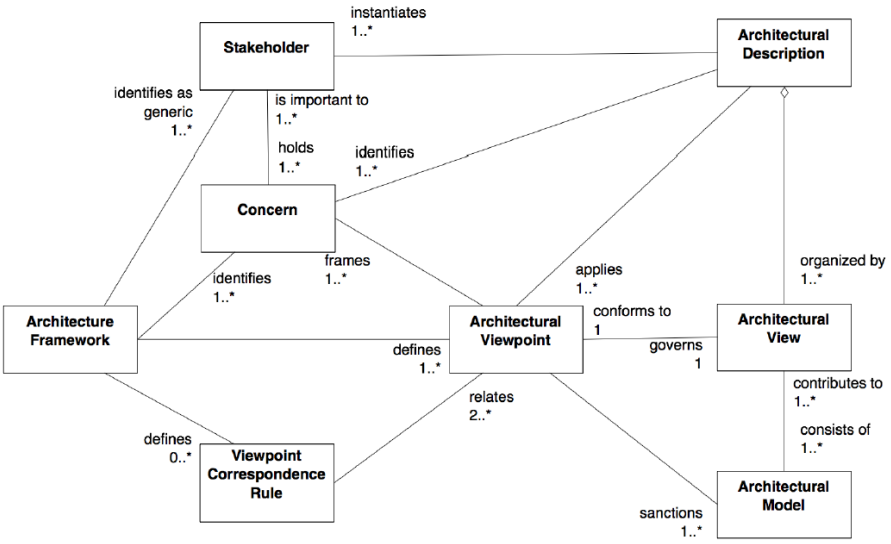


Figure 1.1- Modèle conceptuel architecture logicielle (norme 1471-2008).

La Figure 1.1 montre qu'une description architecturale est organisée en un ensemble de vues architecturales. Les vues sont définies à travers des points de vue. La description architecturale identifie plusieurs préoccupations définies par différents intervenants. Le modèle architectural consiste en un ou plusieurs vues architecturales. Les points de vue architecturaux sont régis par des règles de correspondance pour définir un cadre architectural identifié par les intervenants et leurs différentes préoccupations.

1.3 Généralités

Au-delà des définitions, nous allons aborder dans cette section les concepts généraux qui entourent les architectures logicielles. Nous allons discuter de l'historique de cette appellation ainsi que son rapport avec l'architecture bâtiment. Comme nous allons séparer une nomenclature de domaine qui, à première vue, semble identique. Enfin, nous évoquons quelques motivations pour l'utilisation des architectures logicielles.

1.3.1 Architecture Bâtiment

Les pionniers du domaine des architectures logicielles (Clement et al., 2011; Taylor et al., 2009) associent souvent ce nouveau concept à l'architecture dans le domaine de la construction du génie civil. La ressemblance et la similitude entre les deux domaines sont souvent mises en avant bien que, dans leurs natures, les deux produits architecturaux n'ont rien de commun. Sémantiquement, un logiciel n'a rien de similaire avec un bâtiment ou un château. Le logiciel possède, en outre, cette particularité d'avoir un caractère inusable et intangible ce qui n'est pas le cas pour un immeuble. Il paraît que cette ressemblance vient particulièrement des définitions des deux concepts et leurs aspects fonctionnels.

En plus de quelques outils primitifs, l'architecture dans le domaine du génie civil peut être considérée comme le plus ancien métier technique humain. Ce métier a commencé par le creusement de grottes pour s'abriter du chaud et du froid avant d'aboutir au gratte-ciel et à la ville intelligente. Les vestiges architecturaux peuvent être le seul témoin du passage des civilisations à travers l'histoire humaine dans ses différents espaces. Si notre civilisation actuelle peut compter sur l'écrit ou mieux encore sur la vidéo pour raconter notre histoire aux générations futures, les civilisations anciennes quand elles n'ont eu que les œuvres architecturales pour nous informer de leurs vécus. Il en ressort que l'architecture est le seul produit humain qui a réellement su traverser le temps.

Loin des pyramides égyptiennes, de l'aqueduc romain, des temples grecs et même des merveilles gravées dans la pierre de Petra, ce qui nous intéresse dans notre contexte sont les similarités entre le bâtiment et le logiciel qui doivent être faites encore loin des considérations artistiques. Paul Clement et al. (2011) racontent que ce mariage remonte à la naissance des architectures logicielles avec l'apparition des concepts «style architectural»[1] et «patron architectural»[2].

Historiquement, Mary Shaw (1990) remarquait et décrivait des concepts d'architecture récurrents qu'elle trouvait dans de nombreux systèmes. Elle va les appeler « éléments d'un langage de conception pour l'architecture logicielle » ou « idiomes de conception ». En 1992, Dewayne Perry et Alexander Wolf (1992) ont voulu construire une intuition sur le domaine encore nouveau de l'architecture logicielle. En examinant d'autres types d'architecture – l'architecture de réseau, l'architecture de l'ordinateur, etc. – ils ont trouvé l'architecture du bâtiment riche en concepts fertiles et empruntables. L'un de ces concepts était le style d'architecture. De même pour le patron architectural mis en exergue par Frank Buschmann et ses collègues (1996) qui sont bâtis autour des patrons de conception (Gamma, 1995).

[1] Style architectural : spécialisation des types d'éléments et de relations, ainsi qu'un ensemble de contraintes sur la façon dont ils peuvent être utilisés. Un style d'architecture se concentre sur l'approche de l'architecture, avec des indications plus légères sur le moment où un style particulier peut ou non être utile

[2] Patron architectural: exprime un schéma d'organisation structurelle fondamentale pour les systèmes logiciels. Le patron de concept focus sur le problème et le contexte et sur la solution

(a)

(b)

Figure 1.2- Architecture bâtiment.

Dans les architectures logicielles, les patrons et les styles architecturaux doivent leur sens à l'architecte du bâtiment Christopher Alexander qui dans les années 1970 a écrit plusieurs livres détaillant les approches architecturales pour résoudre les problèmes courants de conception de bâtiments. « Les gens aiment s'asseoir à côté des fenêtres », a-t-il écrit, « alors faites en sorte que chaque pièce ait un endroit où ils puissent le faire confortablement ». « Les gens adorent les balcons », a-t-il écrit, « mais les observations montrent qu'ils ne passeront pas de temps sur un balcon de moins de 10 pieds de large. Faites donc vos balcons au moins 10 pieds de large ». « Les gens aiment les espaces extérieurs », a-t-il écrit, « mais pas s'ils sont à l'ombre d'un bâtiment. Donc, dans l'hémisphère nord, placez vos cours du côté sud ». Il a appelé ces modèles de pépites de conception : « une règle en trois parties, qui exprime une relation entre un certain contexte, un problème et une solution » (Alexander 1979, p. 247). La communauté des patrons (de toute saveur) a essayé de rester fidèle à sa signification (Clement et al., 2011).

Indépendamment de la différence entre les patrons et les styles architecturaux, en termes de détails et d'abstraction, les deux notions impliquent des décisions architecturales prises au préalable pour la construction d'une maison. Cette définition en particulier (décision architecturale) est la seule qui fait consensus dans la communauté des architectures logicielles. Par le pouvoir de l'analogie, le concept de « style architectural » comme le « Pipe&Filter » ou « Client/Serveur » dans les architectures logicielles ont le même aspect dans les architectures du bâtiment comme les styles « Gothique, Baroque, Rococo, … ». Ces styles sont tous reconnus et distingués du premier abord visuel par leurs aspects structurels que fonctionnels.

La Figure 1.2 montre l'architecture d'une maison dans le style moderne. Il est utile de constater que pour le grand public c'est la Figure 1.2 (a) qui représente l'architecture. Pour un vrai architecte ou pour un bon informaticien, c'est la Figure 1.2 (b) qui représente l'architecture de la maison et qui permet de raisonner sur elle. La Figure 1.2 (a) n'est que

l'exécution de cette architecture d'une maison habitable ou d'un système informatique en exécution dans un environnement d'instance. Toutes les questions, les solutions, les décisions sont présentes dans l'architecture ou le « plan » de la maison avant même de la construire. Le plan reste le principal support de raisonnement. Ainsi, une erreur dans les fondations est difficilement récupérable voire impossible quand on est sur le toit.

En conclusion, de nombreux systèmes logiciels contemporains ont des conceptions qui s'inspirent fortement des conceptions d'applications antérieures. D'autres s'inspirent des systèmes sociaux ou de la nature. Tout comme l'architecte bâtiment qui reste entourée de structures, de livres et de plans passés. L'architecte logiciel qualifié dispose de systèmes existants et, surtout, d'un vaste éventail de styles, de techniques, de processus et d'outils qui représentent une palette de choix lorsque nous sommes confrontés à un défi de conception (Taylor et al., 2009).

1.3.2 Architecture vs Modélisation vs Conception

Ces concepts sont assez proches dans l'usage courant du monde de l'ingénierie des logiciels et de leurs cycles de vie. De premier abord, il semblerait déjà que le mot « Architecture » paraît désigner un produit alors que « Modélisation » et « conception » semblent être des opérations. Théoriquement, l'action relative au produit « architecture » devrait être « Architecturisation » ; ce mot est inexistant dans le dictionnaire. Le produit associé à l'opération « Modélisation » est certainement « Modèle » alors que le produit pour « Conception » est littéralement inexistant et ne doit en aucun cas être « Concept ».

Normalement, le mot anglais « Design » n'a pas d'équivalent direct en français. Par abus de langage ; les traducteurs donnent souvent le mot « Conception ». Ceci dit, nous assumons « Conception » comme traduction de « Design ».

Taylor et al. (2009) précisent que l'architecture est par définition vue comme des décisions conceptuelles, elle n'est pas une phase de développement mais c'est le produit de la phase de design ou de conception. Paul Clement et al. (2011) sèment le doute encore sur la question ; ils considèrent l'architecture comme une phase du cycle de vie et diffère de la conception par le niveau de détails. Ils précisent que l'architecture est une conception, mais tout design n'est pas architectural. L'architecte trace la frontière entre la conception architecturale et non-architecturale en prenant les décisions (Van Heesch et al., 2012) qui doivent être liées pour que le système atteigne ses objectifs de développement, de comportement et de qualité. Toutes les autres décisions peuvent être laissées aux concepteurs et aux codeurs en aval. Les décisions sont architecturales ou non, selon le contexte.

À partir de nos différentes lectures sur le sujet, nous traduisons notre façon de voir la différence entre les différents concepts dans la Figure 1.3. La modélisation est très générale par rapport aux autres concepts, c'est l'opération de produire des modèles. Elle est valable et applicable sur toutes les phases du cycle de vie pour produire des modèles associés à chacune d'elle. La conception est une phase parmi d'autres où l'on produit un ensemble de modèles auxquels on ajoute quelques décisions conceptuelles de haut niveau pour obtenir une architecture.

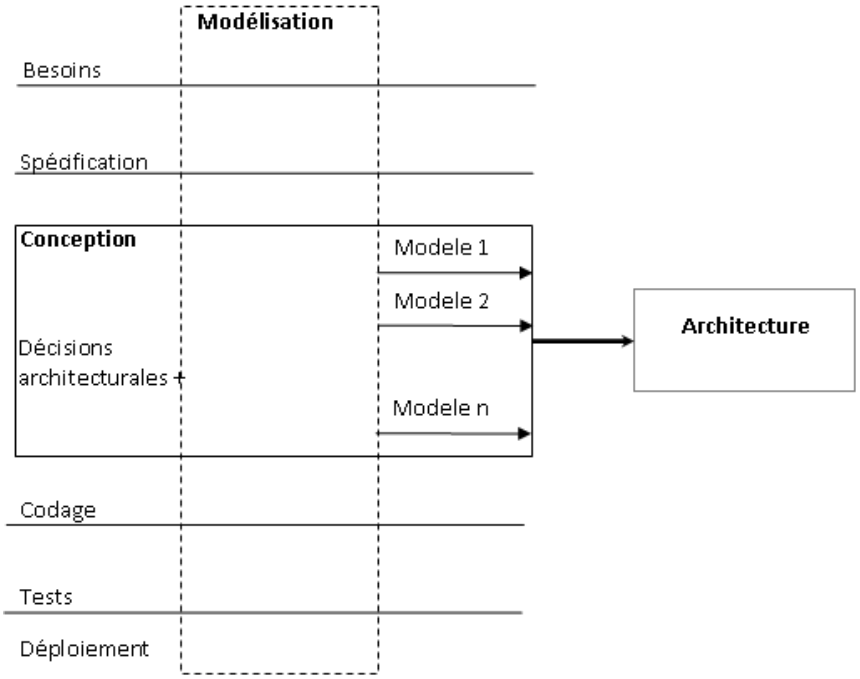


Figure 1.3- Conception vs Architecture.

La Figure 1.3 montre aussi la situation des architectures logicielles et donne un aperçu sur le rôle qu'elles jouent dans le processus de développement. De ce fait, les architectures logicielles jouent généralement un rôle clé en tant que pont entre l'ingénierie des exigences et l'implémentation (Garlan, 2014).

1.3.3 Pourquoi les architectures logicielles

Dans le domaine du génie civil, les objectifs de l'architecture sont que le bâtiment construit réponde aux besoins qu'il doit remplir, qu'il soit robuste dans le temps et plus subjectivement agréable à voir.

Dès lors que le logiciel a obtenu le statut de produit industriel (Brian Randell, 2018), il a eu son propre cycle de vie à l'instar de tout autre produit dans d'autres domaines d'ingénierie. Par conséquent, on a vu naître le concept d'architecture logicielle au même titre que l'architecture bâtiment (Simonnot, 2016) ou une architecture d'un ordinateur de John von Neumann (Von Neumann, 1986).

Il est très utile de pouvoir poser des questions sur le logiciel « en chantier » ou dans un stade précoce durant le processus de développement et d'en obtenir les réponses qui doivent être identiques à celles que le système final aurait donné. Plutôt l'anomalie ou l'erreur est découverte moins cher est son coût de correction. Ceci peut être sous le principe '*the sooner the best*' exhibé par les techniques de '*model-checking'* (Baier & Katoen, 2008).

Ceci étant le principal rôle imparti aux architectures logicielles. Toutefois, on peut énumérer un bon nombre d'autres d'avantages et motivations qui plaident pour leur usage systématique dans le cycle de développement des systèmes complexes :

- Maîtriser la complexité du système en obtenant une vue de haut niveau du système qui fait abstraction des détails techniques,
- Compréhension simplifiée pour les grands systèmes en les présentant à un niveau d'abstraction auquel la conception d'un système peut être facilement comprise. Aussi, la description architecturale expose les contraintes de haut niveau sur la conception du système, ainsi que la justification de certains

choix architecturaux spécifiques (Styles).

- Permettre le raisonnement autour de certaines propriétés (fonctionnelles ou non) du système toujours dans un niveau d'abstraction élevé,
- Utiliser la description comme base pour l'évolution du système et son analyse. L'architecture logicielle peut exposer les dimensions selon lesquelles un système est censé évolué en rendant explicite les « murs porteurs » (Garlan, 2008). Les responsables de maintenance peuvent mieux comprendre les ramifications et les incidences des changements et d'en estimer les coûts.
- Analyse : les descriptions architecturales offrent de nouvelles possibilités d'analyse, notamment la vérification de la cohérence du système (Allen & Garlan, 1997), la conformité aux contraintes imposées par un style architectural (Giesecke et al., 2010), la conformité aux attributs de qualité (Mary Shaw & Clements, 2006), l'analyse de dépendance (Stafford et al., 2003) et spécifiques au domaine analysés pour les architectures construites dans des styles spécifiques (Ducasse & Pollet, 2009).
- Une description architecturale sert souvent de moyen de communication entre les intervenants du logiciel en cours de développement. Elle permet aux parties prenantes d'exprimer leurs opinions sur les poids relatifs des caractéristiques et des attributs de qualité lorsque des compromis architecturaux doivent être pris en compte.
- Prévoir l'érosion architecturale à partir des facteurs d'analyse et de d'évolution (L. De Silva & Balasubramaniam, 2012). L'érosion des architectures logicielles est surtout liée à la distorsion entre l'architecture et le code qui lui est associé,
- Séparation des préoccupations du calcul de celle de l'interaction pour obtenir des composants plus fonctionnels et plus fiables en se déchargeant des contraintes liées à l'interaction,
- Séparation de l'architecture de l'implémentation,
- Les représentations hiérarchiques sont sémantiquement plus riches que de simples relations d'héritage. Elles permettent, par exemple, les représentations multi-vues d'une architecture (Garlan, 2000),
- Réduction du coût et amélioration de la qualité du produit logiciel associé en ayant une vue précise de ces éléments à un stade précoce du cycle de vie,
- Conservation et exploitation des connaissances du domaine pour simplifier la composition et l'usage systématique de la réutilisation. En permettant de créer des composants à partir de composants élémentaires ou des systèmes à partir de sous-systèmes en utilisant par exemple des bibliothèques de composants sur étagères ou des Frameworks d'intégration.
- Conception et mise en œuvre directe du paradigme composant (Composant, connecteur, Configuration),
- Disponibilité d'outils académiques et commerciaux pour la description des architectures logicielles (ADLs),
- Adoptée par les standards à partir d'UML 2.0, en ajoutant le diagramme de composants,

- A ce niveau on peut aussi parler d'architecture exécutable à l'instar du modèle exécutable prôné par l'ingénierie des modèles (Schmidt, 2006) en générale et la démarche MDA (OMG-MDA, 2003) en particulier. Elle consiste au raffinement successif par transformation de modèle jusqu'à obtenir le code de l'application tout en préservant la traçabilité associée.

1.4 Architecture dans les paradigmes de développement

Un paradigme de programmation/développement est une manière de penser fondamentale de programmation/modélisation informatique qui traite de la vue et de la façon dont les solutions aux problèmes doivent être formulées dans un langage de programmation ou une approche de modélisation. C'est un style particulier d'élaboration de ces solutions informatiques, en termes d'analyse, de conception et de développement

Les architectures logicielles ne sont pas propres au paradigme de développement orienté composant, mais ont une existence dans tout cycle de développement indépendamment du paradigme préconisé.

Dans ce qui suit ; nous allons discuter de quelques architectures logicielles à travers les paradigmes de développement (Abdelkrim Amirat et al., 2014; Oussalah, 2014). L'accent sera mis sur l'aspect de la communication pour capitaliser la connaissance et proposer un comparatif à la fin dans le but de dégager un choix du paradigme à retenir pour la suite de cette recherche. Il sera discuté des paradigmes relatifs à l'objet, le composant, le service et l'agent.

1.4.1 Architecture logicielle à base d'objets

Le développement orienté objet est un paradigme (Wegner, 1990) de conception et de programmation apparu au début des années 90s. Il consiste en la définition de briques logicielles appelées objets. Un objet représente un concept, une idée ou toute entité du monde physique. Il possède une identité, une structure interne ou un état et un comportement et sait communiquer avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de réaliser les fonctionnalités attendues du système. Les objets représentent des instances d'une collection de classe définies dans un niveau abstrait.

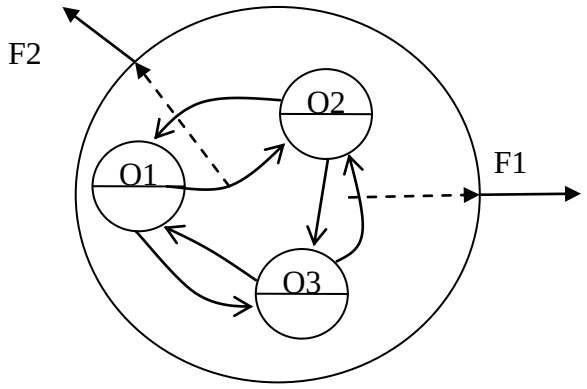


Figure 1.4- Architecture orientée objet

Les fonctions du système F1 et F2 sont obtenues et émergent suite à l'interaction entre les objets O2/O3 ; O1/O2 et O3/O1 comme le montre l'architecture informelle de la Figure 1.4. Dans le paradigme objet la communication est un élément clé sans lequel la fonctionnalité du

système n'a pas lieu sachant que cette dernière est un produit dérivé des interactions entre objets.

La communication dans ce paradigme est essentiellement assurée par l'envoi de messages qui peut être synchrone ou asynchrone. Plusieurs types de messages existent, les plus communs sont : l'envoi d'un signal, l'invocation d'une opération (méthode) et la création ou la destruction d'une instance

Une interruption ou un événement sont des exemples de signaux. Un signal est, par définition, un message asynchrone. Quant aux messages synchrones, l'émetteur reste bloqué dans l'attente de la réponse du récepteur. L'invocation d'une opération est le type de message le plus utilisé en programmation objet :

Il y a aussi l'idée d'adopter le concept du RPC (*Remote Procedure Call*) de la programmation structurée au paradigme objet sous l'appellation RMI (*Remote Method Invocation*) ou l'invocation de méthode à distance qui est une interface de programmation (API) pour Java.

1.4.2 Architecture logicielle à base de composants

Le développement à base de composants (Manickam et al., 2019) est un paradigme apparu dans les débuts des années 1990, en réponse à l'échec de l'approche objet à satisfaire les besoins en réutilisation et en composition. L'approche à base de composants étend le paradigme objet en mettant l'accent sur l'intérêt de la réutilisation, la séparation des préoccupations et la promotion de la composition. La Figure 1.5 montre une architecture informelle qui consiste à l'assemblage et la composition des services offerts (SF) ou fournissant des services requis (SR) pour les composants 1 et 2 en boîte noire.

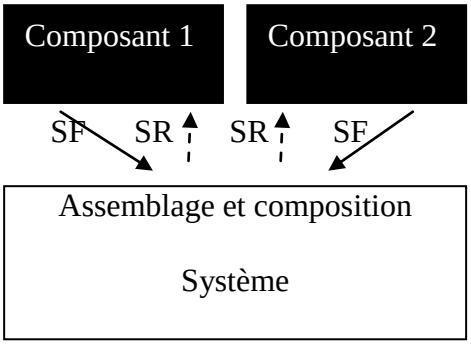


Figure 1.5- Architecture orientée composants.

Lire et comprendre un code existant est toujours une tâche fastidieuse pour les développeurs, mais réutiliser un code existant sous la forme d'un composant s'avère très attractif. En effet, un développeur a juste besoin de savoir ce que fait un composant, et non pas comment il a été implémenté. Aussi, dans l'approche composant, on fait une distinction claire entre le développement d'un composant et celui d'un système. Dans le premier cas, on se concentre sur la construction du composant et dans le second cas, l'effort est porté sur l'assemblage et la composition des composants compatibles. On parle alors de développement pour / par la réutilisation (*For/by reuse*) (Section 1.5.3).

La communication dans le paradigme composant a pris une préoccupation centrale au fil du temps pour favoriser davantage la réutilisation et la composition. Pour cela, elle a pris différentes formes durant son cycle de maturation passant d'un caractère implicite à celui explicite. Deux idées principales ont guidé cette tendance: (i) La première consiste à délocaliser la partie liée à la communication du composant et à ne garder que la partie

fonctionnelle en son sein. (ii) Essayer de conserver cette unité de communication indépendante le long du cycle de vie du système. C'est le rôle imparti au concept de connecteur et d'interface.

1.4.3 Architecture logicielle à base de service

Le développement orienté service (Bouguettaya et al., 2017; Di Francesco, 2017) est un paradigme de développement logiciel (SOSE :*Service Oriented Software Engineering*) directement inspiré des modes d'organisations commerciales réelles entre multinationales. Il se base sur leur notion classique de « service offert ». L'origine du paradigme orienté service vient des demandes liées à des systèmes devant supporter des environnements de plus en plus volatiles et hétérogènes tels que l'Internet et les services Web, les environnements d'intelligence ambiante ou les applications business diffusées sur les réseaux d'entreprises telles que les systèmes ERP (Mahar et al., 2020). La productivité d'un fournisseur et sa réactivité aux changements de besoins représentent des enjeux majeurs dont le SOSE tente d'apporter des solutions dans le développement logiciel.

Le service est une entité logicielle qui représente une fonction bien définie. C'est aussi un bloc de construction autonome qui ne dépend d'aucun contexte ou service externe. Il est divisé en opérations qui constituent autant d'actions spécifiques que le service peut réaliser. Un service correspond donc à un périmètre fonctionnel que l'on souhaite exposer à des consommateurs. Il est sans état et faiblement couplé. Ce paradigme dispose aussi du concept de service composite construit par la composition des descriptions de services. La mise en œuvre de la composition de service se fait durant la phase d'exécution (Oussalah, 2014).

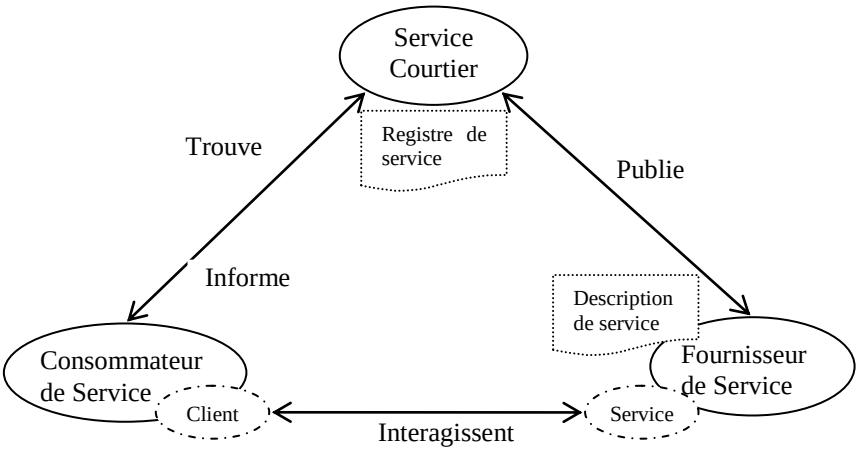


Figure 1.6- Architecture orientée service.

La Figure 1.6 montre une architecture informelle orientée service. Elle considère le service comme une action exécutée (fonctionnalité rendue) par un fournisseur à l'attention d'un client, cependant l'interaction entre le fournisseur et le client est faite par le biais d'un médiateur ou un courtier (qui peut être un bus) responsable de la mise en relation des participants. Les services sont généralement implémentés comme des entités logicielles à forte granularité. Ils englobent et proposent les fonctionnalités des entités des systèmes. Ces systèmes peuvent aussi être définis comme des couches applicatives.

Pour les aspects de communication dans les architectures orientés services ou Web services décrivent, en sélectionnant et composant des services, des logiciels qui interagissent avec d'autres au moyen de protocoles et langages universels (HTTP, XML, …). Ils communiquent également par envoi de messages selon deux formes de protocoles :

- XML – RPC : Protocole RPC (Remote Procedure Call) basé sur XML et

permet donc l'invocation de procédure distante sur Internet.

- SOAP : (Simple Object Access Protocol) est un protocole basé sur XML et qui définit les mécanismes d'échanges d'information entre les clients et les fournisseurs de service Web. Les messages SOAP sont susceptibles d'être transportés en HTTP, SMTP, FTP...

1.4.4 Architecture logicielle à base d'agent

L'approche orientée agent (Dorri et al., 2018; Shehory & Sturm, 2016) est apparue dans les années 70s sous la houlette de l'intelligence artificielle distribuée (IAD). Elle propose le concept *d'Acteur* qui est une entité autonome, interactive et s'exécutant d'une manière concurrente. Au milieu des années 90s, les modèles collectifs SMA (Système multi agents) ont vu le jour. Dans ces modèles, un agent est considéré comme une entité autonome possédant certaines capacités qui l'aident à réaliser ses services ou à utiliser les services d'un autre agent par le biais d'une interaction. Les organisations de systèmes multi-agents sont parmi ces nouveaux modèles.

Les agents sont caractérisés par la capacité sociale de coopérer, coordonner et négocier les uns avec les autres. L'autonomie et les interactions de haut niveau sont les principaux points de différence entre les agents et les approches objets, composants et services. Les agents peuvent être classés en deux catégories :

- les agents *réactifs* attendent l'arrivée d'une action pour répondre aux changements dans leur environnement ;
- les agents *proactifs* prennent l'initiative des décisions dans leur environnement.

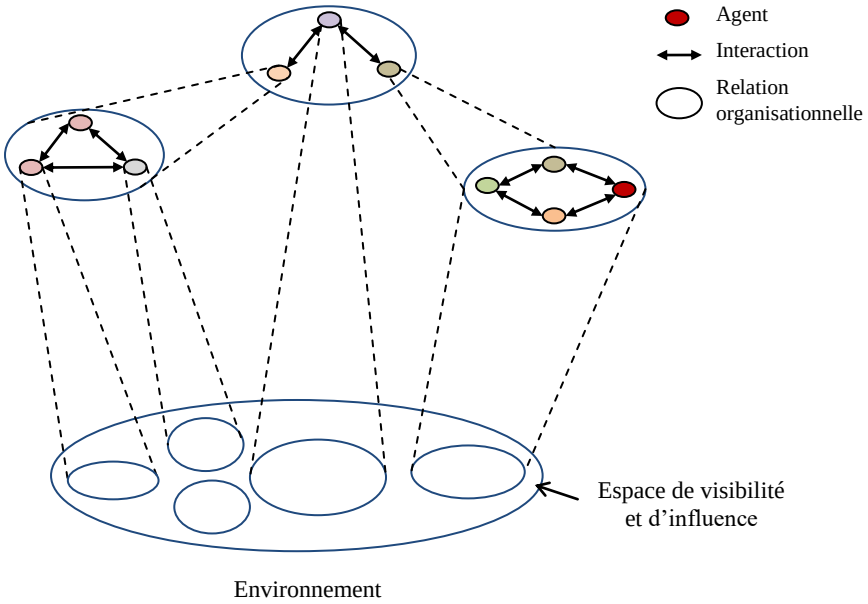


Figure 1.7- Architecture Agent et organisation multi-agent.

Les systèmes multi-agents organisationnels (OMAS) sont des systèmes efficaces pour relever les défis de conception des SMA larges et complexes. Le SMA est un paradigme pour la compréhension et la construction de systèmes distribués. Il y est supposé que les éléments de traitement c'est-à-dire les agents, entités autonomes capables de communiquer, disposent d'une connaissance partielle de ce qui les entoure et d'un comportement privé, ainsi que d'une capacité d'exécution propre (Figure 1.7). Un agent agit pour le compte d'un tiers (un autre

agent, un utilisateur) qu'il représente sans être obligatoirement connecté à celui-ci, réagit et interagit avec d'autres agents. La capacité sociale de coopération, de coordination et négociation entre agents est l'une de leurs principales caractéristiques (Oussalah, 2014).

Un système multi-agents (SMA) est un système composé d'un ensemble d'agents, situés dans un certain environnement et interagissant selon certaines relations. Un agent est une entité caractérisée par le fait qu'elle est, au moins partiellement, autonome. Un système multi agent est plus efficace qu'un agent seul. Ainsi se ressent le besoin en communication pour pouvoir faire des coopérations, des coordinations et des négociations. Le mot « architecture » n'est pas très utilisé dans ce contexte, il est préféré le mot « Organisation ».

Dans le modèle agent la communication est généralement asynchrone. Cela signifie qu'il n'y a pas de flot de contrôle prédéfini d'un agent à un autre. Un agent peut déclencher un comportement autonome interne ou externe à tout moment, et pas seulement quand il est destinataire d'un message.

Les agents peuvent réagir non seulement aux invocations de méthodes spécifiques, mais aussi bien à des événements observables dans l'environnement. Des agents proactifs peuvent effectivement interroger l'environnement pour les événements et d'autres messages pour déterminer les mesures à prendre

Deux agents peuvent se comprendre (échange de messages) s'ils comprennent un même langage et partagent une même ontologie. On parle alors d'ACLs (*Agent Communication Language*) principalement concernés par l'échange d'états mentaux et le sens du vocabulaire.

Un message écrit en utilisant un ACL décrit un état désiré plutôt qu'un simple appel de procédure ou de méthode. Les ACLs s'appuient sur les protocoles de bas niveau pour le transport du message (SMTP, TCP/IP, IIOP, http,..)

Tout langage multi agent doit avoir une structure de donnée comprenant : Émetteur, récepteur, langage utilisé (pour rédiger le message) contenu du message lui-même et l'ontologie du domaine. Exemple : KQML et la norme FIPA

Techniquement, il peut y avoir deux modes de communication :

- Echange directe de Message (point à point, synchrone/asynchrone)
- Echange par mémoire partagée : associative (exemple : architecture tableau noir)

1.4.5 Synthèse

En guise de synthèse, nous avons distingué un ensemble de critères comparatifs à partir de la discussion des différentes approches de descriptions d'architectures logicielles susmentionnées. Dans le tableau 1.1 nous présentons ces différents critères et la position de ces approches par rapport à ces critères.

De cette comparaison, il apparaît que les paradigmes de développement sont issus les uns des autres d'où leur partage de certaines propriétés. Bien que les comparaisons entre l'objet et l'agent soient souvent faites, il demeure quand même qu'il y a une distance sémantique considérable entre les deux concepts. C'est la capacité du raisonnement et d'autonomie qui dicte la particularité de ce paradigme. Pour un système multi-agent on préfère beaucoup plus le terme « organisation » plutôt qu' « architecture ».

Tableau 1.1- Synthèse sur les paradigmes architecturaux.

Critères	Architecture objets	Architecture composants	Architecture service	Architecture Agent
Entité de base	objet	composant	Service	agent
Distinction	Réutilisabilité/Encapsulation	Composabilité	Dynamicité	Autonomie
Méthodologie	bien définie	pas de standard	pas de standard	pas de standard
Outils de support	outils commerciaux	outils plus académique	outils commerciaux	outils plus académique
Familiarité	large communauté	communauté limitée	communauté grandissante	Communauté limitée
Correspondance (modèle→code)	directe	non explicite	aucune	Non explicite
Granularité	faible	élevée	moyenne	élevée
Réutilisation	faible/niveau micro	forte/niveau composant	forte	moyenne
Evolution	difficile	facile	facile	facile
Abstraction	moyenne	élevé	élevé	moyenne
Composition	il faut la programmer	composition native	Composition native	Autonomie
Couplage	fort entre objets	faible entre composants	faible entre service	Faible entre agent
Lisibilité de la structure	peu lisible	très lisible	Peu lisible	Peu lisible
Adaptation à l'assemblage	peu de solutions	plusieurs solutions	Plusieurs solutions	Plusieurs solutions
Intégration d'entités	homogènes	homogènes/ hétérogènes		hétérogène
Schéma de communication	simple	complexe	Complexe	complexe
Eléments de communication	Envoie de massages/ Invocation	connecteur	SOAP protocole	Tableau noir, Point à point
Langages de communication	Langages généralistes	IDL/Protocole	XML	Fipa ACL

L'incidence du composant sur le service est similaire à l'influence de l'objet sur le composant. Si la réutilisation est un caractère natif et existentiel de l'objet, il n'en demeure pas moins qu'elle trouve sa matérialisation concrète dans le composant comme boîte noire sur

étagères. Ceci dit, la principale distinction de composant reste ses aptitudes à la composition ou à la « soudure ». Le service trouve sa naissance dans le composant avec les concepts de « service requis/fourni », sa principale distinction reste la dynamicité qui apporte la principale différence par rapport au composant dans ce qui est la responsabilité et du déploiement côtés Fournisseur/Client. La Figure 1.8 inspirée de (Oussalah, 2014) essaye de traduire ce récapitulatif. Elle montre surtout l'importance et l'originalité du concept « objet » qui représente en quelque sorte son statut de concept carrefour.

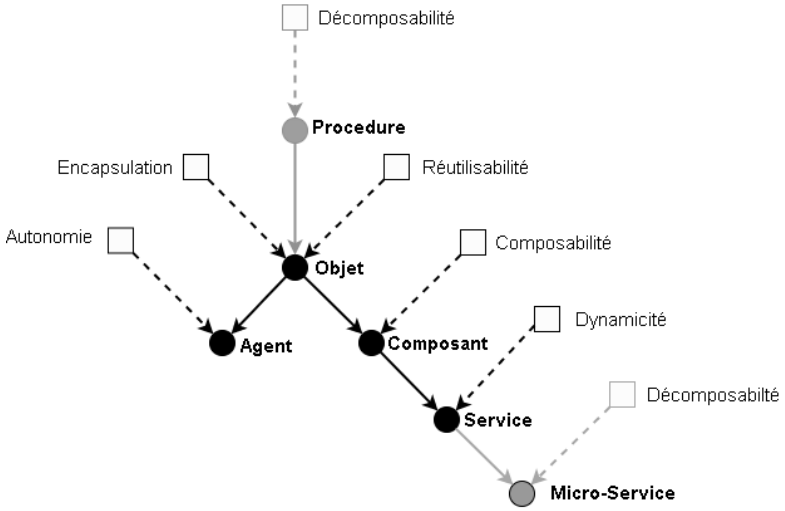


Figure 1.8- Rapport dans les paradigmes de développement.

Comme en génie civil, nous avons besoin d'éléments architecturaux pour pouvoir construire une architecture. Pour des raisons de granularité assez faible, l'objet n'est pas un candidat potentiel pour ce rôle. Une pièce de parpaing ne peut pas être un élément architectural, alors que les murs, les dalles, les poteaux, les portes, … le sont. David Garlan (2008) stipule que le vocabulaire conceptuel de l'objet peut ne pas être idéalement adapté pour représenter des concepts architecturaux, alors qu'Amirat et Oussalah (Amirat et al., 2014; Oussalah, 2014) réitèrent le fait que les modèles à objets exigent l'implémentation de leurs composants avant que l'architecture ne soit complètement définie.

De la même manière, bien que le paradigme orienté service est intimement lié au paradigme composant, il se distingue par le fait de sa concentration sur l'aspect fonctionnel souvent lié et visible en temps d'exécution et non pas en temps de conception. En effet, la fonction d'une cuisine n'est pas très visible sur un plan architectural et donc n'est effective que dans une maison habitable. Ceci est surtout justifié par le caractère dynamique et auto-adaptable du concept « service ».

Toutefois, est-il légitime de poser la question suivante : « Avons-nous fait tout ce chemin pour rien ? quand on sait que nous sommes partis de la décomposition fonctionnelle de la procédure pour aboutir à la décomposition fonctionnelle pour le micro-service ».

Pour toutes ces raisons, seul le paradigme composant est éligible pour servir et valoir comme paradigme dédié aux architectures logicielles. Dans la suite de cette thèse nous nous intéressons exclusivement aux architectures à base composants dans un niveau d'abstraction élevé.

1.5 Architecture logicielle à base de composants

L'architecture logicielle à base de composants représente une évolution naturelle des architectures logicielles à base d'objets. Elle est la matérialisation idéale des notions de

réutilisation et d'encapsulation inhérentes à l'objet. Elle est directement inspirée du monde électronique avec l'utilisation par réutilisation de composants électroniques élémentaires pour former un circuit plus complexe. L'élément est « *réutiliser* » directement à partir des étagères de vente. On ne fabrique plus un circuit, on le « *compose* » ou on l'« assemble ».

De la même manière, une architecture logicielle décrit le système comme un ensemble d'unités de calcul ou de stockage appelées « *composants* ». Le composant est une boîte noire ou un « préfabriqué » acheté sur le marché ou développé par un tiers. Ses fonctionnalités « *services* » ne sont accessibles que via ses points d'interactions « *interface* ». Le système est obtenu par la composition des composants élus entre eux. Cette composition ou assemblage est effectuée avec des connexions ou des « soudures », entre des composants compatibles, communément appelées « *connecteurs* » ou unités d'interaction. Le circuit ainsi obtenu est appelé « configuration » et représente l'architecture du système.

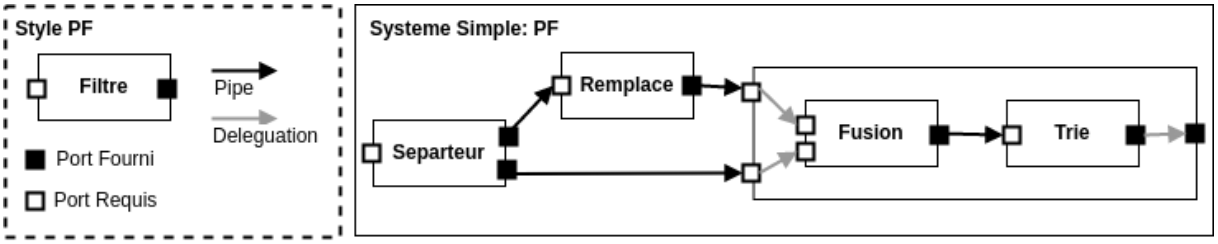


Figure 1.9- Systèmeet style.

Pour réduire la complexité du développement, le coût de maintenance et accroître le niveau de réutilisabilité, deux principes fondamentaux doivent être respectés : « *acheter plutôt que de construire* » et « *réutiliser plutôt que d'acheter* » (McIlroy et al., 1968). Le développement orienté composants est souvent considéré sous deux angles: La catégorie « *for reuse* » qui focalise sur le processus de développement d'un constituant dans le but d'être réutilisé. ii) La catégorie « *by reuse* » se focalise sur le processus de développement d'un composite par réutilisation de constituants existants (Hock-koon & Oussalah, 2011).

1.5.1 Eléments architecturaux de base

Dans le monde de l'architecture bâtiment avec les portes, les fenêtres, les dalles, les murs, …, nous avons besoin d'éléments architecturaux de base pour pouvoir construire une architecture complexe. En fonction du problème posé, un architecte se doit d'abord de sélectionner les éléments architecturaux dont il a besoin pour la solution qu'il propose. Ensuite, il procède à la connexion de ces éléments entre eux dans une forme ou une topologie souhaitée tout en organisant les dépendances entre ces éléments. Le produit ainsi obtenu est une description d'architecture de haut niveau du système en question. Pour ce faire, l'architecte à besoin de disposer d'un langage de description d'architecture (ADL) dédié à la réalisation de telles manipulations. Les architectes bâtiment quant à elles, utilisent principalement AutoCad (Diver, 2010) comme outil ou langage de modélisation.

Comme décrit dans la définition de Perry et Wolf, qui précise qu'une architecture logicielle est un triplet (Elément, Forme, Raisonnement ou logique), les éléments principaux d'une architecture logicielle sont : le composant, le connecteur et la configuration. La forme est définie en termes de propriétés et de relations entre les éléments, c'est-à-dire les contraintes sur les éléments. Le raisonnement concerne la justification fournit la base sous-jacente de l'architecture en termes de contraintes du système, qui découlent le plus souvent des exigences du système (Taylor, 2019).

1.5.1.1 Composant

Le composant trouve son origine dans la notion de « module » et le domaine des

langages d'interconnexion de modules (MILs) (Perry, 1987). La notion de composant logiciel est introduite comme une suite à la notion d'objet. Le composant offre une meilleure structuration de l'application et permet de construire un système par assemblage de briques élémentaires en favorisant la réutilisation de ces briques (M Shaw et al., 1995). Le composant a pris de nombreuses formes dans les différentes approches de l'architecture logicielle. La définition suivante, globalement acceptée, illustre bien ses propriétés.

a) Définition d'un composant

Le composant représente une unité ou un élément de calcul ou de stockage de données dans un système. Il correspond aux boîtes dans les descriptions en boîtes-et-lignes des architectures logicielles. La taille et la complexité d'un composant sont très variables, les exemples de composants incluent les clients, les serveurs, les filtres, les tableaux noirs, les bases de données ou tout simplement une fonction mathématique.

De l'extérieur, un composant est souvent vu comme une « *boîte noire* » qui cache son implémentation. Il n'est accessible que via ses interfaces qui définissent des points d'interactions avec son environnement. A ce titre, les composants logiciels sont donc la concrétisation des principes de génie logiciel en termes d'*encapsulation*, d'*abstraction*, et de *modularité*. Implicitement, cela a un certain nombre de conséquences positives sur la *composabilité*, la *réutilisabilité* et l'*évolutivité* des composants (Taylor et al., 2009).

Comme définition sommaire, nous pouvons citer celle proposée par Richard Taylor et al. (Taylor, 2019) :

Définition 1 : *Un composant logiciel est une entité architecturale qui (i) encapsule un sous-ensemble des fonctionnalités et/ou des données du système, (ii) restreint l'accès à ce sous-ensemble via une interface explicitement définie, et (iii) possède des dépendances explicitement définies sur son contexte d'exécution requis*.

Le principale rôle d'un composant est d'offrir des services (message, opération et variable) à son environnement à travers son interface. Pour ce faire, le composant peut requérir à d'autres services en entrée (fournis par d'autres composants) pour remplir correctement sa mission. À partir de là, on peut dire que les services requis/fournis ainsi que les interfaces qui leurs sont associées sont les seuls moyens d'interactions du composant avec l'extérieur.

Aussi, nous avons les modèles de composants composites ou hiérarchiques qui définissent la hiérarchie entre les composants, c'est-à-dire un composant contenant à son tour des composants ou des systèmes de sous-systèmes.

Dans une bonne démarche de développement, le composant doit conserver ce statut durant tout le cycle de vie de la phase de modélisation à la phase de déploiement et d'exécution pour préserver l'architecture du système de l'érosion. Dans le premier cas ; on parle de « composant type » et de « composant instance » dans les second.

b) Exemple de description d'un composant

L'exemple proposé concerne un système filtre de type Pipe&Filter permettant de lire un flux de caractères pour le transformer en flux contenant les mêmes caractères en lettres capitales. Le schéma présenté dans la Figure 1.10 illustre ce système sous forme de diagramme.

Le système « *Capitalize* » comprend trois composants :

- Le composant « Split » permet de récupérer le flux de caractères en entrée et de créer deux flux :

- un flux identique à celui en entrée pour le composant « Upper » pour la conversion en lettres capitales,
- un flux identique à celui en entrée pour le composant « Merge » en conservant les caractères en minuscule.
- Le composant « Upper » permet de transformer un flux de caractères en flux contenant les mêmes caractères en lettres capitales,
- Le composant « Merge » permet de fusionner les deux flux.

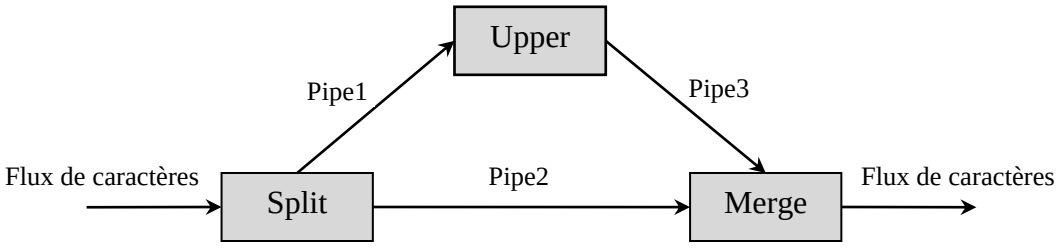


Figure 1.10- Composant de type « Filtre ».

En utilisant la description syntaxique d'ACME (Garlan et al., 2000), le composant *Split* sera décrit comme le montre la Figure 1.11 (Allen & Garlan, 1997).

```
Component Split
{
    Port Input      /* lire les données jusqu'à la fin
    Port Left       /*sortir les données de manière filtrée
    Port Right      /*sortir les données de manière filtrée
    Computation  /* lire continuellement les données à partir du port
                    Input puis les sortir soit dans le port Left s'ils sont en minuscule
                 soit dans le port Right s'ils sont en majuscule */
}
```

Figure 1.11- Description composant en syntaxe ACME.

L'interface du composant « *Split* » est formée de trois ports qui sont le port « Input », « Left » et « Right ». La partie Calcul « *Computation* » du composant permet de mettre en rapport les trois ports.

1.5.1.2 Connecteur

En plus de l'aspect fonctionnel, les architectures logicielles introduisent l'aspect interaction comme concept indépendant de tout élément de calcul relatif au composant. On introduit alors le connecteur qui est défini par Perry et Wolf comme étant l'élément de connexion. Le connecteur est une caractéristique propre et distinctive du développement orienté composant par rapport aux autres paradigmes de développement. Il représente une unité de communication indépendante des composants du système dont le rôle est d'assurer la médiation des activités de communication et de coordination entre les composants.

a) Définition d'un connecteur

Un connecteur modélise l'interaction entre les composants ainsi que les règles qui

régissent cette interaction. Il correspond à la « ligne » dans le modèle de description « boîte-et-ligne ». Le connecteur peut représenter une interaction simple comme un « pipe », un appel de procédure, un diffuseur d'événements, ou bien il peut représenter un protocole complexe, comme le contrôle d'un robot sur Mars par une station de contrôle au sol.

Chaque connecteur possède un type qui spécifie le modèle d'interaction d'une manière explicite et abstraite. Ce modèle peut être réutilisé dans différentes architectures. Selon David Garlan (1993), un connecteur contient deux parties importantes qui sont un ensemble de « *Rôle* » qui détermine son interface et la « *Glu* » qui précise le comportement interne ou son protocole. Aussi, les rôles d'un connecteur permettent d'identifier les participants à l'interaction. Il existe les rôles « *sources* » qui désignent les entrées au connecteur et les rôles « *puits* » qui désignent ses sorties. Les « rôles » représentent l'analogie des « ports » chez les composants. La « glu » définit aussi comment les rôles (source, puits) interagissent entre eux.

Comme définition sommaire, nous pouvons citer celle proposée par Richard Taylor et al. (2019):

Définition 2 : *Un connecteur logiciel est un élément architectural chargé d'effectuer et de réguler les interactions entre les composants*.

Les interfaces du connecteur définissent les rôles joués par les différents participants à l'interaction représentée par le connecteur. Dans le cas des composants composites, on appelle les liens entre les composants de même niveau d'hiérarchie comme étant des « *connecteurs* » et on les appelle « *délégation* » dans le cas contraire. Le connecteur ainsi réifié participe à la capitalisation également des propriétés inhérentes au paradigme composant à savoir : la *composabilité*, la *réutilisabilité* et l'*évolutivité* des composants du système.

Les connecteurs effectuent le transfert de contrôle et de données entre les composants et ils sont extrêmement puissants et variés. Les connecteurs peuvent fournir des services de communication entre les composants en prenant en charge la transmission de données. Ils peuvent fournir des services de coordination en soutenant le transfert de contrôle. Ils peuvent fournir des services de conversion ou effectuer des transformations qui permettent aux composants autrement incompatibles de communiquer. Enfin, ils peuvent effectuer des services de facilitation, tels que l'équilibrage de charge et le contrôle de la concurrence, pour rationaliser les interactions des composants. Cette classification a été proposée dans la taxonomie de Mehta et al. (2000) qui sera présentée en détails dans la 1.5.3).

b) Exemple de description d'un connecteur

En reprenant l'exemple du système de filtre de type « Pipe&Filter », les composants filtres sont liés entre eux par des tubes ou des tuyaux (pipe). Ces derniers fonctionnent de la même façon et obéissent aux mêmes règles. La Figure 1.12 illustre un connecteur « Pipe » définissant un tube (Allen & Garlan, 1997).

```
Connector Pipe
{
        Role  Source    /* lire les données continuellement, signaler la fin et fermer)*/
        Role Sink        /* délivrer les données continuellement, fermer au
                            moment de la signalisation de la fin des données */
        Glue             /*le rôle Sink reçoit les données dans le même ordre que celui
                            utilisé par le rôle source */
}
```

Figure 1.12- Description d'un connecteur dans l'ADL ACME.

De nos jours, le connecteur est devenu un élément architectural central dans les différentes phases du développement. A travers son historique d'évolution, il est passé du statut implicite « appel de procédure » classique au statut explicite c'est-à-dire une entité qui possède. Il obtient une existence propre et tangible (comme le « pipe ») entièrement indépendant du composant. En revanche, une préoccupation sensible est à prendre en compte, c'est celle de conserver ce statut aussi bien en phase de conception qu'en phase de compilation. Richard Taylor (2019) stipule que tel statut donnera plus de flexibilité en temps d'exécution pour effectuer à titre d'exemple des adaptations dynamiques à la volée ou pour effectuer des remplacement « à chaud ».

1.5.1.3 Interface

Dans le paradigme composant, chaque composant explicite les fonctionnalités qu'il offre et qu'il requiert à travers ses interfaces. Partant du fait que le composant est par définition une boîte noire inaccessible, l'interface représente donc son seul moyen pour s'exposer ou communiquer au monde extérieur. La notion d'interface est aussi un concept emprunté au monde du composant électronique (interface série et parallèle).

A titre de définition, les interfaces sont un support de description des composants permettant de décrire l'ensemble de leurs fonctionnalités fournies et requises. Souvent cette description est faite à travers des signatures de méthodes. Aussi, l'interface est considérée comme étant le moyen d'expression des dépendances entre le composant et le monde extérieur.

Toutefois et selon Messabihi (2011), l'interface telle que décrite ci-dessus ne permet pas de décrire complètement le comportement du composant. Pour cela, il faut enrichir cette interface par des éléments spécifiques par l'ajout de certaines propriétés sur les aspects dynamiques et de contraintes liées aux composants et à leurs assemblages. Cet enrichissement permet alors de faciliter la vérification et la prédiction de propriétés d'assemblages de composants.

Ainsi, on introduit le concept de protocole qui est souvent confondu à la notion d'interface. A titre distinctif, on peut dire que le protocole représente l'interface du composant augmentée des contraintes de son utilisation et assemblage avec le monde extérieur. Dans le langage ACME, Garlan (2000) propose séparément la notion de protocole pour le composant tant pour chacun de ses ports en fournit/offert.

Par exemple, un composant dédié à des communications réseau offre les trois fonctionnalités suivantes (Messabihi, 2011):

- ouvrir (adr) pour ouvrir une connexion réseau à l'adresse spécifiée par le

paramètre (adr), l'ouverture est faite une fois.

- envoyer (data) pour envoyer des données 'data' à travers une connexion, l'envoi est répété autant de fois que nécessaire.
- fermer (adr) pour fermer la connexion, la fermeture est faite une seule fois.

Formellement, ce protocole peut s'exprimer par le patron suivant :

- $Protocole = ouvrir\ (adr)\ envoyer^{*}(data)\ fermer\ (adr)$

Ainsi, l'objectif du concept interface est de permettre l'utilisation et surtout la réutilisation du composant en enrichissant ses points d'accès. Ces points d'accès ou d'interaction sont souvent appelés « Port ». Un port de composant donc est un point d'accès à l'ensemble (ou une partie) des fonctionnalités impliquées (fournies ou requises) dans ce composant qui ne peut communiquer que via ses ports.

Il est important de signaler qu'une interface doit être définie de manière indépendante de toute implémentation pour améliorer l'interopérabilité. C'est pour cette raison que l'OMG à proposé un langage complètement dédié à la description d'interface appelé IDL (*Interface Description Language*) (OMG-IDL, 2018). Ce langage avec une syntaxe simple et générique est essentiellement proposé pour promouvoir l'interopérabilité lors de la composition de composants écrits dans différents langages de programmation.

1.5.1.4 Notion de service

Le terme « service » est souvent employé pour désigner une fonctionnalité. Dans les architectures orientées services (SOA), un service est un moyen offrant à un client une fonctionnalité proposée par un fournisseur par des mécanismes de publication/souscription. Le fournisseur ne connaît pas le client au préalable et le client n'a aucune connaissance sur la réalisation du service attendu. Le client n'a aucune responsabilité sur le service en termes de déploiement de sa source de production.

Dans le contexte de paradigme orienté composant, un service est fourni par un composant. Un composant peut également requérir d'autres services pour assurer ceux qu'il doit fournir. Dans ce cas, il est en même temps fournisseur et client.

Généralement le service est exhibé par l'interface. Selon Messabihi (2011), il existe plusieurs manières avec lesquelles un service est rattaché à un composant. On peut citer:

(i) un service correspond à une interface exposée par un composant. Cette interface liste les opérations du service. Le service correspond alors à un ensemble cohérent d'opérations réalisées par un composant et exposées dans une interface.

(ii) une interface expose des services, chaque service correspond à une fonctionnalité.

(iii) un service possède plusieurs interfaces, chaque interface liste les opérations du service. Il est ainsi possible d'accéder de plusieurs manières au service. Ceci est utile lorsque le service évolue, chaque interface peut correspondre alors à une version du service « historique ».

(iv) un service est réalisé par plusieurs composants (vision SOA).

Il existe deux genres de services, les services offerts et les services requis: (i) le service offert d'un composant désigne une fonctionnalité assurée (à travers son interface ou ports fournis) par ce composant à partir de son comportement interne. Cette fonctionnalité est

offerte à l'environnement externe formée essentiellement par les composants qui la requièrent. (ii) le service requis d'un composant qui exprime le besoin d'une fonctionnalité externe et nécessaire pour le bon fonctionnement du composant en cours ou son comportement interne. Ce type de service est assuré ou fourni par d'autres composants connectés en amont à travers l'interface ou les ports requis.

Le nombre de services fournis par un composant est aussi une caractéristique importante. S'il n'offre qu'un service alors ce composant aura une granularité faible. Dans ce cas, le système en question devra comporter autant de composants que de fonctionnalités ou de services et devient ainsi difficilement maîtrisable. Dans le cas contraire, un composant qui offre plusieurs services est mieux souhaité. Le composant propose donc un point d'accès centralisé à plusieurs services permettant d'appréhender plus facilement la complexité globale du système (Messabihi, 2011).

1.5.1.5 Configuration

Les composants et les connecteurs sont composés d'une manière spécifique dans l'architecture d'un système donné pour accomplir son objectif. Cette composition représente la *configuration* du système, également appelée *topologie*. Plusieurs définitions ont été proposées pour le concept de configuration. Dans ce qui suit, nous donnons la définition la plus adoptée.

a) Définition d'une configuration

Une configuration architecturale (ou simplement architecture ou système) est un « *graphe* » qui montre la façon dont un ensemble de composants sont reliés les uns aux autres par l'intermédiaire de connecteurs. Les nœuds du graphe représentent les composants alors que les arcs représentent les connecteurs

Le graphe, qui un ensemble de nœuds et d'arcs (Awodey, 2010), est obtenu en associant les ports, qui forment les interfaces, des composants avec les rôles, qui forment les interfaces, des connecteurs adéquats, en vue de construire l'application. Par exemple, les ports des composants de type Filter « *Split* » sont associés aux rôles de connecteurs de type Pipe « *Pipe1* » à travers lesquels ils lisent et écrivent des flux de données (Figure 1.13).

Comme définition sommaire, nous pouvons citer celle proposée par Richard Taylor et al. (2019):

Définition 3 : *Une configuration architecturale est un ensemble d'associations spécifiques entre les composants et les connecteurs de l'architecture d'un système logiciel*.

La configuration représente une synthèse sur les choix pour les éléments architecturaux. Elle répond aux questions concernant la séparation des préoccupations, le choix des connexions entre composants et l'organisation de leurs dépendances. Ceci détermine forcément les propriétés et les qualités attendues du système. C'est à ce niveau que les principales décisions conceptuelles sont prises.

L'analyse d'une configuration permet par exemple de déterminer si une architecture est « *trop profonde* », ce qui peut affecter la performance due au trafic de messages à travers plusieurs niveaux hiérarchiques, ou « *trop large* », ce qui peut conduire à trop de dépendances entre les composants (Amirat, 2010). Un système peut aussi être hiérarchique, les composants et les connecteurs peuvent représenter des sous systèmes ayant des architectures internes. On parle alors d'Hypergraphe (Wang et al., 2018).

b) Exemple de description d'une configuration

Pour avoir une meilleure compréhension, on prend l'exemple du système de filtre de

type Pipe-Filter (Figure 1.10), à laquelle nous ajoutons un ensemble de détails pour obtenir une configuration plus précise qui correspond à l'architecture du système *Capitalisation* illustrée par la Figure 1.13 adaptée de (Amirat, 2010).

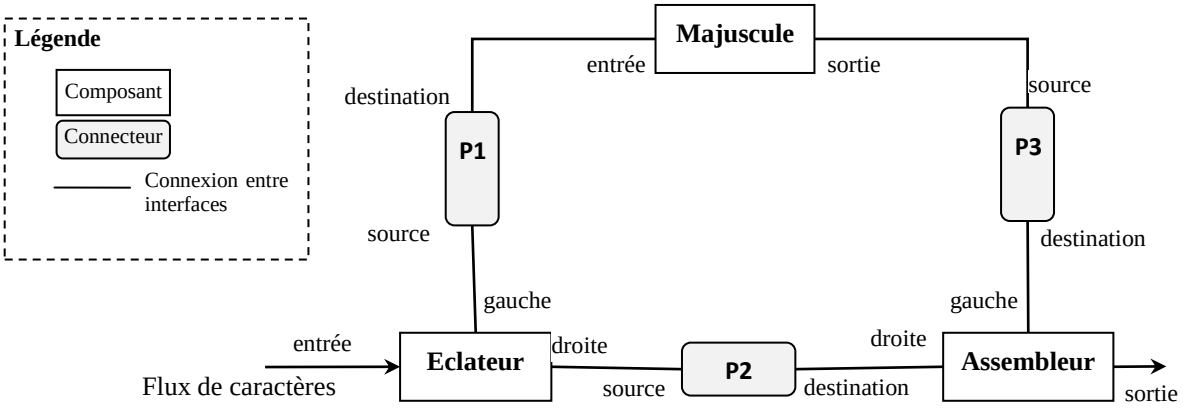


Figure 1.13- Exemple d'une configuration du système "Capitalisation".

Cette figure illustre la configuration architecturale de l'exemple pour convertir en caractère capital. Dans la Figure 1.13, *Eclateur* et *Assembleur* sont des exemples de composants tandis que P2 est le connecteur qui les relie. La configuration illustrée dans le diagramme indique que ces deux composants peuvent interagir mutuellement via un connecteur. Les interfaces ne sont pas affichées, car elles doivent avoir une forme visuelle. Toutefois, on peut considérer le composant « Eclateur » comme ayant trois ports avec un en entrée et deux en sortie (gauche, droite) alors que le composant « Assembleur » possède deux ports en entrée (gauche, droite) et un port en sortie. Le connecteur « P2 » joue deux rôles dans cette connexion : un en entrée (source) en relation avec le port (droite) du composant « Eclateur » et un autre en sortie (destination) en relation avec le port (droite) du composant « Assembleur ». Ceci détermine ce que David Garlan (2000) appelle « *Attâchement* ».

```
Configuration Capitalize
        Component Split, Upper, Merge /*déclaration des composants
        Connector Pipe /* déclaration de connecteurs
        Instances /*déclaration des instances
                Eclateur : Split
                Majuscule : Upper
                Assembleur : Merge
                P1, P2, P3 : Pipe
        Attachments /*description des liens entre les composants et les connecteurs*/
                Eclateur.Left to P1.Source
                Majuscule.Input to P1.Sink
                Eclateur.Right to P2.Source
                Assembleur.Right to P2.Sink
                Majuscule.Output to P3.Source
                Assembleur.Left to P3.Sink
End Configuration
```

Figure 1.14- Description textuelle (ACME) d'une configuration.

La description textuelle selon le langage ACME de la configuration capitalisation

(*Capitalize*) est donnée par la Figure 1.14. Elle présente les types « composant » (Split, Upper, Merge) « connecteur » (Pipe) et leurs instances respectives (Eclateur, Majuscule, Assembleur) pour les composants et (P1, P2, P3) pour le connecteur. Aussi, elle définit clairement le concept d'« *attâchement* » qui assure les correspondances entre les ports des composants et les rôles des connecteurs impliqués.

1.5.2 Style Architecturaux

La notion de style architectural est exclusivement empruntée dans l'architecture du génie civil. On retrouve une chronologie historique en fonction de l'évolution des sociétés et des civilisations. On connaît les styles architecturaux romain, gothique, de la renaissance, classique, islamique ou autre moderne. Chaque style partage un certain nombre de caractéristiques qui le distinguent des autres styles. Utiliser un style revient à utiliser les solutions et les recommandations souvent préétablies. Un style renseigne toujours sur son époque ou son domaine d'appartenance.

La même idée s'était vue utilisée tôt dans le monde de l'informatique et de la programmation en particulier. En effet, la notion de programme principal et de procédure ou la programmation orientée objets ne sont que des exemples typiques ainsi que les patrons de conception dans un niveau conceptuel. Les patrons de conception (Gamma, 2002) représentent l'origine des styles dans les architectures logicielles.

L'objectif essentiel pour l'adoption de la notion de style est la capitalisation et la promotion de la réutilisation. L'adoption d'un style apporte nominalement un certain ensemble d'avantages bien connus et déjà validés à la nouvelle solution (Taylor, 2019). Au fur et à mesure que les styles architecturaux sont utilisés pour capturer des leçons d'expériences applicables à des problèmes à plus grande échelle, ils commencent à capturer des informations spécifiques au domaine et deviennent de fait des solutions connues et localisées dans ce domaine.

Comme définition sommaire, nous pouvons citer celle proposée par Richard Taylor et al. (2019):

Définition 4 : *Un style architectural est un ensemble nommé de décisions de conception architecturale qui :*

- *sont applicables dans un contexte de développement donné*
- *restreignent les décisions de conception architecturale spécifiques à un système particulier dans ce contexte*
- *suscitent des qualités bénéfiques dans chaque système résultant*

On peut aussi retenir une autre définition de David Garlan (1995) qui est le fondateur en personne de ce concept :

Définition 5 : *Les styles architecturaux sont des modèles et des idiomes organisationnels récurrents.*

On plus de la réutilisation de la conception et du code, l'utilisation des styles peut avoir d'autres avantages et bénéfices comme la compréhension de l'organisation conventionnelle (une expression telle que « client-serveur » véhicule beaucoup d'informations), l'interopérabilité supportée par la standardisation des styles, l'analyse spécifique de certains styles et la connaissance préalable des contraintes et attentes et enfin la visualisation ou la représentations spécifiques au style correspondant aux modèles mentaux des architectes.

On identifie plusieurs styles récurrents dans le domaine des architectures logicielles

comme ceux liés à la programmation traditionnelle. On peut citer les styles relatifs aux couches comme la machine virtuelle ou le « Client-Serveur », ceux relatifs aux flots de données comme le « batch séquentiel » ou le « Pipe&Filter », ceux relatifs à la mémoire partagée comme le « tableau noir » ou les systèmes a « base de règle »s, ou ceux relatifs aux invocations implicites comme les systèmes à « base d'événements » ou « Publication-Souscription » et enfin autres comme les systèmes « Peer-to-Peer » pour ne citer que ceux là.

Dans ce qui suit, nous allons nous intéresser à deux styles très connus et très redondants dans la communauté des architectures logicielles à savoir le style « Pipe&Filter » pour les flots de données et de contrôles ainsi le style « Client-Serveur » pour les systèmes en couches ou hiérarchiques. Ces deux styles seront d'ailleurs très sollicités dans cette thèse.

1.5.2.1 Pipe and Filter

Le style « Pipe&Filter » est un exemple standard des styles de flots de données. On peut les voir comme des programmes séparés qui sont exécutés dans l'ordre ; les données sont transmises sous forme d'agrégat d'un programme à l'autre. Ces éléments agrégés sont passés d'un composant à l'autre à la fin de l'exécution du programme producteur. La notion de « Pipe » ou tuyau est empruntée à l'industrie pétrolière avec le concept de « Pipe-Line » qui consiste à créer des stations de pompages interposées pour transporter le brut sur de longues distances. Il est évident que le pompage se fait toujours dans un seul sens.

Dans le style « Pipe&Filter », les composants sont des « filtres » qui transforment le flux de données en entrée pour produire un flux de données en sortie. Les connecteurs sont des « pipe » qui sont des tuyaux ou des conduites pour le flux de données. Les invariants associés à ce style font que les filtres sont indépendants c'est-à-dire qu'ils ne partagent pas leur état interne) ; aussi un filtre n'a aucune connaissance des filtres en amont ou en aval. Les exemples de ce style figurent souvent dans les pipes supportés par le système d'exploitation Unix, les systèmes de traitement de signal, les systèmes distribués et la programmation parallèle.

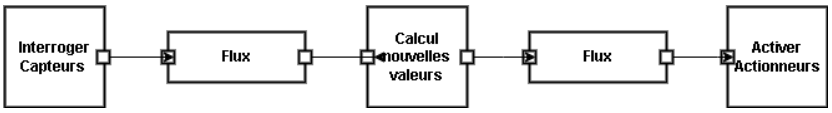


Figure 1.15- Style « Pipe&Folter ».

La Figure 1.15 représente un système de capture dans le style « Pipe&Filter ». Il est formé de trois composants de type filtre et de deux connecteurs (pipe) du type flux. Le composant 1 lit les capteurs et envoie les données via le connecteur de type flux (Section 1.6.3) vers le composant 2. Ce dernier fait les traitements nécessaires sur les données reçues et envoie les résultats dans le deuxième pipe pour être acheminé vers le composant 3 qui, lui, ira agir sur les connecteurs en fonction des données reçues. La diffusion n'est pas forcément linéaire mais elle peut être multiplexée/dé-multiplexée.

1.5.2.2 Client Serveur

Le style « Client-Serveur » est un exemple typique des styles en couches ou hiérarchiques. On parle également de « Client-Serveur multi-couches » où chaque niveau expose son interface pour être utilisé pour le niveau au-dessus. Chaque niveau agit comme un « Serveur » qui joue le rôle de fournisseur de service pour le niveau au-dessus. Le même niveau peut aussi agir comme « Client » consommateur de services du niveau au dessous. Les connecteurs sont des protocoles d'interaction entre les couches. Les architectures n-tiers sont des exemples de ce type de style.

Les avantage directs de ce style spécifique peuvent être énumérés comme suit : a)

augmenter les niveaux d'abstraction b) promouvoir l'évolutivité, les changements dans un niveau affectent au plus les seuls niveaux adjacents c) autoriser différentes implémentations dans une couche tant que l'interface est préservée d) permettre la standardisation des interfaces pour un niveau dans des bibliothèques et des Framework.

La Figure 1.16 montre un système en style « Client-Serveur » pour un système de jeux en ligne en deux niveaux. Les composants sont des clients et des serveurs. Les serveurs ne connaissent pas l'identité et le nombre des clients. Les clients, quant à eux, connaissent l'identité du serveur. Les connecteurs sont des protocoles d'interaction réseau basés sur les appels de procédures distantes ou RPC (*Remote Procedure Call* (Giacomo et al., 2021; Mary Shaw, 1993)).

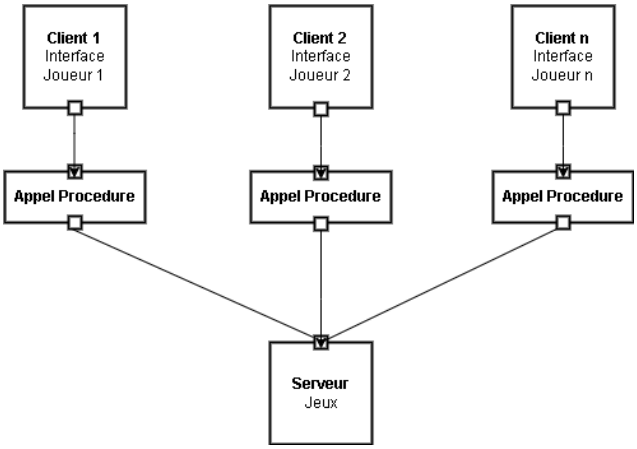


Figure 1.16- Style « Client-Serveur ».

1.5.3 Langage de descriptions d'architecture

En accompagnement de la notion d'architecture logicielle, des formalismes sont apparus au cours des années 90 : les ADLs (*Architecture Description Languages*) ou langages de description d'architecture à base de composants qui sont utilisés pour décrire la structure comme un assemblage d'éléments logiciels. Cela est illustré par la Figure 1.17 où on voit un client et un serveur reliés par un appel de procédure à distance (*Remote Procedure Call*, RPC) (Amirat, 2010).

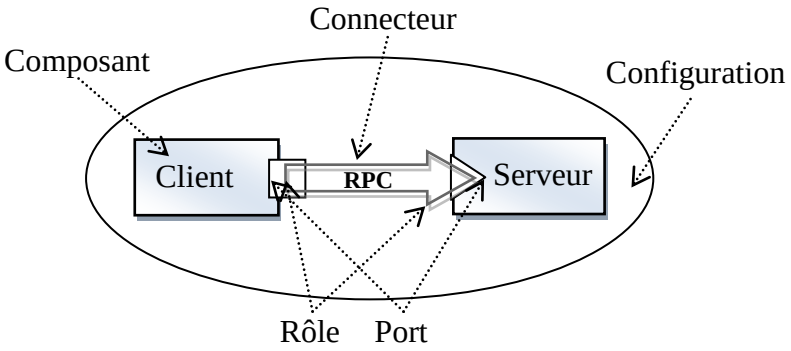


Figure 1.17- Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL.

Darwin, ACME, UniCon, Rapide, Aesop, C2SADL, MétaH (Medvidovic & Taylor, 2000). De manière très générale, les entités manipulées par ces langages sont des éléments bien identifiés, suffisamment abstraits et les plus indépendants possibles les uns des autres. La

spécification d'un assemblage de tels éléments constitue leur construction fondamentale. De ce point de vue, même s'ils affichent a priori une certaine adéquation avec la conception à base de composants lorsqu'elle vise la réalisation d'un système par assemblage de composants logiciels préexistants, nous rappelons qu'ils ont été originellement motivés par la maîtrise de la structure de plus en plus complexe des systèmes logiciels (Amirat, 2010). Dans la section 1.7.2) nous décrivons l'ensemble des concepts que l'on trouve dans les ADLs.

1.5.4 Développent à base de composants

Dans le développement à base de composants on distingue souvent deux aspects ou deux facettes que l'on se place du côté du développeur du composant ou bien de son utilisateur qui est l'architecte ou le développeur de l'application. En effet, le composant est destiné à être « réutilisable » alors que le système est seulement destiné à être « utilisable » pour satisfaire ses fins.

Pour cela, une attente particulière est adressée au développement orienté composant. On parle alors d'une nouvelle façon de voir, elle consiste à distinguer entre deux concepts : développer « pour la réutilisation » et développer « par la réutilisation » ou ce qu'on appelle communément « *For/By reuse developpement* ».

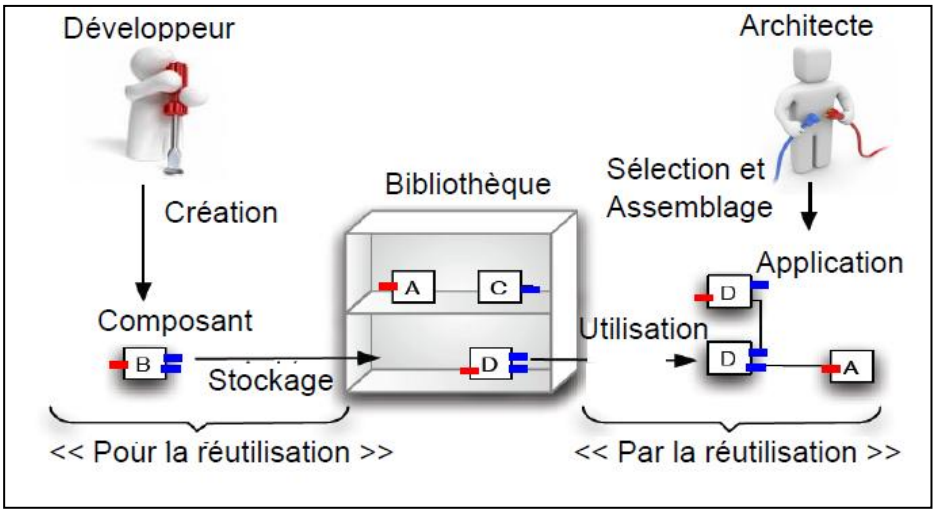


Figure 1.18- Développement 'For/By reuse'.

La Figure 1.18 (Messabihi, 2011) résume parfaitement cette tendance. On voit clairement deux intervenants distincts : le développeur et l'architecte. Le développeur est celui qui crée le composant avec une intention particulière qui est d'être destiné à la réutilisation. Il est alors stocké ou rangé sur les étagères d'une bibliothèque au même titre qu'un composant matériel. L'architecte est celui qui crée le système ou l'application en choisissant et en réutilisant les composants à partir de ces étagères pour procéder à des assemblages ou des compositions. Ces assemblages sont effectués en connectant les parties de liaisons de ces composants pour former une topologie particulière.

Cette topologie est souvent appelée architecture logicielle. À partir de ce qui précède, on voit bien l'émergence des deux propriétés fondatrices du paradigme composant à savoir la « réutilisation » et la « composition ».

1.6 Connecteurs

Les systèmes à base de composants sont une composition d'élément « composants »

préfabriqués, hétérogènes et souvent engagés dans des interactions complexes. Cette interaction, non fonctionnelle, est très généralement réalisée via les connecteurs. Comme l'idée des composants vient des composants électroniques, la notion du « connecteur » logiciel est aussi issue du même contexte. En effet, un connecteur n'est autre qu'une image d'un câble LPT reliant un ordinateur à une imprimante tout en empruntant même le jargon associé comme le « port » et l' « interface ».

Le câble d'imprimante est complètement indépendant des composants qu'il relie et de ce fait n'a aucune connaissance de l'information qu'il transporte. Un ordinateur ou une imprimante n'ont aussi aucune connaissance du câble ou du tuyau et de son mode de fonctionnement. Le rôle de l'émetteur est de mettre les informations qu'il désire transmettre au niveau de ses ports. Le récepteur reste également oisif tant que ses ports d'entrée ne sont pas activés. Bien entendu, il est inutile de rappeler l'existence même du câble puisque les branchements sont inhérents à l'architecte.

Un câble LPT est destiné pour les liaisons simples de type point à point sans aucune intervention sur le contenu de l'information transportée. Pour des transferts plus complexes comme le fait de relier des machines non compatibles ou nécessitant un traitement particulier sur les données, on aura forcément besoin d'un autre type de câble comme les adaptateurs qui peuvent être dotés de fonctions de transfert particulières. Le connecteur logiciel obéit à cette même logique.

Puisque notre approche est centrée sur le connecteur d'une part et pour éclaircir et comprendre en profondeur ce concept, nous allons considérer les connecteurs comme une entité de première classe et proposons une classification rigoureuse ainsi qu'une taxonomie globale du domaine résorbant toute ambiguïté. Pour cela, nous faisons référence dans cette section à la classification et la taxonomie proposée par Mehta et al. dans un article de référence (Mehta et al., 2000). Cette taxonomie est largement admise dans la communauté architecture logicielle et représente un consensus absolu.

Comme tout élément architectural, un connecteur logiciel à une structure et un comportement. La structure représente les constituants élémentaires et leur organisation alors que le comportement décrit toujours l'aspect fonctionnel et dynamique du connecteur.

1.6.1 Structure d'un connecteur

L' « appel de procédure » dans la programmation structurée représente l'origine native du connecteur logiciel. Il met en relation un programme principal avec une procédure ou un module tout en faisant passer un certain nombre de paramètres ou informations d'exécution. Ce concept, en revanche, n'est visible que dans le code source d'un programme. Dans le niveau conceptuel il n'a aucune existence alors qu'au niveau de l'exécution il se résume à un simple saut de branchement.

Au début de la naissance du paradigme de développement orienté composant, l'ensemble des approches d'étude et de développement consacrent tous les efforts sur les composants. Avec l'avènement de l'internet et le besoin de distribution, l'intérêt s'est porté sur l'interaction dans le composant et ce pour des raisons d'adaptation, composition et réutilisation. L'interaction entre composants est matérialisée par la notion de connecteurs logiciels.

A travers son historique, le connecteur a évolué donc en fonction des demandes en interaction plus ou moins complexe ainsi que des besoins d'abstraction et de réification. De ce fait, on vu passé le connecteur d'un statut implicite à un statut explicite pour devenir un élément architectural de première classe. Ce statut doit être désormais conservé tout le long du

cycle développement du système c'est-à-dire de la conception au déploiement.

Les auteurs de la taxonomie de référence de Mehta et al. (2000) identifient trois éléments atomiques d'une interaction à savoir : les canaux ou les conduites, et les mécanismes de transfert de données et ceux de transfert de contrôle.

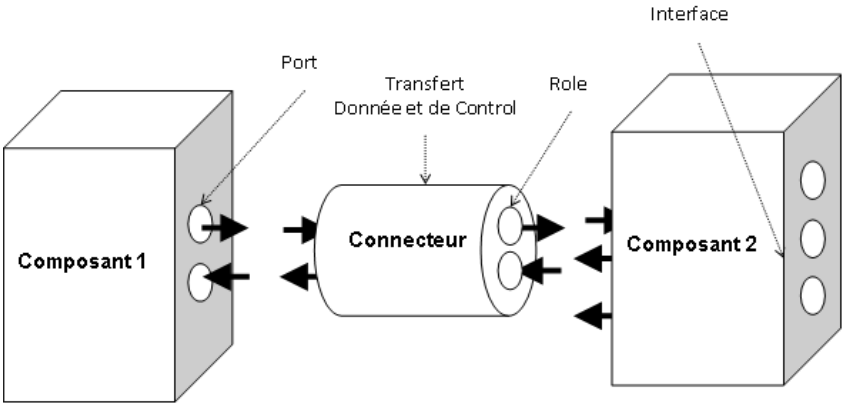


Figure 1.19- Représentation informelle d'un connecteur.

La Figure 1.19 présente une représentation symbolique et informelle d'un système orienté composants. Il est constitué de deux composants et d'un connecteur ou conduite qui transporte le flux de données et/ou le flux de contrôle entre ces deux composants. Il fait apparaêtre tous les éléments architecturaux et leurs distinctions.

Principalement, le connecteur peut être vu comme un composant qui assure des services de communication particulier. Au même titre qu'un composant, c'est une boîte noire qui exhibe ses services à travers son interface. Shaw et Garlan (1993) définissent les éléments principaux pour un connecteur ou sa structure interne comme suit:

- « *Role* » : représente la partie structure du connecteur et constitue son interface. Le « role » est pour le connecteur ce que le « port » est pour le composant. Il détermine les portes d'interaction du connecteur avec l'extérieur. On parle alors de « rôle requis » et « rôle fournis ».
- « *Glue* » ou la « colle » : définit la fonction de transfert du connecteur en faisant la liaison entre les « roles » d'entrée et les « roles » de sortie du connecteur.
- « *Attâchements* » : sont des éléments qui se trouvent à l'extérieur du connecteur. Ils sont de simples liaisons entre le connecteur et les composants qu'il relie. Ils associent le « *role* » du connecteur au « *port* » correspondant du composant. Ces attâchements sont généralement définis au niveau de l'architecture. Ils n'appartiennent ni au composant ni au connecteur mais à la configuration.
- « *Délégations* » : sont des éléments associés aux structures composites. Ils font le lien ou l'attâchement entre les éléments englobant et les éléments englobés (Figure 1.9). Ainsi, ils lient les ports requis/requis en entrée du composite et les ports fournis/fournis en sortie du même composite et ce en fonction de tout niveau d'hiérarchie. Ce concept est largement admis dans les standards à partir d'UML 2.0.

La sémantique dénotationnelle qui décrit le comportement des « roles » et de la « glue » est souvent exprimée en CSP (Lamport & Schneider, 1984) et plus récemment en π-Calcul (Miller, 1992) selon la classification de Mert Ozkaya (Fujita & Herrera-Viedma, 2018). Le

besoin d'exprimer le comportement de ces entités avec des langages formels est souvent justifié par le fait de pouvoir effectuer des analyses et vérifications formelles ultérieures.

1.6.2 Catégorie de service de connecteur

La taxonomie proposée par Mehta et al. (2000) prévoit la structure générale du cadre de classification des connecteurs (Figure 1.20). Chaque connecteur est identifié par sa catégorie de service principale qu'il assure. Ces catégories sont ensuite affinées en fonction des choix effectués pour la réalisation de ces services. Les caractéristiques couramment observées parmi les connecteurs se situent vers le haut du cadre, alors que les variations se situent dans les couches inférieures (Figure 4.2).

Le Framework comprend les catégories de service ou les rôles qu'ils remplissent, les types de connecteurs ou la manière de réaliser ces services, les dimensions (possible sous-dimensions) ou les détails architecturaux. Au niveau le plus bas, on a les valeurs que peuvent prendre ces dimensions.

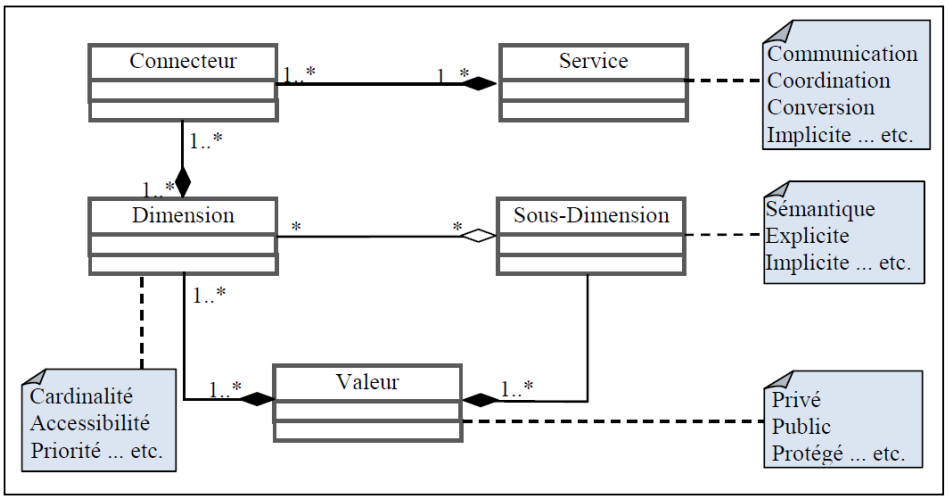


Figure 1.20- Classification des connecteurs

On a identifié quatre catégories de service (Perry, 1997) associées aux connecteurs qui spécifient les services d'interaction fournis par un connecteur en fournissant une classification singulière de haut niveau des rôles que jouent les connecteurs logiciels dans une architecture logicielle. Ces catégories se résument à ce que suit :

Communication : Les connecteurs communicateurs assurent le « transfert ou l'échange des données » et les résultats de calcul parmi les composants. La « communication » est un service ou un rôle principal associé aux connecteurs. Ils supportent les différents mécanismes de communication (appel de procédure, RPC, accès variable partagée, passage de message) ainsi que les contraintes de communications, leur structure/direction (Pipe) et enfin la qualité du service (persistance).

Coordination : Les connecteurs coordinateurs supportent le « transfert de control » à travers les composants. Les composants interagissent en se passant leurs états d'exécution. L'appel de fonction et l'invocation de méthodes sont des exemples de coordination comme de communication. Ils déterminent le contrôle de traitement et de délivrance des données. Ils séparent le contrôle du traitement. Ils *supervisent* l'interaction en matière d'invocation et d'appel de fonctions pour permettre la communication. Aussi, ils peuvent déterminer l'ordre d'interaction ou leur synchronisation.

La coordination est *orthogonale* à la communication, conversion et facilitation car les éléments de contrôles sont nécessaires dans toutes ces catégories.

Conversion : Les connecteurs convertisseurs *convertissent l'interaction* requise par un composant en celle fournie par un autre. Ils permettent à des composants hétérogènes développés indépendamment de communiquer entre eux. La conversion de format de données, les adaptateurs et les *wrappers* pour les composants existants en sont des exemples. La conversion peut concerner la fréquence, le nom des services et leurs nombres, l'ordre d'interactions ...etc.

Facilitation : Les connecteurs facilitateurs assurent la *médiation et la rationalisation* de l'interaction entre des composants hétérogènes destinés initialement à interagir. Ils permettent de gérer l'accès aux variables partagées, d'assurer les profils de performance appropriés (répartition de charge) et de fournir les mécanismes de synchronisation (sections critique, moniteur). Les données échangées doivent être soumises à des règles d'optimisation pour faciliter l'interopérabilité entre composants hétérogènes ou non. C'est dans cette vision que ce type de connecteurs est mis en place. Les problèmes de confidentialité, de compétitivité, de performance, et autres sont à l'origine de ce type de connecteur. Ce type de connecteur peut aussi stocker de l'information.

Chaque connecteur fournit un service appartenant à au moins une de ces catégories. Il est aussi possible d'avoir des connecteurs multi-catégories qui permettent de composer des services d'interactions plus riches. Il est possible d'avoir également un connecteur qui fournit à la fois des services de communication et de coordination.

1.6.3 Type de connecteur

Le niveau de classification précédent donne une catégorisation des connecteurs d'une manière compacte en ignorant les détails nécessaires pour créer de nouveau connecteurs ou de les modéliser et analyser. Ce niveau montre comment les catégories de services sont réalisées.

Les auteurs ont développé un autre niveau de classification autour des *types de connecteurs* basé sur la façon avec laquelle ils *réalisent le service* d'interaction. Ils identifient huit types de connecteur : appel de procédure, événement, accès aux données, liaison, flux, arbitre, adaptateur et distributeur (Figure 1.20).

Les détails de chaque type de connecteur représentent des variations au moment de l'instance et sont traités à travers des *dimensions* et *sous-dimensions* dans la taxonomie comme des paramètres ayant des ensembles de *valeurs* possibles pour chacune. La sélection d'une valeur unique dans chaque dimension donne concrètement une *espèce* de connecteur.

L'instanciation des dimensions d'un type de connecteur unique forme des connecteurs simples; d'autre part, l'utilisation de dimensions provenant de différents types de connecteurs conduit à une espèce de *connecteur composite* («d'ordre supérieur»).

Appel de procédure : Ce connecteur modélise le flux de contrôle entre les composants à travers différentes technique d'invocation (*coordination*). Aussi, il effectue le transfert de données par le biais de paramètres (*Communication*). Il permet de construire des connecteurs composites plus complexes (RPC). Ce sont parmi les connecteurs les plus utilisés surtout dans le langage d'assemblage des logiciels interconnectés (Mary Shaw, 1993).

Évenement : Un événement est un effet instantané pour une terminaison normale ou anormale de l'invocation d'une opération d'un objet. Ces connecteurs sont similaires aux connecteurs appel de procédure car ils modélisent le flux de contrôle entre les composants

(*coordination*). Un événement peut contenir des informations lui concernant comme le temps et le lieu d'apparition ainsi que certaines informations spécifiques à l'application (*Communication*). Une fois qu'un événement est détecté, le connecteur événement procède à la génération de messages pour toutes les parties intervenantes et effectueun contrôle au niveau de tous les composants intéressés par ces messages. Ces connecteurs se trouvent souvent dans les applications distribuées qui requièrent une communication asynchrone.

Accès aux données : Ces connecteurs permettent l'*accès aux de données stockées* par un composant de stockage (*communication*). Les données (permanentes ou temporaires) ne sont pas toujours livrées dans les formats avec lesquels ils ont été stockés, de ce fait le connecteur doit être en mesure de les fournir dans le format sollicité pas le composant ce qui relève du type de connecteurs de (conversion).

Liaison : Ils sont utilisés pour *lier* les composants du système et les *maintenir dans cet état* pendant leur fonctionnement. Ils permettent l'établissement de conduites, de canaux de communication et de coordination. Ces canaux sont ensuite utilisés par les connecteurs d'ordre supérieur pour appliquer la sémantique d'interaction (*facilitation*).

Des exemples de connecteurs de liaison sont les liens entre les composants et les bus dans une architecture de style C2 (Lun & Chi, 2010) et les relations dedépendance entre les modules logiciels décrits par les langages d'interconnexion de modules (MIL) .

Flux : Sont des connecteurs utilisés pour effectuer des transferts *de grandes quantités de données* entre processus autonomes (*communication*). Sont également utilisés dans les systèmes client-serveur avec des protocoles de transfert de données pour fournir des résultats de calcul. Ils sont utilisés aussi dans des modèles architecturaux formels pour représenter des connecteurs avec des protocoles d'utilisation assez complexes.

Ce type de connecteurs peut être combiné avec d'autres types comme, par exemple, avec les connecteurs d'accès aux données pour fournir un connecteur composite de transfert et d'accès aux données. Des exemples de connecteurs de flots sont : les canaux en UNIX, les sockets de communication TCP/UDP et les protocoles client-serveur propriétaires.

Arbitre : Ils sont utilisés pour *résoudre tout conflit* entre les composants et donc aident et assurent la médiation des services entre eux (*Facilitation*). Les besoins ou les états de composants peuvent ne pas exprimer ce qu'ils offrent ou requièrent. Dans de tels cas, les connecteurs arbitres permettent de résoudre les conflits et de rationaliser l'interaction. Ainsi, ils dirigent et redirigent les flux de contrôle entre les composants (*coordination*). En plus, les arbitres offrent des services d'équilibrage de charge, de planification, de fiabilité et de sécurité du système (Stavridou, 1999).

Comme exemple, les systèmes multi-threads qui nécessitent un accès à la mémoire partagée, utilisent la synchronisation et le contrôle de simultanéité pour garantir la cohérence et l'atomicité des opérations.

Adaptateur : *Facilitent l'interaction* entre des composants initialement *hétérogènes* (exp : différents environnement d'exploitation, différents langages, … etc). Ils font correspondre les stratégies et protocoles de communication des différents composants (*conversion*) pour leur interopérabilité. Ils optimisent des interactions pour mieux servir le système en exécution. Ils effectuent également des transformations par « matching » des services requis avec les facilitations désirées.

XML métadata interchange (XMI) est un exemple concret de connecteurs qui supporte l'échange de données entre applications hétérogènes.

Distributeur : Effectuent l'*identification des chemins* d'interaction et le *routage*

ultérieur des informations de communication et de coordination entre les composants le long de ce chemin (*facilitation*). Ce type de connecteur n'existe jamais par lui-même, mais fournit une assistance à d'autres connecteurs pour des exigences de distribution.

Le Tableau 1.2 résume les relations qui lient les types de connecteur avec les catégories de services qu'ils offrent. Cette classification est aussi une instance de la classification présentée dans la Figure 1.20.

Tableau 1.2- Relation entre les types de connecteurs et leurs services

Lien Service/Type	Communication	Coordination	Conversion	Facilitation
Appel de procédure	Fort	Fort	Faible	Moyen
Evènement	Fort	Fort	Faible	Moyen
Accès aux données	Faible	Fort	Fort	Faible
Liens	Moyen	Faible	Faible	Fort
Flots	Faible	Fort	Faible	Faible
Arbitres	Moyen	Fort	Faible	Fort
Adaptateurs	Faible	Faible	Fort	Faible
Distributeurs	Faible	Faible	Faible	Fort

1.6.4 Composition de connecteur

Dans de nombreux systèmes, un connecteur de plusieurs types peut être nécessaire pour entretenir les composants afin de prendre en charge des interactions plus complexes. En fonction des types élémentaires, un connecteur plus complexe peut être composé en tenant compte de leur compatibilité et leur interopérabilité. Le connecteur résultant peut lui-même avoir une architecture interne et peut aussi stocker de l'information. L'ensemble des connecteurs peuvent nécessiter des compromis pour leur composition. Cette composition peut être considérée au niveau des dimensions et sous-dimensions de type connecteur.

A titre d'exemple, le connecteur RPC (*Remote Procedure Call*) est le résultat d'une composition séquentielle entre le connecteur type « *Appel procedure* » et le connecteur « *Distributeur* ». On peut dire que $RPC = PC + Distributeur$ (Taylor et al., 2009).

La distribution est une préoccupation centrale dans les applications modernes. Par conséquent, il est impératif d'avoir des connecteurs pouvant assurer un certain nombre de critères en matière de distribution. Parmi les compositions de distribution connues qui utilisent plusieurs types de connecteurs dont le connecteur « *Distributeur* », on trouve :

Connecteur Grid : Basé sur la technologie des grille de calcul, ce connecteur composé à partir des connecteurs de base suivants : « *appel de procédure* », « *accès aux données* », « *flux* » et « *distributeur* ». Il est invoqué via un appel de procédure nommé et synchrone avec les URLs décrivant où se trouvent les données et qui doit les recevoir. Il accède et manipule les données transitoires et persistantes. Les données sont empaquetées dans des flots, elles sont fournies à l'aide de l'environnement de grille et peuvent être livrées à plusieurs consommateurs.

Connecteur P2P : Ce connecteur est une composition des connecteurs de base suivants : « *arbitre* », « *accès aux données* », « *flux* », et « *distributeur* ». Il repose sur l'arbitrage pour la synchronisation et l'invocation. L'arbitrage implique la redirection du flux de contrôle entre des ressources distribuées, ou des pairs, fonctionnant dans un environnement en réseau. Il utilise les rendez-vous comme mécanisme pour atteindre la simultanéité et la

planification. Aussi, lors de l'invocation, les pairs s'engagent à accéder aux données transitoires et persistantes. Les données sont accessibles et conditionnées via des flux à l'aide d'un mécanisme limité au meilleur effort. Elles sont obtenues en localisant d'autres pairs avec les éléments de données souhaités. Le routage des données est principalement géré par un mécanisme de suivi.

***Connecteur Client/Serveur* :** Ce connecteur est une composition des connecteurs de base suivants : « *appel de procédure* », « *accès aux données* », « *flux* » et « *distributeur* ». Il est invoqué via des appels de procédure distants qui apparaissent au client comme des appels de procédure locaux avec des valeurs de retour possible pour les méthodes. Lors de l'invocation, les données sont accédées, transformées et empaquetées dans des flux. Elles sont ensuite retransmises à l'aide d'un registre de noms pour localiser le consommateur demandeur.

Connecteur basé événement : Il est composé des connecteurs de bases suivant : « *événement* », « *accès aux données* », « *flux* » et « *distributeur* ». Il sert à envoyer et recevoir des notifications asynchrones appelées événements. Les événements arrivent selon un calendrier périodique ou à des intervalles périodiques distincts. Les connecteurs de distribution basés sur des événements utilisent souvent une méthode de livraison asynchrone au « *best-effort*[1] » pour les données. Les connecteurs de distribution basés sur des événements peuvent accéder aux données et les muter à la fois chez le consommateur et chez le producteur. Les données sont généralement empaquetées dans des flux avec une taille de paquet limitée et une mise en mémoire tampon des données.

1.6.5 Propriétés fonctionnelles des connecteurs

Les propriétés fonctionnelles du connecteur sont des détails sémantiques associés aux éléments du système. Elles permettent à un architecte de raisonner sur certains aspects attendus et de supporter certaines analyses et ce à un niveau d'abstraction élevé. Il existe des propriétés relatives aux composants, connecteurs et configurations. Dans cette section nous nous limitons à celles relatives aux connecteurs. Dans (Hirsch et al., 1999), les auteurs donnent ce qu'ils appellent un tableau périodique pour les différents types de connecteurs et les propriétés fonctionnelles qui leur sont associés. Parmi les propriétés fonctionnelles usuelles, on cite :

- Connaissance : C'est au composant source de connaître explicitement le composant cible et non pas le connecteur.
- Demande/Réponse : Le connecteur garantit que pour chaque demande de service, exactement une réponse de service est reçue.
- Synchronicité : Pour qu'une communication ait lieu, la source et la cible doivent être disponibles.
- Flux de Contrôle : Le connecteur permet aux composants d'arrêter, de démarrer et de suspendre la communication.
- Sens de la communication : La communication unidirectionnelle ou multidirectionnelle doit être précisée. Différents connecteurs peuvent prendre l'un ou l'autre des cet aspect.
- Cardinalité: Le connecteur peut diffuser l'information pour plusieurs destinataires. Il peut également concentrer les informations émises de plusieurs émetteurs.

[1] La livraison « *best-effort* » décrit un service réseau dans lequel le réseau ne fournit aucune garantie que les données soient livrées ou que la livraison répond une qualité de service.

- Flux de données : Dans ce mode le connecteur doit assurer la continuité de la transmission jusqu'à sa terminaison.
- Contrôle et gestion : Les composants doivent gérer explicitement les sessions de communication.
- Fiabilité : Le connecteur ne doit jamais perdre l'information. Soit la cible reçoit les données transmises, soit la source est informée que la cible ne les a pas reçues.
- Cryptage : Le connecteur de chiffrement dispose de capacités de sécurité pour le chiffrement.
- Authentification : Le connecteur d'authentification a des capacités de sécurité pour l'authentification.
- Compression : Le connecteur peut compresser de manière transparente les données pour la transmission.
- Monitoring : Le connecteur peut reconnaître et transmettre certaines classes d'événements de communication à un composant de surveillance.
- Typage : Le connecteur typé nécessite que la source et la cible se mettent d'accord sur le type d'informations à transmettre.

Il en ressort que les connecteurs peuvent être plus ou moins complexes. Ils peuvent assurer une liaison point à point comme avoir une architecture interne. Ces éléments qualitatifs/quantitatifs sont souvent transcrits dans les différentes structures via des langages formels pour permettre des analyses concernant ces propriétés (Garlan, 2003) dans des phases précoces durant leur cycle de vie.

1.7 Les langages de descriptions d'architectures (ADLs)

Une préoccupation importante dans l'ingénierie des systèmes à base de composants est de trouver des notations appropriées pour décrire ces systèmes. Une bonne notation permet de documenter clairement les conceptions à base de composants, de raisonner sur leurs propriétés et d'automatiser leur analyse et la génération du système (Garlan et al., 2000).

L'une des approches pour décrire les systèmes à base de composants consiste à utiliser des notations de modélisation d'objets (Duncan, 2003). Le composant peut être représenté par une classe, les interfaces de composants par des interfaces de classe et les interactions entre composants peuvent être définies en termes d'associations.

La modélisation objet des systèmes à base de composants présente un certain nombre d'avantages : i) les notations d'objets sont familières à un nombre important de pratiquants logiciels. ii) elles fournissent une correspondance directe avec les implémentations. iii) elles sont aussi soutenues par des outils commerciaux et des méthodes bien définies pour développer des systèmes à partir d'un ensemble d'exigences. Toutefois, ces notations présentent un certain nombre d'inconvénients par rapport à la description de systèmes à base de composants : i) elles ne fournissent qu'une seule forme d'interconnexion primitive (invocation de méthode). ii) elles ont un faible support pour la description hiérarchique. iii) elles ne supportent pas la définition de familles de systèmes ou style. iv) elles ne fournissent pas de support direct pour caractériser et analyser les propriétés non fonctionnelles.

Pour surmonter ces problèmes, une approche alternative a vu le jour et elle consiste à utiliser un langage de description d'architecture (ADL). Nombre de langages ont été alors développés pour gérer la description de haut niveau de systèmes logiciels complexes. Ces langages exposent la structure brute d'un système comme une collection de composants en

interaction et permettant aux ingénieurs de raisonner sur leurs propriétés à un niveau d'abstraction élevé.

En bref, les ADLs sont considérés comme des DSLs (*Domain Specific Languages*) (Fowler, 2010) pour les architectures logicielles.

1.7.1 Description architecturale

Une question cruciale pour les architectes logiciels est de savoir comment décrire leurs conceptions architecturales. Idéalement, ces descriptions doivent transmettre clairement leur intention de conception aux autres, permettre une évaluation critique et nécessiter une faible surcharge pour la création et la maintenance.

Selon David Garlan (2014), il existe à ce jour trois approches générales pour la description d'architecture et qui se distinguent en fonction du langage de description utilisé. On trouve trois approches fondamentales : Les approches informelles, semi-formelles et complètement formelles.

1.7.1.1 Approche informelle

Cette descriptions utilise généralement des outils d'édition graphique à usage général (PowerPoint, Visio, etc.) associés à de la « prose » pour expliquer la signification des dessins. Cette approche est assez simpliste mais reste toujours utilisable.

La description informelle a l'avantage d'être facile à produire et de ne pas nécessiter d'expertise particulière. Mais elle a une foule d'inconvénients. La signification de la conception peut ne pas être claire car les conventions graphiques n'auront probablement pas une sémantique bien définie (Figure 1.21). Les descriptions informelles ne peuvent pas être formellement analysées en termes de cohérence, d'exhaustivité ou d'exactitude. Les contraintes architecturales supposées dans la conception initiale ne sont pas appliquées à mesure qu'un système évolue. Il existe peu d'outils pour aider les architectes dans leurs tâches.

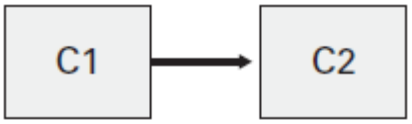


Figure 1.21-Notation informelle d'une description architecturale.

La Figure 1.21 montre un exemple de notation informelle pour une description architecturale. Elle est censée modéliser deux composants en interaction. Or, telle quelle se présente, cette figure est assez ambigüe et peut être sujette à différentes interprétations :

- C1 appel C2
- Flux de données de C1 vers C2
- C1 instancie C2
- C1 envoi un message à C2
- C1 est un sous type de C2
- C2 est un répertoire et C1 est y entrain d'écrire
- C1 est un répertoire et C2 est en train de lire les données …
-

Cette natation traduit ce qu'on appel l'approche de « boxologie » avec une sémantique floue. Au mieux, on peut définir uniquement les interfaces des différents modules

1.7.1.2 Approche semi-formelle

Cette approche utilise des notations de modélisation génériques qui peuvent manquer de sémantique détaillée, mais fournissent un vocabulaire graphique standardisé pris en charge par des outils commerciaux.

L'exemple principal d'un langage de description semi-formel est UML. Bien que certains types de diagrammes de la famille UML des notations de modélisation ont une sémantique formelle, mais pas les principaux diagrammes dédiés à la modélisation de la structure architecturale. En 2005, l'(OMG, 1989) a adopté UML 2.0 (Booch & Rumbaugh, 2000), incorporant des constructions explicites pour la modélisation architecturale, telles que des composants, des connecteurs, des ports et des descriptions hiérarchiques.

Un langage de modélisation à usage général, tel qu'UML, présente l'avantage d'utiliser des notations avec lesquelles les praticiens sont familiers, qui sont pris en charge par des outils commerciaux et qui fournissent un lien vers la modélisation et le développement orientés objet. Mais ces langages sont limités par leur manque de support pour l'analyse formelle et leur manque d'expressivité pour certains concepts architecturalement pertinents. L'exemple le plus important est celui des connecteurs qui ne sont pas des entités de première classe en UML comme le sont les composants (Clement et al., 2011).

1.7.1.3 Approche formelle

Comme alternative aux problèmes associés aux précédentes approches, les chercheurs ont proposé une troisième voie dite notation formelle qui permet de représenter et d'analyser les conceptions architecturales. Généralement appelées « langages de description d'architecture » (ADL), ces notations fournissent à la fois un cadre conceptuel et une syntaxe concrète pour la modélisation formelle des architectures logicielles. Elles fournissent également des outils pour analyser, afficher, compiler, analyser ou simuler des descriptions architecturales. C'est donc le rôle clé impartis aux ADLs qui peuvent être considérés des outils formels à partir de cette catégorie de notation.

Les vues des composants et des connecteurs représentent les unités d'exécution ainsi que les chemins et les protocoles de leur interaction (Clement et al., 2011). Un défi important pour les langages de description d'architecture est d'être capable de décrire les architectures logicielles statiques et dynamiques à partir des perspectives structurelle et comportementale du système (Haider et al., 2018).

1.7.2 Quelques ADLs

Nous allons essayer de présenter quelques langages de description d'architectures connus dans différents domaines et dans différentes communautés. L'accent sera toujours mis sur leurs principales caractéristiques et distinctions au fil du temps d'existence des ADLs en fonction de leurs capacités de décrire, analyser et produire un système. Nous proposons notre propre synthèse dans la section suivante ainsi que notre propre ADL dans le dernier chapitre.

1.7.2.1 Wright

Wright (Allen & Garlan, 1997) a été une des premières approches à permettre la description du comportement de composants. Il se centre sur la spécification de l'architecture et de ses éléments. Il n'y a pas de générateur de code, ni de plate-forme permettant de simuler l'application. Les trois notions de base d'un ADL à savoir le composant, le connecteur et la configuration sont présentes dans Wright.

Dans l'ADL Wright, un connecteur contient deux parties importantes qui sont un ensemble de rôles et la glu. Chaque rôle indique comment se comporte un composant qui

participe à l'interaction. Le comportement du rôle est décrit par une spécification en CSP. La glu décrit comment les participants (les rôles) interagissent entre eux pour former une interaction. Par exemple, la glu d'un connecteur « appel de procédure » indiquera que l'appelant doit initialiser l'appel et que l'appelé doit envoyer une réponse en retour. Enfin, Wright supporte l'évolution des connecteurs via le paramétrage; par exemple, le même connecteur peut être instancié avec une glu différente.

La configuration permet de décrire l'architecture d'un système en regroupant des instances de composants et des instances de connecteurs. La description d'une configuration est composée de trois parties qui sont la déclaration des types des composants et des connecteurs utilisés dans l'architecture, la déclaration des instances de composants et de connecteurs, les descriptions des liens entre les instances de composants et des connecteurs.

1.7.2.2 ACME

ACME (Garlan et al., 2000) est un langage de description d'architecture proposé par la communauté scientifique dans le domaine des architectures logicielles. Il a pour buts principaux de fournirun langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADLs, qui soit compatible avec leurs terminologies et qui propose un langage permettant d'intégrer facilement de nouveaux ADLs.

ACME prend en charge la définition de quatre aspects distincts de l'architecture. i) Structure : i) l'organisation d'un système en ses éléments constitutifs. ii) Propriétés d'intérêt : informations sur un système ou ses parties qui permettent de raisonner de manière abstraite sur le comportement global (à la fois fonctionnel et non fonctionnel). iii) Contraintes : lignes directrices sur la façon dont l'architecture peut changer au fil du temps. iv) Types et styles : définition des classes et familles d'architecture.

ACME propose trois concepts de base pour définir une architecture à savoir le *composant*, le *connecteur* et le *système* qui représente la configuration d'une application. Les points d'interface des connecteurs dans ACME sont appelés des « *rôles* », qui sont nommés et typés. Dans l'ADL ACME les types de connecteurs sont spécifiés par des protocoles d'interactions. ACME fournit des facilités de paramétrage qui permettent une spécification flexible des signatures de connecteurs et des contraintes sur la sémantique des connecteurs. Plusieurs ADLs emploient les mêmes mécanismes utilisés dans l'évolution des composants pour faire évoluer les connecteurs ; ACME définit le sous-typage structurel des connecteurs comme mécanisme pour leur évolution.

1.7.2.3 AADL

Architecture Analysis & Design Language (AADL) (Feiler et al., 2003) est un langage de description d'architectures pour les systèmes embarqués d'avioniques, sous l'autorité du SAE (Society of Automotive Engineers), organisme de standardisation mondiale dans le domaine du transport. Ce langage est issu d'un organisme de normalisation qui rencontre un très bon accueil auprès des industriels du domaine. De ce fait, il a subi de nombreuses extensions.

Dans AADL les composants peuvent avoir des sous-composants décrits dans leurs implémentations en utilisant des règles de composition. AADL utilise deux concepts pour désigner une configuration : i) les *systèmes* qui permettent de structurer l'architecture et contiennent des composants qui peuvent être manipulés de façon indépendantes. ii) les *composants abstraits* qui sont de simples conteneurs sans aucune sémantique et permettent aussi de structurer l'architecture.

Les interactions entre les composants sont matérialisées par la définition de connexions

entre leurs ports. Une connexion permet de relier deux ports. Les ports peuvent être déclarés en entrée (*in*), sortie (*out*), ou entrée-sortie (*in out*). Une vérification est faite qu'il y a bien conformité de type et de sens entre les ports connectés. Les connecteurs n'ont pas de comportement associé, mais il est possible de leur ajouter des propriétés sur le délai de transmission des données.

De par la nature même des composants, le modèle d'AADL est hiérarchique. Par exemple, un processus peut contenir des processus légers. En outre, la notion de *système* qui peut lui-même être composé de sous-systèmes permet de décomposer le système sur une infinité de niveaux.

1.7.2.4 UML 2.0

L'Object Management Groupe (OMG) à pris en charge la notion d'architecture dans sa nouvelle version UML 2.0 à partir de 2005 (UML, 2017). Elle propose deux diagrammes (diagramme de composant, diagramme de structure composite) pour prendre en charge les architectures à base de composants en décrivant le système sous forme de "schémas de câblage" d'unités ou parties autonomes du systèmes ayant des dépendances. Uncomposant représente une partie modulaire d'un système qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Il exhibe ses services via ses interfaces requises/fournies auxquelles sont associés des ports. La notion de connecteur est implicite dans la notation UML.

Dans le diagramme de composants, qui a pour principale avantage de donner une vue de haut niveau du système, on retrouve les principaux concepts liés aux architectures logicielles à base de composants en proposant généralement souvent deux approches: soit une notation visuelle précise soit une notation stéréotypée avec des flèches de dépendances.

Les entités de base dans la représentation visuelle tels que les composants, les ports ainsi que les connecteurs qui sont représentés implicitement avec les interfaces requises et fournies en forme de rotule avec une partie fixe et une partie articulaire ou sucette et prise. Tout en identifiant deux types de connecteurs : les connecteurs d'assemblage pour lier deux composants de même niveau d'hiérarchie et les connecteurs de délégation pour connecter un port externe au port d'un sous-composant interne, on parle alors de structure composite.

1.7.2.5 COSA

Component-Object based Software Architecture –(COSA) (Maillard et al., 2007), développé par l'équipe MODAL de l'Université de Nantes, est défini comme une approche de description d'architecture basée sur la description architecturale et la modélisation par objets. Dans COSA, les composants, les connecteurs et les configurations sont des classes qui peuvent être instanciés pour établir différentes architectures. Ils peuvent être réutilisés, redéfinis et évolués efficacement par des mécanismes opérationnels bien définis tels que l'instanciation, l'héritage ou la composition. Ces mécanismes sont inspirés des mécanismes du paradigme objet.

COSA confère aux connecteurs un niveau de granularité et de réutilisation semblable à celui des composants. Un connecteur est principalement défini par une interface et une glu. En principe, l'interface décrit les informations nécessaires du connecteur, y compris le nombre de rôles, le type de services fourni par le connecteur et le mode de connexion. Un rôle est soit de type requis ou de type fourni. Le nombre de rôles d'un connecteur représente le degré d'un type de connecteur. La glu décrit les fonctionnalités attendues d'un connecteur

primitif.

Dans COSA, il y a trois types d'associations qui relient les différents éléments architecturaux : *Attachment*, *Binding* et *Use*. Les *Attachments* permettent de relier les ports d'un composant ou d'une configuration aux rôles d'un connecteur. Un *Binding* fournit une association entre ports (respectivement rôles) internes et les ports (respectivement rôles) externes des composants composites (respectivement des connecteurs composites). L'association *Use* relie des services aux ports (ou aux rôles pour les connecteurs).

Dans COSA, une architecture peut être déployée de plusieurs manières, sans réécrire le programme de configuration/déploiement. Un autre avantage de COSA est la définition explicite des connecteurs et le fort support de leur réutilisation. Les fonctionnalités des connecteurs et des configurations sont exprimées par les services. Elles sont représentées au niveau de l'interface. Ceci a pour but de faciliter la recherche d'un connecteur ou d'une configuration dans une bibliothèque.

1.7.2.6 Phoenix

Phoenix (Auguston, 2009) est un d'ADL académiques de deuxième génération qui se concentre principalement sur la modélisation comportementale. C'est un ADL de spécification formelle dans lequel le comportement du système est défini comme un ensemble d'événements (traces d'événements) avec deux relations de base : la précédence et l'inclusion. La structure de la trace d'événements est spécifiée à l'aide de grammaires d'événements et d'autres contraintes organisées en schémas. Différents types de modèles (tels que alternatifs, facultatifs, etc.) sont définis sous la forme d'une trace d'événement qui se produit dans une transaction. Mais il leur manque la notation visuelle unique pour chacun de ces modèles d'événement. Un schéma est défini comme un ensemble de transactions qui inclut toutes les traces d'événements possibles.

Le framework schéma se prête au raffinement progressif de l'architecture jusqu'aux modèles de conception et de mise en œuvre exécutables, à la réutilisation, à la composition, à la visualisation et à l'application d'outils automatisés « Alloy Analyzer » (Kelsen & Ma, 2008) pour les contrôles de cohérence. A chaque étape de raffinement, l'architecture cible est validée formellement. C'est le seul ADL qui évoque le concept d'architecture « Exécutable ». Le langage utilisé est basé sur une grammaire d'événement avec un *schémas* d'une machine abstraite inspiré de Z *schema*. Dans ce type d'approche, le connecteur est supposé être explicite.

1.7.2.7 MontiArc

MontiArc (Haber et al., 2014) est un langage classique de modélisation des architectures des composants&Connecteurs dédié pour les systèmes Cyber-Physique (Ahmed et al., 2013; Giese et al., 2012). Les systèmes cyber-physiques sont intrinsèquement distribués et interagissent de diverses manières à l'aide de signaux, de messages et de données. MontiArc est développé pour la tâche de modélisation des systèmes interactifs distribués en capturant des éléments de leur distribution logique ou physique

Il inclut la décomposition hiérarchique des composants, le sous-typage par héritage structurel, les définitions de types de composants et les déclarations de référence pour la réutilisation, les types de composants génériques et les composants configurables, le sucre[1] syntaxique pour les connecteurs et la création implicite contrôlée de connexions et de

[1] Le sucre syntaxique exprime le fait de donner au programmeur des possibilités d'écriture plus succinctes ou plus intuitives.

déclarations de sous-composants.

MontiArc est implémenté en utilisant le framework DSL dédié MontiCore sous Eclipse. Les outils disponibles incluent un éditeur avec coloration syntaxique et complétion de code ainsi qu'un framework de simulation avec un générateur de code Java. MontiArc est étendu avec un cadre de simulation qui peut exécuter un comportement implémenté en Java et attaché de manière déclarative aux modèles MontiArc.

L'utilisation d'un ensemble de vues de fonctionnalités permet de tracer les exigences jusqu'à l'architecture du système logique pour lier conceptuellement les artefacts tout au long du processus de développement. Il utilise les composants, les composants hiérarchiques et une gamme de connecteurs explicites unidirectionnels utilisés en fonction du contexte.

1.7.2.8 Breeze

Breeze (Chen et al., 2015) considère l'ingénierie des besoins comme un défi principal dans l'ingénierie logicielle. Il s'appuie sur l'architecture logicielle comme un moyen pour la modélisation et l'analyse des besoins et particulièrement l'analyse des exigences non-fonctionnelles au niveau architectural. Breeze modélise, analyse et améliore l'architecture logicielle, en mettant l'accent sur ses exigences non fonctionnelles. En particulier, Breeze dispose de trois modules clés : (1) un module de modélisation (Breeze/ADL) qui facilite la modélisation des systèmes logiciels, (2) un module d'analyse (NFR) qui vérifie les exigences non fonctionnelles (par exemple, la sécurité, la fiabilité et l'exactitude) au niveau de l'architecture, et (3) un module de reconfiguration qui permet aux utilisateurs de réparer des défauts ou d'améliorer encore les architectures.

Les modules Breeze/ADL et NFR sont intégrés ensemble sous le framework Eclipse et sont par conséquent compatibles avec la norme UML. Breeze est open source et permet la modélisation de trois types d'analyse (la sécurité, la fiabilité et l'exactitude) des besoins. Breeze permet aux utilisateurs de réparer les défauts de leurs architectures conçues en définissant des règles de reconfiguration. Le mécanisme permet aux utilisateurs d'améliorer encore leurs architectures en fonction de leurs résultats d'analyse.

Breeze/ADL est un langage de description d'architecture basé sur XML, qui spécifie les propriétés du système logiciel en termes de composants, de connecteurs et de ports. Il est facilement extensible vers d'autres domaines puisque son XML-schéma peut être facilement étendu. Il dispose d'une interface graphique qui permet aux utilisateurs de définir des éléments de fonctionnalité d'architecture logicielle (par exemple, des composants, des connecteurs et des liens), via les boutons d'icône correspondants.

Pour chaque architecture conçue, Breeze génère un méta-modèle basé sur les spécifications de Breeze/ADL. Le méta-modèle, appelé « *TrustConfiguration* », se concentre sur les NFR et capture les données d'expérience et la connaissance du domaine des experts en exigences. La version actuelle de Breeze prend en charge les types analyses suivants :

- Analyse d'exactitude : Génération de spécifications de model-Checking via Breeze/ADL pour détecter les problèmes d'inter blocage ou d'incohérence, avec le support de NuSMV.
- Analyse de sécurité : Tisser des éléments de sécurité définis dans les exigences dans le modèle Breeze/ADL et identifier et hiérarchiser les événements de défaillance possibles avec l'analyse de l'arbre de défaillance, l'analyse du chemin minimum et l'analyse de l'ensemble de coupe.
- Analyse de fiabilité : Mappage du modèle Breeze/ADL à un modèle de chaîne de Markov à temps discret pour prédire la fiabilité de l'architecture.

Le dernier module de reconfiguration dynamique est réalisé en fonction d'un outil basé sur

une grammaire de graphe avec ses deux parties gauche (initiale) et droite (finale).

1.7.2.9 EAST-ADL

EAST-ADL (Blom et al., 2016) est un langage de description d'architecture (ADL) initialement défini dans plusieurs projets de recherche européens et aligné sur le standard AUTOSAR (Qureshi et al., 2011) et la norme de sécurité fonctionnelle ISO26262. Il fournit une approche globale pour définir les systèmes électroniques automobiles grâce à un modèle d'information qui capture les informations d'ingénierie sous une forme standardisée. Le langage fournit un large éventail d'entités de modélisation, y compris les caractéristiques du véhicule, les fonctions, les exigences, la variabilité, les fonctions d'analyse, les composants logiciels, les composants matériels et la communication.

EAST-ADL définit clairement plusieurs niveaux d'abstraction et à chacun de ces niveaux, les fonctionnalités logicielles et électroniques du véhicule sont modélisées avec un niveau de détail différent. Les niveaux d'abstraction proposés et les éléments de modélisation contenus fournissent une séparation des préoccupations et un style implicite pour l'utilisation des éléments de langage. Le système embarqué est complètement défini à chaque niveau d'abstraction, et des parties identiques du modèle sont liées entre les niveaux d'abstraction avec diverses relations de traçabilité. Cela permet de tracer une entité depuis la fonctionnalité jusqu'aux composants matériels et logiciels.

Les caractéristiques au niveau du véhicule représentent le contenu et les propriétés du véhicule dans un haut niveau sans exposer sa réalisation. Il est possible de gérer le contenu de chaque véhicule et des lignes de produits de manière systématique. Une représentation complète de la fonctionnalité électronique sous une forme abstraite est modélisée dans l'architecture d'analyse fonctionnelle (FAA). Une ou plusieurs entités (fonctions d'analyse) de la FAA peuvent être combinées et réutilisées pour réaliser des fonctionnalités. Le FAA capture les principales interfaces et le comportement des sous-systèmes du véhicule. Il permet la validation et la vérification du système intégré ou de ses sous-systèmes à un haut niveau d'abstraction. Des enjeux critiques pour la compréhension ou l'analyse peuvent ainsi être envisagés, sans le risque qu'ils soient masqués par des détails de mise en œuvre.

Cet ADL est destiné pour les systèmes embarqués, par conséquent, il considère les composants logiciels et les composants matériels d’une manière commune sur les différents niveaux d’abstraction. La notion de connecteur telle qu’elle est connue est absente dans cet ADL, il propose des mécanismes dédiés pour la communication. De ce fait, le connecteur est considéré comme implicite dans cet ADL.

Le langage EAST-ADL est formellement spécifié comme un domaine de capture de méta-modèle spécifique, c'est-à-dire les concepts automobiles. Le méta-modèle suit les lignes directrices provenant d'AUTOSAR pour la définition des modèles. Les concepts de modélisation sont représentés par les notions de base de MOF complétées par le modèle de profil AUTOSAR. Le méta-modèle s'inscrit ainsi comme une spécification d'un environnement d'outil spécifique au domaine, et définit également un format d'échange XML (appelé EAXML, similaire au format d'échange AUTOSAR ARXML). Ce modèle de domaine représente la définition réelle du langage EAST-ADL et constitue le cœur de la spécification du langage EAST-ADL.

1.7.2.10 πσADL

πσADL (Papapostolu, 2020) est un ADL particulier conçu pour prendre en charge les styles architecturaux orientés micro-services. La notion de micro-service (Giacomo et al., 2021) a considérablement gagné en popularité ces dernières années en raison de son application à des systèmes logiciels à grande échelle avec des résultats positifs. Cet ADL est conçu afin de fournir aux architectes logiciels et divers intervenants un moyen pratique de décrire les systèmes logiciels qui suivent le style architectural des micro-services. En prenant en

compte la particularité de la notion de micro-service comme entité architecturale, les constructions fournies par le langage concernent la description et la reconfiguration dynamique d'une architecture de micro-services.

L'objectif de cet ADL est de construire des applications comme un ensemble de services indépendants et aussi faiblement couplés que possible. Bien qu'elles présentent des similitudes avec les architectures orientées services (SOA), il existe des différences. Chaque micro-service possède son propre stockage persistant qui ne peut pas être modifié par d'autres micro-services.

L'idée de base développée par cet ADL est de translater la notion de service vers la notion de composant. Elle consiste à traduire les attributs des entités auxquelles sont attachés les micro-services. De cette façon, lors de la traduction, un connecteur et un composant sont générés et peuvent en outre être configurés. Le langage utilisé dans cet ADL est un langage dédié pour la structure et le π-calcul pour le comportement. Cette approche est essentiellement comportementale puisque nous sommes dans un contexte de vue dynamique pour le système.

Une caractéristique commune des architectures de micro-services est la nécessité d'une reconfiguration dynamique - c'est-à-dire le changement (prévu ou imprévu) de la topologie d'un système logiciel pendant l'exécution. Un processus de paiement par exemple peut changer à l'avenir. Une nouvelle instance du micro-service reconfiguré doit donc remplacer l'ancienne. Les constructions de langage « attach » et « detach » permettent de décrire facilement un tel changement au niveau de l'instance de l'architecture donnée.

1.8 Synthèse

A l'instar des différentes études et enquêtes (Amirat, 2010; Fujita & Herrera-Viedma, 2018; Haider et al., 2018; Medvidovic & Taylor, 2000) où chacune considère les langages de descriptions d'architecture sous une vue différente, nous présentons aussi notre propre synthèse sur les quelques ADLs que nous venons d'étudier.

Nous avons essayé de suivre l'évolution des ADLs dans le temps des plus anciens aux plus récents. Nous avons essayé aussi de proposer des catégories de langages différentes pour montrer leurs champs d'application. Vu que notre sujet de thèse est centré sur les connecteurs, nous avons insisté sur la manière dont ces ADLs traitent et prennent en charge le concept de connecteur ; qu'il soit explicite ou implicite. Aussi, nous avons insisté sur les langages utilisés par ces ADLs pour décrire les architectures logicielles. Un intérêt particulier à été accordé aux différents types d'analyses qu'ils peuvent assurer tout en insistant sur la vue considérée par ces ADLs en référence avec la norme ANSI/IEEE 1471-2000.

Les ADLs les plus récurrents dans la littérature sont ACME pour la communauté académique et AADL pour la communauté industrielle. Largement admis, ils ont subi de nombreuses extensions depuis leurs créations à ce jour.

ACME est l'ADL le plus cité dans la littérature dans la gamme des ADLs académiques à usage général. Aussi, c'est le langage qui à subit un nombre important d'extensions dans la communauté scientifique. Ces extensions ont pour but de prendre en charge les nouveautés dans des domaines spécifiques. Parmi ces extensions on peut citer : CloudADL (Cavalcante et al., 2013) pour prendre en charge les spécificités relatives aux applications orientées Cloud et qui s'appuient sur la notion de « *Service* » comme élément architectural principal. Pl-AspectualACME (Barbosa et al., 2011) est une extension d'ACME pour prendre en charge la variabilité et les lignes de produit en se basant sur le paradigme orienté aspect (AOP). LightWeightACME (E. Silva et al., 2013) est une extension ACME pour les lignes de produits.

Tableau 1.3- Synthèse sur les ADLs étudiés

ADL	Catégorie	Connecteur	Langage description	Analyse	Vue
Wright 1997	Généraliste	Explicite	Textuel ACME + CSP	Inter-blocage	Comportement
ACME 2000	Académique Généraliste	Explicite	Textuel Langage ACME +CSP	Plug-in outil analyse	Structure Comportement
AADL 2003	Industriel avionique Standard	Implicite	Visuel + Textuel dédié	Système critique Zéro Erreur	Structure Comportement
UML 2.0 2005	Standard	Implicite	Visuel Composant/ Structure composite	Aucune	Structure
COSA 2007	Académique	Explicite	Visuel	Aucune	Structure
Phoenix 2009	Académique Formel	Explicite	Grammaire événement Z Schema	Cohérence	Comportement
MontiArc 2014	Académique Cyber-Physic	Explicite 5 types	Textuel DSL dédié	Comptabilité Interface + Vivacité + Simulation	Structure / Comportement
Breeze 2015	Académique Ing. Besoins	Explicite	Visuel/Textuel Grammaire Graph	Exactitude Fiabilité Securité .. Scénarios	Structure Comportement
East-ADL 2016	Industriel automobile Standard	Implicite	Visuel (UML profile) Textuel (XML)	Analyse fonctionnelle. V&V. Sureté	Structure Comportement
πσADL 2019	Académique MicroService	Explicite Bus message	Textuel Langage dédié π-Calcul	Reconfiguration dynamique	Comportement

Dans son enquête concernant les connecteurs sur 142 ADLs relativement anciens, Mert Ozkaya (Fujita & Herrera-Viedma, 2018) stipule que la majorité des ADLs implémentent les différents types de connecteurs (Section 1.6.3) à l'exception du connecteur « *Distributeur* ». Quant aux langages formels pour modéliser le comportement, la tendance statistique (non motivée) est au π-Calcul au lieu de l'initial CSP.

Dans (Papapostolu, 2020), les auteurs parlent d'un manque d'engouement envers les ADLs dans l'industrie du logiciel ces dernières années. Ils avancent pour cela deux arguments : i) la complexité et les formalités rencontrées dans de nombreux ADLs constituent l'une des principales raisons pour lesquelles ils ne sont pas largement adoptés dans les processus de développement de logiciels industriels, mais principalement dans les universités (Malavolta et al., 2012). ii) le fait qu'aujourd'hui la tendance générale des architectes est d'utiliser des langages architecturaux décrivant les architectures orientées service (Bouguettaya et al., 2017; Di Francesco, 2017).

Dans la liste que nous avons pris en compte, nous estimons l'ADL « Breeze » comme le

plus abouti. Bien que destiné à des fins académiques, il est le seul à même d'assurer le tout ou presque des fonctionnalités attendues d'un langage de description d'architecture. Fort de son aspect « open source » sous Eclipse et de ses assises formelles, il assure les trois phases : Modélisation, Analyse et Reconfiguration. Le reste des ADLs est sélectif quant à ces propriétés.

Volontairement nous n'avons pas voulu introduire dans cette liste les tendances technologique (Giacomo et al., 2021) souvent traduites par la concurrence entre les grands constructeurs comme les EJB (Sun, 1997) ou .Net (Microsoft, 2003). La raison est que ces technologies donnent des solutions et des implémentations pratiques de bas niveau sans aucune précision sur les niveaux l'abstraction des architectures logicielles.

1.9 Conclusion

Dans ce chapitre nous avons essayé de présenter dans le détail le concept d'architecture logicielle et les éléments et la philosophie qui les entourent. L'analogie avec l'architecture en génie civil à été a chaque fois évoquée en introduisant chaque nouveau concept. Nous avons évoqué les différentes définitions et points de vue, les différentes normes associées au développement orienté composant. Nous avons effectué un comparatif des architectures logicielles dans les différents paradigmes de développement. Les éléments architecturaux, les styles ainsi que certaines propriétés qualitatives ont été détaillées. Nous avons essayé de donner à chaque fois les motivations et justifications nécessaires.

Comme notre sujet de thèse s'articule sur la partie communication ou interaction dans les architectures logicielles, nous avons estimé nécessaire de donner un maximum d'information sur l'élément architectural dédié à l'interaction dans le développent à base de composants. Nous avons essayé de donner les détails et les frontières relatifs au concept de connecteur. Nous avons insisté sur la façon dont le connecteur est vu dans la communauté architecture logicielle en faisant appels aux différents travaux des écoles les plus connues dans le domaine. Nous donnerons une autre vue de ces mêmes concepts par rapport à une autre communauté dans le chapitre suivant.

La dernière section à été consacrée à l'étude des langages de descriptions d'architecture et leur évolution dans le temps. Nous avons considéré une dizaine de langages tout en focalisant à chaque fois sur la manière avec laquelle ils prennent en charge le concept de connecteur. Nous avons essayé de présenter quelques études connexes. Nous avons également dressé un comparatif et une synthèse à la fin. Dans le chapitre 4, nous reviendrons pour présenter une liste d'approches plus détaillées qu'elles soient outillées ou non mais qui considèrent toutes le connecteur comme une entité de première classe.

CHAPITRE

2 Description des Processus Logiciels

2.1 Introduction

Nous avons vu dans le premier chapitre que la naissance –même subtile- des architectures logicielles revient à la fameuse conférence de l'OTAN des années soixante-dix pour étudier et chercher à maîtriser la complexité sans cesse croissante du logiciel. On peut considérer aussi cette date comme la naissance du concept du processus logiciel à travers l'émergence d'une nouvelle discipline : le génie logiciel. En effet, dès lors que logiciel y a obtenu le statut de produit industriel ; alors il va en obtenir toutes les propriétés associées.

Pour produire une voiture, nous avons besoin d'une usine, d'ingénieurs, de méthodes, de techniques, d'outils et machines et enfin de savoir faire que l'on capitalise au fil du temps. C'est exactement le même sort qui est désormais réservé au logiciel. En préservant bien entendu certaines spécificités inhérentes au logiciel par rapport à tout autre produit industriel. C'est le caractère inusable et intangible du logiciel qui fait la richesse des grands constructeurs ou éditeurs logiciel de nos jours.

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

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

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

Mais pourquoi nous intéressons-nous au concept de processus ou procédé logiciel dans ce chapitre ? Bien que notre sujet de thèse soit réservé aux architectures logicielles, qui sont finalement le produit d'une phase (conception) du processus logiciel. Nous discutons de la modélisation de processus logiciels car c'est le cœur même de notre proposition et

contribution principale. En fait, nous suggérons de modéliser la communication dans les architectures logicielles à travers un processus logiciel.

Pour cette raison nous allons présenter dans le cadre de ce chapitre tout ce qui a trait au processus logiciels. Nous présentons aussi une section pour rapprocher les concepts entre les processus logiciels et les architectures logicielles à base de composants. Aussi, une section sera dédiée aux langages de modélisation des processus logiciels et enfin nous allons présenter le standard de modélisation SPEM est les mécanismes opérationnels qui l'entourent.

2.2 Processus Logiciel

Dans sa forme basique, «*un processus est une Suite continue d'opérations, d'actions constituant la manière de faire, de fabriquer quelque chose* » (Rey, 2018). L'origine des processus logiciels ou des modèles de processus logiciels revient certainement à l'émergence de la discipline du génie logiciel suite à la conférence de l'OTAN (Naur, 1968) des années 70s consacrée à la première crise du logiciel.

Il est utile de rappeler que cette appellation n'est pas propre au domaine informatique en particulier mais au domaine industriel en général. La notion de procédé existait bien avant la naissance du génie logiciel.

2.2.1 Procédés industriels

Si les processus logiciels et les processus métiers sont des concepts en relation directe avec la technologie de l'information, les procédés industriels quant à eux sont très anciens. Ils sont en relation avec les tout premiers produits de la révolution industrielle. Egalement, un procédé peut être artisanal ou manuel comme la poterie ou le tissage.

A titre d'exemple, pour fabriquer une pièce mécanique on doit suivre un ensemble d'étapes suivant une certaine démarche et une certaine organisation en utilisant un ensemble de moyens et de techniques. Pour produire en série, on doit réitérer la même démarche pour toutes les pièces similaires. Ce procédé et sa commande peuvent être manuels ou automatiques par le biais de machines dédiées.

Au fil du temps, on a également vu naître une multitude de langages pour normaliser cette activité et capturer ses différents aspects. Par conséquent, et comme le logiciel est un produit industriel, sa production a hérité des mêmes qualificatifs de l'industrie en générale. Le meilleur exemple de ces langages est sans doute les Réseaux de Petri (RdP) (Murata, 1989). Les Réseaux de Petri est un langage légendaire et outil formel initialement conçu pour modéliser les systèmes et les tâches industriels. Ayant fait ses preuves dans ce monde, il est de facto intégré dans les processus logiciels comme outil de spécification et analyse formel. Un descendant direct des RdPs, qui lui est resté fidèle au monde industriel, est le Grafcet (Menasria & Djebali, 1992) de Télémécanique (Telemecanique, n.d.).

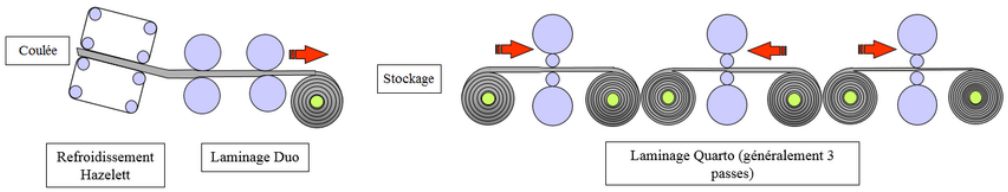


Figure 2.1 - Procédé industriel de laminage de tôles.

La Figure 2.1 donne un exemple de procédé industriel de laminage. À partir d'une pièce plate, on obtient un rouleau de lame métallique après une série de passes d'amincissement (laminage) et d'enroulage. On obtient alors des tôles de plusieurs épaisseurs comme sont

disponibles sur le marché.

Le concept de procédé industriel (productique) a évolué dans les temps. Avec l'avènement d'Internet et la naissance des capacités de distribution, on a vu naître une nouvelle discipline appelée les systèmes cyber-physiques (CPS) (Ahmed et al., 2013) pour décrire ces mêmes thématiques.

2.2.2 Quelques définitions

Il existe plusieurs définitions pour les processus logiciels dans la littérature, nous citons celles que nous jugeons assez significatives :

Définition 1 : *« Un processus logiciel est l'ensemble des étapes de processus partiellement ordonnées, avec des ensembles d'artefacts associés, des ressources humaines et informatisées, des structures organisationnelles et des contraintes, destinés à produire et à maintenir les livrables logiciels demandés »* (Lonchamp, 1993)

Définition 2 : *« Un processus logiciel peut être défini comme l'ensemble cohérent de politiques, de structures organisationnelles, de technologies, de procédures et d'artefacts nécessaires pour concevoir, développer, déployer et maintenir un produit logiciel »* (Fuggeffa, 2000).

Définition 3 : *« Un processus logiciel est défini comme une séquence d'étapes qui doivent être exécutées par les agents humains pour poursuivre les objectifs du génie logiciel »*(Zamli, 2001).

Définition 4 : *".Un processus logiciel désigne l'ensemble des activités nécessaires à la réalisation d'un système logiciel, exécutées par un groupe de personnes organisé selon une structure organisationnelle donnée et comptant sur le support d'outils techno-conceptuels"* (ACUÑA & FERRÉ, 2005).

Définition 5 : *"Le processus logiciel est un ensemble partiellement ordonné d'activités entreprises pour gérer, développer et maintenir des systèmes logiciels..."*(ACUÑA & FERRÉ, 2005).

Définition 6 : *" On décrit les procédés logiciels comme étant une suite d'étapes réalisées dans un but donné qui servent à gérer et assister le développement logiciel..."* (GALLARDO, 2000).

Définition 7 : « *Essentiellement, la modélisation des processus logiciels est la construction d'une description abstraite des activités par lesquelles le logiciel est développé. Dans le domaine des environnements de développement logiciel, l'accent est mis sur les modèles qui sont exécutables, c'est-à-dire interprétables ou se prêtant à un raisonnement automatisé* » (Finkelstein et al., 2019).

Il est évident de remarquer que toutes ces définitions partagent les concepts, de tâches, d'ordres, de ressources, de livrables et d'organisation. Ces éléments seront certainement des entités clés quand on définit les langages utilisés pour modéliser les processus logiciels.

2.2.3 Procédé vs Processus vs modèle de processus logiciel

Un processus correspond à une activité réalisée, alors que la procédure explique comment réaliser cette activité. Loin de ce jeu de mot, on peut dire que les termes « processus » et « procédé » ont le même sens et désignent la même chose dans l'industrie logicielle. En réalité, dans le domaine des processus logiciels il n'y a pas de différence avérée entre les termes "Processus" et "Procédé", et leur traduction en anglais est la même : "*Process*".

Plus concrètement et plus sommairement, les processus logiciels sont les activités de conception, de mise en œuvre et de test d'un système logiciel. Le processus de développement logiciel est compliqué et implique bien plus que des connaissances techniques. C'est là que les modèles de processus logiciels sont utiles. Un modèle de processus logiciel est une représentation abstraite du processus de développement.

Un modèle de processus logiciel est donc une abstraction du processus de développement logiciel. Les modèles spécifient les étapes et leurs ordres dans le processus. C'est une représentation de l'ordre des activités du processus et de la manière et de la séquence dans laquelle elles sont exécutées ainsi que les itérations et les sélections.

Outre les tâches à effectuer, un modèle de processus logiciel doit définir l'entrée et la sortie de chaque tâche, ses pré et post conditions et enfin le déroulement et la séquence de chaque tâche. Les séquences de tâches sont des enchaînements d'actions qui ne sont pas forcément linéaires ; l'itération, le parallélisme et l'ordonnancement partiel des actions sont admis.

L'objectif d'un modèle de processus logiciel est de fournir des orientations pour contrôler et coordonner les tâches afin d'atteindre le produit final attendu et les objectifs fixés dès le départ aussi efficacement que possible.

2.2.4 Cycle de vie et processus logiciel

Un modèle de cycle de vie du logiciel montre les phases principales et le livrable principal, tandis qu'un modèle de processus décrit les tâches de bas niveau, les artefacts nécessaires et produits, et les acteurs responsables de chaque tâche de bas niveau. Le concept de cycle de vie est antérieur au concept de processus logiciel. On peut dire que le cycle de vie est plus fonctionnel alors que le processus logiciel est plus technologique et de granularité plus fine. Aussi, on peut considérer le modèle de cycle de vie comme étant un modèle descriptif alors que le modèle de processus peut lui être exécutable.

La force des modèles de procédés logiciels réside dans leur richesse sémantique et syntaxique qui leur confère la possibilité d'être exécutable (comme le logiciel, un processus peut être exécutable) (Osterweil, 2011), contrairement aux modèles de cycle de vie de logiciels qui sont considérés comme des modèles contemplatifs seulement.

Le modèle de procédé logiciel est aussi une représentation des séquences et des étapes à suivre pour réaliser un logiciel, cependant, la représentation est de fine granularité. En d'autres termes, le modèle de procédé logiciel permet de décrire de manière plus détaillée les tâches, les outils et les ressources utilisés lors du développement logiciel (Stoica et al., 2013). La description des modèles de processus doit se faire dans un langage dédié.

2.2.4.1 Notion de cycle de vie

Selon (Fuggeffa, 2000), un cycle de vie logiciel définit les différentes étapes dans la vie d'un produit logiciel. Il s'agit généralement du développement, de l'analyse et de la spécification des exigences, de la conception, de la vérification et de la validation, du déploiement, de l'exploitation, de la maintenance et du retrait. De plus, un cycle de vie logiciel définit les principes et les lignes directrices selon lesquels ces différentes étapes doivent être réalisées.

Cette définition résume le cycle de vie d'un logiciel en deux aspects : i) des étapes ii) des démarches et ressources. Plus clairement on peut dire que le cycle de vie du logiciel est toute forme de relation avec ce logiciel depuis l'idée de sa création jusqu'à son abondant.

2.2.4.2 Notion de modèle de cycle de vie

Le modèle de cycle de vie est la traduction du cycle de vie du logiciel sous une forme plus ou moins formalisée dans une démarche précise. Le cycle de vie étant un ensemble de phases, le rôle du modèle de cycle de vie est la manière avec laquelle on va appréhender ces phases dans le temps et dans l'espace. Il consiste à décrire un modèle abstrait qui décrit particulièrement l'enchaînement temporel des différentes phases et leur ordonnancement.

Généralement, on peut classer les modèles de cycle de vie dans trois familles : i) les modèles en tubes comme le modèle en cascade ou en « V » ou le logiciel n'est obtenu qu'à l'issu de la dernière phase, ii) les modèles itératifs comme le modèle en spiral, incrémentale et itératif où l'on dispose de versions intermédiaires ou prototypes de l'application à chaque itération, iii) les modèles agiles avec une intervention soutenue du client dans le cycle de développement.

2.2.5 Modèle de cycle de vie

Les phases de cycle de vie obéissent à l'effort de standardisation ISO/IEC 12207. D'une manière exhaustive, les principales phases dans une organisation de projet de développement de logiciel sont :

- Recueil/Analyse des besoins : Étude informelle des fonctionnalités (externes) du système sans considération technique (Point de vue métier / utilisateur).
- Spécification : Que doit faire le système côté client ?. Identifier les éléments intervenants (acteurs) hors et dans le système: fonctionnalités, structures et relations, états par lesquels ils passent suivant certains événements. (Premier modèle du système).
- Conception : Comment réaliser le système, les choix technique. Choix et production d'une architecture technique (matérielle, logicielle) suivant certaines contraintes (robustesse, efficacité, performances, portabilité …).
- Implémentation : Phase en lien direct avec les langages, les outils les environnements de développement. Nécessite un savoir de la réutilisabilité des composants, voir d'outils de génération de code
- Test : Vérification et validation du logiciel.
- Déploiement/Maintenance : maintenance corrective et évolutive après déploiement.

A titre d'exemple, le modèle en cascade suggère qu'une phase spécifique ne doit être démarrée que lorsque les livrables de la précédente ont été achevés. À l'inverse, le modèle en spirale considère le développement de logiciels comme l'itération systématique d'un certain nombre d'activités guidées par l'analyse des risques.

Les modèles de cycle de vie (Adel & Abdullah, 2015) sont des représentations souvent graphiques (diagramme, schéma...etc.) traduisant ces phases dans des ordres et une organisation différente comme suit :

2.2.5.1 Modèle en cascade

Le modèle en cascade (ou chute d'eau) est le premier modèle initié dans le génie logiciel. Il traduit une démarche purement séquentielle des différentes phases. Chaque étape du cycle de vie est caractérisée par des activités dont le but est d'élaborer un ou plusieurs produits intermédiaires. Chaque étape utilise le produit de la phase immédiatement précédente et fournit un produit de la phase suivante.

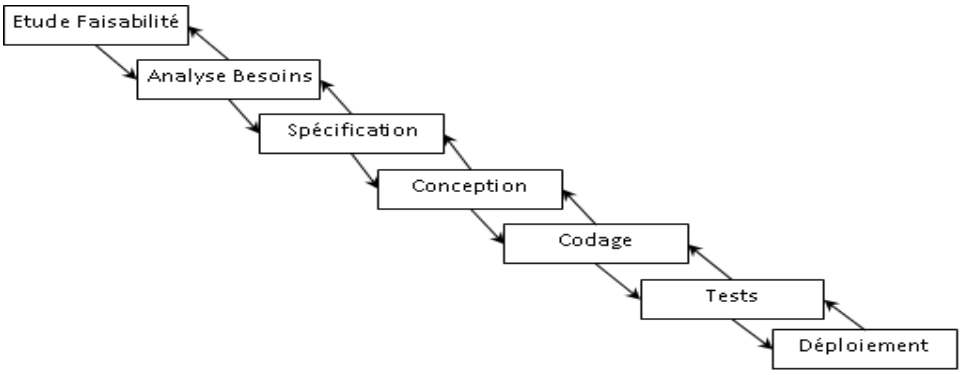


Figure 2.2- Modèle en Cascade

La Figure 2.2 montre cette séquence de phase. Une phase peut débuter avant la fin totale de la précédente. Les flèches de retour indiquent les corrections des erreurs qui exigent souvent de revenir en arrière ou aux étapes précédentes. Le coût de correction est d'autant plus élevé que l'erreur est décelée tardivement dans le processus de développement.

2.2.5.2 Modèle en Spiral

Le modèle en spirale fait partie des modèles itératifs. Il est principalement orienté pour l'évaluation et la gestion des risques. Les activités du projet commencent par le spiral le plus profond. Chaque tour passe par les régions tâches. L'accomplissement des phases du projet est le résultat de l'application des tâches prescrites par les régions. Pour chaque tour du modèle, il en résulte un prototype. De ce fait, on obtient un raffinement du produit en parcourant plusieurs tours du modèle.

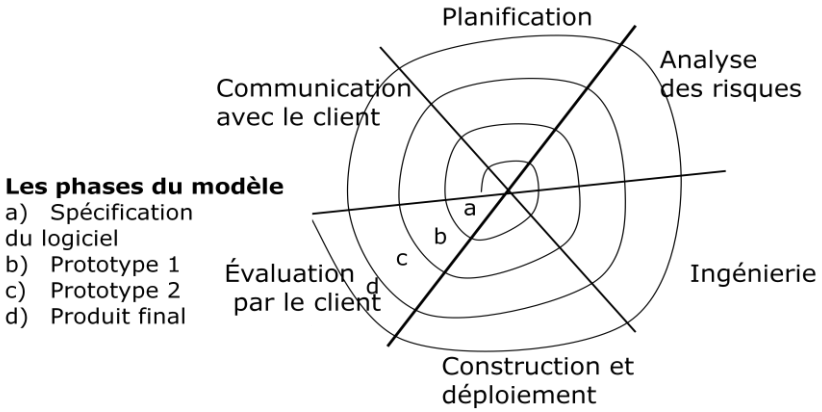


Figure 2.3- Modèle en spiral

La Figure 2.3 montre le modèle en spiral qui fait montrer aussi la notion de régions réaliser pour chaque tour de la spire. On réalisera les autres tours du modèle de la même façon et on profitera de la région « planification » pour réajuster le plan du projet

Le premier tour résulte de la spécification du logiciel, les tâches à réaliser pour cette spécification sont: Communiquer avec le client pour obtenir les spécifications, planifier le projet et les étapes de l'analyse des spécifications, évaluer les risques technologiques des techniques d'analyse, décider sur les techniques à utiliser pour l'analyse des spécifications, réalisation proprement dite de l'analyse des spécifications et enfin présenter les résultats au client

2.2.5.3 Le modèle agile

Le modèle de processus agile (P. Kruchten, 2013) encourage les itérations continues de développement et de test. Chaque pièce incrémentielle est développée sur une itération, et chaque itération est conçue pour être petite et gérable afin qu'elle puisse être complétée en quelques semaines. Chaque itération se concentre sur la mise en œuvre complète d'un petit ensemble de fonctionnalités. Il implique les clients dans le processus de développement et minimise la documentation en utilisant une communication informelle.

Bien que l'agilité offre une approche très réaliste du développement logiciel, elle n'est pas idéale pour les projets complexes. Cela peut également présenter des défis lors des transferts car il y a très peu de documentation. Le développement agile est idéal pour les projets avec des exigences changeantes.

La principale motivation des modèles agiles est l'intégration orientée service (Stoica et al., 2013). Le modèle agile utilise une approche adaptative où il n'y a pas de planification et seules des tâches futures claires sont celles liées aux caractéristiques qui doivent être développées. L'équipe s'adapte aux changements dynamiques des exigences du produit. Le produit est fréquemment testé, ce qui minimise le risque de défauts majeurs à l'avenir. L'interaction avec les clients est le point fort de l'agilité alors que le modèle agile est mieux adapté aux petits et moyens projets.

2.2.6 Processus logiciel vs Processus métier

Souvent, il y a une confusion entre ces deux concepts ou une incompréhension quant au rapport de l'un avec l'autre. Il existe aussi des similitudes et des différences. Il s'avers aussi que ces deux concepts sont assez proches et nécessitent donc une petite investigation pour déterminer les ressemblances et les distinctions. De premier abord, on peut d'ores et déjà dire que les processus métiers ont un rapport humain alors que les processus logiciels ont un rapport avec la machine.

Dans les années 2000, un standard (Wellington & Smith, 1995) a mis en avant que les outils logiciels peuvent être utilisés pour automatiser les procédures de travail impliquant la circulation de documents papier. La gestion des *flux* de *travaux* (*workflow*) est la technologie logicielle inspirée de cette idée. Elle permet de modéliser les procédures de travail et d'en assurer la mise en œuvre. Il est utile de rappeler que cette date est une date de standardisation, l'idée initiale a germé beaucoup plutôt.

On assiste alors aux développements de systèmes dédiés à la gestion des documents dans des organisations caractérisées par des procédés métiers de type traitements de dossier : « *Pour le calcul de salaire d'un employé, on doit avoir toutes les informations constantes et variables qui le concernent. Il doit avoir un salaire de base et une liste d'indemnités qui peuvent être cotisables et/ou imposables et enfin des allocations familiales. Chaque mois, en fonction des absences, des maladies ou des changements familiaux, on calcule le salaire en faisant la somme du salaire de base, des indemnités et des allocations pour avoir un salaire brut. De ce salaire, on défalque les impôts, les cotisations sociales et les retenues sur salaire pour obtenir le salaire net* ».

Le processus ou le scénario, précédemment cité, est fait indépendamment de tout support technologique. Quant à la démarche adoptée par la communauté procédée logiciel, elle est plus orientée vers l'étude de modèles pour les procédés que dans la production de systèmes directement exploitables. Les aspects technologiques étudiés se focalisent sur l'élaboration d'environnements destinés à supporter les modèles développés. Donc le rôle du procédé logiciel est de produire un support automatique du processus métier de calcul de

salaire en utilisant un modèle de processus en cascade (Figure 2.2) par exemple.

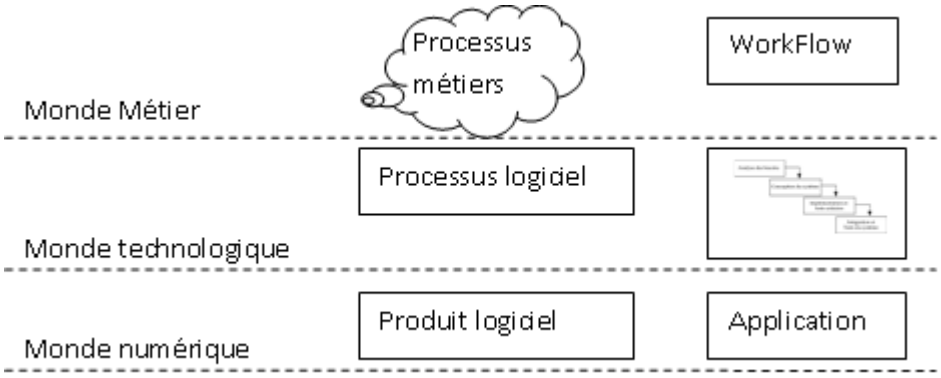


Figure 2.4- Processus logiciel et Processus métier

La Figure 2.4 essaye de faire la liaison et les distinctions entre les deux mondes celui des logiciels et celui de métiers. Toutefois, il est utile de remarquer l'existence d'une similitude très subtile entre les deux : il s'avère que le processus logiciel est lui-même un métier humain qui a cette particularité de ne pas produire le calcul de la paie mais de produire le logiciel pour le calcul de la paie. De ce fait, il est légitime d'utiliser les processus métier pour décrire aussi les processus logiciels.

2.2.7 Eléments de langages des modèles de processus logiciels

Comme il a été défini plutôt, les processus logiciels ne sont autres que des tâches ou activités associés avec un certain ordre, utilisant un certain nombre de ressources pour produire des livrables logiciels sous la responsabilité d'une organisation bien précise. La Figure 2.5 traduit, tant bien que mal, ces différents aspects. Au-delà du cycle de vie, cette figure précise l'ensemble des intervenants dans ce processus et leurs relations.

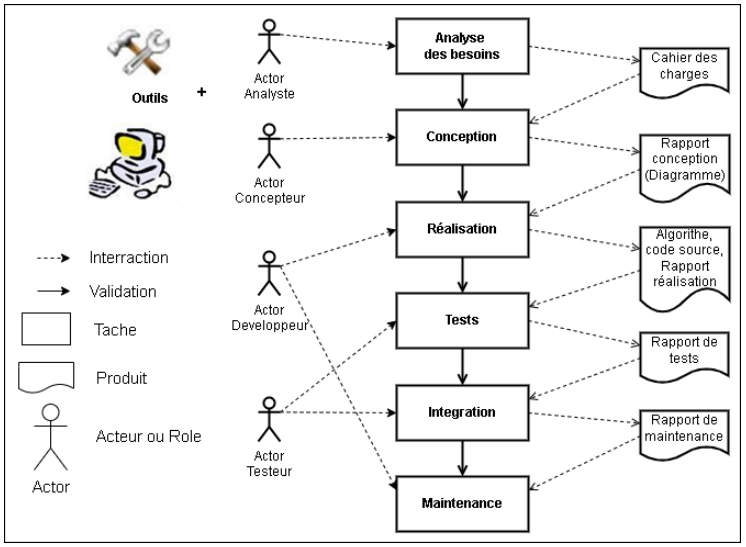


Figure 2.5- Instance d'éléments et environnement de processus logiciels.

Pour normaliser de tels aspects, le consortium (OMG) a fait un effort pour aboutir à un langage qui définit formellement les éléments du langage de modélisation des processus logiciels. La standardisation va permettre de parler ou d'utiliser le même langage par une large communauté et, par la même, l'émergence et la prolifération d'outils dédiés. Il est à rappeler que l'objectif ultime du génie logiciel en général et les processus logiciels en

particulier est d'automatiser la production ou l'industrie du logiciel (Brian Randell, 2018). A ce titre, la *représentation*, *l'exécution*, la *réutilisation* et *l'analyse* des processus logiciels sont des fins en elles-même.

Partant de l'idée qu'UML, tel qu'il est défini même à partir de sa version 2.0, ne permet pas de décrire les processus logiciels car il n'a aucun moyen pour le faire. UML est orienté produit et non pas processus. Dans ce genre de situation, on fait souvent recours à la notion de Profil c'est-à-dire on fait faire à UML ce qu'il ne fait pas habituellement. L'effort de normalisation a abouti en 2008 et 2011 à un langage en bonne et dû forme pour la prise en charge des éléments des processus logiciels.

À partir de là, on a donc vu naître la notion de profil ou méta-modèle SPEM (*Software & System Process Engineering Méta-model*) (OMG-SPEM, 2008). Au même titre qu'UML, SPEM est un langage et non pas une méthode qui décrit une démarche d'utilisation. Il offre les éléments qui permettent de décrire le processus logiciel selon plusieurs points de vue. Pour cela, il intègre sept packages distincts utilisables en tout ou partie dans une opération de modélisation. Les packages de SPEM sont ce que sont les différents diagrammes pour UML.

La Figure 2.6 donne une portion du méta-modèle (SPEM 1.1) du package « Processus ». On y trouve l'essentiel comme : (i) «*WorkDefinition* » qui précise les activités, les étapes, les phases, les itérations et le cycle de vie (ii) « *WordProduct* » qui précise par exemple les rapports et les codes (iii) « *ProcessPerformer* » qui précise les intervenants et les outils dans le cycle. Plus loin (Section 2.5), nous consacrons une section complète pour présenter les aspects importants de la nouvelle version du méta-modèle (SPEM 2.0).

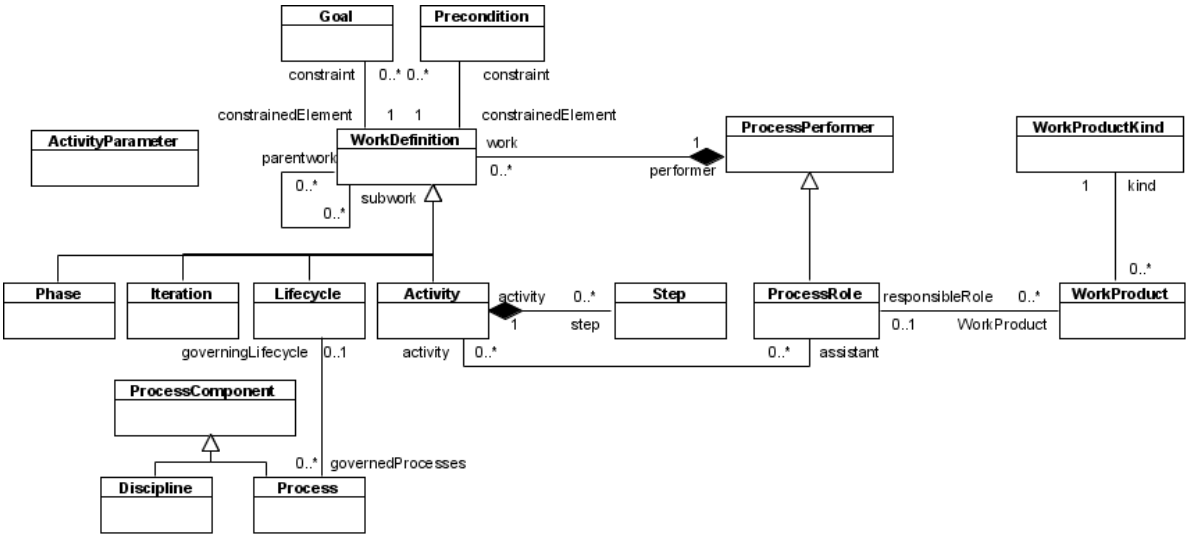


Figure 2.6- Méta-modèle de processus (SPEM 1.1).

2.3 Avantages et objectifs des processus logiciels

De nos jours, l'activité de nombreuses entreprises et organisations repose essentiellement sur les logiciels. Les logiciels ajoutent souvent une valeur significative à de nombreux produits et services et permettent une différenciation concurrentielle sur le marché. L'importance croissante des logiciels ainsi que les nouveaux paradigmes de développement de logiciels imposent de nombreux défis et exigences en matière de développement, d'exploitation et de maintenance de ces mêmes logiciels.

Les processus logiciels s'insèrent comme une réponse directe aux précédentes préoccupations. La qualité du logiciel dépend certainement de la qualité du processus qui l'a

produit. Ainsi, s'établit le rapport étroit entre ces deux concepts. En règle générale, les logiciels et les systèmes logiciels intensifs sont développés par des équipes avec des centaines ou des milliers de personnes. Ils effectuent une multitude d'activités différentes, appelées processus.

2.3.1 Caractéristiques des processus logiciels

Le développement en équipe présente plusieurs caractéristiques difficiles à gérer lors de la conduite de projets. Certaines caractéristiques typiques sont (Münch et al., 2012):

- De nombreuses activités ne sont pas effectuées par des individus, mais sont partagées entre différents développeurs travaillant ensemble sans heurts.
- Dans les grands projets, une multitude d'activités peuvent être réalisées en parallèle. Cela nécessite une bonne coordination afin que les résultats de ces tâches s'imbriquent de manière planifiée.
- Il existe de nombreuses relations et des dépendances temporelles entre les activités. Des documents ou du code, par exemple, peuvent être échangés entre activités ou peuvent être utilisés conjointement par différentes activités.
- De nombreuses activités doivent être synchronisées afin qu'elles contribuent aux objectifs généraux du projet. Dans l'ingénierie des systèmes, par exemple, les processus d'ingénierie logicielle doivent souvent être synchronisés avec les processus d'ingénierie mécanique et électrique.
- La constitution d'équipes avec des responsabilités claires est un enjeu important. Le travail d'équipe implique, par exemple, la sélection d'une équipe, l'harmonisation des contributions de chaque membre, l'intégration de compétences et d'intérêts différents et la résolution de conflits.
- La gestion des processus humains nécessite de grandes compétences en leadership. L'une des principales tâches consiste à motiver les gens à contribuer à des objectifs communs.
- Outre les exigences du produit, les chefs de projet doivent tenir compte des exigences du processus lors de l'exécution de projets et de la direction d'équipes. Des exemples d'exigences de processus incluent le respect des normes de processus ou la productivité requise.

Les grands systèmes sont généralement constitués de composants de différentes disciplines (génie électrique, génie mécanique, génie logiciel). De plus, les systèmes logiciels pénètrent de plus en plus de domaines de notre vie quotidienne, ce qui signifie que ces systèmes doivent être faciles à utiliser pour les non-experts. Par conséquent, des disciplines telles que la sociologie et la psychologie deviennent de plus en plus pertinentes pour le développement de logiciels. Ainsi, des spécialistes de nombreuses disciplines doivent travailler ensemble lors du développement, de la maintenance ou de l'exploitation de systèmes et de services logiciels.

2.3.2 Avantages et objectifs de la modélisation des processus

La modélisation des processus logiciels prend en charge un large éventail d'objectifs. Basé sur Curtis et al. (1992), les objectifs de base suivants pour la modélisation des processus logiciels peuvent être envisagés (entre autres) :

- Faciliter la compréhension et la communication humaines

- Soutenir l'amélioration des processus
- Prise en charge de la gestion des processus
- Fournir des conseils automatisés dans l'exécution du processus
- Fournir un support d'exécution automatique.

Globalement, nous pouvons indiquer les objectifs suivants représentants un consensus dans la communauté des processus logiciels :

- La représentation : en définissant un langage de description ou une notation précise compréhensible et utilisable par tous pour la formalisation de la représentation.
- L'exécution : bien formaliser les notations conduit forcément à l'utilisation d'outils dédiés ainsi que l'automatisation de certaines phases du processus (Bendraou et al., 2008).
- La réutilisation : la bonne décomposition du processus sous forme de composants, le savoir faire et la récurrence des situations doit favoriser la réutilisation de tout ou partie d'un processus dans des cas similaires (Aoussat et al., 2014).
- L'analyse : un processus obéissant à des formalismes précis peut être directement ou indirectement sujet d'application de méthode d'analyse durant les différentes phases du processus. Il est évident de signaler, qu'une erreur dans le processus impliquerait forcément un produit dévié (Kaur & Sengupta, 2013).
-

Parmi d'autres, les avantages suivants sont attendus de la modélisation systématique des processus :

- Meilleure transparence des activités d'ingénierie logicielle.
- Réduction de la complexité des grands efforts de développement.
- La capacité d'effectuer des mesures de processus (c'est-à-dire que les modèles de processus utilisés dans la pratique sont une condition préalable à la mesure de processus et, par conséquent, à l'amélioration de processus).
- La capacité de subir des évaluations de processus (c'est-à-dire que des modèles de processus explicitement définis sont une condition préalable pour démontrer la maturité du processus).
- La prévisibilité en ce qui concerne les caractéristiques du processus et les caractéristiques des résultats n'est réalisable qu'avec des modèles explicites (c'est-à-dire que permettre la prévisibilité pour des caractéristiques telles que l'effort consommé, la date d'achèvement ou la fiabilité d'un composant logiciel produit nécessite l'existence de modèles de processus explicites).
-

2.4 Langage de modélisation des processus logiciel

Dans la vie de tous les jours, nous acquérons continuellement des connaissances en réalisant les travaux quotidiens. Ces connaissances peuvent être réutilisées plus tard dans des tâches similaires. Elles peuvent aussi êtres communiquées et analysées par un tiers. De ce fait,

ces connaissances doivent êtres capitalisées voire même formulées ou formalisées dans des langages dédiés compréhensibles par tous.

En termes de formalisation, les techniques et langages de modélisation de processus peuvent être classées en deux catégories :

- Les modèles descriptifs utilisent des techniques de modélisation graphique semi-formelles. Ils adoptent une perspective analytique, visant à fournir un environnement de communication, pour mieux comprendre (utiliser) et améliorer le processus.
- Les techniques de modélisation formelle sont fondées sur des paradigmes mathématiques rigoureux (algèbre de processus, réseaux de Petri, notations Z). Ils sont souvent utilisés pour décrire l'aspect dynamique du processus à des fins d'automatisation ou d'analyse et la simulation.

Dans le cadre de cette thèse, nous nous intéressons à la description des processus logiciels ou aux modèles descriptifs. Les techniques de modélisation de processus concernent généralement la cartographie (visuelle) et le flux de travail pour faciliter la compréhension, l'analyse et les changements positifs apportés au processus réel.

Plusieurs langages de modélisation ont été conçus à des fins différentes (García-Borgoñón et al., 2014). Ils ont apporté de nombreux avantages tels que la gestion de la complexité, ou la préservation et la réutilisation des connaissances expertes. La diversité des langages de modélisation est due aux différentes perspectives de modélisation, qui nécessitent souvent des concepts différents avec des propriétés différentes ainsi que la particularité de certains domaines.

La modélisation des processus vise à capturer les principales caractéristiques de l'ensemble des activités réalisées dans la production et/ou la maintenance d'un système logiciel afin de définir un langage de modélisation de processus approprié pouvant exprimer le processus dans un modèle compréhensible.

Dans ce qui suit, nous allons essayer de présenter un ensemble de langages de modélisation des processus logiciels. Chacun des langages choisis est considéré comme un langage de modélisation pour un domaine spécifique (DSL), celui des processus logiciels et de processus métier. Nous optons pour les standards OMG en relation avec ce domaine : SPEM, Essence et BPMN. Cette présentation sera assortie d'un comparatif de synthèse.

2.4.1 SPEM 2.0

Le méta-modèle d'ingénierie des processus logiciels et des systèmes (SPEM2.0) (OMG-SPEM, 2008) est l'un des méta-modèles les plus utilisés pour développer plusieurs modèles de processus logiciels. Il fournit un moyen de concevoir le processus de développement avec les conceptions de base pour modéliser le contenu des processus et le contenu de la méthode qui le manipule. Ce méta-modèle (SPEM 2.0) définit une séparation claire entre les éléments du contenu de la méthode et les éléments de processus (comme dans Figure 2.7). Cela signifie qu'il sépare le contenu de la méthodologie (par exemple, les tâches, les produits de travail et les rôles), de son instanciation dans un processus particulier (par exemple, les activités, l'utilisation des rôles et l'utilisation des produits de travail).

SPEM est un méta-modèle d'ingénierie des processus ainsi qu'un cadre conceptuel. Il fournit des concepts standards pour la modélisation des processus de développement de logiciels et il est également spécifié en tant que méta-modèle basé sur MOF 2.0 et en tant que

profil basé sur la superstructure UML2.0

La Figure 2.7 montre les concepts clés définis dans la spécification SPEM. Les concepts de contenu de méthode tels que « Task », « Work Product » et « Role » sont utilisés pour définir des descriptions réutilisables qui peuvent être incorporées dans plusieurs processus.

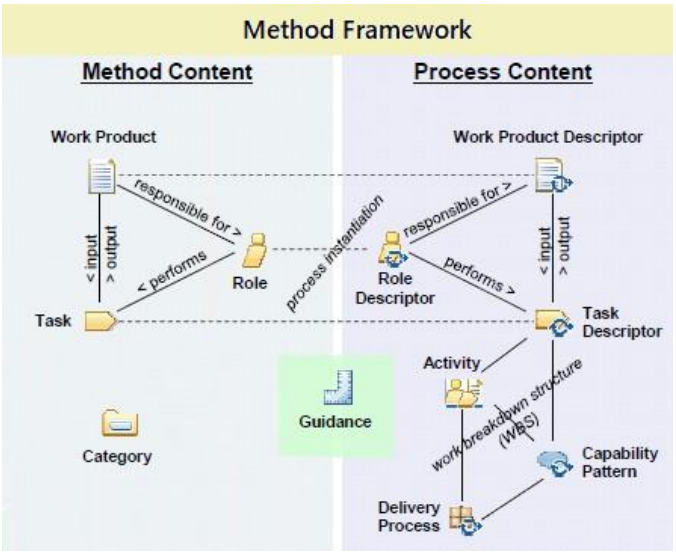


Figure 2.7- Aperçu des concepts clés de SPEM 2.0.

Les éléments correspondants dans le contenu du processus, c'est-à-dire que « Task Descriptor », « Work Product Descriptor » et « Role Descriptor », sont essentiellement des copies qui contiennent les informations des homologues du contenu de la méthode. Cependant, ces descripteurs peuvent être modifiés localement dans le cadre de la description du processus sans interférer avec les définitions d'origine utilisées ailleurs.

Le contenu du processus comprend également d'autres éléments comme « Activity », « Capability Pattern » et « Delivery Process » qui peuvent être utilisés pour organiser et définir des structures de répartition du travail et des modèles de processus réutilisables. Le contenu de la méthode comprend « Category » qui peut être utilisée pour définir des catégories personnalisées, (Exemple, discipline pour catégoriser les tâches et Outil pour catégoriser les conseils ou orientation « Guidance » spécifiques aux outils).

2.4.2 Essence 2.0

Le projet REMICS (*REuse and Migration of legacy applications to Interoperable Cloud Services*) (Ilieva et al., 2010) vise à développer des méthodologies agiles qui traitent spécifiquement de la modernisation dirigée par les modèles des applications existantes pour desservir les services du Cloud. Le projet REMICS a participé à l'initiative SEMAT (*Software Engineering Method and Theory*) (Striewe et al., 2012) qui a présenté une nouvelle spécification intitulée "*Essence - Kernel and Language for Software Engineering Methods*"(OMG-Essence, 2018).

Essence a été la toute première étape d'un effort visant à révolutionner l'ingénierie logicielle. Un terrain d'entente était nécessaire pour construire l'avenir. La norme Essence a été créée en 2014 par le SEMAT et l'OMG pour fournir un langage universel permettant de définir des méthodes et des pratiques communes à tout le génie logiciel. D'ores et déjà, la spécification Essence promet un meilleur support pour la définition des pratiques agiles et l'adoption de méthodes par rapport à la spécification SPEM.

Essence décrit intuitivement les éléments essentiels de tout développement logiciel et aide les équipes à comprendre où elles en sont, ce qui manque ou ce qui doit être résolu.

Essence est indépendante des méthodes et réunit la pléthorique de méthodes et de pratiques déconnectées, conflictuelles, coûteuses, contradictoires et transitoires qui existent dans la plupart des grandes organisations.

Essence aide les praticiens (par exemple, les architectes, les concepteurs, les développeurs, les testurs, les développeurs, les ingénieurs des exigences, les ingénieurs de processus, les chefs de projet, etc.) à comparer les méthodes et à prendre de meilleures décisions concernant leurs pratiques (OMG-Essence, 2018).

La spécification Essence introduit une architecture de méthode qui fait la distinction entre les trois entités suivantes : les noyaux, les pratiques, les méthodes et les langages qui les définissent.

- Un noyau fournit un modèle simplifié et léger des aspects essentiels du génie logiciel.
- Une pratique est une description de la façon de gérer un aspect spécifique d'une entreprise de génie logiciel (Par exemple « users stories » pour la spécification des exigences et « Scrum » pour la gestion de projet agile).
- Une méthode est la composition d'un noyau et d'un ensemble de pratiques pour atteindre un objectif spécifique (décrire comment un effort est effectué). Elle fournit des conseils spécifiques pour former une manière adaptée à l'effort logiciel et aux besoins de l'équipe de développement

La Figure 2.8 prise dans REMICS montre l'architecture d'Essence dans laquelle le langage représente le DSL (*Domain Specific Language*) pour définir les méthodes, les pratiques et les éléments essentiels du noyau.

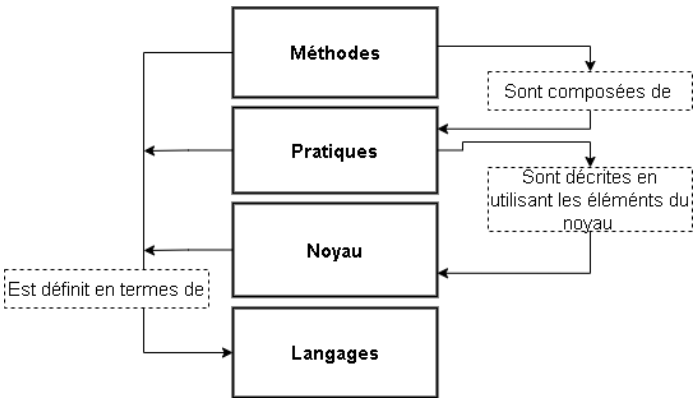


Figure 2.8- - Architecture de la méthode Essence.

La Figure 2.9 montre les concepts clés définis dans la spécification Essence. Un noyau « Kernel » est exprimé en utilisant les concepts « Alpha », « Activity Space » et « Competency ». Les « Alphas » représentent des éléments avec lesquels travailler, ils ont des « Alpha States » pour suivre leur progression et peuvent être considérés comme un espace réservé pour les « Work Product ». Les « Activity spaces » représentent des choses à faire et peuvent être considérées comme un espace réservé pour les « Activity ». Les « Competencies » représentent les ensembles combinés d'habiletés, de connaissances et de comportements. Une « Pratique » ajoute des détails spécifiques en définissant des « Produits de travail » et des « Activités », mais peut également introduire de nouveaux « Alphas », « Espaces d'activités » et « Compétences ». Le langage définit également « Ressource » pour modéliser différents types de ressources, par exemple « Template » et « Pattern » pour modéliser des modèles génériques ou spécifiques comme le « Team Role ».

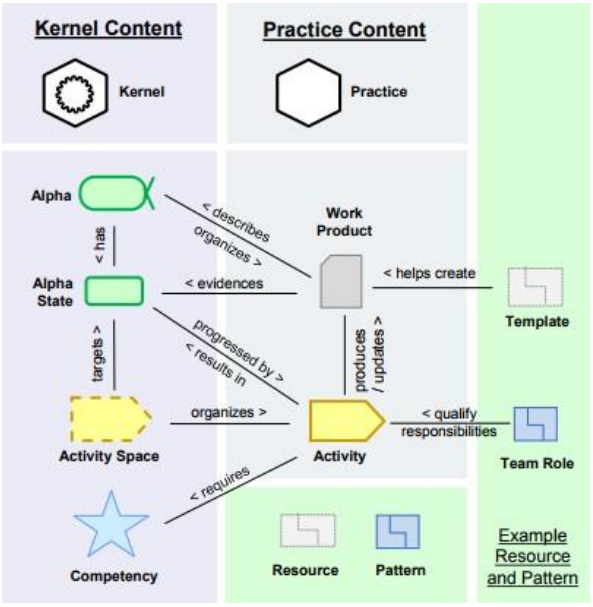


Figure 2.9- Aperçu des concepts clefs d'Essence.

2.4.3 BPMN 2.0

Dans la Figure 2.4, nous avons précisé la relation qui existe entre les processus métier et les processus logiciels. Partant du fait que le processus logiciel lui-même est un métier d'ingénierie ayant ses propres spécificités, nous pouvons considérer BPMN comme un langage ou une norme pour la description des processus logiciels. Par conséquent, une comparaison entre les trois langages est envisageable.

Le modèle et la notation de processus métier BPMN (Business Process Model and Notation) sont devenus la norme pour les diagrammes de processus métier (OMG-BPMN, 2011). Il est destiné à être utilisé directement par les parties prenantes qui conçoivent, gèrent et réalisent des processus métier, mais en même temps suffisamment précis pour permettre la traduction des diagrammes BPMN en composants de processus logiciels. BPMN a une notation de type organigramme facile à utiliser qui est indépendante de tout environnement de mise en œuvre particulier.

Ce modèle et cette notation contribuent pour atteindre les premiers objectifs suivants :

- fournir une notation facilement compréhensible par tous les intervenants (utilisateurs métier, analystes métier qui créent les ébauches initiales des processus, développeurs techniques qui sont responsables de la mise en œuvre de la technologie qui exécutera ces processus, les commerciaux).
- créer un pont standardisé pour l'écart entre la conception des processus métier et la mise en œuvre des processus.
- s'assurer que les langages XML conçus pour l'exécution de processus métier, tels que WSBPEL (*Web Services Business Process Éxécution Language*), peuvent être visualisés avec une notation orientée métier.

Cette spécification représente la fusion des meilleures pratiques au sein de la communauté de modélisation d'entreprise pour définir la notation et la sémantique des diagrammes de collaboration, des diagrammes de processus et des diagrammes de chorégraphie. L'intention de BPMN est de normaliser un modèle de processus métier et une notation face à de nombreuses notations et points de vue de modélisation différents. Ce faisant, BPMN fournira un moyen simple de communiquer des informations de processus à

d'autres utilisateurs professionnels, développeurs de processus, clients et fournisseurs.

Puisque BPMN est un langage de notation permettant de modéliser des processus métier dans divers domaines, il fournit plus de 100 éléments de notation pour représenter les modèles de processus métier. Par rapport au noyau de SPEM, il existe des éléments (comme un sous-processus et une tâche) pour représenter une « Work Unit » qui doit être effectuée afin de produire des « Work Product » ou « Data-Object » dans BPMN (Dashbalbar et al., 2017b).

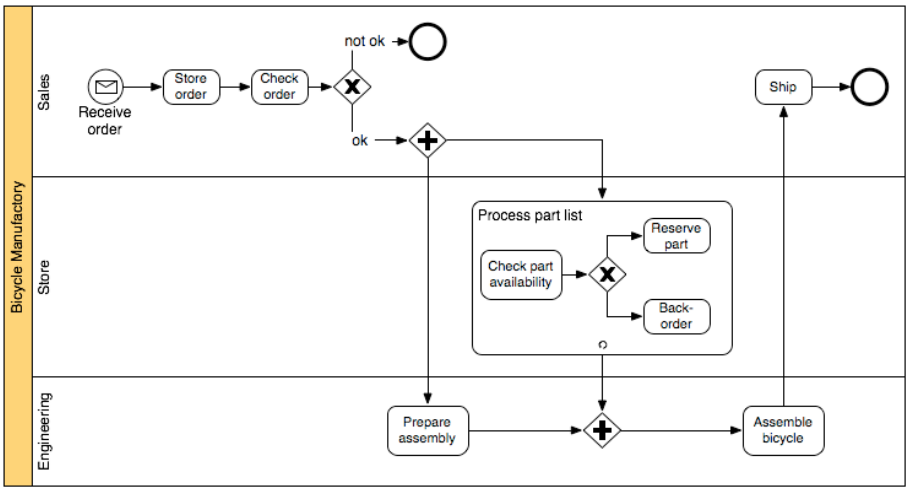


Figure 2.10- Exemple de modèle de processus BPMN.

Figure 2.10 montre un processus métier modélisé dans BPMN, ce qui nous donne l'occasion de nous familiariser avec les éléments de base de BPMN. Il existe un processus métier nommé « Bicycle Manufacturer », qui comporte trois voies où chacune présente un rôle. L'élément contenu dans les limites d'une voie signifie qu'il est exécuté par un travailleur correspondant à la voie.

2.4.4 Synthèse

Nous avons opté volontaire pour comparer ces trois langages pour des raisons évidentes dans le sens qu'ils représentent tous des normes ou des standards de modélisation des processus logiciels soutenus par l'OMG. D'autre part, ces langages manipulent plus ou moins les mêmes concepts et concourent à des mêmes objectifs. Ceci d'une part, d'autre part si SPEM et Essence ont une même vocation moyennant quelques variations, le BPMN se distingue par le fait qu'il s'intéresse à modéliser les processus métier alors que les deux autres sont dédiés aux processus logiciels.

Or, il est utile de rappeler que les processus logiciels représentent eux-mêmes un métier et par conséquent ils peuvent être formalisés par les BPMN. Par conséquent, les trois langages deviennent comparables (Cruz et al., 2020; Dashbalbar et al., 2017a; Fonseca et al., 2021).

Aussi, il est utile de signaler que les BPMN obtiennent un intérêt grandissant dans le monde de la recherche par rapport aux deux autres notations (Cruz et al., 2020; Díaz et al., 2021; Lübke & Ahrens, 2022; Ocampo-Pineda et al., 2022). Ceci peut être justifié par le besoin pour chercher à formaliser davantage les langages des processus métiers. L'objectif ultime pour cette tendance est sûrement le fait d'aboutir au concept de « métier exécutable » comme jadis était le cas pour le « modèle exécutable ». L'idéal serait de générer l'application directement à partir des processus métiers.

Tableau 2.1- Correspondance des concepts manipulés par les standards.

Concept	SPEM	Essence	BPMN
Processus	Process	Practices	Business Process
Activité	Activity	Action	Sub-Process
Tâche atomique	Task	Atomic Action,	User Task, Send Task, Receive Task, Manual Task, Service Task, Business Rule Task, Script Task
Itération	Iteration	Activity Space	sub-process avec Standard Loop attribute
Type de produit	Kind WorkProduct Class	WorkProduct	Input, Output Message
Produit/ Artifact/ Livrable	WorkProductDefinition, WorkProductUse	WorkProduct Alpha	Data Object Élément
Domaine	Groupe of related Work Product	//	Data Object Collection
Groupe de travail	RoleUSe, Role	Pattern	Lane, Pool
Borne (cycle)	Milestone	Event	Event
Groupe d'éléments, tâches	Category, Discipline	Alpha/ Area of Concern	Group
Outils utilisés par role	ToolDefinition	Tool	Text Annotation
Variété d'orientation	Guidance	Competency	Text Annotation

Suite à l'étude des travaux de comparaison et de correspondance entre les trois standards (Cruz et al., 2020; Dashbalbar et al., 2017b; Fonseca et al., 2021; García-Borgoñón et al., 2014; Henderson-Sellers & Gonzalez-Perez, 2005) nous avons dégagé les tableaux Tableau 2.1 et Tableau 2.2. Le Tableau 2.1 essaye de traduire et de lisser les différents concepts dans les différentes notations pour d'éventuels passages et mappings alors que le Tableau 2.2 dresse un comparatif conceptuel direct entre ces trois standards.

Tableau 2.2- Eléments comparatifs sur les standards de modélisation de processus.

Propriété	SPEM	Essence	BPMN
Vocation	Généraliste	Agile/ Scrum (2017)	Métier
Sémantique de processus	Oui	Oui	Non
Méta-modèle	Oui	Non	Oui
Concept de cycle de vie	Oui	Oui	Non
Outils	EPF[1] /Outils UML	EPF Composer	BPEL[2]

2.5 Eléments du langage SPEM 2.0

La première version du standard SPEM a été introduite par l'OMG en 2002, elle a été construite sur UML 1.4. Elle a été révisée en 2005 puis en 2008 pour apporter des changements majeurs dans la version SPEM 2.0 qui demeure compatible à UML 2.0. Cette conformité peut permettre l'usage des standards UML tels que les diagrammes d'activité ou les diagrammes d'état pour visualiser les modèles de processus.

On parle souvent de Méta-modèle SPEM et de Profil SPEM d'une manière qui semble être ambigüe à première vue. Or, il suffit juste de considérer la vue et le point de vue avec lesquels ont voit ces concepts pour lever cette ambiguïté. Le concept de Méta-modèle SPEM est équivalent en termes de niveau de modélisation à UML, il peut instancier une infinité de modèles de processus. La notion de profil SPEM en revanche montre le langage d'expression de ce méta-modèle qui n'est autre qu'un profil UML partant du fait qu'UML n'a aucun moyen pour décrire les processus. Ceci dit, les deux concepts sont équivalents comme le traduit la Figure 2.11 tirée du format standard.

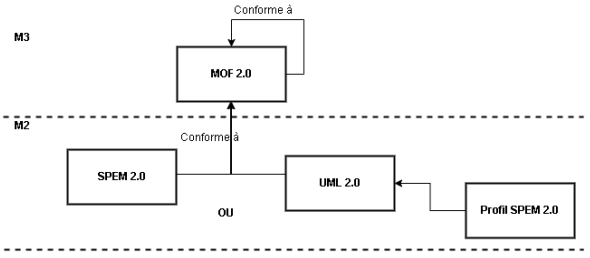


Figure 2.11- Méta-modèle ou Profil SPEM.

Définition 8 : « *Le méta-modèle d'ingénierie des processus logiciels et systèmes (SPEM) est un méta-modèle d'ingénierie des processus ainsi qu'un cadre conceptuel, qui peut fournir les concepts nécessaires pour modéliser, documenter, présenter, gérer, échanger et mettre en œuvre des méthodes et des processus de développement.* » (Münch et al., 2012).

2.5.1 Concepts de SPEM

Dans cette section, nous allons nous intéresser au cadre conceptuel de SPEM, des notations et enfin de la structure de ce standard.

[1] Eclipse Process Framework Project (EPF) (Haumer, 2007): https://www.eclipse.org/epf/
[2] Business Process Éxécution Language (BPEL) (Juric et al., 2006): https://docs.oracle.com/bpel_install.htm

2.5.1.1 Cadre conceptuel de SPEM

Le cadre conceptuel de SPEM donne les principaux objectifs du standard. Il s'agit de fournir une approche pour créer des bibliothèques de contenus de méthodes réutilisables et de fournir des concepts pour le développement et la gestion des processus. La combinaison de ces deux objectifs de base est considérée comme une solution qui permet la configuration de cadres de processus plus élaborés et enfin leur transposition dans de vrais projets de développement (Münch et al., 2012).

La principale distinction de SPEM 2.0 par rapport aux versions précédentes réside dans le fait de voir et de considérer la notion d'opération (Hug, 2009). L'opération représente généralement l'unité de travail. Dans la version SPEM 1.0 (Figure 2.6), l'unité de travail désigne l'opération à réaliser (l'activité, l'action) et le temps imparti pour sa réalisation (phase, itération, cycle de vie). Ces concepts sont regroupés sous la même super-classe « *WorkDefinition* » alors qu'ils n'ont pas la même sémantique. Dans SPEM 2.0 en revanche, l'aspect opérationnet et l'aspect temporel sont nettement séparés. Ces deux notions forment deux méta-modèles différents présentés sous forme de packages : «*Method Content* » pour l'aspect opération et «*Process Structure* » pour l'aspect temporel (Figure 2.12).

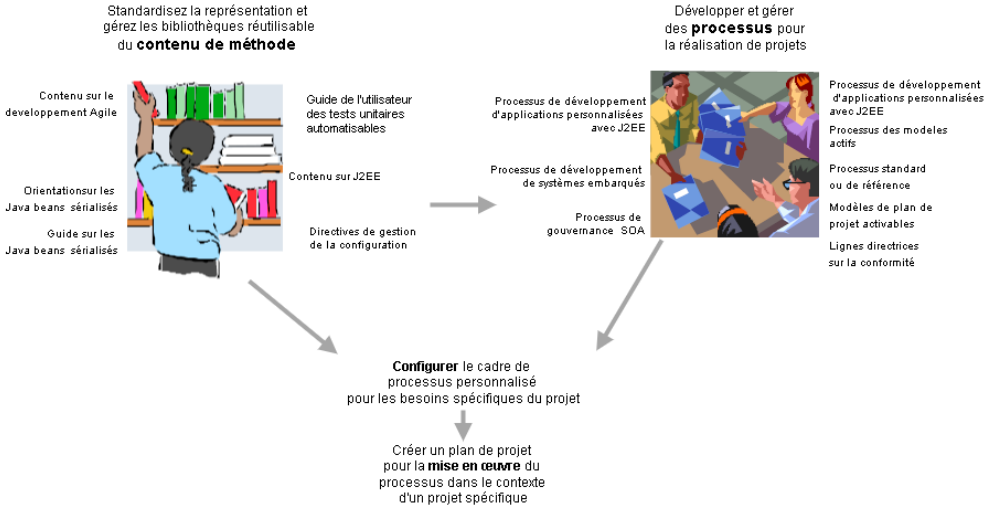


Figure 2.12- Cadre conceptuel de SPEM 2.0.

La Figure 2.12 montre une vue empirique de l'usage de SPEM pour la modélisation d'un processus logiciel. Ce que montre la figure est une démarche commune et non pas une méthode. Modéliser un processus logiciel revient à passer par les quatre étapes suivantes :

To provide a standardized representation and managed libraries of reusable method content: Developers need to understand the methods and key practices of software development. They need to be familiar with the basic development tasks, such as how to elicit and manage requirements, how to do analysis and design, how to implement for a design or for a test case, how to test implementations against requirements, how to manage the project scope and change, and so on. They further need to understand the work products such tasks create as well as which skills are required. SPEM 2.0 aims to support development practitioners in setting-up a knowledge base of intellectual capital for software and systems development that would allow them to manage and deploy their content using a standardized format. Such content could be licensed, acquired, and, more importantly, their own homegrown content consisting of, for example, method definitions, whitepapers, guidelines, templates, principles, best practices, internal procedures and regulations, training material, and any other général descriptions of how to develop software. This knowledge base can be

used for reference and education and forms the basis for developing processes (see the next bullet point).

- Contenu de la méthode « *MethodContent*) » : Dans le but de standardiser la représentation et gérer les bibliothèques réutilisables des méthodes, les développeurs doivent comprendre les méthodes et les pratiques clés du développement logiciel. Ils doivent être familiarisés avec les tâches de développement de base, telles que la façon de définir et de gérer les exigences, comment faire l'analyse et la conception, comment implémenter une conception ou un cas de test, comment testr les implémentations par rapport aux exigences, comment gérer le portée et changement du projet, etc. SPEM 2.0 vise à soutenir les praticiens du développement dans la mise en place d'une base de connaissances du capital intellectuel pour le développement de logiciels et de systèmes qui leur permettrait de gérer et de déployer leur contenu en utilisant un format standardisé. Ce contenu pourrait être sous licence, acquis et, plus important encore, leur propre contenu interne composé, par exemple, de définitions de méthodes, de livres blancs, de lignes directrices, de modèles, de principes, de meilleures pratiques, de procédures et de réglementations internes, de matériel de formation et de toute autre description générale.
- Processus « *Process* » : Pour soutenir le développement, la gestion et la croissance systématiques des processus de développement : les équipes de développement doivent ***définir comment*** appliquer leurs méthodes de développement et leurs meilleures pratiques tout au long du cycle de vie d'un projet. Par exemple, les méthodes de gestion des exigences doivent être appliquées d'une manière au cours des premières phases d'un projet, où l'accent est davantage mis sur l'élicitation des besoins et des exigences selon des visions des parties prenantes ou intervenants. Les équipes ont également besoin d'une compréhension claire de la façon dont les différentes tâches au sein des méthodes sont liées les unes aux autres : par exemple, comment la méthode de gestion du changement affecte la méthode de gestion des exigences ainsi que la méthode de test de régression tout au long du cycle de vie. SPEM 2.0 prend en charge la création systématique de processus basés sur un contenu de méthode réutilisable.
- *Configuration* : Pour prendre en charge le déploiement du contenu de la méthode et du processus, il est nécessaire définir des configurations de processus et du contenu de la méthode. Dans la pratique, aucun projet de développement n'est exactement commun à un autre et le même processus de développement n'est jamais exécuté deux fois (OMG-SPEM, 2008). Les notions, « *Activity use* », « *Configurability* » et « *Variability* » pour les processus de développement (ainsi que le contenu des méthodes) dans SPEM 2.0 offrent des capacités de réutilisation de processus ou de modèles de processus, de modélisation de la variabilité et de personnalisation permettant aux utilisateurs de définir leurs propres extensions, omissions et points de variabilité sur des processus standard réutilisés. Par conséquent, le scénario d'utilisation SPEM est que les organisations peuvent fournir des bibliothèques de processus et de méthodes réutilisables en utilisant les capacités décrites dans les deux premiers points. Les chefs d'équipe peuvent ensuite sélectionner et adapter le contenu de la méthode et les

processus dont ils ont besoin. Ils peuvent ensuite décrire ces sélections et personnalisations avec une configuration de méthode SPEM, qu'ils peuvent déployer auprès de leurs équipes, en ne fournissant que le contenu dont ils ont vraiment besoin.

- *Mise en Œuvre* : Pour soutenir la mise en place d'un processus de projets de développement, une définition de processus n'apporte de la valeur que si elle impacte et oriente le comportement des équipes de développement. Les processus ainsi que le contenu des méthodes de guidage doivent être disponibles dans le contexte du travail quotidien des chefs de projet, des responsables techniques et des développeurs. En bref, les systèmes de mise en œuvre sont les systèmes de planification de projets et de ressources, les systèmes de suivi des arriérés de travail et les moteurs de flux de travail. SPEM 2.0 fournit des définitions de la structure de processus qui permettent aux ingénieurs de processus d'exprimer comment un processus doit être mis en œuvre dans un contexte précis.
-

2.5.1.2 Eléments et notations de base

Le noyau de SPEM 2.0, comme la majorité des langages de description des processus, est constitué de trois éléments principaux qui sont : l'activité, le rôle et le produit. Comme nous l'avons cité plutôt, les deux concepts clés qui forment la base conceptuelle de SPEM 2.0 sont : le contenu de la méthode « MethodContent » et le Processus « Process ». Nous avons dit que le contenu de la méthode définit les éléments pour être utilisés par le processus.

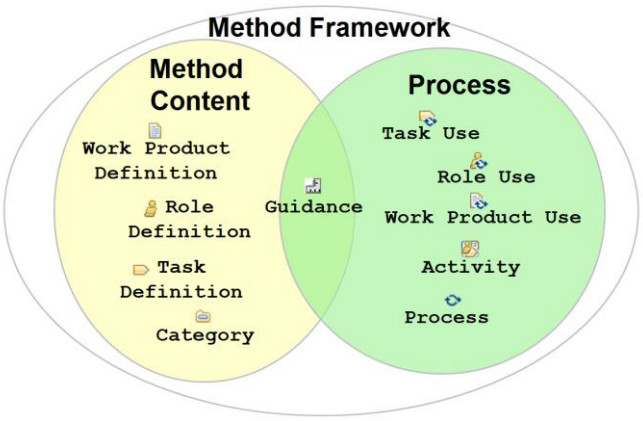


Figure 2.13- Terminologie clé associée au contenu de la méthode vs le processus.

La Figure 2.13 (OMG-SPEM, 2008) regroupe et positionne les éléments et concepts de base qui forment le noyau de SPEM 2.0 ou le « Method Framewrok ». Il est important de remarquer que ces éléments de base sont définis avec le mot « Definition » dans la « Method Content » et avec le mot « Use » dans le processus. Ce qui signifie ce que nous avons dit précédemment c'est-à-dire ce que nous définissons dans la méthode, nous l'utilisons dans le processus. Tout en remarquant que « Guidance » est partagé entre les deux concepts. Le Tableau 2.3 résume ces différents éléments en précisant la description, le rôle et l'icône associé :

Tableau 2.3- Définition et representation des éléments de base.

Elément	Description	Icône
« Work Product Definition »	élément de « Method Content » qui qui est utilisé, modifié et produit par « Task Définition ».	
« Role Definition »	élément de "Method Content" qui définit un ensemble d'aptitudes, de compétences et de responsabilités connexes. Les rôles sont utilisés par les «Task Definition» pour définir qui les exécute ainsi que pour définir un ensemble de «Work Product Définition» dont ils sont responsables.	
« Task Definition »	élément de « Method Content » et un « Work Definition » qui définit le travail effectué par les instances de « Role Definition ». Une tâche est associée à des produits de travail d'entrée et de sortie.	
« Category »	élément descriptible « Describable Élément » utilisé pour catégoriser, c'est-à-dire regrouper n'importe quel nombre d'éléments descriptibles de n'importe quel sous-type en fonction de critères définis par l'utilisateur. Elles peuvent aussi être utilisées pour catégoriser de manière récursive et imbriquée.	
« Guidance »	Élément descriptible qui fournit des informations supplémentaires liées à d'autres éléments de processus définis dans d'autres packages. Les orientations peuvent être des lignes directrices, des modèles, des listes de contrôle, des mentors d'outils, des estimations, des documents de support, des rapports, etc.	
« Work Product Use »	représente un type d'entrée et/ou de sortie pour une activité ou représente un participant général de l'activité. Il représente une occurrence de « Work Product »	
« Role Use »	élément de répartition spécial qui représente soit un exécutant d'une activité, soit un participant à l'activité.	
« Task Use »	Une « TaskUse » est une utilisation du contenu d'une méthode et un élément décomposble qui représente un proxy pour une « TaskDefinition » dans le contexte d'une « Activité » spécifique.	
« Activity »	est un élément décomposable « WorkBreakdownÉlément » et « WorkDefinition » qui définit les unités de travail de base au sein d'un processus ainsi qu'un processus lui-même. Chaque activité représente un processus dans SPEM 2.0. Il concerne les instances « WorkProductUse » via les instances de la classe « ProcessParameter » et les instances « RoleUse » via les instances « ProcessPerformer ».	
« Process »	est une activité spéciale qui décrit une structure pour des types particuliers de projets de développement ou des parties de ceux-ci. Pour réaliser un tel projet de développement, un processus serait adapté à la situation spécifique de l'organisation ou du projet, puis «instancié» en attribuant des ressources concrètes à RoleUses, en créant plusieurs instances pour WorkProductUses, etc.	

2.5.2 Méta-modèle SPEM 2.0

SPEM 2.0 est un standard adopté par l'OMG, c'est un méta-modèle qui permet de décrire les concepts et principes de base pour la modélisation des processus logiciels. L'objectif de SPEM n'est pas de devenir un langage générique de modélisation mais de couvrir la description des concepts d'un large éventail de processus logiciels, sans se spécialiser dans un type particulier, tout en prenant en compte leur diversité reconnue (OMG-SPEM, 2008).

2.5.2.1 Noyau conceptuel

Comme tout langage de description de processus logiciel, le noyau conceptuel de SPEM est formé par des éléments et des concepts de base. Au niveau le plus élevé, il existe trois méta-classes: "Activité" appelée "Unité de travail", "Produit de travail", "Rôle" avec les relations qui les lient (Figure 2.14). Cette figure indique simplement que les activités utilisent des produits de travail afin de créer de nouveaux produits de travail. Le travail est entrepris par (généralement) des personnes jouant des rôles (Henderson-Sellers & Gonzalez-Perez, 2005).

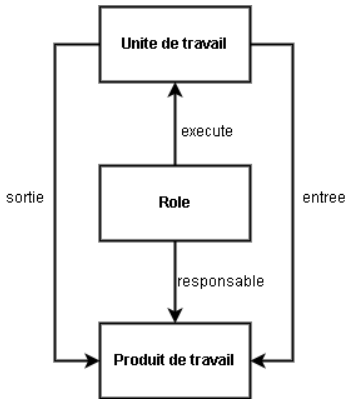


Figure 2.14- Noyau de SPEM.

Pour représenter ces concepts et les relations entre ces concepts, SPEM a défini quatre types de classes (Figure 2.6):

- Classes représentant la notion d'unité de travail, par exemple : TaskDefinition, TaskUse, Activity...etc.
- Classes représentant le réalisateur de l'unité de travail, par exemple : ProcessPerformer, DefaultProcessperformer , ProcessRole.
- Classes représentant le produit de travail, par exemple : WorkProdutDefinition, WorkProductKind, WorkProductUse...etc.
- Classes représentant la notion d'élément ; ces classes permettent de décrire d'autres concepts de processus logiciel tels que composant, outil, orientation, etc. Ces concepts sont définis par les classes suivantes (ProcessComponent, ToolDefinition, Guidance,...etc.) .

Ces quatre types de classes sont bien agencés et sont présents à plusieurs niveaux de SPEM, ils permettent d'offrir plusieurs vues des concepts de processus logiciels selon différents angles (Aoussat, 2012).

L'ensemble des packages SPEM sont basés sur ces éléments conceptuels. En fonction des relations qui lient, chaque package « étend » ces éléments en fonction de l'angle et de

l'aspect qu'il est censé prendre en charge.

2.5.2.2 Packages SPEM 2.0

En procédant par décomposition / composition, Le méta-modèle SPEM 2.0 est structuré en sept packages de méta-modèles principaux, comme illustré par la figure de dépendance (Figure 2.15). La structure divise le modèle en unités logiques dépendantes. Chaque unité étend les unités dont elle dépend, fournissant des structures et des capacités supplémentaires aux éléments définis ci-dessous. Globalement, le mécanisme de « fusion » de packages UML 2 appliqué aux packages réalise une extension progressive des capacités modélisées unité par unité. Par conséquent, les unités définies sur une couche inférieure peuvent être réalisées par une implémentation de sous-ensemble SPEM 2.0 sans les unités de niveau supérieur. Dans de nombreux cas, les classes de méta-modèles sont introduites dans un package de niveau inférieur aussi simplement que possible, puis sont étendues dans des unités de niveau supérieur via le mécanisme de « fusion » de packages avec des propriétés et des relations supplémentaires pour réaliser des exigences de modélisation de processus plus complexes (OMG-SPEM, 2008).

Le méta-modèle SPEM est ainsi découpé et organisé, en différents packages et relations entre ces packages, pour atteindre les objectifs suivant :

- couvrir les besoins de modélisation et d'exécution d'un large éventail de processus logiciels, chaque méta-modèle décrit une vision particulière.
- faciliter la compréhension et l'extension de SPEM.
- augmenter la réutilisation des concepts décrits par SPEM, que ce soit les structures des processus logiciels, les connaissances des méthodes de développement ou la documentation et assistance.

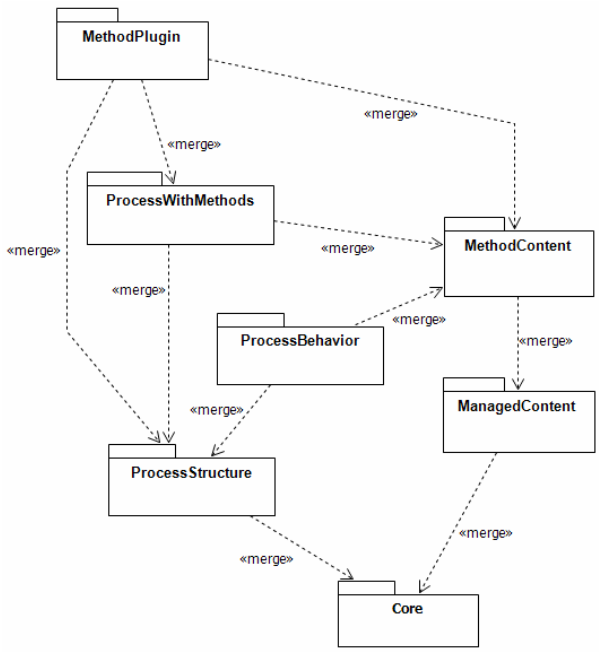


Figure 2.15- Structure générale de SPEM 2.0.

a) Package noyau (Core)

Le package de méta-modèle « *Core* », contient les classes de méta-modèle et les abstractions qui constituent la base de tous les autres packages de méta-modèle et qui

permettent leurs descriptions. Autrement dit, toutes les classes communes à tous les niveaux de conformité définissant le cœur de SPEM 2.0 ont été placées dans ce package de méta-modèles. De ce fait, il fournit la base conceptuelle pour l'extension des autres méta-modèles. Le nombre des classes de ce package est restreint.

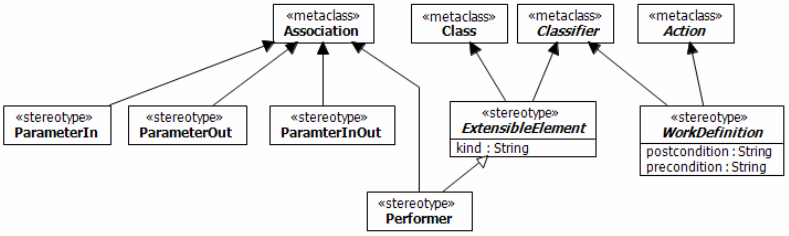


Figure 2.16- Structure de package "Core".

Core contient les classes communes définies dans les différents packages. Ce package contient plusieurs éléments importants, il définit des classes pour deux fonctionnalités de SPEM 2.0:

- *Kind (Classe ExtensibleElment)*: permet aux utilisateurs de SPEM de préciser des termes spécifiques à leur environnement, il permet de distinguer différents "types" d'instances de classes. Par exemple, *Phase* peut être définie comme un type (*Kind*) de *BreakdownÉlément* (classe définie dans le package *Process Structure*).De même que *Iteration* ou *Increment* pour spécifier différents types d'éléments de processus.
- Un ensemble de classes abstraites pour définir le travail exprimé sous forme de processus SPEM 2.0. comme les trois concepts clés de SPEM des stéréotypes: Unité de travail (*WorkDefinition*) a des produits en paramètres (*ParametreIn, Out , InOut*) et est réalisée par des rôles (*Performer*). Ces concepts sont spécialisés dans les packages *Process Structure* et *Method Content*.
-

b) Package contenu de la méthode (Method Content)

Le package de contenu de méthode définit les éléments de base de chaque méthode, tels que les définitions de rôles, de tâches et de produits de travail. Il décrit des rôles qui exécutent des tâches pour réaliser des produits, sans préciser l'enchaînement des tâches et leur place précise dans un processus.

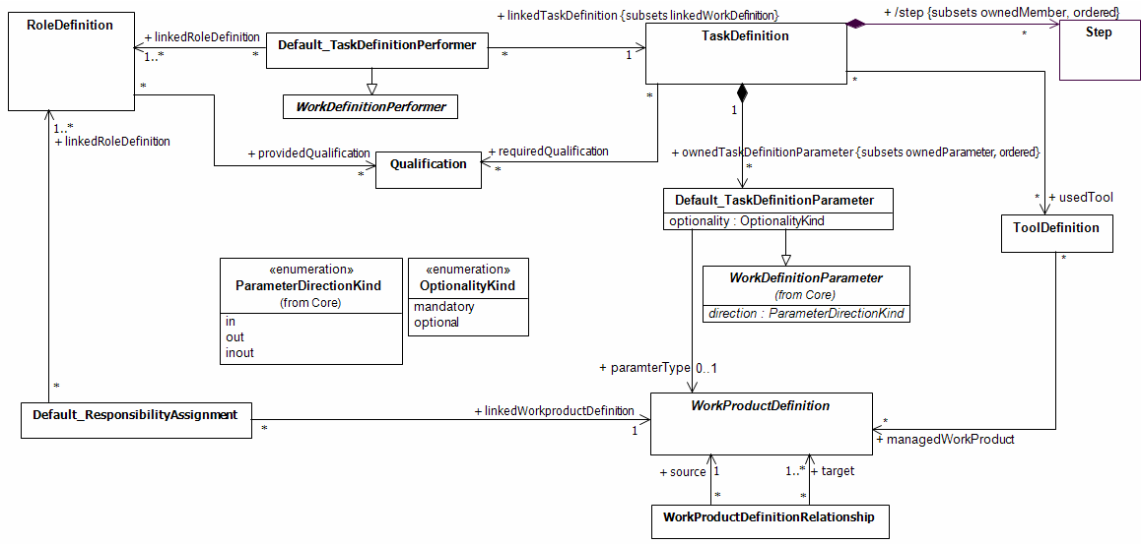


Figure 2.17- Structure du méta-modèle « Method Content ».

Le package *Method Content* décrit les opérations à réaliser, les produits créés durant ces opérations et les acteurs intervenants. Nous pouvons comparer les modèles créés avec *Method* Content à des composants de méthodes qui seront ensuite assemblés dans une structure temporelle via le package *Process Structure*.

La Figure 2.17 (OMG-SPEM, 2008) donne la structure générale de ce package. La figure illustre ces éléments de base avec leur modèle de relations en tant que sous-classes de *MethodContentÉlément*. Les relations sont étiquetées avec un préfixe "*Default_*", car elles décrivent la manière "standard" ou "normale" d'exécuter la méthode décrite. Les opérations à réaliser sont décrites dans la classe (*TaskDefinition*) ou tâches pouvant être décomposées en étapes (*Step*) atteindre un objectif pas à pas. Aucune notion de temps d'exécution, de commencement ou de fin n'est attachée à une tâche.

Pour chaque tâche, on peut définir des produits (*WorkProductDefinition*) en paramètres, optionnels ou non. De la même manière, des rôles (*RoleDefinition*) sont associés à une tâche. Il est possible de décrire des qualifications nécessaires pour la réalisation d'une tâche ou les qualifications dont un rôle dispose, ceci dans le but de faire correspondre au mieux les rôles aux tâches à réaliser, en fonction des compétences requises.

Il est également possible de définir les outils (*ToolDefinition*) qui peuvent être utilisés pour manipuler des produits. Des produits peuvent avoir des relations entre eux, par exemple la composition ou l'agrégation de produits, ce que modélise la classe *WorkProductDefinitionRelationship*. Enfin, un rôle peut avoir une responsabilité sur un produit (*Default_ResponsabilityAssignment*), ces responsabilités peuvent être par exemple : responsable, pour signature, pour consultation (Hug, 2009) .

c) Package structure du processus (Process Structure)

Le package « *Process Structure*) est la première extension du package « *Core* ». Le noyau de ce package est la représentation d'un fragment procédé « *BreakdownÉlément* » défini indépendamment de toute méthode ou cycle de vie de logiciel. L'exemple typique de cette tendance sont : les procédés SCRUM (Münch et al., 2012; Srivastava et al., 2017) , les procédés itératifs et incrémentaux utilisés souvent dans les développements logiciels qui se basent sur l'auto-organisation des équipes. Plus généralement, ce mécanisme *BreakdownÉlément* (structure de décomposition) est défini indépendamment des modèles de cycle de vie concrets que l'ingénieur de procédés souhaite exprimer avec eux.

Ce *package* décrit la structuration des processus logiciel, c'est-à-dire comment un processus est organisé en termes d'enchaînement et d'organisation: les éléments (*BreakdownÉlément*) sont décomposés jusqu'aux activités (*Activity*) qui comprennent des produits (*WorkProductUse*) et des rôles (*RoleUse*). Les éléments faisant référence à des unités de travail (*WorkBreakdownÉlément*) peuvent être définis en séquence ou en parallèle (*WorkSequence*) (Figure 2.18).

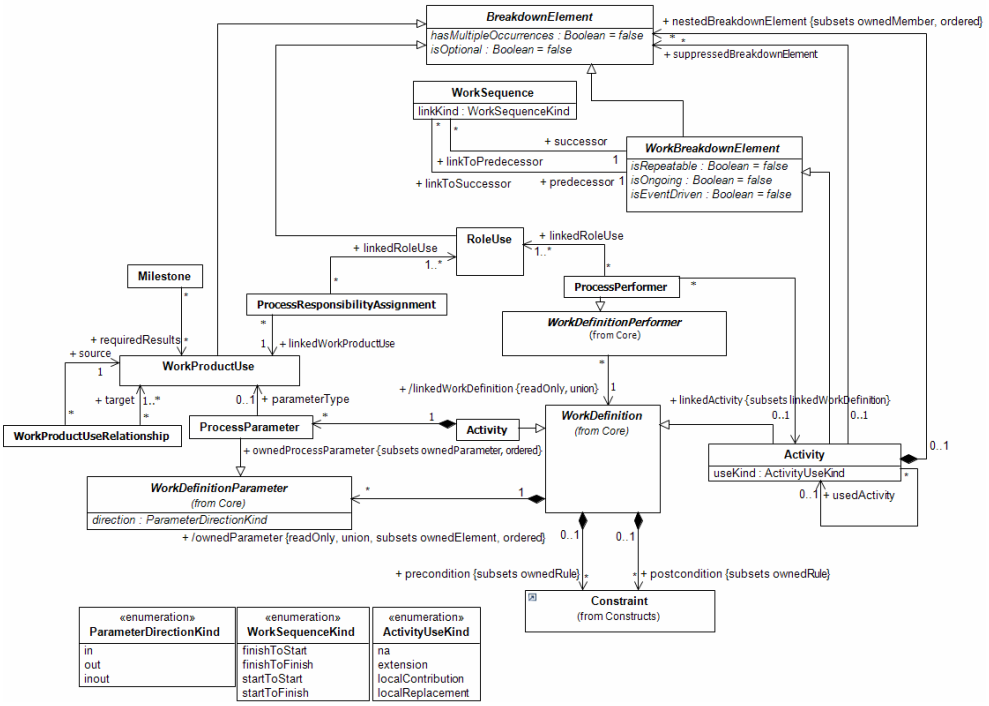


Figure 2.18- Aperçu du Package « Process Structure ».

Une unité de travail (*WorkDefinition*) est exécutée par un exécutant (*ProcessPerformer*). Elle peut avoir des pré/post conditions (*Constraint*). La classe *Activity* est une unité de travail concrète, elle admet des produits (*WorkProductUse*) en paramètres (*ProcessParameter*). Ces paramètres indiquent particulièrement la nature du *WorkProduct* qu'il soit en entrée ou en sortie. *WorkProductUse* est la correspondance de la classe *WorkProductDefinition* dans *Method Content*. Comme dans le méta-modèle de *Method Content*, un rôle possède des responsabilités sur un produit (*ProcessResponsabilityAssignment*), et un produit peut avoir des relations avec d'autres produits (*WorkProductUseRelationship*). Une borne (*Milestone*) modélise un événement significatif dans le processus ou à un jalon particulier.

La classe abstraite centrale pour définir des structures est *BreakdownÉlément* qui peut être spécialisée par *WorkProductUse*, *RoleUse* ou *WorkBreakdownÉlément qui précise que ces éléments peuvent être décomposés*. *WorkBreakdownÉlément* représente une unité de travail comme une activité (*Activity*) ou une borne (*Milestone*). Il est possible de définir des séquences entre *WorkBreakdownÉlément* avec la classe *WorkSequence*, l'attribut *linkKind* fait référence à la classe énumérée *WorkSequenceKind* qui définit quatre types de séquence entre deux *WorkBreakdownÉlément du genre (finishToStart, finishToFinish, startToStart, startToFinish)* . Une activité peut contenir des *BreakdownÉlément*, des activités peuvent donc être elles-mêmes composées d'activités ou de bornes, ou uniquement de produits ou de rôles, ceci dans le but de permettre le maximum de flexibilité.

d) Package contenu géré (Managed Content)

Le package Contenu géré définit les concepts fondamentaux de gestion des descriptions

textuelles (souvent en langage naturel) pour les éléments de contenu de processus et de méthode. Les processus de développement sont souvent représentés sous forme de modèles mais également sous forme de documents et de descriptions informelles. Pour la plupart des méthodes de développement, la documentation textuelle est aussi importante que la modélisation de la méthode elle-même.

Il introduit la classe abstraite *DescribableÉlément* qui, par le biais de fusions de packages, sert de super classe pour les éléments de processus définis dans le package de méta-modèle *Process Structure*, ainsi que pour l'élément de contenu de méthode dans le package de méta-modèle *Method Content*. A titre d'exemple, de cette classe peuvent être dérivés les classes ou les stéréotypes *Category* et *Guidance*. Parmi les guidances, on peut citer en particulier la classe *Metric*.

Une catégorie sert à regrouper un ensemble d'éléments descriptibles en fonction de critères définis par l'utilisateur. Une guidance fournit des informations supplémentaires liées aux éléments descriptibles. Ces informations peuvent être, entre autres, des métriques pour mesurer l'utilisation des ces éléments descriptibles.

e) Package comportement du processus (Process Behavior)

Ce package ne présente pas ses propres modèles comportementaux mais établit des liens avec le méta-modèle UML 2 pour définir les processus sous forme de diagrammes d'activités ou les produits sous forme de diagrammes états-transitions. Ce mécanisme est autorisé, car SPEM 2.0 est défini comme un profil d'UML 2.0 et par conséquent pouvant utiliser les éléments de sa superstructure. Ce package permet également d'utiliser le formalisme des diagrammes BPMN (2.4.3) proposés pour la modélisation des processus métier conformément au méta-modèle BPDM (*Busines Process Définition Méta-model*).

Ce package permet, d'une part, la réutilisation de processus logiciels existants qui ne sont pas conformes à SPEM, et d'autre part, une grande flexibilité en permettant à l'utilisateur SPEM de choisir une approche de modélisation de comportement qui lui convient.

f)Package processus avec method (Process with Method)

Ce package définit des processus logiciels en intégrant des instances du package *Process Structure* avec des instances du package *Method Content*. Il permet de décrire un processus logiciel structuré qui respecte les concepts du package « *Process Structure* », et en même temps, suit une méthode de développement définie dans le package « *Method Content* », plaçant le processus logiciel dans un contexte particulier qui respecte une méthode ou un cycle de vie particulier.

Aussi, il spécifie les liens entre les éléments des packages *Method Content* et *Process structure*, pour réutiliser les opérations à réaliser dans les structures temporelles spécifiques au processus défini. Il introduit la classe principale *PackageableÉlément* qui peut être spécialisée par deux classes secondaires : *MethodContentPAckage* et *ProcessStructurePackage*.

g) Package plugin des méthodes (Method Plugin)

Le package *Method Plugin* définit les fonctionnalités de gestion de bibliothèques entières de contenu et de *processus* de *méthode*. Il répond au problème de mise à l'échelle vers de grandes bibliothèques de méthodes (MethodLibrary) en définissant des plugins de méthode (MethodPlugin) et des configurations de méthode (MethodConfiguration). En d'autre termes, il permet d'étendre et de personnaliser des instances de *Method Content* et *Process Structure*

sans les modifier directement, mais grâce à un système *d'extension* et de *variabilité*, le plugin en lui-même.

Une *MethodConfiguration* est un ensemble de *MethodPlugin*. Une *MethodPlugin* décrit une association entre un élément *Method Content* et un élément *Process Structure*. Elle décrit aussi les changements à opérer sur ces éléments pour pouvoir les réutiliser et sans les modifier directement. Aussi, *MethodPlugin* intègre la réutilisation à base de composants en définissant des classes décrivant *certains* concepts architecturaux tels que : *ProcessComponent*, *WorkProductPort* et *WorkProductPortConnector*. Un composant processus est différent d'un processus dans le sens qu'il est composable et n'a aucune existence autrement. Un processus par contre peut être complet dans son contexte.

2.5.3 Restriction et extension

Cette section peut être considérée comme une synthèse sur le Méta-modèle SPEM et son utilisation dans la modélisation des processus logiciels. Elle présente quelques scénarios pour une utilisation restreinte de SPEM ainsi que quelques mécanismes d'extensions et orientations dans des usages particuliers qu'il ne couvre pas à l'origine.

2.5.3.1 Scénario d'utilisation

L'utilisation de SPEM pour la modélisation des processus de développement ne requiert pas toujours l'utilisation complète de tous les packages qui le composent. Ce standard (OMG-SPEM, 2008) précise quelques restrictions d'utilisation en présentant quelques scénarios qui limitent le nombre de package SPEM à mettre en œuvre. Cette limitation à pour objectif de limiter les ressources et accroître ainsi la possibilité de réutilisation dans certains cas particuliers. Le nombre important de classes dans les packages SPEM est souvent décourageant pour une utilisation complète.

A mesure que les packages de méta-modèles fournissent progressivement plus de fonctionnalités le long de leurs dépendances de fusion, de nombreuses autres combinaisons de packages de méta-modèles sont possibles et pourraient apporter de la valeur à d'autres fins. Par exemple, on pourrait choisir d'implémenter les combinaisons suivantes ci-dessous[1] :

- *Process Structure* et *Process Behavior* : un utilisateur peut choisir de fournir une solution de modélisation pure pour SPEM 2.0 ou une solution dans laquelle la capacité de l'infrastructure UML 2 à attâcher des commentaires à chaque élément de modèle est suffisante. Le responsable de la mise en œuvre ne réaliserait donc pas le package de contenu géré qui fournit une documentation supplémentaire ainsi que des capacités de catégorisation de contenu.
- *Process Structure* et *Managed Content :* Si un utilisateur souhaite se concentrer uniquement sur les représentations de la structure de répartition des processus, mais souhaite documenter et publier les modèles de processus avec une documentation textuelle, l'utilisateur peut choisir d'implémenter ces deux packages de méta-modèles uniquement.
- *Process Structure, Process Behavior* et *Method Plugin*: Si l'objectif d'un utilisateur est de fournir des bibliothèques de modèles de processus réutilisables sans documenter ces modèles de processus, il peut choisir de

[1] Toutes ces combinaisons devraient inclure le package Core. Il n'a donc pas été explicitement répertorié ci-dessous.

combiner les packages *ProcessStructure* et *Behavior* avec le package *MethodPlugin*. Cela permettra à l'utilisateur de définir des bibliothèques de processus réutilisables ainsi que des extensions de processus dynamiques utilisant la variabilité et d'organiser ces processus et extensions dans des plug-ins configurables.

- *Process with Methods*, *Process Structure*, *Method Content* et *Managed Content:* Cette combinaison serait intéressante pour un utilisateur qui pourrait vouloir fournir une réalisation à petite échelle de SPEM 2.0. Il peut être intéressé par l'utilisation complète de la séparation du contenu de la méthode des processus, mais n'a pas besoin des capacités de variabilité et de gestion des bibliothèques de méthodes, des plugins de méthode et des configurations définies dans le package *MethodPlugin*.

2.5.3.2 Mécanisme d'extension

Le principal reproche fait à SPEM est le nombre important de classes qui le composent pour rendre sa compréhension délicate (Combemale et al., 2006; Shaked & Reich, 2021). A chaque concept de base du processus logiciel (Unité de travail, Produit de travail et Rôle) est associé à un nombre important d'éléments UML qui le décrivent selon plusieurs facettes et qui sont distribués à travers les différents packages.

La Figure 2.19 (Aoussat, 2012) synthétise les éléments ou les classes les plus importantes qui caractérisent chacun des packages de SPEM 2.0. Elle montre aussi le chemin de leurs liaisons et extensions en définissant les différents stéréotypes associés. Cette notion d'extension traduit les relations de base définies par la Figure 2.15.

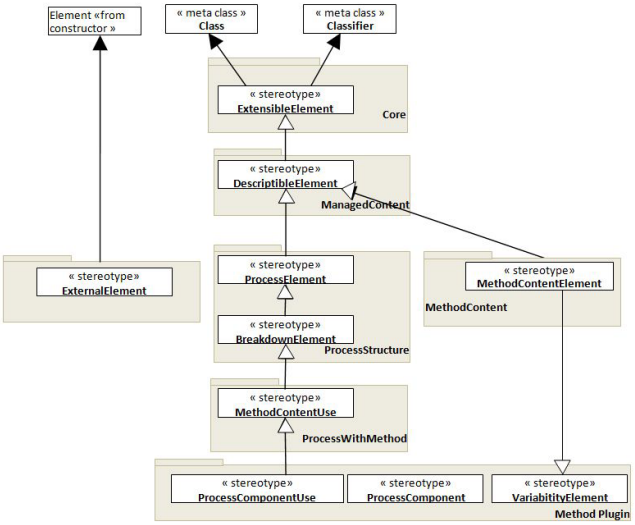


Figure 2.19- Synthèse des stéréotypes de base et leurs relations.

L'élément principal dans la package « *Core* » est le stéréotype généraliste « *ExtensibleÉlément* ». Ce dernier est étendu par le stéréotype « *DescribableÉlément* » du package « *ManagedContent* » qui à son tour peut être étendu par les stéréotypes « *ProcessÉlément* » du package « *ProcessStructure* » et « *MethodContentÉlément* » du package « *MethodContent* ». Du même package, le stéréotype « *BreakDownÉlément* » étend le stéréotype « *ProcessÉlément* » et il peut être étendu à son tour par le stéréotype « *MethodContentUse* » du package « *ProcessWithMethod* ». Le package « *MethodPlugin* » est le dernier niveau de la hiérarchie, il est constitué de trois stéréotypes de base

« *ProcessComponentUse* », « *ProcessComponent* » pour la réutilisation et « *VariabilityÉlément* ».

Outre l'organisation de SPEM en packages et en extensions entre packages, les classes de SPEM sont bien agencées. Selon (Aoussat, 2012), leur organisation respecte les règles suivantes :

- Chaque concept de base du processus logiciel (unité de travail, produit de travail, rôle, relation entre ces concepts) a une classe correspondante dans chaque package SPEM,
- Chaque classe abstraite du package est la classe génératrice des classes décrivant les concepts de base du processus logiciel (selon la vision du package),
- Chaque package (qui constitue un méta-modèle) possède une classe abstraite qui décrit le comportement commun des éléments de ce package, excepté le package « *MethodPlugin* » qui n'a pas de classe générique.

Ainsi, le package « *MethodPlugin* » est le seul point d'entrée de tout autre extension de SPEM pour prendre en charge de nouvelles fonctionnalités et orientations. Ceci étant fait pour préserver la cohérence du modèle global en inhibant l'accès au reste des packages dans des opérations d'extension.

Plusieurs travaux ont été consacrés à différentes extensions en proposant des stéréotypes précis pour répondre à certaines évolutions ou adaptations du profil SPEM pour des spécifications particulières. Ces extensions représentent dans l'ensemble une forme de variabilité du profil initial. Le Tableau 2.4 présente un échantillon de quelques travaux pour l'extension du profil SPEM dans des domaines variés.

Tableau 2.4- Quelques extensions de SPEM.

Extensions	Appellations	Objet d'extension	Motivations
(Khemissa et al., 2012)		Spécification des « guidances » adaptatives pour la modélisation des processus.	Permettre des éléments d'interventions d'assistance spécifiques (guidage correctif, constructif et automatique)
(Diaw et al., 2017)	SPEM4MDE	Spécification d'éléments pour le raffinement des concepts SPEM pour définir les processus dirigés par le modèle	Permettre la prise en charge des processus d'ingénierie dirigée par les modèles (MDE).
(Oliveira Junior et al., 2013)	SMartySPEM	Spécification d'éléments pour représenter les variabilités dans les lignes de processus logiciels (SPrL) pour la gestion de la variabilité.	Prise en charge de la variabilité dans les lignes de processus pour améliorer la mise en place de processus personnalisés pour un domaine spécifique.
(Aoussat et al., 2014)		Spécification des concepts architecturaux pour la modélisation de processus logiciels basés sur des	Permettre la réutilisation des processus logiciels à travers les concepts de composants et

		architectures logicielles.	connecteurs softwares.
(Gallina et al., 2014)	S-TunExSPEM	Spécification des éléments relatifs à la sécurité dans les processus de développement des systèmes critiques.	Permettre aux ingénieurs procédés et responsables sécurité de modéliser et d'échanger des processus orientés sécurité
(Alajrami et al., 2016)	EXE-SPEM	Spécification d'éléments pour la prise en charge de la création de modèles de processus exécutables sur le cloud.	Permettre l'exécution de processus logiciels dans le cloud.
(Abboud et al., 2016)	SArEM	Spécification des éléments relatifs au processus d'extraction d'architectures à partir du code.	Reverse engineering pour maintenir les systèmes et surmonter l'usure logicielle.

Dans le chapitre 4 nous proposons notre propre extension du méta-modèle SPEM 2.0 pour supporter la charge sémantique des architectures logiciels à base de connecteur orienté processus « Cap ». Nous présentons les différents stéréotypes ajoutés pour les différents éléments structurels et comportementaux.

2.6 Conclusion

Nous avons consacré ce chapitre à présenter les différents éléments, concepts, langages et outils qui ont trait aux processus logiciels. Nous discutons de cette thématique car la notion de processus représente le cœur même de notre proposition. En effet, nous considérons la communication dans les architectures logicielles comme un processus complexe et non pas de simples connexions points à points.

Pour cela, nous avons étudié la notion de processus, son origine et ses différentes définitions. Nous avons essayé de distinguer les différents concepts souvent ambigus en insistant sur les notions de métier, processus et produit et leurs relations ainsi que les notions de conception et d'architecture.

Dans ce chapitre, il a été discuté la notion de processus logiciels en particulier ainsi que les langages de description des processus. Un comparatif à été dressé entre les standards de représentation. Aussi, une section a été réservée au standard SPEM pour la modélisation des processus logiciel proposé par l'OMG. Une attention particulière à été donnée pour éclaircir les concepts les plus importants dans cette notation. Cette étude a été assortie par un ensemble de travaux pour l'extension du modèle SPEM pour supporter de nouvelles contributions.

CHAPITRE

3 Méta-modélisation et Transformation de modèles

3.1 Introduction

Dans ce chapitre, nous allons aborder les techniques et les concepts que nous utilisons dans la suite de thèse pour élaborer nos propositions. Nous utilisons deux concepts clefs dans les chapitres 3 et 4. Ces concepts, souvent liés, formes un binôme commun dans de très nombreuses approches et théories : La méta-modélisation et la transformation de modèles. Aussi, comme notre approche est basée sur l' « abstraction » de la communication, on doit donc faire appel systématiquement au concept de méta-modélisation pour décrire le modèle proposé.

Comme la modélisation est l'opération de création de modèles, la méta-modélisation est par analogie l'opération de création de méta-modèles. En d'autres termes, la méta-modélisation est l'opération de création des langages pour la manipulation de modèles. Le terme « méta » dans notre contexte à un double sens : répétition du même mot comme (modèle du modèle) ou (langage du langage) en termes de théorie de langage ou en termes d'abstraction pour le changement de niveau de modélisation.

Les transformations de modèles sont une nouvelle discipline souvent associée à la méta-modélisation. Avec la notion de modèle, la transformation de modèle forme la paire du domaine émergent de l'ingénierie des modèles ou en anglais « *Model Driven Engineering* » (MDE) (Bruel et al., 2020; Czarnecki & Helsen, 2006; Whittle et al., 2013). La principale motivation de l'ingénierie est ce qu'on appelle la notion de modèle « exécutable » où le code n'est plus une préoccupation centrale dans un monde où tout est modèle. L'objectif ultime et lointain du génie logiciel étant de limiter au maximum l'intervention humaine dans le cycle de développement en automatisation le maximum des phases dont certainement celle de codage à partir de l'architecture du système.

La première partie de ce chapitre sera consacrée à la méta-modélisation. Nous étudierons les concepts de modèle, de méta-modèle, de MOF, nous parlerons à titre de sommaire de l'approche MDA (OMG-MDA, 2003) et nous décrivons enfin la méta-

modélisation des architectures logicielles et la méta-modélisation des processus logiciels. La deuxième partie, quant à elle, sera réservée au concept de transformation de modèle en définissant les principes, les règles et les différentes approches. Nous présentons également l'une d'elles à savoir les transformations de graphes qui seront utilisées dans le cadre de cette thèse.

3.2 Méta-modélisation

Il nous semble que la meilleure manière de définir ce concept est de reprendre les propos de Marvin Minsky (Minsky, 1965), il disait : « Pour un observateur B, un objet M* est un modèle d'un objet M dans la mesure que B peut utiliser M* pour répondre aux questions qu'il se pose au sujet de M ». Dans ce cas, B utilisera le modèle M * comme support de raisonnement. L'action de modélisation permet donc de passer de l'objet M vers le modèle M* et ce à travers un principe d'abstraction visant à simplifier la représentation de l'objet à modéliser.

De la même manière, il est possible de poursuivre ce travail d'abstraction pour créer des méta-modèles. La création d'un méta-modèle revient à construire une représentation d'un modèle, toujours dans le but de fournir un support de raisonnement. L'action de méta-modélisation permet donc de passer d'un modèle M* vers le méta-modèle M**. L'observateur B peut également utiliser M** pour répondre aux questions qu'il se pose sur M*.

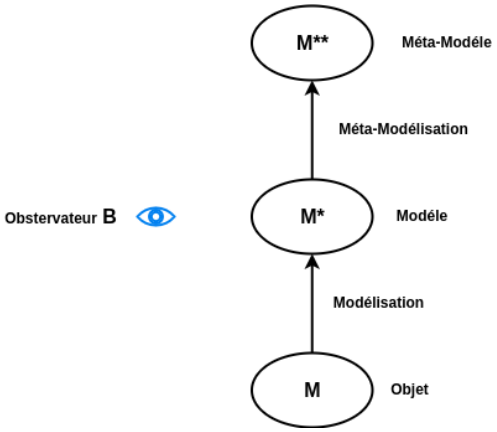


Figure 3.1-Modélisation et Méta-modélisation

La Figure 3.1 (adaptée d'(Abdelkrim Amirat, 2010)) décrit ce phénomène d'abstraction et de méta modélisation selon la vision de Minsky, elle reflète les relations qui lient et les niveaux qui séparent les concepts d'objet, modèle et méta-modèle.

On peut aussi prendre un autre exemple dans le monde des langages beaucoup plus connu que le monde des modèles : On peut parler de football ou de cinéma en français. Dans ce cas nous avons une langue ou un « langage » et un « sujet ». Si maintenant le sujet de la discussion est la langue française elle-même, alors nous ne sommes plus au même niveau. Nous devrons donc passer à un niveau d'abstraction plus élevé pour que le français (ou toute autre langue) soit un langage et non pas du football ou du cinéma.

Tout comme un programme, un modèle destiné à être traité par une machine ne doit pas avoir une interprétation ambiguë, et doit donc être exprimé en respectant des règles bien définies. Il s'agit là de la démarche de méta-modélisation qui s'attâche à définir des formalismes pour les langages de modélisation. Une fois le formalisme défini, on peut garantir que tout modèle conforme à ce formalisme pourra être traité correctement.

L’activité de méta-modélisation consiste alors à représenter les langages de modélisation par le biais des modèles, a permis l’émergence de plusieurs DSMLs (*Domain Specific Modelling Languages*) (Kelly & Tolvanen, 2008) ou littéralement (langages de modélisation spécifiques à un domaine).

A l’image des langages de programmation, un DSML est défini par :

- sa syntaxe abstraite qui décrit les constructions du langage et leurs relations ;
- sa syntaxe concrète qui décrit le formalisme permettant à l’utilisateur de manipuler les constructions du langage ;
- sa sémantique qui décrit la dynamique des constructions du langage.

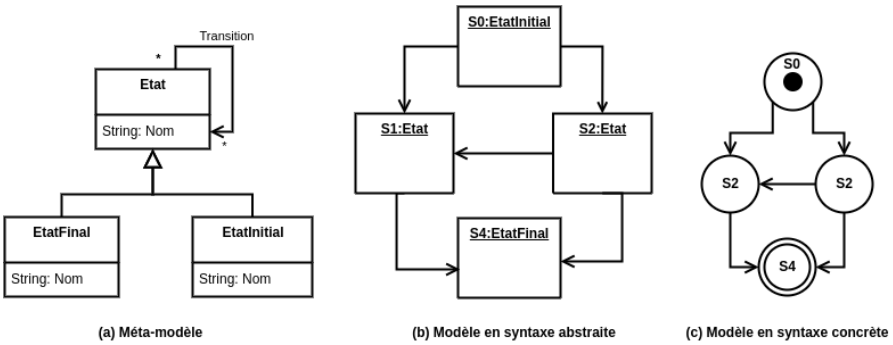


Figure 3.2-Syntaxes des modèles

La Figure 3.2 montre un exemple de modélisation d’un domaine spécifique celui des automates d’états finis. La figure (a) montre un méta-modèle simplifié de ces automates, l’automate est une suite d’états reliés par des transitions. L’état initial et l’état final sont des états particuliers. Il est exprimé obligatoirement en diagramme de classe du langage UML. La figure (b) montre une instance d’un automate de 4 états exprimé en syntaxe abstraite sous forme d’un diagramme objet UML. La figure (c) exprime le même automate en syntaxe concrète après avoir défini les représentations visuelles de chaque classe du méta-modèle. La sémantique est définie à travers une ensemble de contraintes qui, par exemple, interdisent qu’un état final ait des transitions sortantes.

3.2.1 Notion de modèle

La notion de modèle et de méta-modèle à toujours eu différentes définitions selon plusieurs aspects.

Définition *1* *: « un modèle est une abstraction de quelque chose dans le but de le comprendre avant de le construire »* *Rumbaugh et al. (1991)* .

Définition 2 *: « un* modèle *est une simplification d'un système construit avec un objectif en tête. Le modèle devrait être capable de répondre aux questions à la place du système réel »* (Bézivin & Gerbé, 2001).

Comme exemple, on peut dire que la carte géographique de l’Algérie représente le modèle du pays Algérie. A la question de savoir les pays limitrophes, la carte de l’Algérie devrait donner une réponse identique à la réalité.

À partir de ces définitions, il est clair qu'un modèle est utile s'il aide à mieux comprendre le système et à décider des actions appropriées qui doivent être prises pour atteindre et maintenir l'objectif du système (Boussaïd et al., 2017).

3.2.2 Notion de méta-modèle

Au même titre qu'un modèle, un méta-modèle devrait avoir également des définitions plus ou moins variées en fonction des contextes et des disciplines. Une longue liste de définitions existe dans, nous retenons quelques une :

Définition 1 *: « un méta-modèle est la spécification explicite d'une abstraction (une simplification). Il utilise un langage spécifique pour exprimer cette abstraction »* (Bézivin & Gerbé, 2001).

Définition 2 *: « les méta-modèles peuvent être considérés comme des définitions de langages. Le préfixe « méta » est utilisé chaque fois qu'une opération de modélisation a lieu deux fois»* (Kühne, 2006).

Définition 3 « *le méta-modèle est un modèle d'un langage de modélisation. Le méta-modèle définit la structure, la sémantique et les contraintes pour une famille de modèles »* (Mellor et al., 2004).

Définition 4 *« Un méta-modèle est un modèle utilisé pour modéliser la modélisation elle-même. Le modèle MOF 2 est utilisé pour se modéliser soit même ainsi que pour modéliser d'autres modèles et d'autres méta-modèles comme UML 2 et CWM »* (OMG-MOF, 2003).

Les définitions suscitées indiquent différents points de vue pour considérer le concept de méta-modèle. De la notion d'abstraction et simplification, à la notion de langage des modèles, à la notion d'itération de l'opération de modélisation ...etc.

Pour reprendre l'exemple de la carte géographique, la légende de la carte représente le méta-modèle qui permet définir le sens des différents éléments de la carte pour être interprété sans ambiguïté. Sans cette légende, la carte géographique serait un simple dessin.

3.2.3 MOF 2.0

Méta-Object Facility (MOF) (OMG-MOF, 2003) est la réponse du consortium OMG pour définir le langage des méta-modèles pour le paradigme objet. Il définit les différents éléments et leurs sémantiques pour le langage de modélisation unifié (UML) (UML, 2017). Le but du MOF est de définir un langage unique et standard pour décrire des méta-modèles. Le MOF est une ontologie de représentation basée sur des graphes non orientés avec étiquetage des extrémités des arêtes.

MOF est le fondement de l'environnement standard de l'OMG où les modèles peuvent être exportés d'une application, importés dans une autre, transportés sur un réseau, stockés dans un référentiel puis récupérés, rendus dans différents formats (y compris XMI : le format standard basé sur XML d'OMG pour la transmission et le stockage du modèle), transformé et utilisé pour générer du code d'application. Ces fonctions ne sont pas limitées aux modèles structurels, ni même aux modèles définis en UML - les modèles comportementaux et les modèles de données participent également à cet environnement, et les langages de modélisation non-UML peuvent également participer, tant qu'ils sont basés sur MOF (OMG-MOF, 2003).

Comme précisé dans la définition 4, MOF s'auto-définit soit même (on utilise également la relation se décrit par soit même). MOF, soit même, est modélisé avec les éléments d'UML en utilisant des restrictions exprimées en termes de contraintes avec le langage OCL (*Object Constraint Language*) (OMG-OCL, 2015).

MOF est composé de deux packages : (i) EMOF (Essential MOF), conçu pour

correspondre aux capacités des langages de programmation orientés objet et des mappages vers XMI. (ii) CMOF (Complet MOF), il fournit toutes les capacités de méta-modélisation de MOF 2.

D'autres notations pour la méta-modélisation objet existent. Parmi elles on peut citer ECORE (Steinberg et al., 2008) ou Kerméta (Muller et al., 2005).

MOF 2 Core est basé sur d'autres spécifications OMG MOF, notamment les suivantes (dans cette liste, « modèle basé sur MOF » désigne tout modèle qui instancie un méta-modèle défini à l'aide de MOF, qui inclut les méta-modèles eux-mêmes (entre autres):

- XMI : pour échanger des modèles basés sur MOF en XML (OMG-XMI, 2014).
- QVT :Vues et transformations des requêtes QVT : MOF - pour transformer des modèles basés sur MOF (OMG-QVT, 2016).
- OCL : Object Constraint Language – OCL : pour spécifier des contraintes sur des modèles basés sur MOF (OMG-OCL, 2015).
- SPEM : Voir Chapitre 2
- CWM : Common Warehouse Métamodel : une spécification de l'OMG qui décrit un langage d'échange de métadonnées à travers un entrepôt de données (OMG-CWM, 2003).

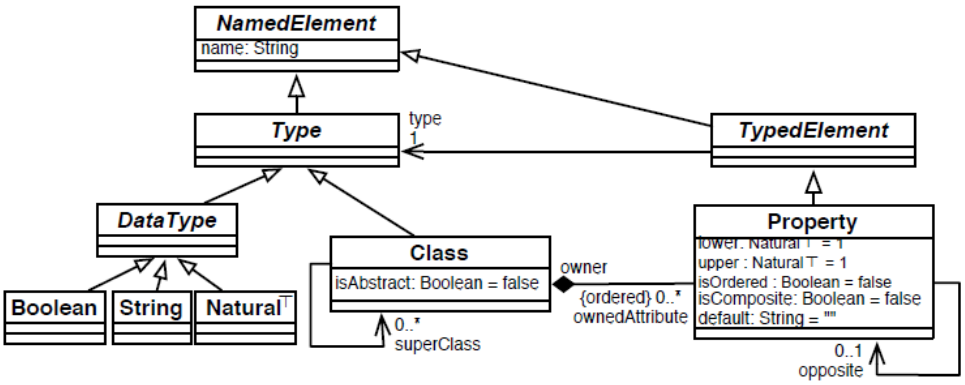


Figure 3.3- Portion MOF

La Figure 3.3 est une portion ou un extrait du MOF standard, elle définit une classe comme un type d'un élément nommé. Les associations et les attributs sont vus aussi comme des classes. Nous donnant un exemple concret de ces concepts plus loin (Figure 3.6).

3.2.4 Approche MDA

L'approche *MDA (Model Driven Archiecture)* (OMG-MDA, 2003) est également la réponse de l'OMG au concept de l'ingénierie des modèles (Boussaïd et al., 2017; Czarnecki & Helsen, 2006). Elle s'appuie sur deux concepts essentiels : la méta-modélisation et la transformation de modèle basée sur QVT en facilitant l'interopérabilité entre les différentes technologies ou les plateformes d'implémentation. Dans cette section nous nous intéressons uniquement à l'aspect méta-modélisation.

L'approche MDA est basée sur l'utilisation des modèles et essaye de répondre aux « *comment* », « *quand* », « *quoi* » et « *pourquoi* » modéliser. Elle favorise surtout la séparation des préoccupations entre le monde métier et le monde technologique. Elle identifie un ensemble de niveaux de modélisation et un ensemble de modèles associés.

L'approche MDA pose le principe d'indépendance vis-à-vis de la plateforme d'exécution comme un élément central. Elle promet de nombreux avantages parmi lesquels on peut citer :

- L'amélioration de la portabilité due à la séparation des connaissances métiers de la technologie spécifique d'implémentation,
- L'augmentation de la productivité due à la transformation automatique,
- L'amélioration de la qualité due à la réutilisation de patrons bien éprouvés avec les bonnes pratiques des transformations,
- L'amélioration de la maintenabilité due a une bonne séparation des préoccupations et une meilleure cohérence et traçabilités entre les modèles et le code.

Il en ressort que le couple (Qualité-Délais), qui demeure la raison d'être du génie logiciel, est garantie par l'approche MDA et ce en s'appuyant sur son idée de base et son mode opératoire, à savoir la séparation, donc indépendance, des préoccupations métiers et technologie des plateformes et par les transformations automatique de modèles.

3.2.4.1 Niveaux MDA

Les niveaux de modélisation est l'une des principales contributions de l'approche MDA. En effet, l'architecture dirigée par les modèles identifie quatre niveaux de modélisation ou d'abstraction ou (3+1) ainsi que quatre relations qui les lient (Bézivin & Gerbé, 2001).

Les différents niveaux d'abstraction MDA liés au concept de méta-modélisation sont résumés dans le Tableau 3.1.

Tableau 3.1- Niveaux de modélisation MDA

Concepts	Description	Commentaire
Méta-méta-modèle	Langage de spécification des méta-modèles	Le MOF, 1 Seul
Méta-modèle	Définition du langage utilisé pour exprimer le modèle	Le méta-modèle UML et autres méta-modèles : SPEM, CWM, sysML
Modèle	Abstraction du système	Modèles UML et autres Modèles
Système	Information et flux de contrôle d'un domaine	Utilisation variée de ces modèles

Au premier niveau M0 (ou instance), se trouve le système à étudier: il peut s'agir selon le cas d'un système d'information d'une entreprise, ou d'une tâche que l'on souhaite automatiser. Ce sont les informations réelles de l'utilisateur. M0 est une instance de M1.

Au second niveau M1 (ou modèle), se trouve le modèle qui décrit certains aspects du système que l'on veut étudier: il peut s'agir d'un diagramme de classes UML, PIM, PSM, d'un modèle conceptuel de traitement MERISE, ou de tout schéma qui représente une vue abstraite des objets modélisés.

Au troisième niveau M2 (ou méta-modèle) se trouve le langage de modélisation: par exemple les définitions d'un modèle de conception de données MERISE, d'un diagramme d'objet UML ou d'une spécification dans le langage B. C'est ce langage qui doit faire l'objet d'une spécification formelle. C'est à ce niveau qu'on définisse les structures internes de tout

modèle ou profil UML[1].

Au quatrième niveau M3 (ou méta méta-modèle), se trouve le langage qui doit être assez générique pour définir les différents langages de modélisation existants et assez précis pour exprimer les règles que chaque langage doit respecter pour pouvoir être traité automatiquement. Il est composé d'une unique entité qui s'appelle le MOF. Le MOF permet de décrire la structure des méta-modèles, d'étendre ou de modifier les méta-modèles existants. Le MOF est réflexif, il se décrit lui-même.

La hiérarchie de ces différents modèles est représentée par la Figure 3.4 où on voit apparaître le mode réel formé par le niveau M0 qui conserve la relation « ReprésenterPar » avec le monde des modèles qui, lui, est formé de 3 autres niveaux de modèles : M1(modèle), M2 (méta-modèle) et M3 (méta méta modèle) liés entre eux par la relation « ConformeA », le niveau M3 est conforme à soi même. La forme pyramidale est justifiée par le fait que l'on réduise significativement les concepts en partant du niveau M0 vers le niveau M3 ce qui est un objectif en soi car il facilite et simplifie leurs manipulations. Le monde réel a le maximum de détails, le MOF a le minimum de concepts.

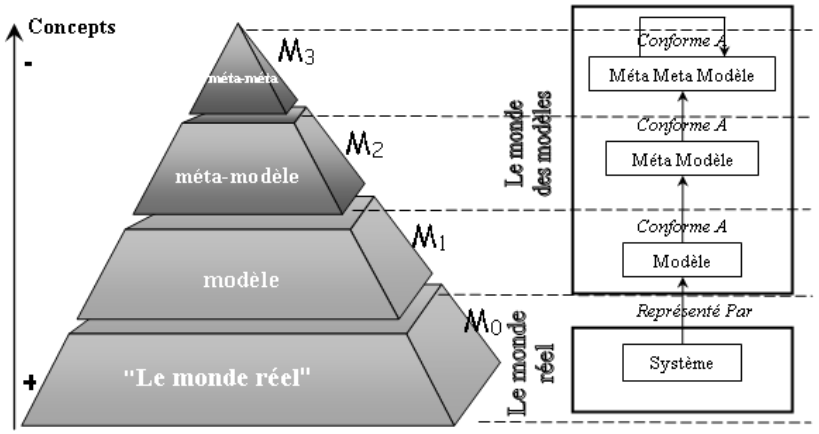


Figure 3.4- Représentation pyramidale de modèles de l'organisation 3+1 (MDA).

Des exemples applicatifs et illustratifs de la notion de méta-modélisation sont montrés dans la Figure 3.5. Il est nécessaire de remarquer l'analogie entre (a) et (b). La Figure 3.5 (a) représente la hiérarchisation de niveaux MDA[2] définie pour les langages de modèles alors que la Figure 3.5 (b) représente la hiérarchisation classique pour les langages textuels. En fait, la démarche MDA n'a rien inventé de neuf en termes de nombre de niveaux de modélisation. Elle a juste copié ou adapté ce qui jadis été connu et appliqué pour les langages textuels aux langages des modèles.

[1] Un profil UML permet d'adapter le langage UML à un domaine qu'il ne pouvait couvrir correctement.
[2] Aussi appelé Modèle de Bezivin

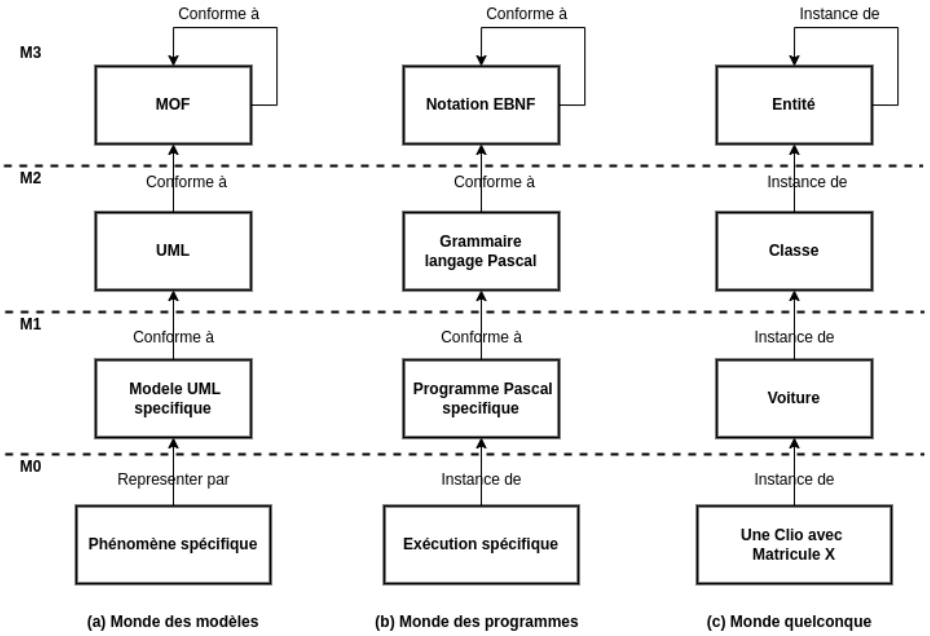


Figure 3.5- Interprétation de la pile de modélisation multi-niveau de l'OMG

En effet, nous pensons que cette hiérarchisation est connue depuis la naissance de langages de programmation. Une exécution d'un programme (M0) avec des valeurs bien données était toujours considérée comme une instance d'un programme qui représente un niveau supérieur. On peut avoir plusieurs exécutions. Les programmes ainsi écrits représentent le niveau M1, ils doivent être conformes au niveau M2. La grammaire (M2) est unique, elle doit être conforme à sa notation BNF[1] ou EBNF[2]. Cette notation (M3) est unique, elle est toujours conforme à elle-même. On dit aussi qu'elle se décrit par elle-même. Ce raisonnement est appliqué dans son intégralité dans le monde des modèles visuels. On considère que quatre niveaux de modélisation sont nécessaires et suffisants pour décrire tout langage de modélisation.

Pour définir la relation de conformité, on peut dire qu'un modèle est dit conforme à un méta-modèle si : (i) tous les éléments du modèle sont des instances du méta-modèle. (ii) les contraintes exprimées sur le méta-modèle sont respectées dans le modèle. La Figure 3.6 essaye de montrer un exemple concret de modélisation sur les différents niveaux en reprenant la modélisation des automates d'états finis précédemment discutés (Figure 3.2).

[1] Backus-Naur form
[2] Extended BNF

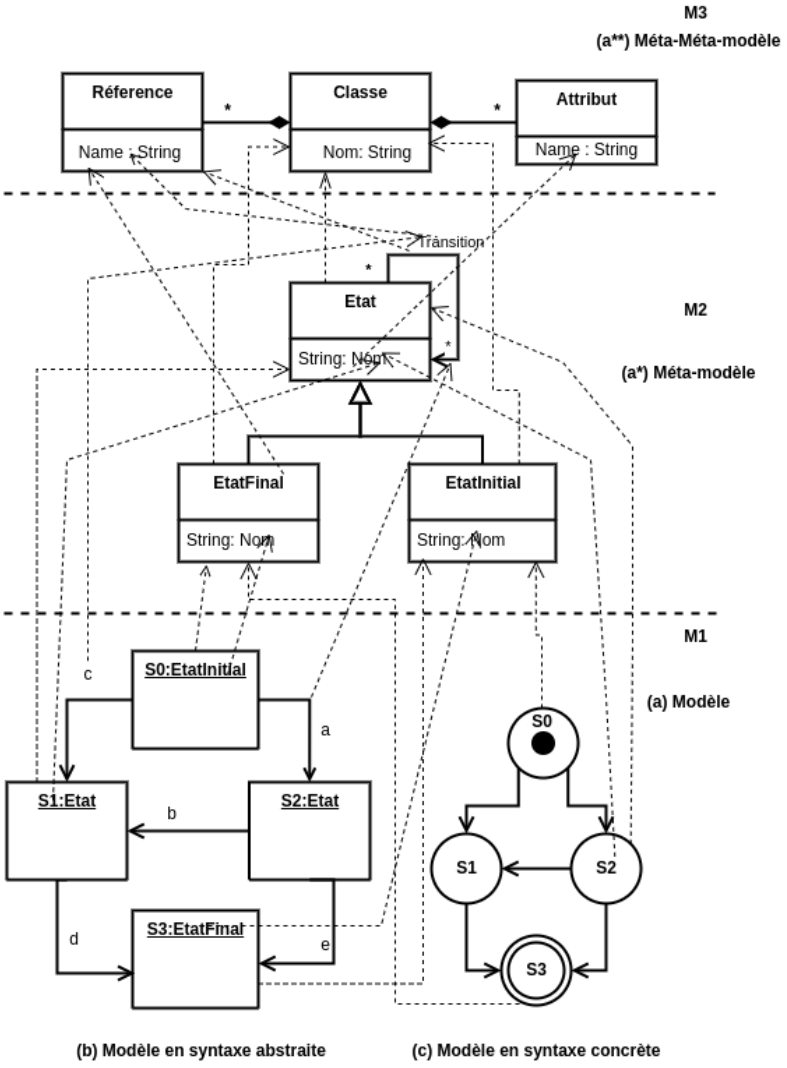


Figure 3.6- Méta-modélisation des automates d'états finis.

La Figure 3.6 montre le niveau des modèles (M1) avec les deux syntaxes, le niveau (M2) pour le méta-modèle et la nouveauté au niveau (M3) qui montre le méta-méta-modèle. Le niveau M3 contient le minimum de concepts : un diagramme de classe UML est en premier lieu formé de deux classes (la classe « Classe » et la classe « Association » ici « Reference ») Les flèches en pointillés représentent les relations de conformité des différents éléments entre les différents niveaux (de bas en haut). Le niveau M0 n'est pas représenté, c'est un simple déroulement de l'automate.

3.2.4.2 Modèles MDA

Le principe clé de la démarche MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, MDA préconise l'élaboration de modèles d'exigences (CIM : *Computation Independent Model*), d'analyse et de conception (PIM : *Platform Independent Model*) et de code (PSM : *Platform Specific Model*). L'objectif majeur de MDA est l'élaboration de modèles indépendants des détails techniques des plateformes d'exécution (J2EE, .Net, PHP ou autres), afin de permettre la génération automatique de la totalité du code des applications et d'obtenir un gain significatif de productivité (Blanc & Salvatori, 2011). Chaque plateforme doit avoir son propre modèle (PDM : *Platform Description Model*) (OMG-MDA, 2003).

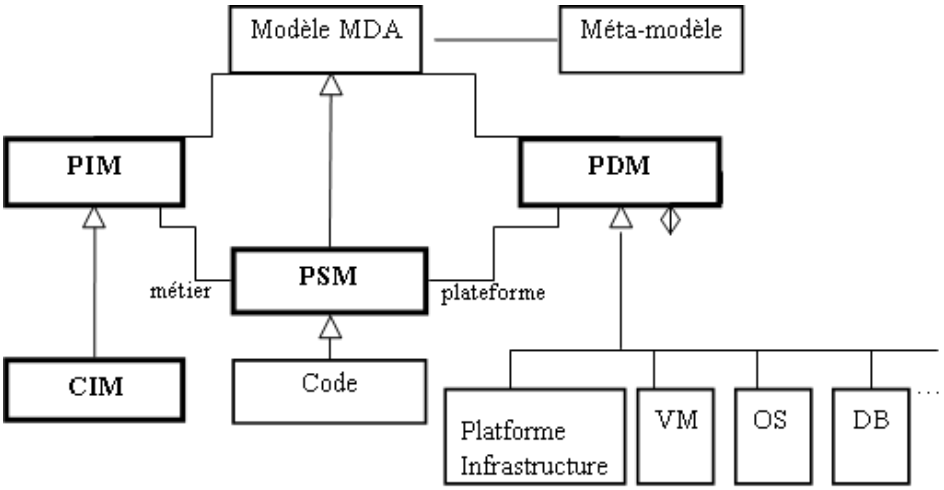


Figure 3.7- Relations entre les modèles MDA

La Figure 3.7 montre le diagramme de classe qui définit les relations entre les différents modèles de cette démarche où l'on remarque l'existence d'une seule doctrine « Tout est modèle ». Ces différents modèles se présentent comme suit :

a) Le modèle des besoins CIM

La première tâche à réaliser lors de la construction d'une nouvelle application est de *spécifier les besoins* (exigences) du client. L'objectif est de créer un modèle des besoins de la future application. En UML, un diagramme de cas d'utilisation peut être utilisé comme modèle des besoins Ce modèle doit définir les fonctionnalités de l'application et les autres entités avec lesquelles elle interagit. La création d'un modèle des besoins est d'une importance capitale. Cela permet d'exprimer clairement les liens de traçabilité avec les modèles qui seront construits dans les autres phases du cycle de développement de l'application, comme les modèles d'analyse et de conception (Blanc & Salvatori, 2011).

b) Le modèle d'analyse et de conception abstraite PIM

Le travail d'analyse et de conception peut commencer à partir du modèle des besoins. Ce travail utilise aussi un modèle. L'analyse et la conception sont les étapes où la modélisation est la plus présente, d'abord avec les méthodes Merise et Coad/Yourdon puis avec les méthodes objet Schlear et Mellor, OMT, OOSE et Booch. Ces méthodes proposent toutes leurs propres modèles. Aujourd'hui, le langage UML s'est imposé comme la référence pour réaliser tous les modèles d'analyse et de conception (Blanc & Salvatori, 2011).

c) Le modèle de code PSM

Le travail de génération de code peut commencer à partir des modèles d'analyse et de conception. Cette phase, la plus délicate du MDA, doit elle aussi utiliser des modèles. MDA considère que le code d'une application peut être facilement obtenu à partir de modèles de code. La différence principale entre un modèle de code et un modèle d'analyse et de conception réside dans le fait que le modèle de code est lié à une plate-forme d'exécution. Ces modèles de code sont appelés des PSM. Les modèles de code servent essentiellement à faciliter la génération de code à partir d'un modèle d'analyse et de conception. Ils contiennent toutes les informations nécessaires à l'exploitation d'une plate-forme d'exécution. Pour MDA, le code d'une application se résume à une suite de lignes textuelles, comme un fichier C++, alors qu'un modèle de code est plutôt une représentation structurée incluant les concepts de boucle, condition, instruction, composant, événement, etc. L'écriture de code à partir d'un modèle de code est donc une opération assez triviale (Blanc & Salvatori, 2011). Le PSM est souvent obtenu en faisant le tissage des PIMs avec les modèles de chaque plateforme (PDM)

suivant le modèle dit en « Y ».

d) Le modèle de plateforme PDM

Un PDM (*Platform Description Model*) contient des informations pour la transformation de modèles vers une plate-forme en particulier et il est spécifique de celle-ci. C'est un modèle de transformation qui va permettre le passage du PIM vers le PSM. Normalement, chaque fournisseur de plate-forme devrait le proposer. La classification de l'ensemble de ces modèles est donnée par la figure 2.6 suivante :

3.2.5 Méta-modélisation des architectures logicielle

L'étude des architectures et des ADLs nous a fait constater que la spécification d'une architecture peut passer aussi par plusieurs niveaux de modélisation. Les quatre niveaux de modélisation proposés par l'OMG sont initialement réservés au concept objet et non pas composant. Or, les architectures sont essentiellement basées sur les éléments architecturaux évoqués dans le chapitre 1. L'objectif essentiel couru est de permettre l'outillage et l'industrialisation qui doit rester conforme aux standards.

3.2.5.1 Méta-modélisation standard (ISO)

Suite à l'effort de normalisation (1.2.6) fait pour les architectures logicielles dans la norme ((IEEE, n.d.)) et le standard ISO définissant les différentes vues d'une architecture logicielle, il a été aussi fait un effort pour la description des architectures logicielles à travers les niveaux de modélisation.

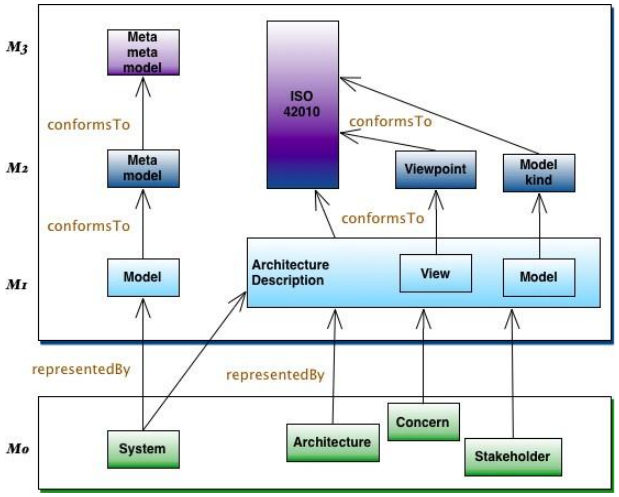


Figure 3.8- Modèle hiérarchique pour une description architecturale.

Sous une forme hiérarchique (Figure 3.8), on peut représenter les différentes positions des concepts architecturaux par rapport aux niveaux de modélisation. La partie gauche du diagramme représente le schéma original dit aussi de « Bézivin » (OMG-MDA, 2003). La partie droite montre l'alignement des constructions ISO/CEI/IEEE 42010 sur les différents niveaux.

Le *niveau M0* décrit le monde réel des systèmes, des architectures, de leurs parties prenantes et de leurs préoccupations.

Le *niveau M1* est le monde des modèles. Les modèles représentent des choses dans le monde réel. Les descriptions d'architecture sont des modèles, composés d'autres modèles: Vues (d'architecture) et Modèles (d'architecture), qui représentent des architectures. Le niveau M1 est distinct de M0. Cette distinction est utile pour éviter la confusion entre architectures et descriptions d'architecture.

Le *niveau M2* capture les conventions, ou règles, auxquelles les modèles du niveau M1 se conforment ou adhèrent. C'est le monde des méta-modèles. Dansla norme, les points de vue (architecture) et les types de modèle se trouvent à ce niveau. Le niveau M2 est distinct de M1. Souvent, les termes « *vue* » et « *point de vue* » sont utilisés de manière indifférente. Cela explique cette différence: lespoints de vue fournissent les conventions, les «légendes» des vues.

Le niveau M3 est le monde des méta-méta-modèles, où sont localisées les règles d'expression des conventions. La norme couvre M2 et M3: elle spécifie les conventions sur les descriptions d'architecture et les conventions sur laspécification de points de vue et de types de modèle.

3.2.5.2 Méta-modélisation par composant

La méta-modélisation telle qu'elle est définie plus haut tient compte beaucoup plus des vues et des intervenants dans les architectures logicielles. Aussi, les limites ne sont pas bien claires entre le niveau M2 et M2. UML est initialement orienté objet, il n'a reconnu le concept composant que dans sa version 2.0 en 2005 tout en restant assez primitif sur quelques éléments architecturaux comme le connecteur.

On aimerait alors avoir une hiérarchisation dédiée qui tient compte des particularités des architectures logicielles avec ses éléments architecturaux. Pour cette fin, nous reproduisant les travaux d'Adel Smeda (Smeda, 2006; Smeda et al., 2008) qui propose une contribution pour définir un méta-méta-modèle (MADL : *Méta Architecture Description Language*) pour les architectures logicielles. MADL est l'équivalent du MOF pour les architectures et les langages de description d'architecture. Cette méta-architecture a servi de support pour définir les architectures hiérarchiques (Amirat, 2010) ainsi que les méta-styles dans les travaux (Djibo et al., 2020; Hassan, 2018; Le Goaer, 2009) qui définissent les styles d'évolution dans les architectures logicielles.

L'objectif recherché dans la méta modélisation des architectures logicielles est de faciliter la manipulation, la réutilisation et l'évolution des architectures logicielles, ainsi que de permettre la transformation et la comparaison entre les ADLs (Smeda et al., 2008). Un acte de méta-modélisation a les mêmes objectifs qu'un acte de modélisation avec pour seule différence l'objet de la modélisation. Dans le cas d'un modèle réflexif, la méta-modélisation permet à un modèle de s'auto décrire : il est à la fois l'enjeu et le moyen de modélisation (Amirat, 2010).

Dans cette section, nous nous intéressons à la méta-modélisation par composants dont le résultat est une hiérarchie à différents niveaux d'architectures, qui passe de la définition d'une méta-méta-architecture à une application réelle à travers les différents niveaux intermédiaires. Les niveaux définis sont ceux du Tableau 3.2.

Tableau 3.2 - Hiérarchie Objet/Composant

	Modélisation par objets	Modélisation par composants
M3 – Niveau méta-méta-modèle	MOF	MADL
M2 – Niveau méta-modèle	UML, CWM, SPEM, etc.	ACME, COSA, ADL Fractal, C2,…
M1 – Niveau modèle	Modèles UML	Architectures
M0 – Niveau instance	Informations réelles (Objets)	Applications

Naturellement comme avec le MOF, MADL identifie quatre niveaux de modélisation dans les systèmes : le niveau méta- méta-architecture, le niveau méta-architecture, le niveau architecture et le niveau application. Ces niveaux sont schématisés sous forme d'une « *architecture pyramidale* » présentée dans la Figure 3.9, où chaque étage de la pyramide se conforme à l'étage immédiatementsupérieur (Abdelkrim Amirat, 2010; A Smeda, 2006).

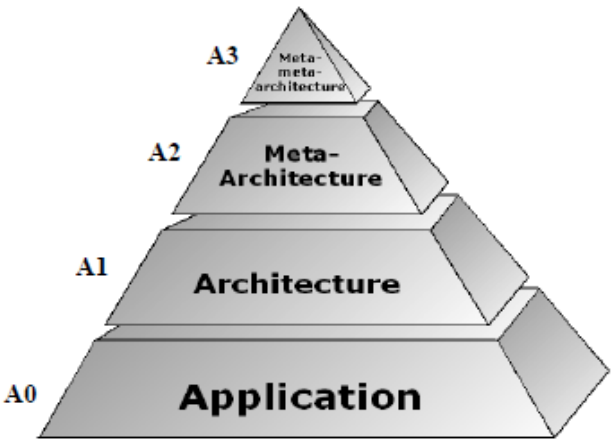


Figure 3.9- Les niveaux (3+1) pour les architectures.

*Le niveau méta-méta-architecture (*A3*)* : est le niveau le plus abstrait qui fournit les éléments minimaux de modélisation d'une architecture. Une méta-méta-architecture se conforme à elle-même (*i.e.*, s'auto définie). Les concepts de base d'un méta ADL sont définis à ce niveau.

*Le niveau méta-architecture (*A2) : fournit les éléments de modélisation de base pour un langage de description d'architecture (ADL) : composant, connecteur, configuration, port, rôle, etc. Ces concepts de base permettent de définir différentes architectures. Les méta-architectures se conforment au méta-méta-architecture fixée à l'avance. Dans le cadre d'une relation de conformité, chaque élément de (A2) est associé à un élément de (A3).

Le niveau architecture (A1) : à ce niveau, plusieurs types de composants, de connecteurs et de configurations sont décrits. Les architectures se conforment aux méta-architectures (ADLs), donc chaque élément de (A1) est associé à un élément de (A2).

Le niveau application (A0) : est le niveau où les unités d'exécution sont localisées. Une application est vue comme un ensemble de composants, de connecteurs et de configurations. Les applications sont conformes au niveau architecture. Chaque élément de (A0) est associéà un élément de (A1).

Le modèle MADL (Smeda, 2006) est organisé en trois paquetages : méta-méta-architecture(M^2A), méta-architecture(MA), et architecture (A). Le paquetage M^2A traduit le fait que toute architecturedoit dériver d'une MA. Le paquetage MA classifie et définit des architectures et il contient les méta-éléments d'architectures (méta-composant, méta-connecteurs et méta-interface) (

Figure 3.10 (Hassan, 2018)). Le paquetage architecture hérite de composant pour respecter le principe « tout est un composant » dans MADL. Il est guidé par les principes suivants :

- MADL est orienté composant dans le sens où tout est composant (tous les éléments sont des sous-types de la classe abstraite *Composant*).
- Chaque architecture doit être explicitement dérivée d'une méta-architecture.
- Chaque architecture est une instance de son architecture supérieure, à l'exception de la méta-méta-architecture, qui est une instance d'elle-même.
- Les dépendances entre les architectures et les éléments sont basées sur les dépendances utilisées dans le modèle MOF, à savoir : l'instanciation, l'héritage et la composition.

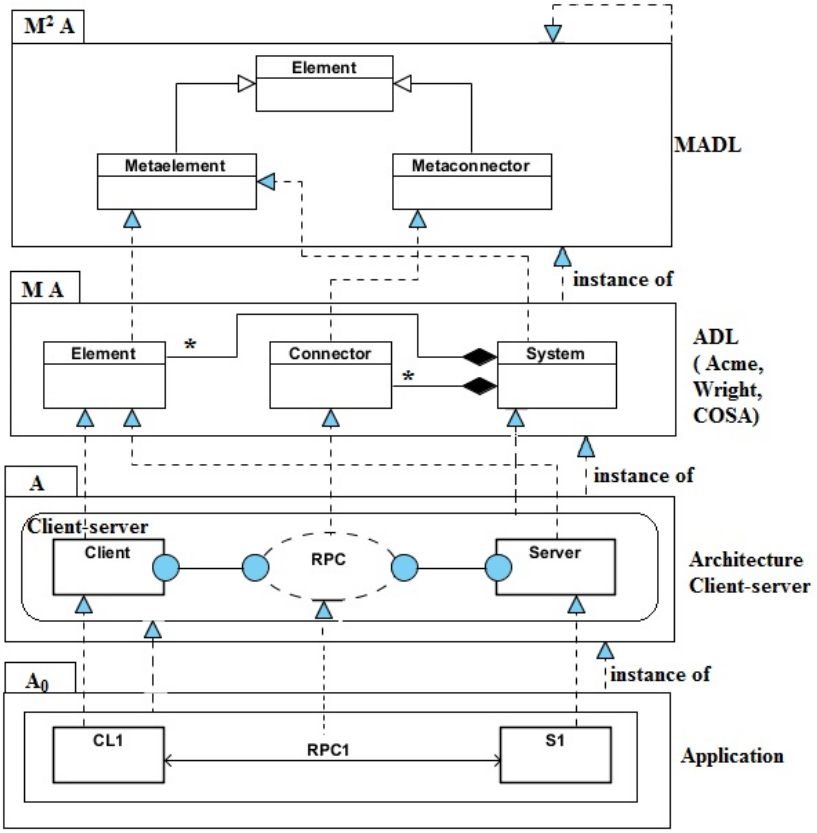


Figure 3.10- Niveaux de modélisation des architectures logicielles.

3.2.6 Méta-modélisation des processus logiciels

Comme nous avons proposé une méta-modélisation, nous devons aussi discuter de la modélisation de processus. Ceci va nous permettre certainement de réunir et unifier les deux concepts (avec les mêmes standards) pour réussir notre principale proposition dans le cadre de cette thèse qui reste la fusion des concepts d'architecture logicielle et de processus logiciel (Chapitre 4).

Dans sa dynamique de normalisation, l'OMG à également soutenu un effort de méta-modélisation en matière de processus logiciel (Figure 3.11). Jusque là, cette organisation à consacré tous ses efforts pour normaliser les produits (logiciels, documents, données, modèles, etc). En bref, on peut résumer le rapport entre les deux concepts comme suit: le processus est l'enchaînement d'étapes pour obtenir un produit, un produit est le résultat d'un processus (Figure 3.12).

La méta-modélisation des processus permet donc de conclure l'effort de normalisation de l'OMG du produit jusqu'au processus afin de standardiser les concepts et définir des langages de leurs manipulations. Cette normalisation va certainement favoriser l'industrialisation des processus et accroître leur prise en charge par les outils. Le souci majeur de la méta-modélisation des processus est de permettre leur description, manipulation et surtout leur réutilisation. Osterweil (Osterweil, 2011) dit que «Le processus est aussi un logiciel ». L'effort de l'OMG à aboutit à SPEM (OMG-SPEM, 2008) qui, au même titre qu'UML pour le produit, représente le méta-modèle pour la définition des langages de processus qui reste conforme au même MOF.

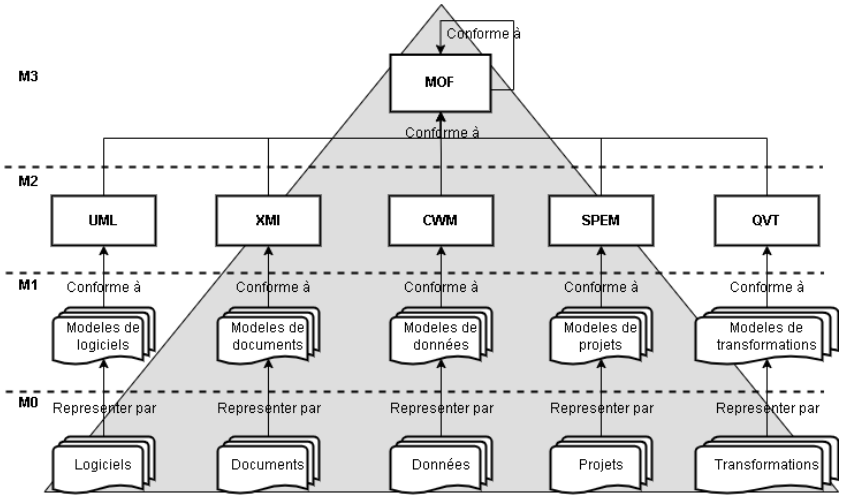


Figure 3.11- Effort de Méta-modélisation OMG.

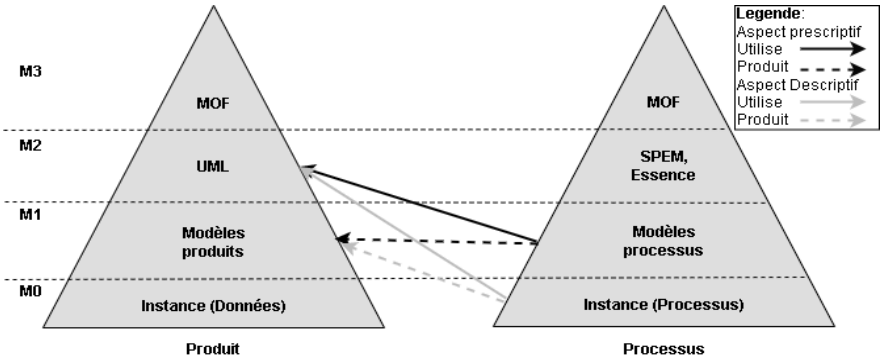


Figure 3.12- Méta-modélisation des procédés logiciels.

Comme le montre la Figure 3.13 (inspirée de (Hug, 2009)) qui donne un exemple pratique des différentes vues et concepts à travers les quatre niveaux MDA. Les processus étant régis par les modèles de processus, par conséquent ils respectent eux aussi cette architecture de méta modélisation. Ainsi, la méta modélisation des processus logiciels est organisée comme suit (Aoussat, 2012):

- Le niveau M0 décrit les processus logiciels.
- Le niveau M1 décrit les modèles de processus logiciels.
- Le niveau M2 décrit les méta-modèles de processus logiciels.
- Le niveau M3 décrit les méta-méta-modèles de processus logiciels.

3.2.6.1 Les procédés logiciels (M0)

Un procédé est défini comme une suite d'opérations ou d'événements faisant intervenir des équipes de personnes, des outils et des techniques pour assurer le développement et la maintenance de produits ou de services. Un processus logiciel est, par conséquent, un procédé qui permet d'assurer le développement et la maintenance de produits logiciels.

Autrement dit, un processus logiciel est un enchaînement d'unités de travail, de produits logiciels requis et fournis par ces unités de travail, des équipes, des outils, des techniques et des stratégies utilisés dont le but d'assurer le développement et la maintenance de produits logiciels. Concrètement, le processus logiciel représente le monde réel du développement (Figure 3.13 M0).

3.2.6.2 Les modèles de procédés logiciels (M1)

Un modèle est une abstraction, une simplification d'un système qui est suffisante pour comprendre le système modélisé. Un modèle de processus logiciel est la représentation formalisée du processus, il explicite les propriétés et les variables qui régissent le monde réel du développement.

« Un modèle de processus est une description abstraite d'un processus réel qui représente des éléments de processus sélectionnés considérés comme importants pour l'objectif du modèle et peuvent être mis en œuvre par un humain ou une machine » (Münch et al., 2012).

L'objectif principal du modèle de processus est de fournir une représentation plus ou moins formelle pour le développement logiciel. Il doit prendre en charge, entre autres, l'évolution de la réalité du développement. Le modèle de processus représente le processus à utiliser, il peut s'agir d'un paradigme avec une représentation et une méthodologie précise comme le RUP ou le SCRUM pour le développement agile (Figure 3.13 M1).

3.2.6.3 Les méta-modèles de procédés logiciels (M2)

Le méta-modèle de processus logiciel est un modèle regroupant les concepts de base du langage d'expression d'un modèle de processus logiciel. Le méta-modèle de processus logiciel doit prendre en considération les points de vue et les orientations des processus logiciels. Le noyau de toute description de processus logiciel est la suivante : Le processus est un enchaînement d'« activité », chaque activité a besoin de produits « produit de travail » en entrée pour fournir des produits en sortie. Le processus logiciel étant centré humain (et même machine), une activité est sous la responsabilité d'un rôle « Rôle ». Les exemples types sont les standards OMG comme SPEM et Essence (Figure 3.13 M2).

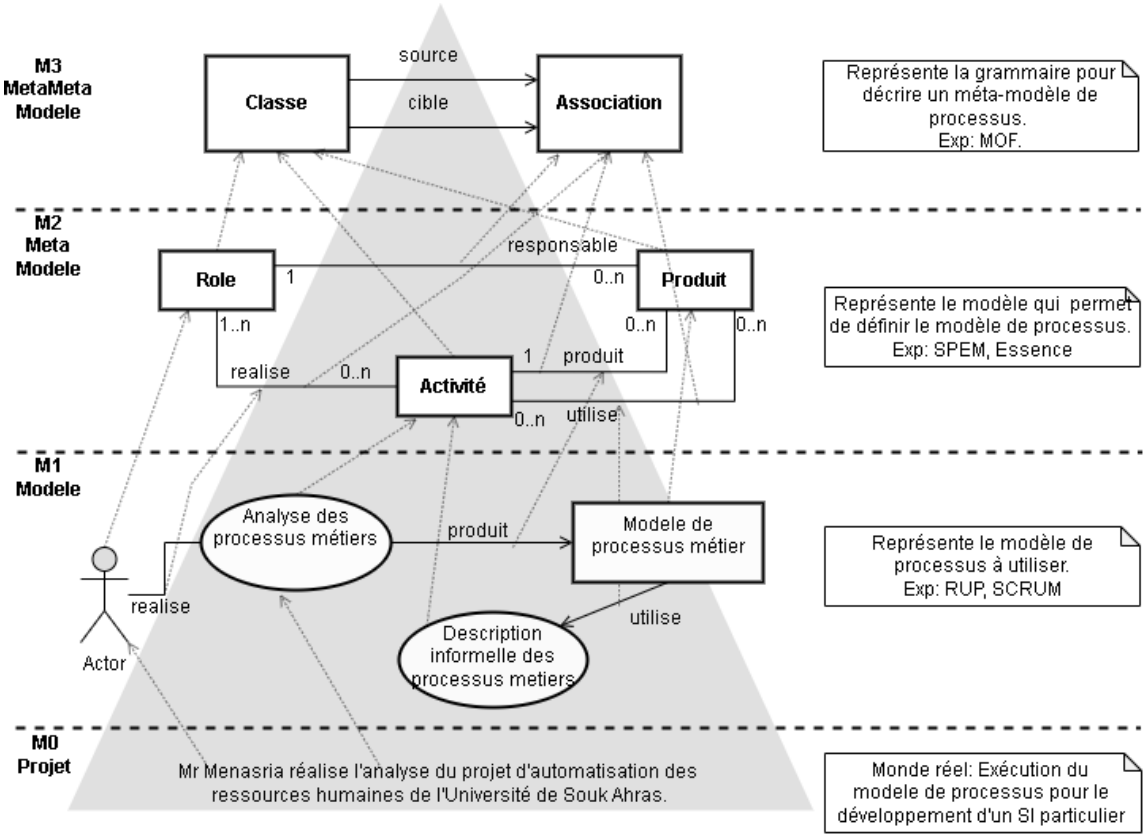


Figure 3.13- Les 4 niveaux de modélisation des processus.

3.2.6.4 Les méta-méta-modèles de procédés logiciels (M3)

Le méta-méta-modèle est une représentation qui permet de décrire les concepts et les structures qui permettent de décrire les méta-modèles. L'OMG spécifie un méta-méta-modèle unique et auto-descriptif qui est le MOF. Il décrit les concepts de base de tous les méta-modèles de tous les domaines y compris celui des processus logiciels (Figure 3.11 et Figure 3.13 M3). Dans ces figures, il faut noter également l'interprétation de la pyramide grisée qui indique la réduction des concepts en parcourant les niveaux de méta-modélisation de bas en haut.

3.3 Ingénierie des modèles

L'ingénierie dirigée par les modèles est une discipline émergente en génie logiciel. Elle implique plusieurs concepts et connaissances en termes de types de transformation, d'axes, de taxonomie, de propriétés, de règles avec leur organisation et ordonnancement, de relation entre les modèles source/cible et enfin en termes d'approches de transformation. Vu le nombre important de concepts dans cette discipline, ils ne seront évoqués dans cette section que les concepts utilisés dans l'approche proposée (Chapitre 0 notamment).

3.3.1 Notions de base

La définition la plus générale et qui fait l'unanimité au sein de la communauté IDM (Bézivin & Briot, 2004) consiste à dire qu' « une transformation de modèles est la génération automatique d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources ». Dans l'approche par modélisation, cette transformation se fait par l'intermédiaire de règles de transformations qui décrivent la correspondance entre les entités du modèle source et celles du modèle cible. En réalité, la transformation se situe entre les méta-modèles source et cible qui décrivent la structure des modèles cible et source. Le moteur de

transformation de modèles prend en entrée un ou plusieurs modèles sources et crée en sortie un ou plusieurs modèles cibles.

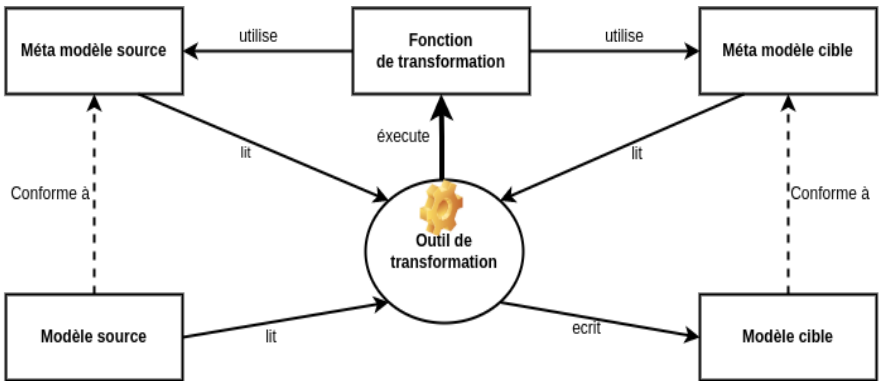


Figure 3.14- Schémas de base des transformations de modèles.

Dans le cas le plus simple, la Figure 3.14 donne un aperçu sur les concepts de base invoqués dans la transformation de modèles. Elle montre un simple scénario de translation avec un seul modèle en entrée (*source*) et un seul modèle en sortie (*cible*). Les deux modèles sont conformes à leurs méta-modèles respectifs. Un méta-modèle définit typiquement la syntaxe abstraite de la notion de modélisation. La transformation est définie en respectant les méta-modèles. La fonction de transformation est exécutée par le moteur de transformation sur des modèles concrets. En général, une transformation peut avoir multiples modèles sources et destinations, en outre, les méta-modèles sources et cibles peuvent être les mêmes dans certaines situations (Menasria, 2011). Evidemment, les deux méta-modèles et la fonction de transformation doivent être conformes au même méta-méta-modèle (MOF).

Une transformation des entités du modèle source met en jeu deux étapes. La première étape permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs méta-modèles, c'est le rôle accordé à la fonction de transformation qui est applicable à toutes les instances du méta-modèle source. La seconde étape consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme ou le moteur de transformation ou d'exécution. C'est au moteur de transformation de faire respecter ce scénario. Il lit le méta-modèle source, le modèle source et le méta-modèle cible. Il exécute les règles de transformation pour générer le modèle cible.

Ces transformations de modèle –dites de 3ème génération- visent à considérer l'« opération » de transformation comme un autre modèle conforme à son propre méta-modèle (lui-même défini à l'aide d'un langage de méta-modélisation, par exemple le MOF). La transformation d'un modèle Ma^* (conforme à son méta-modèle MMa^*) en un modèle Mb^* (conforme à son méta-modèle MMb^*) par le modèle *Mt.* Cette transformation peut donc être formalisée par le patron suivant:

$$Mb^* = f(MMa^*, MMb^*, Mt, Ma^*)$$

3.3.2 Règles de Transformation

La fonction de transformation (Figure 3.14) représente le support ou le moyen pour définir modéliser la transformation. Elle est essentiellement constituée d'une règle ou d'un ensemble de règles qui permettent le *mapping*, le passage ou la réécriture de tout ou partie d'un modèle source vers son correspondant dans un modèle cible. On peut définir une fonction de transformation comme un ensemble de règles de transformation.

Une règle de transformation est une description de la correspondance entre une (ou

plusieurs) construction(s) du modèle source et une (ou plusieurs) construction(s) du modèle cible. Elle est considérée, en large termes, comme la plus petite unité de transformation. Une fonction ou procédure qui implémente une suite de transformation, peut être considérée comme une règle (Czarnecki & Helsen, 2006).

3.3.2.1 Structure générale d'une règle

Dans le cadre de cette thèse, on s'intéresse aux règles de réécriture qui sont composées de deux parties ou deux membres: Une partie gauche (LHS : *Left Hand Side*) qui accède au modèle source et une partie droite (RHS : *Right Hand Side*) qui accède au modèle cible. Une règle peut comprendre aussi une logique ou un style d'écriture qui exprime des contraintes ou calculs sur les éléments du modèle source ou cible. Une logique peut avoir la forme déclarative ou impérative. Une logique déclarative consiste à spécifier des relations entre les éléments du modèle source et de ceux du modèle cible.

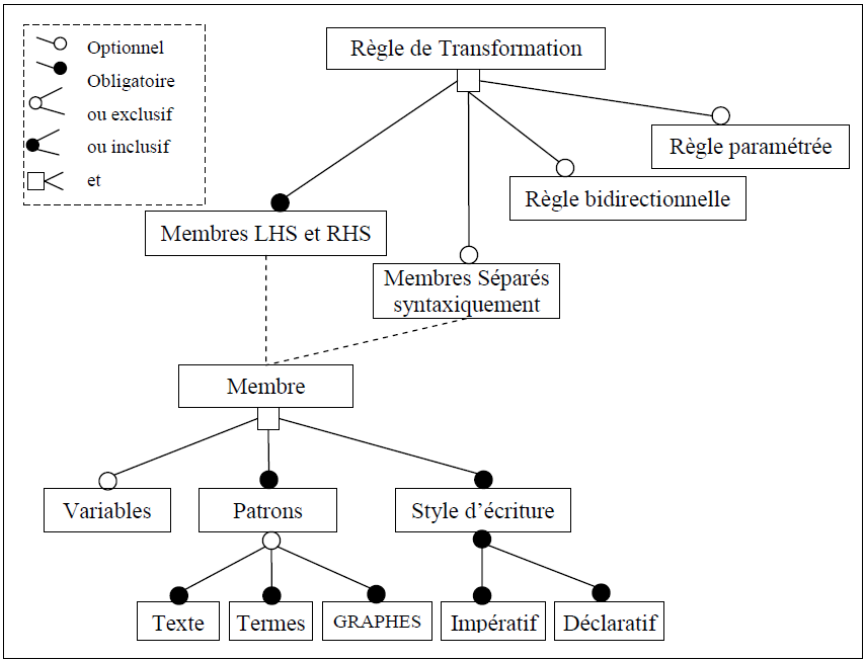


Figure 3.15- Structure générale d'une règle de transformation.

Une logique impérative correspond le plus souvent, mais pas nécessairement, à l'utilisation de langages de programmation pour manipuler directement les éléments des modèles par le biais d'interfaces dédiées.

Optionnellement, une règle (Figure 3.15) peut être bidirectionnelle, comporter des paramètres, ou encore nécessiter la construction de structures intermédiaires. Les membres ou parties LHS et RHS peuvent être syntaxiquement équivalentes (appartenant au même domaine sémantique ; méta-modèle) ou non. Un membre doit obligatoirement obéir à un patron bien défini qu'il soit textuel ou visuel.

3.3.2.2 Organisation et ordonnancement des règles

L'organisation concerne la composition et la structuration de plusieurs règles. Les règles peuvent être organisées de façon modulaire, avec la notion d'importation. Les règles peuvent également utiliser la réutilisation qui permet de définir une règle basée sur une ou plusieurs autres règles, par le biais de mécanismes d'héritage entre règles, dérivation, extension et spécialisation ainsi que la composition, par le biais d'un ordonnancement explicite. Enfin, les règles peuvent être organisées selon une structure dépendante du modèle source ou du modèle cible.

Les mécanismes d'ordonnancement déterminent l'ordre dans lequel les règles sont appliquées. Dans le cas d'un ordonnancement implicite, l'algorithme d'ordonnancement est défini par l'outil de transformation. Dans le cas d'un ordonnancement explicite, des mécanismes permettent de spécifier l'ordre d'exécution des règles. Cet ordre d'exécution peut être défini de manière externe ou interne : tandis qu'un mécanisme externe établit une séparation claire entre les règles et la logique d'ordonnancement, un mécanisme interne permet aux règles d'invoquer d'autres règles.

Enfin, l'ordonnancement des règles peut se baser également sur des conditions d'application positives ou négatives (NAC), des itérations ou sur une séparation en plusieurs phases où certaines règles ne pouvent être appliquées que dans certaines d'entre elles (pré-condition). La sélection d'une règle peut se faire d'une manière déterministe ou non déterministe. On peut associer une action liée à l'exécution de la règle ou (post-condition). Ce genre de mécanisme est étroitement lié à l'outil de modélisation ou de transformation utilisé.

3.3.3 Relations entre modèles source et cible

Pour certains types de transformations, la création d'un nouveau modèle cible est nécessaire, pour d'autres, la source et la cible sont le même modèle, ce qui revient en fait à une modification de modèle ou transformation sur place (Figure 3.16).

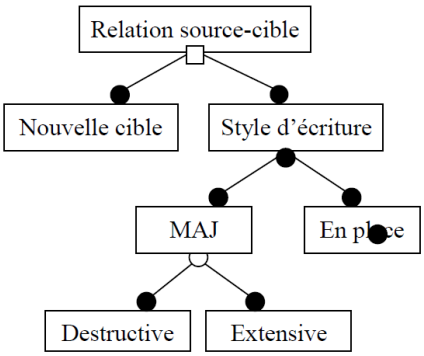


Figure 3.16- Relation modèle source-cible

La transformation peut créer un modèle cible tout en préservant le modèle source. Elle peut également remplacer le modèle source par le modèle cible, on appelle alors transformation sur place. Elle peut aussi être intacte, c'est-à-dire qu'elle n'a aucun effet visible, c'est lé cas généralement des processus de génération de codé comme c'est dans le cadre de la présente thèse.

À partir de là, une transformation de modèle peut être unidirectionnelle ou bidirectionnelle auquel cas les relations entre modèle source et modèle cible sont préservées pour permettre un passage ou retour éventuel du modèle cible vers le modèle source (Hidaka et al., 2016).

3.3.4 Caractéristiques des transformations de modèle

Les propriétés et caractéristiques sont vues de façons différentes en fonction des visions de différentes études. Toutefois, les différentes recherches dans le domaine se réfèrent toujours à deux classifications de base : (Czarnecki & Helsen, 2006) et (Mens & Van Gorp, 2006). Nous utilisons leurs taxonomies d'une manière très sommaire dans cette section.

Selon la cardinalité de la transformation, on identifie souvent le type des transformations de modèle en fonction du nombre de modèles cibles/sources. La

transformation est dite simple (1x1) si pour chaque modèle source, on a un seul modèle cible. Elle est dite multiple (M, N) si ce nombre est indifférent. Les transformations de décomposition de modèles (1 vers N) et de fusion de modèles (N vers 1) sont des cas particuliers de transformations multiples.

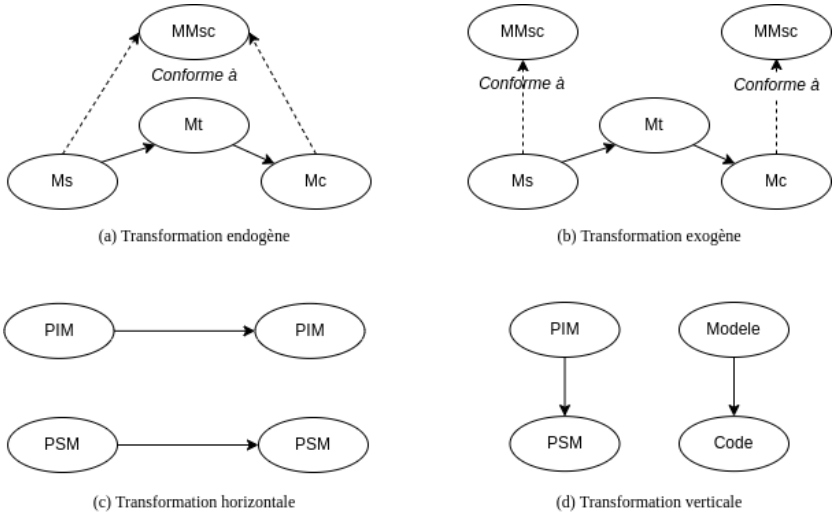


Figure 3.17- Taxonomie des transformations de modèles.

Selon l'axe de transformation on distingue aussi trois catégories de transformation. On identifie l'axe des processus, celui des méta-modèles et celui des paramètres :

- L'axe processus permet de positionner fonctionnellement les transformations par rapport au processus global d'ingénierie. Il est composé de deux sous-axes : le sous-axe vertical qui consiste à faire une transformation de modèles en changeant de niveau d'abstraction (raffinement d'un modèle, génération de code …), et le sous-axe horizontal qui consiste à faire une transformation de modèles en restant au même niveau d'abstraction (restructuration, fusion, …) comme le précise la Figure 3.17 (c-d).
- L'axe méta-modèle permet de caractériser l'importance des méta-modèles mis en jeu dans une transformation. En programmation classique, un algorithme vérifie la conformité des types des variables mis en jeu dans une opération. Dans le cas de la transformation, les méta-classes des méta-modèles correspondent aux types des variables, d'où l'influence des méta-modèles dans le code d'une transformation de modèles.
- L'axe paramétrage permet de caractériser le degré d'automatisation et de réutilisation des transformations avec la présence de données pour paramétrer la transformation.

Dans la même taxonomie, on identifie les transformations dites endogènes ou exogènes en fonction de la nature des méta-modèles correspondants aux modèles sources et cibles. Quand on a le même modèle de part et d'autre, on parle de transformation endogène sinon elle est dite exogène (Figure 3.17 (a) et (b)). Le Tableau 3.3 dresse cette classification en fonction de l'axe des processus et l'axe des méta-modèles. Dans le cadre de cette thèse, nous sommes dans le cas d'une génération de code donc nous changeons de niveau d'abstraction en utilisant des méta-modèles distincts. Notre transformation est donc verticale et exogène.

Tableau 3.3- Classes de transformations typiques

Transformation	Horizontale	Verticale
Endogène	Restructuration Normalisation Intégration de patron	Raffinement
Exogène	Migration de logiciels	PIM vers PSM Rétro-conception Génération de code

3.3.5 Approche des transformations de modèles

Selon (Czarnecki & Helsen, 2006) et (Erata et al., 2015), il existe deux catégories d'approches de transformation, les transformations du type modèle vers code[1] et celles du type modèle vers modèle. Ces deux catégories se distinguent de la nature du modèle cible.

3.3.5.1 Approches de type modèle vers code

Les approches modèle-modèle créent leurs cibles comme une instance du méta-modèle cible, la cible des approches modèle-code est juste une chaîne de caractères ou un texte. Les transformations du type modèle-code sont utiles pour la génération du code et des artefacts non-code tels que les documents. Dans cette catégorie, on trouve également de nos jours les catégories de texte-modèle généralement utiliser au reverse engineering ou à l'extraction des modèles à partir du code.

Dans les approches modèle-texte, on trouve deux tendances :

- Les approches basées sur le principe du « *visiteur* » qui reposent sur le principe d'injecter dans les modèles des éléments (mécanismes visiteurs) pour réduire la différence de sémantique entre le modèle et le langage de programmation cible. Le code est obtenu en parcourant le modèle enrichi pour créer un flux de texte.
- *Les* approches basées sur le principe des « *patrons* » -les plus utilisées-. Le code cible contient des morceaux de méta-code utilisés pour accéder aux informations du modèle source. La majorité des outils MDA couramment disponibles supporte ce principe de génération de code à partir de modèle.

3.3.5.2 Approches de type modèle vers modèle

Les transformations de type *modèle vers modèle* ont toujours été les moins maitrisées, néanmoins, depuis l'apparition de la MDA, ces approches ont suscitées beaucoup d'intérêts et ont subi de grandes évolutions ces dernières années. On distingue généralement cinq approches:

Les approches par manipulation directe, qui sont basées sur une représentation interne des modèles source et cible, et sur un ensemble d'APIs pour les manipuler.

- Les approches basées sur la structure, qui se résument en deux phases. La première consiste à créer la structure hiérarchique du modèle cible. La seconde consiste à ajuster les attributs et références dans le modèle cible. L'idée de base est de copier les éléments du modèle source vers modèle

[1] Le code est aussi considéré comme un modèle d'où la première famille représente un cas particulier de la deuxième, il suffit d'ajouter un méta-modèle pour le langage de programmation cible.

cible qui peuvent alors être adaptées pour achever l'effet de la transformation voulue.

- Les approches opérationnelles, qui sont similaires à celles par manipulations directes mais qui offrent plus de supports dédiés aux transformations de modeles.Une solution typique est d'étendre le formalisme de méta modélisation utilisé avec des moyens d'expression de calcul. Un exemple serait d'étendre un langage de requête comme OCL avec des constructions impératives.
- Approches basées sur les patrons où les modèles de patrons sont des modèles avec un méta-code imbriqué qui traite les parties variables des instances de patrons résultants. Les modèles de patrons sont souvent exprimés en syntaxe concrète du langage cible ce qui aide le développeur à prédire le résultat de l'instanciation de patrons.
- Les Approches relationnelles qui utilisent une logique déclarative reposant sur des relations d'ordre mathématique, elles peuvent être vues comme des résolutions de contraintes. L'idée de base est de spécifier les relations entre les éléments des modèles source et cible par le biais de contraintes. L'utilisation de la programmation logique est particulièrement adaptée à ce type d'approche. Généralement, les transformations produites sont bidirectionnelles.
- Approches basées sur les transformations de graphes, elles exploitent les travaux réalisés sur les transformations de graphes (Andries et al., 1999). Elles sont similaires aux approches relationnelles dans le sens où elles permettent l'expression des transformations sous une forme déclarative. Néanmoins, les règles ne sont plus définies pour des éléments simples mais pour des fragments de modèles: on parle de filtrage de motif '*pattern matching'*. Les motifs dans le modèle source, correspondant à certains critères, sont remplacés par d'autres motifs du modèle cible. Les motifs, ou fragments de modèles, sont exprimés soit dans les syntaxes concrètes respectives des modèles soit dans leur syntaxe abstraite.

3.3.6 Transformation de graphe

Les transformations de graphe (Hartmut Ehrig et al., 2008; König et al., 2018) est l'approche qui émerge d'une manière naturelle et intuitive parmi les approches de transformation de modèles, ceci est dû principalement à la nature même des deux concepts (graphe, modèle).

En effet, les graphes et les diagrammes fournissent une approche simple et puissante pour représenter une grande variété de problèmes liés à l'informatique en général et au génie logiciel en particulier. Pour la majorité des activités liées aux processus de développement de logiciels, plusieurs notations visuelles ont été proposées dont les diagrammes d'états, l'analyse structurée, les graphes de flot de contrôle, les langages de description d'architecture et la famille de langage liée à UML. Ces notations produisent des modèles qui peuvent être vu comme des graphes et, par conséquent, les transformations de graphes sont impliquées, soit explicitement ou implicitement, lorsqu'on spécifie comment ces modèles doivent être construits, interprétés, évolués dans le temps et dérivés en implémentation.

En même temps, les graphes fournissent une structure de données universellement adoptée, comme un modèle pour la topologie des systèmes orientés objets, orientés composants et répartis. Les traitements dans de tels systèmes sont donc naturellement

modélisés comme des transformations de graphes.

Les transformations de graphes ont, à l'origine, évolué en réaction aux imperfections dans l'expressivité (Heckel, 2006)des approches classiques de réécriture, comme des grammaires de Chomsky avec la réécriture de termes textuels, et leurs limites pour traiter les structures non linéaires. Cependant, elles peuvent être considérées comme une généralisation des grammaires de Chomsky pour les graphes (Kuske, 2001). Les deux concepts se rejoignent et se substitues en remplaçants les « termes » par les « graphes », la « réécriture des termes » par le « collage des graphes » et enfin la « description textuelle » par la « modélisation visuelle ».

Les graphes (nœuds/arcs) ont beaucoup évolué en mathématique avec un fondement théorique solide (théorie des ensembles et des catégories) (Awodey, 2010) d'une part, les transformations de graphe ont hérité de ces assises formelles pour construire une variété d'approches supportées par les outils.

La transformation de graphe, en un mot, est une forme de programmation par l'exemple où l'on essaye de généraliser par l'extraction de règles d'évolution à partir d'un comportement observable d'un phénomène donné (Heckel, 2006) (Figure 3.18 et Figure 3.19).

La transformation de graphe est un processus de réécriture de graphes basé sur les grammaires de graphes. Une grammaire de graphe n'est autre qu'une suite de règles bien formées, par analogie aux grammaires de Chomsky où les termes sont remplacés par des graphes et la réécriture de terme est remplacée par le collage de graphes. La partie gauche (*précondition* ou *prémisse*) RHS est un morceau de graphe et la partie droite (*post-condition*) est aussi un morceau de graphe. Appliquer une pareil règle à un graphe d'accueil remplacerit l'occurrence de la partie gauche dans ce même graphe par la partie droite de la règle pour obtenir un graphe modifié, les graphes intermédiaires sont appelés graphes de contexte.

Les images suivantes montrent une représentation significative de graphes dans un domaine de modélisation. Elles indiquent lesquels des éléments sont à supprimer et ceux qui sont à ajouter. La règle dans cet exemple lance le corps d'une barque de son support dans l'eau (Figure 3.18).

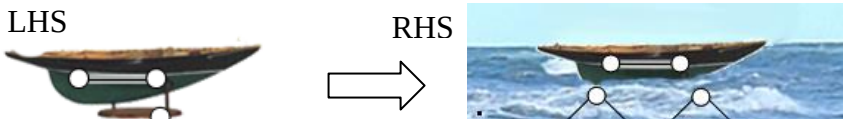


Figure 3.18- Règles de graphes

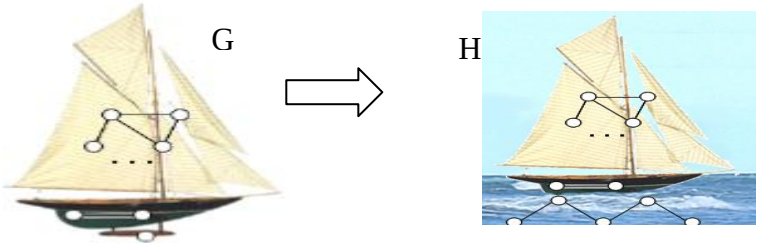


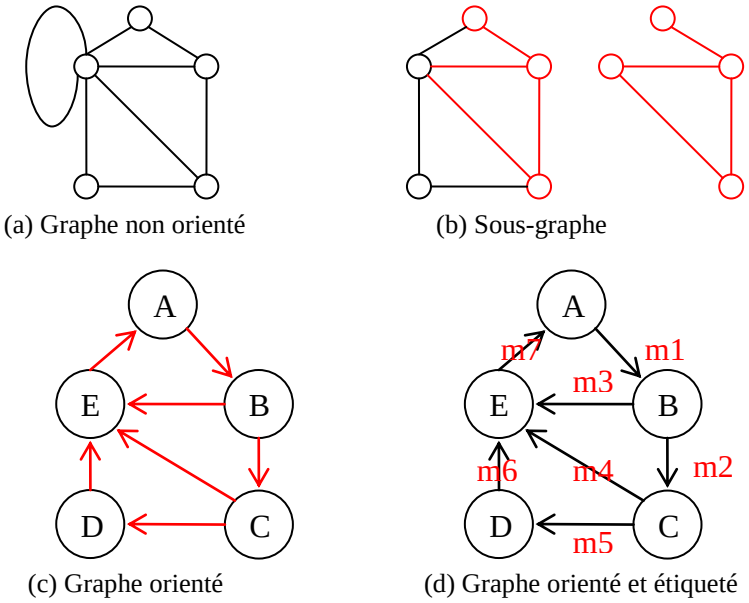
Figure 3.19- Application de la règle

Une transformation directe, utilisant cette règle, fait plonger la barque entière y compris la voile (G) dans l'eau (Figure 3.19). La connexion du graphe de la partie gauche de la règle (LHS) à la partie du graphe de contexte dans le graphe d'application est préservée. Par conséquent, le corps de la barque dans l'exemple demeure connecté à la voile (H).

Comme l'ingénierie dirigée par les modèles s'appuie sur les modèles et les transformations de modèles, la transformation de graphe s'appuie obligatoirement sur deux concepts de base: les graphes et les réécritures de graphe.

3.3.6.1 Notion de graphe

Un graphe n'est autre qu'un ensemble de Sommets (nœuds) reliés par des arcs *G*(*V*,*E*). Formellement et plus généralement, soit $\sum$ un ensemble de symbole. Un graphe à travers $\sum$ est un système $G = (V, E, s, t, l)$ où V est un ensemble fini de nœuds, E est un ensemble fini d'arcs avec $V_G \cap E_G = \phi$, $s,t : E \to V$ sont des fonctions de correspondances assignant une *source* *s*(e) et une *cible* *t*(e) à chaque arc dans E et $l : E \to \sum$ associe a chaque arc un symbole. Quand $s(e) = t(e)$, l'arc est appelé une boucle. Les composants *V*, *E*, *s*, *t* et *l* sont aussi notés V_G, E_G, s_G, t_G et l_G. respectivement. L'ensemble de tous les graphes à travers $\sum$ est noté $\mathcal{G}_{\sum}$[1].



(a) Graphe non orienté (b) Sous-graphe

(c) Graphe orienté (d) Graphe orienté et étiqueté

Figure 3.20- Notion de graphes

Il existe une variété de graphes applicables dans différentes situations en fonction de leurs sémantiques. On peut trouver entre autres :

Un *graphe non orienté* est constitué de sommets (nœuds) qui sont reliés par des arêtes. Deux sommets reliés par une arête sont adjacents. Le nombre de sommets présents dans un graphe est appelé ordre du graphe. Le graphe de la Figure 3.20 (a) est d'ordre 5. Le degré d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité. Le degré du sommet E de la Figure 3.20 (c) est 4.

Un *graphe orienté* est un graphe dont les arêtes ont une orientation ou direction: on parle alors de l'origine et de l'extrémité d'une arête. Dans un graphe orienté une arête est dénommée arc (Figure 3.20 (c)).

Un *graphe étiqueté* est un graphe orienté, dont les arcs possèdent des étiquettes. Si toutes les étiquettes sont des nombres positifs, on parle de graphe pondéré (Figure 3.20 (d)).

Un *graphe attribué* est un graphe qui peut contenir un ensemble prédéfini d'attributs.

[1] Définition pour un graphe orienté étiqueté

Un attribut peut être un nombre, un texte, une expression, une liste ou des événements. Les attributs peuvent être de différents types et les opérations compatibles avec ces types doivent être disponibles pour les manipuler. Un diagramme de classe UML avec les relations de dépendance peut être considéré comme un graphe attribué (Andries et al., 1999).

a) Graphes typés

Pour les transformations de modèles par les transformations de graphes, on a besoin de convenir d'une manière de consensus pour spécifier un modèle. Les modèles sont des graphes. Ceci exige la définition de méta-modèle pour décrire ce que doit être valide ou un modèle bien formé. On peut alors utiliser aussi un graphe type (Mens, 2006) pour représenter un méta-modèle et les graphes pour représenter les modèles (Figure 3.21).

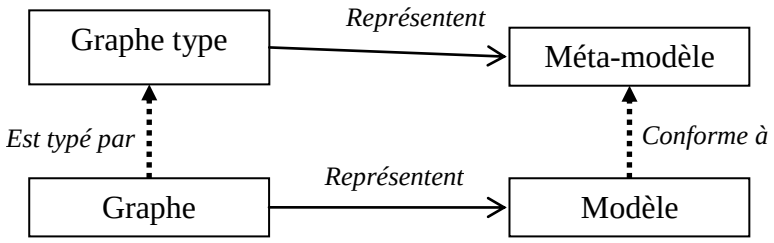


Figure 3.21-Relation entre les modèles et les graphes.

Formellement, soit *TG* un graphe type. Un graphe typé à travers *TG* est une paire (G,t) tel que *G* est un graphe et $t:G\to TG$ est morphisme de graphe total. Un morphisme de graphe typé $(G,t_G)\to(H,t_H)$ est un morphisme de graphe partiel (Figure 3.22) $m:G\to H$ qui préserve le typage c.-à-d. $t_H \circ m = t_G$.

Cette définition requiert un morphisme de graphe total dans le but d'assurer que chaque nœud et arc dans le graphe G ait son type correspondant dans TG.

Elle peut être étendue de plusieurs façons pour ajouter des contraintes additionnelles sur des graphes typés concrets qui sont des instances d'un graphe type:

- le graphe type peut être attribué pour contraindre les noms et les types des nœuds dans les graphes typés concrets.
- le graphe type peut contenir des cardinalités sur les nœuds et les arcs pour borner leurs nombres dans les graphes typés concrets.
- le graphe type peut contenir des relations d'héritages entre les nœuds pour exprimer le fait que les types de tous les attributs, cardinalités et les arcs adjacents d'un super-type sont hérités par un sous-type par analogie à la relation de généralisation dans un méta-modèle UML.
-

b) Sous-graphe et Morphisme de graphe

Un sous-graphe d'un graphe G est un graphe G' composé de certains sommets de G, ainsi que toutes les arêtes qui relient ces sommets. Un sous-graphe est noté par $G\subseteq H$, si $V_G\subseteq V_H$ et $E_G\subseteq E_H$ (Figure 3.20 (b)).

Un morphisme de graphe (Figure 3.22) du graphe *G* vers un graphe *H* consiste en deux fonctions de correspondance : $g_V = V_G \to V_H$, $g_E = E_G \to E_H$ tels que les étiquettes, les sources et les cibles sont préservées (préservation de structure), c.-à-d.

$g_v(s_G(e)) = s_H(g_E(e))$, $g_v(t_G(e)) = t_H(g_E(e))$ et $l_H(g_E(e)) = l_G(e)$ et ce pour tout $e \in E_G$. Sachant que $f \circ g(x) = f(g(x))$, on peut réécrire les formules précédentes comme suit: $l_G = l_H \circ g_V$, $m_G = m_H \circ g_E$, $g_v \circ s_G = s_H \circ g_E$ et $g_v \circ t_G = t_H \circ g_E$. L'image de *G* dans *H* est aussi appelé une occurrence de G dans H notée par $g(G)$. Les graphes *G* et *H* sont isomorphiques si g_v et g_E sont bijectives. Si g_v et g_E sont des inclusions, alors *G* est un sous-graphe de *H*. Pour un morphisme de graphe $g_V = V_G \to V_H$ l'image de *G* dans *H* est appelé aussi un *matching* de *G* dans *H*, le match de *G* avec le respect de g est un sous-graphe $g(G) \subseteq H$ (König et al., 2018; Kuske, 2001).

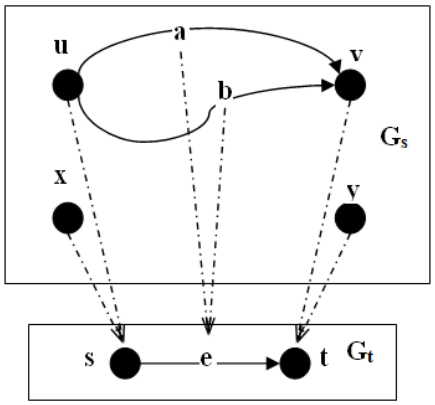


Figure 3.22- Représentation d'un morphisme de graphes.

3.3.6.2 Grammaires de graphes

Les graphes et les diagrammes sont un moyen très pratique pour la description des structures complexes et des systèmes ainsi que pour la modélisation des idées de manière directe et intuitive. Si les graphes servent à visualiser les structures complexes des modèles d'une façon simple et intuitive, les transformations de graphes peuvent être exploitées pour spécifier comment ces modèles peuvent évoluer.

Une transformation de graphe (Andries et al., 1999) consiste en l'application d'une règle à un graphe et répéter ce processus. Chaque application de règle transforme un graphe par le remplacement d'une de ses parties par un autre graphe. Autrement dit, la transformation de graphe est le processus de choisir une règle d'un ensemble indiqué, appliquer cette règle à un graphe et réitérer le processus jusqu'à ce qu'aucune règle ne puisse être appliquée.

La transformation de graphe est spécifiée sous forme d'un modèle de grammaires de graphes. Elles sont composées de règles dont chacune est composée d'un graphe de côté gauche (**LHS**) et d'un graphe de côté droit (**RHS**).

Une *grammaire de graphe* (Andries et al., 1999) est un ensemble *P* de règles muni d'un graphe initial *S* et d'un ensemble *T* de symboles terminaux, $P = \{S, T\}$. Les grammaires de graphes sont une généralisation des grammaires textuelles de Chomsky (Mens, 2006).

Une grammaire de graphes distingue les graphes non terminaux, qui sont les résultats intermédiaires sur lesquels les règles sont appliquées, des graphes terminaux dont on ne peut plus appliquer de règles, on dit que ces derniers sont dans le langage engendré par la grammaire. Pour vérifier si un graphe *G* est dans les langages engendrés par une grammaire de graphe, il doit être analysé. Le processus d'analyse va déterminer une séquence de règles dérivant *G*.

Dans le but de définir une règle de transformation de graphe et son application, on a besoin des concepts de sous-graphe et de morphisme de graphe. Intuitivement, un graphe *G*

est un sous-graphe de *H* si G est une partie de *H* tandis qu'un morphisme de graphe du graphe *G* vers un graphe *H* relie *G* à une image de *G* dans *H* qui soit structurellement équivalente à *G*. Deux graphes structurellement équivalents (avec égalité de nœuds et d'arcs : bijection) sont dits isomorphes (Kuske, 2001).

a) Le principe de règle de graphe

L'idée d'une règle de transformation de graphe est d'exprimer quelle partie de graphe est à remplacer par une autre. Différemment des chaînes de caractère, pour remplacer un sous-graphe, il y a plusieurs façons pour le relier au reste du graphe (plusieurs arcs). Par conséquent, une règle doit spécifier aussi les types de liens autorisés. Ceci est fait avec l'aide d'un troisième graphe qui soit commun au graphe remplacé et au graphe de remplacement (Lambers & Weber, 2018) . C'est le graphe qui joue le rôle d'interface ou de parties partagées entre les graphes de gauche et de droite. Sans ce graphe, communément dénoté *K*, on perd toutes les liaisons et par conséquent on aura aucune façon pour recoller la partie droite de la règle.

Dans sa forme la plus générale, une règle de transformation de graphe est définie par : p = (*L*, *R*, *K*, *glue*, *emb*, *cond*) qui consiste en:

- deux graphes : graphe *L* de côté gauche et graphe *R* de côté droit.
- un sous-graphe *K* de *L*.
- une occurrence *glue* de *K* dans *R* qui relie le sous-graphe avec le graphe de côté droit.
- une relation d'enfoncement (intégration) *emb* qui relie les sommets du graphe de côté gauche et ceux du graphe du côté droit.
- un ensemble *cond* qui spécifie les conditions d'application de la règle, on distingue les conditions d'application positive (PAC) et négative (NAC).
-

b) Application des règles (Réécriture)

Comme pour la réécriture dans les grammaires textuelles, où il s'agit de produire la partie droite d'une règle de production (tant qu'elle est dérivable) en fonction des prémisses de gauche pour engendrer un langage avec des termes terminaux seulement, les grammaires de graphes utilise le même mécanisme de réécriture.

Les transformations de graphes sont basées essentiellement sur la réécriture de graphes dans les grammaires de graphes. Une grammaire de graphe est de suite de règles. Une règle est essentiellement formée de deux parties, gauche (LHS) et droite (RHS) qui sont tous les deux des graphes. La réécriture consiste à « remplacer ou réécrire» les parties gauches des règles par leurs parties droites dans un graphe source ou initial pour obtenir un graphe cible.

Les grands livres pour les pionniers du domaine (H Ehrig et al., 1999; Rozenberg, 1997) ont pour titre '*Handbook of graph grammar and computing by graph* transformation'. Il est mentionné dans la deuxième partie du titre « traitement par transformation de graphe », c'est-à-dire que nous ne sommes plus dans les solutions algorithmiques et que nous pouvons désormais faire un « traitement » d'une autre manière. C'est toute la magie des transformations de graphes.

L'application d'une règle (Andries et al., 1999) p= (*L*, *R*, *K*, glue, *emb*, *cond*) à un graphe *G* produit un graphe résultant *H*. Le graphe *H* fourni, peut être obtenu depuis le graphe d'origine *G* en passant par les cinq étapes suivantes (Figure 3.23 (Amirat & Menasria, 2016;

Menasria et al., 2012)):

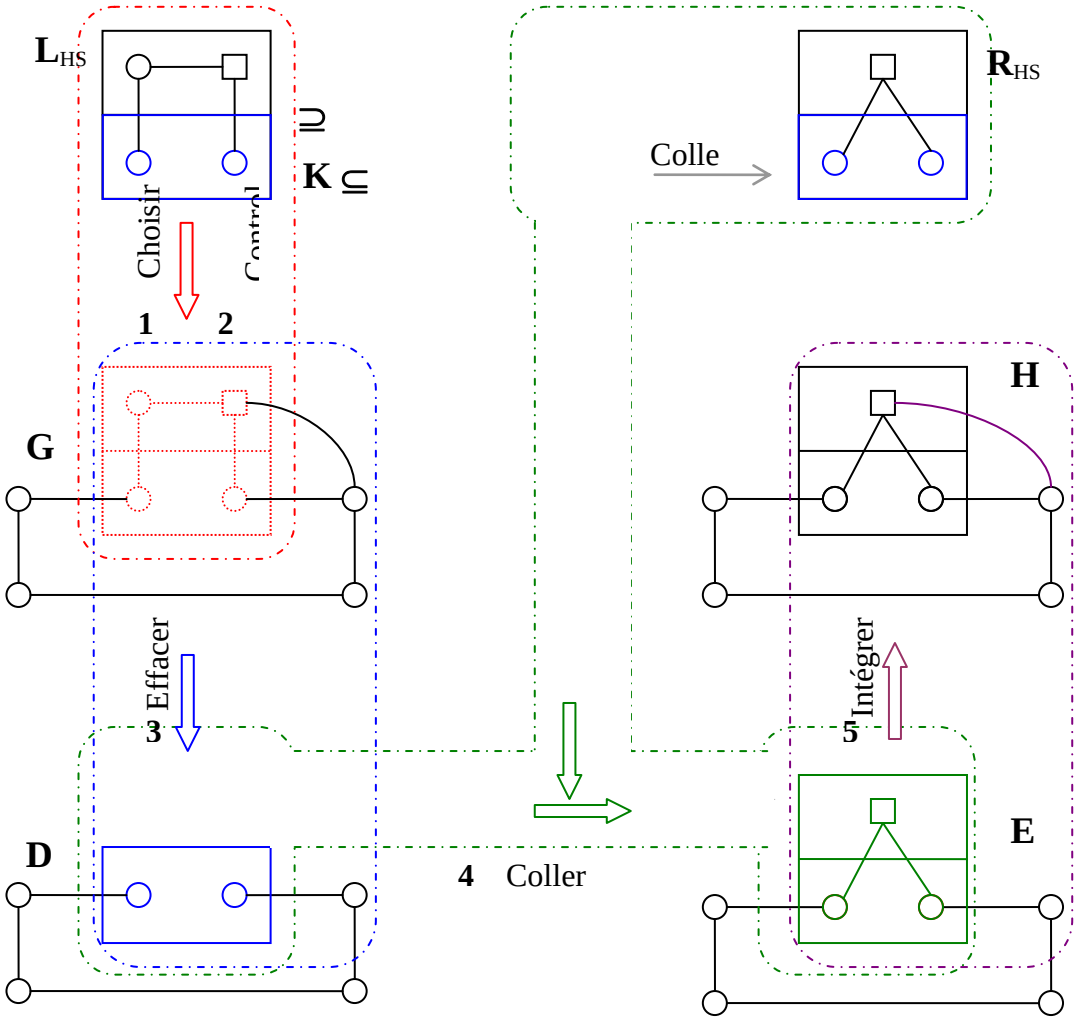


Figure 3.23- Réécriture de graphes

1. Choisir ou trouver une occurrence du graphe de côté gauche L dans G (Figure 3.23, partie rouge).
2. Vérifier les conditions d'application d'après *cond* (Figure 3.23, partie rouge).
3. Retirer l'occurrence de *L* (jusqu'à *K*) de *G* ainsi que les arcs pendants, c-à-d tous les arcs qui ont perdu leurs sources et/ou leurs destinations. Ce qui fournit le graphe de contexte *D* de *L* qui a gardé une occurrence de *K* (Figure 3.23, partie bleue).
4. Coller le graphe de contexte *D* et le graphe de côté droit *R* suivant l'occurrence de *K* dans *D* et dans *R*. c'est la construction de l'union de disjonction de *D* et *R* et, pour chaque point dans *K*, identifier le point correspondant dans *D* avec le point correspondant dans *R (*Figure 3.23, partie verte).
5. Enfoncer (Intégrer) le graphe du côté droit dans le graphe de contexte de L suivant la relation d'enfoncement *emb* pour rétablir les liaisons aux sommets qui demeurent dans R et qui sont supprimés par L (Figure 3.23, partie mauve).

L'application de p sur un graphe G pour fournir un graphe H est appelée une dérivation

directe depuis G vers H à travers p, elle est dénotée par $G \xrightarrow{p} H$ ou simplement par $G \Rightarrow H$

En donnant les notions de règle et de dérivation directe comme étant les concepts élémentaires de la *transformation* de graphe, on peut définir les systèmes de transformation de graphe, et la notion de langages engendrés.

2.5.3 Approche Algébrique de transformation de graphes

Selon (Ehrig et al., 2008) il existe en somme six approches du domaine des transformations de graphes dont les plus répandues sont celles inspirées directement des grammaires textuelles. Nous nous intéressons sommairement dans notre contexte aux approches algébriques en double/simple '*pushout'* du fait que c'est l'approche implémentée par notre outil de modélisation.

Les approches algébriques pour les transformations de graphes sont basées sur le concept de collage des graphes correspondants aux *pushout* dans des catégories appropriés de graphes et de graphe morphismes. Cela permet de donner, non seulement, une description algébrique ou avec la théorie des ensemble pour la construction mais également d'utiliser les concepts et les résultats de la théorie des catégories (Awodey, 2010) afin de construire une théorie riche et de donner des preuves élégantes dans la situation complexe (Hartmut Ehrig et al., 2008).

On dénombre dans ces approches les catégories suivantes : Double *Pushout*, Simple *pushout*, Double *Pullback* et Simple *Pullback*.

Les constructions *pushout* sont utilisées pour modéliser le collage (*gluing*) de graphes où deux construction de collage sont utilisées dans les étapes de transformation de graphes (Figure 3.24) c'est pour cette raison qu'on les dénomme approches *double-pushout* DPO (Voire aussi *Simple-pushout*) SPO.

Dans l'approche DPO, une règle de production est donnée par $p = (L \leftarrow K \rightarrow R)$ où L et R sont les graphes des parties gauche et droite, K est l'interface commune entre L et R. Soit une production p et un graphe de contexte D, qui inclus aussi K, le graphe d'accueil G d'une transformation de graphe $G \Rightarrow H$ via p est donné par le collage de L et D via K écrit $G = L +_K D$ et le graphe modifié H par le collage de R et D via K, écrit $H = R +_K D$. Plus précisément, on utilise les morphismes de graphe $K \rightarrow L$, $K \rightarrow R$ et $K \rightarrow D$ pour exprimer comment K est inclus dans L, R et D respectivement. Ce qui permet de définir les constructions de collage $G = L +_K D$ et $H = R +_K D$ comme des constructions *pushout* (1) et (2) donnant la double *pushout* de la Figure 3.24[1]

[1] Comme exemple des graphes L, K, R, G, D et H : (voire Figure 3.23).

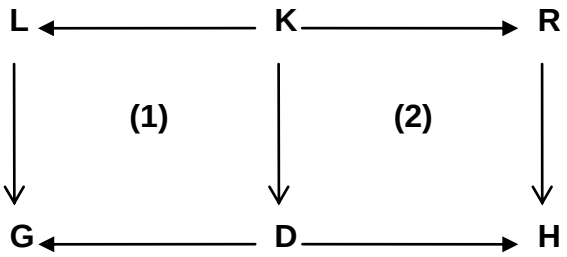


Figure 3.24- Transformation de graphe en DPO.

3.3.7 Outils de transformation de modèle

Du au regain d'intérêt des transformations de modèle et notamment à partir de sa prise en charge par les consortiums de normalisation, les outils de transformation de modèle ont vu une prolifération accrue tant dans le domaine académique qu'industriel. Dans (Kahani et al., 2019), les auteurs dressent une étude détaillée et une classification d'une soixantaine d'outils de transformation de modèles qui couvrent toute la taxonomie suscitée. Comme les modèles sont généralement des graphes, l'usage des transformations de graphes et ses outils deviennent un choix naturel et intuitif (H Ehrig et al., 2015; Guerra & Lara, 2013) dans les techniques et approches de transformation de modèles.. L'étude (Rose et al., 2014) essaye de mesurer l'équivalence entre les outils de transformation de graphes et les outils de transformation de modèles pour cette fin. Dans (Erata et al., 2015), les auteurs font état de la nécessité future de réduire le gap sémantique entre ces disciplines pour faire évoluer QVT (OMG-QVT, 2016) pour prendre en charge les grammaire de graphes triples (TGG) (Hildebrandt et al., 2013; Schürr, 1994) comme pattern de transformation. Ces grammaires ont l'avantage d'être formelles et bidirectionnelles.

Dans cette section, nous allons essayer de comparer trois outils éligibles pour servir et faire valoir notre approche afin de choisir le plus indiqué. Les outils que nous allons considérer assurent, dans leur ensemble, les deux concepts manipulés (séparément ou conjointement) -comme dans le cadre de cette thèse- à savoir la méta-modélisation et la transformation de modèle. ATL est un outil de transformation de modèle pure, alors qu'AGG et AToM3 sont des outils de transformation de graphes pure. La synthèse est dressée dans un tableau comparatif.

3.3.7.1 ATL

ATL Transformation Language (Jouault et al., 2006; Jouault & Kurtev, 2006) est un langage et une boîte à outils de transformation de modèles. Il a été initié par l'équipe AtlanMod (anciennement ATLAS Group ATLAS (INRIA et LINA- actuel LS2N), Université de Nantes-France). Dans le domaine de l'ingénierie dirigée par les modèles (MDE), ATL fournit des moyens de produire un ensemble de modèles cibles à partir d'un ensemble de modèles sources. Publié sous les termes de la licence publique Eclipse, ATL est un composant M2M (Eclipse), à l'intérieur du projet de modélisation Eclipse (EMP). ATL est basé sur le QVT (OMG-QVT, 2016) qui est une norme (OMG) pour effectuer des transformations de modèle. Il peut être utilisé pour faire de la traduction syntaxique ou sémantique. ATL est construit comme une machine virtuelle de transformation de modèle.

3.3.7.2 AGG

Le système AGG (*Attributed Graph Grammars*) (Taentzer, 2004) (Université de Berlin-Allemagne) est un outil d'usage universel de transformation de graphe utilisant les approches

algébriques. AGG permet de spécifier et de fournir rapidement un prototype d'application en utilisant une structure de données complexe basée sur les graphes. L'environnement d'AGG est essentiellement basé sur une interface utilisateur graphique comportant plusieurs éditeurs visuels et un interpréteur. La transformation avec AGG est endogène car il utilise les graphes types pour définir les méta-modèles incluant l'héritage et la multiplicité. L'application d'une règle peut être non déterministe et par conséquent contrôlée par des niveaux de règles. Due à ses fondement théorique, AGG offre des support de validation pour le contrôle de cohérence des graphes et les systèmes de transformation de graphes selon les contraintes de graphes, l'analyse des paires critiques[1] pour déceler les conflits entre les règles et le contrôle des critères de terminaisons[2] pour les systèmes de transformation de graphes.

3.3.7.3 AToM3

Le système AToM^3 (*A Tool for Multi-formalisme and Méta-Modeling*) (de Lara & Vangheluwe, 2002) (Université Autonoma-Espagne et McGill-Canada) est un outil de conception pour les langages visuels de domaines spécifiques. Il permet de définir les syntaxes abstraites et concrètes du langage visuel par la méta-modélisation et d'exprimer la manipulation de modèles par la transformation de graphes en s'appuyant sur les approches algébriques par *PushOut* (König et al., 2018).

Avec les informations du méta-modèle, AToM^3 génère un environnement de modélisation approprié pour les besoins de son utilisateur afin de manipuler le langage souhaité et décrit par le méta-modèle. Ceci étant la notion de multi-formalisme d'AToM^3. Aussi une extension est faite pour lui permettre d'utiliser les grammaires de graphes triples (Schürr, 1994).

De plus amples détails sur AToM^3 seront donnés dans le chapitre 4 avec les motivations du choix de cet outil dans le cadre de cette thèse.

3.3.7.4 Synthèse

Pour choisir un outil de modélisation, il faut avoir un raisonnement et une motivation. Les outils retenus sont tous éligibles dans notre cas d'étude et parviennent à satisfaire tant bien que mal nos attentes. Il est vrai qu'ATL est fort par le fait d'appartenir à la classe des standards dans l'environnement Eclipse. AGG et AToM3 ont cette autre qualité d'appartenir à cette gamme d'outils basés sur des assises formelles. Nous pensons comme (Erata et al., 2015) que les transformations de graphes auront un avenir certain dans les standards de transformation de modèles. Plus est, pour les architectures logicielles qui exigent les phases d'analyse que les outils formels réussissent bien alors qu'ATL n'offre aucun support dans ce sens.

Le Tableau 3.1 dresse un comparatif entre les trois outils sur plusieurs axes comme les objectifs, les environnements, les langages de modélisation, les syntaxes et les mécanismes de transformation. Nous nous appuyons sur ces critères pour choisir l'outil de modélisation. Nous reviendrons à ces motivations dans la section (5.2.2).

[1] Analyse paire critique : recherche des situations conflictuelles pendant la réécriture.
[2] Critère de terminaison : l'application d'une règle doit converger à chaque fois (Taentzer, 2004).

Tableau 3.4- Synthèse sur les outils de transformation de modèles

Propriété / Outil	ATL	AGG	AToM3
Méta-modélisation	Oui	Oui	Oui
Transformation Modèle	Oui	Oui	Oui
Modèle vers Modèle	Oui	Oui	Oui
Modèle vers Texte	Oui	Non	Oui
Transformation graphe	Non	Oui	Oui
Génération environnement	Non	Non	Oui
Langage Méta-modèle	Ecore	Type Graphe	UML, MOF personnalisé
Environnement	Eclipse, Ecore, EMF, GMF	Compacte	Compacte
Mode opératoire	Impératif	Déclaratif	Déclaratif
Modélisation règles de transformation	Helper Règle textuelle	Grammaire de graphe	Grammaire de Graphe
Mode exécution des règles	Parser Dashbord	Réécriture de graphe en couches	Réécriture de Graphe parallèle
Approche de transformation	Interprétation	Algébrique DPO	Algébrique SPO
Langage de manipulation	Java	Java	Python
Représentation interne	XML	Graphe	Graphe
Syntaxe des modèles	Abstraite (UML)	Abstraite (graphe)	Concrète

La Figure 3.25 décrit une autre manière de comparer ces trois outils. Elle montre clairement que sur le plan fonctionnel, l'outil AToM3 possède un atout de taille. En effet, il est le seul à pouvoir opérer sur les quatre niveaux de modélisation en utilisant ses propres ressources contrairement à ATL qui fait recours à EMF, GMF et Ecore alors qu'AGG introduit la notion de graphe typé qui reste loin des standards OMG.

Pour le seul cas de méta-modélisation, AToM3 s'auto suffit pour décrire les quatre niveaux quand il modélise un ou plusieurs domaines spécifiques (DSL): (i) au niveau M0, il peut servir pour la simulation d'exécution (ii) au niveau M1, il peut agir comme un outil de modélisation du modèle est question (iii) au niveau M2, il peut définir le méta-modèle du domaine (iv) au niveau M3, il implémente une version réduite du MOF en utilisant la classe « classe » et la classe « association » pour servir de langage de définition de n'importe quel méta-modèle.

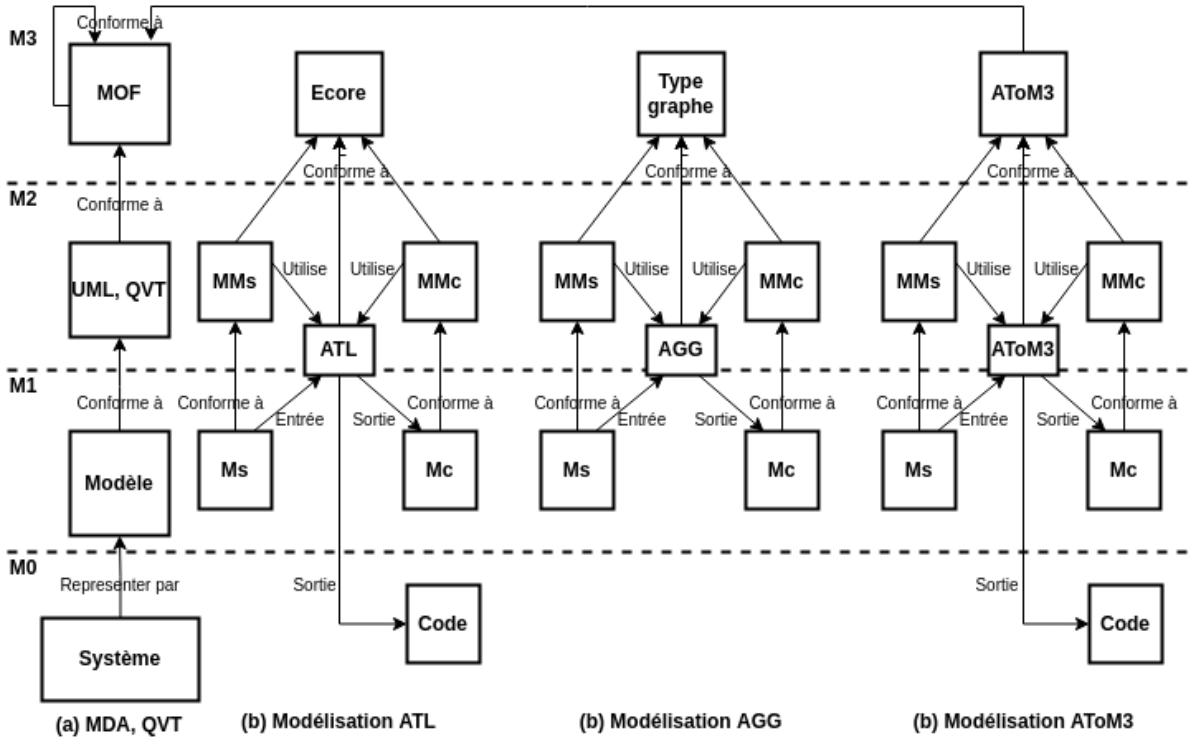


Figure 3.25- Comparatif des outils dans l'environnement MDA.

3.4 Conclusion

Dans ce chapitre nous avons discuté les supports conceptuels et les outils et moyens que nous devrions utiliser pour formaliser nos différentes contributions (Chapitres 40). Ce chapitre discute principalement deux disciplines souvent intimement liées : la méta-modélisation et l'ingénierie des modèles.

Comme le titre de la thèse l'indique, notre problématique première est l'abstraction de la communication dans les architectures logicielles. La réponse intuitive à la notion d'« abstraction » consiste à faire un recours direct à la notion de méta-modélisation pour définir l'ensemble des modèles conceptuels qui forment notre contribution. De la même façon que le connecteur que nous proposons, le langage de description d'architecture qui lui est dédié est également basé sur les techniques de méta-modélisation.

Pour cela, nous avons discuté la littérature de la méta-modélisation en générale et la méta-modélisation des architectures logicielles ainsi que la méta-modélisation dans les processus logiciels. Nous avons discuté également la démarche d'architecture dirigée par les modèles (MDA) qui représente la contribution de l'OMG à l'effort de méta-modélisation et de transformation de modèles.

Le deuxième aspect dont nous avons discuté se réfère à l'ingénierie dirigée par les modèles (IDM). L'usage intensif des modèles favorise naturellement le recours à l'IDM pour leurs manipulations. Pour montrer la faisabilité de notre proposition, nous doter notre ADL de la faculté de générer le pseudo code Java de l'application à partir de sont architecture par des transformations de modèles.

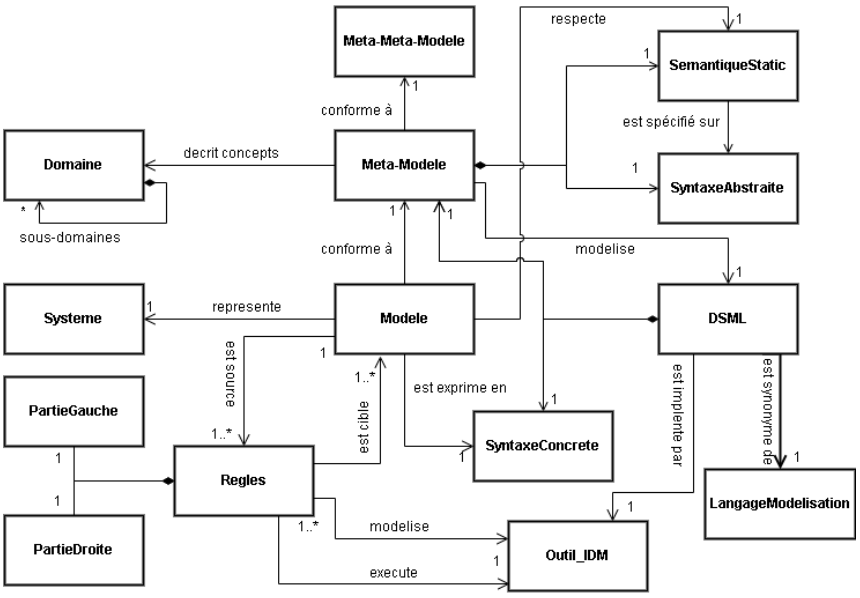


Figure 3.26- Synthèse sur la méta-modélisation et l'ingénierie des modèles.

Pour cela nous avons discuté et synthétisé le domaine de transformation de modèles. Comme ce domaine est assez vaste, nous nous sommes limités à discuter seulement les aspects qui ont un rapport direct avec nos contributions. A titre d'exemple, nous avons discuté les seules transformations de graphes comme approches de transformation de modèle assortie par une modèle comparatifs de quelques éligibles outils.

D'une manière visuelle, la Figure 3.26- Synthèse sur la méta-modélisation et l'ingénierie des modèles. (mise à jour à partir de (Diaw, 2012)) synthétise parfaitement bien ces deux concepts qui sont la méta-modélisation et l'ingénierie des modèles. Elle précise une vraie conclusion entre les différents éléments impliqués et leurs relations en précisant les contours qui délimitent nos deux concepts émergeants par rapport aux autres paradigmes du génie logiciel.

CHAPITRE

4 Approche proposée: CaP, Connecteur comme processus

« Une théorie de la communication doit être développée dans le domaine de l'abstraction. Etant donné que la physique a franchi ce pas dans la théorie de la relativité et de la mécanique quantique, l'abstraction ne devrait pas être en soi un obstacle »
(Luhmann, 2000) dans 'Art as a social systems'.

4.1 Introduction

Dans des environnements distribués et hétérogènes, la communication est une préoccupation fondamentale qui peut conduire à un ensemble de risques importants. Ces risques et préoccupations doivent être pris en compte dès les premières étapes de la conception et du développement de systèmes complexes.

Le développement orienté composants représente un paradigme de maturité avérée en termes de séparation des préoccupations entre la partie fonctionnelle et la partie interactionnelle qui peut apporter des solutions cohérentes aux problèmes mentionnés ci-dessus. Ainsi, le paradigme des composants considère une configuration ou un système architectural comme un ensemble de composants autonomes dont la fonctionnalité globale émerge de leurs interactions.

Pour assurer sa finalité essentielle de réutilisation, le composant en tant que boîte noire ne peut être modifié en fonction de l'environnement particulier ou des fonctionnalités globales des systèmes auxquels il appartient. Il ne peut afficher ses services qu'à travers ses interfaces. Pour construire un système à base de composants, la solution est désormais assurée par une opération de composition utilisant des connexions particulières entre les composants impliqués. Ces connexions sont réalisées par des éléments indépendants de l'application ; communément appelés : connecteurs.

Le rôle déterminant du connecteur dans les architectures à base de composants lui a conféré un impact critique, gagnant ainsi en importance au fil du temps pour devenir une entité de premier ordre dans les différentes phases du cycle de vie du développement à base de composants (Mary Shaw, 1993). A ce titre, le concept de connecteur est intégré comme un élément architectural dans les standards ainsi que dans les langages dédiés principalement les langages de description d'architecture (ADLs) (Medvidovic & Taylor, 2000).

Le connecteur est un lien entre les interfaces des composants en interaction et peut en outre gérer la communication (transfert de données) et la coordination (transfert de contrôle) du composant (Fujita & Herrera-Viedma, 2018). De plus, les connecteurs peuvent représenter des fonctionnalités plus complexes pour agir en tant qu'adaptateurs, distributeurs et parfois arbitres pour garantir la qualité de l'interaction en assurant certaines propriétés telles que la performance et la sécurité (Mehta et al., 2000).

En pratique, l'intérêt dans la recherche comme l'industrie s'est souvent porté sur les composants ou la partie fonctionnelle. Dans des environnements fortement interactifs, les applications récentes sont de plus en plus communicantes pour atteindre des proportions souvent équivalentes entre la partie interaction par rapport à la partie fonctionnelle.

La communauté des architectures logicielles (Fujita & Herrera-Viedma, 2018; Mehta et al., 2000; Mary Shaw & Clements, 2006; Taylor, 2019) considère que tous les connecteurs, quelle que soit leur complexité, ne sont que des mécanismes de transfert de données et de contrôle via une conduite ou un canal. Le même constat est partagé par la communauté des processus logiciels (Aoussat et al., 2014; Dai et al., 2008; Dami et al., 1998; Ismail et al., 2017) qui définit les connecteurs de transfert de données et les connecteurs de transfert de contrôle séparément pour la réutilisation des processus logiciels.

Compte tenu de ce qui précède, notre approche est un type intégral d'approches centrée sur le connecteur. Partant du constat que la communication est un processus dynamique et un phénomène de plus en plus complexe, elle ne peut donc plus être modélisée par un produit logiciel ou un artefact. Comme palliatif, nous estimons que la notion de processus logiciel peut être imaginée comme modèle pour la communication capable d'absorber les disparités syntaxiques et sémantiques entre les entités en interactions.

En général, les théories et modèles de communication (Lasswell & Tréanton, 1952; Shannon & Weaver, 1963) identifient certains éléments et mécanismes communs dans le processus de communication. Ils peuvent être résumés comme suit : i) Produit communiqué entre deux ou plusieurs entités en entrée/sortie (Quoi). ii) Activités pour réaliser cette communication (COMMENT). iii) Contrôleur dont le rôle est d'assurer le succès de cette communication (QUI).

Nous rencontrons exactement la même structure de communication avec des éléments ayant la même sémantique dans le concept de processus logiciel ("WorkProduct", "WorkUnit", et "Role").

Ainsi, il nous paraît évident que modéliser le monde réel de la communication à travers un processus logiciel est une voie naturelle et prometteuse pour surmonter les multiples disparités conceptuelles et techniques. De plus, « WorkUnit » peut être une tâche complexe et, par conséquent, elle peut être décomposée en plusieurs étapes. En parallèle, modéliser la communication comme un produit logiciel ou artefact large et fermé (connecteur classique) est assez limitatif et ne traduit pas fidèlement la véritable nature du processus de communication.

En outre, l'utilisation du processus logiciel pour modéliser les connecteurs nous permet d'analyser et vérifier facilement certaines propriétés de processus logiciel connues. Cette orientation permet d'appliquer certaines guidances et d'utiliser la boîte à outils et les profils disponibles.

Pour toutes ces raisons, nous intégrons dans le cadre de cette thèse le concept de processus dans le connecteur afin de modéliser sa structure et son comportement. Nous proposons une nouvelle structure qui accorde au connecteur un statut égal à celui du composant avec des ports qui exportent des services dans son interface. Une telle interface est

formée de ports ou points d'interaction utilisés en entrée/sortie par ses services. Le comportement est considéré comme le service de communication exposé par l'interface. Les services listés dans l'interface donnent l'identité et le rôle du connecteur. L'ensemble du connecteur est décrit avec les éléments et la sémantique d'un processus logiciel. Nous baptisons notre proposition : 'Connector as Process' (CaP).

De manière plus pratique, le processus logiciel est un ensemble de trois éléments liés : « Unité de travail », « Produit de travail » et « Rôle ». C'est une séquence d'activités ou « unité de travail », chaque activité consomme des produits « input work product » pour fournir un « output work product ». Le produit est sous la responsabilité d'un « Rôle » qui est en mesure de lancer et de contrôler l'exécution de l' « Activité » qui le produit.

Pour décrire la sémantique et la structure du connecteur proposé, nous présentons un méta-modèle UML qui définit les différents éléments structurels et comportementaux et leurs relations. Ces relations représentent les différentes dérivations et spécialisations ainsi que l'intégration de la structure relative aux processus logiciels. Afin de doter ce modèle d'un pouvoir plus expressif, il à été augmenté par des stéréotypes du profile SPEM pour lui donner une dimension plus pratique. Deux exemples assez distincts sont utilisés pour montrer la puissance du modèle en termes de généricité à travers un mécanisme de conformité visuelle.

4.2 Structure de l'approche proposée

Partant du fait que la communication est un processus dynamique qui peut être simple ou complexe, l'approche proposée essaye de répondre à la préoccupation principale discutée dans cette thèse. Il est question de l'abstraction de la communication dans les architectures logicielles dans un modèle conceptuel qui soit générique assez pour absorber l'hétérogénéité entre les paradigmes de développement. L'approche proposée est constitué de deux phases qui se résument comme suit :

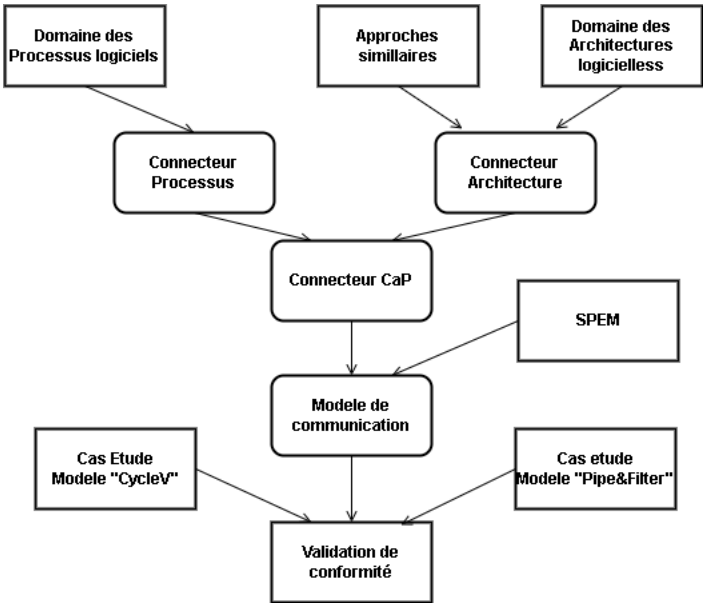


Figure 4.1- Structure de l'approche proposée.

La Figure 4.1 donne un aperçu sur les différentes phases et concepts proposés. L'étude des processus logiciels nous révèle l'existence du connecteur processus logiciel généralement destiné à l'exécution et la réutilisation des processus logiciels. L'étude du domaine d'architecture logicielle et de certaines approches similaires nous donne le connecteur architecture logicielle généralement dédié à la connexion des composants dans le paradigme orienté composants.

La première phase consiste à faire une fusion de ces deux concepts pour obtenir le connecteur CaP (pour *Connector as Process*) pour donner une première réponse de l'abstraction de la communication dans les architectures logicielles à travers un processus logiciel. Afin de le doter d'un pouvoir plus expressif, la seconde phase consiste à augmenter ce modèle par des stéréotypes du profil SPEM pour lui donner une dimension plus pratique. Deux exemples assez distincts sont utilisés pour montrer la puissance du modèle en termes de généricité à travers un mécanisme de conformité visuelle.

4.3 Motivation pour le connecteur de première classe

Dans les architectures à base de composants, un système est généralement défini comme un ensemble d'entités en interaction. Ainsi, nous avons deux concepts distincts : une partie fonctionnelle et une partie interaction appelées respectivement composants et connecteurs. Si le composant est identifié comme un élément autonome encapsulant sa fonctionnalité et qui correspond à une unité de compilation dans les langages de programmation conventionnels, les connecteurs ne le sont pas (Allen & Garlan, 1994; Garlan, 1998; Mary Shaw, 1993). Le connecteur n'a pas d'existence propre dans la phase de compilation. Il n'existe que pour répondre au besoin de communication entre d'autres entités. L'appel de procédure n'est qu'un branchement au moment de l'exécution sans aucune forme d'existence lors de la compilation.

Il est clair que les atouts majeurs du paradigme à base de composants restent la réutilisation, la composition et le dynamisme. Cependant, voir le connecteur comme une entité explicite de première classe nécessite un ensemble de conditions et de critères (propriétés) qui permettent un ensemble d'avantages intrinsèques et dérivés en fonction des domaines d'application. Nous pouvons résumer ces motivations comme suit:

- *Abstraction* : L'abstraction est le moyen par lequel, à un niveau supérieur omettant un certain nombre de détails, on peut donner l'existence à un concept afin de pouvoir le représenter, l'étudier et d'en effectuer un certain nombre d'opérations en termes de structure, comportement et évolution. Dans les architectures logicielles, le connecteur est choisi comme le seul moyen explicite de capturer et d'abstraire la communication entre les composants logiciels en interaction d'un niveau architectural à un niveau d'implémentation. Étant donné que les règles d'interaction des composants sont assez sophistiquées dans la pratique, l'abstraction des connecteurs est nécessaire pour capturer précocement toutes les décisions de conception afin qu'elles puissent être utilisées à des fins de compréhension, d'analyse ou de maintenance. En rendant les abstractions de connecteurs explicites, on peut faire des choix de conception basés sur différents schémas d'interaction, prendre en charge l'analyse de ces interactions et, dans certains cas, générer automatiquement des implémentations pour ces interactions (Garlan, 1998).
- *Le manque d'abstraction* des mécanismes utilisés pour modéliser les connecteurs ne donne pas aux connecteurs une vraie place au sein du paradigme des composants. De plus, ce manque d'abstraction des connecteurs empêchera les architectes et les développeurs de décrire, maintenir et faire évoluer facilement leurs architectures et codes associés. De ce fait, ils sont souvent amenés à appréhender l'intégralité des fonctionnalités d'un composant pour distinguer les connecteurs implicites qui s'y trouvent, leur déclaration (ensemble de composants) et leur sémantique mélangée dans le code des composants.

- *Séparation des préoccupations* : La conséquence directe de l'abstraction est la séparation de la partie traitement de la partie interaction. La déportation de l'interaction au niveau du connecteur permettra au développement du composant de se consacrer entièrement à la partie traitement et aux services qu'il doit rendre à son environnement. De plus, le développement du connecteur ignorera la complexité fonctionnelle inhérente aux composants. Ainsi, il devient possible de localiser le code du système relatif à la communication dans le connecteur puis d'éviter sa dispersion entre les composants pour le faire changer ou maintenir plus facilement en amont.
- *Réutilisation* : la séparation des préoccupations augmente en fait la réutilisation et favorise l'indépendance à la fois des composants et des connecteurs. Ainsi, un même composant peut être utilisé dans des environnements différents avec des primitives de communication différentes. Lorsque des composants et des connecteurs sont mélangés, il devient difficile d'identifier les comportements intrinsèques d'un composant ou de ses fonctionnalités. Cela compromet à la fois la réutilisation et l'évolution des composants et des connecteurs.
- *Composition* : Le composant déchargé de la partie interactionnelle devient plus adapté aux compositions ultérieures. Pour la création d'un système, un architecte verra son rôle limité aux connexions externes par le biais de rattâchements de rôles de connecteurs aux ports des composants. Il n'aura pas à intervenir à l'intérieur des composants (souvent boîte noire) pour réaliser de telles configurations.
- *Dynamisme* : le connecteur offre un potentiel supplémentaire pour prendre en charge le flux de contrôle via ses rôles et ses protocoles. La reconfiguration dynamique est un enjeu majeur pour les systèmes complexes. En cours d'exécution, il est possible d'ajouter, de supprimer et de modifier un composant sans compromettre le fonctionnement global du système. Le connecteur, défini indépendamment, jouera certainement cette action ou point de soudure lors des opérations de reconfiguration structurelle. La partie interaction ou la relation entre les composants n'est pas codée en dur dans les composants.
- *Évolution* : Le connecteur est le moyen de prendre en charge l'évolution du logiciel (au niveau du composant, du connecteur et du système). Considérer un système comme un ensemble d'éléments indépendants reliés par des mécanismes indépendants permet également de faire évoluer tout ou partie du système quelles que soient les régressions du reste des parties. L'indépendance initiale des éléments garantit leurs évolutions séparées. Ainsi, on peut faire évoluer le composant indépendamment du connecteur et inversement en plus de l'ensemble du système lui-même.
- *Simplicité* : La description explicite du connecteur réduit considérablement la complexité du composant. En effet, l'existence de nombreuses interactions entre les composants d'un système contribue à une augmentation significative de la complexité des composants. De plus, chaque nouvelle interaction ajoute de la complexité au composant. Toute décomposition ultérieure peut entraîner des incohérences et une augmentation significative du couplage entre les composants.
- *Mise en œuvre* : Le connecteur indépendant simplifie grandement la mise en œuvre du composant et du système global qui se compose de simples

attâches externes. Sinon, la mise en œuvre des connecteurs (sous la forme, par exemple, des propriétés des composants) peut être difficile. Tout d'abord, le développeur doit, dès la définition des composants, avoir réfléchi aux interactions auxquelles ses instances pourraient participer. Deuxièmement, la création dynamique de connecteurs est difficile, et il convient de souligner que certaines interactions ont des comportements dynamiques complexes. De ce fait, il appartient alors au développeur de mettre en œuvre ces mécanismes et les contrôles associés.

4.4 Approches similaires

La section (1.7.2) à été consacrée à l'étude d'une dizaine de langages de description d'architecture (ADLs) où nous avons proposé un comparatif en fonction d'un certain nombre de critères. Cette classification à été connecteur indépendante, on y trouve des connecteurs implicites et explicités. Dans cette section, nous allons proposer des études qui considèrent, toutes, le connecteur comme un élément de première instance en proposant des modèles conceptuels pour l'abstraction de la communication dans les architectures logicielles à base de composants. Cette étude sera assortie d'un comparatif qui servira de moyen de comparaison à l'approche que nous proposons.

Les approches que nous proposons cernent souvent trois axes différents : (i) les approches dédiées à l'aspect comportement où nous étudions aussi les approches théorique de classifications. (ii) les approches de structures, où nous verrons certaines variations dans les éléments du connecteur. (iii) les approches consacrées au cycle de vie du connecteur, de la conception à l'implémentation. Bien que très récurrente et légendaire dans la littérature, nous avons omis volontairement l'approche du connecteur Reo (Arbab, 2004). En effet, cette approche est centrée connecteur (elle ne tient pas compte du composant) et est destinée exclusivement à la composition. Par conséquent, elle sort du contexte de notre étude, ceci-dit, nous avons étudié quelques approches qui en sont basées comme dans (Nawaz & Sun, 2018).

La capitalisation de savoir-faire en termes de techniques, concepts et outils autour des architectures logicielles a fait l'objet de plusieurs recherches. Dans cette section, nous nous intéressons aux travaux qui considèrent le connecteur comme un élément architectural lui conférant un intérêt majeur dans la description d'architecture ou ce que l'on peut appeler des approches consacrées aux connecteurs.

En plus des approches comportementales qui précisent les éléments de base de l'existence du connecteur et leur classification et sémantique, nous nous intéressons également aux approches qui se concentrent sur le cycle de développement du connecteur, les approches qui apportent des variations structurelles et enfin les approches d'adaptation pour des usages spécifiques.

4.4.1 Approches formelles et comportementales

Mary Shaw a été la première à accorder un statut de premier ordre aux connecteurs (Mary Shaw, 1993). Allen et Garlan (1994) proposent pour la première fois la théorie des interactions entre composants avec l'idée de base de définir le connecteur comme un élément architectural indépendant ou une entité sémantique précise. Ils proposent une spécification formelle du connecteur en considérant son « rôle » comme un comportement de chacune des parties en interaction et sa « glue » comme une combinaison de ces comportements pour former une communication. Cette approche décrit ces comportements comme des protocoles d'interaction. Elle utilise une algèbre de processus CSP (Hoare, 1978) pour modéliser les traces d'événements

de communication et définie les protocoles pour les « rôles », les « ports » et la « glue ».

A ce jour, cette terminologie est encore utilisée dans la majorité des approches et des ADLs. Pour les auteurs, les « rôles » représentent les points d'interaction de l'interface des connecteurs alors que la « glue » représente une fonction de transfert ou une association entre les « rôles » d'entrée et les « rôles » de sortie d'un même connecteur.

Ehrig et al. (2004) proposent un cadre générique pour un connecteur. Ils considèrent des architectures avec plusieurs interfaces « d'importation » de connecteurs et des composants avec plusieurs interfaces « d'exportation » et des parties du corps connectées par l'imbrication, la transformation et les compositions de composants avec une sémantique de transformation compositionnelle. Le framework est instancié dans une architecture de connecteurs basée sur CSP et système de réécriture vers Petri Nets (Murata, 1989).

Liu et al. (2012) proposent une approche similaire à celle d'Ehrig et al. pour la composition/décomposition en utilisant les réseaux de Petri et la logique temporelle LTL pour exprimer les propriétés qui seront vérifiées via SPIN model Checker (Holzmann, 1997)et le langage Promela (*Process Méta LAnguage*). Les mêmes auteurs identifient également dans (C. Liu et al., 2018) un ensemble de comportements spécifiques aux connecteurs tels que l'identification de la méthode d'interaction, l'identification de la cardinalité de l'interface, l'identification du comportement du connecteur, et l'identification des multiples vues des modèles architecturaux.

Dans (Nawaz & Sun, 2018), les auteurs utilisent le prouveur de théorème PVS pour la modélisation, l'analyse et la vérification de connecteurs de composants formels. Ils utilisent un connecteur « Reo » (Arbab, 2004) modélisé avec des canaux primitifs et des opérateurs de composition utilisés pour combiner des canaux pour la construction de connecteurs complexes. Ils montrent comment modéliser et analyser le comportement des connecteurs dans PVS et prouvent quelques propriétés intéressantes des connecteurs.

Rana et al. (2019) se concentrent sur le connecteur en tant qu'élément de communication logiciel pour la construction et la composition du système. Ils définissent un langage de contrainte de flux (FCL) en tant que propriété de coordination par les connecteurs exogènes pour personnaliser le comportement des connecteurs pour la construction/l'exécution du système. Afin de vérifier la sémantique opérationnelle des contraintes FCL de leur connecteur exogène, ils utilisent des réseaux de Petri colorés pour modéliser et simuler les connecteurs avec contraintes.

Enfin, dans son analyse de 142 langages architecturaux, Mert Ozkaya (Fujita & Herrera-Viedma, 2018) indique que la majorité des langages qui prennent en charge la spécification formelle des connecteurs de haut niveau utilisent l'algèbre de processus basée sur le π-calcul pour la spécification formelle des protocoles d'interaction.

Il en ressort que les articles mentionnés ci-dessus se concentrent sur les outils et les langages formels (CSP, Promela/SPIN, réseaux de Petri et démonstration de théorème, etc.) pour décrire la sémantique des concepts et capturer le comportement du connecteur. Une telle modélisation pour le connecteur va permettre d'analyser l'interaction du système, la cohérence lors du raffinement des architectures à travers les niveaux d'abstraction et la mise en œuvre des contraintes d'interconnexion de communication.

Encore une fois, au prix de nous répéter, Mehta et al. (2000) rédigent un article de référence pour proposer une taxonomie et une classification des connecteurs en définissant les types de connecteurs de base ainsi que les catégories de services (communication,

coordination, conversion et facilitation) qui leur sont accordés. Egalement, dans (Medvidovic & Taylor, 2000), les auteurs développent une classification des langages de description d'architecture (ADL) en fonction de leurs perceptions des connecteurs. Ces deux travaux sont cités car ils sont des référentiels dans le domaine des architectures logicielles. La majorité des travaux de recherches ultérieures utilisent notamment la taxonomie de Mehta et al. et adoptent le framework de classification de la Figure 4.2. Les éléments de cette classification ont été discutés en détails dans la section (1.5.3).

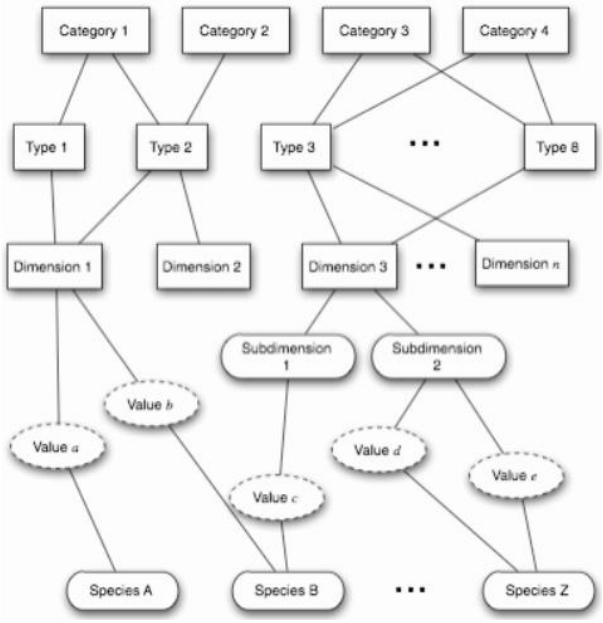


Figure 4.2- Classification des connecteurs

4.4.2 Approches de cycle de vie

Les approches qui abordent le cycle de vie du connecteur essaient de préserver le statut de première classe de la phase de spécification à celle de la mise en œuvre. Elles proposent différentes démarches essentiellement basées sur les transformations de modèles. Nous discutons de deux approches dans cette catégorie.

4.4.2.1 Approche Cariou

Dans sa contribution pour un processus de réification d'abstraction de communication, E. Cariou (2003) propose un composant « medium » pour l'interaction qu'il préserve en tant qu'entité de première classe pendant tout le cycle de vieou développement. Le processus qu'il propose est composé de plusieurs points selon le niveau de manipulation considéré en utilisant les modèles MDA. Le premier est une méthodologie de spécification de médiums en UML ou d'un contrat qu'il doit respecter le long du cycle de développement. Le deuxième élément est une architecture de déploiement de médiums qui offre la flexibilité nécessaire pour implémenter a priori n'importe quelle abstraction d'interaction. Enfin, le troisième élément est un processus de raffinement qui permet de transformer une spécification abstraite de médium en une ou plusieurs spécifications d'implémentation. Dans la terminologie de la démarche MDA, le processus de raffinement sert à transformer une spécification de niveau PIM en une ou plusieurs spécifications de niveau PSM.

La Figure 4.3 traduit en quelque sorte les éléments de cette approche et de leur emplacement ainsi que les différentes transformations effectuées le long du processus de développement ou du cycle de vie du connecteur qui imite le modèle en

« Y » de l'approche MDA. Le « rôle » est généralement attribué au développeur ou à l'architecte de l'architecture, les « unités de travail » ou les tâches du cycle de développement sont associées aux transformations successives de raffinement des modèles de spécification qui peuvent être vus comme des CIM pour obtenir des architectures ou des modèles indépendants de toute plateforme (PIM). Aussi, l'activité *Transformation* représente en quelque sorte une opération de tissage entre les PIMs -précédemment obtenus- avec les modèles de plateforme cible (PDM) avec le modèle de contraintes du domaine. L'issue de cette dernière transformation donne lieu au modèle d'implémentation (PSM).

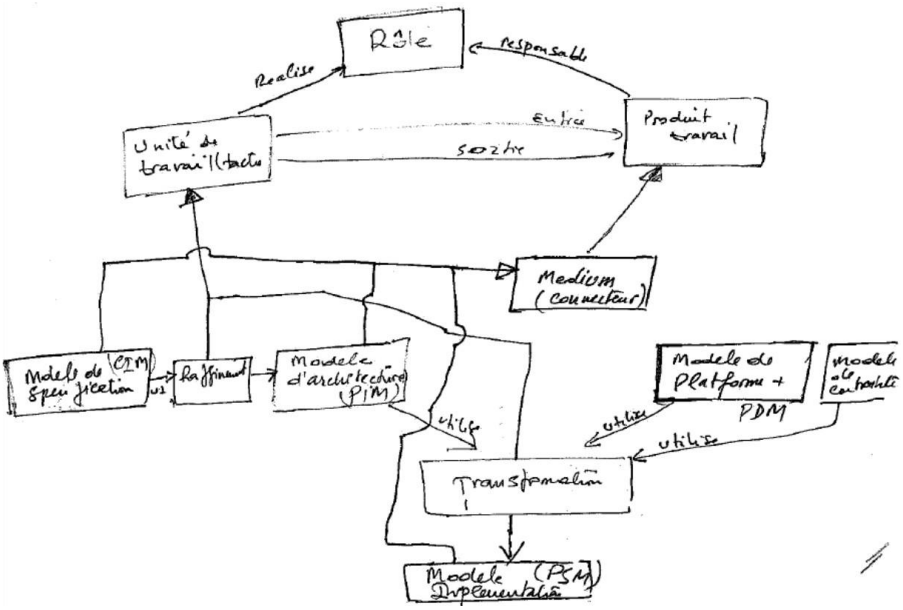


Figure 4.3- Modèle de l'approche Cariou

Cette figure (Figure 4.3) n'a pas été proposée par les auteurs, mais représente notre compréhension à nous de leur démarche que nous avons reportée avec les concepts de notre contribution (CaP) à nous. Toutefois, cette approche à comme principale avantage l'usage des normes et des standards MDA de l'OMG.

4.4.2.2 Approche Matougui et Beugnard

Dans (Matougui & Beugnard, 2005), les auteurs proposent également un processus de réification d'abstraction de la communication comme un connecteur augmenté par des générateurs. Il est question de proposer un modèle de connecteur indépendant des plateformes ainsi que son processus de d'implémentation dans le cadre du projet ACCORD (Accord, 2002). Pour cela, ils définissent un connecteur comme une entité d'architecture abstraite qui se transforme durant le cycle de développement pour se concrétiser. Ce connecteur existe par sa propriété et non pas par ses services et il définit des interfaces implicites appelées prises. Afin de préserver le statut d'entité de première classe pour le connecteur, les auteurs proposent une famille de générateurs pour son implémentation sur les différentes plateformes de mise en œuvre. En outre, cette approche distingue entre un connecteur et un composant de communication.

Dans cette approche les auteurs reconsidèrent la définition des connecteurs proposés par Mehta et all. en ajoutant les concepts de propriété, prise et protocole. La notion de propriété correspond à l'intention de communication que doit assurer le connecteur. La communication est assurée à travers les prises, qui sont les points

d'attâche du connecteur, au-dessus d'un protocole, qui représente les règles de communication ou de coordination. Ils proposent aussi une représentation sous forme d'ellipse du connecteur qu'ils jugent plus adaptée qu'à celles avec un simple trait (line) car ne pouvant pas représenté les communications complexes ainsi que celles sous forme d'un rectangle (box) carils distinguent dans leur approche le connecteur du composant de communication.

Cette approche considère aussi le connecteur comme une entité qui évolue en parcourant les niveaux d'abstraction du processus de développement en lui attribuant un cycle de vie propre indépendant du cycle de vie des composants.Ce cycle de vie est défini sur étapes, orthogonales deux par deux : deux phases et deux états. Les deux phases importantes dans le cycle de vie sont : la spécification et l'implémentation du connecteur. Les deux états du connecteur sont : *isolé*, et *assemblé* avec des composants fonctionnels.

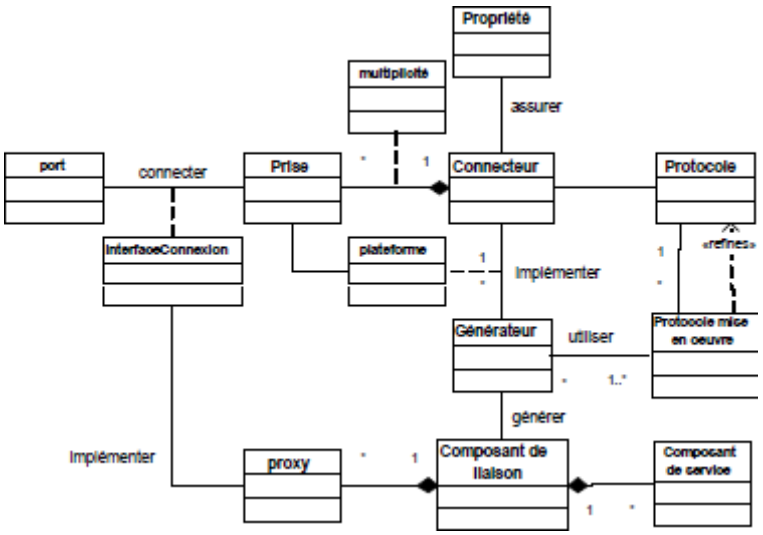


Figure 4.4- Modèle de la démarche Matougui

Pour les auteurs, le mot connecteur prend des appellations distinctes au fil de son évolution à travers le cycle de développement désignées par des noms propres à chacune des transformations de la spécification à l'implémentation pour arriver à l'assemblage et le déploiement. Dans son cycle de vie, le connecteur prend plusieurs formes : Connecteur, générateur, connexion et composant de liaison (Figure 4.4).

Appartenant à la même classe, ces deux approches (Cariou, 2003; Matougui & Beugnard, 2005) proposent un processus vertical de développement des connecteurs, de la spécification à l'implémentation et le déploiement. Elles sont, en quelque sorte, orthogonales à notre approche puisque nous cherchons à proposer un modèle de connecteur sous forme d'un processus de communication sur un même niveau d'abstraction pour un ensemble distinct de paradigmes de développement. Par contre elles sont similaires à la nôtre quand à la phase de génération de code qui concerve le statut du composant pour notre connecteur.

4.4.3 Approches structurelles

Pour arriver à leurs fins en matière de composition de toutes ses démissions, ces approches apportent des variations structurelles significatives. Deux approches sont discutées dans cette catégorie également. Elles se présentent comme suit :

4.4.3.1 L'approche Oussalah et al.

Les auteurs (Oussalah et al., 2004) donnent une définition explicite du connecteur comme une entité de première classe au même titre qu'un composant et ce à travers leur approche de description d'architecture COSA (Maillard et al., 2007; Adel Smeda et al., 2004, 2009). Ils décrivent un système comme une collection de composants qui interagissent via les connecteurs. Ils attribuent à ces éléments architecturaux un nombre de mécanismes opérationnels pour leur évolution et leur réutilisation dont : l'instanciation, l'héritage, la composition et la généricité. Définir le connecteur comme une entité de première classe permet de le redéfinir et de le réutiliser efficacement à travers des mécanismes bien définis tels que l'instanciation, la paramétrisation, l'évolution, etc. Les éléments de cette approche sont listés dans le Tableau 4.1.

Tableau 4.1- Eléments de l'approche COSA

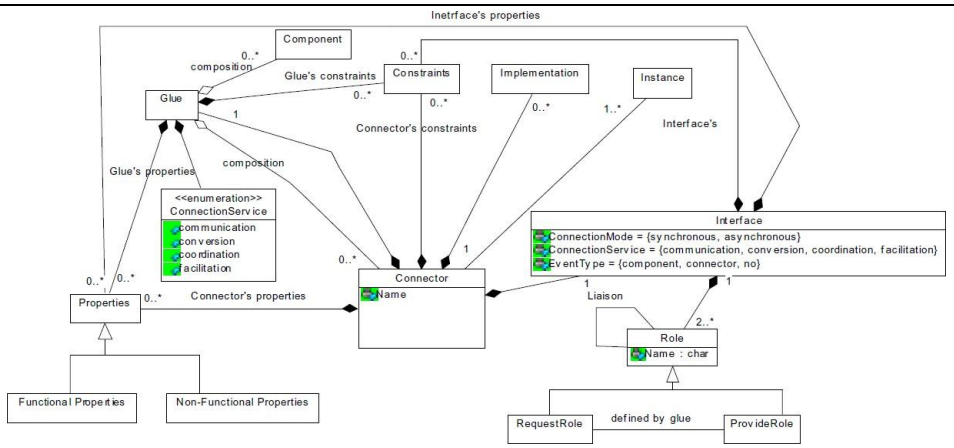


Figure 4.5- Structure abstraite d'un connecteur COSA

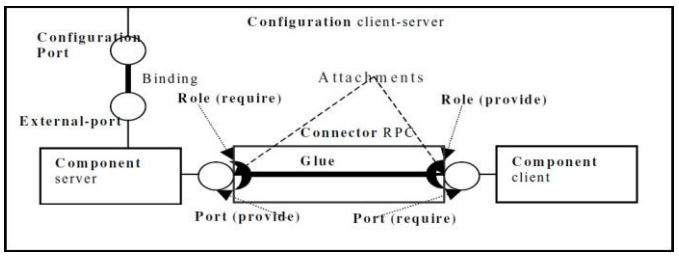


Figure 4.6- Représentation d'un connecteur COSA

Ils précisent en outre qu'un connecteur est généralement composé de deux parties principales: une partie visible qu'est l'interface et une partie cachée qu'est la glue (Figure 4.6). L'interface montre les informations nécessaires sur le connecteur comme le nombre de rôles, le type de service assuré, le mode de connexion, le mode de transfert etc. Le point d'interaction de l'interface est appelé rôle pour se connecter aux ports de composant correspondants ou aux rôles d'un autre connecteur. La glue quant à elle décrit la fonctionnalité attendue du connecteur. Elle peut être un simple protocole de liaison entre les rôles ou un protocole complexe comme la conversion de donnée, le transfert, l'adaptation etc.

La Figure 4.5 montre la structure générale ou le Méta-modèle d'un connecteur COSA, où l'on trouve les différents éléments constituants ainsi que leurs relations.

On remarque particulièrement dans cette approche le concept de composition de connecteur où ce dernier peut avoir une configuration interne contrairement aux autres approches qui considèrent la composition de connecteur sous forme de chaînage seulement comme l'approche C2 (Lun & Chi, 2010; Medvidovic et al., 1996).

4.4.3.2 L'approche Amirat et Oussalah

Dans (Abdelkrim Amirat & Oussalah, 2009), les auteurs reconsidèrent le connecteur pour en proposer une nouvelle structure afin de lui permettre d'être l'accessoire central pour la description hiérarchique des architectures logicielles (Tableau 4.2). Ils proposent en outre la description d'architectures à hiérarchies multiples (structurelle, fonctionnelle, conceptuelle, et de méta-modélisation) à travers le modèle C3 (Figure 4.7) défini comme méta modèle de description d'architecture. Cette approche identifie également le connecteur comme ayant une partie visible « interface » et une partie comportement « glu ».

Tableau 4.2- Eléments de l'approche C3

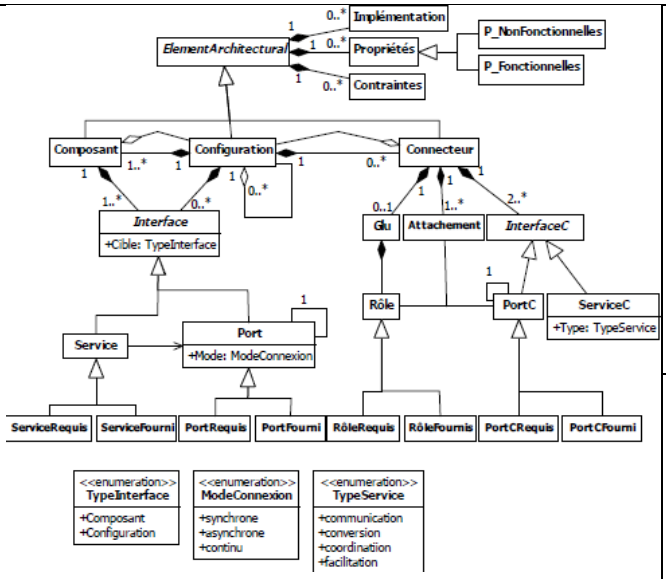


Figure 4.7- Structure générale du modèle C3

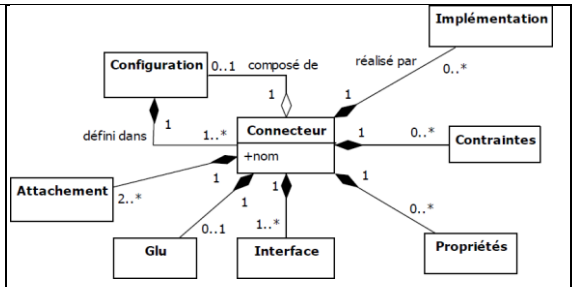


Figure 4.8- Structure d'un connecteur C3

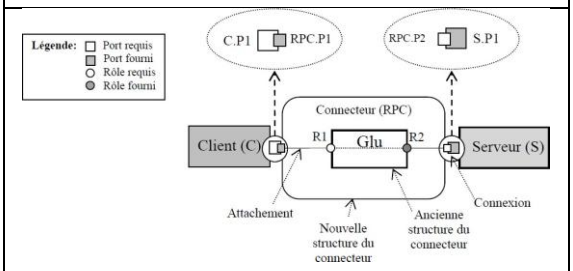


Figure 4.9- Représentation d'un connecteur C3

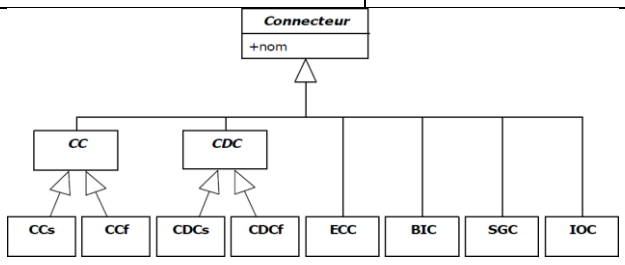


Figure 4.10- Structure de la bibliothèque de connecteurs C3.

La principale contribution de cette approche dans la structure de connecteur est d'intégrer l'attâchement à l'intérieur même du connecteur même du connecteur comme traduit par le méta modèle de la (Figure 4.8) ou dans l'exemple concret d'un connecteur RPC de la (Figure 4.9). Il en ressort directement que le connecteur est

élevé au rang de composant. Les auteurs plaident ainsi pour un gain en généricité. Ils préconisent que le concepteur d'architectures n'aura plus aucun effort à dépenser dans la description des liens d'attâchements entre connecteur et les composants compatibles. La configuration devient alors plus simple et facilement lisible par l'encapsulation des attâchements à l'intérieur du connecteur et avec une spécification exacte de la signature.

Dans leur démarche à proposer les hiérarchies multiples, les auteurs développent une bibliothèque de connecteurs typés utilisable selon l'axe d'hiérarchisation (Figure 4.10). Pour la hiérarchie structurelle, ils proposent les connecteurs de composition/décomposition structurelle-CDCs, de connexion structurelle (CCs) et le connecteur d'expansion/compression (ECC). De même pour la hiérarchie fonctionnelle avec le connecteur de décomposition/composition fonctionnelle-CDCf, de connexion fonctionnelle (CCf) et le connecteur de lien d'identité (BIC). Pour la hiérarchie conceptuelle et la hiérarchie de méta-modélisation, ils proposent respectivement le connecteur de spécialisation/généralisation (SGC) et le connecteur d'instanciation (IOC).

4.4.4 Approches spécifiques

Dans cette catégorie d'une même école, les auteurs proposent des connecteurs complexes. Leurs connecteurs sont dotés d'architectures internes pouvant assurer certaines fonctionnalités comme l'adaptation ou la sécurité. Nous présentons également deux approches dans cette catégorie qui se présentent comme suit :

4.4.4.1 Approche Derdour et al.

Pour les environnements pervasifs (*a priori hétérogènes et mobiles*), les appareils peuvent utiliser tout type de contenu allant d'un contenu textuel à des documents multimédia complexes et riches. Dans leurs approches MMSA *(Metamodel for Multimedia Software Architecture)* (Tableau 4.3) (Derdour, Roose, et al., 2010), les auteurs proposent un méta-modèle d'architecture logicielle pour les applications intégrant les propriétés des flux de données multimédia. Ce Méta-modèle permet de décrire les systèmes multimédias comme une collection de composants qui manipulent différents types et formats de données multimédias et qui interagissent via desconnecteurs d'adaptation (Figure 4.11).

L'adaptation des flux de données est déportée sur les connecteurs appelés ici connecteurs d'adaptation. Ils intègrent les services d'adaptation nécessaires ainsi que des extensions qualitatives de ces services pour offrir une mesure de qualité de service (QoS). Le superviseur est chargé de conserver les contraintes des flux reçus par l'interface d'entrée et émis par l'interface de sortie. Il est également chargé de la supervision du connecteur pour pouvoir demander sa reconfiguration ou le remplacement des services d'adaptation dans le cas où le gestionnaire de QoS n'arrive pas à offrir la qualité exigée (Figure 4.12, Figure 4.13).

Tableau 4.3- Elément de l'approche MMSA

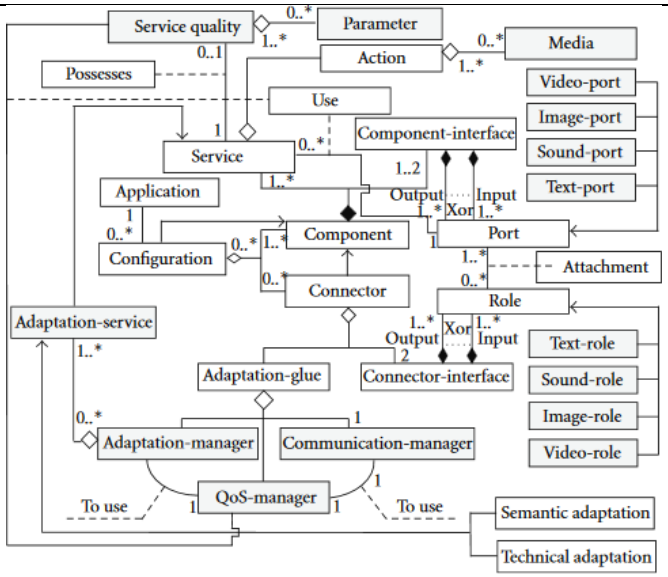


Figure 4.11- Structure de l'approche MMSA

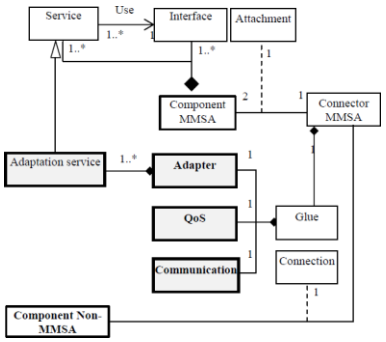


Figure 4.12- Structure d'un connecteur MMSA

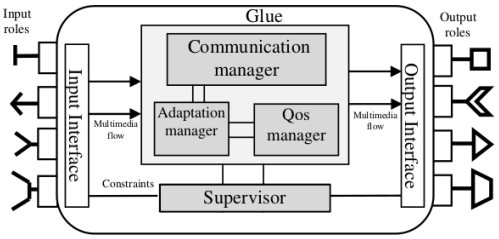


Figure 4.13- Représentation connecteur MMSA

Le modèle proposé vise à résoudre le problème d'hétérogénéité entre les données échangées entre composants en se basant sur quatre types d'interfaces selon les flux des données existantes (image, son, texte et vidéo) (Figure 4.13). Ce modèle sertà faciliter l'opération d'adaptation entre médias du même type (exemple : image vers image) ou entre différents types de médias (exemple :.texte vers son). Le connecteur MMSA est défini par deux interfaces « Inputs » et « Output » et une unité Glue représentée par trois gestionnaires : Communication, Adaptationet QoS (Figure 18). Ces gestionnaires gèrent le transfert de données entre composants ainsi que l'adaptation à opérer en cas de besoin ainsi que l'assurance et la mesure de la qualité de service. L'interface requise/offert est composée d'un ensemble de rôles types en fonction des types de médias à transférer entre composants (Figure 4.13Pour cela ils proposent une bibliothèque de connecteurs (Derdour, Dalmau, et al., 2010) pour la transformation, le transcodage et le stransmodage de l'information multimédia dont les connecteurs : *Connector V-I, Connector I-V, Connector V-S, Connector S-V, Connector S-T, Connector T-S, Connector T-I,* etc. L'issue de ce travail a débouché sur la proposition par les mêmes auteurs (Derdour et al., 2012) du développement d'une plateforme CSC (*Component Service Connector*) qui repose sur un modèle composant/service permettant l'adaptation d'applications à base de composants et utilise une architecture orientée service pour offrir des services d'adaptation à intégrer dans des connecteurs d'adaptation.

4.4.4.2 Approche Alti et al.

Dans le même sens que l'approche précédente, mais dans le cadre du typage et de la spécialisation des connecteurs, Alti et al.,(2015) proposent un connecteur de sécurité pour assurer un assemblage sécurisé afin de prendre en charge la vulnérabilité liée au transfert de flot de données dans les environnement de déploiement instables lié à un Internet et aux

communications sensibles. Ils définissent l'approche SMSA (Security Méta-model for Software Architecture) (Tableau 4.4). Elle est basée sur un méta-modèle (Figure 4.14) générique pour décrire un système comme une collection de composants qui interagissent via un connecteur de sécurité. Ils proposent également un système de transformation de modèle pour intégrer le connecteur de sécurité. Ils construisent un profil UML 2.0 pour SMSA à même de définir une spécification complète pour intégrer les nouveaux concepts de sécurité à UML.

Le connecteur proposé intègre les services de sécurité requis ainsi que l'extension qualitative de ces services pour mesurer la qualité de service (QoS) qui reflète l'évolution des besoins de sécurité pour le flux de données échangé entre composants. Les auteurs discutent trois types d'aspect de sécurité (authentification, confidentialité et intégrité) et proposent également une bibliothèque de connecteurs y afférents. Les connecteurs proposés sont : connecteur de d'authentification (Figure 4.16), connecteur de confidentialité et le connecteur d'intégrité.

Tableau 4.4- Eléments de l'approche SMSA

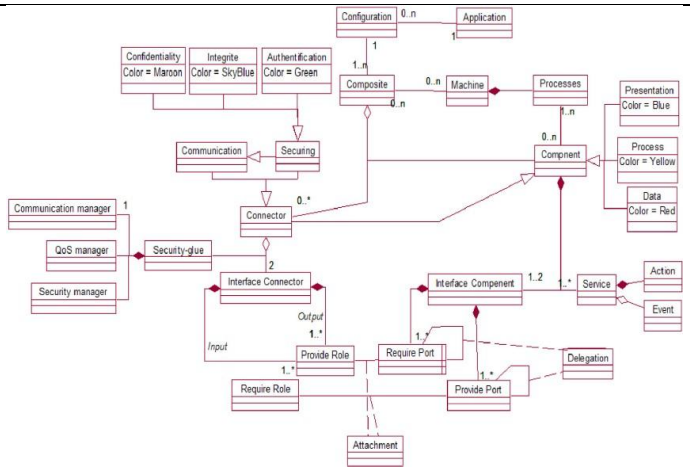


Figure 4.14- Structure de l'approche SMSA

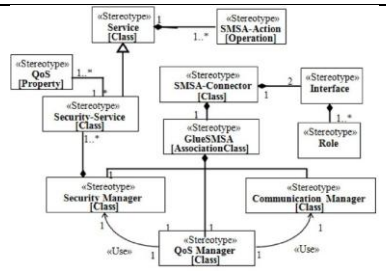


Figure 4.15- Structure du connecteur MMSA.

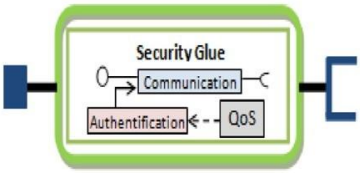


Figure 4.16- Représentation du connecteur d'authentification MMSA.

Le connecteur SMSA (Figure 4.15) inclut la partie visible qui est l'interface (input/ output) qui décrit les rôles qui définissent le mode communication (synchrone, asynchrone ou continu) ainsi que le type de connexion (GPRS, WAP, MMS, etc.) entre les composants. La deuxième partie est la glue qui implémente les mécanismes de sécurité pour la communication ou l'échange des informations et des services pour sécuriser et gérer la qualité de service des composants. La glue est représentée par trois gestionnaires : communication, sécurité et QoS. Elle gère le transfert des données entre composants en assurant les opérations de sécurité nécessaires.

4.4.5 Synthèse

Outre les approches formelles de spécification ou de classification qui définissent la base théorique des connecteurs, le reste des approches propose soit une revue du cycle de

développement, des structures, des hiérarchies et une spécialisation pour des cas particuliers. Aucun d'entre eux ne considère la partie générique et comportementale comme un processus de communication visible à travers l'interface du connecteur pour gagner en efficacité et en réutilisabilité comme notre approche prétend le remplir.

Au terme de cette étude nous avons identifié un ensemble de critères motivationnels ou de propriétés intrinsèques aux connecteurs qui nous permettent d'avoir un regard comparatif (Tableau 4.5) autour des approches étudiées et qui peuvent se résumer comme suit :

- Abstraction : traite du mode, des notations, du langage d'abstraction utilisé, et des différentes opérations permise par cette abstraction sur le connecteur.
- Séparation des préoccupations : Ce critère doit spécifier s'il existe une séparation précise de la communication entre le connecteur et le composant et si elle doit être maintenue dans les différentes phases du cycle de vie.
- Réutilisation & adaptation : il s'agit de déterminer les mécanismes associés à la réutilisation en précisant la capacité d'adaptation et de typage.
- Potentiel de granularité : il faut préciser la volumétrie du connecteur, un maillon simple (faible) ou ayant un certain nombre d'éléments (moyen) ou gros grains (Fort).
- Dynamicité : ceci spécifie les différentes techniques qui peuvent être utilisées pour assurer la dynamicité du système. Il se réfère particulièrement aux techniques utilisées pour la description du comportement.
- Evolution : précise les possibilités et les mécanismes ainsi que les éléments concernés par le processus d'évolution.
- Catégorisation : Identifie l'existence de catégories de connecteurs et leur niveau. Il détermine si l'approche appartient à une classe précise et distinctive.
- Préservation dans le cycle de vie : précise l'existence et la manière avec laquelle le connecteur peut conserver son statut d'entité de premier ordre tout au long du Cycle de Vie. Ce critère est essentiel dans la mesure qu'il va permettre entre autres la configuration dynamique dans les architectures logicielles.

Toutes les approches décrites dans cette section traitent le connecteur comme une instance de première classe. Elles discutent et partagent parfois tout ce qui a trait au monde des connecteurs dans les architectures logicielles à base de composants. Elles s'articulent essentiellement autour de quatre axes: Structure, comportement, classification et le cycle de vie des connecteurs, de la conception à la mise en œuvre.

Il est important de signaler que les approches que nous appelons théoriques ou comportementales ont défini au préalable le jargon et la taxonomie relative et qui restent d'usage à ce jour depuis la création du concept de connecteur, il y a une trentaine d'années. La majorité des contributions actuelles dans la recherche appartiennent à cette catégorie. Leur assise formelle pour la description du comportement ou du protocole d'interaction est un atout de leur adoption. C'est un support clé pour l'analyse des architectures logicielles souvent prise en charge dans les langages de description d'architectures.

Les approches qui traitent du cycle de vie ont tout de même fait un effort particulier pour la réification des connecteurs. L'approche Cariou a eu l'audace et l'avantage d'avoir adopté très tôt les standards (MDA) presque à leur naissance. L'approche Maatougui et

Beugnard a eu le privilège d’avoir servi dans un projet académique et industriel dans le cadre de la télécommunication. Avoir la preuve de conserver le statut de première instance du connecteur de la conception à la réalisation est un atout fondamental et sélectif pour la prise en charge, en autre, de la reconfiguration dynamique très récurrentes dans les applications de nos jours.

Les approches Oussalah et al. et Amirat et Oussalah, ont apporté des ramifications significatives dans la structure des connecteurs pour répondre à des usages particuliers. On dénombre la composition des connecteurs et les mécanismes opérationnels dans l’ADL COSA soit la description hiérarchique des architectures logicielles dans le modèle C3.

Les approches Derdour et al. ainsi que Alti et al. proposent des changements conséquents en proposant des connecteurs lourds et complexes. Leurs contributions apportent des distinctions tant dans l’interface que dans le comportement interne. Ils proposent des connecteurs avec une architecture interne pour la prise en charge de certaines vocations, en plus de la communication, comme l’adaptation et la sécurité des transferts.

Notre approche est à la fois structurelle et comportementale avec une variation sémantique qui considère le connecteur comme un processus d'abstraction de communications complexes et hétérogènes.

Tableau 4.5- Synthèses des approches étudiées.

Approches	Abstraction	Séparation préocup	Réutilisation Adaptation	Potentiel de granularité	Dynamique	Evolution	Catégorisation	Préservation Cycle de vie
Allen & Garlan (1994)	CSP	Non	instanciation	Faible	Protocole CSP	-	Formelle	Oui
Mehta (2000)	Graphe	Non	Framework	Faible	-	-	Categorie Service Type Connecteur	-
Cariou (2003)	Spécification UML Contrat Raffinement	Oui	Tissage/code génération modèle	Moyen	Aucune précision sur le protocole	-.	Dans le cycle de vie: PIM, PDM, PSM	Raffinement Modèle "Y"
Ehrig (2004)	Graphe Import/Export Interface	Non	Generic framework Instantiation	Connecteur imbriqué	CSP, Réseaux de Pétri	-	Formelle	par réécriture de Graphe
Oussalah (2004)	Pseudo code ACME instanciation Composition Parametrage	Oui	Instanciation Héritage Typesetting	Connecteur composite	Aucune précision sur le protocole	Sous-typage structurel via le mécanisme "extends"	-	Oui
Matougui (2005)	Notation Ellipse vs line/box Modèle Processus Implémentation	Oui	Générateur pour différentes plateformes	Moyen	Aucune précision sur le protocole	-	Dans le cycle:de vie Connecteur, Generateur, Connexion, Medium	Evolution dans le cycle de vie
Amirat (2009)	Pseudo code ACME Hiérarchie	Oui	Héritage Sélection dans libraire	Composition hiérarchique	Aucune précision sur le protocole	Sous-typage structurel via le mécanisme "extends"	Hiérarchie: structurelle, fonctionnelle, Conceptuelle, MModélisation	-
Dardour et al. (2010) Alti et al.(2015)	Notation graphique Conversion de Type & Data Format Assemblage, sécurité	Oui	Connecteur dédié	Multiport Type + Architecture interne	Gestionnaire communication	-	Adaptation: Son, Video, Image, Text. Security	Oui
Liu et al. (2012, 2018)	Graph	Non	Composition/Decomposition	Hight	R.Petri, Promela operation Specifique	-	Formelle	Oui avec SPIN Model-checker
Nawaz (2018)	Canal Primitif Composition operateur /Reo	Non	Composition	Connecteur composite		-	Formelle	prouver théorème PVS
Rana et al. (2019)	Élément Communication.	Oui	Adaptation connecteur exogène	Faible	R. Petri Coloré simulation FCL	-	Formelle	-

4.5 Approche CaP

Toutes les approches qui traitent le connecteur comme une entité de première classe le définissent et le considèrent toujours comme un produit logiciel ou un artefact. Nous considérons cette tendance pas assez généraliste et favorise la spécialisation. Dans notre approche (CaP : *Connector as Process*) (Menasria et al., 2021), en revanche, nous considérons le connecteur comme un processus logiciel ouvert dédié à la description, de manière naturelle, de l'abstraction d'une communication complexe.

La dualité produit/processus logiciel dans les architectures logicielles est mise en évidence par Boehm (1995) et Bhuta et al, (2005) qui précisent que « Si une approche donnée (programmation disciplinée, définition et validation des exigences, réutilisation, gestion des risques) est bonne pour les produits logiciels, alors son homologue de processus est bon pour les processus logiciels ». Plus simplement, cette précision est appuyée par le constat d'Osterweil (2011) qui stipule que « Les processus logiciels sont aussi des logiciels » ainsi que Bendraou et al. (2008) qui précisent que « L'excitabilité des modèles de processus logiciel » et par conséquent ces mêmes processus peuvent être réutilisés et manipulés par les outils.

L'idée de base de notre approche est d'élever le connecteur au rang de composant ayant une partie interne et une interface. L'interface est formée de ports typés ou de points d'interaction et fournit des listes de services d'interaction typés avec une sémantique précise. De plus, le connecteur est considéré comme un processus de communication. Partant de l'idée que l'interaction entre les composants peut varier d'un aspect très simple de point à point à un aspect très complexe, notre connecteur est vu comme un processus global et entier de communication. Nous considérons que la modélisation du comportement du connecteur par un processus logiciel va être une idée assez générique qui peut absorber ces variations et apporter une solution compacte pour la gestion de la complexité en prenant en charge l'hétérogénéité et en favorisant l'interopérabilité.

4.5.1 Notion base de processus logiciel

Puisque notre principale contribution consiste à abstraire le comportement et les services des connecteurs à travers les mécanismes de processus logiciels, nous devons refaire appel aux différents éléments connus des processus logiciels. Nous avons déjà discuté en détail ces notions dans le chapitre 2, nous nous limiterons dans cette section aux seuls concepts essentiels qui décrivent le cœur du méta-modèle SPEM. Nous utilisons ces éléments pour décrire notre proposition loin des nombreuses classes et packages qui rendent souvent difficile la compréhension du profil SPEM lui-même (Bendraou et al., 2007; Shaked & Reich, 2021).

Le processus logiciel est une représentation détaillée des diverses tâches, techniques, outils et ressources à utiliser pour produire un produit logiciel. Il sert de représentation de base pour la planification, la coordination, l'exécution et le contrôle du développement logiciel. Les séquences de tâches sont des séquences d'actions linéaires, sélectives, itératives et parallèles ou partiellement ordonnées. Les modèles de processus logiciels ont une richesse sémantique et syntaxique qui leur permet d'être exécutables.

De manière plus pratique et plus simple, le processus logiciel est une séquence d'activités ou "unité de travail", chaque activité nécessite ou utilise un produit d'entrée ou "produit de travail" (entrées) pour fournir un produit de sortie (sorties). Le produit est sous la responsabilité d'un « Rôle » ou un réalisateur et contrôleur qui est en mesure de lancer et de contrôler l'exécution de l'activité qui le produit. Autrement dit, un processus de

développement logiciel est une collaboration entre des entités abstraites actives appelées « rôles » qui effectuent des opérations appelées « activités » sur des entités concrètes et réelles appelées « produit ».

Les différents rôles agissent les uns sur les autres ou collaborent en échangeant des produits et en déclenchant l'exécution de certaines activités. L'objectif d'un processus est de conduire un ensemble de produits à un état bien défini.

La Figure 4.17 montre le méta-modèle du processus logiciel qui représente le noyau conceptuel de la structure de SPEM ou de tout autre langage de modélisation des processus logiciel (Aoussat et al., 2014; Combemale & Crégut, 2006) où l'on retrouve les trois entités et les classes de base avec les relations qui les lient:

- « Activité » : Représenter l'unité de travail effectuée. Les séquences et les états d'avancement des activités peuvent être décrits dans une séquence d'étapes. C'est le « rôle » qui exécute l'activité. Chaque activité est sous la responsabilité d'un seul rôle. Plusieurs activités sont liées par des relations de précédence entre elles.
- « WorkProduct » : représentent les produits manipulés en entrée-sortie. Les transmissions, le format, les versions et le stockage de ces produits peuvent également être traités. Les produits représentent les entrées/sorties des activités. Ils ont sous la responsabilité du même rôle qui exécute l'activité.
- « Rôle » : décrire, en général, les responsabilités et qualifications requises ou existantes qu'un acteur peut ou doit avoir lors du développement d'un logiciel ou simplement lors de l'exécution d'une tâche. Il exécute, contrôle les activités et leurs produits.

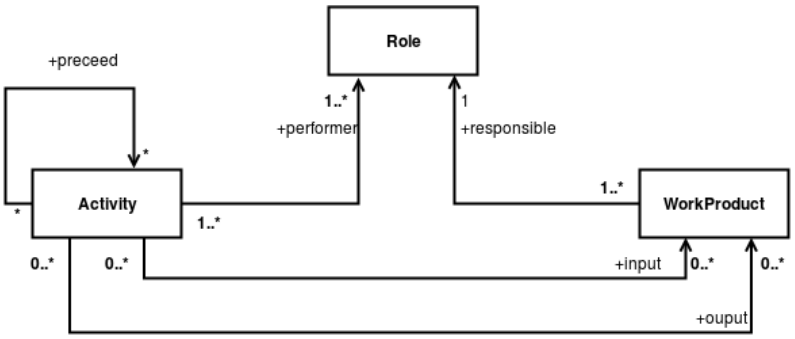


Figure 4.17- Méta-modèle de base du processus logiciel.

En parallèle, certains auteurs de la communauté des processus logiciels (Aoussat et al., 2014; Dai et al., 2008; Dami et al., 1998) traduisent de manière similaire l'utilisation d'architectures logicielles pour la réutilisation de processus logiciels. Nous adoptons la même approche pour intégrer la notion et les concepts de processus logiciel au sein même du connecteur, notamment, pour modéliser sa partie comportementale pour supporter les communications complexes.

Pour exécuter et réutiliser les processus logiciels dans le cycle de développement, cette communauté identifie à ce propos deux concepts clés pour se rapprocher de du domaine des architectures logicielles et des descriptions architecturales qui sont : le composant processus et le connecteur processus. Une configuration représentera alors un modèle quelconque de processus qui sera en finalité un ensemble de composants processus reliés entre eux par des connecteurs processus. Egalement, le méta-modèle SPEM identifie la notion de « composant processus » dans son package « *MethodPlugin* ». On peut définir sommairement ces concepts comme suit :

- Composant processus : décrit un traitement réalisé sur des produits en entrée pour « la création » de nouveaux produits en sortie. Il implique une interface composée essentiellement de ports de données (transmission des produits en entrée sortie) et des ports de contrôle (transmission du flux d'exécution) qui permettent d'identifier l'ordre et l'état d'exécution.
- Connecteur processus : décrit le traitement à réaliser sur des produits en entrée afin de « les adapter ou les contrôler » pour les besoins du composant processus connecté. Il ne fait pas de création de produits. Son interface est formée de « Rôle » qui représente l'image du port chez le composant. Les rôles représentent des points d'interaction pour la transmission des produits et des flux d'exécution à travers les rôles de données et les rôles de contrôle.
-

4.5.2 Structure générale du connecteur

L'objectif principal du connecteur est d'assurer la communication entre les composants. Dans notre approche CaP, nous proposons de décrire cette communication ou connecteur comme un processus logiciel, ce qui nécessite une fusion sémantique de deux domaines : l'architecture logicielle et le processus logiciel. Pour cela, il faut donc essayer de réunir et intégrer les éléments de base du processus logiciel dans les constituants structurels et comportementaux du connecteur.

Ainsi, nous assumons les correspondances suivantes pour rapprocher les disparités sémantiques qui permettent entre autres d'élever le statut du connecteur au rang de composant de communication ayant une interface et assurant un ensemble de services:

4.5.2.1 Port

Le terme « port » est volontairement choisi pour assurer dans un premier temps une homogénéité syntaxique avec le composant. Il se substitue alors au terme « rôle », communément admis dans les architectures logicielles pour décrire les points d'interactions dans l'interface. Aussi, le terme « rôle » est remplacé par le terme « port » pour éviter toute confusion avec le concept « rôle » dans les processus logiciels. Ainsi définis, les ports représentent les points d'intégration du connecteur ou composant de communication avec l'extérieur pour former son interface.

Aussi, le port joue le rôle de "Produit du travail" car il représente la ressource ou le produit consommé et produit par le processus de communication ou l'activité. Ce choix représente une idée essentielle dans notre proposition. De facto, nous avons deux types de ports : (i) les ports requis qui sont les point d'entrée des informations et services en amont utilisés par le connecteur, les ports fournis qui sont les point de sortie des informations et services produits pour son environnement. Contrairement aux composants, le connecteur doit avoir au moins deux ports (un en entrée et un en sortie) tandis que le composant peut n'en avoir qu'un seul fourni. Un connecteur doit son existence pour relier au moins deux composants.

4.5.2.2 Service

Généralement, un service dans le développement orienté composant est dans sa forme minimale soit un message, soit une opération ou une variable. A notre niveau, le terme est également choisi pour des raisons syntaxiques avec le composant ainsi que sémantique avec le processus en représentant « l'activité » du processus de communication ou « unité de travail ». Cette activité va consommer et produire des informations depuis et vers les ports. Le service

assure également la connexion entre les ports d'entrée et de sortie, il joue le rôle de la « glu » ou de la fonction de transfert qui peut effectuer des traitements et des stockages particuliers.

4.5.2.3 Acteur

Acteur : Représente l'élément distinctif associé au concept de processus et constitue désormais un constituant de base du connecteur proposé pour agir en tant que superviseur de la communication. Dans les applications sensibles, le transfert ne doit être autorisé que sous certaines conditions. Il peut être automatique ou manuel (humain). Ce qu'on appelle, il remplace l'entité « Rôle » du processus pour éviter toute confusion avec le traditionnel « Rôle » dans la taxonomie des connecteurs.

Constamment, l'acteur surveille les ports. A toute nouvelle information ou service disponible dans les ports d'entrée, il cherche à déclencher et contrôler l'action ou l'activité associée en fonction de la sémantique du connecteur qui le renferme. De la même manière, il assure l'exportation du service vers le port de sortie correspondant.

4.5.2.4 Forme générale

La structure générale du connecteur CaP est décrite de manière compacte dans la Figure 4.18 sous la forme d'un composant de communication. Elle est constituée des éléments de base sus-cités. Elle doit contenir aussi les propriétés et contraintes globales (Oussalah, 2014; Tibermacine et al., 2017) du connecteur ainsi que celles des trois concepts de qui le constituent. « Port » et « Service » peuvent être concernés par certaines règles d'évolution (Gasmallah et al., 2019) pour répondre à de nouvelles exigences.

L'ensemble du connecteur est décrit par le processus logiciel et ses éléments. Nous ne traitons pas ces aspects dans le cadre de cette thèse, nous les évoquons seulement à titre indicatif. Toutefois, nous évoquons quelques contraintes nécessaires dans les sections suivantes ainsi que dans le chapitre 5.

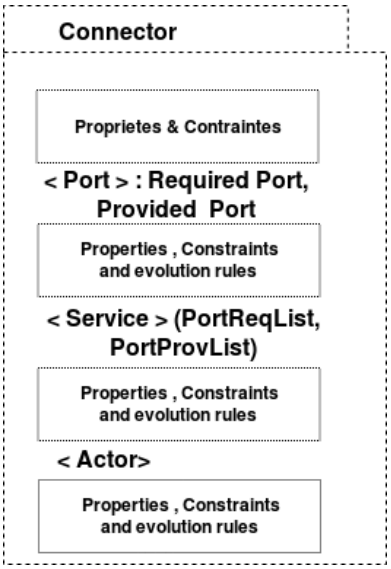


Figure 4.18- Structure générale du connecteur.

L'interface est la partie visible du connecteur ; elle est essentiellement constituée de ports et des services d'interactions qu'il exporte. Ces services formeront l'identité et le rôle du connecteur sur étagère pour l'indexation et la réutilisation. La partie cachée définit et représente le service minimum qui peut être « glu » ou les différentes fonctions de transformations et transferts. Le service peut être mis en œuvre sous la forme d'un simple lien, d'un protocole complexe ou d'un processus complet. L'acteur agit également sur le connecteur

à travers son interface qui représente le point d'action. Ainsi, nous n'obtenons plus un connecteur vu seulement comme une entité de première classe, mais ayant également le même degré de citoyenneté en tant que composant.

Dans la suite, nous considérons le connecteur selon trois axes fondamentaux. i) La structure : définit les constituants de base du bâtiment qui forment les bases du connecteur, leurs représentations et leurs relations. ii) Comportement : définit l'ensemble des actions ou opérations qu'un connecteur est capable d'effectuer au cours de son fonctionnement. Il décrit, à travers ses services, la dynamique résultante de l'exécution d'une fonction de communication ainsi que son aspect réactif. iii) Evolution : reflète les changements de la structure et/ou du comportement du connecteur dans le temps et en fonction de l'évolution de l'environnement ou des exigences initiales. Elle concerne l'ajout, la suppression ou la modification d'éléments structurels tels que les ports ou comportementaux tels que les services.

4.5.3 Connecteur et processus

Il est évident de rappeler que la problématique étudiée dans le cadre de cette thèse est de proposer une abstraction de la communication dans les architectures logicielles. La réponse intuitive et triviale est de faire appelle aux techniques de méta-modélisation discutées dans la section 3.2). Dans notre contexte, nous considérons les verbes « abstraire » et « méta-modéliser » comme synonymes.

A ce stade, nous allons essayer de formaliser les concepts et les idées discutées dans les deux sections précédentes. À partir des Figure 4.17 et Figure 4.18 qui décrivent respectivement le méta-modèle des processus logiciels et la structure du connecteur pour architectures logicielles, nous proposons la partie principale du méta-modèle de notre connecteur CaP (Figure 4.19). Ce méta-modèle imbrique tous les concepts de base de notre contribution avec toutes les relations associées. Il est évident de rappeler

Le méta-modèle proposé prend en compte le fait de considérer la communication (le connecteur) comme un processus complexe et pas seulement des connexions point à point comme la majorité des approches similaires étudiées plus haut. De ce fait, ce méta-modèle ne peut que fédérer et fusionner deux domaines complémentaires issus de deux communautés différentes : la communauté des architectures logicielles et la communauté de la modélisation des processus logiciels.

Celui-ci décrira la communication, dans sa partie comportementale, entre composants sous la forme d'un processus. Cette approche vise à étendre et reconstruire le concept de connecteur tel que défini dans le domaine des architectures logicielles en intégrant principalement les éléments qui définissent le processus logiciel. Cependant, une extension supplémentaire pour d'autres communautés et paradigmes peut également être imaginée selon d'autres exigences.

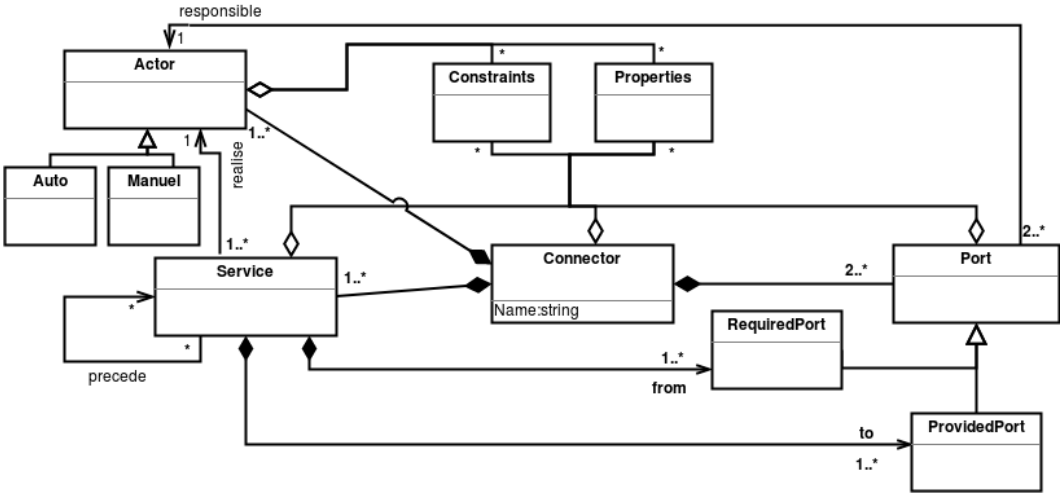


Figure 4.19- Méta-modèle de base du connecteur.

Comme un composant, le connecteur assure des services de communication pour les composants qui le sollicitent pour interagir via ses ports requis/fournis. La structure du connecteur proposé est décrite dans sa syntaxe abstraite qui montre la structure des différents éléments fondamentaux et leurs relations (Figure 4.19). Il constituera le support de base sur lequel s'ajouteront les éléments liés au concept de processus ainsi que quelques éléments de spécialisation supplémentaires de notre connecteur.

Dans le méta-modèle proposé pour le connecteur, on retrouve les entités du processus logiciel et on identifie les classes suivantes :

- La classe « Connector » représente la classe englobante du connecteur proposé. Elle est composée des classes « Actor », « Service » et « Port » avec une relation de composition forte. L'élimination de l'une de ces classes détruit le connecteur lui-même. Le connecteur peut avoir au moins deux ports (entrée/sortie) et un ou plusieurs services.
- La classe « Service » représente l'« unité de travail » ou les différentes « activités » et services fournis par le connecteur. Nous confondons donc à ce niveau le concept de « Service » des architectures logicielles et le concept « Activité » des processus logiciels. Le service est sous le contrôle de l'(acteur » et il utilise le port requis la relation « from » pour produire un port fournis avec la relation « to ». Les relations de composition avec les classes « port » est justifié par le fait que le port est un élément constitutif du service. Sans les ports, le service n'a pas de raison d'être.
- De plus, nous avons la classe « Acteur » au lieu de « Rôle » afin d'éviter toute confusion avec le rôle de connecteur. Ces deux entités sont les principaux éléments qui décrivent le comportement du connecteur. Le Service est modélisé, déclenché, dirigé et supervisé par l'Acteur qui peut être soit un programme, un composant, le connecteur lui-même, ou un humain.
- La classe « Port » concerne le dernier élément du processus logiciel à savoir le « work product », que nous l'avons volontairement associé au « Port » du connecteur. Le Service est censé consommer et produire des flux d'entrée/sortie véhiculés par des ports de connecteurs. Les ports sont naturellement de deux types requis « RequiredPort » et fournis « ProvidedPort ». Les informations du port requis représentent les

exigences du connecteur de son environnement. Le Port représente le « work product » de l'Activité du processus (en tant que constituant obligatoire) sous la responsabilité de l'Acteur.

- Les classes « Constraints » et « Properties » sont associées a chacune des autres classes par une relation d'agrégation pour justifier leurs optionalités.

4.5.4 Méta-modèle des services du connecteur

Afin de reconstruire la définition du connecteur selon notre approche, il est nécessaire de le reconsidérer d'abord dans sa définition atomique et élémentaire. Dans (Clement et al., 2011; Mehta et al., 2000; M Shaw & Garlan, 1993; Taylor, 2019; Taylor et al., 2009), les auteurs soutiennent que le connecteur, quel que soitsa complexité, n'est autre que des mécanismes de transfert de données et/ou de contrôle le long d'une conduite ou un tuyau.

Pour réaliser une telle construction, nous recommandons que le connecteur soit défini dans sa forme la plus basique. A cet effet, nous proposons les deux seules variantes de connecteurs et statuons en définitive que la fonctionnalité de tout connecteur peut être réduite à un transfert de données ou à un transfert de contrôle en proposant des connecteurs de flux de données et de flux de contrôle (Mary Shaw & Garlan, 1996).

Cette orientation ne peut que conforter également la communauté des processus logiciels qui séparent par définition et très clairement les connecteurs dits de flux de contrôle et les connecteurs dits de flux de données. Cette séparation de préoccupation contrôle/données est à même de permettre au connecteur de gagner en généricité. Ceci représente une propriété tant attendue dans notre proposition.

Le connecteur de flux de données permet le transfert de données en consommant, en transformant et en produisant des services entre les composants tandis que le connecteur de flux de contrôle est chargé de contrôler ce transfert et son exécution en termes de priorité, de sélection, d'itération, d'interruption, de reprise et de gestion des exceptions. Chaque connecteur est composé d'un port dédié à savoir un port de flux de contrôle et un port de flux de données. Le méta-modèle de la Figure 4.20 apporte les changements nécessaires tant sur la classe « Service » que la classe « Port ».

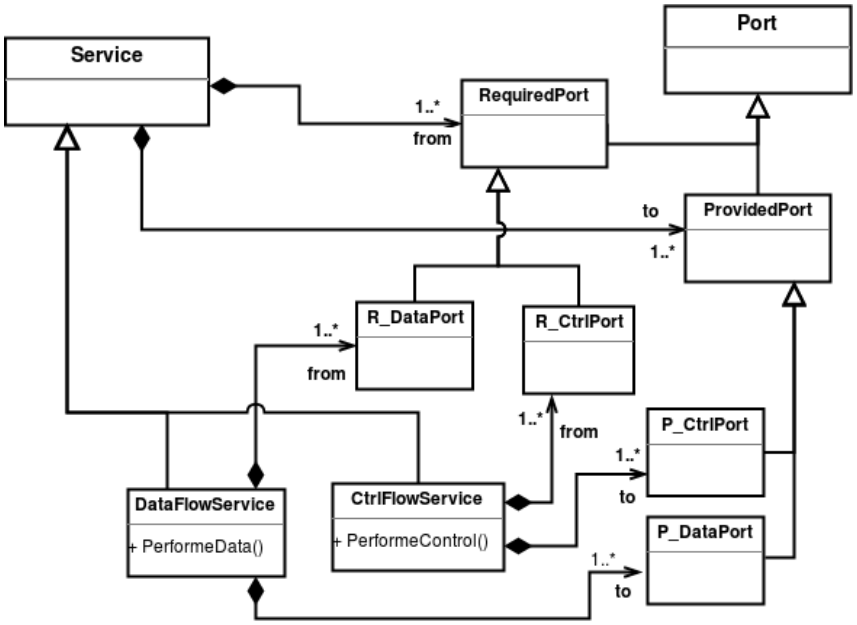


Figure 4.20- Méta-modèle des services/ports du connecteur

Pour répondre à cette distinction entre le contrôle et les données, des changements ont été apportés à la fois à la structure et au comportement sur le méta-modèle initial. Ainsi, deux types de ports sont ajoutés: les ports pour le transfert de données et les ports pour le transfert de contrôle. Ce qui est traduit par les classes suivantes :

- Les classes « R_CtrlPort » et « R_DataPort » désignent les ports requis pour le flux de contrôle et le flux de données. Ils acceptent ces deux flux séparément en entrée à partir de l'environnement amont.
- Les classes « P_CtrlPort » et « P_DataPort » désignent les ports fournis pour le flux de contrôle et le flux de données. Après les différents traitements, ils délivrent ces deux flux séparément en sortie pour les composants demandeurs ou pour l'environnement en aval.
- Les rôles de l'association ("from", "to") indiquent la sémantique des extrémités de notre connecteur.

La partie comportement est matérialisée par le concept « Service » ; il existe également deux éléments de base : les services associés au transfert de données et les services associés au transfert de contrôle. Ce qui traduit par les classes suivantes :

- La classe « DataFlowService » qui désigne le traitement associé et le transfert du flux de données. Le service de flux de données récupère également le flux de données du port requis correspondant, le traite en fonction du flux de contrôle et de la manipulation de service souhaitée, et le réplique sur le port de données fourni.
- La classe « CtrlFlowService » qui désigne le traitement associé et le transfert du flux de contrôle. Ce service récupère les informations du flux de contrôle à partir du port de flux de contrôle requis, le traite et le réplique vers le port de flux de contrôle fournit.
- Les services mettent aussi en relation les ports en entrées et les ports en sorties. Les services sont intimement liés aux ports correspondants ce qui justifie la relation de composition entre les classes des ports avec les

classes de services correspondantes.

- Selon les relations d'héritage sur le méta-modèle, ils peuvent être instanciés dans un connecteur de flux de contrôle, un connecteur de flux de données ou les deux dans le même connecteur. A noter que la nature d'un connecteur dépend de la nature de son service.
-

4.5.5 Bibliothèque de connecteur

Dans notre approche, le service représente l'abstraction du comportement du connecteur. Il peut s'agir d'une homogénéisation de différentes appellations données dans d'autres approches telles que « glu », « adaptation » ou « multiplexage », …etc. L'objectif principal d'une telle vision est d'obtenir une correspondance syntaxique avec le composant pour produire un connecteur « On-the-Shelf » qui peut appartenir à une bibliothèque de connecteurs réutilisables.

Cependant, il y a une distinction importante à faire : le composant possède des services à usage général tandis que le connecteur possède des services typés et restrictifs pour une connexion ou des fonctionnalités de communication spécifiques uniquement. Par conséquent, la «glu» vue dans les approches comportementales peut être un service spécifique et basique dans le connecteur CaP.

Notre connecteur peut jouer deux rôles majeurs dans l'architecture à base de composants ; il définit la structure de la configuration par les différentes liaisons lors du choix des éléments ou composants à relier ainsi que leurs occurrences. Aussi, il définit son comportement en fonction de la sémantique inhérente aux connecteurs choisis dans la bibliothèque de connecteurs (Figure 4.21) pour chacune de ces connexions.

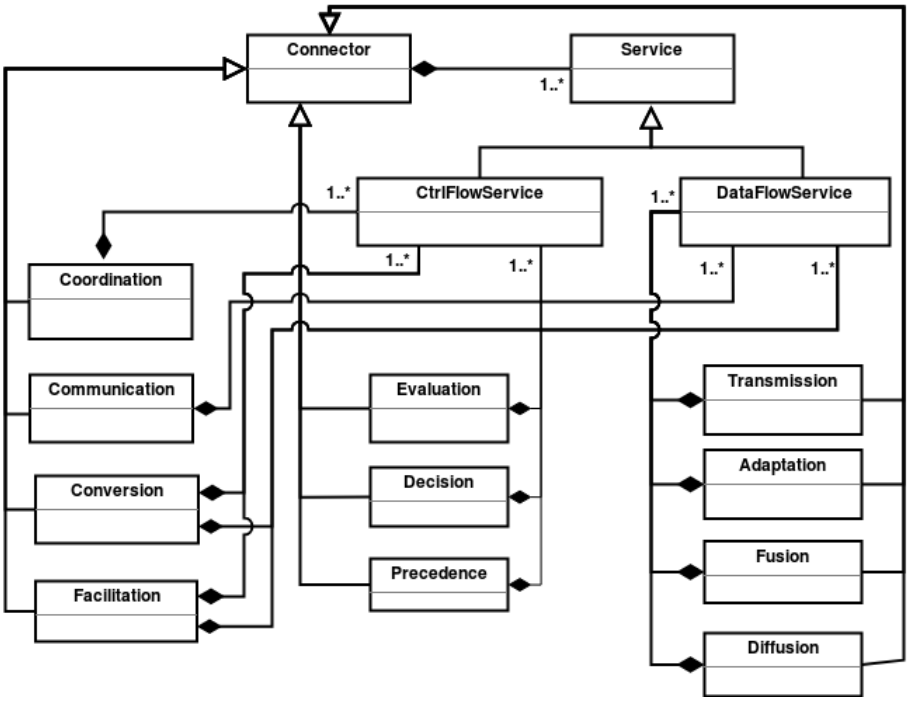


Figure 4.21- Structure de la librairie pour connecteurs

La Figure 4.21 représente un ajout au méta-modèle initial pour des spécialisations possibles du connecteur. Il donne la structure de la bibliothèque de connecteurs évoqués dans les différentes communautés. Nous considérons ensuite les connecteurs orientés architecture

logicielle ainsi que les connecteurs orientés processus logiciels et leurs relations.

Il est important de signaler que la sémantique du connecteur dépend de la sémantique des services qu'il assure ou qui le constituent. Ceci justifie le cycle de relations « composition/héritage » qui existent entre les classes « Connector » et « Service » d'une part et les classes « Coordination » ou « Evaluation » et la classe « Connector ». A ce titre, un connecteur de coordination est tout simplement un connecteur ayant un service de coordination à titre d'exemple. Aussi, le connecteur « Coordination » est composé essentiellement de la famille des connecteurs « CtrlFlowService ».

4.5.5.1 Connecteurs orientés architecture

Comme déjà évoqué dans la section (1.5.3), nous reprenons la famille de connecteurs connue dans la communauté de l'architecture logicielle. Les catégories de service qui définissent les principaux rôles joués par les connecteurs tels que définis dans la classification de Mehta et al. (2000) représentent les principaux types de connecteurs orientés architecture.

Comme le montre la Figure 4.21, nous listons les quatre types de connecteurs principaux appartenant à cette catégorie :

- Connecteur de « Communication » : les connecteurs qui fournissent ce type de service prennent en charge la transmission de données entre les composants. Les composants transmettent régulièrement des messages, échangent des données à traiter et communiquent des résultats. La communication est essentiellement définie comme un transfert de données, d'où la relation de composition avec la classe « DataFlowService ». Dans cette catégorie on retrouve les types de connecteurs suivants (selon la classification de Mehta et al. (2000)) : Appel de procédure, Evénement, Accès mémoire partagée et le connecteur flux.
- Connecteur de « Coordination » : Supporte le transfert du contrôle entre les composants. Les composants interagissent en se passant les threads d'exécution entre eux sous forme d'appel de procédure, d'appel de méthode. Ce type de connecteur peut fournir des interactions plus riches et plus complexes comme l'équilibrage de charge essentiellement basé sur les services de coordination. Le service de coordination peut être considéré comme orthogonal à tous les autres services c'est-à-dire qu'il est utilisable dans tous les autres types de services. La coordination est essentiellement définie comme un transfert de contrôle d'où la relation de composition avec la classe « CtrlFlowService ». Dans cette catégorie on retrouve les types de connecteurs suivants (selon la classification de Mehta et al. (2000)) : Appel de procédure, Evénement et Arbitre
- Connecteur de « Conversion » : Les connecteurs qui fournissent un service de conversion, transforment les interactions requises par un composant en celles fournies par un autre. Ils permettent à des composants hétérogènes et inadaptés d'établir des connexions et d'interagir. Cette catégorie peut effectuer à la fois des transferts de données et de contrôle, d'où la double relation de composition avec les classes « DataFlowService » et « CtrlFlowService ». Les conversions de format et les « *wrapper* » pour les composants déjà existants et qui ne sont pas conçus pour interagir, sont des exemples pour ce type de connecteurs.
- Connecteur de « Facilitation » : Les connecteurs qui fournissent les services de facilitation assurent la médiation et la rationalisation de

l'interaction des composants. Même si ceux-ci sont conçus pour interagir, il est toujours nécessaire de prévoir des mécanismes pour faciliter et optimiser leurs interactions. Des mécanismes tels que l'équilibrage de charge, la planification et le contrôle de la concurrence sont nécessaires pour répondre à certaines exigences système et pour réduire les interdépendances entre les composants en interaction. En outre, cette catégorie peut effectuer à la fois des transferts de données et de contrôle, mais dépend principalement du connecteur de flux de contrôle d'où la double relation de composition avec les classes « DataFlowService » et « CtrlFlowService ». Parmi les exemples, on a les types suivants (selon la classification de Mehta et al. (2000)): Lien, Arbitre et Distributeur.

4.5.5.2 Connecteurs orientés processus

Dans sa quête d'exécuter et réutiliser les processus logiciels, la communauté processus logiciel a définit deux concepts clés pour se rapprocher des architectures logicielles qui sont le composant processus logiciel et le connecteur processus logiciel (Aoussat et al., 2014; Dai et al., 2008; Dami et al., 1998; Ismail et al., 2017):

- Composant processus logiciel : il décrit un traitement réalisé sur des produits en entrée pour "la création" de nouveaux produits en sortie. Il possède une interface formée de ports processus logiciel (fournis/requis). Ces ports sont de deux types : (i) port de flux de données pour l'envoie et la réception de flux de données, (ii) port de flux de contrôle (envoie/réception) qui permettent d'identifier l'ordre et l'état d'exécution des composants de processus logiciel.
- Connecteur processus logiciel : il décrit le traitement à réaliser sur des produits en entrée afin de "les adapter ou les contrôler" pour les besoins du composant processus logiciel connecté. Il possède une interfaces formée de ports processus logiciel (fournis/requis) de la même manière que les composants processus logiciel. On a également les deux types de port (contrôle/donnée) pour le transport des flux correspondants.

Le concept de connecteur logiciel et les ports associés sont déjà pris en charge dans notre méta-modèle de la Figure 4.20. Ceci pour la partie structurelle du connecteur, pour la partie comportementale nous nous appuyons toujours sur la même taxonomie de la communauté des processus logiciels pour détailler les classes « DataFlowService » et « CtrlFlowService ». Ces classe représentent des composants de la classe "Service", qui est définie comme l'activité dans notre modèle de processus, et donc elles utilisent les classes pour les ports requis « R_DataPort » et « R_CtrlPort » en entrée pour consommer les flux entrants et fournir les flux sortants portés par les ports requis représentés par les classes « P_DataPort » et « P_CtrlPort ». Ces ports représentent des spécialisations en termes de données et de contrôle du produit du travail de notre processus de communication.

Puisque nous proposons un connecteur orienté processus, chacun des deux types de services doit être spécialisé à travers un ensemble de services inspirés du monde des processus logiciels (Dami et al., 1998; Aoussat, 2012). On retrouve implicitement la catégorie de service de flux de données dans la sémantique des connecteurs « Communication », « Conversion » et « Facilitation » ce qui justifie la relation de composition entre ces connecteurs et la famille de connecteur « DataFlowService ». Pour le service de transfert de données « DataFlowService », on peut avoir les connecteurs et services suivants (Figure 4.21):

- Connecteur « Transmission » : Il s'agit du service de base de cette catégorie

; il traite de la transmission ou du transport de données ou « WorkProduct » d'un composant à un autre sans aucune intervention sur le contenu. Sa principale caractéristique est qu'il utilise un seul port de flux de données d'entrée et un seul port de flux de données de sortie.

- Connecteur « Adaptation » : permet d'adapter ou de convertir puis de transmettre les données d'entrée aux données de sortie selon les exigences du composant connecté. Sa principale caractéristique est qu'il utilise un seul port de flux de données d'entrée et un seul port de flux de données de sortie.
- Connecteur « Fusion »: permet de fusionner un ensemble de flux de données entrants en un seul flux de données sortant. Sa principale caractéristique est d'utiliser des ports de flux de données à entrées multiples et un seul port de flux de données de sortie.
- Connecteur « Diffusion » : s'occupe de récupérer les données entrantes puis de les diffuser sur l'ensemble des composants connectés en aval. Sa principale caractéristique est qu'il utilise un seul port de flux de données d'entrée et plusieurs ports de flux de données de sortie.

Pour le service « CtrlFlowService », on retrouve aussi une sémantique similaire avec tous les connecteurs de famille des architectures logicielles ce qui justifie aussi la relation de composition qui les lie. Dans la catégorie « CtrlFlowService » nous avons les connecteurs et services associés suivants (Figure 4.21):

- Connecteur « Predecence » : ce connecteur correspond à des activités de transfert et de contrôle de flux d'exécution. Il formalise l'ordre d'exécution des composants, il assure l'exécution standard sans évaluation ni traitement particulier sur le flux d'exécution. Il peut également arrêter, reprendre, répéter et acheminer un flux de données à transférer ainsi que la gestion des exceptions. Ces services expliquent les ordres d'exécution traditionnels d'un modèle de communication qui dépend du début et de la fin des activités relatifs généralement à la gestion de projet. On peut avoir les séquences de synchronisation suivantes : (fin-début, début-début, début-fin et début-fin).
- Connecteur « Evaluation » : Ces services sont définis pour évaluer les performances de transfert de données en fonction du temps, du coût ou de la qualité de la transmission.
- Connecteur « Decision » : ce type de service est défini pour évaluer puis prendre des décisions concernant la progression de l'exécution du transfert. Les décisions sont rendues selon les critères d'évaluation : délai, coût et qualité. Le connecteur « Decision » peut être vu comme une résultante du connecteur « Evaluation ». Les approches Derdour et al. (2010) Alti et al. (2015) définissent des connecteurs similaires à ces deux derniers.

4.6 Extension SPEM pour le connecteur CaP

Le langage UML, en soit, n'est pas assez expressif en termes de connecteur explicité bien qu'il présente une forme visuelle intéressante. Aussi, il n'a aucun moyen pour décrire et modéliser un processus logiciel. Par conséquent, aucun outil ne peut accepter la mixture de notre méta-modèle dans son état actuel. Nous avons besoin d'un minimum de concept en architecture logicielle comme (composant, connecteur et configuration) et de concept en processus logiciel comme (Rôle, Activité et produit).

Pour cela, nous faisons recours au méta-modèle SPEM (Section 2.5) pour supporter la charge sémantique du connecteur CaP. Nous allons définir les stéréotypes nécessaires qui nous permettent de modéliser les éléments architecturaux (composant, connecteur et configuration) à base de connecteur processus.

Nous allons essayer de valider notre configuration finale avec deux exemples appartenant à deux mondes très distants sémantiquement ; celui des architectures logicielles et celui des processus logiciels. Par un mécanisme de conformité visuelle, nous allons essayer de trouver les types de chacun des éléments des deux exemples dans le méta-modèle final. Les mêmes éléments doivent avoir le même type. Les deux exemples seront exprimés en syntaxe concrète. L'objectif de notre démarche est d'essayer de montrer la puissance de notre proposition en termes de sa capacité à modéliser, avec les mêmes moyens, des mondes si différents.

4.6.1 Profil SPEM

Dans une architecture orientée composants, les composants sont interconnectés via des connecteurs pour obtenir une configuration architecturale. Par conséquent, nous devons décrire et présenter notre connecteur dans une architecture ou une configuration globale avec tous ses éléments descriptifs (Garlan et al., 2000) à l'aide d'un diagramme de classes UML ou d'un diagramme de composants. Cela nous permet d'exécuter et d'expérimenter notre modèle à l'aide d'outils et de normes UML disponibles pour la manipulation en simulation ou en génération de code.

Cependant, d'une part, UML n'est pas très expressif par défaut en termes de connecteurs. Ils sont généralement traités comme de simples liens entre les interfaces des composants en relation. Par conséquent, UML tel qu'il est ni même un profil UML dédié tel que dans (A Amirat et al., 2009; Nassar et al., 2016; Adel Smeda et al., 2009) ne peut pas prendre en charge notre connecteur avec sa riche charge sémantique.

D'autre part, la définition de notre connecteur en tant que processus logiciel et non plus en tant que produit logiciel nécessite un autre défi qu'UML ne peut pas relever. En effet, UML ne propose pas de description directe au travers de ses éléments pour les processus logiciels. A cet effet, l'OMG[1] utilisant les fondements orientés objet, propose une extension UML pour prendre en charge les concepts et procédures liés à la modélisation des processus logiciels à travers le profil SPEM 2.0 (*Méta-modèle Software Process Engineering*) (OMG-SPEM, 2008) qui reste conforme au MOF (*Méta Object Facilities*) (OMG-MOF, 2003). A ce titre, nous optons pour la description de notre architecture à base de connecteurs CaP en utilisant le profil SPEM pour décrire le modèle proposé sous forme d'architecture de processus logiciel.

En guise de bref rappel, SPEM (Section 2.5) est un méta-modèle organisé en sept packages avec certaines dépendances. Ces packages décrivent les différents points de vue à travers des stéréotypes introduits en se concentrant sur deux entités de base : le « processus » et les « méthodes » qui le gèrent. Conceptuellement, le méta-modèle SPEM est basé sur l'idée qu'un processus de développement logiciel est une collaboration entre des entités actives et abstraites, les rôles qui effectuent les opérations, les activités, sur des entités concrètes et réelles qui sont les produits. Les différents rôles interagissent ou collaborent en échangeant des produits et en initiant l'exécution de certaines activités (Benoît Combemale & Crégut, 2006).

Pour définir l'architecture du processus logiciel, SPEM considère le composant comme une activité en introduisant la notion de « Process Component » pour la réutilisation à base de

[1] OMG :(Object Management Group, https://www.omg.org/)

composants dans le profil « Method Plugin » en définissant des stéréotypes et des classes à cet effet. Cependant, quelques insuffisances sont constatées concernant les difficultés d'assemblage des composants du procédé dans les spécifications SPEM (Aoussat et al., 2014) ainsi que l'omission totale de la notion de "Connector Process" : Ces difficultés sont principalement dues à l'absence de certains notions :

- L'absence de connecteurs prédéfinis ou explicites : le connecteur "WorkProductPortConnector" dans le Méta-modèle SPEM est un connecteur implicite, simple lien entre le "WorkProductPort". Son rôle est limité aux séquences entre les composants du processus.
- L'absence du concept "ConnectorRole" ou dans notre cas "connector port"
- L'absence de contraintes d'assemblage : selon les multiplicités SPEM ; un connecteur peut connecter plusieurs ports sans aucune contrainte sur le nombre et le type.
- L'absence d'abstraction d'architecture : la notion de configuration inhérente aux architectures logicielles relatives aux processus est inexistante. "MethodeConfiguration" dans le package "Method Plugin" n'est pas un élément architectural du processus mais une classe ou un classifieur des éléments contenus dans le modèle. Aussi, aucune contrainte d'assemblage pouvant donner une modélisation incohérente n'est définie.
- Absence de styles architecturaux : la formalisation de structures récurrentes n'a pas été prise en compte.

En fonction de besoins particuliers, SPEM autorisation les extensions nécessaires uniquement à travers le package « Method Plugin ». Ceci, afin de conserver les restes des packages et de garder ainsi la structure du standard SPEM inchangé.

4.6.1.1 Stéréotypes architecturaux de base

Soutenir les nouvelles définitions de notre connecteur dans un système à base de composants tout en nous rappelant que nous considérons le connecteur et le composant comme ayant un statut égal et en nous inspirant des travaux d'Aoussat et al. (2014) pour étendre le profil « Method Plugin » avec deux stéréotypes abstraits liés aux aspects structurels et comportementaux (Figure 4.22) définis comme suit :

« ProcessArchitecturalÉlément » : il s'agit d'un stéréotype qui décrit les caractéristiques communes des stéréotypes de composants architecturaux qui constituent la vue structurelle de l'architecture de processus logiciel.

« MethodeContentArchitecturalÉlément » : il s'agit d'un élément abstrait qui décrit le comportement commun des styles des éléments architecturaux du processus.

La Figure 4.22 décrit comment ces deux stéréotypes personnalisés se rapportent aux stéréotypes de base du modèle SPEM. Le stéréotype "ProcessArchitecturalÉlément" est considéré comme un stéréotype "ProcessÉlément" défini dans le package "Process Structure". Nous définirons les éléments structurels sous ce stéréotype.

Le stéréotype "MethodeContentArchitectureÉlément" est un stéréotype "MethodeContentElemen" défini dans le package "Method Content". Les stéréotypes « ProcessÉlément » et « MethodeContentÉlément » sont définis dans SPEM comme des stéréotypes « DescribableÉlément » du package « Managed Content », qui est considéré comme un stéréotype « ExtensibleÉlément » du package « Core ». Nous définirons les aspects comportementaux sous ce stéréotype en particulier. Les stéréotypes ajoutés sont grisés sur la Figure 4.22. Ces différents stéréotypes et liens sont conformes au patron définit dans la Figure 2.19- Synthèse des stéréotypes de base et leurs relations.

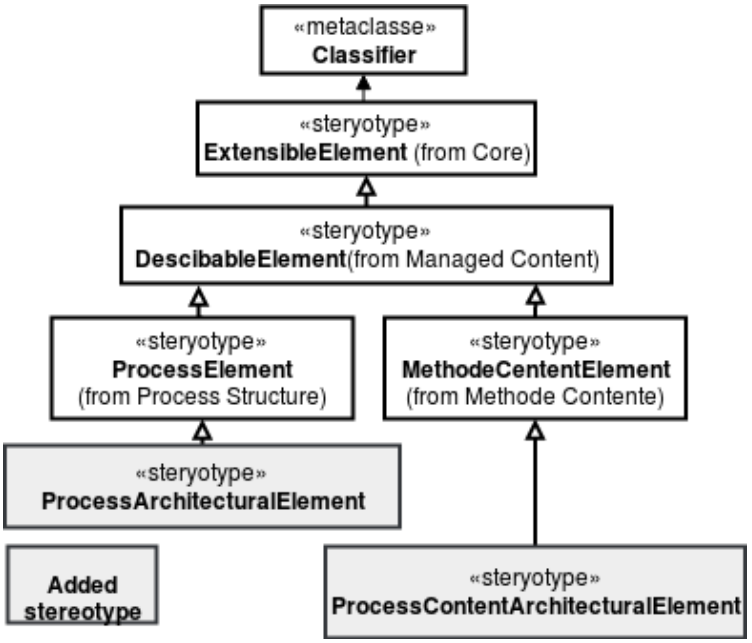


Figure 4.22- Extension du profil SPEM

4.6.1.2 Stéréotypes des éléments structurels

Les éléments structurels architecturaux sont souvent les composants, les connecteurs, la configuration, les interfaces avec les ports associés ainsi que les éléments structurels qui se repportent au processus.

L'ensemble des stéréotypes d'éléments architecturaux liés aux processus dérivés du stéréotype « ProcessArchitecturalÉlément » sont définis dans la Figure 4.23. Les stéréotypes « ProcessComponent », « ProcessConnector » et « ProcessConfiguration » décrivent les processus effectués pour la création (pour les composants) et l'exécution. Contrôle et adaptation (pour les connecteurs) pour produire de nouveaux flux de données circulant dans la configuration.

Aussi, le stéréotype "InterfaceÉlément" est considéré comme un élément architectural du processus. Le stéréotype "ProcessPort" représente les ports des composants du processus tandis que "ProcessConPort" et "Service" représentent l'interface des connecteurs du processus ainsi que l'"Actor" car il ne peut agir sur le connecteur que via son interface. On retrouve également tous les éléments explicites qui composent le connecteur, et qui représentent notre principale contribution, à savoir « ProcessConPort », « Service » et «Actor». « ProcessComponent » est un sous-type de la méta-classe UML « Component », tandis que « ProcessusConnector » est décrit par la méta-classe « Class » car UML ne traite pas le connecteur comme une entité explicite. Les stéréotypes ajoutés sont grisés sur la Figure 4.23.

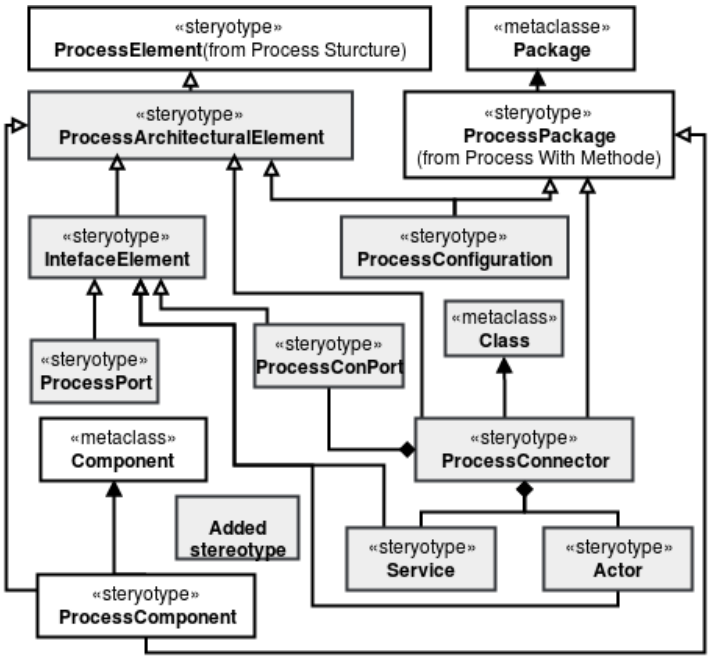


Figure 4.23- Stéréotype des éléments architecturaux pour le processus

4.6.1.3 Stéréotype des concepts comportementaux

Puisque les composants de processus et les connecteurs sont identifiés comme étant des activités, nous introduisons le stéréotype « ActivityDefinition » pour décrire leurs types. Cette définition représente la spécialisation principale du stéréotype "MethodeContentArchitecturalÉlément" ajouté dans la Figure 4.22. Elle est également une sorte de stéréotype "MethodeContentPackage" du package SPEM "Process With Method".

Le stéréotype "ArctivityDefinition" trouve son type dans le stéréotype "WorkDefinition" défini dans le package "Core". "DefaultActivityDefinitionParameter" est un stéréotype qui décrit la direction et l'assemblage par défaut de "ActivityDefinition". Ainsi, il hérite du stéréotype « WorkDefinitionParameter » défini dans le package « Core » (Figure 4.24).

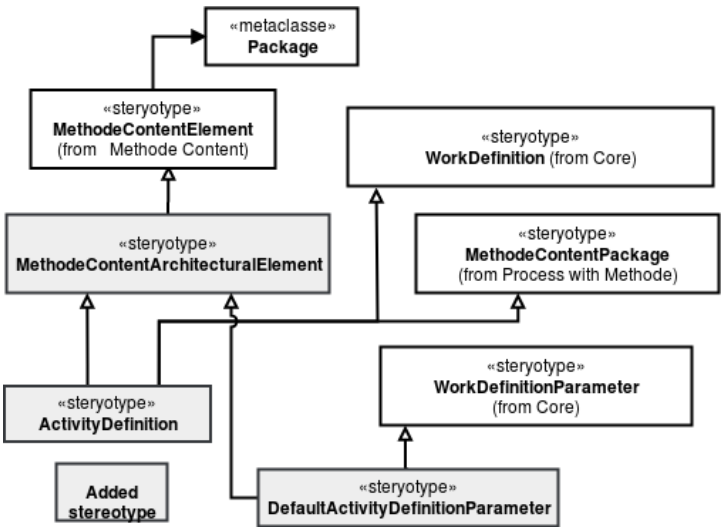


Figure 4.24 Stéréotype « ActivityDEfinition »

4.6.1.4 Stéréotypes pour la configuration architecturale

Contrairement au composant, un composant ne peut jamais exister seul. Il relie au moins deux composants. Par conséquent, les connecteurs doivent obligatoirement être décrits au sein d'une architecture logicielle. Dans la configuration architecturale de la Figure 4.25, on retrouve le détail des associations de base entre les stéréotypes des éléments architecturaux entrant dans l'extension du modèle SPEM et le stéréotype « ActivityDefinition ». Ce stéréotype joue à la fois le rôle de type pour les composants de processus et les connecteurs de processus.

Les composants et les connecteurs sont considérés comme des activités; en d'autres termes; fragments de processus logiciels composés des éléments de base d'un processus logiciel à savoir "TaskUse", "RoleUse", "ToolUSe" et "UseQualification" qui explique leurs définitions successives pour composer "ActivityDefinition" considéré comme "BreakDownElment" ou décomposition. "DefaultActivityParameter" est utilisé pour décrire "WorkProductDefinition" qui à son tour définit le stéréotype "Port". Le stéréotype "Port" joue le rôle de port de connecteur et de port de composant à la fois, ils sont distingués par l'attribut "KindPort".

Le stéréotype "ProcessConnector" est composé, comme dans le Méta-modèle initial, de ses trois éléments et conserve également la structure générale du processus logiciel. Ces éléments sont décrits par les éléments de "ActivityDefinition". Ainsi, "Acteur" est défini par "RoleDefinition" comme un "ProcessPerformer". « Service » est défini par "TaskDefinition" en considérant les services offerts/requis par le connecteur comme un ensemble de tâches dans toute l'activité du processus. Enfin, « Port » (fourni/requis) définit le « work product » à travers le stéréotype "WorkProductDefiniton".

"Port" et "Service" sont les types de ports et de services de type Flux de données et flux de contrôle. "ServDataFlow" et "ServiCtrlFlow" produisent et consomment respectivement "DatFlowPort" et "CtrFlowPort". Les catégories de service du connecteur et leurs types sont définis dans les énumérations au bas de la Figure 4.25.

Le stéréotype "ProcessConfiguration" définit une configuration d'une architecture logicielle basée sur les composants et connecteurs Processus. C'est pourquoi elle est définie aussi à travers "ActivityDefinition". Naturellement, "ProcessConfiguration" est composée de "ProcessComponen" et "ProcessConnector" reliés par "Port" qui joue le rôle d'attâchement.

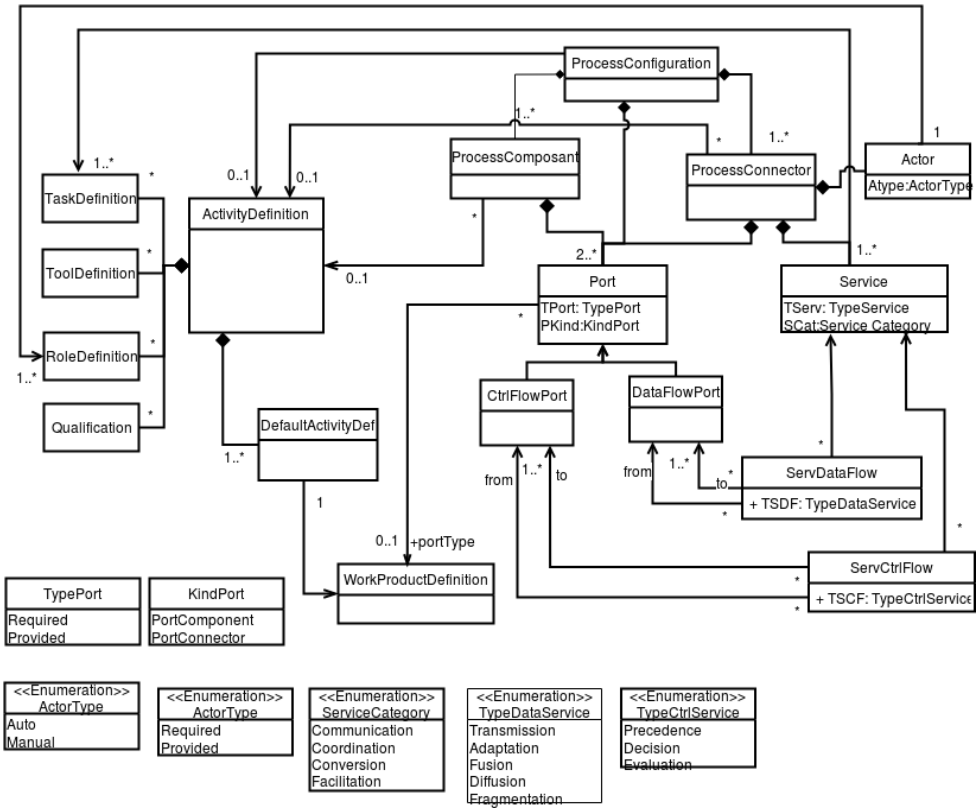


Figure 4.25- Configuration architecturale

4.6.2 Étude de cas

Afin de montrer la viabilité du connecteur proposé dans différents domaines de communication, nous l'utilisons pour montrer son efficacité dans des domaines très distincts: le domaine d'architecture logicielle avec un exemple "Pipe&Filter" et le domaine de processus logiciel avec un exemple "Cycle-Vie V". Pour plus de commodité, nous définissons ces exemples avec la syntaxe concrète de la Figure 4.26. Pour des raisons de volume de l'exemple, la syntaxe est utilisée de manière compacte pour le deuxième exemple.

4.6.2.1 Syntaxe concrète du connecteur

Pour montrer l'utilisation du connecteur proposé de manière plus pratique et compréhensible, nous définissons sa syntaxe concrète (Figure 4.26) qui peut être utilisée simplement dans les exemples qui suivent. Cette syntaxe utilise les conventions définies dans la section (4.3) qui assimile les ports requis et fournis avec « WorkProduct », « Actor » à « Role » et « Service » à « WorkUnit » ou « Activity ».

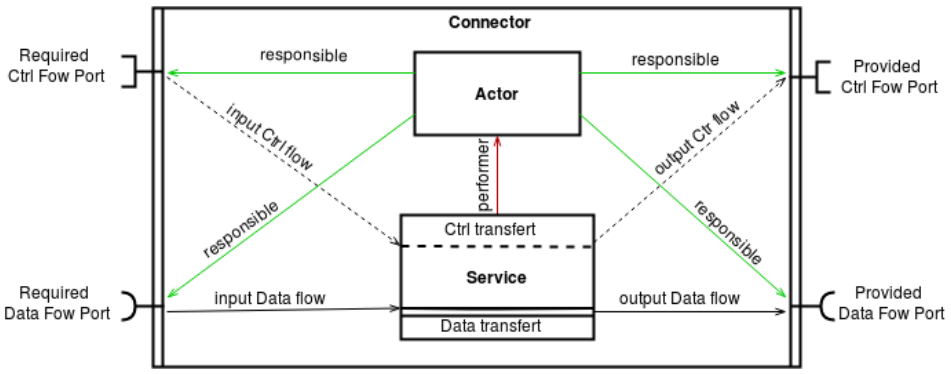


Figure 4.26- Syntaxe concrète du connecteur

Cette représentation reste cohérente avec les concepts définis dans les Figure 4.17, Figure 4.18, Figure 4.19 et Figure 4.20. Les relations "input", "output", "responsable" et "performer" sont définies dans la Figure 4.17 qui décrit le méta-modèle de processus. Ils sont associés à chaque port puisque le connecteur peut avoir plusieurs ports d'entrée et de sortie. Nous avons également adopté la notation de « rotule » d'UML (diagramme de composants) pour représenter les ports avec une petite variation entre les flux de contrôle et les flux de données.

L'« Acteur » surveille les ports requis, analyse le flux de contrôle d'entrée et détermine le comportement associé (synchronisation, ordonnancement, boucle, sélection et interruption). Puis il délègue et contrôle les traitements nécessaires et/ou le transfert assuré par l'entité «Service » sur les flux de données et de contrôle entrants. Le transfert réussi est validé par l'« Acteur » une fois que les flux de données et de contrôle sont dans les ports respectifs fournis ou sortants.

4.6.2.2 Exemple de Pipe&Filter

"Pipe&Filter" (Section 1.5.2.1) est un modèle architectural récurrent dans le domaine des architectures logicielles (Taylor et al., 2009). Il a des entités indépendantes appelées filtres (composants) qui effectuent des transformations sur les données et traitent l'entrée qu'ils reçoivent. Il a aussi une tuyauterie qui sert de connecteurs (conduite) pour le flux de données et de contrôle en cours de transformation, chacun connecté au suivant composant dans le pipeline. Le style architectural « Pipe&Filter » est donc utilisé pour diviser une tâche de traitement plus importante en une séquence de tâches plus petites. Dans notre cas, le connecteur est également un composant de communication.

Nous utilisons un problème classique du type Fragmenter/Fusionner (Figure 4.27) avec une séquence de composantes (Filtres) P1, P2, P3, P4 et P5. L'exemple consiste à découper un message ou une requête en plusieurs paquets ou fragments par P1 que chacun peut être traité avec un composant intermédiaire spécifique P2, P3 et P4. Enfin, ces fragments seront regroupés en sortie par un autre composant P5.

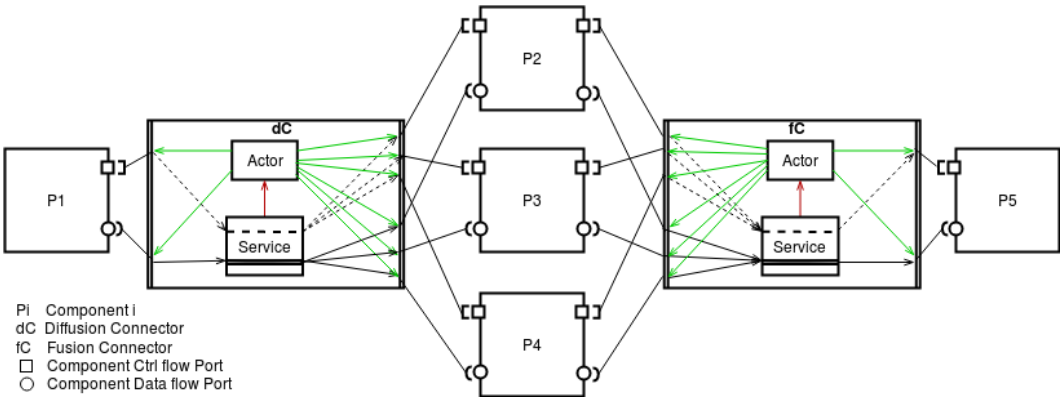


Figure 4.27- Configuration Pipe&Filter

Pour cela, nous n'utilisons que deux connecteurs (Pipe). Le premier connecteur (dC) est de type "Diffusion" ayant six ports fournis vers les composant P2, P3 et P4 (3 pour le transfert des flux de contrôle et 3 pour le transfert des flux de données) ainsi que deux ports requis pour recevoir le flux de contrôle et le flux de données venant du composant P1. Le second connecteur (fC) est de type "Fusion" avec six ports requis (données/contrôle) avec les composants P2, P3 et P4 ainsi que deux ports fournis. Ces connecteurs sont déjà définis dans notre bibliothèque (Figure 4.21) en tant que connecteurs sur étagère.

Le composant P1 fragmente le message d'origine et désigne le composant destinataire de chaque fragment avec le contrôle correspondant. À partir du flux de contrôle entrant du composant P1 (avec deux ports fournis pour le contrôle et le flux de données), le connecteur (dC) dirige le contrôle et les fragments de données qu'il reçoit dans ses ports requis correspondants vers le composant souhaité (P2, P3 ou P4) via ses six ports fournis. Les composants P2, P3 et P4 traitent chacun le fragment et le contrôle qu'ils ont reçu et les mettent dans les ports correspondants du connecteur de fusion (fC). Le connecteur (fC) récupère les flux de contrôle et les paquets de données, et les transfère dans l'ordre d'origine à P5 pour reconstruction du message.

Les services fournis par les connecteurs à ce niveau sont :

• Pour le connecteur dC : on a un service « Diffusion » pour le transfert de données et les services « Evaluation » / « Décision » pour le contrôle-transfert. De ce fait ; il assure implicitement les services de « Communication » et de « Coordination » (Figure 4.29).

• Pour le connecteur fC : on a un service « Fusion » pour le transfert de données et le service « Precedence » de type « Decision » en mode de synchronisation« Fin/Fin » pour le transfert de contrôle et de paquets.

4.6.2.3 Exemple de cycle de vie en « V »

Le cycle de vie du logiciel est considéré comme un modèle descriptif qui décrit les étapes et séquences connues et reconnues pour la réalisation d'un produit logiciel. Parmi les modèles de cycle de vie, on a le « cycle de vie en V » choisi est décrit comme un modèle en cascade dans lequel les étapes de développement et la mise en œuvre des tests sont effectuées de manière synchrone (Clement et al., 2011). Les étapes formeront les composants du processus logiciel (composant SP) tandis que les séquences seront modélisées par des connecteurs de processus logiciel (connecteur SP). Le modèle en V est assez bien connu dans la communauté des processus logiciels (Figure 4.28).

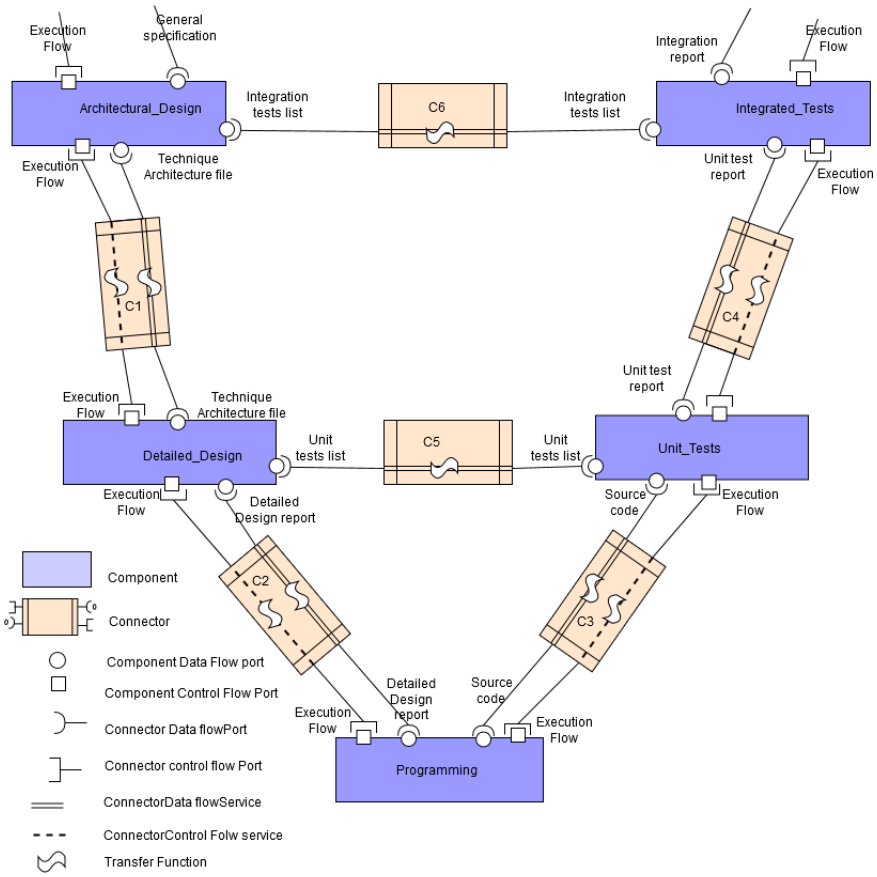


Figure 4.28- Configuration Cycle de vie en V

En général, le composant processus logiciel décrit les processus exécutés sur les produits d'entrée pour « créer » de nouveaux produits de sortie. Il doit avoir au moins deux ports dans son interface : un port pour le transfert de données et un port pour le transfert de contrôle. Les deux ports sont de deux types : « requis » pour capturer le flux entrant et « fourni » pour acheminer le flux sortant.

Le connecteur processus, quant à lui, décrit les traitements à effectuer sur les produits d'entrée afin de les "adapter ou contrôler" aux besoins du composant processus logiciel connecté en aval. Il ne crée pas de produit dans ce cas mais fournit deux services essentiels qui forment le cœur de notre proposition : le transfert de données et le transfert de contrôle. Les activités de transmission telles que la diffusion, la fragmentation ou la conversion de données permettent la gestion des transferts de données. Les activités de transfert de flux de contrôle aident à assurer l'ordre d'exécution des composants processus logiciels. Certaines de ces activités nous permettent d'analyser la qualité d'exécution des transferts.

La relation d'héritage dans la composition du service (Figure 4.20) nous permettra également d'avoir un ou plusieurs services (soit pour les données, soit pour le contrôle ou soit pour les deux en même temps) selon les besoins. Les services fournis par les connecteurs à ce niveau sont : le service « Transmission » pour le transfert de données et le service « Precedence » de type « End-Begin » pour le transfert de contrôle (Figure 4.29).

Dans cet exemple, nous avons plusieurs connecteurs (C1, C2, C3 et C4) dans une syntaxe concrète compacte du connecteur proposé pour transférer les différents produits ("Technique Architecture file", "Detailed Design Report", "Source Code", "Unit Test Report", …etc) entre les composants ("Architectural_Design", "Detailed_Design", "Programming", "Unit_Tests" et "Integrated_Tests") via les ports de flux de données ainsi que les flux "Éxécution Flow" via les ports de flux de contrôle.

La communication est bidirectionnelle afin de prendre en charge le retour d'erreur entre les composants. Le transfert d'erreur, dans le "cycle de vie V", s'effectue en fonction de l'ascension de la cascade. Les connecteurs horizontaux (C5, C6) sont censés ne transférer que les données de test (unitaire, intégration) dans un seul sens à partir duquel il a été initié sans aucune information de contrôle.

A titre d'exemple, le connecteur « C1 » accepte le flux de données "Technique Architecture file" dans son port requis de données en tant que « produit de travail » et le transmet à son port de contrôle fournit selon le "Éxécution Flow". Le transfert se fait sans aucune modification des données du fait de l'utilisation du connecteur de flux de contrôle « Precedence » du type « Fin-Debut ».

4.6.3 Conformité des modèles étudiés

Après avoir proposé le méta-modèle SPEM ainsi que les exemples discutés dans notre étude de cas, nous optons pour un mécanisme de conformité pour montrer que chaque élément des deux exemples trouve bien son type. Pour des raisons d'espace et de clarté, nous nous limitons à la représentation de la branche entre les composants P1 et P3 avec le connecteur dC qui constituent l'Architecture Pipe/Filtre d'un côté. De l'autre côté; nous considérons le fragment "Architectural_Desig" et "Detailed_Design" avec le connecteur "C1" pour constituer la configuration du cycle en V dans la même figure (Figure 4.30). Tous les mêmes éléments doivent avoir les mêmes stéréotypes.

Tous les concepts similaires entre les configurations « Pipe/Filtre » et « V-Cycle » du niveau « M1 » ont les mêmes types au niveau supérieur « M2 ». Les composants P1, P3, "Architectural_Design" et "Detailed_Design" sont typés par les stéréotypes "ProcessComponent" définis comme le type "ActivityDefinition" indiquant que les composants sont des activités. L'attribut "ActivityKind" distingue le composant du connecteur. Les ports des composants sont typiques "WrokProductDefinition" et peuvent être décrits avec les stéréotypes "CtrlFlowPort" ou "DataFlowPort". La distinction entre les ports fournis/requis est faite par l'attribut "TypePort".

Les connecteurs "dC" et "C1" sont à la fois transfert de données et transfert de contrôle. Ils sont typés par "ProcessConnector" qui est défini par le stéréotype "ActivityDefinition" indiquant que le connecteur est également une activité. Pour des raisons de commodité de représentation, « C1 » est sous une forme compacte mais il a exactement la même architecture interne que « dC ». Le service de transfert de données (ligne double) est décrit par le stéréotype "ServDatFlow" tandis que le service de transfert de contrôle (ligne pointillée) est défini par le stéréotype "ServCtrlFlow". Ces deux services sont respectivement "DataFlowPort ports" et "CtrlFlowPort" du connecteur d'entrée/sortie. Ces deux ports se distinguent de ceux du composant par l'attribut "KindPort". Le service est décrit par le stéréotype « Service » défini comme une "TaskDefinition" tandis que l'acteur est décrit par le stéréotype « Acteur » défini comme le stéréotype "RoleDefinition".

Les connecteurs « dC » et « C1 » sont des abréviations visuelles des connecteurs respectivement « Diffusion/Dicision » et « Transmission/Precedence » qui justifient les héritages multiples (Figure 4.29). Nous sommes dans une catégorie de connecteur « Communication » et « Coordination » avec service de diffusion et service de décision pour « dC » tandis qu'un service de transmission et service de précédence (Begin-End) sont pour « C1 » selon les types d'énumération respectifs : « ServiceCategory » , "TypeDatService" et "TypeCtrlService" (Figure 4.25).

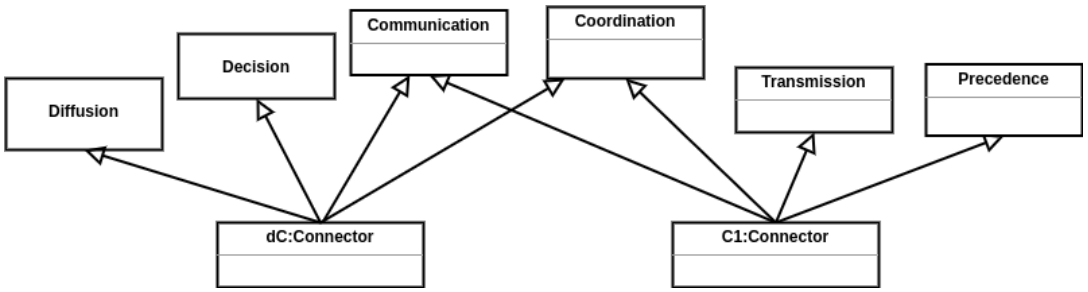


Figure 4.29 Typage des connecteurs utilisés

En guise de synthèse, on peut dire que la Figure 4.30 montre que l'architecture Pipe&Filter ainsi que l'architecture V-cycle du niveau M1 peuvent être instanciées à partir de notre profil SPEM pour une configuration à base de connecteur CaP défini au niveau M2. Les niveaux M1 et M2 se rapportent à la démarche MDA. Les deux modèles utilisent la même bibliothèque de connecteurs d'architecture. Leurs éléments correspondants et similaires sont décrits par les mêmes stéréotypes comme on peut dire qu'ils ont le même type dans leur niveau supérieur car ils sont définis par le même méta-modèle. Les deux modèles sont donc conformes (par une correspondance visuelle) au même méta-modèle. Par conséquent, notre approche est suffisamment puissante et assez générique pour réduire l'écart entre des domaines qui, à l'origine, étaient très distants sémantiquement.

Notre objectif dans cette thèse est de proposer une nouvelle réflexion pour l'abstraction de la communication dans les architectures logicielles. Il s'agit de surmonter l'hétérogénéité et de promouvoir l'interopérabilité. Cela justifie le fait que nous traitons à ce niveau d'un aspect conceptuel et abstrait. Un exemple expérimental pour la mise en œuvre est présenté au chapitre 5 alors que les fondements formels sont reportés pour des travaux ultérieurs.

4.7 Conclusions

Dans ce chapitre nous avons étudié les approches qui considèrent le connecteur comme une entité de première instance. Nous avons relevé un certain nombre d'insuffisances qui nous ont conduits à préconiser une nouvelle approche. La solution consiste à adopter et à étendre le modèle de communication dans des architectures à base de composants en utilisant des connecteurs comme entité de première classe pour aboutir à un composant de communication à part entière. Ce connecteur (baptisé CaP pour : *Connector as Process*) fournit des services de communication typés sur les ports de son interface. Ainsi, le connecteur est destiné à être utilisé pour/par la réutilisation ("*for/by reuse*") tel qu'un composant sur étagère.

Nous avons assimilé le connecteur à un processus logiciel ouvert et agile en intégrant les différents éléments associés répondant à des interactions complexes. Nous avons introduit les éléments syntaxiques et la définition sémantique de ce modèle de communication en précisant sa structure et ses éléments avec leurs relations à travers un Méta-modèle. Ce dernier à été augmenté d'un profil SPEM définissant les stéréotypes nécessaires pour la prise en charge des éléments structurels et comportementaux d'une configuration à base de connecteur CaP.

Pour ce faire, au préalable, nous avons considéré le connecteur dans ses définitions de base. En effet, le connecteur n'est autre qu'un transfert de données, un transfert de contrôle ou les deux à la fois le long d'un canal. Cela nous permet d'identifier un ensemble de connecteurs à partir de ces blocs de base identifiés à la fois dans les communautés d'architecture logicielle et de processus logiciel.

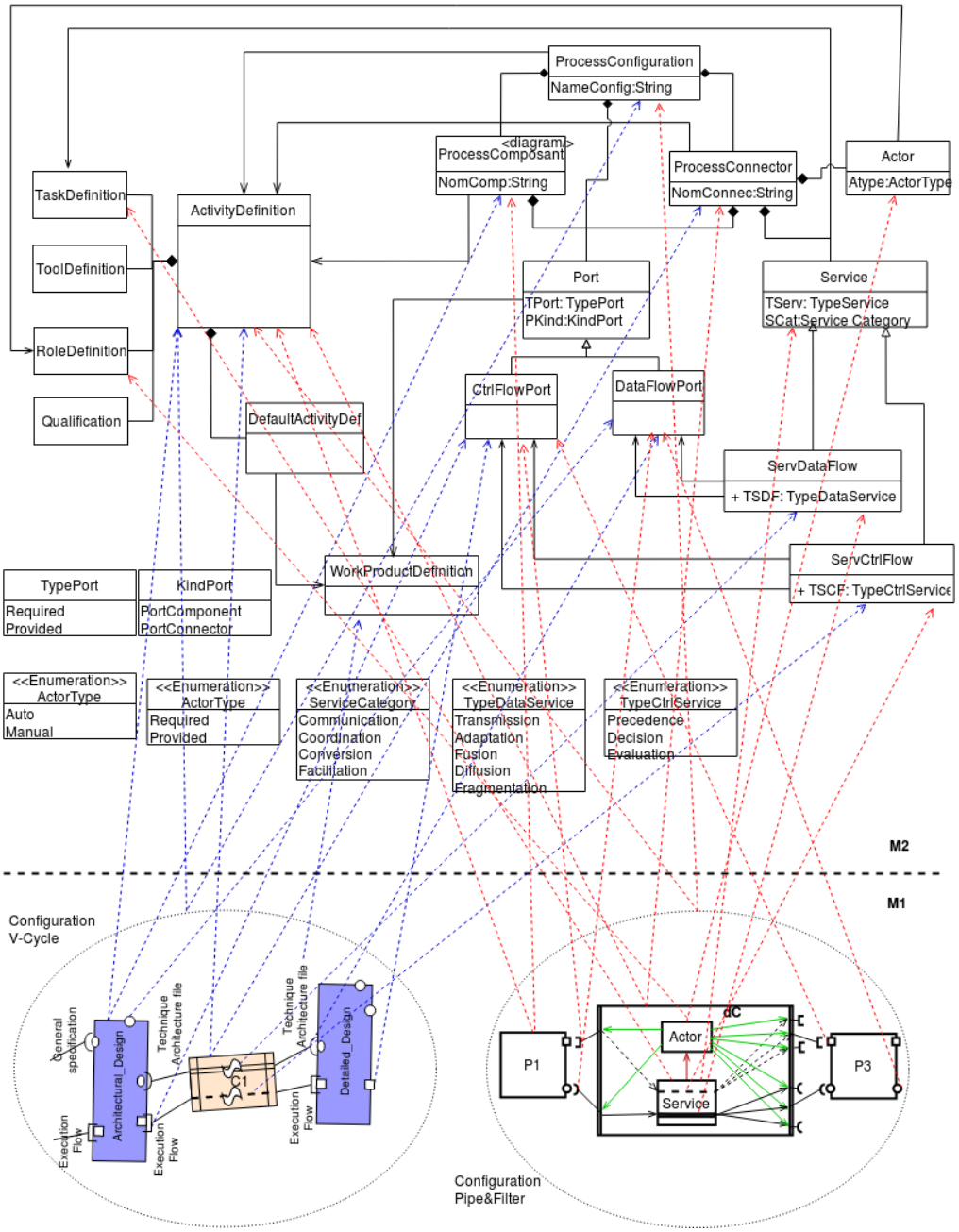


Figure 4.30 Conformité duale : Model « Cycle-V » et Pipe&Filter »

En d'autres termes, notre approche visait à étendre et reconstruire le concept de connecteur tel que défini dans le domaine des architectures logicielles en intégrant principalement les éléments qui définissent le processus logiciel. Cependant, une extension supplémentaire pour d'autres communautés et paradigmes peut également être imaginée selon d'autres exigences.

En outre, la décomposition du flux en données et contrôle permet d'identifier et de gérer la complexité ainsi que la réutilisation et le gain en généricité pour la communication multi-paradigmes.

CHAPITRE

5 Langage de Description d'Architecture: capADL

« L'humanité, par la grâce de Dieu, a soif de paix spirituelle, de réalisations esthétiques, de sécurité familiale, de justice et de liberté. Aucune de ces aspirations n'est directement satisfaite par la productivité industrielle. Mais la productivité industrielle permet de partager l'abondance plutôt que de se disputer la pénurie; elle offre du temps pour les questions spirituelles, esthétiques et familiales. Elle permet à la société de déléguer du temps et des compétences particulières aux institutions religieuses, judiciaires et ainsi à la préservation de la liberté. »

Harlan Mills dans 'DPMA et productivité humaine'

5.1 Introduction

Dans cette thèse, il était question de proposer un modèle conceptuel pour l'abstraction de la communication dans les architectures logicielles. Nous aurions pu donc nous limiter à la proposition annoncée dans le chapitre 4. L'exigence était de rester dans un niveau conceptuel (M2) ou d'abstraction pour omettre la disparité des détails techniques et d'avoir une vue générique à travers les techniques de méta-modélisation.

Le Tableau 3.2 et la Figure 3.9 précisent que les ADLs appartiennent également au niveau M2 de modélisation. Un ADL n'est pas un langage de programmation mais un langage de description et d'analyse d'architecture du système en cours de développement. Son rôle se limite au niveau architectural et donc à un stade précoce dans le cycle de développement. A ce titre, il devient impératif et enrichissant d'associer à notre proposition un support technique dédié pour consolider sa faisabilité.

Dans ce chapitre, nous proposons notre propre langage de description d'architecture que nous baptisons (capADL) pour prendre en charge les architectures logicielles basées sur les connecteurs orientés processus. Pour ce faire, nous allons utiliser comme outil AToM3 pour créer et manipuler l'ADL souhaité. De la même manière, la Figure 3.25 précise également que les outils de modélisations comparés appartiennent tous à des niveaux de modélisations élevés. C'est le cas de l'outil retenu (AToM3) qui appartient aux niveaux (M1, M2 et M3) et peut les prendre en charge séparément. Par conséquent, utiliser cet outil va nous permettre d'opérer toujours à un niveau d'abstraction élevé que ce soit pour le connecteur CaP comme pour l'ADL capADL associé.

Dans ce qui suit, nous allons présenter l'outil de modélisation et les justificatifs associés ainsi que les différentes démarches pour la création de capADL. Une ultime étape est consacrée à une transformation de modèle qui permet, à partir d'une architecture logicielle donnée, de générer le squelette de l'application en langage Java. Le système sera validé en utilisant les mêmes exemples manipulés dans le chapitre 4, à savoir « Pipe&Filter » et le modèle de cycle de vie en « V ».

5.2 Outil de modélisation

Parmi les trois environnements étudiés (Figure 3.25), nous avons retenu l'outil de modélisation AToM3 pour des raisons évidentes que nous présentons dans la section Motivation du choix (5.2.2). Dans ce qui suit, nous donnerons un aperçu sommaire sur l'outil. Nous utiliserons des exemples assez expressifs (transformation modèle vers modèle) (Menasria, 2011) pour la compréhension, étant donné que les transformations de modèles vers code n'apportent aucun changement visible dans les parties gauche et droite de la règle de transformation.

5.2.1 Généralité

AToM^3 (*A Tool for Multi-paradigm and Méta-Modelling*) (de Lara & Vangheluwe, 2002) est un outil de modélisation développé en collaboration entre le laboratoire MSDL (*Modelling, Simulation and Design Lab*) à l'université de McGill Montréal, Canada et de l'Université Autónoma de Madrid (UAM), Espagne. Ecrit dans le langage Python, comme son acronyme l'indique, il s'appuie sur deux concepts de base : le Multi- Paradigme (multi-formalisme) et la méta-modélisation.

Le Multi-Paradigme de l'outil est motivé par le fait que la modélisation des systèmes complexes est une tâche ardue. En effet, de tels systèmes ont souvent des composants et des aspects dont la structure ainsi que le comportement ne peuvent être décrits dans un seul formalisme (de Lara & Vangheluwe, 2002). Aussi, le multi-formalisme est une propriété inhérente aux transformations de modèles par le constat naturel qu'on manipule très souvent des formalismes distincts.

Aussi, les DSVL (*Domain-specific visual languages*) sont des notations graphiques spécialement dédiées pour les besoins spécifiques et les connaissances d'un certain groupe d'utilisateurs. Les DSVL ont l'avantage de se situer à un très haut niveau d'abstraction et d'être très efficaces et intuitifs pour la tâche à traiter. Dans le domaine où les DSVL sont intensivement utilisés, où la notation doit évoluer, on a besoin d'une manière pour réduire l'effort de construction et de maintenance des outils du DSVL (de Lara & Vangheluwe, 2004).

La méta-modélisation se présente comme est une façon idéale pour réduire ce problème où l'on peut utiliser des notations graphiques de haut niveau (comme les diagrammes de classe UML avec les contraintes OCL) pour définir la syntaxe et la sémantique du langage visuel en question. À partir de ce méta-modèle, un outil ou un environnement visuel de modélisation est généré pour le formalisme désiré. L'outil ainsi produit, possède des fonctionnalités réduites à l'édition, la sauvegarde, le chargement de modèles et leurs transformations et/ou simulation tout en assurant leur conformité par rapport aux méta-modèles.

Comme les modèles, les méta-modèles ainsi que les méta-méta-modèles peuvent être représentés comme des graphes typés et attribués, ces opérations peuvent alors être exprimées à l'aide de grammaire de graphes (3.3.6.2) (H Ehrig et al., 1999). Dans ce cas, les traitements deviennent des modèles de haut-niveau exprimés dans des notations formelles,

graphiques et intuitives. De cette façon, on élimine le besoin d'ajouter des fonctionnalités à l'outil généré en utilisant les langages de bas niveaux et on obtient le potentiel pour améliorer les facteurs clés du développement de logiciels, tels que la productivité, la qualité et l'aisance de la maintenance. Ceci représente un atout essentiel pour la problématique discutée dans le cadre de cette qui consiste à rester dans un niveau conceptuel.

Les vocations essentielles d'AToM3 sont la simulation, la modélisation, l'optimisation et la génération de code, qui peuvent, toutes, être spécifiées sous forme de transformation de modèles dès lors qu'on peut les spécifier de manière déclarative et opérationnelle.

Les deux principales tâches d'AToM^3 sont la méta-modélisation et la transformation de modèles. La méta-modélisation concerne la description ou la modélisation de différentes sortes de formalismes utilisés pour représenter les systèmes y compris leurs simulations, tandis que la transformation de modèles se concentre sur le processus de conversion (automatique), translation ou modification d'un modèle dans un formalisme bien donné dans une autre qui peut ou non être dans le même formalisme.

5.2.2 Motivation du choix

Parmi une panoplie d'outils de transformation de modèle (Kahani et al., 2019), nous avons opté pour AToM3 pour quatre raisons essentielles assurées par l'outil d'une manière réunie. Ces caractéristiques représentent nos attentes pressantes de l'outil admis:

- Méta-modélisation : la méta-modélisation est un concept central dans notre approche ainsi que la manipulation de la syntaxe abstraite et la syntaxe concrète en même temps.

- Génération Automatique de l'ADL : pour créer notre ADL, notre rôle se limite à bien définir le méta-modèle associé, les contraintes et les représentations visuelles. AToM3 assure à lui seul la génération du nouvel environnement et la vérification de conformité des architectures définies par l'architecte par rapport au méta-modèle.

- Transformation de modèle : comme la majorité des ADLs assurent la génération de code, notre capADL doit de telles facultés. AToM3 est par définition un outil de transformation de modèle, il assure entre autres les transformations « modèle vers code ». Cette faculté peut nous permettre également de générer des langages formels (CSP, π-Calcul) pour des fins d'analyse des protocoles d'interactions. Aussi, elle peut être le support pour l'obtention des protocoles dédiés à d'autres paradigmes comme le Protocole Zigbee (Farahani, 2011) pour les objets connectés et le langage Fipa-Acl (Pitt & Mamdani, 1999) pour les systèmes agents.

- Transformation de graphes : comme la majorité des modèles manipulés sont des graphes attribués, le recours aux transformations de graphe devient évident pour profiter de la simplicité des transformations en s'appuyant sur des assises formelles telles que de la théorie des graphes et la théorie des catégories. Par définition une architecture logicielle est considérée unanimement comme un graphe. Sa manipulation par les transformations de graphe devient un recours naturel.

On aurait pu utiliser d'autres outils comme ATL ou AGG qui sont reconnus comme des standards dans leurs domaines. ATL est un environnement de transformation de modèle pure alors qu'AGG est un outil de transformation de graphe pure comme précisé dans le

comparatif du Tableau 3.4. Ceci dit, hormis AToM3, aucun de ces deux autres outils ne peut répondre aux quatre critères de sélection susmentionnés. Aucun ne permet la génération automatique de l'ADL. Aussi, ATL s'appuie sur la description textuelle des transformations alors qu'AGG manipule les types graphes et non pas le diagramme de classe.

Ceci d'une part, d'autre part la Figure 3.25 montre un argument de taille en faveur d'AToM3. Il est le seul outil à même de pouvoir servir et parcourir les quatre niveaux de modélisation. Son méta-formalisme du niveau M3 basé sur les classes (classe, association) est conforme au MOF. Ce mécanisme lui permet de définir la structure et la sémantique de tout formalisme ou langage du niveau M2 (dans notre cas un ADL). Le langage ainsi défini nous permet de décrire tout modèle (M1) (dans notre cas architecture). Au niveau M0, l'outil peut assurer une exécution ou simulation du système modélisé (dans notre cas une génération de code). Aussi, l'outil assure les relations de conformité entre tous les niveaux en guise de moyens d'analyse.

Dans son étude sélective entre des outils similaires, Roy Grønmo (2010) compare trois outils AGG, ATL et CGT (*Concrète syntax-based graph transformation*) et opte pour le dernier en raison de sa capacité à manipuler les modèles dans leurs syntaxes concrètes. Il a montré que les règles de CGT sont plus concises et nécessitent beaucoup moins d'efforts qu'avec AGG et ATL. Avec AGG et ATL, le modélisateur de transformation doit avoir accès et connaître le méta-modèle et la représentation dans la syntaxe abstraite. Dans les règles CGT, en revanche, le modélisateur de transformation peut se concentrer sur la syntaxe concrète familière des langages source et cible.

Pour la même raison, nous optons aussi pour AToM3 qui est le seul outil où la manipulation de la syntaxe concrète est native. ATL et AGG manipulent la syntaxe abstraite seulement, bien que dans leurs nouvelles versions ils essayent tardivement d'introduire la manipulation de syntaxe concrète d'une manière moins réussie. Manipuler les règles de transformation en syntaxe concrète est un atout majeur.

5.2.3 Présentation d'AToM^3

La version 3.0 d'AToM3 est essentiellement visuelle le long du processus de transformation de modèles : Méta-modélisation, modélisation, édition de grammaire et son exécution sur des modèles concrets à travers une interface utilisateur graphique (GUI) dans un même canevas (Figure 5.1).

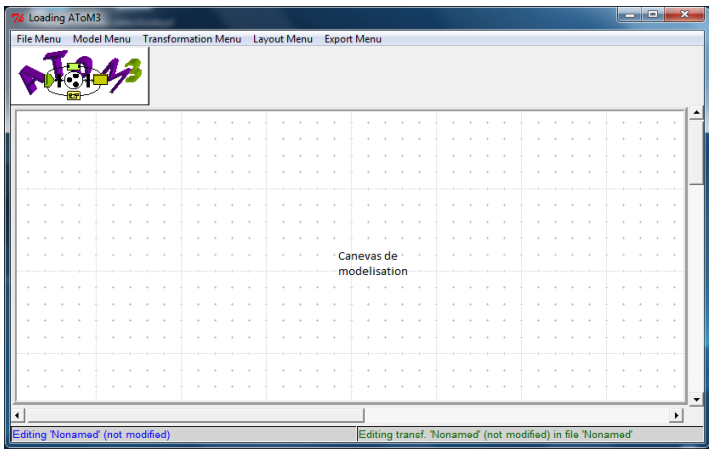


Figure 5.1- Fenêtre principale AToM3.

5.2.3.1 Méta-Modélisation avec AToM3

Pour AToM^3, les modèles sont décrits comme des graphes. Il intègre les méta-

formalismes d'Entité/Relation ou diagrammes de classe (classes, associations, héritages et multiplicités) auxquelles on peut associer des contraintes textuelles (OCL, Python) pour exprimer une méta-spécification de laquelle un outil de manipulation visuelle de modèles d'un formalisme spécifique (dans notre cas un ADL) est généré à travers le bouton *GEN* (Figure 5.1). Les transformations de modèles sont effectuées par réécriture de graphes. Les transformations elles-mêmes peuvent ainsi être exprimées d'une manière déclarative par le biais de modèles de grammaire de graphes.

Tableau 5.1- AToM3 au niveau M3 - (MDA).

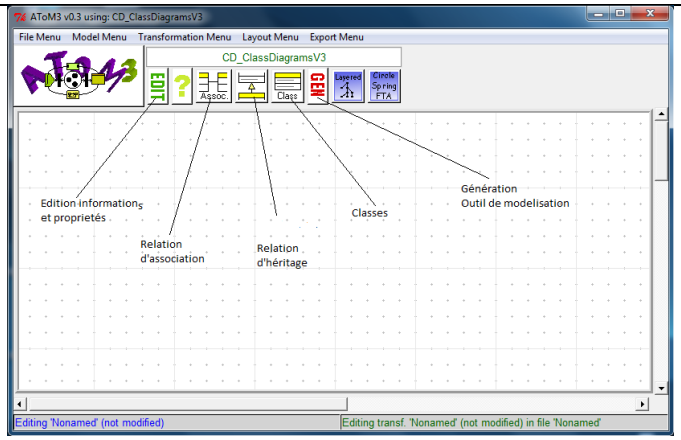


Figure 5.2- Méta-Formalisme diagramme de classes.

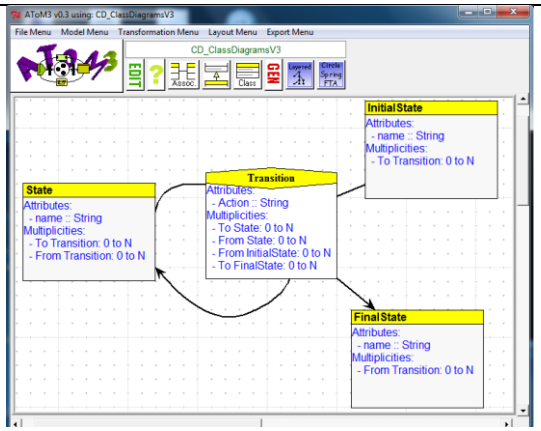


Figure 5.3- Définition du méta-modèle du DSL pour les automates.

Le Tableau 5.1 contient les deux figures qui montrent le contenu et l'usage de l'outil AToM3 au niveau M3 conformément à la démarche MDA (Méta-méta-modèle). La Figure 5.2 montre les éléments qui nous permettent de définir n'importe quel formalisme ou tout langage visuel pour un domaine spécifique (VDSL pour *Visual Domain Specific Language*). La Figure 5.3 nous montre l'usage de ces éléments pour définir le méta-modèle du langage désiré. Dans ce cas, nous avons défini la version améliorée du méta-modèle pour un automate d'état fini que nous avons présenté au chapitre 3 (Figure 3.2 (a)).

AToM^3 est capable de manipuler la syntaxe abstraite ainsi que la syntaxe concrète de tout formalisme. En effet, pour définir le diagramme de classe d'un méta-modèle, il intègre deux types d'apparences graphiques : icônes et flèches. Les premières servent à représenter

les entités (classes) et les classes associations[1] tandis que les flèches matérialisent leurs connexions tout en prenant en charge le mécanisme d'héritage.

Aussi, à chaque entité, AToM^3 peut associer une représentation graphique sous forme d'icône définie au gré de l'utilisateur lui permettant de pouvoir exprimer (instanciation) les modèles et les grammaires en syntaxe concrète. C'est d'ailleurs, l'un des grands avantages d'AToM^3 par rapport à d'autres outils (AGG, ATL) qui ne manipulent que les syntaxes abstraites sous forme de graphes typés ou classe UML. Les classes associations ou relations peuvent aussi avoir une apparence concrète en fonction du formalisme modélisé. Les utilisateurs sont plus familiers généralement à manipuler les syntaxes concrètes lors d'une modélisation (Section 5.2.2).

L'interface utilisateur d'AToM^3 change en fonction du méta-formalisme chargé en mémoire avec un bouton pour chaque classe concrète définie dans le méta-modèle pour pouvoir instancier les objets dans le canevas de modélisation. Egalement, l'apparition des boutons peut être changée en icône spécifiques en fonction de la sémantique du domaine modélisé, ce qui facilite grandement la manipulation des modèles impliqués.

Avec AToM^3, l'utilisateur peut manipuler plusieurs méta-modèles (Multi-formalismes) (Figure 5.4) en même temps pour définir la grammaire de graphe. Durant la transformation qui s'effectue dans un même canevas, le modèle est alors un « mélange » de formalismes cible et source sans aucune vérification de conformité. A la fin, seul le modèle cible est décrit et vérifié conforme à son méta-modèle. La Figure 5.4 montre la manipulation de deux formalismes distincts (Diagramme de séquence UML et automate d'états finis). Nous montrons avec cela seulement la capacité de l'outil. Dans le cadre de cette thèse, nous avons besoin que d'un seul formalisme (ADL) pour notre transformation de modèle puisqu'il s'agit d'une génération de code.

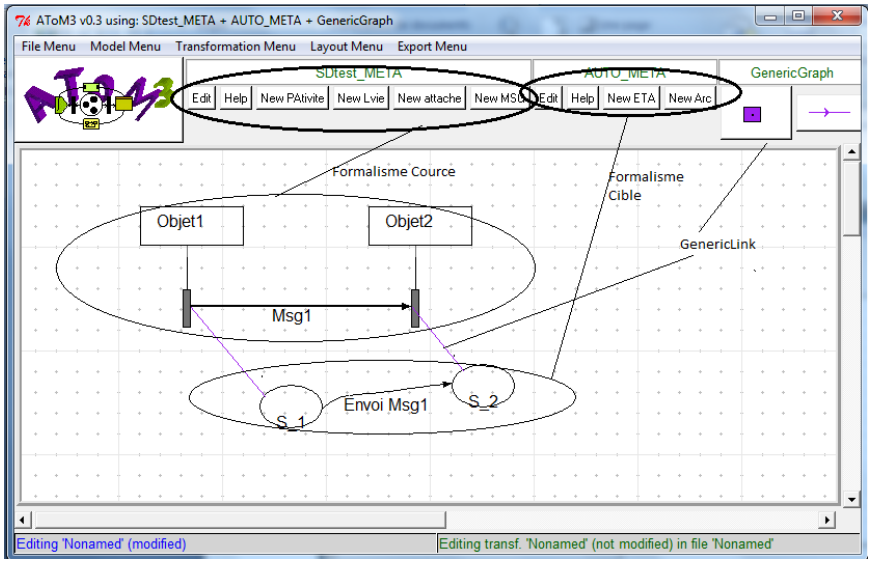


Figure 5.4- Multi-formalisme d'AToM^3.

5.2.3.2 Grammaire de graphes

La transformation de modèles est un élément crucial dans tous les efforts basés sur les modèles. Comme les modèles et les méta-modèles sont essentiellement des graphes typés et attribués, on peut les transformer en utilisant la réécriture de graphes d'une manière naturelle.

[1] Avec une légère différence par rapport à l'entité ou classe (haut rectangle arrondi) (Figure 5.3)

La transformation de modèles est spécifiée sous forme de grammaire de graphes. Les grammaires de graphes sont une représentation naturelle, formelle, visuelle, déclarative et de haut niveau de la transformation. La grammaire de graphe est composée d'un ensemble ordonné de règles selon un mécanisme de valeurs de priorité accordée à chaque règle par l'utilisateur.

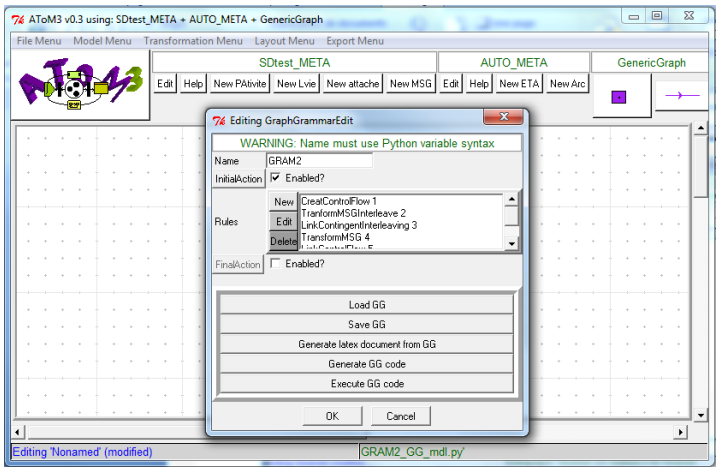


Figure 5.5- Editeur de grammaire de graphe.

La structuration de la grammaire de graphe est montrée dans la Figure 5.5, où on retrouve les actions initiales, la liste de règles avec leur ordonnancement selon leurs priorités et les actions finales. Les actions initiales peuvent contenir les scriptes Python exécuté une seule fois pour définir, en outre, les valeurs locales et globales ainsi que les ouvertures de fichiers dans le cas des générations de code. Les actions finales peuvent concerner à titre d'exemple les fermetures de fichiers et la suppression des variables globales. En plus du chargement / sauvegarde de grammaire, on peut lui générer sa documentation et sa compilation, pour être alors exécutée. L'ensemble de la grammaire peut être exportée dans un fichier pdf qui contient les patterns des règles sans l'interface de l'outil.

Chaque règle consiste en une paire de graphes (gauche LHS, droite RHS). La partie droite peut éventuellement être vide. La règle peut avoir des conditions d'applications (pré-conditions) et des actions (post-actions) qui sont respectivement vérifiées et exécutées à chaque itération durant son application (Figure 5.6). Pour des raisons de clarté, nous présentons un exemple avec des modèles sources et cibles différents (Diagramme de séquence et automate d'interaction) adapté de (Menasria, 2011). La règle de la Figure 5.6 transforme les événements d'un message entre deux lignes de vie en des états correspondants dans le modèle cible[1].

[1] Nous avons évité volontairement de donner un exemple de notre contexte (capADL) car dans le cas de génération de code, généralement, la partie gauche est identique à la partie droite. On ne distingue pas réellement les deux modèles (source-cible).

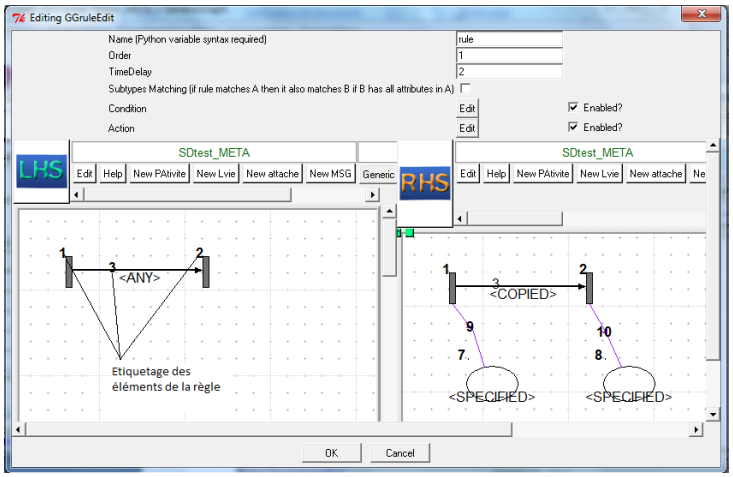


Figure 5.6- Editeur de règles de grammaire.

Les règles sont évaluées sur un graphe d'accueil qui représente le modèle à transformer ou les modèles intermédiaires. Lorsqu'une correspondance (*matching*) -conditionnée par la pré-condition- est trouvée entre la partie gauche LHS de la règle et le graphe d'accueil, le sous-graphe sous-jacent est alors remplacé par la partie droite RHS. Ce processus est réitéré par le processeur d'AToM^3 (*GraphRewritingSystem*) jusqu'à ce que la règle ne soit plus applicable. L'exécution de la grammaire de graphe ne se termine que lorsqu'aucun *matching* n'est possible pour l'ensemble des règles. Ceci dans le cas des transformations de modèles à modèles, dans les transformations de modèles vers texte aucun changement n'est apporté au graphe d'accueil et les deux parties des règles restent identiques. Nous reviendrons à ces détails dans la sous-section 5.3.2.2).

Pour chaque élément (nœuds et liens) des parties LHS et RHS, AToM^3 attribue une étiquette numérique pour l'identifier (*GGlabel*) (Figure 5.6). Quand une même étiquette apparaît dans les deux parties, l'élément correspondant est maintenu (*GGlabel* 1, 2, 3). Elle signifie l'ajout d'un élément (*GGlabel* 7, 8, 9, 10) ou sa suppression selon qu'elle apparaisse ou non dans la partie RHS et ce lors de la modélisation et l'exécution de chaque règle.

Cette notion d'étiquetage est un mécanisme pratique utilisée par les concepteurs d'AToM^3 pour omettre le graphe d'interface (*glue*) *K* largement récurrents dans la littérature des approches algébriques *Double PushOut* de transformation de graphe afin de simplifier la modélisation des règles. Les éléments de *K* sont ceux ayant les mêmes étiquettes dans les deux parties pour indiquer le morphisme de graphe (de Lara & Vangheluwe, 2004; Guerra & de Lara, 2007).

Un autre aspect important concernant ce concept d'étiquette qui est à prendre en compte (Figure 5.6). Pour tout élément de la partie LHS, les attributs doivent avoir une valeur spécifique ou une valeur quelconque (*<ANY>)* (*GGlabel* 3). Ces valeurs seront aussi comparées lors du processus de *matching*. Quant aux attributs des éléments de la partie droite RHS, ils peuvent soit garder la même valeur (*<COPIED>*) (cas de l'étiquette *GGlabel* 3) pour les éléments déjà contenus dans LHS. Ces valeurs d'attributs peuvent être modifiées avec des valeurs à spécifier (*<SPECIFIED>*) (cas des étiquettes *GGlabel* 7, 8) pour les éléments retenus de la partie gauche ou éventuellement ceux qu'on vient d'ajouter dans la partie droite. Un code Python est associé à chaque étiquette pour déterminer leurs nouvelles valeurs en fonction de tout élément de la partie gauche.

5.2.3.3 Exécution d'une grammaire de graphe

AToM^3 permet plusieurs modes d'exécutions pour une grammaire de graphes:

- continu, où juste le modèle final est affiché,
- par étape, où l'enchaînement d'exécution est contrôlé d'une manière interactive (click-souris),
- animé, par un délai d'attente fixé par l'utilisation pour chaque règle. Ce mode est surtout dédié aux simulations.

Pour pallier à l'indéterminisme du processus de *matching* on peut agir d'une manière aléatoire, interactive ou en parallèle dans le cas d'indépendance ou de disjonction des zones d'application entre les règles (Figure 5.7).

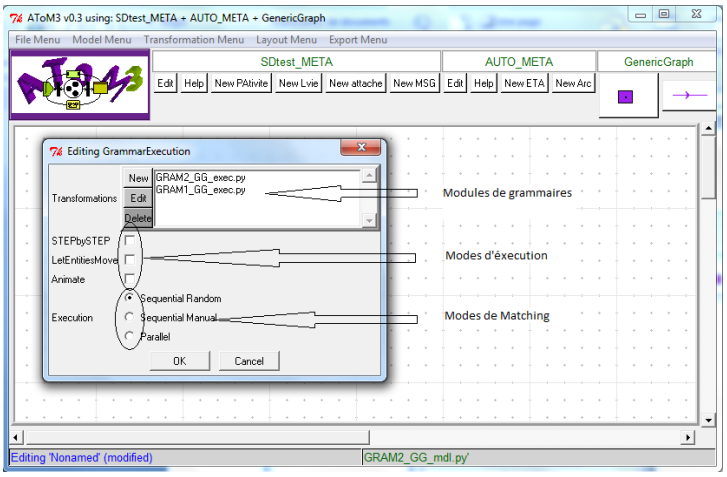


Figure 5.7- Exécution de grammaire de graphes.

Aussi, AToM^3 permet l'application d'une liste (Figure 5.7) de grammaires de graphe à un modèle. Décomposer la grammaire de graphe en blocs de règles indépendants la rend modulaire et réutilisable pour devenir plus efficace et plus facile à comprendre.

5.2.3.4 Exemple

A titre illustratif, soit donc à présenter un exemple simple pour la transformation exogène entre deux modèles distincts. Il s'agit de transformer un graphe ou un réseau de carrés en un graphe ou un réseau de cercles. Le Tableau 5.2 contient l'ensemble des concepts impliqués dans cette transformation. On retrouve les méta-modèles source/cible avec les deux syntaxes abstraite/concrète. Egalement une grammaire de graphe de trois règles suffit pour la transformation de n'importe quel modèle source vers un modèle cible. Enfin, l'application de cette grammaire sur le modèle source présenté donne le modèle cible du tableau.

5.2.4 Modélisation avec AToM^3

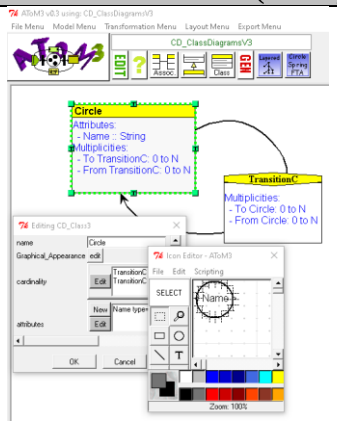
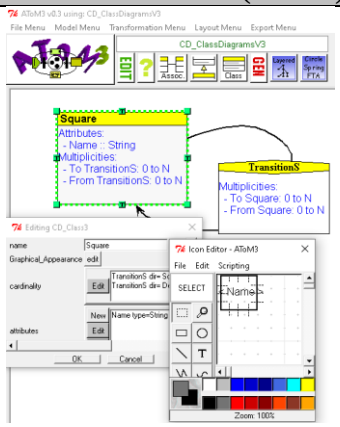
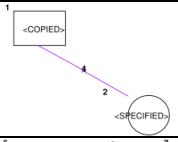
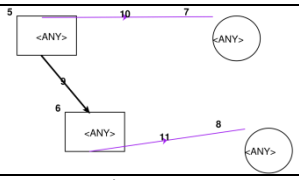
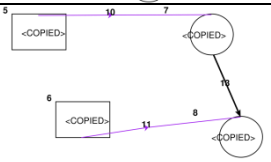
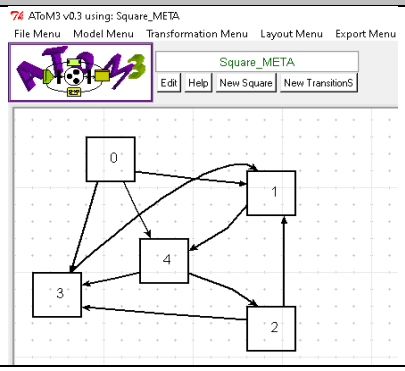
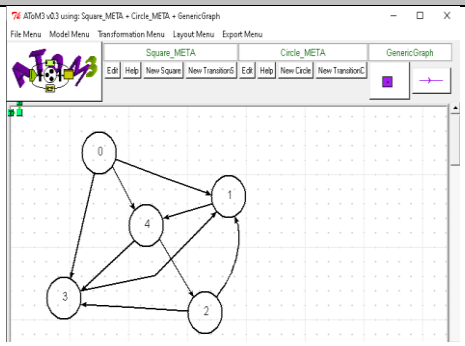
La Figure 5.8 montre la démarche ou les différentes étapes à suivre pour toute modélisation utilisant AToM^3. On identifie généralement trois phases qu'il faut appliquer de manière séquentielle :

- la première phase est réservée à la méta-modélisation et la génération automatique des environnements visuels pour modéliser les modèles (source/cible) en syntaxes concrètes respectives. Elle a pour but de définir les méta-modèles ou les langages de modélisation capables d'instancier les modèles en question. Dans notre cas, notre méta-modèle est un ADL qui peut instancier des architectures logicielles à base de connecteur orienté processus (CaP).
- la deuxième phase consiste à la modélisation de la grammaire de graphe pour définir la fonction de transformation de modèles. Les deux parties de

chaque règle sont exprimées en syntaxe concrète des modèles impliqués. Dans le cas de la génération de code, ces deux parties sont équivalentes.

- La troisième phase est celle de transformation de modèles proprement dite. Elle s'appuie sur les résultats des deux phases précédentes pour pouvoir accepter et transformer une infinité de modèles sources vers des modèles cibles correspondantes en exécutant la grammaire de graphe.

Tableau 5.2- Exemple complet : transformation de Carré à Cercle.

<table>
<tr><th>Méta-modèle source (Carré)</th><th>Méta-modèle cible (Cercle)</th></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>Règles</td><td colspan="2">Grammaire de graphe : Square2Circle</td></tr>
<tr><td></td><td>LHS</td><td>RHS</td></tr>
<tr><td>Rule 1</td><td> 
</td><td> 
</td></tr>
<tr><td>Rule 2</td><td> 
</td><td> 
</td></tr>
<tr><td>Rule 3</td><td> 
</td><td>Rien</td></tr>
<tr><td>Modèle source</td><td>Modèle cible</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

Toutes ces phases sont évidemment intégrées et assurées par le seul et même outil. On voit clairement la capacité d'AToM3 à parcourir les différents niveaux de modélisation de ceux de modèles à ceux des méta-modèles. L'outil assure également la vérification des

conformités des modèles entre les différents niveaux.

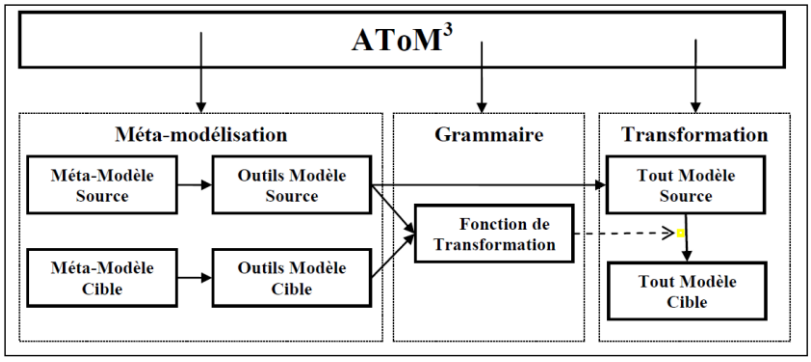


Figure 5.8- Phases de modélisation AToM^3.

5.3 Réalisation et expérimentation

Dans cette partie nous allons proposer une orientation pratique à notre solution pour montrer sa faisabilité et la validation de sa mise en œuvre. Il est évident de rappeler que cette tendance ne va en aucun cas compromettre nos hypothèses initiales. Il était convenu que l'on se doit de proposer un modèle conceptuel pour abstraire la communication dans les architectures logicielles. Il est donc question de rester dans des niveaux conceptuels loin des détails de réalisation pour conserver la propriété de généricité du modèle proposé.

L'objectif essentiel de ce chapitre est double : (i) créer par méta-modélisation un ADL baptisé (capADL) (niveau M2) pour la description d'architecture logicielles (niveau M1) à base de connecteur orienté processus. (ii) À partir de modèle d'architecture (niveau M1), on génère par transformation de modèles le code Java correspondant (M1)[1].

Heureusement, la section 5.2.2) nous rappelle les quelques avantages motivants pour le choix de l'outil AToM3. Parmi lesquels, on réitère sa faculté à parcourir tous les niveaux de modélisation MDA du niveau M0 jusqu'au niveau M3. Or, dans notre cas, la phase de méta-modélisation est concernée par les niveaux (M3 et M2) alors que la phase de transformation de modèle est concernée par les niveaux (M2 et M2). De plus, pour la transformation de modèle, nous utilisons les transformations de graphes qui sont par défaut des structures de haut niveau puisqu'elles sont déclaratives sans aucun code exécutable. Comme conséquence à tout cela, nous restons toujours à un niveau conceptuel même pour la réalisation et l'expérimentation et donc en phase avec l'hypothèse initiale.

Pour implémenter la solution en utilisant AToM^3 comme outils de méta-modélisation et de transformation de modèles par le biais des transformations de graphes (Figure 5.1 et Figure 5.8), nous sommes donc appelés à proposer le méta-modèle du modèle source, de générer l'environnement de modélisation (capADL) et de proposer la grammaire de graphes pour faire la translation des architectures logicielles à base de connecteur orienté processus vers un pseudo code Java de l'application. Nous ne précisons pas le méta-modèle du modèle cible puisque nous assurons une transformation de type modèle vers code. L'environnement capADL est utilisé pour modéliser les architectures logicielles en syntaxe concrète ainsi que pour la modélisation de la grammaire de graphe.

[1] Beaucoup considèrent le code comme un modèle et parle alors de modèle de code. C'est pour cette raison que nous faisons référence au niveau M1. Si on considère que le code est une application, alors il s'agit du niveau M0.

5.3.1 Méta-modélisation

Cette tâche consiste donc à proposer le méta-modèle source, de proposer la syntaxe concrète pour chaque entité mise en jeu et de générer enfin l'environnement visuel capADL proprement dit. Nous utilisons le méta-formalisme (classe, association) du niveau M3 qui reste conforme à lui-même (MOF) pour générer l'ADL du niveau M2 qui reste conforme au méta-formalisme. Nous sommes dans une phase pour définir la structure et la sémantique de notre langage de modélisation (capADL) des architectures logicielles à base de connecteur CaP. Par définition (Bruel et al., 2020), la méta-modélisation et la transformation de modèles promeuvent la création de langages spécifiques aux domaines (DSL).

Martin Fowler (2010) définit un langage spécifique à un domaine (DSL : *Domain Specific Language*) comme « *un langage de programmation informatique d'expressivité limitée axé sur un domaine particulier* ». Puisqu'ils sont dépourvus de la complexité habituelle des langages à usage général, les DSLs présentent un avantage particulier pour leurs utilisateurs. En effet, les experts du domaine peuvent facilement comprendre et apprendre ces langages et peuvent ainsi créer ou améliorer le code de leur application.

5.3.1.1 Méta-modélisation ADL (capADL)

À partir de la Figure 4.25 qui va décrire une configuration architecturale augmentée de stéréotype SPEM pour prendre en charge la sémantique du connecteur orienté processus logiciel (CaP), nous allons décrire notre langage de description d'architecture proposé. En fait, nous adoptons et admettons le méta-modèle de la configuration architecturale SPEM comme le méta-modèle pour notre ADL (*Architecture Description Language*) ou notre DSL (*Domain Specific Language*). Le dit méta-modèle répond à toutes nos attentes notamment pour modéliser des architectures logicielles dans des domaines distincts en définissant tous les éléments architecturaux nécessaires comme le montre la figure de conformité (Figure 4.30).

Pour cette fin, nous allons utiliser le méta-formalisme de diagramme de classe UML d'AToM^3 basé sur les classes et les classes relations (Figure 5.2). Nous proposons de modéliser le méta-modèle relatif à la Figure 4.25 pour décrire la sémantique des architectures logicielles augmentées des concepts liés aux processus logiciels. Pour cela, nous nous appuyons sur un modèle réduit de cette même figure qui traduit une configuration architecturale et ce pour des raisons de simplification.

La simplification du méta-modèle est, d'une part, opérée pour limiter le nombre de concepts afin de rendre lisible le résultat et d'éliminer les concepts qui non pas d'impacte vital sur les modèles d'architecture étudiés. D'autre part, le méta-formalisme ou méta-paradigme d'AToM3, comme le MOF d'ailleurs, ne permet de modéliser les relations de composition. A ce titre, nous devons donc changer ces relations pour créer des cycles entre les classes associées afin de ramener cette composition à la relation du genre « élément de l'élément ». Cette modification va, malheureusement, encore augmenter la complexité du modèle puisque le nombre de relations va augmenter.

Il est utile de rappeler également que l'outil AToM3 définit et instancie la relation d'association sous forme de classe qui va prendre beaucoup d'espace dans le méta-modèle résultant (Figure 5.3). Lors de l'instanciation du méta-modèle à partir du méta-formalisme, l'association est représentée par une classe UML (avec une forme légèrement ovale) au même titre qu'une classe ordinaire. Aussi, l'outil ne reconnaît pas les types énumérés, donc nous devons changer ces éléments du méta-modèle initial avec des relations d'héritages dans le méta-modèle modifié.

Ces contraintes ne sont pas totalement injustifiées. En effet, le méta-formalisme d'AToM3 joue le rôle du MOF ou du quatrième niveau dans les niveaux d'hiérarchisation et

par conséquent ne permet que deux classes : la classe « classe » et classe « association ». Ceci crée une superposition des éléments par manque d'espace sur le canevas de modélisation.

Le méta-modèle proposé est constitué de classes concrètes et abstraites, des relations visibles et invisibles afin de représenter les aspects les plus utiles pour ce projet.

- *Classe ProcessConfiguration* : c'est la classe englobante qui renferme l'ensemble de l'architecture avec tous ses éléments architecturaux. Elle est composée usuellement des classes Process*Component*, *ProcessConnector*, *PortComponent* et *PortConnectror* (connecteur et composant) pour former les attâchements. Elle est aussi considérée comme une instance de la classe abstraite ActiviteDefinition du méta-modèle SPEM au même titre que la classe *ProcessComponent* et *ProcessConnector*.
- *Classe ProcessComponent* : elle définit la structure générale du composant auquel sera associé des méthodes pour le comportement et des ports de données et de contrôle pour l'interface. Elle est composée d'au moins quatre ports (*PortComponent*: deux ports requis (Donnée, Contrôle) et deux ports fournis.
- *Classe ProcessConnector* : elle définit la structure générale du connecteur à laquelle seront ajoutés les ports de données et de contrôle pour définir son interface ainsi que sa structure interne qui renferme les éléments des processus logiciels. Par des relations de composition, cette classe renferme les classes *Actor*, *Service* et *PortConnector* qui décrit au moins quatre Ports (Requis/Fournis/Data/Ctrl).
- *Classe Actor ou Role*: c'est une classe qui compose la classe connecteur. Elle a une relation d'appartenance avec la classe connecteur, des relations de surveillance et de contrôle des différents ports et une relation de supervision avec les services du connecteur. C'est une instance de la classe *RoleDefinition* du profil SPEM.
- *Classe PortComponent* : c'est une classe abstraite qui représente les ports associés au composant. Elle peut être décrite par la classe *WorkProductionDefinition* vu que le composant est l'origine du « WorkProduct » traité par le connecteur selon les concepts SPEM.
- *Classe PortConnector* : c'est une classe abstraite qui représente les ports associés au connecteur. Elle est décrite par la classe *WorkProductionDefinition* qui représente le « WorkProduct » traité par le connecteur selon les concepts SPEM. Cette classe est liée par la relation *responsible* qui précise que le port est sous la responsabilité de la classe *Actor*.
- Les classes *Prov_PortCom*, *Req_PortCom* : représentent les classes abstraites pour identifier les deux ports requis et les deux ports fournis (minimum) associés au composant.
- Les classes *Prov_PortCon*, *Req_PortCon* : représentent les classes abstraites pour identifier les deux ports requis et les deux ports fournis (minimum) associés au connecteur.
- *Classe R_CtrPortCon* : (ou *RequiredCtrFlowPortConnector)*, cette classe définit le port pour récupérer le flux de contrôle entrant pour le connecteur. Toutes les classes concrètes relatives aux ports sont dans la nouvelle version du méta-modèle. Elles doivent exister séparément vu qu'elles sont

des classes concrètes avec des représentations graphiques distinctes. Les classes relatives aux portes sont toutes des classes qui entrent dans la composition de la classe connecteur. Cette classe est reliée avec la classe *P_CtrPortCom* avec une relation de type (1 :1) pour signifier qu'un port requis de contrôle doit être attaché à un seul port de contrôle fournit du composant amont.

- *Classe P_CtrPortCon* : (ou *ProvidedCtrFlowPortConnector*) cette classe définit le port pour exporter le flux de contrôle sortant du connecteur. Cette classe est reliée avec la classe *R_CtrPortCom* avec une relation de type (1 :1) pour signifier qu'un port fourni de contrôle doit être attaché à un seul port de contrôle requis du composant aval.
- *Classe R_DataPortCon* : (ou *RequiredDataFlowPortConnector),* cette classe définit le port pour récupérer le flux de données entrant pour le connecteur. Cette classe est reliée avec la classe *P_DataPortCom* avec une relation de type (1 :1) pour signifier qu'un port requis de contrôle doit être attaché à un seul port de données fourni du composant amont.
- *Classe P_DataPortCon* : (ou *P_DataFlowPortConnector*) cette classe définit le port pour exporter le flux de données sortant du connecteur. Cette classe est reliée avec la classe *R_DataPortCom* avec une relation de type (1 :1) pour signifier qu'un port fourni de données doit être attaché à un seul port de données requis du composant aval.
- *Classe R_CtrPortCom* : (ou *RequiredCtrFlowPortComponent),* cette classe définit le port pour récupérer le flux de contrôle entrant pour le composant. Il est attaché à un seul port fourni de contrôle du connecteur en amont s'il existe.
- *Classe P_CtrPortCom* : (ou *ProvidedCtrFlowPortComponent*) cette classe définit le port pour exporter le flux de contrôle sortant du composant. Il est attaché à un seul port requis de contrôle du connecteur en aval s'il existe.
- *Classe R_DataPortCom* : (ou *RequiredDataFlowPortComponent),* cette classe définit le port pour récupérer le flux de données entrant pour le composant. Il est attaché à un seul port fourni de données du connecteur en amont s'il existe.
- *Classe P_DataPortCom* : (ou *P_DataFlowPortComponent*) cette classe définit le port pour exporter le flux de données sortant du composant. Il est attaché à un seul port requis de contrôle du connecteur en aval s'il existe.
- Classe Service : est une classe abstraite principale qui rentre dans la composition du connecteur. Elle est liée par la relation *realize* avec la classe *Actor* et la classe *TaskDefinition* qui stipule que le service peut être une ou plusieurs tâches d'une activité définie dans le profil SPEM.
- *Classe CtrFlowService* : elle définit l'entité représentant les différents services assurés par le connecteur sur l'information de contrôle émanant des ports associés en entrée. C'est une classe abstraite et par conséquent n'a pas de représentation concrète. Dans sa forme basique, un service de contrôle lie les ports d'entrée avec les ports de sortie du connecteur (en d'autre terme il assure la fonction « Glu »). C'est pourquoi elle est liée avec les classe *P_CtrlPortCon* et *R_CtrlPortCon* avec les relations *to* et *from* de type (1 : 2..*).

- *Classe DataFlowService* : elle définit l'entité représentant les différents services assurés par le connecteur sur les données émanant des ports associés en entrée. C'est une classe abstraite et par conséquent n'a pas de représentation concrète. Dans sa forme basique, un service de données lie les ports d'entrée avec les ports de sortie du connecteur (en d'autre terme il assure la fonction « Glu »). C'est pourquoi elle est liée avec les classes *P_DataPortCon* et *R_DataPortCon* avec les relations *to* et *from* de type (1 : 2..*).
- Les classes *Transmission*, *Adaptation*, *Fusion* et *Diffusion* représentent les classes concrètes des différents types des services de transfert et de traitement de données. Ces types de services définissent la sémantique du connecteur qui les renferme.
- Les classes *Decison*, *Evaluation*, représentent les classes concrètes des différents types des services de transfert et de traitement du contrôle. Ces types de services définissent aussi la sémantique du connecteur qui les renferme.
- La classe abstraite *Precedence* représente le dernier type des services de contrôle. Elle peut être matérialisée par l'une des classes concrètes *EndBegin*, *BeginEnd*, *BeginBegin* et *EndEnd*. Ces dernières représentent les différentes formes de synchronisation des envois et des réponses possibles par les paires communicantes.
- Les classes abstraites *ActivityDefinition*, *TaskDefinition*, *RoleDefinition*, *DefaultActivityDef* et *WorkProductionDefinition* sont des classes relatives au profil SPEM précédemment défini. Elles sont utilisées ici pour donner une sémantique aux autres entités qui en dépendent. Elles peuvent en outre contenir des méthodes spécifiques lors d'une modélisation plus détaillée.

Pour des raisons de clarté, nous présentons ici même la version modifiée du méta-modèle qui respecte les exigences de l'outil sus-énumérées. Nous le présentons dans sa version originale du fait que celle représentée par l'outil est assez dense, volumineuse et difficilement lisible à cause de la limite et de la taille de l'écran par rapport à la taille du méta-modèle. Ce méta-modèle contient l'essentiel des entités et des relations impliquées dans la génération de l'outil et de sa sémantique. Le méta-modèle est représenté dans la Figure 5.9. Les différentes contraintes associées sont décrites comme pré-conditions en langage Python pour chaque classe ou relation concernée.

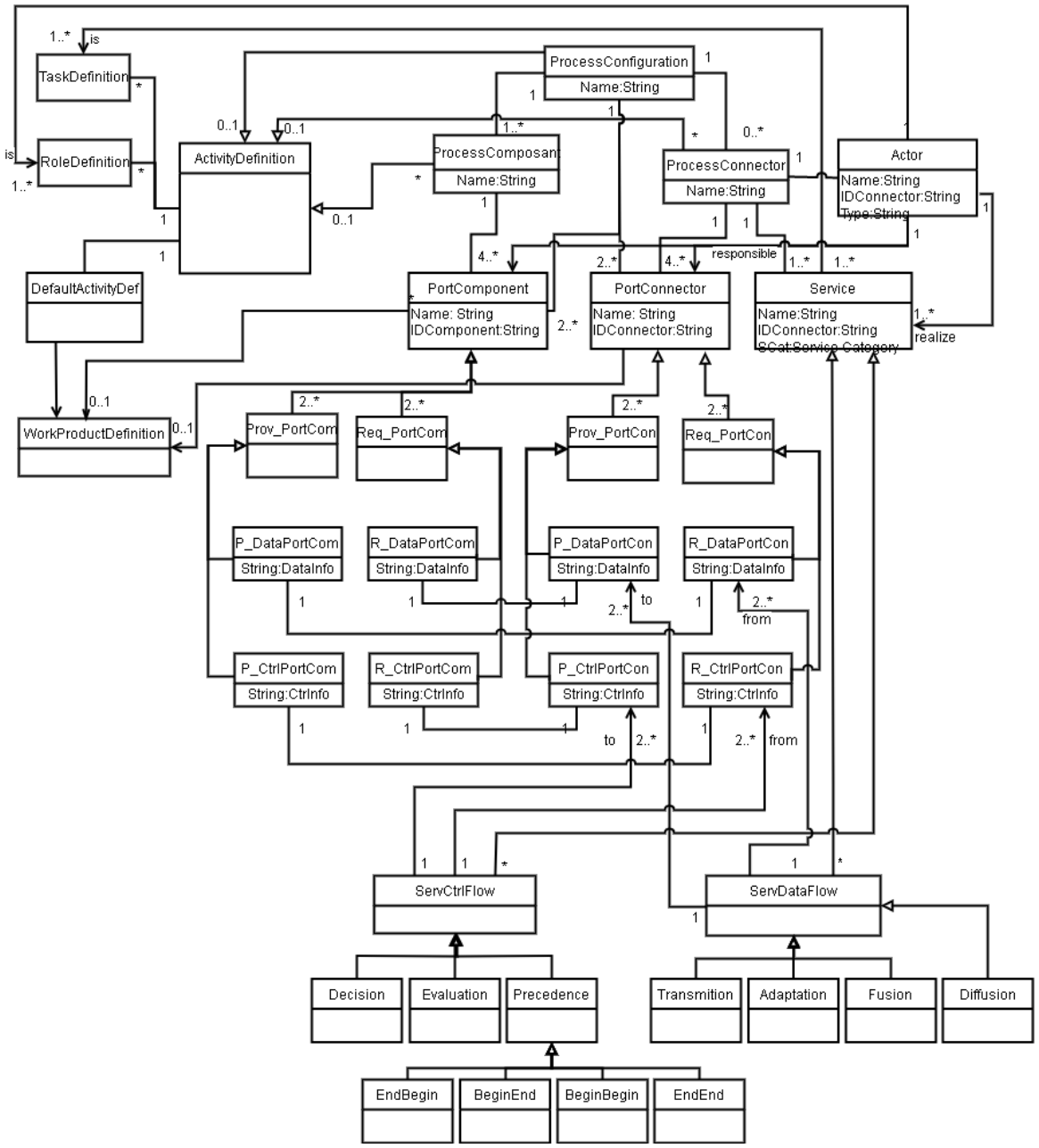


Figure 5.9- Méta-modèle capADL.

Toutefois, à titre indicatif, nous présentons dans la Figure 5.10 le méta-modèle correspondant modélisé en utilisant l'outil. C'est un modèle réduit (démuni des stéréotypes SPEM secondaires) et qui reste malgré tout illisible en dépit de sa taille. Ce méta-modèle est défini avec le méta formalisme ou le méta-méta-modèle d'AToM3 composé par deux entités de bases : classe et association. Le méta-modèle tient sur deux écrans 22 pouces.

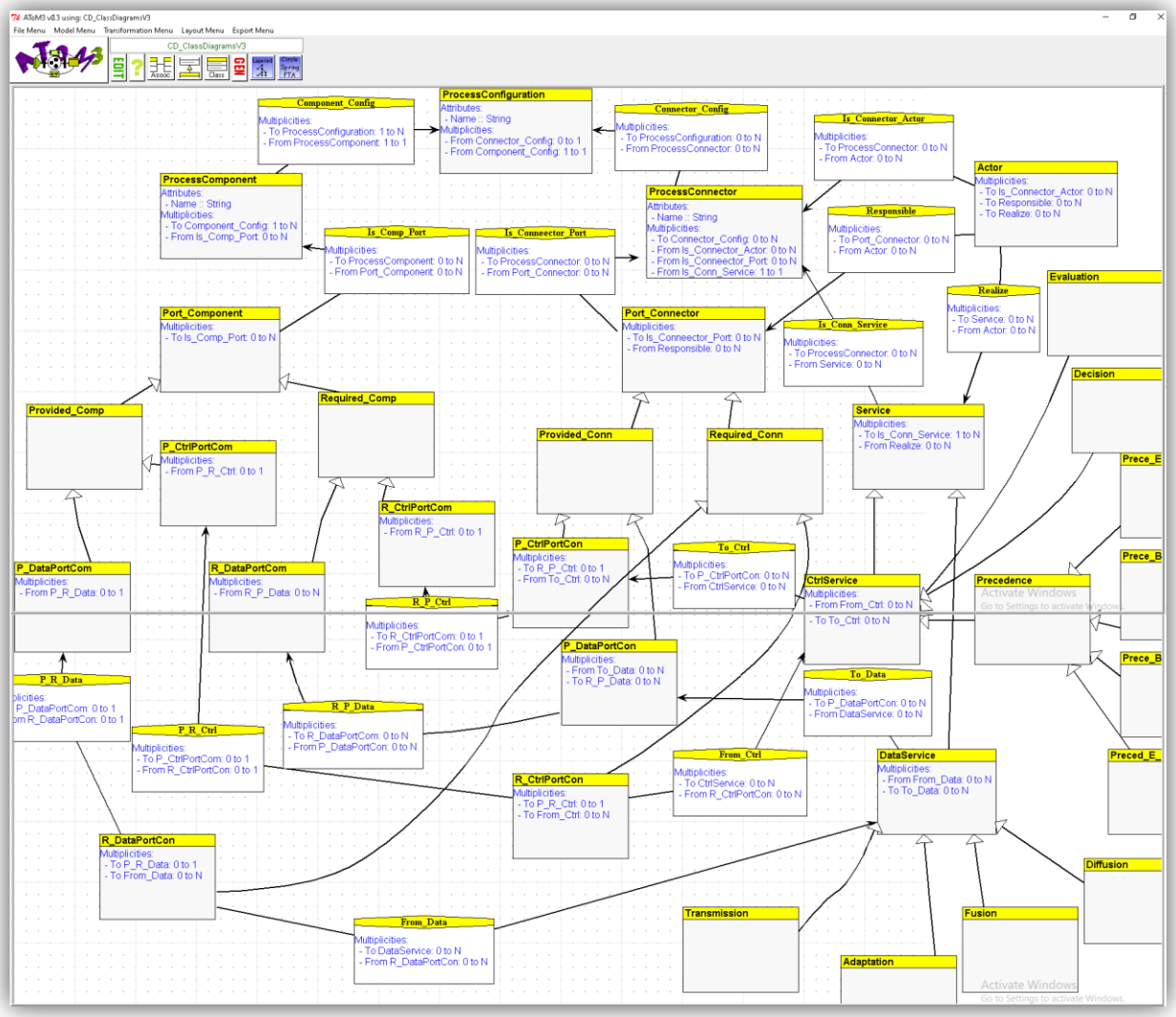


Figure 5.10- Méta-modèle cap ADL version AToM3.

5.3.1.2 Contraintes

Pour définir correctement la sémantique de notre langage de description, nous devons préciser un certain nombre de contraintes que tout modèle architectural instance doit satisfaire. Comme nous l'avons déjà mentionné, AToM3 exprime ces aspects dans le langage OCL ou directement en code Python que nous avons adopté pour les classes et les associations concernées. Aussi, beaucoup des contraintes listées sont résolues d'une manière visuelle, soit via un jeu de multiplicités soit avec l'utilisation de certaines relations d'associations.

Dans ce qui suit nous listons l'ensemble des contraintes prises en compte que nous exprimons sous forme littérale:

- Un connecteur doit relier au moins deux composants : un en amont et un en aval. Contrairement au composant, dans une configuration, un connecteur ne peut exister seul ou déconnecté d'un côté.
- Un composant peut exister seul ou peut n'avoir aucune entrée.
- Deux composants ne peuvent être liés sans un connecteur.
- Les ports requis d'un connecteur doivent être connectés impérativement

aux ports fournis du composant en amont.

- Les ports fournis d'un connecteur doivent être connectés aux ports requis du composant en aval.
- Les ports de données (requis/fournis) du connecteur ne peuvent être liés qu'aux ports de données (fournis/requis) des composants.
- Les ports de contrôle (requis/fournis) du connecteur ne peuvent être liés qu'aux ports de contrôle (fournis/requis) des composants.
- Les relations entre ports sont uniques. Un port est relié à un autre port au plus.
- Pour le service diffusion, il faut avoir un port de données en entrée et plusieurs ports de données en sortie.
- Pour le service fusion, il faut avoir plusieurs ports de données en entrée et un seul port de données en sortie.
- Pour le service transmission, il faut avoir un port en entrée et u n port en sortie pour les données.

5.3.1.3 Définition de la syntaxe concrète

Après avoir définit le méta-modèle et les contraintes associées, qui représentent la syntaxe abstraite de notre domaine de modélisation (capADL), nous définissons les apparences graphiques de chaque classe concrète ainsi que les relations spécifiques qui peuvent les lier. Cette nouvelle définition représente la syntaxe concrète associée à notre langage de modélisation en cours de définition. Cette syntaxe sert à la description de la forme finale des modèles architecturaux instanciés à partir du méta-modèle. Elle formera les éléments concrets manipulés par l'outil généré plustard dans ce document.

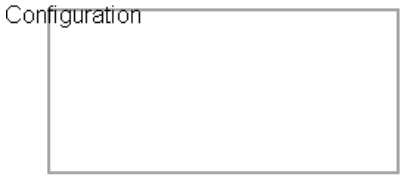
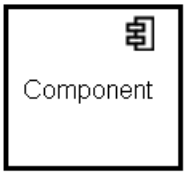
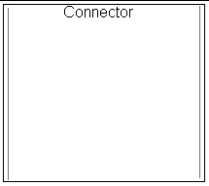
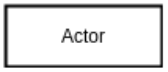
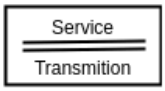
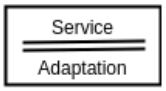
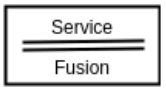
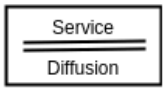
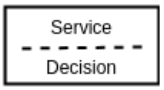
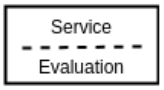
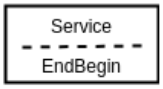
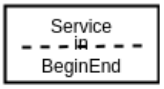
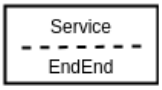
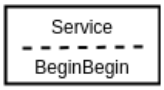
Pour cette fin, AToM3 dispose d'un outil de dessin contenant une palette de formes, de styles et de couleurs qui nous permettent de définir, pour chaque classe ou relation, sa forme graphique finale. Aussi, cet outil nous permet de positionner les quelques attributs que nous cherchons à faire apparaître sur les graphiques.

Ainsi, on rencontre encore un autre avantage d'AToM3 qui est sa capacité à manipuler en même temps les deux syntaxes (abstraite, concrète) pour langages de modélisations spécifiques qu'il peut définir. Le Tableau 5.3 résume les différentes apparences graphiques des éléments architecturaux de notre langage de description d'architecture (capADL). Elles vont définir la syntaxe concrète qui va nous permettre d'exprimer des modèles architecturaux à base de connecteur orienté processus.

La majorité des icônes proposées sont des notations standards UML 2.0 à l'exception de celles relatives au connecteur et ses éléments. La notion de connecteur n'est pas précisée dans les diagrammes UML étant donné qu'il n'est pas reconnu en tant qu'entité de première classe. Nous avons alors adopté une forme rapprochée surtout en utilisant la notation de rotule pour l'interface avec une légère variation entre les ports de données et les ports de contrôle.

On pouvait également adopter les notations adoptées par SPEM pour les éléments du processus, mais nous avons jugé plus simple d'utiliser des rectangles et des lignes. Pour plus de clarté des modèles, nous avons laissé les relations de contenance invisibles.

Tableau 5.3 - Représentation visuelle des entités de l'ADL.

Configuration	Composant	Connecteur
		
Port fourni composant –Data	Port requis composant –Data	Port fourni composant –Ctrl
		
Port requis composant –Ctrl	Port fourni connecteur –Data	Port requis connecteur –Data
		
Port fourni connecteur –Ctrl	Port requis connecteur –Ctrl	Acteur
		
Service Transmission	Service Adaptation	Service Fusion
		
Service Diffusion	Service Decision	Service Evaluation
		
Service Prec EndBegin	Service Prec BeginEnd	Service Prec EndEnd
		
Service Prec BeginBegin	Relation « *Realize* »	Relation « *Responsible* »
	*Fleche rouge*	*Fleche verte*
Relation « *From* » et « *To* »	Relation Port à (Comp, Conn)	Autres relations
Fleche grise	*Trait noir*	*Invisibles*

On pouvait également adopter les notations adoptées par SPEM pour les éléments du processus, mais nous avons jugé plus simple d'utiliser des rectangles et des lignes. Pour plus de clarté des modèles, nous avons laissé les relations de contenance invisibles.

5.3.1.4 Environnement de modélisation visuelle (capADL)

Au terme de la tâche de méta-modélisation où ont été définies la sémantique et les différentes syntaxes associées, un environnement visuel (Figure 5.11) est généré automatiquement par l'outil AToM3. Cet environnement que nous baptiserons capADL va nous permettre de modéliser n'importe quelle architecture logicielle à base de connecteur orienté processus.

Dans une syntaxe concrète, cette fois-ci, l'ADL va nous permettre de modéliser nos deux cas d'études énumérés dans la section 4.6.2) pour valider notre approche. Sur un même canevas nous pouvons modéliser le fragment d'architecture « Pipe&Filter » et le fragment de modèle cycle de vie en « V ». L'outil généré accepte les deux exemples pour rejoindre un aspect pratique à la figure de conformité de la Figure 4.30. Les modèles ainsi décrits sont assurés conformes au même méta modèle (capADL) par AToM3.

Arrivé à ce point, notre langage de description d'architectures (capADL) basées sur les connecteurs orientés processus prend naissance. Nous pouvons alors l'utiliser pour décrire tout modèle architectural avec tous les éléments architecturaux associés.

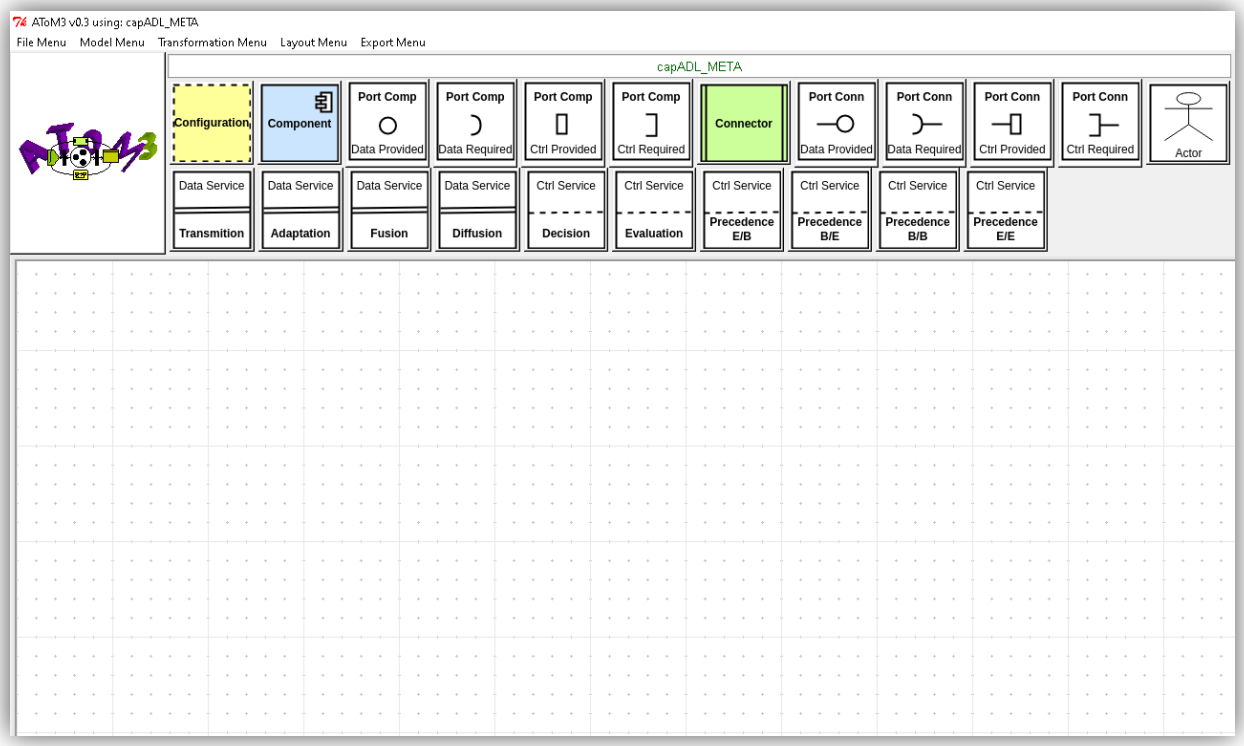


Figure 5.11- Environnement Visuel capADL.

Parallèlement à la génération d'une version exécutable, une version en open source Python est aussi proposée. Elle permet, en outre, à l'architecte d'apporter des modifications désirées dans l'interface utilisateur à titre d'exemple. Voire l'amélioration de la barre d'outils dans la Figure 5.11 avec l'apparition des icônes des éléments architecturaux au lieu des boutons textuels initiaux. On peut aussi supprimer carrément quelques boutons (comme ceux des relations) sans compromettre le bon fonctionnement de l'ADL et sa logique.

L'interface utilisateur de capADL est formée de deux parties, la barre d'outils et le canevas de modélisation. La barre d'outils contient l'ensemble des éléments architecturaux.

Le choix de l'apparence visuelle des boutons à été proposé en essayant de respecter les normes et les standards pour les éléments déjà connus. Pour les nouveaux éléments nouveaux, nous avons proposé notre propre vision des choses. Sauf pour l'acteur, ces boutons traduisent fidèlement la syntaxe concrète pour les modèles du Tableau 5.3.

La première partie de la barre d'outils contient la configuration, le composant et ses ports. La deuxième partie est réservée au connecteur et ses éléments usuels et spécifiques qui le composent. Nous avons voulu faire apparaître le concept de connecteur sur étagère que l'on peut composer à partir d'une bibliothèque de services. Nous avons déjà précisé dans le chapitre 4 que la sémantique du connecteur dépend de la sémantique des services qu'ils assurent.

Pour modéliser un connecteur, un architecte doit choisir le connecteur, les ports associés avec leurs nombres et types désirés. Ensuite, il doit choisir le/les services contrôles/données qu'il cherche à implémenter. Enfin, il faut établir les différentes relations qu'elles soient visibles ou non pour créer les différents blocs du connecteur et leurs contenus.

5.3.2 Transformation de modèle

Nous nous sommes fixés depuis le départ de conserver le statut de modèle conceptuel de haut niveau pour notre proposition. Pour pouvoir manipuler les différents modèles issus de notre ADL, que ce soit leurs passages à des modèles formels ou leurs raffinements tout respectant cette exigence, il est judicieux de faire recours aux techniques de transformation de modèles. Ces techniques sont souvent alliées sûrs des techniques de méta-modélisations pour le parcours des différents niveaux de méta modélisation de la démarche MDA en particulier ou de l'ingénierie des modèles en général.

La transformation de modèle est une activité centrale et une technique clé en main à toutes les étapes du domaine de l'ingénierie dirigée par les modèles (MDE : *Model Driven Engineering*) pour la manipulation des DSLs.

5.3.2.1 Contour de la solution

Pour montrer la faisabilité de notre contribution dans le cadre de cette thèse et pour lui donner une dimension plus pratique, nous avons proposé une démarche ouverte qui consiste à octroyer à l'architecte utilisant notre ADL la possibilité de générer automatiquement un pseudo code Java du modèle du qu'il est en train de concevoir. Ce code servira d'un support perfectible pour espérer obtenir la structure finale de l'application.

Pour cette fin, nous faisons appel aux techniques des transformations de modèles (Section 3.3) en utilisant l'approche des transformations de graphe (Section 3.3.6) avec toujours le même outil AToM3.

Nous proposons une grammaire de graphe comme un processus ouvert qui peut permettre à l'architecte d'adapter aussi le code de chaque action pour les cas inattendus. Il peut même changer la sémantique des services des connecteurs à sa guise.

En reprenant la taxonomie des transformations de modèles, on peut dire que nous allons proposer une transformation de modèle avec les propriétés suivantes :

- *endogène* (on ne définit pas le méta-modèle cible). On reste dans le même domaine sémantique puisqu'il n'est pas nécessaire de définir un méta-modèle cible.
- de *type* modèle vers code (Java),
- *sur place* (on ne change pas le modèle de départ),

- *verticale* (on passe à un niveau d'abstraction inférieur). On peut également la considérer comme une transformation horizontale si on considère le code lui-même comme un modèle.
- de *cardinalité* (1:1) (pour chaque modèle source, on a un code cible),
- en utilisant *l'approche* basée sur les transformations de graphes.

La Figure 5.12 décrit la forme générale de notre seconde contribution à définir et utiliser le langage de description d'architecture (capADL). Cette figure est un modèle concret ou instance de celui employé dans la section (3.2.1, Figure 3.14). Cette figure montre les principales phases réalisées dans ce chapitre (le méta-modèle et la génération de l'ADL ainsi que la fonction de transformation). Elle montre aussi la puissance d'AToM^3 plongé dans un environnement réel de transformation de modèles comme engin de méta-modélisation et de transformation de modèle. Il assure à lui seul toutes les tâches et interactions requises.

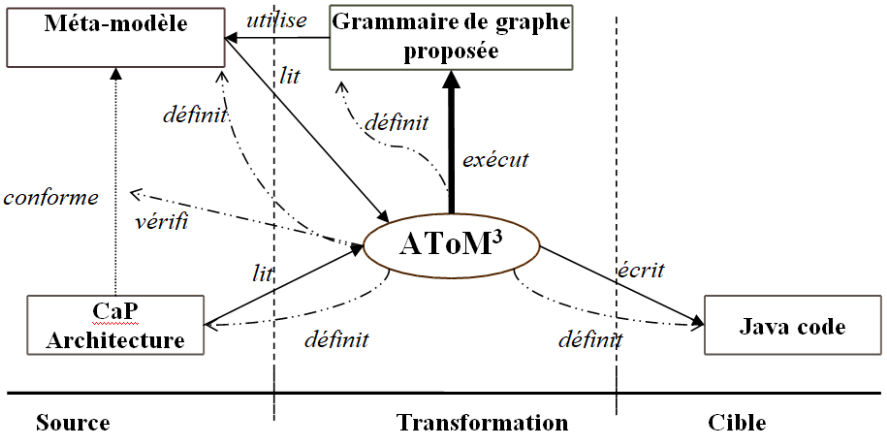


Figure 5.12- Environnement de production capADL.

Le modèle source est une architecture logicielle à base de connecteur orienté processus. Le modèle cible est un « code Java » générer pour chaque modèle d'architecture logicielle. Ce code donne la structure générale de l'application en cours de développement. Vu que la majorité des ADLs rencontrés dans la littérature et la pratique permettent de générer du code ou la structuré de code de l'application, nous avons doté notre ADL également de telles facultés.

Pour toute autre fin utile, loin de la génération de code applicatif, comme la vérification et l'analyse, on peut générer tout autre modèle (comme les modèles formels) à partir de ce point précisément. Il suffit de définir le méta-modèle associé au modèle cible et modéliser ou définir la fonction ou les règles de transformation associées.

Comme énoncé plutôt dans cette thèse, il existe plusieurs approches de transformation de modèle selon la manière avec laquelle on voit ou on considère le modèle lui-même ainsi que les éléments et les outils de transformation. Sans oublier les niveaux d'abstraction de la transformation qu'elle soit par interprétation directe ou le plus souvent par méta-modélisation.

Dans notre cas, on peut associer directement le concept de graphe au modèle et par conséquent on peut le modèle comme un graphe. Vu que nous manipulons par outrance des notions de modèle et de méta-modèle, ainsi que la disponibilité d'un outil (AToM3) qui à lui seul assure également les traitements sur les graphes, nous avons opté alors pour la transformation de graphes comme une approche triviale et intuitive pour notre transformation de modèle.

Pour autant, il est nécessaire de rappeler quelques avantages motivants pour l'usage des transformations de graphes dans le cadre des transformations de modèles. On peut citer entre autres les raisons justificatives des transformations de graphes pour dire qu'elles sont :

- *naturelles*, les modèles et les modèles architecturaux sont considérés comme des graphes attribués orientés. Donc, utiliser les techniques appliquées aux graphes est un choix évident,
- *de haut niveau*, en se situant à un niveau d'abstraction élevé pour la manipulation directe des modèles,
- *déclaratives*, on s'intéresse seulement à « Quoi » faire et non pas au « Comment » faire. C'est une forme de programmation par l'exemple sans aucune manipulation algorithmique,
- *formelles,* étant basées sur les modèles mathématiques comme la théorie des ensembles et la théorie des catégories,
- *outillées*, une bonne gamme d'outils formels est disponible notamment pour les approches algébriques,
- *visuelles,* pour la manipulation directe des modèles loin des représentations textuelles internes comme XML,
- *intuitives*, puisqu'on ne dispose pas de méthodes à suivre pour modéliser une grammaire de graphe mais plutôt une présence d'esprit. Ce point est aussi considéré par certains auteurs comme un défaut.

5.3.2.2 Grammaire de graphe proposée

Une grammaire de graphe est similaire dans le principe à une grammaire textuelle de Chomsky. Elle se diffère de la première par le fait que la partie gauche (LHS) et la partie droite (RHS) sont des graphes. Une grammaire de graphes englobe toutes les règles nécessaires à la transformation préalablement définies.

L'outil assure l'exécution des règles dans l'ordre de leurs priorités. A chaque itération, une règle peut devenir exécutable en fonction de son contexte. AToM3 réitère l'exécution de la grammaire jusqu'à ce qu'aucune règle n'est applicable. Dans le cas de génération de code, comme dans situation, un seul passage suffit du fait que la transformation ne fait pas changer le modèle.

En effet, dans notre cas, nous sommes dans une situation où la transformation de modèle consiste à opérer un passage de modèle vers code. C'est un cas qui apparaît étrange quand on applique une transformation de graphe. En effet, ce type de transformation n'a pas pour vocation de modifier le modèle initial mais de produire un code qui lui est associé. C'est pour cette raison que toutes nos règles de transformation vont apparaître avec une partie droite identique à la partie gauche dans chacune d'elles.

Notre approche est similaire à celles définies dans (Amirat et al., 2012; Rehab & Chaoui, 2015) pour la génération de code. Dans ce cas, les transformations de graphes jouent seulement le rôle de « carburant » qui fait tourner la machine de transformation en respectant, bien entendu, la logique du modèle source. La génération de code avec les transformations de graphe est action dérivée dans le processus de transformation et non pas une action centrale.

Dans tout ce qui suit, nous allons modéliser la transformation selon les particularités d'AToM3 que ce soit pour la modélisation des règles ou leurs exécutions sur des modèles concrets. Puisque la transformation n'apporte aucun changement sur le modèle source, elle ne

produit donc aucun modèle cible. Chaque règle va faire le « *matching* » ou essayer de trouver l'occurrence de la partie gauche de la règle (LHS) dans le modèle source, exécuter l'action associée à la règle et enfin elle va marquer ces éléments comme « déjà visité ». Ce marquage est essentiel pour ne pas traiter les mêmes éléments dans les itérations suivantes. La grammaire va s'exécuter jusqu'à ce qu'il y a plus aucun élément à traiter. On voit donc le rôle de la grammaire qui sert de manivelle pour faire tourner la machine de réécriture de graphes.

À partir d'un modèle source donné, le code Java correspondant à chaque élément « matché » est généré par l'action interne de la règle dans un fichier texte. Le parcours de toutes les occurrences des parties gauche (LHS) de toutes les règles de la grammaire va produire le code global correspondant au modèle source.

Pour chaque règle nous allons décrire son nom, sa priorité d'exécution, son rôle, ses condition d'exécution ou ses contraintes (en Python), son action finale qui entre autres va nous générer le code Java et enfin le patron de la règle proprement dit avec ses parties gauche (LHS) et droite (RHS). Bien entendu, il y a une phase d'initialisation de la grammaire qui consiste à l'ouverture de fichier texte, la définition et l'initialisation des variables globales et enfin quelques balises ou tags pour identifier certains points dans le code résultant. Ces tags servent à identifier les endroits d'insertion des fragments de code durant la transformation.

Dans AToM3, une grammaire de graphes (Figure 5.5) a un identifiant unique « capADL2Java » dans notre cas. Elle se compose de trois parties : (i) actions initiales « InitialAction », (ii) les règles « Rules » et (iii) les actions finales « FinalAction ».

Dans notre cas les actions finales se résument à la fermeture des fichiers textes alors que les actions initiales se résument à ce qui suit:

- l'ouverture du fichier texte qui contiendra le code Java de sortie,
- initialisation de certaines balises utilisées pour la manipulation de ce code,
- association à chaque élément du modèle cible une variable nommée « *Visited* » initialisée à la valeur « 0 » (Figure 5.13). Cette variable servira à contrôler le passage et le traitement des règles sur chaque élément du modèle. Quand une règle passe par l'élément, elle doit mettre cette valeur à « 1 ». Pour les ports, elle peut mettre cette valeur à « 2 » car nous passons par ces éléments à deux reprises. Elle l'incrémente à chaque fois de 1 dans d'autres situations.

Pour plus de clarté encore, nous n'allons pas présenter le code complet des actions des différentes règles car il va être enchevêtré avec le code et la syntaxe Python ainsi que les méthodes AToM3. Pour cela, nous allons faire en sorte de ne faire apparaître que le code utile à la compréhension en redéfinissant la méthode essentielle dans une forme plus simple.

Soit donc la convention suivante qui va remplacer dans cette rédaction le code originel (outil) par le code de substitution (thèse) :

```
conf=self.GetMatched(graphID,self.LHS.nodeWithLabel(1))
conf.Visited = 1
self.graphRewritingSystem.JavaFile.write("public class " +conf.Name.toString()+
"extends Configuration"+ "\n"+"{"+"\n" +
"public void start()"+"\n"+"{"+"\n"+"}"+"\n"+"{"+"\n"+"}"+"\n"
```

Figure 5.13- Exemple de code réel d'une action AToM3.

Le code de la Figure 5.13 est relatif à l'action de la règle qui fait la transformation de l'élément architectural « Configuration » mêlé au code et méthodes AToM3 « *GetMatched* » qui identifie l'objet actuellement *Matcher* ayant l'étiquette « 1 ». La seconde méthode « *graphRewrintingSystem* » écrit dans le fichier « *JavaText* » les chaînes de caractères et l'attribut « *Name* » de l'objet en cours « *conf* » avec la méthode « *toString* ».

Pour cela, nous remplaçons ce code (Figure 5.13) en redéfinissant les méthodes AToM3 avec une seule méthode : « *GetMatched(GGlabel).NameAttribut* », qui signifie d'écrire directement le nom de l'élément *Matcher* ayant le l'étiquette (dans ce cas « *GGlabel* =1 »). Nous obtenons alors le code de la Figure 5.14 que nous retenons comme code de l'action de la règle de transformation de l'élément « *Configuration* ». Dans ce cas, « *GetMatched(1).Name* » nous donne le nom de la configuration en cours en renvoyant par exemple « PipeAndFilter » (voir Règle 9).

```
public class GetMatched(1).Name extends Configuration
        {
           public void start()
            {

            }
        }
```

Figure 5.14- Code retenu pour l'action de la règle.

Nous adoptons la même démarche pour le reste des règles de même nature. La forme générale de la méthode d'AToM3 « *GetMatched(GGlabel).NameAttribut* » renvoi l'attribut (souvent le nom) de l'élément courant « *Matcher* » par la règle. Aussi, nous évitons de faire apparaître les tags qui nous permettent d'identifier l'endroit d'insertion de certains codes. L'ensemble des règles qui composent la grammaire de graphe proposée se présentent comme suit :

a) Règles d'initialisations

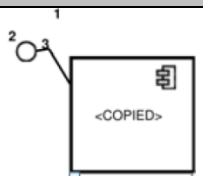
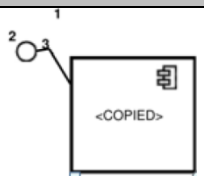
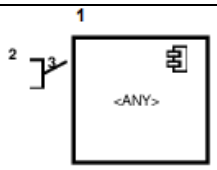
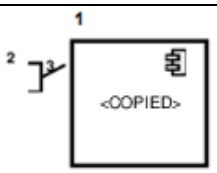
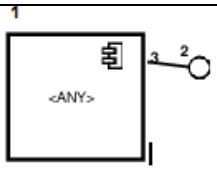
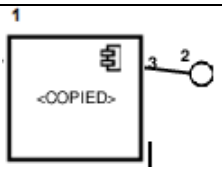
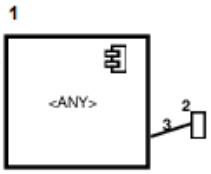
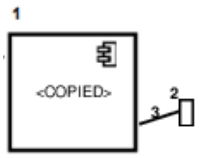
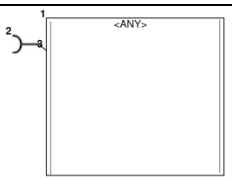
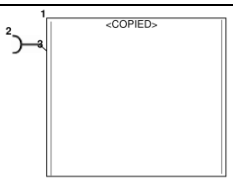
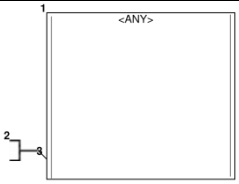
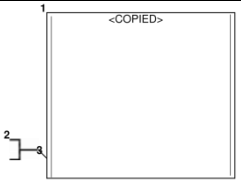
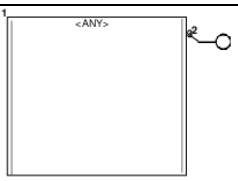
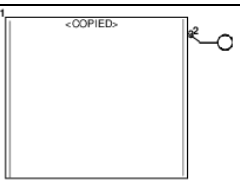
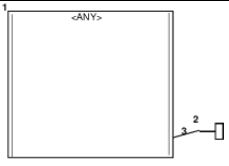
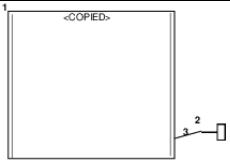
Cette catégorie de règles représente un prétraitement pour les ports notamment. Elles vont donner un nommage uniforme des différents objets « ports » et de leurs attributs. Ce nommage est effectué en fonction de l'ordre visuel pour chaque élément architectural (composant/ connecteur). Cet ordre est fixé par l'utilisateur de « capADL ». Cette catégorie se compose de 8 règles (car nous disposons de huit types de ports au total) qui se présentent comme suit :

Règle 1 à 8 : Nommage des ports (Tableau 5.4)

- *Priorité* : 01 à 8
- *Rôle* : Elles permettent de nommer et de créer les listes des ports associés aux composants et aux connecteurs. Les noms invisibles sont de la forme « Pi », où i est un compteur, pour tous les types de ports et ce en fonction de leur ordre d'apparence. Ces règles s'exécutent autant de fois qu'il y a de ports de chaque type dans le modèle.
- *Condition* : concernée par le *matching visuel*. Le port doit être traité une fois.

– *Action* : Elle met à jour les flags à « *Visited* = 1» pour les ports pour ne plus être « *matcher* » lors des itérations suivantes. Ces règles ne génèrent aucun code et elles se présentent comme suit :

Tableau 5.4- Règle de nommages des ports.

Numéro	Nom	LHS	RHS
Règle 1	*Naming_RDataPortComp*		
Règle 2	*Naming_RCtrlPortComp*		
Règle 3	*Naming_PDataPortComp*		
Règle 4	*Naming_PCtrlPortComp*		
Règle 5	*Naming_RDataPortConn*		
Règle 6	*Naming_RCtrlPortConn*		
Règle 7	*Naming_PDataPortConn*		
Règle 8	*Naming_PCtrlPortConn*		

b) Règles pour les éléments architecturaux de base

Cette catégorie de règles est réservée pour transformer les éléments architecturaux de base (configuration, composant, connecteur). Quel que soit le nombre de ces éléments dans le modèle architectural en cours, trois règles suffisent pour les transformer. Les règles en question se présentent comme suit :

Règle 9 : Transformation des configurations (Tableau 5.5)

- *Nom* : *TransConfiguration*
- *Priorité* : 09
- *Rôle* : Elle permet de générer le code associé à la configuration. En général, il existe une configuration par modèle. C'est cette règle qui donne les limites du code Java global à générer.
- *Condition* : concernée par le *matching* visuel et que l'élément ne soit pas déjà traité.
- *Action* : Elle met à jour les flags à « *Visited* » pour la configuration afin de ne plus être « matcher » lors des itérations suivantes. Cette classe génère aussi toutes les classes abstraites et concrètes relatives aux méta-modèles de base ((Figure 5.9 et Figure 5.10)). Elle génère globalement les fragments de code Java qui doivent être définis une seule fois dans un fichier texte. Ils se présentent comme suit :

```
// Au debut du fichier
public class GetMatched(1).Name extends ProcessConfiguration {
            public void start(){
             }
          }
// A la suite de cette classe : Définition de toutes les classes
//abstraites du méta-modèles (Figure 5.9 et Figure 5.10). Cette règle s'exécute
// une seule fois

public class ProcessConfiguration {
            public void start() {
           }
}
public class ProcessComponent {

}
public class ProcessConnector {

}
public class PortComponent {
    public String name;
    public String IDConnector;
}
public class R_DataPortCom extends PortComponent {
    public String dataInfo = "";
}
public class R_CtrlPortCom extends PortComponent {
    public String ctrlInfo = "";
}
public class P_DataPortCom extends PortComponent {
    public String dataInfo = "";
}
public class P_CtrlPortCom extends PortComponent {
    public String ctrlInfo = "";
}
public class PortConnector {
    public String name;
    public String IDConnector;
}
public class R_DataPortCon extends PortConnector {
    public String dataInfo = "";
}
public class R_CtrlPortCon extends PortConnector {
    public String ctrlInfo = "";
}
public class P_DataPortCon extends PortConnector {
```

```
    public String dataInfo = "";
}
public class P_CtrlPortCon extends PortConnector {
    public String ctrlInfo = "";
}

public class Service {
    public String name;
    public String IDConnector;
    public Category SCate;
}
enum Category {
    COMMUNICATION,
    COORDINATION,
    FACILITATION,
    CONVERSION,
}
public class ServDataFlow extends Service {

}
public class ServCtrlFlow extends Service {

}
public class Precedence extends ServCtrlFlow {

}
```

Tableau 5.5- Règle Transformation de la configuration.

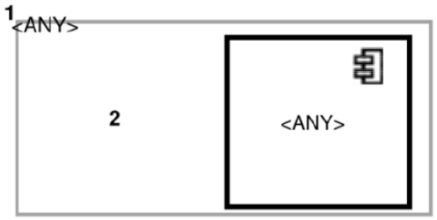
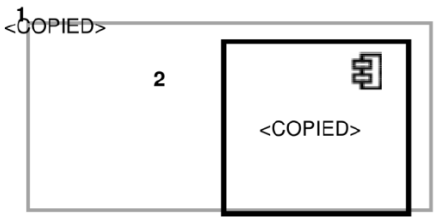
LHS	RHS
	

Règle 10 : Transformation des composants (Tableau 5.6)

- *Nom* : *TransComponent*
- *Priorité* : 10
- *Rôle* : Elle permet de générer le code associé pour chaque composant du modèle. En général, il peut exister plusieurs composants dans le modèle. C'est cette règle sera exécutée autant de fois que le nombre de composant dans l'architecture
- *Condition* : concernée par le *matching visuel* seulement.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* » pour que le composant transformé ne soit plus considéré lors des itérations suivantes. Elle génère et insère dans la classe « Configuration » les deux portions de code suivantes dans le même fichier texte: l'instanciation de l'objet composant dans la configuration et la génération de la classe « Composant » par la suite :

```
// dans debut classe configuration
         ProcessComponent GetMatched(2).Name = new  GetMatched(2).Name ();

// A la suite de la classe configuration
         public class GetMatched(2).Name extends ProcessComponent {
         }
```

Tableau 5.6- Règle transformation des composants.

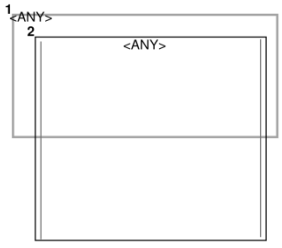
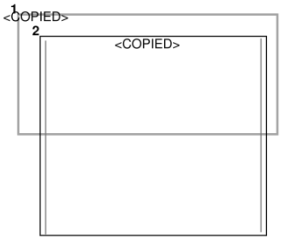
LHS	RHS
	

Règle 11: Transformation des connecteurs (Tableau 5.7)

- *Nom* : *TransConnector*
- *Priorité* : 11
- *Rôle* : Elle permet de générer le code associé pour chaque connecteur du modèle. En général, il peut exister plusieurs connecteurs dans le modèle. Cette règle sera exécutée autant de fois que de connecteur dans l'architecture
- *Condition* : concernée par le *matching* visuel seulement.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* » pour que le connecteur transformé ne sera plus considéré lors des itérations suivantes. Elle génère et insère dans la classe « Configuration » les deux portions de code suivantes dans le même fichier texte: l'instanciation de l'objet connecteur dans la configuration et la génération de la classe « Connector » avec la méthode de lancement de la transmission par la suite :

```
// dans debut classe configuration

        ProcessConnector GetMatched(2).Name = new  GetMatched(2).Name ();

// dans debut de la method start() de la classe configuration on ajoute l'invocation
// de la méthode LaunchConnectorÉxécution()relatif au connecteur en cours

        ((GetMatched(2).Name) GetMatched(2).Name).LaunchConnectorÉxécution;

// A la suite de la classe configuration

        public class GetMatched(2).Name extends ProcessConnector {

                public void LaunchConnectorÉxécution() {
                }
        }
```

Tableau 5.7- Règle transformation des connecteurs.

LHS	RHS
	

c) Règles relatives aux ports

Cette catégorie de règles est destinée à la transformation des ports en fonction de leurs sémantiques précises pour chacun des éléments architecturaux (composant, connecteur). Quel

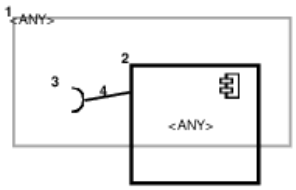
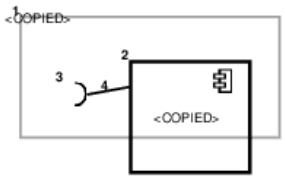
que soit le nombre de ports dans le modèle en cours de transformation, huit règles suffisent pour assurer leur traitement. Toutes les règles relatives aux ports sont aux nombre de huit en fonction des différents types de ports. Ces règles (de 12 à 19) se présentent comme suit :

Règle 12: Transformation des ports de données requis pour le composant (Tableau 5.8)

- *Nom* : *TransRequiredDataPortComponent*
- *Priorité* : 12
- *Rôle* : Elle permet de générer le code associé pour chaque port de données requis des composants du modèle. En général, il peut exister plusieurs ports de données requis pour les composants dans le modèle. Cette règle sera exécutée autant de fois que de port de ce type dans la configuration.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Component » à laquelle il est rattaché l'instance de l'objet port du type en question comme suit :

```
// au debut classe Composant en cours

public R_DataPortCom GetMatched(3).Name = new  R_DataPortCom ();
```

Tableau 5.8- Règle pour les ports de données requis des composants.

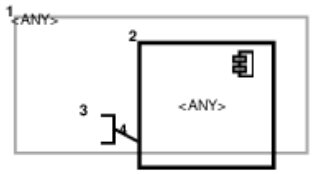
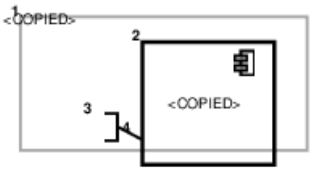
LHS	RHS
	

Règle 13: Transformation des ports de contrôle requis pour le composant (Tableau 5.9)

- *Nom* : *TransRequiredCtrlPortComponent*
- *Priorité* : 13
- *Rôle* : Elle permet de générer le code associé pour chaque port de contrôle requis des composants du modèle. Cette règle sera exécutée autant de fois que de port de ce type dans la configuration.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Component » à laquelle il est rattaché l'instance de l'objet port du type en question comme suit :

```
// au debut classe Composant en cours
            public R_CtrlPortCom GetMatched(3).Name = new  R_CtrPortCom ();
```

Tableau 5.9- Règle pour les ports de contrôle requis par les composants.

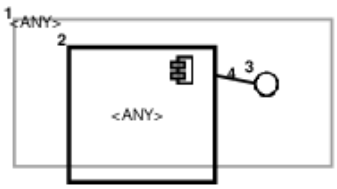
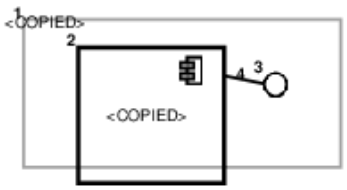
LHS	RHS
	

Règle 14: Transformation des ports de données offerts pour le composant (Tableau 5.10)

- *Nom* : *TransProvideDataPortComponent*
- *Priorité* : 14
- *Rôle* : Elle permet de générer le code associé pour chaque port de contrôle fournitt des composants du modèle. Cette règle sera exécutée autant de fois que de port de ce type dans la configuration.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Component » à laquelle il est rattaché l'instance de l'objet port du type en question comme suit :

```
// au debut classe Composant en cours
        public P_DataPortCom GetMatched(3).Name= new  P_DataPortCom ();
```

Tableau 5.10- Règle pour les ports offerts des données des composants.

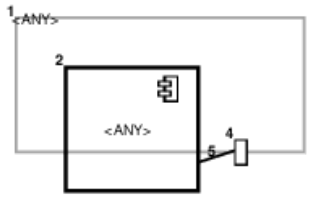
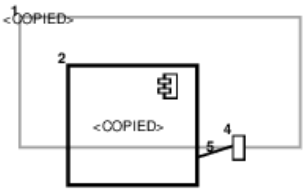
LHS	RHS
	

Règle 15: Transformation des ports de contrôle offerts par le composant (Tableau 5.11)

- *Nom* : *TransProvidedCtrlPortComponent*
- *Priorité* : 15
- *Rôle* : *Rôle* : Elle permet de générer le code associé pour chaque port de contrôle fournit des composants du modèle. Cette règle sera exécutée autant de fois que de port de ce type dans la configuration.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Component », à laquelle il est rattaché, l'instance de l'objet port du type en question comme suit :

```
// au debut classe Composant en cours
        public P_CtrlPortCom GetMatched(4).Name = new  P_DataPortCom ();
```

Tableau 5.11- Règle pour les ports de contrôle offerts des composants.

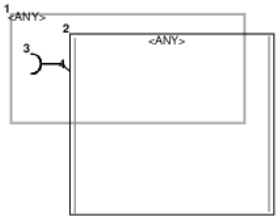
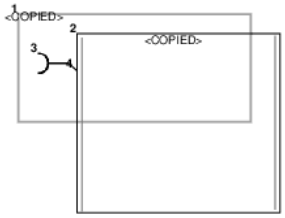
LHS	RHS
	

Règle 16: Transformation des ports de données requis pour le connecteur (Tableau 5.12)

- *Nom* : *TransRequiredDataPortConnector*
- *Priorité* : 16
- *Rôle* : Elle permet de générer le code associé pour chaque port de données requis pour les connecteurs du modèle. En général, il peut exister plusieurs ports de données requis pour les connecteurs dans le modèle. Cette règle sera exécutée autant de fois que de port de ce type dans l'architecture.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Connector », à laquelle il est rattaché, l'instance de l'objet port du type en question comme suit ::

```
// au debut classe Connector en cours
        public R_DataPortCon GetMatched(3).Name = new  R_DataPortCon ();
```

Tableau 5.12- Règle pour les ports de données requis du connecteur.

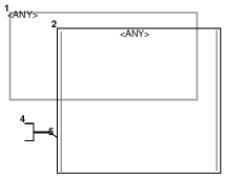
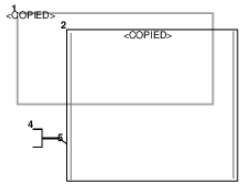
LHS	RHS
	

Règle 17: Transformation des ports de contrôle requis pour le connecteur (Tableau 5.13)

- *Nom* : *TransRequiredCtrlPortConnector*
- *Priorité* : 17
- *Rôle* : Elle permet de générer le code associé pour chaque port de contrôle requis pour les connecteurs du modèle. Cette règle sera exécutée autant de fois que de port de ce type dans l'architecture.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Connector », à laquelle il est rattaché, l'instance de l'objet port du type en question comme suit :

```
// au debut classe Connector en cours

        public R_CtrlPortCon GetMatched(4).Name = new  R_CtrlPortCon ();
```

Tableau 5.13- Règle pour les ports de contrôle requis des connecteurs.

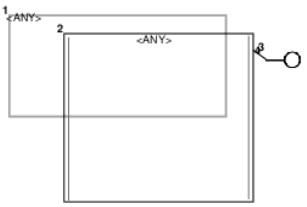
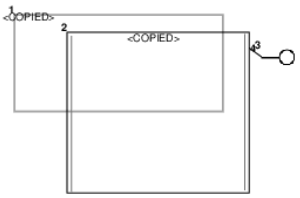
LHS	RHS
	

Règle 18: Transformation des ports de données fournis du connecteur (Tableau 5.14)

- *Nom* : *TransProvideDataPortConnector*
- *Priorité* : 18
- *Rôle* :
- *Rôle* : Elle permet de générer le code associé pour chaque port de données offert par les connecteurs du modèle. Cette règle sera exécutée autant de fois que de port de ce type dans l'architecture.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Connector », à laquelle il est rattaché, l'instance de l'objet port du type en question comme suit :

```
// au debut classe Connector en cours

        public P_DataPortCon GetMatched(3).Name = new  P_DataPortCon ();
```

Tableau 5.14- Règle pour les ports de données fournis par les connecteurs.

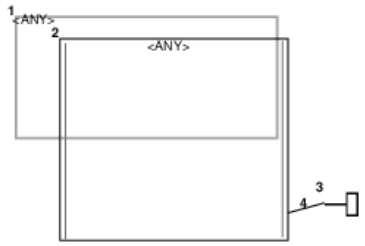
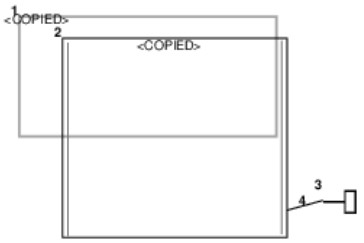
LHS	RHS
	

Règle 19: Transformation des ports de contrôle requis pour le connecteur (Tableau 5.15)

- *Nom* : *TransRequiredCtrlPortComponent*
- *Priorité* : 19
- Elle permet de générer le code associé pour chaque port de contrôle offert par les connecteurs du modèle. Cette règle sera exécutée autant de fois que de port de ce type dans l'architecture.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* == 1» c-à-d que le port a été déjà nommé.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited* =2». Elle génère et insère dans la classe « Connector », à laquelle il est rattaché, l'instance de l'objet port du type en question comme suit :

```
// au debut classe Connector en cours

        public P_CtrlPortCon GetMatched(3).Name = new  P_CtrlPortCon ();
```

Tableau 5.15- Règle pour les ports de contrôle offerts par les connecteurs.

LHS	RHS
	

d) Règles relatives à l' « Acteur »

Cette catégorie de règle est consacrée à traiter le concept de l'« Acteur » qui l'un des constituant structurel et sémantique du connecteur. Elle donne une vue pratique de son vrai rôle dans le connecteur. Ces règles définissent le noyau de ce concept clé, sa surveillance des ports d'entrée et sa prise en charge des différents services du connecteur. Cette catégorie est constituée alors de trois règles qui se présentent comme suit :

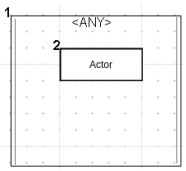
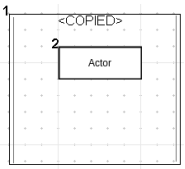
Règle 20: Transformation de l'Acteur (Tableau 5.16)

- *Nom* : *TransformActor*
- *Priorité* : 20
- Rôle : Elle permet de générer le code qui va prendre en charge le concept de l' « Actor ». Il représente le noyau qui contrôle le fonctionnement du connecteur. Son rôle est de surveiller les ports en entrée en les scrutant continuellement. A chaque changement dans ces ports, il lance les services associés (Données/Contrôle) impliqués dans l'architecture. Après traitement, il active les ports de sortie (Set) et réinitialise les ports d'entrée (Reset). Cette règle sera exécutée pour chaque connecteur de l'architecture, chaque connecteur est doté d'un seul « Actor ».
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé à cet objet.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Elle génère et insère dans la classe « Connector », à laquelle il est rattaché, la définition de la classe de l'acteur associé au connecteur en cours et puis son instance dans le corps de la classe « Connector » en ajoutant le lancement de l'acteur dans la méthode de lancement en plus de l'instance de l'objet « Actor ».. Aussi, elle définit la classe Abstraite « Actor » à la suite de la classe connecteur qui le concerne. Les fragments de code associés se présentent comme suit:

```
// au debut classe Connector en cours
        public P_CtrlPortCon GetMatched(3).Name;
        class "Actor"+ GetMatched(2).Name extends Actor {
                @Override
                public void run() {
                    surveillanceActor();
                    super.run();
                }
                public void surveillanceActor() {

                    if () {
                    }

                }
        }
// A la suite, l'instanciation de l'objet Actor dans classe Connecteur en cours
        public Actor GetMatched(3).Name a1 = new "Actor"+ GetMatched(2).Name;

// Dans la méthode de lancement d'exécution de Connecteur, on ajouter l'appel de
```

```
// méthode d'exécution du par défaut de l'Acteur

          (("Actor"+ GetMatched(2).Name) this. GetMatched(3).Name a1).run()

// A la suite de la classe Connecteur définir la classe abstraite globale de
//  l'Acteur
          public class Actor implements Runnable {
              String name;
              String nature;
              String mode;

              public static Actor defineActor() {
                  Actor actor = null;
                  return actor;
              }
              @Override
              public void run() {
              }
          }
```

Tableau 5.16- Règle de transformation de l'Acteur.

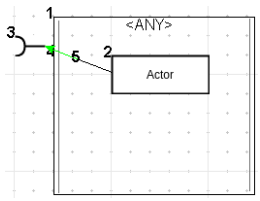
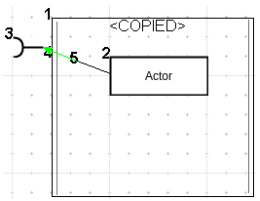
LHS	RHS
	

Règle 21: Transformation de la prise en charge des ports de données en entrée par l'Acteur (Tableau 5.17)

- *Nom* : *MonitoringRequiredDataPort*
- *Priorité* : 21
- Rôle : Elle permet de générer le code en opération logique dans le test effectué par l'acteur sur les ports de données en entrées. Le test consiste à vérifier si le l'attribut « DataInfo » du port en cours est différent d'une chaîne vide, en d'autre termes, il a été activé par le composant amont.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé à ces deux objets (port et actor).
- *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Elle génère et insère dans la méthode de surveillance de classe « Actor », à laquelle il est rattaché, le code suivant pour construire la condition de test. Il suffit que l'un des ports soit activé pour lancer le reste de la logique de l'acteur qui sera définie plus loin:

```
// entre les parenthèses de la condition if on ajoute le test sur le port

          GetMatched(3).DataInfo != null ||
```

Tableau 5.17- Règle de surveillance du port de données d'entrée par l'Acteur.

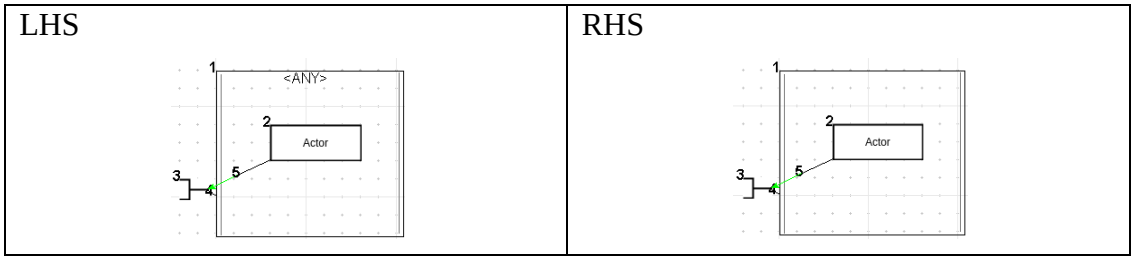
LHS	RHS
	

Règle 22: Transformation de la prise en charge des ports de contrôle en entrée par l'Acteur (Tableau 5.18) :

- *Nom* : *MonitoringRequiredCtrlPort*
- *Priorité* : 22
- Rôle : Elle permet de générer le code en opération logique dans le test effectué par l'acteur sur les ports de contrôle en entrées. Le test consiste vérifier si le l'attribut « CtrlInfor » du port en cours est différent d'une chaîne vide, en d'autre termes, il a été activé par le composant amont.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé à ces deux objets (port et actor).
- *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Elle génère et insère dans la méthode de surveillance de classe « Actor », à laquelle il est rattaché, le code suivant pour construire la condition de test. Il suffit que l'un des ports soit activé pour lancer le reste de la logique de l'acteur qui sera définie plus loin::

```
// entre les parenthèses de la condition « if » on ajoute le test sur le port
        GetMatched(3).CtrlInfo != null ||
```

Tableau 5.18- Règle de surveillance du port de ctrl en entrée par l'Acteur.



e) Règles relatives aux services

Cette catégorie de règle est consacrée à traiter le concept de « Service» qui l'un des constituant structurel et sémantique du connecteur. Elle donne une vue pratique de la sémantique des différents services pour chaque connecteur. La logique de chaque connecteur n'est pas traitée en détail. Ces règles indiquent aussi les relations des services avec les ports ainsi que les mécanismes de leurs prises en charge par l'entité « Acteur ». Ces règles traitent aussi les mises à jour des ports de sortie du connecteur. Ces règles traitent uniquement les services impliqués dans les deux études de cas, toutefois elles prennent différentes variantes en cardinalité (1x1 et 1xn). A juste titre, cette catégorie est constituée de six règles (de 23 à 28) pour les quatre services traités. L'ensemble de ces règles se présente comme suit :

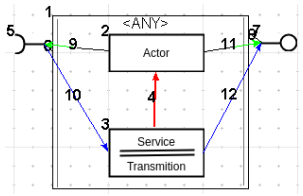
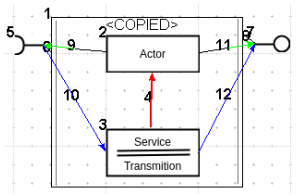
Règle 23: Transformation de service de données « Transmition » (Tableau 5.19)

- *Nom* : *TransformDataServiceTransmition*
- *Priorité* : 23
- Rôle : Elle permet de générer les différents fragments de code associés au service de données « Transmition » ainsi que l'association des ports de données (entrée/sortie) qui lui sont rattachés. Entre autres, elle génère le code de la méthode du service de données en question avec ses paramètres (les ports). Ce service est du type (1x1) c-à-d qu'il a un port de donnée en entrée et un seul port en sortie. Cette règle s'exécute une fois pour chaque connecteur.

– *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au service actuel et ses différentes liaisons.
– *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Elle génère et insère dans la classe « Actor », à laquelle il est rattaché, le code de l'invocation de la méthode qui définit la logique de ce service. Elle instancie dans la classe « Connector » associée l'objet du service courant. Enfin, elle définit la classe concrète du service avec la méthode qui détermine sa logique avec les paramètres (ports) impliqués. Enfin, à la suite du corps de la logique de cette méthode, elle ajoute un code très important qui consiste à faire le « Set » du port de sortie en fonction de celui en entrée. Les différents codes se présentent comme suit:

```
// Dans la classe « Connector » en cours et dans le corps du bloc « if » de la
//méthode de la méthode de surveillance de l'acteur, faire l'invocation de méthode
//qui décrit la logique du service en cours avec des arguments ports (entrée/sortie)

          ((Transmition) GetMatched(3).Name).TransmitionLogic(GetMatched(5).Name ,
                                        GetMatched(7).Name);// Avec Casting

// Dans l'instanciation de l'objet dans classe « Connector » on ajoute

              public ServDataFlow GetMatched(3).Name = new Transmition();

//  A la suite de la classe « Connector », ajouter la classe pour le service
// « Transmition » et la définition de la méthode qui définit sa logique avec des
// paramètres ports (entrée/sortie) typés. +« Set » ou affectation du port de
// données de sortie en fonction de celui en entrée

          public class Transmition extends ServDataFlow {
              public void TransmitionLogic(R_DataPortCon GetMatched(5).Name,
                                           P_DataPortCon GetMatched(7).Name) {
                  // logique de service transmition.
                  {
                      // core
                  }
                    // Affectation (Set) du ports de données de sortie
                 GetMatched(7).Name.dataInfo = GetMatched(5).Name.dataInfo;
            }
        }
```

Tableau 5.19- Règle pour le service de données « Transmisstion ».

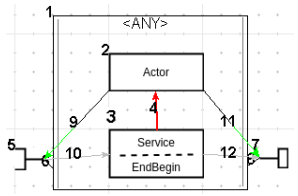
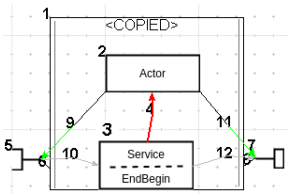
LHS	RHS
	

Règle 24: Transformation de services de contrôle « EndBegin » (Tableau 5.20)
– *Nom* : *TransformCtrServiceEndBegin*
– *Priorité* : 24
– Rôle : Elle permet de générer les différents fragments de code associés au service de contrôle « EndBegin » ainsi que l'association des ports de contrôle (entrée/sortie) qui lui sont rattachés. Entre autres, elle génère le code de la méthode du service de données en question avec ses paramètres (les ports). Ce service est du type (1x1) c-à-d qu'il a un port de données en entrée et un seul port en sortie. Cette règle s'exécute une fois pour chaque connecteur.
– *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au service actuel.

– *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Sachant que ce service admet deux ports (1 en entrée et 1 en sortie), cette règle génère et insère les paramètres de la méthode qui définit la logique de ce service dans invocation (dans la class « Actor » et sa définition dans la classe « EndBegin »). Enfin, à la suite du corps de la logique de cette méthode, elle ajoute un code très important qui consiste à faire le « Set » du port de sortie en fonction de celui en entrée. Les différents codes se présentent comme suit:

```
// Dans la classe « Connector » en cours et dans le corps du bloc « if » de la
//méthode de la méthode de surveillance de l'acteur, faire l'invocation de méthode
//qui décrit la logique du service en cours avec des arguments ports (entrée/sortie)

          ((EndBegin) GetMatched(3).Name).EndBeginLogic(GetMatched(5).Name ,
                                        GetMatched(7).Name);// Avec Casting

// Dans l'instanciation de l'objet dans classe « Connector » on ajoute

              public ServDataFlow GetMatched(3).Name = new Transmition();

//  A la suite de la classe « Connector », ajouter la classe pour le service
// « EndBegin » et la définition de la méthode qui définit sa logique avec des
// paramètres ports (entrée/sortie) typés. + « Set » ou affectation du port de
// contrôle de sortie en fonction de celui en entrée

          public class EndBegin extends Precedence {
              public void EndBeginLogic(R_DataPortCon GetMatched(5).Name,
                                        P_DataPortCon GetMatched(7).Name) {
                  // logique de service EndBegin.
                  {
                      // core
                  }
                    // Affectation (Set) du port de contrôle de sortie
                  GetMatched(7).Name.ctrlInfo = GetMatched(5).Name.ctrlInfo;
              }
          }
```

Tableau 5.20- Règle pour le service de contrôle « EndBegin ».

LHS	RHS
	

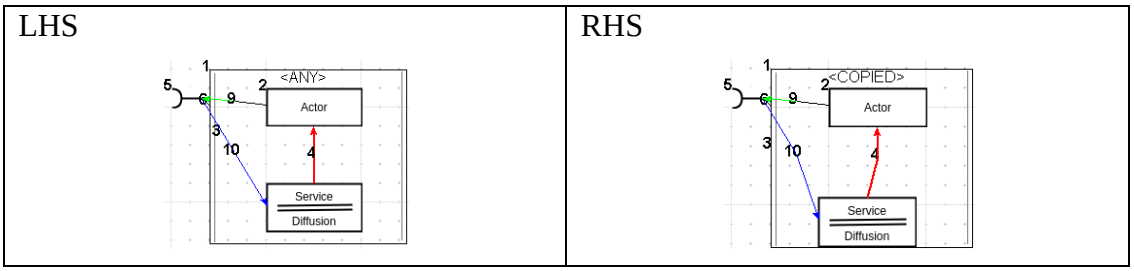
Règle 25: Transformation de service de données « Diffusion » (Tableau 5.21)

– *Nom* : *TransformDataServiceDiffusion*
– *Priorité* : 25
– Rôle : Elle permet de générer les différents fragments de code associés au service de données « Diffusion ». Entre autres, elle génère le code de la méthode du service de données en question avec ses paramètres (le port d'entrée seulement). Ce service est du type (1xn) c-à-d qu'il a un port de données en entrée et peut avoir plusieurs ports en sortie. Cette règle traite le port d'entrée seulement, les ports de sortie sont traités par la règle suivante. Cette règle s'exécute une fois pour chaque connecteur.
– *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au service actuel.
– *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Elle génère et insère dans la classe « Actor », à laquelle il est rattaché, le code de l'invocation de la méthode qui définit la logique de ce service. Elle instancie dans la classe « Connector » associée l'objet du service courant. Enfin, elle définit la classe

concrète du service avec la méthode qui détermine sa logique avec le paramètre (port entrée) impliqué. Les différents codes se présentent comme suit:

```
// Dans la classe « Connector » en cours et dans le corps du bloc « if » de la
//méthode de la méthode de surveillance de l'acteur, faire l'invocation de méthode
//qui décrit la logique du service en cours avec un argument port (entrée seulement)

          ((Diffusion) GetMatched(3).Name). DiffusionLogic(GetMatched(5).Name ,);

// Dans l'instanciation de l'objet dans classe « Connector » on ajoute

              public ServDataFlow GetMatched(3).Name = new Diffusion ();

//  A la suite de la classe « Connector », ajouter la classe pour le service
// « Diffusion » et la définition de la méthode qui définit sa logique avec des
// paramètres ports (entrée seulement) typés en plus du type des arguments (ports)
// de sortie sans les nommer.

          public class Diffusion extends ServDataFlow {
              public void TransmitionLogic(R_DataPortCon GetMatched(5).Name,
                                           P_DataPortCon ) {
                  // logique de service Diffusion.
                  {
                      // core
                  }
              }
          }
```

Tableau 5.21- Règle pour le service de données « Diffusion ».



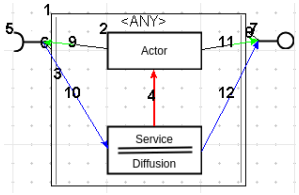
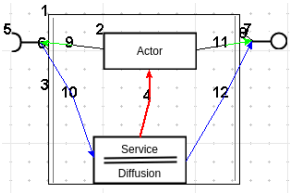
Règle 26: Transformation associant les ports de données avec le service de données « Diffusion » (Tableau 5.22)

- *Nom* : *AssociationDataPortToDataServiceDiffusion*
- *Priorité* : 26
- Rôle : Elle permet d'associer les ports de données (sortie seulement) avec le service de données « Diffusion ». Entre autres, elle remplit les paramètres (ports de sortie) des méthodes laissés vides par la règle précédente en plus de l'affectation (SET) des ports de sortie par le port d'entrée. Ce service est de cardinalité (1xn) par conséquent on peut avoir plusieurs ports de sortie. Pour chaque connecteur, cette règle s'exécute autant de fois que de nombre de ports de sortie.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au service actuel. Le port d'entrée n'est pas concerné par le test.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Cette règle génère et insère les paramètres (ports de sortie) de la méthode qui définit la logique de ce service dans invocation (class « Actor » et dans sa définition classe « Diffusion ». Enfin, à la suite du corps de la logique de cette méthode, elle ajoute un code très important qui consiste à faire le « Set » des ports de sortie en fonction de celui en entrée. Les différents codes se présentent comme suit:

```
// Dans la classe « Actor » en cours  on ajoute les paramètres (les ports de sortie)
// lors de l'invocation de la méthodes -entre les parenthèses définit par la règle
// précédente- suivi d'une virgule
```

```
            GetMatched(7).Name,

// Dans la définition de la méthode dans la classe « Diffusion », on ajoute les
// paramètres de sortie dont le type à été ajouté par la precedente règle avant la
// parenthèse fermante suivi d'une virgule

            GetMatched(7).Name,

//  à la suite du corps de la logique de cette méthode, ajoute un code pour faire
// les « Set » ou mise à jour des ports de sortie en fonction de celui en entrée en
//  métant ajour l'attribut « datalInfo ».

                  GetMatched(7).Name.dataInfo = GetMatched(5).Name.dataInfo;
```

Tableau 5.22- Règle pour l'association des ports de données avec le service de données « Diffusion ».

LHS	RHS
	

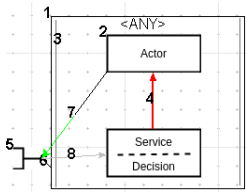
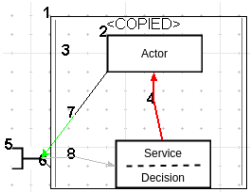
Règle 27: Transformation de service de contrôle « Decision » (Tableau 5.23)

- *Nom* : *TransformCtrlServiceDecision*
- *Priorité* : 27
- Rôle : Elle permet de générer les différents fragments de code associés au service de de contrôle « Diffusion ». Entre autres, elle génère le code de la méthode du service de contrôle en question avec ses paramètres (le port d'entrée seulement). Ce service est du type (nxn) c-à-d qu'il peut avoir plusieurs ports de donnée en entrée/sortie. Cette règle traite le port d'entrée seulement, les ports de sortie sont traités par la règle suivante. On admet une restriction à ce niveau pour faire fonctionner les deux services « Diffusion » et « Decision » ensemble d'où on considère le service « Decision » comme étant (1xn). Cette règle s'exécute alors une fois pour chaque connecteur.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au service actuel.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Elle génère et insère dans la classe « Actor », à laquelle il est rattaché, le code de l'invocation de la méthode qui définit la logique de ce service. Elle instancie dans la classe « Connector » associée l'objet du service courant. Enfin, elle définit la classe concrète du service avec la méthode qui détermine sa logique avec le paramètre (port entrée) impliqué. Les différents codes se présentent comme suit:

```
// Dans la classe « Connector » en cours et dans le corps du bloc « if » de la
//méthode de la méthode de surveillance de l'acteur, faire l'invocation de méthode
//qui décrit la logique du service en cours avec un argument port (entrée seulement)

          ((Decision) GetMatched(3).Name). DecisionLogic(GetMatched(5).Name ,);

// Dans l'instanciation de l'objet dans classe « Connector » on ajoute

              public ServDataFlow GetMatched(3).Name = new Decision();

//  A la suite de la classe « Connector », ajouter la classe pour le service
// « Decision» et la définition de la méthode qui définit sa logique avec des
// paramètres ports (entrée seulement) typés en plus du type des arguments (ports)
// de sortie sans les nommer.
```

```
public class Decision extends ServDataFlow {
    public void DecisionLogic(R_DataPortCon GetMatched(5).Name,
                              P DataPortCon ) {
        // logique de service Decision.
        {
            // core
        }
    }
}
```

Tableau 5.23- Règle pour le service de contrôle « Decision».

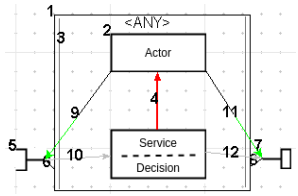
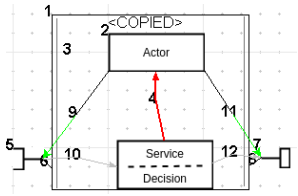
LHS	RHS
	

Règle 28: Transformation associant les ports de contrôle de sortie avec le service de données « Diffusion » (Tableau 5.24)

- *Nom* : *AssociationCtrlPortToCtrlServiceDecision*
- *Priorité* : 28
- Rôle : Elle permet d'associer les ports de données (sortie seulement) avec le service de données « Decision ». Entre autres, elle remplit les paramètres (ports de sortie) des méthodes laissés vides par la règle précédente en plus de l'affectation (SET) des ports de sortie par le port d'entrée. Ce service est supposé de cardinalité (1xn) par conséquent on peut avoir plusieurs ports de sortie. Pour chaque connecteur, cette règle s'exécute autant de fois que de nombre de ports de sortie.
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au service actuel. Le port d'entrée n'est pas concerné par le test.
- *Action* : Elle met à jour les flags associés à la valeur « *Visited*». Cette règle génère et insère les paramètres (ports de sortie) de la méthode qui définit la logique de ce service dans invocation (class « Actor » et dans sa définition classe « Decision »). Enfin, à la suite du corps de la logique de cette méthode, elle ajoute un code très important qui consiste à faire le « Set » des ports de sortie en fonction de celui en entrée. Les différents codes se présentent comme suit:

```
// Dans la classe « Actor » en cours  on ajoute les paramètres (les ports de sortie)
// lors de l'invocation de la méthodes -entre les parenthèses définit par la règle
// précédente- suivi d'une virgule

          GetMatched(5).Name,

// Dans la définition de la méthode dans la classe « Diffusion », on ajoute les
// paramètres de sortie dont le type à été ajouté par la précédente règle avant la
// parenthèse fermante suivi d'une virgule

          GetMatched(5).Name,

//  à la suite du corps de la logique de cette méthode, ajoute un code pour faire
// les « Set » ou mise à jour des ports de sortie en fonction de celui en entrée en
//  métant ajour l'attribut « ctrlInfo ».

                GetMatched(5).Name.ctrlInfo = GetMatched(2).Name.ctrlInfo;
```

Tableau 5.24- Règle pour l'association des ports de contrôle avec le service de données « Decision ».

LHS	RHS
	

f)Règles relatives aux attâchements

Cette catégorie de règle est réservé au traitement des différents « attâchements » dans la classe « Configuration » qui matérialise les différents liens externes entre les composants et les connecteurs. Ces règles traitent les connexions en amont et en aval du connecteur pour les différents types de ports. Cette catégorie contient quatre règles qui se présentent comme suit :

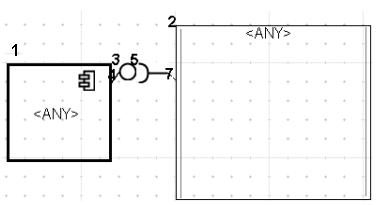
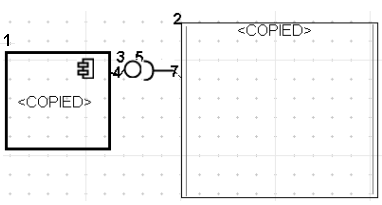
à traiter le concept de l'« Acteur » qui l'un des constituant structurel et sémantique du connecteur. Elles donnent une vue pratique de son vrai rôle dans le connecteur. Ces règles définissent le noyau de ce concept clé, sa surveillance des ports d'entrée et sa prise en charge des différents services du connecteur. Cette catégorie est constituée alors de trois règles qui se présentent comme suit :

Règle 29: Attâchement (ports de données) avec les composants amont (Tableau 5.25)

- *Nom* : *AttâchementDataPortIncomingComponent*
- *Priorité* : 29
- Rôle : Elle permet créer les attâchements de données entre les connecteurs et les composants en amont. Cette règle s'exécute autant de fois que de connexion de données entre le connecteur et le composant (généralement une fois).
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au pattern actuel.
- *Action* : Au début de la méthode (start()) de la classe configuration, elle ajoute la ligne de code suivante :

```
// Au de but de la méthode (start() ) de la classe configuration, on ajoute

 ((GetMatched(2).Name) GetMatched(2).Name). GetMatched(5).Name.dataInfo =
((GetMatched(1).Name) GetMatched(1).Name). GetMatched(3).Name.dataInfo;
```

Tableau 5.25- Règle pour l'attâchement des ports de données avec les composants en amont.

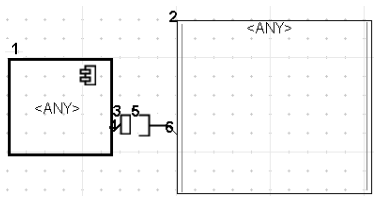
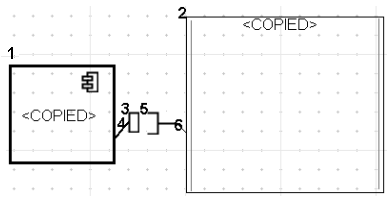
LHS	RHS
	

Règle 30 Attâchement (ports de contrôle) avec les composants amont (Tableau 5.26)

- *Nom* : *AttâchementCtrlPortIncomingComponent*
- *Priorité* : 30
- Rôle : Elle permet créer les attâchements de contrôle entre les connecteurs et les composants en amont. Cette règle s'exécute autant de fois que de connexion de données entre le connecteur et le composant (généralement une fois).
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au pattern actuel.
- *Action* : Au début de la méthode (start()) de la classe configuration, elle ajoute la ligne de code suivante :

```
// Au de but de la méthode (start() ) de la classe configuration, on ajoute

 ((GetMatched(2).Name) GetMatched(2).Name). GetMatched(5).Name.ctrlInfo =
((GetMatched(1).Name) GetMatched(1).Name). GetMatched(3).Name.ctrlInfo;
```

Tableau 5.26- Règle pour l'attâchement des ports de contrôle avec les composants en amont.

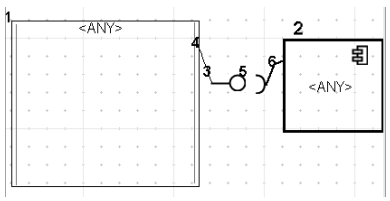
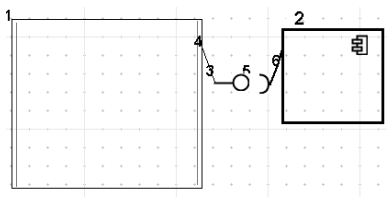
LHS	RHS
	

Règle 31: Attâchement (ports de données) avec les composants en aval (Tableau 5.27)

- *Nom* : *AttâchementDataPortOutgoingComponent*
- *Priorité* : 31
- Rôle : Elle permet créer les attâchements de données entre les connecteurs et les composants en aval. Cette règle s'exécute autant de fois que de connexion de données entre le connecteur et le composant (généralement une fois).
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au pattern actuel.
- *Action* : A la fin de la méthode (start()) de la classe configuration, elle ajoute la ligne de code suivante :

```
// A la fin de la méthode (start() ) de la classe configuration, on ajoute

 ((GetMatched(2).Name) GetMatched(2).Name). GetMatched(5).Name.dataInfo =
((GetMatched(1).Name) GetMatched(1).Name). GetMatched(3).Name.dataInfo;
```

Tableau 5.27- Règle pour l'attâchement des ports de données avec les composants en aval.

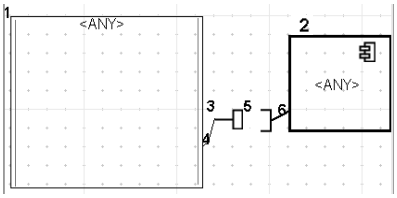
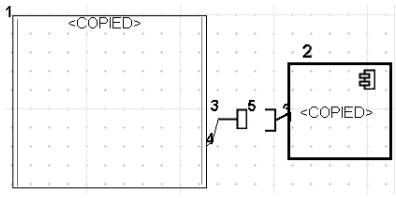
LHS	RHS
	

Règle 32 : Attâchement (ports de contrôle) avec les composants en aval (Tableau 5.28)

- *Nom* : *AttâchementCtrlPortOutgoingComponent*
- *Priorité* : 32
- Rôle : Elle permet créer les attâchements de contrôle entre les connecteurs et les composants en aval. Cette règle s'exécute autant de fois que de connexion de données entre le connecteur et le composant (généralement une fois).
- *Condition* : En plus du *matching* visuel, cette règle test l'attribut « *Visited* » associé au pattern actuel.
- *Action* : A la fin de la méthode (start()) de la classe configuration, elle ajoute la ligne de code suivante :

```
// A la fin de la méthode (start()) de la classe configuration, on ajoute

 ((GetMatched(2).Name) GetMatched(2).Name). GetMatched(5).Name.ctrlInfo =
((GetMatched(1).Name) GetMatched(1).Name). GetMatched(3).Name.ctrlInfo;
```

Tableau 5.28- Règle pour l'attâchement des ports de contrôle avec les composants en aval.

LHS	RHS
	

Dans la grammaire proposée, nous avons présenté la modélisation des services impliqués dans les études cas seulement. Le reste des services doit se traiter de la même façon en respectant juste leur cardinalité en entrée/sortie.

5.4 Etude de cas

Après avoir défini l'outil de modélisation visuelle ou l'ADL de description d'architecture pour la prise en charge des architectures logicielles à base de connecteur orienté processus, nous allons l'utiliser pour modéliser les deux exemples cités dans la section (3.5). L'objectif est de montrer que capADL accepte bel et bien ces modèles qui sont exprimés dans la même syntaxe concrète tout en restant conformes par rapport au méta-modèle SPEM proposé dans la même section.

A ce niveau nous considérons ces deux exemples d'une manière très pratique et expérimentale pour montrer les capacités de modélisation ou abstraction de notre méta-

modèle et l'ADL associé ainsi que la simplicité de générer du code Java pour chaque modèle en utilisant la même grammaire de graphe comme support de transformation de modèle du type (modèle vers texte).

Le choix de tels exemples est fait pour montrer les capacités de généricité et d'interopérabilité de notre ADL. Avec exactement les mêmes moyens, et sans changement aucun, capADL est capable de modéliser et de prendre en charge -en matière de génération de code- des modèles architecturaux issus de deux domaines initialement très distants sémantiquement. Le modèle « Pipe&Filter » est issu du monde des architectures logicielles alors que le modèle « V_Cycle_Life » est issu du domaine des processus logiciels.

Cette dernière remarque est assez encourageante pour étendre l'usage des concepts développés dans cette thèse pour être le moyen (connecteur CaP) et l'outil (capADL) pour l'abstraction de la communication dans d'autres paradigmes. On imagine, les domaines de services, d'IoT, d'agents, de systèmes cyber-physique etc.

5.4.1 Modèle Pipe&Filter

Le style architectural Pipes&Filter est utilisé (entre autres) pour diviser une tâche de traitement plus importante en une séquence de tâches plus petites. Dans notre cas, le connecteur est également un composant de communication. Cet exemple représente un concept largement admis dans la communauté architecture logicielle et va donc représenter une variante d'un tout autre domaine par rapport l'exemple « CycleV ». Ceci étant fait pour montrer les capacités de généricité de notre proposition.

5.4.1.1 Modélisation

Pour des contraintes d'espace et de longueur de code Java, nous modélisons une portion de l'exemple seulement. Ils sont impliqués les composants C1 et C3[1] ainsi que le connecteur de diffusion « dC ». La Figure 5.15 montre le résultat de la modélisation de l'exemple par notre capADL en syntaxe concrète respectant la sémantique du méta-modèle définit plus haut.

Les éléments du modèle sont instanciés à partir du méta-modèle. Il suffit d'appuyer sur l'un des boutons de la barre d'outils « capADL_MÉTA » pour avoir l'instance de l'objet en question en syntaxe concrète. Pour des raisons de lisibilité, les classes qui représentent les relations entre les entités ont été supprimées de la barre d'outils. Il suffit d'appuyer sur la touche « Ctrl » du clavier en mettant la souris sur l'objet de départ et de glisser jusqu'à l'objet de destination pour créer visuellement (flèche) la relation définie dans le méta-modèle entre ce type d'entités. La conformité du modèle instance par rapport au méta-modèle est assurée par l'ADL cap_ADL via les routines de l'outil AToM3.

[1] Dans la figure initiale (Chapitre 4) pour le modèle Pipe&Filter, nous avons nommé les composants par Pi. Dans cette section, nous les renommons Ci car les Pi sont des noms cachés réservés aux ports.

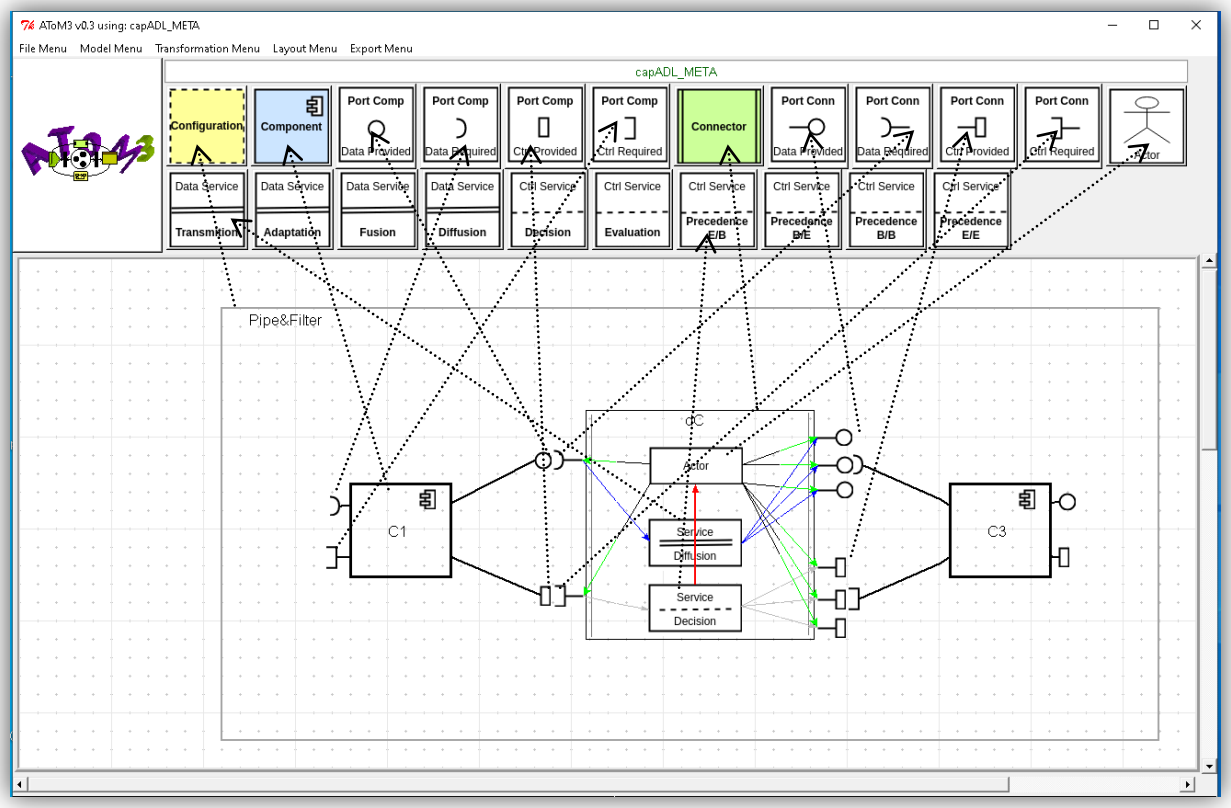


Figure 5.15- Modélisation Pipe&Filter par capADL.

5.4.1.2 Exécution de la transformation

En utilisant la version exécutable de la grammaire de graphe « capADL2Java.py » définit dans la section (5.3.2.2) on peut exécuter la transformation de modèle en utilisant l'approche de transformation de graphe. Comme le montre la Figure 5.16, la grammaire va prendre le modèle « Pipe&Filter » décrit dans le canevas principal de l'outil comme étant le graphe d'accueil sur lequel elle doit s'exécuter.

Doit-on le rappeler que nous sommes dans un cas de transformation sur place c'est-à-dire que le modèle cible est substitué sur le modèle source dans le même canevas. Or, pour la génération de code, il n'y a aucun changement apporté sur le graphe d'accueil comme le montre les différents patrons des règles où toutes les parties droites (RHS) sont identiques aux parties gauches (LHS). De cette façon, le graphe de départ est maintenu intact et identique après la transformation.

Les règles de la grammaire s'exécutent dans l'ordre de leur présentation en fonction de la valeur de priorité séquentielle qui leurs est accordée. Chaque règle s'exécute autant de fois que le nombre d'éléments architecturaux concernés dans le modèle source qu'elle traite. Pour chaque élément, la règle correspondante s'exécute une fois grâce à ses pré-conditions textuelles. Comme le modèle source est inchangée, la notion de « boucle d'exécution »[1] propre à AToM3 n'a pas lieu d'être.

L'application de la grammaire de graphe proposée « capADL2Java_GG_exe.py » sur le

[1] Contrairement à AGG qui a une exécution par niveaux des règles, AToM3 implémente la notion de « boucle » pour l'exécution des règles avec les priorités. Quand le modèle source change, une règle de priorité inférieure peut devenir applicable. C'est pour cette raison qu'AToM3 reconsidère toutes les règles avec les priorités inférieures à la priorité de la règle en cours à chaque itération.

modèle source « Pipe&Filer » de la Figure 5.18 génère dans un fichier texte le pseudo code Java correspondant présenté dans l'Annexe A.

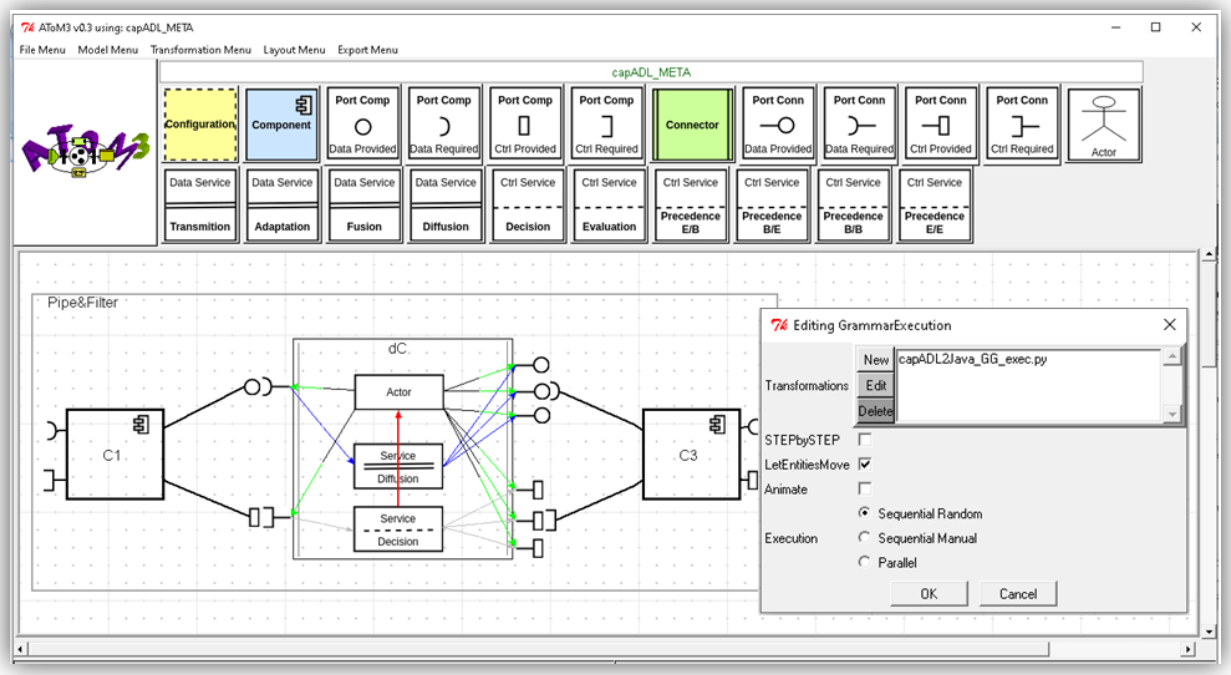


Figure 5.16- Éxécution de la transformation pour le modèle "CycleV".

5.4.2 Modèle Cycle en « V »

Le cycle de vie du logiciel est considéré comme un modèle descriptif qui décrit les étapes et séquences connues et reconnues pour la réalisation d'un produit logiciel. Le « cycle de vie en V » choisi est décrit comme un modèle en cascade dans lequel les étapes de développement et la mise en œuvre des tests sont effectuées de manière synchrone. Les étapes formeront les composants du processus logiciel (composant SP) tandis que les séquences seront modélisées par des connecteurs de processus logiciel (connecteur SP). Le modèle en V est assez bien connu dans la communauté du processus logiciel.

5.4.2.1 Modélisation

De la même manière que le premier cas, et pour des contraintes d'espace et de longueur de code Java, nous modélisons une portion de l'exemple seulement. Dans la configuration « V_Cycle_Life », les composants impliqués sont « Architectural_Design » et « Detailed_Design » ainsi que le connecteur de précédence « C1 ». La Figure 5.17 montre le résultat de la modélisation de l'exemple par notre capADL en syntaxe concrète respectant la sémantique du méta-modèle définit plus haut (Figure 5.9 et Figure 5.10).

Comme dans le premier cas, la manipulation est exactement la même et sans aucun changement. Les éléments du modèle sont instanciés à partir du méta-modèle. Il suffit d'appuyer sur l'un des boutons de la barre d'outils « capADL_MÉTA » pour avoir l'instance de l'objet en question en syntaxe concrète. Les classes qui représentent les relations entre les entités ne sont pas visibles dans la barre d'outils. Il suffit d'appuyer sur la touche « Ctrl » du clavier en mettant la souris sur l'objet de départ et de glisser jusqu'à l'objet de destination pour créer visuellement (flèche) la relation définie dans le méta-modèle entre ce type d'entités. La conformité du modèle instance par rapport au méta-modèle est assurée par l'ADL cap_ADL via les routines de l'outil AToM3.

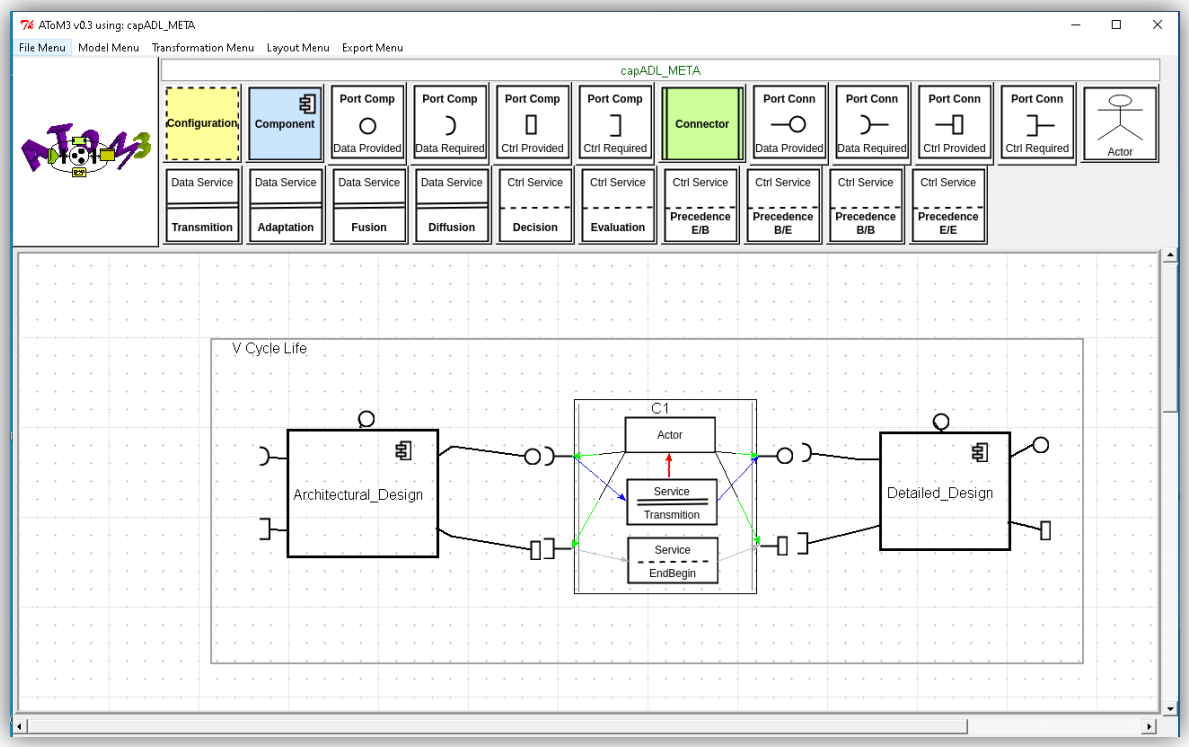


Figure 5.17- Modélisation cycle en « V » par capADL.

5.4.2.2 Exécution de la transformation

Comme dans le précédent exemple, en appliquant la même la grammaire de graphe proposée « capADL2Java_GG_exe.py » sur le modèle source « V_Cycle_Life » de la Figure 5.18 génère dans un fichier texte le pseudo code Java correspondant présenté dans l'Annexe B. On suppose dans ce cas que le connecteur est en mode automatique et non pas en mode manuelle.

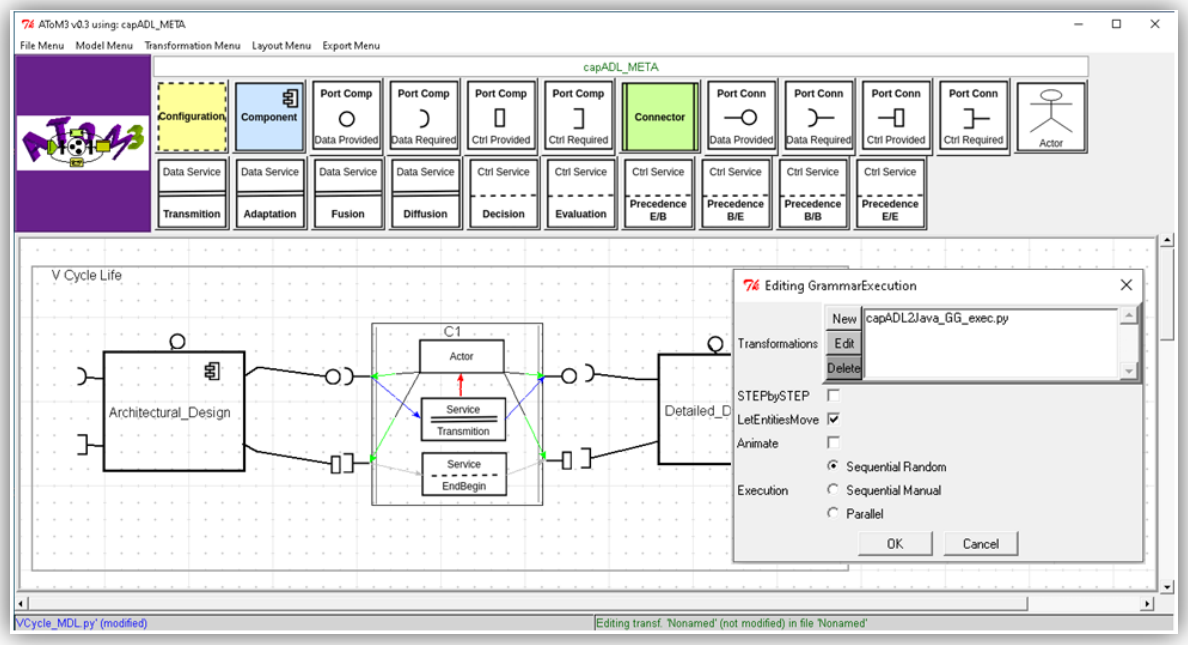


Figure 5.18- Éxécution de la transformation du modèle "Pipe&Filter".

5.5 Conclusion

Ce chapitre contient les éléments pratiques associés à notre proposition. Après présentation des différents aspects de l'outil de modélisation, AToM3, nous avons abordé la phase de méta-modélisation. Au terme de cette phase, nous avons pu générer notre environnement visuel ou notre langage de description d'architectures logicielles (capADL) à base de connecteur orienté processus logiciel (CaP).

La seconde phase nous l'avons consacré à la transformation de modèle et aux techniques de transformations de graphes appliquées dans le cadre de cette thèse pour produire un code Java à partir de modèles architecturaux. Pour cela nous avons proposé une grammaire de graphe formée d'un module de 32 règles graphiques capable de transformer aisément toute architecture modélisée avec capADL en son squelette en code Java.

En guise d'étude de cas, nous avons utilisé deux modèles architecturaux connus dans des paradigmes très distincts à savoir les architectures logicielles et les processus logiciels pour montrer la faisabilité et la puissance de notre contribution. Nous avons utilisé capADL pour modéliser un modèle architectural Pipe&Filter et le modèle de cycle de vie en « V ». Nous avons ensuite appliqué la grammaire de graphe proposée pour générer le code Java de ces deux exemples.

Conclusion générale

Cette thèse avait pour but initial de réfléchir et de proposer un modèle conceptuel de communication capable d'abstraire la communication dans les architectures logicielles. Ce modèle devait être assez générique pour traverser les paradigmes de développements afin de remédier essentiellement au problème d'interopérabilité. Aussi, il était question de consacrer une thèse destinée exclusivement à la communication étant donné que beaucoup d'autres études ont été réservées aux autres aspects des architectures logicielles.

Nous avons abordé ce sujet en nous consacrant à l'étude des modèles de communications dans les paradigmes de développement usuels. Nous avions l'intention de proposer un modèle unificateur à la fin du processus. Or, le niveau de détails ainsi que la disparité des distances sémantiques entre les paradigmes ont rendu cette démarche peu fructueuse. Nous avons donc décidé de procéder par l'inverse, soit de partir d'un modèle de communication bien établi et d'essayer de l'enrichir afin de servir et valoir de moyen de communication inter-paradigmes.

Le paradigme orienté composant est le seul qui propose une abstraction de la communication à travers le concept de connecteur. Les travaux de recherche ont fait évoluer ce concept pour atteindre présentement le statut d'un élément architectural explicite doté d'une maturité avérée particulièrement dans son aspect conceptuel. Aussi, pour des raisons de granularité et de complétude de représentation, les architectures logicielles sont souvent associées et confondues avec le paradigme orienté composant.

Partant de cela, nous avons été amenés à étudier un bon nombre d'approches qui s'intéressent au connecteur comme un citoyen de première classe. Nous avons constaté leurs atouts et leurs insuffisances surtout dans leurs prises en charge des interactions complexes. La forte interactivité, technologiques et la disparité des environnements de distributions ont fait que les interactions deviennent des processus complexés pour être supportées par un connecteur classique et usuel.

Nous avons donc eu besoin de proposer un modèle conceptuel de communication qui soit en mesure de prendre en charge les interactions complexes, d'absorber l'hétérogénéité et de promouvoir l'interopérabilité. Aussi, ce modèle doit être réutilisable et assez générique pour pouvoir être généralisé à travers les paradigmes de développement.

La solution préconisée consiste à adopter et à étendre le modèle de communication dans des architectures à base de composants en utilisant des connecteurs comme entité de première classe. Pour cela, nous avons eu l'intuition de modéliser et d'étendre donc le connecteur par les moyens de processus logiciel qui reste ouvert et agile. Nous avons intégré les éléments de processus logiciel au sein même du connecteur pour former sa nouvelle structure et sa riche sémantique. Ainsi formé, le connecteur obtient le statut de composant de communication offrant des services de communication particuliers à travers son interface.

Ce choix est motivé par le fait que la théorie de la communication et les processus ont des similarités structurelles et comportementales, des sémantiques proches et partagent l'essentiel des éléments (Quoi, Comment et Qui). Les notions de « produit », d' « activité » et de « rôle » existent avec des façons très proches dans les deux concepts er représente leur dénominateur commun.

Tel qu'un composant, nous avons proposé un connecteur « for/by reuse », sur étagère et une bibliothèque pour les connecteurs de contrôle et de données pour augmenter sa généricité et sa généralisation. Le connecteur (CaP pour *Connector as Process*) proposé est en mesure de servir les différentes phases de son cycle de vie. L'approche globale proposée préserve le

statut d'entité de première classe du connecteur (CaP) depuis la conception à la mise en œuvre. Ceci est fait en s'appuyant sur les mécanismes méta-modélisation et des transformations de modèles lors des différentes phases.

Pour ce faire, nous considérons le connecteur dans ses définitions de base. En effet, le connecteur n'est autre qu'un transfert de données, un transfert de contrôle ou les deux à la fois le long d'un canal. Cela nous permet d'identifier un ensemble de connecteurs à partir de ces blocs de base identifiés à la fois dans les communautés d'architecture logicielle et de processus logiciel. Ces concepts ont été détaillés en proposant les méta-modèles associés.

Aussi, nous avons proposé les différents stéréotypes du profil SPEM pour lier les concepts de processus logiciel et concept d'architecture logiciel pour prendre en charge notre connecteur (CaP) assortie d'une phase de validation qui montre comment surmonter les problèmes d'interopérabilité. Une validation visuelle est proposée sur deux exemples très distincts sémantiquement.

Nous avons également proposé un langage de description d'architecture (capADL) pour prendre en charge la nouvelle charge sémantique de notre approche en utilisant l'Outil de modélisation AToM3 et le langage Python pour l'expression des contraintes. En utilisant capADL, nous avons pu instancier les modèles architecturaux pour le patron « Pipe&Filter » et le modèle de cycle de vie «V».

En utilisant le même outil, nous avons pu faire une transformation automatique de ces modèles pour obtenir le pseudo code Java des deux modèles. Cette transformation de modèles est réalisée par une approche une transformation de graphes. Une grammaire de réécriture de graphes de 32 règles a été proposée en utilisant l'outil AToM3. La grammaire a été validée sur les mêmes exemples.

Le choix des deux exemples de validation, soit le « Pipe&Filter » et le « Cycle_V » qui n'ont aucun rapport sémantique entre eux, montre la force de notre contribution. On peut dire que le connecteur (CaP) est assez générique pour modéliser et prendre en charge des domaines ou paradigmes très distants sémantiquement qui sont les architectures logicielles à base de composants et les processus logiciels. Le reste des paradigmes est certainement compris entre ces deux bornes. En outre, la décomposition du flux en données et contrôle permet d'identifier et de gérer la complexité ainsi que la réutilisation et la généricité pour la communication multi-paradigme.

Au-delà des contours de la solution proposée et de ses éléments, nous avons aussi fait un grand effort pour cerner l'état du domaine avec toute sa disparité. Ainsi, nous avons étudié les architectures et les connecteurs logiciels, les processus logiciels ainsi que la méta-modélisation et l'ingénierie des modèles. Des comparatifs et synthèses ont été à chaque fois élaborés. Cette étude est faite de sorte que toute notion évoquée dans les chapitres de la solution proposée soit évoquée au préalable dans cette partie.

Durant cette thèse, les conditions qui n'ont pas été souvent favorables, le déroulement de cette a été plein de difficultés le long de ces années. Chaque étape avait ses principaux défis, ses échecs et ses réussites. Une fois passées, on garde souvent quelques bons souvenirs des situations cauchemardesques auxquelles nous avons fait face. Toutefois, le nombre important de concepts manipulés dans le cadre de cette thèse et les efforts consentis nous ont procuré un apprentissage et un enseignement certain. Dans le monde de la recherche, l'effort paie toujours.

Synthèse

Selon les critères discutés dans la section (4.4.5), nous pouvons établir la discussion suivante pour notre connecteur proposé par rapport à d'autres approches :

Abstraction : la notion de "Processus" semble être le niveau d'abstraction le plus élevé et le plus adéquat pour la communication. Il reflète fidèlement sa réalité. En ajoutant des éléments de processus tels que « Acteur », « Activité » et « Produit », nous représentons les parties essentielles impliquées dans un processus de communication avec leur sémantique opérationnelle.

Séparation des préoccupations : Le connecteur proposé représente un composant indépendant de l'application dédié à la communication entre les composants fonctionnels impliqués dans un système en cours de développement. Un architecte peut se concentrer sur un service de communication spécifique et typé.

Réutilisation et adaptation : La décomposition fine et élémentaire de nos classes de connecteurs nous permet d'avoir une bibliothèque plus riche ainsi qu'un meilleur potentiel pour composer des connecteurs plus complexes. La démarche est une proposition de connecteur usage/par usage. C'est-à-dire que par une sélection dans la bibliothèque de connecteurs, un architecte peut choisir le connecteur approprié pour relier deux ou plusieurs composants en fonction d'une situation réelle. L'attâchement se fait avec une simple correspondance des ports de type compatible.

Potentiel de granularité : Par rapport à d'autres approches, la nôtre semble avoir le potentiel de granularité le plus élevé puisqu'elle fait abstraction de l'ensemble du processus de communication dans un processus logiciel systématique avec tous ses éléments (Acteur, Activité et Produit).

Dynamique : En tant que processus avec ses éléments, le connecteur donnera plus d'agilité et d'ouverture de la phase d'architecture à celle de mise en œuvre. Les services de communication, exportés par l'interface du connecteur, aident à déterminer l'activité ou le protocole de communication requis.

Évolution : En matérialisant chaque élément de notre connecteur avec le profil SPEM, l'évolution pour de nouvelles exigences ou paradigmes peut se faire par simple ajout ; modifier ou supprimer le stéréotype correspondant dans le package "Method Plugin".

Catégorisation : En plus des catégories définies dans Mehta et al. (2000), nous proposons deux familles de connecteurs concernés par la manipulation ou le contrôle de données. Ces familles sont organisées dans une bibliothèque de connecteurs dédiée.

Préservation du cycle de vie : comme indiqué avec le code Java dans l'exemple 1 et 2, il existe une représentation stricte des éléments décrits au niveau conceptuel avec les méta-modèles ci-dessus. Le connecteur reste une entité indépendante dans sa mise en œuvre préservant son statut de composant de communication avec sa propre interface formée de ports et de services.

Il en ressort que notre approche remplit l'ensemble des critères avec un avantage encore plus important, celui d'avoir doté notre approche d'un langage de description d'architecture capable de décrire des architectures logicielles à base du connecteur orienté processus et d'en générer le code Java associé.

Aussi, par rapport aux approches similaires discutées dans le cadre du chapitre (4), on peut réitérer l'ensemble des ressemblances et des distinctions qui les lient et qui les séparent de l'approche que nous proposons. Cette synthèse peut se résumer comme suit :

- Approches formelles : force est de constater que nous n'avons pas encore des assises formelles pour la description des protocoles de communication par des modèles formels. Ceci est certainement dû au fait que nous nous sommes attardés d'abord à proposer le connecteur (CaP) à un niveau conceptuel comme composant de communication réutilisable dans son intégralité en précisant, une structure et toute nouvelle sémantique conceptuelle. Par conséquent, nous étions préoccupés à cerner le nouveau concept et de définir ses contours, ses atouts et ses limites plutôt qu'à le surcharger à ce stade par les translations vers les modèles formels. Toutefois, nous avons reporté cette orientation pour des travaux futurs. Nous utiliserons exactement la même démarche entreprise pour la génération de code que nous avons réalisé.
- Approche de cycle de vie : ces approches proposent un processus vertical de développement des connecteurs, de la spécification à l'implémentation et le déploiement. Elles sont, en quelque sorte, orthogonales à notre approche puisque nous cherchons à proposer un modèle de connecteursous forme d'un processus de communication sur un même niveau d'abstraction pour un ensemble distinct de paradigmes de développement. Ceci dit, nous partageons le principal objectif de ces approches en termes de préservation du connecteur comme entité de première classe le long du cycle de sa vie.
- Approche structurelles : ces approches considèrent certes le connecteur comme une entité de première classe mais focalisent en revanche sur sa partie structurelle uniquement contrairement à notre approche qui aspire à accorder au connecteur un statut de communication réutilisable ayant une structure et un comportement. Aussi, certaines de ces approches ne sont pas outillées.
- Approche d'adaptation : ces approches proposent des connecteurs et des démarches pourprendre en charge des problèmes spécifiques et par conséquent seront sous utilisés par rapport au volume de leurs architectures internes quand les problèmes posés sont usuellement simples. Aussi, à cause de leur caractère spécifique et dédié, on assiste à un déclin en termes de réutilisation pourtant propriétés phare du paradigme composant. Cependant, ces approches sont les plus semblables à la nôtre par rapport à leur charge sémantique. Le méta-modèle MMSA évoque le concept de processus sous forme d'une seule classe muette sans rien de plus.

En guise de synthèse générale pour les connecteurs vus comme entité de première classe, il en ressort que les normes et standards sont peu expressifs vis-à-vis des connecteurs et ce, sûrement, par soucis de consensus pour leur industrialisation par les outils ou autres ADLs. Hormis les approches de classification ou de spécification formelles qui définissent les assises théoriques des connecteurs, le reste des approches proposent soit une revue des processus de développement, des structures et des hiérarchies ainsi que la spécialisation pourdes cas spécifiques.

D'aucune de ces approches ne considère la partie générique et comportemental sous forme de processus de communication visible à travers les interfaces des connecteurs pour justement gagner en efficacité et réutilisabilité comme le prétend

remplir notre approche. Elle est la seule à même de proposer un connecteur avec un statut égal à un composant offrant des services de communication spécifiques. L'architecture interne de notre connecteur est basée sur les modèles de processus logiciels pouvant absorber les communications complexes et favoriser l'interopérabilité.

Bien que hors contexte, il est tout de même utile de présenter un bref comparatif (Figure 5.19) entre les différents travaux de thèses de l'école du Pr Oussalah des architectures logicielles au sein du laboratoire LS2N (e LINA). Ces approches ont fait un effort pour parcourir les différents niveaux de méta-modélisation des standards OMG initialement destinés au paradigme objet.

Dans sa thèse rarissime, Adel Smeda (2006) a réussi à faire un glissement de ces standards dans le paradigme à base de composants. Avec des mécanismes de mapping à partir du MOF, il a proposé un méta-méta-modèle baptisé (MADL) pour la définition des langages de description d'architectures logicielles à base de composants. L'auteur propose ultérieurement un ADL COSA pour la prise en charge des architectures logicielles basées sur les structures composites.

En s'appuyant sur MADL, Abdelkrim Amirat (2010) s'intéresse aux architectures logicielles hiérarchiques. Il propose le méta-modèle C3 (au niveau M3) et une bibliothèque de connecteurs réutilisables (Niveau M2). Ces connecteurs peuvent assurer différentes formes d'hiérarchie, les contraintes sont formulées en OCL. Pour cela, il apporte une variation structurelle importante en intégrant les attachements au sein du connecteur.

Dans un contexte légèrement différent, Adel Hassen (2018) consacre sa thèse à la discussion des styles d'évolutions en vue de leur réutilisation dans les architectures logicielles à base de composants. A travers les quatre niveaux de modélisation, il propose un méta-modèle pour l'évolution logicielle et un méta-méta-modèles qui précise la définition des langages d'évolution qu'il dénomme MES. MES (pour *Méta-Style evolution*) est basé sur MADL (*Méta Architecture Description Language*).

Dans notre approche (Menasria, 2022), nous proposons un modèle conceptuel de la communication dans les architectures logicielles à base de composants. Le connecteur basé processus (CaP) proposé fusionne les concepts d'architectures logicielles et de processus logiciels en utilisant le profil SPEM. L'ADL capADL qui prend en charge la nouvelle charge sémantique et proposé. Il utilise les concepts de SPEM et reste conforme méta-formalisme d'AToM3. La force de l'outil est de servir les quatre niveaux de méta-modélisation.

Le reste des approches se sont consacrées exclusivement à la description des produits logiciels à une exception près pour l'approche de Hassen. Notre approche en revanche est destinée à la fois au produit et au processus. Seuls Smeda et Hassen parcourent les quatre niveaux de méta-modélisation. Au même titre que notre approche, celle d'Amirat utilise trois niveaux seulement. Elles sont fortes à réutiliser les concepts de MADL et de méta-formalisme de l'outil AToM3 qui reste conforme au MOF.

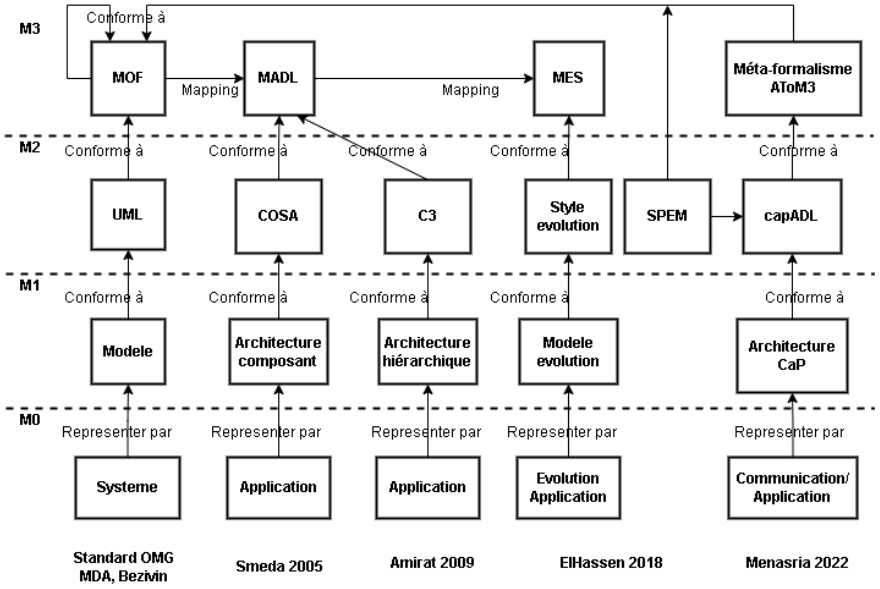


Figure 5.19- Quelques approches de l'Ecole Oussalah d'architectures logicielles.

Travaux futures

Enfin, nous considérons que ce travail a atteint les fins attendues. En fonction de l'expérience acquise, nous avons l'intention d'explorer des questions intéressantes dans le futur pour donner plus d'envergures et d'horizons à notre contribution. Parmi les futures ambitions on peut citer :

- étendre le connecteur défini ici comme étant le moyen de communication entre les paradigmes. Nous voulons proposer l'extension de stéréotypes nécessaires à notre méta-modèle pour servir et valoir comme moyen de communication pour les architectures multi-agents ainsi que les architectures orientées services, ou encore l'Internet des objets et les systèmes cyber-physiques. Ce faisable, en précisant seulement les protocoles associés. Ces paradigmes sont surement compris entre le paradigme composant et le celui des processus logiciels.
- doter les protocoles de communication de spécification formelle en CSP ou en tout autre modèle formel pour pouvoir assurer les différentes analyses possibles.
- proposer une composition des connecteurs pour prendre en charge des communications encore plus complexes.
- reprendre le contenu de la génération de code pour détailler davantage la bibliothèque des connecteurs.
- donner la main à l'architecte pour préciser son propre protocole de communication et de traitement et ce principalement pour la simulation des tâches manuelles.
- proposer des analyses et des vérifications d'architecture (ADL) à des stades précoces en proposant un passage vers des modèles formels (automate, réseau de Petri, Algèbre de processus….).
- de la même manière que la génération de code, nous pouvons également générer des protocoles de communication pour d'autres paradigmes pour faire de notre connecteur le moyen de communication dans les différents paradigmes comme discuté dans la section (1.6.1).

- il est aussi utile de reconsidérer toute la partie pratique relative au capADL avec les outils successeurs d'AToM3 à l'instar de l'étude comparative (Molesini et al., 2021) à laquelle j'ai eu accès tardivement dans le déroulement de cette thèse. Dans sa quête des approches de modélisation multi-paradigmes (Carreira et al., 2020), cette étude compare les trois outils issus du même laboratoire: AToM3 (de Lara & Vangheluwe, 2002), AToMPM (Syriani et al., 2013) et Modelverse (Van Tendeloo et al., 2019; Van Tendeloo & Vangheluwe, 2017).
- il faut aussi penser à prévoir les mécanismes et les styles d'évolution pour le connecteur CaP.

Bibliographie

Abboud, M., Oussalah, M., Naja, H., & Dbouk, M. (2016). SArEM : A SPEM extension for software architecture extraction process. *International Journal on Computer Science and Engineering*, *8*(4), 152--159.

Accord, P. (2002). Assemblage de composants par contrats en environnement ouvert et réparti. *INRIA*, *6*, 120.

ACUÑA, S. T., & FERRÉ, X. (2005). Software Process Modelling. *Software Process Modeling*, *4200*, 111–139.

Adel, A., & Abdullah, B. (2015). A Comparison Between Three SDLC Models Waterfall Model, Spiral Model, and Incremental/Iterative Model. *IJCSI International Journal of Computer Science Issues*, *12*(1), 106–111. https://www.academia.edu/10793943/A_Comparison_Between_Three_SDLC_Models_Waterfall_Model_Spiral_Model_and_Incremental_Iterative_Model

Ahmed, S. H., Kim, G., & Kim, D. (2013). Cyber Physical System: Architecture, applications and research challenges. *2013 IFIP Wireless Days (WD)*, 1–5.

Alajrami, S., Gallina, B., & Romanovsky, A. (2016). EXE-SPEM: towards cloud-based executable software process models. *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 517–526.

Allen, R., & Garlan, D. (1994). *Formal connectors*. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

Allen, R., & Garlan, D. (1997). A formal basis for architectural connexion. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(3), 213–249.

Alti, A., Gasmi, M., & Roose, P. (2015). Security architecture métamodel for Model Driven security. *Journal of Innovation in Digital Ecosystems*.

Amirat, A, on, M. O.-I. C., & 2009, undefined. (2009). Towards an UML profile for the description of software architecture. *Hal.Archives-Ouvertes.Fr*, 226–232. https://hal.archives-ouvertes.fr/hal-00483680/

Amirat, Abdelkrim. (2010). *Contribution à l'élaboration d'architectures logicielles à hiérarchies multiples*. Université de Nantes.

Amirat, Abdelkrim, Hock-Koon, A., & Oussalah, M. (2014). *Paradigmes objet, composant, agent et service dans les architectures logicielles*. Lavoisier.

Amirat, Abdelkrim, Menasria, A., Oubelli, M. A., & Younsi, N. (2012). Automatic generation of PROMELA code from sequence diagram with imbricate combined fragments. *Second International Conference on the Innovative Computing Technology (INTECH 2012)*, 111–116.

Amirat, Abdelkrim, & Oussalah, M. (2009). First-class connectors to support systematic construction of hierarchical software architecture. *The Journal of Object Technology*, *8*(7), 107–130.

Amirat, Abedelkrim, & Menasria, A. (2016). Visual decomposition of UML 2.0 interactions. *Technol., A Menasria - Int. Arab J. Inf.* http://ccis2k.org/iajit/PDF/Vol.13, No.3/7430.pdf

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A., & Taentzer, G. (1999). Graph transformation for specification and programming. *Science of Computer Programming*, *34*(1), 1–54.

Aoussat, F. (2012). *Réutilisation des procédés logiciels : Une approche à base d ' architectures logicielles*.

Aoussat, F., Oussalah, M., & Nacer, M. A. (2014). A spemontology for software processes reusing. *Computing and Informatics*, *33*(1), 35–60.

Arbab, F. (2004). Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, *14*(3), 329–366.

Auguston, M. (2009). Monterey phoenix, or how to make software architecture executable. *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, 1031–1040.

Awodey, S. (2010). *Category theory*. Oxford university press.

Baier, C., & Katoen, J.-P. (2008). Principles Of Model Checking. In *MIT Press* (Vol. 950). MIT press. http://mitpress.mit.edu/books/principles-model-checking

Bakshi, K. (2017). Microservices-based software architecture and approaches. *2017 IEEE Aerospace Conference*, 1–8.

Barbosa, E. A., Batista, T., Garcia, A., & Silva, E. (2011). Pl-aspectualacme: an aspect-oriented architectural description language for software product lines. *European Conference on Software Architecture*, 139–146.

Bass, L., Clements, P., & Kazman, R. (2003). Software Architecture in Practice , Second Edition. In *Software Architecture*. Addison-Wesley Professional.

Bendraou, R., Combemale, B., Crégut, X., & Gervais, M.-P. (2007). Definition of an Executable SPEM 2.0. *14th Asia-Pacific Software Engineering Conference (APSEC'07)*, 390–397.

Bendraou, R., Gervais, M.-P., Blanc, X., & Jézéquel, J.-M. (2008). Vers l'Exécutabilité des Modèles de Procédés Logiciels. *BT - Langages et Modèles à Objets, LMO 2008, Montréal, Québec, Canada*, 153–168. http://editions-rnti.fr/?inprocid=1000610

Bézivin, J., & Briot, J.-P. (2004). Sur les principes de base de l'ingénierie des modèles. *Obj. Logiciel Base Données Réseaux*, *10*(4), 145–157.

Bézivin, J., & Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework. *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 273–280.

Bhuta, J., Boehm, B., & Meyers, S. (2005). Process Éléments: Components of Software Process. *International Software Process Workshop 2005*, 332–346. https://doi.org/10.1007/11754305_9

Blanc, X., & Salvatori, O. (2011). *MDA en action: Ingénierie logicielle guidée par les modèles*. Editions Eyrolles.

Blom, H., Chen, D.-J., Kaijser, H., Lönn, H., Papadopoulos, Y., Reiser, M.-O., Kolagari, R. T., & Tucci, S. (2016). East-ADL: An architecture description language for automotive software-intensive systems in the light of recent use and research. *International Journal of System Dynamics Applications (IJSDA)*, *5*(3), 1–20.

Boehm, B. W. (1995). Software process architectures. *Proceedings of the First International Workshop on Architectures for Software Systems*.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Connallen, J., & Houston, K. A. (2008). Object-oriented analysis and design with applications. *ACM SIGSOFT Software*

Engineering Notes, *33*(5), 29.

Bouguettaya, A., Singh, M., Huhns, M., Sheng, Q. Z., Dong, H., Yu, Q., Neiat, A. G., Mistry, S., Benatallah, B., Medjahed, B., Ouzzani, M., Casati, F., Liu, X., Wang, H., Georgakopoulos, D., Chen, L., Nepal, S., Malik, Z., Erradi, A., … Papazoglou, M. (2017). A service computing manifesto: The next 10 years. *Communications of the ACM*, *60*(4), 64–72. https://doi.org/10.1145/2983528

Boussaïd, I., Siarry, P., & Ahmed-Nacer, M. (2017). A survey on search-based model-driven engineering. *Automated Software Engineering*, *24*(2), 233–294.

Brian Randell, B. (2018). *Fifty Years of Software Engineering - or - The View from Garmisch*. *19*(3), 74–76. http://arxiv.org/abs/1805.02742

Bruel, J. M., Combemale, B., Guerra, E., Jézéquel, J. M., Kienzle, J., de Lara, J., Mussbacher, G., Syriani, E., & Vangheluwe, H. (2020). Comparing and classifying model transformation reuse approaches across métamodels. *Software and Systems Modeling*, *19*(2), 441–465. https://doi.org/10.1007/S10270-019-00762-9

Buschmann, F., Schmidt, D., Stal, M., & Rohnert, H. (1996). *Pattern-Oriented Software Architecture Volume 1:... - Google Scholar* (U. John Wiley and Sons Chichester (ed.); John Wiley). John Wiley and Sons Chichester, UK. https://scholar.google.com/scholar?q=Pattern-Oriented+Software+Architecture+Volume+1%3A+A+System+of+Patterns.+Wiley.&hl=fr&as_sdt=0%2C5&as_ylo=1996&as_yhi=1996

Cariou, E. (2003). *Contribution à un processus de réification d'abstractions de communications*. These de doctorat, Université de Rennes 1.

Carreira, P., Amaral, V., & Vangheluwe, H. (2020). Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems. In *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*. https://doi.org/10.1007/978-3-030-43946-0

Cavalcante, E., Medeiros, A. L., & Batista, T. (2013). Describing cloud applications architectures. *European Conference on Software Architecture*, 320–323.

Chen, L., Huang, L., Zhong, H., Li, C., & Wu, X. (2015). Breeze: A modeling tool for designing, analyzing, and improving software architecture. *2015 IEEE 23rd International Requirements Engineering Conference, RE 2015 - Proceedings*, 284–285. https://doi.org/10.1109/RE.2015.7320440

Chu, S. C. (2005). From component-based to service oriented software architecture for healthcare. *Proceedings of 7th International Workshop on Enterprise Networking and Computing in Healthcare Industry, 2005. HEALTHCOM 2005.*, 96–100.

Clement, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2011). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional.

Combemale, Benoît, & Crégut, X. (2006). Towards a rigorous process modeling with spem. *ICEIS (3)*, 530–533.

Combemale, Benoit, Crégut, X., Caplain, A., & Coulette, B. (2006). Towards a rigorous use of SPEM. *Eighth International Conference on Enterprise Information Systems*, 530–533.

Cruz, P., Gómez-Álvarez, M. C., & Astudillo, H. (2020). Mapping SETMAT practices to SPEM. *2020 39th International Conference of the Chilean Computer Science Society (SCCC)*, 1–8.

Curtis, B., Kellner, M. I., & Over, J. (1992). Process modeling. *Communications of the ACM*, *35*(9), 75–90.

Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, *45*(3), 621–645. https://doi.org/10.1147/sj.453.0621

Dai, F., Li, T., Zhao, N., Yu, Y., & Huang, B. (2008). Evolution process component composition based on process architecture. *2008 International Symposium on Intelligent Information Technology Application Workshops*, 1097–1100.

Dami, S., Estubler, J., & Amiour, M. (1998). Apel: A graphical yet executable formalism for process modeling. In *Process Technology* (pp. 61–96). Springer.

Dashbalbar, A., Song, S.-M., Lee, J.-W., & Lee, B. (2017a). Towards enacting a spem-based test process with maturity levels. *KSII Transactions on Internet and Information Systems (TIIS)*, *11*(2), 1217–1233.

Dashbalbar, A., Song, S. M., Lee, J. W., & Lee, B. (2017b). Towards enacting a SPEM-based test process with maturity levels. *KSII Transactions on Internet and Information Systems*, *11*(2), 1217–1233. https://doi.org/10.3837/tiis.2017.02.034

de Lara, J., & Vangheluwe, H. (2002). Atom3: A tool for multi-formalism and méta-modelling. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *2306*, 174–188. https://doi.org/10.1007/3-540-45923-5_12

de Lara, J., & Vangheluwe, H. (2004). Defining visual notations and their manipulation through méta-modelling and graph transformation. *Journal of Visual Languages and Computing*, *15*(3–4), 309--330. Elsevier.

De Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, *85*(1), 132–151.

Derdour, M., Dalmau, M., Roose, P., & Ghoualmi-Zine, N. (2010). Typing of adaptation connectors in MMSA approach case study: sending MMS. *International Journal of Research and Reviews in Computer Science*, *1*(4), 39–49.

Derdour, M., Roose, P., Dalmau, M., & Ghoualmi-Zine, N. (2012). An adaptation platform for multimedia applications CSC (component, service, connector). *Journal of Systems and Information Technology*.

Derdour, M., Roose, P., Dalmau, M., Ghoualmi Zine, N., & Alti, A. (2010). MMSA: métamodel multimedia software architecture. *Advances in Multimedia*, *2010*, 1–17.

Di Francesco, P. (2017). Architecting microservices. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 224–229.

Diaw, S. (2012). *Spem4mde : un métamodèle et un environnement pour la modélisation et la mise en œuvre assistée de processus Samba Diaw To cite this version : HAL Id : tel-00668956*. Thèse de doctorat, Université de Toulouse. France.

Diaw, S., Cisse, M. L., & Bah, A. (2017). Using the SPEM 2.0 kind-based extension mechanism to define the SPEM4MDE métamodel. *ACM International Conference Proceeding Series*, *Part F1306*, 63–69. https://doi.org/10.1145/3129186.3129199

Díaz, E., Panach, J. I., Rueda, S., & Vanderdonckt, J. (2021). An empirical study of rules for mapping BPMN models to graphical user interfaces. *Multimedia Tools and Applications*, *80*(7), 9813–9848.

Dijkstra, E. (1968). *Notes on structured programming*. http://dea.unsj.edu.ar/informatica1/recursos/Apuntes/Unidad6/2-NotesOnStructuredProgramming-Dijkstra.PDF

Diver, P. (2010). *Les secrets du dessinateur AutoCAD*. Pearson Education France.

Djibo, K., Oussalah, M., & Konaté, J. (2020). Evolution Style Mining in Software Architecture. *ENASE*, 313–322.

Dorri, A., Kanhere, S. S., & Jurdak, R. (2018). Multi-agent systems: A survey. *Ieee Access*, *6*, 28573–28593.

Ducasse, S., & Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, *35*(4), 573–591.

Duncan, S. (2003). Component software: Beyond object-oriented programming. *Software Quality Professional*, *5*(4), 42.

Ehrig, H, Ermel, C., Golas, U., … F. H. T. C. S., & 2015, U. (2015). Graph and Model Transformation. *Springer*. https://link.springer.com/content/pdf/10.1007/978-3-662-47980-3.pdf

Ehrig, H, Rozenberg, G., & Kreowski, H. rg. (1999). *Handbook of graph grammars and computing by graph transformation*. https://books.google.com/books?hl=fr&lr=&id=nwD_oRtfJKkC&oi=fnd&pg=PA2&dq=book+ehrig+computing+by+graph+transformation&ots=zxjQuMncds&sig=YhCTPplfNrBUpXUjVlMMjWP2R-0

Ehrig, Hartmut, Engels, G., Parisi-Presicce, F., & Rozenberg, G. (2008). *Graph transformations*. Springer.

Ehrig, Hartmut, Padberg, J., Braatz, B., Klein, M., Orejas, F., Pérez, S., & Pino, E. (2004). A generic framework for connector architectures based on components and transformations. *Electronic Notes in Theoretical Computer Science*, *108*, 53–67.

Erata, F., Challenger, M., & Kardas, G. (2015). Review of Model-to-Model Transformation Approaches and Technologies. *Itea 3 (Eureka Strategic ICT Programme)*, 1–20.

Farahani, S. (2011). *ZigBee wireless networks and transceivers*. newnes. Elsevier.

Feiler, P. H., Lewis, B., & Vestal, S. (2003). *The SAE avionics architecture description language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering*.

Finkelstein, A., Kramer, J., & Hales, M. (2019). Process Modelling: A Critical Analysis. *Integrated Software Reuse: Management and Techniques*, *Wile*, 141–152. https://doi.org/10.4324/9780429455520-11

Fonseca, I. O., Fritsch, L. G., Bernardino, M., & Basso, F. (2021). *BPMN , SPEM e Essence no contexto da Modelagem de Processos de Software : uma Revisão Sistemática da Literatura*.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Fuggeffa, A. (2000). Software Process: A Roadmap. *International Journal of Project Management*, *19*(4), 75–90.

Fujita, H., & Herrera-Viedma, E. (2018). Architectural languages' connector support for modeling various component interactions: A review. *New Trends in Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 17th International Conference*

SoMeT_18, *303*, 474.

GALLARDO, L. (2000). *Une approche à base de composants pour la modélisation des processus logiciels*. Université de grenoble -France.

Gallina, B., Pitchai, K. R., & Lundqvist, K. (2014). S-TunExSPEM: Towards an extension of SPEM 2.0 to model and exchange tunable safety-oriented processes. *Studies in Computational Intelligence*, *496*, 215–230. https://doi.org/10.1007/978-3-319-00948-3_14

Gamma, E. (2002). Éléments of Reusable Object-Oriented Software. In *Design* (Vol. 99). Addison-Wesley Reading, Massachusetts.

García-Borgoñón, L., Barcelona, M. A., García-García, J. A., Alba, M., & Escalona, M. J. (2014). Software process modeling languages: A systematic literature review. *Information and Software Technology*, *56*(2), 103–116. https://doi.org/10.1016/j.infsof.2013.10.001

Garlan, D. (1998). *Higher order connectors*.

Garlan, D. (2008). *Software architecture*.

Garlan, D. (2014). Software architecture: a travelogue. In *Future of Software Engineering Proceedings* (pp. 29–39).

Garlan, D. (1993). Formal approaches to software architecture. *Workshop on Studies of Software Design*, 64–76.

Garlan, D. (2003). Formal modeling and analysis of software architecture: Components, connectors, and events. *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, 1–24.

Garlan, D. (2000). Software architecture: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, 91–101.

Garlan, D. (1995). What is style. *Proceedings of Dagshtul Workshop on Software Architecture*, *100*.

Garlan, D., Monroe, R. T., & Wile, D. (2000). Acme: Architectural description of component-based systems. *Foundations of Component-Based Systems*, *68*, 47–68.

Garlan, D., & Perry, D. E. (1995). Introduction to the special issue on software architecture. *IEEE Trans. Software Eng.*, *21*(4), 269–274.

Gasmallah, N., Amirat, A., Oussalah, M., & Seridi-Bouchelaghemi, H. (2019). Developing an evolution software architecture framework based on six dimensions. *International Journal of Simulation and Process Modelling*, *14*(4), 325–337.

Giacomo, G. De, Lenzerini, M., Leotta, F., & Mecella, M. (2021). From Component-based Architectures to Microservices: A 25-years-long Journey in Designing and Realizing Service-based Systems. In *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future* (pp. 3–15). Springer.

Giese, H., Rumpe, B., Schätz, B., & Sztipanovits, J. (2012). Science and engineering of cyber-physical systems (dagstuhl seminar 11441). *Dagstuhl Reports*, *1*(11).

Giesecke, S., Gottschalk, M., & Hasselbring, W. (2010). *The ArchMapper Approach to Architectural Conformance Checks: An Eclipse-based Tool for Style-oriented Architecture to Code Mappings*.

Grady Booch, & James Rumbaugh, I. J. (2000). The Unified Modeling Language User. *Addison-Welsley Longman Inc*, *15*, 285.

Grønmo, R. (2010). *Using Concrète Syntax in Graph-based Model Transformations.* [Phd thesis. Universty of Oslo, Norway.]. http://hdl.handle.net/10852/10142

Guerra, E., & de Lara, J. (2007). Event-driven grammars: Relating abstract and concrète levels of visual languages. *Software & Systems Modeling*, *6*(3), 317–347.

Guerra, E., & Lara, J. De. (2013). *Model transformation by graph transformation : A comparative study*. *January*.

Haber, A., Ringert, J. O., & Rumpe, B. (2014). Montiarc-architectural modeling of interactive distributed and cyber-physical systems. *ArXiv Preprint ArXiv:1409.6578*.

Haider, U., McGregor, J. D., & Bashroush, R. (2018). The ALI Architecture Description Language. *ACM SIGSOFT Software Engineering Notes*, *43*(4), 52–92.

Hassan, A. (2018). Style and Méta-Style: Another Way to Reuse Software Architecture Evolution [University of Toronto Press]. In *These de Doctorat*, *Universite de Nantes-France*. https://doi.org/10.3138/9781442676121-003

Haumer, P. (2007). Eclipse process framework composer (EPF). *Eclipse Foundation*.

Heckel, R. (2006). Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, *148*(1), 187–198.

Henderson-Sellers, B., & Gonzalez-Perez, C. (2005). A comparison of four process métamodels and the creation of a new generic standard. *Information and Software Technology*, *47*(1), 49–65. https://doi.org/10.1016/j.infsof.2004.06.001

Hidaka, S., Tisi, M., Cabot, J., & Hu, Z. (2016). Feature-based classification of bidirectional transformation approaches. *Software and Systems Modeling*, *15*(3), 907–928. https://doi.org/10.1007/s10270-014-0450-0

Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., & Schürr, A. (2013). A survey of triple graph grammar tools. *Electronic Communications of the EASST*, *57*.

Hirsch, D., Uchitel, S., & Yankelevich, D. (1999). Towards a periodic table of connectors. *COORDINATION*, *1594*, 418.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, *21*(8), 666–677.

Hock-koon, A., & Oussalah, M. (2011). Vers une meilleure compréhension de la différence théorique entre un composant et un service. *INFORSID*, 9–24.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, *23*(5), 279–295. https://ieeexplore.ieee.org/abstract/document/588521/

Hug, C. (2009). *Méthode , modèles et outil pour la méta-modélisation des processus d ' ingénierie de systèmes d ' information*. Université Joseph-Fourier-Grenoble I. France.

IEEE. (n.d.). *IEEE SA - IEEE 1471-2000*. Retrieved February 4, 2022, from https://standards.ieee.org/ieee/1471/2187/

Ilieva, S., Krasteva, I., Benguria, G., & Elvesæter, B. (2010). *Deliverable D2. 8 REMICS Methodology with agile extension, Final Release*.

Ismail, A., Srewil, Y., & Scherer, R. J. (2017). Integrated and collaborative process-based

simulation framework for construction project planning. *International Journal of Simulation and Process Modelling*, *12*(1), 42–53.

Jacobson, I. (1993). *Object-oriented software engineering: a use case driven approach*. Pearson Education India.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., & Valduriez, P. (2006). ATL: a QVT-like transformation language. *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, 719–720.

Jouault, F., & Kurtev, I. (2006). Transforming models with ATL. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *3844 LNCS*, 128–138. https://doi.org/10.1007/11663430_14

Juric, M. B., Mathew, B., & Sarang, P. G. (2006). *Business process éxécution language for web services: an architect and developer's guide to orchestrating web services using BPEL4WS*. Packt Publishing Ltd.

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J., & Varró, D. (2019). Survey and classification of model transformation tools. *Software and Systems Modeling*, *18*(4), 2361–2397. https://doi.org/10.1007/s10270-018-0665-6

Kaur, R., & Sengupta, J. (2013). Software process models and analysis on failure of software development projects. *ArXiv Preprint ArXiv:1306.1068*.

Kelly, S., & Tolvanen, J.-P. (2008). *Domain-specific modeling: enabling full code generation*. John Wiley & Sons.

Kelsen, P., & Ma, Q. (2008). A lightweight approach for defining the formal semantics of a modeling language. *International Conference on Model Driven Engineering Languages and Systems*, 690–704.

Khemissa, H., Ahmed-Nacer, M., & Oussalah, M. (2012). Adaptive Guidance based on Context Profile for Software Process Modeling. *International Journal of Information Technology and Computer Science*, *4*(7), 50–60. https://doi.org/10.5815/ijitcs.2012.07.07

König, B., Nolte, D., Padberg, J., & Rensink, A. (2018). A tutorial on graph transformation. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *10800 LNCS*, 83–104. https://doi.org/10.1007/978-3-319-75396-6_5

Koubaa, A. (2019). Service-oriented software architecture for cloud robotics. *ArXiv Preprint ArXiv:1901.08173*.

Kruchten, P. (2013). Contextualizing agile software development. *Journal of Software: Evolution and Process*, *25*(4), 351–361.

Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *IEEE Software*, *12*(6), 42–50.

Kühne, T. (2006). Matters of (méta-) modeling. *Software & Systems Modeling*, *5*(4), 369–385.

Kuske, S. (2001). A formal semantics of UML state machines based on structured graph transformation. *International Conference on the Unified Modeling Language*, 241–256.

Lambers, L., & Weber, J. (2018). Graph Transformation. *11th International Conference, ICGT 2018, Held as Part of STAF 2018, Toulouse, France, June 25–26, 2018,*.

Lamport, L., & Schneider, F. B. (1984). The``Hoare Logic''of CSP, and All That. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *6*(2), 281–296.

Lasswell, H. D., & Tréanton, J.-R. (1952). L'" Analyse du Contenu" et le Langage de la Politique. *Revue Française de Science Politique*, *2*(3), 505–520.

Le Goaer, O. (2009). *Styles d'évolution dans les architectures logicielles*. Université de Nantes; Ecole Centrale de Nantes (ECN).

Liu, C., van Dongen, B., Assy, N., & van der Aalst, W. (2018). Software architectural model discovery from éxécution data. *13th International Conference on Evaluation of Novel Approaches to Software Engineering*, *3*.

Liu, S., Zeng, R., Sun, Z., & He, X. (2012). SAMAT-A Tool for Software Architecture Modeling and Analysis. *SEKE*, 352–358.

Lonchamp, J. (1993). A structured conceptual and terminological framework for software process engineering. *Proceedings of the 2nd International Conference on the Software Process: SPCON 1993*, 41–53. https://doi.org/10.1109/SPCON.1993.236823

Lübke, D., & Ahrens, M. (2022). *Towards an Experiment for Analyzing Subprocess Navigation in BPMN Tooling*.

Luhmann, N. (2000). *Art as a social system*. Stanford University Press. USA.

Lun, L., & Chi, X. (2010). Software architecture testing in the C2 Style. *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, *1*, V1--123.

Mahar, F., Ali, S. I., Jumani, A. K., & Khan, M. O. (2020). ERP system implementation: planning, management, and administrative issues. *Indian J. Sci. Technol*, *13*(01), 1–22.

Maillard, S., Smeda, A., & Oussalah, M. (2007). COSA: An Architectural Description Méta-Model. *ICSOFT (SE)*, 445–448.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., & Tang, A. (2012). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, *39*(6), 869–891.

Manickam, P., Sangeetha, S., & Subrahmanya, S. V. (2019). *Component-Oriented Development and Assembly: Paradigm, Principles, and Practice using Java*. Auerbach Publications.

Matougui, S., & Beugnard, A. (2005). How to implement software connectors? a reusable, abstract and adaptable connector. *IFIP International Conference on Distributed Applications and Interoperable Systems*, 83–94.

McIlroy, M. D., Buxton, J., Naur, P., & Randell, B. (1968). Mass-produced software components. *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, 88–98.

Medvidovic, N., Oreizy, P., Robbins, J. E., & Taylor, R. N. (1996). Using object-oriented typing to support architectural design in the C2 style. *ACM SIGSOFT Software Engineering Notes*, *21*(6), 24–32.

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, *26*(1), 70–93.

Mehta, N. R., Medvidovic, N., & Phadke, S. (2000). Towards a taxonomy of software connectors. *Proceedings of the 22nd International Conference on Software Engineering*, 178–187.

Mellor, S. J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional.

Menasria, A. (2011). *Une approche basée Transformation de Graphe pour le Model-Checking des Intéractions UML 2.0*. Mémoire de magistère, Université Souk-Ahras.

Menasria, A. (2022). *Abstraction de la communication dans les architectures logicielles.* Thèse doctorat - Université d'Annaba, Algerie.

Menasria, A., Chaoui, A., & Amirat, A. (2012). Graph Rewriting for UML 2.0 Sequence Diagram Decomposition. *MISC 2012- Constantine*.

Menasria, A., & Djebali, M. (1992). *Grafcet pour la commande des procedés industriels: Cas d'un ascensseur*. Université Annaba -Algerie.

Menasria, A., Oussalah, M., Amirat, A., & Bahi, H. (2021). Software Connector as Software Process. *International Journal of Simulation and Process Modelling.*, *17*(4), 231–249.

Mens, T. (2006). On the use of graph transformations for model refactoring. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4143 LNCS*, 219–257. https://doi.org/10.1007/11877028_7

Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, *152*(1–2), 125–142. https://doi.org/10.1016/j.entcs.2005.10.021

Messabihi, E. M. (2011). *Contribution à la spécification et à la vérification des logiciels à base de composants : enrichissement du langage de données de Kmelia et vérication de contrats* [Thèse de doctorat, Université de Nantes]. https://hal.inria.fr/tel-01146895/document

Miller, D. (1992). The $π$-calculus as a theory in linear logic: Preliminary results. *International Workshop on Extensions of Logic Programming*, 242–264.

Minsky, M. (1965). *Matter, mind and models*.

Molesini, A., Denti, E., & Omicini, A. (2021). MDE and MDA in a Multi-Paradigm Modeling Perspective. In *Advancements in Model-Driven Architecture in Software Engineering* (pp. 64–87). IGI Global.

Muccini, H., Arbib, C., Davidsson, P., & Tourchi Moghaddam, M. (2019). An iot software architecture for an evacuable building architecture. *Proceedings of the 52nd Hawaii International Conference on System Sciences*.

Muller, P.-A., Fleurey, F., & Jézéquel, J.-M. (2005). Weaving executability into object-oriented méta-languages. *International Conference on Model Driven Engineering Languages and Systems*, 264–278.

Münch, J., Armbrust, O., Kowalczyk, M., & Sotó, M. (2012). *Software process definition and management*. Springer.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541–580.

Mustapha, K., & Frayret, J.-M. (2016). Agent-based modeling and simulation software architecture for health care. *2016 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 1–12.

Nassar, M., Coulette, B., Guiochet, J., Ebersold, S., Asri, B. El, Crégut, X., & Kriouile, A.

(2016). Vers un profil UML pour la conception de composants multivues. *Revue Des Sciences et Technologies de l'Information-Série L'Objet: Logiciel, Bases de Données, Réseaux*, *11*(4). https://doi.org/10.3166/objet.11.4.83-113ï

Naur, P. (1968). Software engineering-report on a conference sponsored by the NATO Science Committee Garimisch, Germany. *Http://Homepages. Cs. Ncl. Ac. Uk/Brian. Randell/NATO/Nato1968. PDF*.

Nawaz, M. S., & Sun, M. (2018). Reo2PVS: Formal Specification and Verification of Component Connectors. *SEKE*, 390–391.

Ocampo-Pineda, M., Posenato, R., & Zerbato, F. (2022). TimeAwareBPMN-js: An editor and temporal verification tool for Time-Aware BPMN processes. *SoftwareX*, *17*, 100939.

Oliveira Junior, E. A., Pazin, M. G., Gimenes, I., Kulesza, U., & Aleixo, F. A. (2013). SMartySPEM: a SPEM-based approach for variability management in software process lines. *International Conference on Product Focused Software Process Improvement*, 169–183.

OMG-BPMN. (2011). *About the Business Process Model And Notation Specification Version 2.0*. https://www.omg.org/spec/BPMN/2.0/About-BPMN/

OMG-CWM. (2003). *About the Common Warehouse Métamodel Specification Version 1.1*. https://www.omg.org/spec/CWM/1.1/About-CWM/

OMG-Essence. (2018). *About the Essence Specification Version 1.2*. https://www.omg.org/spec/Essence/

OMG-MDA. (2003). *Model Driven Architecture (MDA) | Object Management Group*. https://www.omg.org/mda/

OMG-MOF. (2003). *MétaObject Facility | Object Management Group*. https://www.omg.org/mof/

OMG-OCL. (2015). *About the Object Constraint Language Specification Version 2.4*. https://www.omg.org/spec/OCL/2.4/About-OCL/

OMG-QVT. (2016). *About the MOF Query/View/Transformation Specification Version 1.3*. https://www.omg.org/spec/QVT/1.3/About-QVT/

OMG-SPEM. (2008). *About the Software & Systems Process Engineering Métamodel Specification Version 2.0*. https://www.omg.org/spec/SPEM/2.0/About-SPEM/

OMG-XMI. (2014). *About the XML Métadata Interchange Specification Version 2.5.1*. https://www.omg.org/spec/XMI/2.5.1/About-XMI/

OMG. (1989). *OMG | Object Management Group*. https://www.omg.org/

Osterweil, L. (2011). Software processes are software too. In *Engineering of Software* (pp. 323–344). Springer.

Oussalah, M. (2014). *Architectures logicielles: principes, techniques et outils* (Chapitre 2 (ed.)). Hermes, Lavoisier.

Oussalah, M., Smeda, A., & Khammaci, T. (2004). An explicit definition of connectors for component-based software architecture. *Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2004.*, 44–51.

Papapostolu, T. (2020). μσADL: An architecture description language for MicroServices.

Advances in Intelligent Systems and Computing, *1018*, 885–889. https://doi.org/10.1007/978-3-030-25629-6_138

Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. *Pioneers and Their Contributions to Software Engineering*, 479–498. https://doi.org/10.1007/978-3-642-48354-7_20

Perry, D E. (1997). Software architecture and its relevance to software engineering, invited talk. *Second International Conference on Coordination Models and Languages*.

Perry, Dewayne E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52. https://doi.org/10.1145/141874.141884

Perry, Dewayne E. (1987). Software Interconnexion Models. *ICSE*, *9*, 69.

Pitt, J., & Mamdani, A. (1999). A protocol-based semantics for an agent communication language. *IJCAI*, *99*, 486–491.

Qureshi, T. N., Chen, D., Lönn, H., & Törngren, M. (2011). From east-adl to autosar software architecture: a mapping scheme. *European Conference on Software Architecture*, 328–335.

Rana, T., Bangash, Y. A., & Abbas, H. (2019). Flow constraint language for coordination by exogenous connectors. *IEEE Access*, *7*, 138341–138352.

Randell, B., & Buxton, J. (1970). *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969*. https://eprints.ncl.ac.uk/file_store/production/55593/5C8B829F-C598-48F6-8726-0EF473BB7338.pdf

Rehab, S., & Chaoui, A. (2015). TGG-based process for automating the transformation of UML models towards B specifications. *Int. J. Comput. Aided Eng. Technol.*, *7*(3), 378–400. https://doi.org/10.1504/IJCAET.2015.071299

Rey, A. (2018). *Le Petit Robert: dictionnaire alphabétique et analogique de la langue française/sous la direction de Alain Rey et Josette Rey-Debove*. Le Robert.

Rose, L. M., Herrmannsdoerfer, M., Mazanek, S., Van Gorp, P., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B., & Wimmer, M. (2014). Graph and model transformation tools for model migration. *Software & Systems Modeling*, *13*(1), 323–359. https://doi.org/10.1007/s10270-012-0245-0

Rozenberg, G. (1997). *Handbook of graph grammars and computing by graph transformation* (Vol. 1). World scientific.

Schmidt, D. C. (2006). Model-driven engineering. *Computer-IEEE Computer Society-*, *39*(2), 25.

Schürr, A. (1994). Specification of graph translators with triple graph grammars. *International Workshop on Graph-Theoretic Concepts in Computer Science*, 151–163.

Shaked, A., & Reich, Y. (2021). Requirements for model-based development process design and compliance of standardized models. *Systems*, *9*(1), 1–19. https://doi.org/10.3390/systems9010003

Shannon, C. E., & Weaver, W. (1963). The Mathematical Theory of Communication,(first published in 1949). *Urbana: University of Illinois Press*.

Shaw, M, DeLine, R., Klein, D., … T. R.-I. transactions on, & 1995, U. (1995). Abstractions

for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, *21*(4), 314--335. https://ieeexplore.ieee.org/abstract/document/385970/

Shaw, M, & Garlan, D. (1993). An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering. *AMBRIOLA, V.; TORTORA, G. Advances in Software Engineering and Knowledge Engineering. New Jersey: World Scientific*, *1*.

Shaw, Mary. (1990). Éléments of a design language for software architecture. *Position Paper for IEEE Design Automation Workshop.*

Shaw, Mary. (1993). Procedure calls are the assembly language of software interconnexion: Connectors deserve first-class status. *Workshop on Studies of Software Design*, 17–32.

Shaw, Mary, & Clements, P. (2006). The golden age of software architecture. *IEEE Software*, *23*(2), 31–39.

Shaw, Mary, & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline* (Vol. 1). prentice Hall Englewood Cliffs. http://www.amazon.com/Software-Architecture-Perspectives-Emerging-Discipline/dp/0131829572

Shehory, O., & Sturm, A. (2016). *AGENT-ORIENTED SOFTWARE ENGINEERING*. Springer.

Shroff, G. (2010). *Enterprise cloud computing: technology, architecture, applications*. Cambridge university press.

Silva, E., Medeiros, A. L., Cavalcante, E., & Batista, T. (2013). A lightweight language for software product lines architecture description. *European Conference on Software Architecture*, 114–121.

Simonnot, N. (2016). Tout sur l'architecture: panorama des styles, des courants et des chefs-d'œuvres. *Critique d'art. Actualité Internationale de La Littérature Critique Sur l'art Contemporain*.

Smeda, A. (2006). *Contribution à l'élaboration d'une métamodélisation de description d'architecture logicielle*. https://www.theses.fr/2006NANT2015

Smeda, Adel, Alti, A., Oussalah, M., & Boukerram, A. (2009). Cosastudio: A software architecture modeling tool. *World Academy of Science, Engineering and Technology*, *49*, 263–266.

Smeda, Adel, Khammaci, T., & Oussalah, M. (2004). Software Connectors in the COSA Approach. *Lorenz and Coady [764]*.

Smeda, Adel, Oussalah, M., & Khammaci, T. (2008). My architecture: A knowledge representation méta-model for software architecture. *International Journal of Software Engineering and Knowledge Engineering*, *18*(7), 877–894. https://doi.org/10.1142/S0218194008003921

Srivastava, A., Bhardwaj, S., & Saraswat, S. (2017). SCRUM model for agile methodology. *2017 International Conference on Computing, Communication and Automation (ICCCA)*, 864–869.

Stafford, J. A., Wolf, A. L., & Caporuscio, M. (2003). The application of dependence analysis to software architecture descriptions. *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, 52–62.

Stavridou, V. (1999). Provably dependable software architectures for adaptable avionics.

Gateway to the New Millennium. 18th Digital Avionics Systems Conference. Proceedings (Cat. No. 99CH37033), *2*, 9--C.

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.

Stoica, M., Mircea, M., & Ghilic-Micu, B. (2013). Software development: Agile vs. traditional. *Informatica Economica*, *17*(4).

Striewe, M., Mcneile, A. T., & Berre, A.-J. (2012). Towards an Agile Foundation for the Creation and Enactment of Software Engineering Methods: The SEMAT Approach. In T. U. of D. (DTU) (Ed.), *ECMFA 2012 Joint Proceedings: Co-located Events at the 8th European Conference on Modelling Foundations and Applications* (pp. 279–290). Citeseer.

Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, V., & Ergin, H. (2013). AToMPM : A Web-based Modeling Environment. *16th International Conference on Model Driven Engineering Languages and Systems (MODELS' 2013): September 29-October 4, 2013, Miami, USA*, 21–25.

Taentzer, G. (2004). AGG: A graph transformation environment for modeling and validation of software. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *3062*, 446–453. https://doi.org/10.1007/978-3-540-25959-6_35

Taylor, R. N. (2019). Software Architecture and Design. In *Handbook of Software Engineering* (pp. 93–122). Springer.

Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). *Software architecture: foundations, theory, and practice* (Wiley (ed.)).

Telemecanique. (n.d.). *Telemecanique Sensor*. https://tesensors.com/fr/fr

Tibermacine, C., Sadou, S., Minh, T., That, T., Dony, C., & Ton That, M. T. (2017). Software architecture constraint reuse-by-composition. *Future Generation Computer Systems*, *61*, 37--53. https://doi.org/10.1016/j.future.2016.02.006ï

UML, O. M. G. (2017). *OMG (2017) Unified Modeling Language®(OMG UML®) Version 2.5. 1 https://www. omg. org/spec*.

Van Tendeloo, Y., Van Mierlo, S., & Vangheluwe, H. (2019). A Multi-Paradigm Modelling approach to live modelling. *Software and Systems Modeling*, *18*(5), 2821–2842. https://doi.org/10.1007/s10270-018-0700-7

Van Heesch, U., Avgeriou, P., & Hilliard, R. (2012). A documentation framework for architecture decisions. *Journal of Systems and Software*, *85*(4), 795–820.

Van Tendeloo, Y., & Vangheluwe, H. (2017). The Modelverse: a tool for multi-paradigm modelling and simulation. *2017 Winter Simulation Conference (Wsc)*, *110265*, 944--955.

Von Neumann, J. (1986). *Papers of John von Neumann on computers and computer theory*.

Vu, T. M., Probst, C., Nielsen, A., Bai, H., Buckley, C., Meier, P. S., Strong, M., Brennan, A., & Purshouse, R. C. (2020). A software architecture for mechanism-based social systems modelling in agent-based simulation models. *Journal of Artificial Societies and Social Simulation: JASSS*, *23*(3).

Wang, L., Egorova, E. K., & Mokryakov, A. V. (2018). Development of hypergraph theory. *Journal of Computer \& Systems Sciences International*, *57*(1), 109–114.

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger*, *1*(1), 7–87.

Wellington, J., & Smith, B. (1995). *ISO TC 184/SC4 Reference Manual*.

Whittle, J., Hutchinson, J., & Rouncefield, M. (2013). The state of practice in model-driven engineering. *IEEE Software*, *31*(3), 79–85.

Zamli, K. Z. (2001). Process modeling languages: A literature review. *Malaysian Journal of Computer Science*, *14*(2), 26–37.

Annexe A :

Squelette de code Java généré par la grammaire « capADL2Java » pour l'architecture Pipe&Filter

```
public class Pipe&Filter extends ProcessConfiguration {

    ProcessComponent C1 = new C1();
    ProcessComponent C2 = new C2();

    ProcessConnector dc = new dC();

    public void start() {

        ((dC) dc).p1.dataInfo = ((C1) c1).p4.dataInfo
        ((dC) dc).p2.ctrlInfo = ((C1) c1).p5.ctrlInfo

        ((dC) dc).LaunchConnectorÉxécution();

        ((C2) c2).p1.dataInfo = ((dC) dc).p4.dataInfo;
        ((C2) c2).p2.ctrlInfo = ((dC) dc).p7.ctrlInfo;
    }
}

public class C1 extends ProcessComponent {

    public R_DataPortCom p1 = new R_DataPortCom();
    public R_CtrlPortCom p2 = new R_CtrlPortCom();
    public P_DataPortCom p3 = new P_DataPortCom();
    public P_DataPortCom p4 = new P_DataPortCom();

          //Component methods
}

public class C2 extends ProcessComponent {

    public R_DataPortCom p1 = new R_DataPortCom();
    public R_CtrlPortCom p2 = new R_CtrlPortCom();
    public P_DataPortCom p3 = new P_DataPortCom();
    public P_DataPortCom p4 = new P_DataPortCom();

          //Component methods
}

public class dC extends ProcessConnector {

    class ActordC extends Actor {

        @Override
        public void run() {
            surveillanceActor();
            super.run();
        }

        public void surveillanceActor() {

            if (p1.dataInfo != null || p2.ctrlInfo != null) {
                ((Diffusion) d1).DiffusionLogic(p1, p3,p4,p5);
                ((Decision) ds1).DecisionLogic(p2, p6,p7,p8);

                // reset PortEntree
                p1.dataInfo = "";
                p2.ctrlInfo = "";
            }

        }
    }

    public R_DataPortCon p1 = new R_DataPortCon();
    public R_CtrlPortCon p2 = new R_CtrlPortCon();
    public P_DataPortCon p3 = new P_DataPortCon();
    public P_DataPortCon p4 = new P_DataPortCon();
    public P_DataPortCon p5 = new P_DataPortCon();
```

```
    public P_CtrlPortCon p6 = new P_CtrlPortCon();
    public P_CtrlPortCon p7 = new P_CtrlPortCon();
    public P_CtrlPortCon p8 = new P_CtrlPortCon();

    public Actor a1 = new ActordC();

    public ServDataFlow d1 = new Diffusion();
    public ServCtrlFlow ds1 = new Decision();

    public void LaunchConnectorÉxécution() {
        ((ActordC) this.a1).run();
    }
}
public class Decision extends ServCtrlFlow {

    public void DecisionLogic(R_CtrlPortCon p2, P_CtrlPortCon p6,p7,p8) {

        // service Decision Logic.

        {
            // core
        }

        P6.ctrlInfo = p2.ctrlInfo;
        P7.ctrlInfo = p2.ctrlInfo;
        P8.ctrlInfo = p2.ctrlInfo;

    }

}
public class Diffusion extends ServDataFlow {

    public void DiffusionLogic(R_DataPortCon p1, P_DataPortCon p3,p4,p5) {

        // logic de service Diffusion.

        {
            // core
        }

        P3.dataInfo = p1.dataInfo;
        P4.dataInfo = p1.dataInfo;
        P5.dataInfo = p1.dataInfo;

    }

}

public class Actor implements Runnable {
    String name;
    String nature;
    String mode;

    public static Actor defineActor() {
        Actor actor = null;
        return actor;
    }

    @Override
    public void run() {

    }
}

public class ProcessConfiguration {
        public void start() {
        }
}
public class ProcessComponent {

}
public class ProcessConnector {

}
public class PortComponent {
    public String name;
    public String IDConnector;
```

```
}
public class R_DataPortCom extends PortComponent {
    public String dataInfo = "";
}
public class R_CtrlPortCom extends PortComponent {
    public String ctrlInfo = "";
}
public class P_DataPortCom extends PortComponent {
    public String dataInfo = "";
}
public class P_CtrlPortCom extends PortComponent {
    public String ctrlInfo = "";
}
public class PortConnector {
    public String name;
    public String IDConnector;
}
public class R_DataPortCon extends PortConnector {
    public String dataInfo = "";
}
public class R CtrlPortCon extends PortConnector {
    public String ctrlInfo = "";
}
public class P DataPortCon extends PortConnector {
    public String dataInfo = "";
}
public class P CtrlPortCon extends PortConnector {
    public String ctrlInfo = "";
}

public class Service {
    public String name;
    public String IDConnector;
    public Category SCate;
}
enum Category {
    COMMUNICATION,
    COORDINATION,
    FACILITATION,
    CONVERSION,
}
public class ServDataFlow extends Service {

}
public class ServCtrlFlow extends Service {

}
public class Precedence extends ServCtrlFlow {

}
```

Annexe B :

Squelette de code Java généré par la grammaire « capADL2Java » pour l'architecture Cycle « V »

```
public class V_Cycle_Life extends ProcessConfiguration {

    ProcessComponent architecturalDesign = new ArchitecturalDesign();
    ProcessComponent detailedDesign = new DetailedDesign();

    ProcessConnector c1 = new C1();

    public void start() {

        ((C1) c1).p1.dataInfo = ((ArchitecturalDesign) architecturalDesign).p4.dataInfo
        ((C1) c1).p2.ctrlInfo = ((ArchitecturalDesign) architecturalDesign).p5.ctrlInfo

        ((C1) c1).LaunchConnectorÉxécution();

        ((DetailedDesign) DetailedDesign).p1.dataInfo = ((C1) c1).p3.dataInfo;
        ((DetailedDesign) DetailedDesign).p2.ctrlInfo = ((C1) c1).p4.ctrlInfo;
    }
}

public class ArchitecturalDesign extends ProcessComponent {

    public R_DataPortCom p1 = new R_DataPortCom();
    public R_CtrlPortCom p2 = new R_CtrlPortCom();
    public P_DataPortCom p3 = new P_DataPortCom();
    public P_DataPortCom p4 = new P_DataPortCom();
    public P CtrlPortCom p5 = new P CtrlPortCom();

          //Component methods
}

public class DetailedDesign extends ProcessComponent {

    public R_DataPortCom p1 = new R_DataPortCom();
    public R_CtrlPortCom p2 = new R_CtrlPortCom();
    public P_DataPortCom p3 = new P_DataPortCom();
    public P_DataPortCom p4 = new P_DataPortCom();
    public P CtrlPortCom p5 = new P CtrlPortCom();

          //Component methods
}

public class C1 extends ProcessConnector {

    class ActorC1 extends Actor {

        @Override
        public void run() {
            surveillanceActor();
            super.run();
        }

        public void surveillanceActor() {

            if (p1.dataInfo != null || p2.ctrlInfo != null) {
                ((Transmition) t1).TransmitionLogic(p1, p3);
                ((EndBegin) eb1).EndBeginLogic(p2, p4);

                // reset PortEntree
                p1.dataInfo = "";
                p2.ctrlInfo = "";
            }

        }
    }

    public R_DataPortCon p1 = new R_DataPortCon();
```

```
    public R_CtrlPortCon p2 = new R_CtrlPortCon();
    public P_DataPortCon p3 = new P_DataPortCon();
    public P CtrlPortCon p4 = new P CtrlPortCon();

    public Actor a1 = new ActorC1();

    public ServDataFlow t1 = new Transmition();
    public ServCtrlFlow eb1 = new EndBegin();

    public void LaunchConnectorÉxécution() {
        ((ActorC1) this.a1).run();
    }
}
public class EndBegin extends Precedence {

    public void EndBeginLogic(R_CtrlPortCon p1, P_CtrlPortCon p2) {

        // service Transmition Logic.

        {
            // core
        }

        p2.ctrlInfo = p1.ctrlInfo;
    }

}
public class Transmition extends ServDataFlow {

    public void TransmitionLogic(R DataPortCon p1, P DataPortCon p2) {

        // logic de service transmition.

        {
            // core
        }

        p2.dataInfo = p1.dataInfo;
    }

}
public class Actor implements Runnable {
    String name;
    String nature;
    String mode;

    public static Actor defineActor() {
        Actor actor = null;
        return actor;
    }

    @Override
    public void run() {

    }
}

public class ProcessConfiguration {
        public void start() {
        }
}
public class ProcessComponent {

}
public class ProcessConnector {

}
public class PortComponent {
    public String name;
    public String IDConnector;
}
public class R DataPortCom extends PortComponent {
    public String dataInfo = "";
}
public class R_CtrlPortCom extends PortComponent {
    public String ctrlInfo = "";
}
```

```
public class P_DataPortCom extends PortComponent {
    public String dataInfo = "";
}
public class P CtrlPortCom extends PortComponent {
    public String ctrlInfo = "";
}
public class PortConnector {
    public String name;
    public String IDConnector;
}
public class R_DataPortCon extends PortConnector {
    public String dataInfo = "";
}
public class R CtrlPortCon extends PortConnector {
    public String ctrlInfo = "";
}
public class P_DataPortCon extends PortConnector {
    public String dataInfo = "";
}
public class P CtrlPortCon extends PortConnector {
    public String ctrlInfo = "";
}

public class Service {
    public String name;
    public String IDConnector;
    public Category SCate;
}
enum Category {
    COMMUNICATION,
    COORDINATION,
    FACILITATION,
    CONVERSION,
}
public class ServDataFlow extends Service {

}
public class ServCtrlFlow extends Service {

}
public class Precedence extends ServCtrlFlow {

}
```