Modélisation rigoureuse au service du logiciel et des systèmes

Pascal Andre

Nantes Université

HABILITATION À DIRIGER DES RECHERCHES

HDR

NANTES UNIVERSITÉ

Spécialité : *Informatique*

Par

Pascal ANDRE

Modélisation rigoureuse au service du logiciel et des systèmes

Rester modeste et avancer ensemble

présentée et soutenue à Nantes, le 1 juillet 2024
Unité de recherche : Unité de recherche : LS2N CNRS UMR 6004

Rapporteurs avant soutenance :

Agnès FRONT	Professeur, Université Grenoble Alpes
Antoine BEUGNARD	Professeur, IMT Atlantique - Brest
Pascal POIZAT	Professeur, Université Paris Nanterre

Composition du Jury :

Présidente :	Catherine DA CUNHA	Professeure, Ecole Centrale de Nantes
Examinateurs :	Agnès FRONT	Professeure, Université Grenoble Alpes
	Antoine BEUGNARD	Professeur, IMT Atlantique - Brest
	Pascal POIZAT	Professeur, Université Paris Nanterre
	Stéphane DUCASSE	Directeur de recherche, INRIA - Lille
	Mireille BLAY-FORNARINO	Professeure, Université de Nice

Invité(s) :

Christian ATTIOGBE	Professeur, Université de Nantes
Benoit DELAHAYE	MCF HDR, Université de Nantes

"Savoir écouter, c'est posséder, outre le sien, le cerveau des autres".

Léonard De Vinci

Remerciements

Je remercie Stéphane Ducasse, Agnès Front, Antoine Beugnard et Pascal Poizat d'avoir accepté de lire mon manuscrit et rapporter sur mes travaux de recherche. Leurs suggestions ont permis d'améliorer ce manuscrit. Je remercie Mireille Blay-Fornarino et Catherine Da Cunha de me faire l'honneur de participer à ce jury et de prendre le temps d'examiner ma trajectoire de recherche. Qu'ils trouvent ici mon estime non seulement pour leurs qualités scientifiques mais aussi leurs qualités humaines.

Je dédie ce mémoire à tous ceux qui m'ont guidé sur le chemin. Merci à Jean-Claude Royer qui m'a mis sur la voie de la recherche. Sa rigueur et son honnêteté scientifique m'ont toujours inspiré. Merci à Henri Habrias et aux collègues de l'équipe Méthodes et Spécifications Formelles (MSF) de l'Institut de Recherche en Informatique de Nantes (IRIN) qui ont permis mon recrutement comme MCF et donc pouvoir démarrer une carrière d'enseignant-chercheur. Merci à Christian Attiogbé, longtemps mon responsable d'équipe, qui ressemble à une famille, mais aussi co-encadrant principal et collaborateur de nombreux articles et relecteur complet de ce manuscrit. Christian est un modèle de Directeur de recherches. Merci à Benoit Delahaye, qui dirige l'équipe avec la même bienveillance ; merci pour ton soutien au quotidien et nos nombreuses discussions. Je remercie aussi les Directions successives du LS2N et les personnels, les assistant.e.s d'équipe, qui au fil du temps ont construit un environnement de travail épanouissant. Chacun se reconnaîtra. Merci à Alexiane pour la gestion logistique efficace de cette HDR et de l'équipe.

L'essence même du métier de chercheur est d'échanger, de discuter, de confronter sa vision à celle des autres. Les travaux présentés ici sont le résultat de rencontres, de collaborations. Outre les personnes mentionnées ci-dessus, je remercie mes collègues de l'équipe, notamment Gilles avec qui je partage un bureau depuis vingt et un ans, les collègues d'enseignement en informatique et LEA, les doctorants et les étudiants que j'ai co-encadré qui m'ont permis de réfléchir sur le métier lui-même, les chercheurs rencontrés dans les différentes manifestations, celles et ceux avec qui j'ai collaboré temporairement ou durablement. Tous m'ont fait vivre la variété des pratiques d'enseignement-recherche, et réaliser que les piliers sont l'honnêteté, la confiance, le partage et le travail.

Je remercie Karine, pour son soutien sans faille, Baptiste et Edgar pour leur compréhension d'un métier où il est difficile à la fois de borner le temps de travail et l'esprit qui vagabonde.

Merci à vous de lire sans imprimer ce document, pensez à la planète.

Sommaire

Introduction

La notion de modèle étant au cœur de ce manuscrit, je ne résiste pas à lancer la réflexion que je me suis faite au sujet de l'HDR par un premier modèle de l'HDR illustré par la carte visuelle ou carte mentale (*mind map*) de la Figure 1, qui représente l'idée que je me fais de ce diplôme. En partant du bas à gauche, on trouve les `Thèmes` de recherche autour du génie logiciel et des systèmes, qui seront expliqués en détail dans le chapitre 1, puis les thèmes de la `Réflexion` sur les problèmes pour arriver à des travaux organisés menant à des `Contributions` ou des `Projets` collaboratifs menés avec des `Personnes` et des `Acteurs` avec en particulier l'encadrement de `Doctorants`. L'activité liée à tous ces sujets est valorisée sous forme de `Publications`, outils et transferts. Pour mener à bien

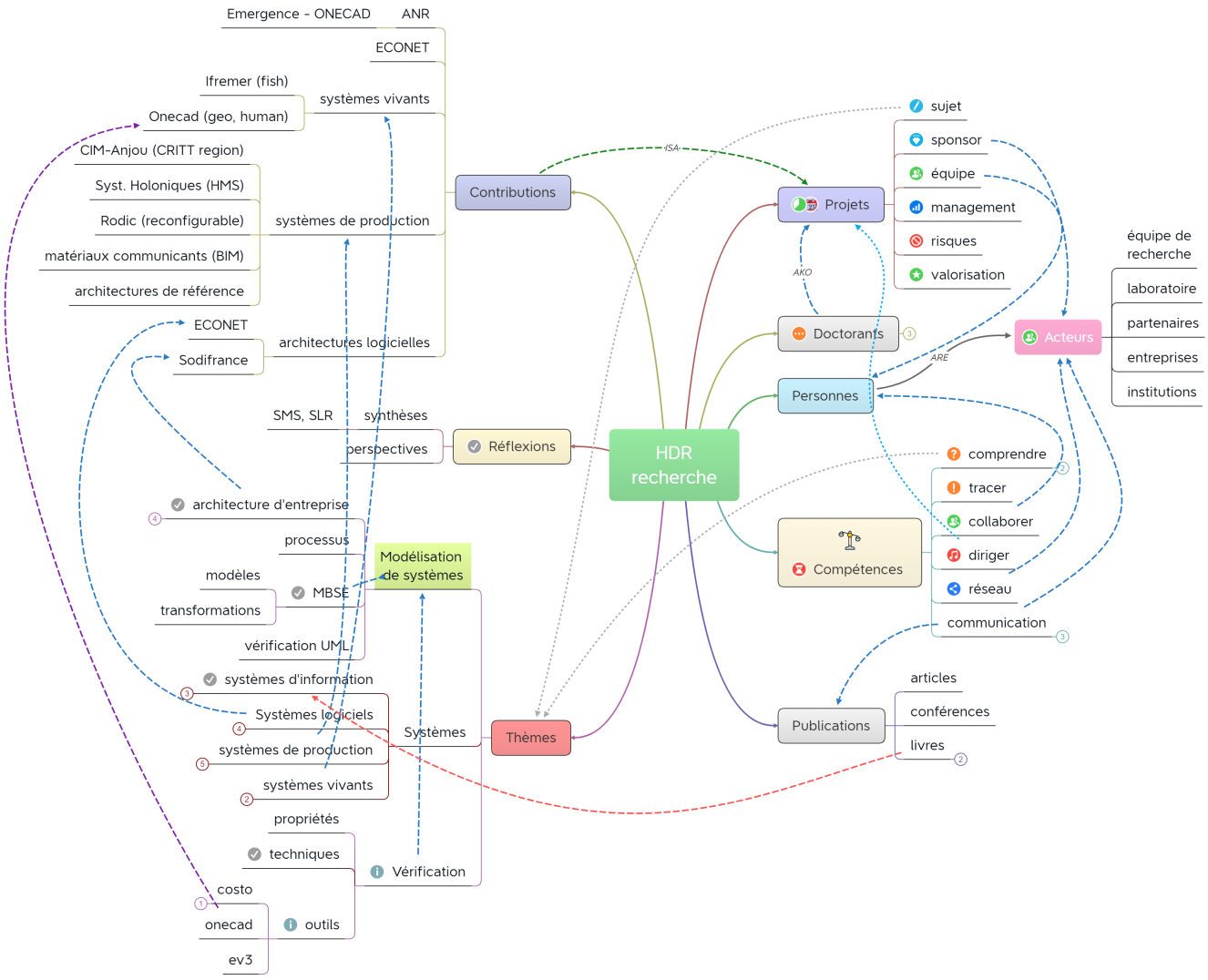


Figure 1 – Un modèle de l'HDR

tous ces sujets, il faut développer des `Compétences` qui font qu'un directeur de recherche doit couvrir un profil complet du point de vue scientifique, organisationnel et humain avec des compétences de type savoir, savoir-faire et savoir-être. L'objectif de ce manuscrit est finalement de démontrer, par mon expérience, que j'ai acquis en partie toutes ces compétences, mais, restons modeste, à un niveau suffisant sans pour autant être (encore) excellent.

Il n'y a pas de standard pour présenter une habilitation à Diriger des Recherches, ce qui en soi est un indicateur pour le candidat qui doit chercher sa voie, comme il le fait en recherche depuis plus ou moins longtemps. Me concernant, cela fait plus de 25 ans que je mène un parcours non tracé à l'avance, et vécu au fil des rencontres, d'opportunités, d'échecs aussi. La vie de l'enseignant-chercheur n'est pas un long fleuve tranquille, c'est le lieu où se mèlent de multiples activités, enrichissantes ou pas, découvertes et initiatives plus ou moins contraintes, des soutiens mais aussi des coups bas...

Mon parcours de recherche a démarré au siècle dernier, lors de mon second diplôme d'études supérieures spécialisées (DESS) d'informatique, intitulé Génie Informatique (1990-1991) à l'Université de Nantes, diplôme spécialisé dans l'approche à objets, à l'époque émergente. J'ai effectué un stage de recherche avec Jean-Claude Royer, dans lequel j'ai proposé une technique d'optimisation de la recherche de méthodes dans un graphe d'héritage, travail publié à la conférence OOPSLA en 1992 [AR92]. Cette période marque mon entrée dans le monde du génie logiciel avec Jean-Claude Royer et Jean Bézivin, sur une thèse autour des méthodes formelles et à objets pour le développement du logiciel, présentée à l'université de Rennes I. Je suis finalement resté fidèle à cette thématique autour de la rigueur, de la modularité et des modèles pour le génie logiciel tout au long de mon parcours. Ce dernier est aussi marqué, notamment en enseignement, par la conception de systèmes d'information et l'usage de méthodes pour construire des applications logicielles.

Le développement du logiciel est un processus de production complexe dont les qualités restent insatisfaisantes : les programmes informatiques contiennent trop d'erreurs, trop de projets échouent. Mes recherches contribuent au génie logiciel et visent à améliorer la qualité des produits et du processus. Deux axes sont privilégiés : des modèles modulaires composables et de la rigueur. Le premier axe s'inscrit dans le principe du logiciel construit à haut niveau d'abstraction par assemblage de briques réutilisables et adaptables. Le second axe s'appuie sur les méthodes formelles et la vérification de propriétés. La conjonction des deux permet de mettre en œuvre des architectures logicielles qui seront raffinées vers le code. Cette relation entre des modèles abstraits et du code (modèles concrets) s'inscrit aussi dans l'ingénierie et la rétro-ingénierie des modèles, approches qui permettent d'automatiser une partie du développement.

Le génie logiciel est destiné non seulement aux informaticiens développeurs de solutions logicielles de qualité mais aussi à tous les participants ayant des rôles divers et variés dans le processus de construction de ces solutions. Les acteurs du développement comprennent ainsi les utilisateurs finaux, les architectes, les développeurs, les testeurs, les exploitants.

Le domaine d'application couvre quasiment toutes les activités de la vie courante des systèmes d'information, des systèmes numériques ou cybernétiques. La qualité couvre non seulement la sûreté de fonctionnement ou la fiabilité mais de nombreuses autres propriétés telles que l'efficacité, la sécurité, la maintenabilité, etc. Je m'intéresse ainsi à l'application du génie logiciel dans différents domaines d'application et pour différents acteurs ayant différents objectifs. Ceci m'amène, sur la base de compétences acquises sur ces thématiques, à échanger ou à participer à des projets pluridisciplinaires. J'ai ainsi collaboré sur des projets pluridisciplinaires avec des géographes, des gestionnaires, des juristes, des linguistes, des mécaniciens, des automaticiens... Je me considère ainsi comme un Docteur généraliste en Génie Logiciel et Systèmes d'information dont le métier, enseignant-chercheur, couvre à la fois l'envie d'aller plus loin sur certains sujets en proposant une vision originale mais aussi celle d'aider les étudiants à avancer dans leur parcours vers le monde professionnel.

Candidater à l'HDR est l'occasion de réfléchir à son parcours et ses expériences en imaginant aider d'autres chercheurs à avancer dans leur parcours en fixant des caps dans le temps. Cela nécessite une certaine maturité pour analyser succès et échecs, pour réfléchir aux bonnes ou mauvaises pratiques de recherche. Pour présenter cette réflexion, ce manuscrit est organisé en quatre parties et huit chapitres.

1. La partie I pose les bases de ma réflexion en positionnant les modèles au centre du débat.
   - Le chapitre 1 *"A propos de logiciel, systèmes, modèles et ingénierie"* définit le domaine thématique principal de mes travaux, le génie logiciel. J'ai essayé de le rendre autosuffisant à la fois dans la description des concepts et dans mon interprétation. Il se termine par un modèle du domaine.
   - Le chapitre 2, au nom évocateur *"Mythes et réalités"*, met en évidence le périmètre choisi dans ce domaine avec les hypothèses, puis les enjeux, les problèmes et les défis auquels je m'intéresse. J'y précise ma vision et le positionnement de mes travaux. Il se termine par des pistes de recherche.
2. La partie II se situe au cœur de mon domaine de recherche, le génie logiciel.
   - Le chapitre 3 *"Raisonner sur les architectures"* présente une synthèse des travaux et résultats autour de l'approche Kmelia pour le développement par composants et services. Ces travaux ont été très enrichissants sur la manière de développer un projet scientifique collaboratif structurant.
   - Le chapitre 4 *"Développer du logiciel par les modèles"* est au cœur des problématiques du chapitre 2, à savoir comment automatiser en partie le développement du logiciel. Ces travaux sont ambitieux, voire illusoires, et donc difficiles à structurer en projets, c'est aussi ce qui en fait l'intérêt.
   - Le chapitre 5 *"Améliorer la sécurité des applications"* se focalise sur un aspect qualité, souvent négligé par les développeurs, celui de la sécurité. A l'instar d'autres qualité du logiciel, l'idée est bien souvent de définir des modèles de sécurité et de vérifier des propriétés sur ces modèles.

3. La partie III explore les interactions avec des domaines d'autres disciplines que l'informatique.
   — Le chapitre 6 *"Aligner le logiciel aux processus métiers de l'entreprise"* correspond à la thématique de la conception des systèmes d'information mentionnée ci-avant. On s'appuie sur les modèles pour rendre cohérentes les visions informatique et gestion des systèmes d'information.
   — Le chapitre 7 *"Rationaliser les systèmes de production"* met en évidence l'apport du génie logiciel dans les pratiques de génie industriel et notamment du contrôle adaptatif des systèmes de production.
4. Pour aspirer à diriger des recherches, il faut raisonnablement se poser la question de ce que c'est et quelles sont les compétences attendues, au delà de la maîtrise scientifique des sujets. La partie IV explore le champ des compétences nécessaires. Le chapitre 8 *"Approche multi-compétences de la recherche"* synthétise le fruit de mes recherches et de ma réflexion sur les compétences des chercheurs, de l'encadrement et de la direction de recherches.

Ce manuscrit est rédigé sous forme de discussions. Il aborde de nombreux sujets et chaque chapitre peut se lire indépendamment des autres. La FIGURE 2 représente un modèle de lecture de ce document sous forme d'un arbre. Le lecteur est invité à commencer

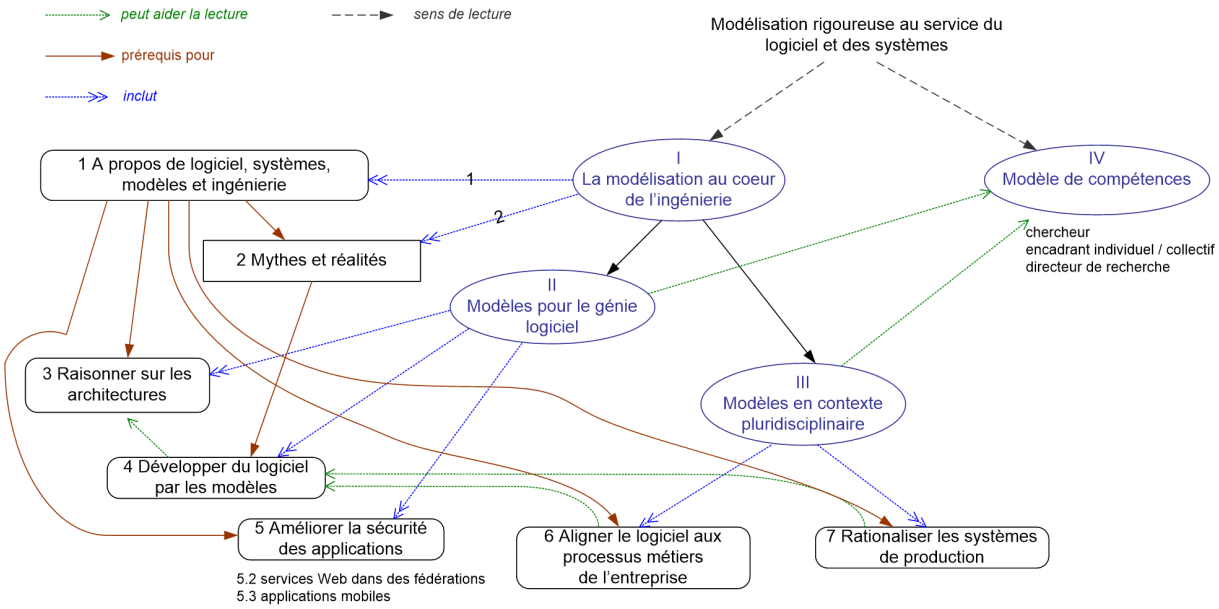


FIGURE 2 – Un modèle de lecture

par la partie I pour avoir une vue d'ensemble, cette partie est indispensable pour comprendre les parties II et III. La partie IV, dédié aux compétences est une réflexion assez générique et donc indépendante. Toutefois les illustrations et la personnalisation peuvent faire références à mes travaux et donc aux aux parties II et III. Inversement, à la fin de chaque chapitre je mentionne des compétences individuelles qui seront synthétisées dans le chapitre 8. Mes perspectives de recherche sont motivées en fin de document à partie de la page 259. Les annexes fournissent une liste de publications et des éléments complémen-

taires pour aller plus loin dans la compréhension du périmètre génie logiciel (annexes B et C).

Certaines parties sont mises en évidence avec des boîtes. Voici la sémantique des boîtes anonymes.

Énoncé important tel qu'une définition, une classification, une assertion...

Remarque d'ordre méthodologique

Commentaire important

Règle préconisée

C

Description d'une compétence C

Tous les travaux que j'ai menés ne sont pas décrits dans ce document. Seuls ceux pour lesquels je suis à l'initiative directe ou indirecte depuis 2005 sont mentionnés. Les travaux autour des systèmes hétérogènes et les approches formelles [AAL18a ; AAL18b ; AAL20] avec Christian Attiogbé, approches algébriques [And+00 ; ARR00] avec Jean-Claude Royer, [HA04 ; AH06] avec Henri Habrias, ou ceux [AAS04 ; AA05] avec Gilles Ardourel et Gerson Sunyé par exemple ne sont pas détaillés dans le manuscrit. De même je ne mentionne que les projets dans lesquels j'ai été activement impliqué.

Première partie

La modélisation au cœur de l'ingénierie

Engineering is the closest thing to magic that exists in the world.

Elon Musk

L'informatique est omniprésente (ubiquitaire, *pervasive*) dans la vie personnelle et professionnelle. Tous les métiers sont concernés ou presque et tous les individus le sont. Le logiciel est donc omniprésent et sa conception nécessite des techniques d'ingénierie [1]. Quel que soit le domaine, ingénierie et modélisation sont indissociables. *"Modeling should be an independent scientific discipline."* [CV22] Il est donc logique et nécessaire d'en faire un objet de recherche.

Dans cette partie, je vais poser mon domaine de recherche sur ces sujets dans le chapitre 1 puis mes questions de recherche dans le chapitre 2.

1. *Existe-t-il un modèle français de l'ingénierie ?* Nous laissons le lecteur trouver des réponses à cette question dans [Bar13]. L'ingénierie du logiciel est une synthèse de différents courants de pensée et de bonnes pratiques issues de l'expérience.

Chapitre 1

A propos de logiciel, systèmes, modèles et ingénierie

Software and cathedrals are much the same – first we build them, then we pray.

ISPW 1988
Sam Redwine

Ce titre peut paraître curieux mais il montre que le domaine est vaste et le périmètre reste ouvert ; dans ce chapitre nous allons ouvrir et fermer des portes et des connexions. La présentation se veut didactique et non technique. Le lecteur connait certainement tout ou partie des termes définis dans ce chapitre mais il est important d'en expliciter mon interprétation pour éviter des confusions dans la suite du manuscrit. L'idée est d'aller au fond des choses pour fixer les concepts du domaine, tout en restant abstrait et non exhaustif pour rester concis.

Remarque

La classification joue un grand rôle dans la maîtrise et l'explication d'un domaine. C'est un exercice difficile, surtout à cause des exceptions ou lorsque plusieurs critères interviennent ; mais aussi parce qu'une classification évolue en permanence.

1.1 Du logiciel au génie logiciel

Mes travaux s'inscrivent majoritairement dans le cadre du génie logiciel, avec un domaine d'application historique celui des systèmes d'information mais il touche aussi à l'ingénierie des systèmes et la cyber-physique, partout où des modèles sont utiles. Essayons de définir et commenter les concepts importants de ce contexte.

1.1.1 Logiciel

Selon le Larousse, un **logiciel** est un "*ensemble des programmes, procédés et règles, et éventuellement de la documentation, relatifs au fonctionnement d'un ensemble de traitement de données.*".[1] Ce n'est donc pas simplement une suite d'instructions exécu-

1. La définition de Presssman est plus précise sur le rôle des données : *"Software is : (1) instructions (computer programs) that when executed provide desired features, function, and performance ; (2) data*

tées par un ordinateur (un programme informatique). Rappelons au passage la trilogie DIC [BMD10], selon laquelle les **données** sont structurées en **informations** qui une fois interprétées forment des **connaissances**. Le Larousse précise qu'un **logiciel d'application**, que nous appelerons plus simplement une **application** est un "*logiciel adapté à la résolution d'un problème spécifique.*" tandis qu'un logiciel de base est un "*logiciel chargé d'assurer le fonctionnement interne d'un ordinateur.*" et un logiciel malveillant est "*un programme développé dans le but de nuire à un système informatique.*"

La donnée suit un cycle de vie formé d'actions CRUD (*Create, Read, Update, Delete*) qui sert parfois de patron pour structurer les traitements.

Nous classons les applications en trois catégories :

1. L'informatique de gestion met en œuvre les systèmes d'information d'entreprise. Ces systèmes sont caractérisés par un stockage important d'informations et de nombreux traitements simples en consultation et mise à jour de ces informations *e.g.* eCommerce, gestion administrative et financière.
2. L'informatique temps réel (automatisme, contrôle de processus) se caractérise par une réactivité très forte du système contrôlé *e.g.* pilotage automatique d'avion, surveillance de centrale nucléaire. Nous rangeons, un peu arbitrairement, sous cette dénomination l'informatique embarquée (automatismes), le contrôle de processus (fabrication, surveillance), les applications où le temps joue un rôle majeur ou encore les réseaux informatiques. L'interface entre le système d'information et son objet naturel (le processus) prend ici une grande importance. Elle inclut des aspects matériels non négligeables : capteurs (pour récupérer les informations), actionneurs (pour piloter un objet réel), bus de communications...
3. L'informatique scientifique se caractérise par des calculs complexes et nombreux *e.g.* simulation, prévisions météo, imagerie médicale, analyse de données, apprentissage en intelligence artificielle. Une base d'information peut servir à stocker les informations de base ou les résultats mais intervient assez peu dans le calcul lui-même. L'utilisation de modèles mathématiques est incontournable, qu'il soient statistiques, probabilistes, numériques... Des critères importants sont l'efficacité des traitements et la quantité de données nécessaires et leur qualité.

Nous estimons que ces trois catégories représentent grossièrement les différents types de systèmes logiciels même s'il existe des classifications plus détaillées. Wikipedia propose cinq grand domaines[2]. Pressman propose sept domaines (système, application, scienti-

structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs." [Pre10]

2. `https://fr.wikipedia.org/wiki/Informatique`

fique, embarqué, lignes de produits, web apps, intelligence articielle) [Pre10]. Sommerville en propose huit (personnel, transactionnel, embarqué, *batch*, *entertainment*, simulation, *data*, systèmes de systèmes) [Som11]. L'Object Management Group classes les applications en 12 domaines métiers et 10 domaines techniques [3].

Bien que ce ne soit pas toujours explicite, il est important de mentionner que le logiciel a deux rôles, comme le mentionne Pressman : *"Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a* ***product****, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware. [...] As the* ***vehicle*** *used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments)"* [Pre10]. Cette faculté à pouvoir réfléchir ou agir sur soi-même (réflexion, métainformation) est une caractéristique originale, liée à la caractéristique "logique" de l'objet "logiciel" par rapport à des objets "physiques" comme les objets techniques des sciences de l'ingénieur ou les êtres vivants des sciences naturelles. Ces objets sont étudiés à travers la notion de systèmes et manipulés en utilisant des modèles, que nous verrons dans la section 1.2. Ce n'est pas la seule différence, le logiciel est caractérisé par le fait que (i) contrairement aux objets manufacturés, la qualité du logiciel ne dépend que de sa conception et pas de sa production, (ii) il ne se dégrade pas inexorablement avec le temps et (iii) malgré l'industrialisation, il reste fortement personnalisé [Pre10]. Contrairement à un objet technique, un logiciel peut évoluer indéfiniment. Le coût du logiciel est majoritairement celui de sa maintenance soit 75% du total [Boe82].

Le génie logiciel se focalise plutôt sur le second rôle (la fabrication du logiciel) dans l'objectif d'améliorer la qualité du premier rôle (l'application du logiciel). Nous y reviendrons dans la section 1.1.4. Il cherche à répondre à la question de comment le concevoir, ce que nous abordons dans la dans la section 1.1.2.

1.1.2 Développement du logiciel

Développer, c'est passer de l'idée au code (exprimer, programmer, vérifier) avec méthode (dans le bon ordre, éviter l'anarchie, travailler en groupe) et qualité à la fois des modèles produits dans le développement (corrects, fiables, évolutifs...) et du processus de développement (efficace, rentable...).

— Dans un développement individuel, l'analyse peut être légère ou « dans la tête », la programmation incrémentale et itérative sur le test comme dans les méthodes agiles ou XP (*eXtreme Programming*).

— Dans un développement par équipe, on communique et collabore, on a donc besoin de mettre en place une organisation (organiser et répartir les tâches, définir les participants, gérer la communication et les normes d'échange, suivre l'avancement...).

3. `https://www.omg.org/about/structure-and-governance.htm`

— La troisième dimension prend en compte le client. Aux éléments précédents on ajoute une dimension *coût* (financier, temporel, humain...) qui fait aussi intervenir une analyse de compétence et de performance (rentabilité). De nouvelles étapes sont nécessaires dans la gestion du projet (opportunité, faisabilité, risques, décisions) ainsi qu'une prise en compte du rôle du client, notamment pour la validation. De manière orthogonale, on doit prévoir dans les projets l'assurance qualité, l'automatisation... La qualité des résultats est liée d'une part à la satisfaction du client et d'autre part à la facilité à maintenir les applications. La qualité du processus se voit dans le respect des délais et des coûts et donc dans sa rentabilité.

— La quatrième dimension met en jeu le prestataire (SSII ou service interne) qui développe ou assure la maintenance corrective ou évolutive (TMA, tierce maintenance applicative). Ses préoccupations sont similaires à celles du client, mais du point de vue de la production : gestion des ressources humaines (carrières, formations...), gestion commerciale (rentabiliser les produits sur plusieurs clients), rentabilité en réutilisant tout ou partie de la production, automatisation du codage...

Un *projet informatique* met donc en œuvre des aspects techniques (méthode de développement), organisationnels (gestion de projet) et méthodologiques (qualité). Il met à contribution différents acteurs avec des objectifs et des visions différentes.

1.1.3 Méthode de développement du logiciel

> Une **méthode** est à la fois une philosophie dans l'approche des problèmes, une démarche (*process*) ou un fil conducteur dans la résolution, des outils d'aide et enfin un formalisme ou des normes.

Dans la méthode Merise [TRC91] la philosophie est fortement inspirée des bases de données, de la programmation structurée et des réseaux de Petri [GV03]. Merise propose une double démarche : par niveau d'abstraction (conceptuel, organisationnel, physique ou opérationnel) et par étapes (études préalables, détaillées, fonctionnelles, techniques et mise en production). Le terme abstraction est synonyme d'éloignement vis-à-vis des considérations matérielles et logicielles (*cf.* Section 1.2.5). Les étapes servent à baliser le développement et exiger des résultats intermédiaires. Elles sont étudiées dans la section B.3.2. Les formalismes tels que Entité/Association/Propriétés, réseaux de Petri colorés, diagramme des flux, schéma de circulation des informations sont utilisés dans Merise [BM88]. Les ateliers de génie logiciel (AGL) tels que WinDev ou AMC designor permettent de saisir des modèles et générer du code.

Dans l'approche à objets [Mey89], de nombreuses méthodes ont été proposées dans les années 80[4] et ont convergé vers les éléments suivants. (1) La philosophie est celle d'un ensemble d'entités autonomes (objets, acteurs...), qui collaborent pour réaliser les fonctionalités du système. Le chapitre 2 de [AV13] détaille ces aspects génériques de la

4. Pour plus de détails, consulter la section 2.3 de [And95].

conception à objets. (2) Un formalisme standard, la norme *Unified Modelling Language* UML [AV13] permet de masquer la diversité des approches, formalismes et outils. *Nous en donnons les principes généraux dans l'annexe C.1.* UML inclut des notations pour les trois catégories d'application que nous avons mis en évidence page 20. (3) Il n'y a pas une démarche unique, mais quatre principes issus de bonnes pratiques de développement : le développement est itératif et incrémental, basé sur les cas d'utilisation et l'architecture. Le *processus unifié* est une proposition de Rational (rachetée par IBM) qui illustre bien cette démarche [RJB99]. Nous détaillons ces éléments du processus dans l'annexe C.2. (4) Les outils couvrent une large gamme de produits, du logiciel de dessin comme Visio à l'AGL complet permettant le *round trip* (lien étroit entre le code et son modèle), le déploiement ou la qualité. *Nous proposons un échantillon dans l'annexe C.3.* Notons enfin les approches qui insistent sur de bonnes pratiques de développement pour améliorer la qualité du logiciel. Par exemple les méthodes agiles centrent le développement sur le besoin par itération avec l'utilisateur. L'*extreme programming* se focalise sur des architectures techniques de qualité pour mettre en œuvre les applications.

Le processus de développement se confond souvent avec le processus logiciel (*software process*). Le second inclut la maintenance. Pour enchaîner les activités du développement, plusieurs cycles sont proposés : linéaire, contractuel, en "V", en "Y", itératifs... Le lecteur trouvera dans l'annexe B, un exposé plus détaillé du concept de méthode (B.1.3) et de processus de développement (B.3) et dans l'annexe C.2 des précisions sur le développement avec UML. Illustrons ici le propos avec le processus *Two Tracks Unified Process (2TUP)*, détaillé en annexe C.2.4. Bien que ce processus ne mette pas en avant le côté itératif et incrémental, il nous semble intéressant d'un point de vue pédagogique car il sépare les analyses en deux aspects : la branche fonctionnelle (métier) et la branche technique, comme le montre la Figure 1.1. Il met ainsi en évidence que dans le développement la phase critique est celle de la conception.

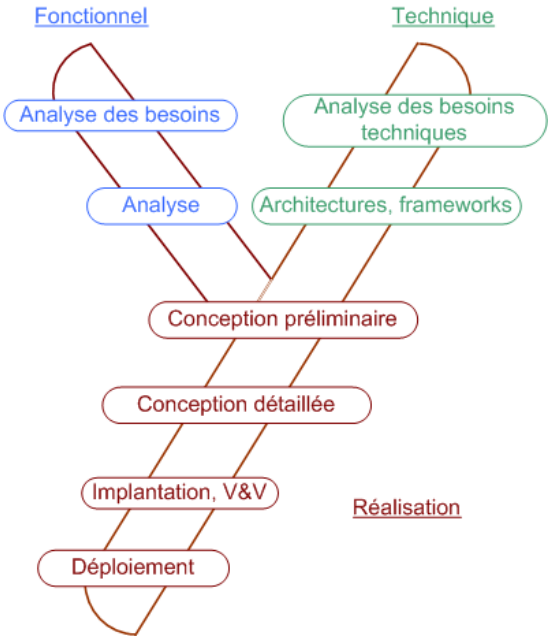


Figure 1.1 – Processus (2TUP) ou en "Y" [RV11]

1.1.4 Génie logiciel

Le génie logiciel vise à répondre aux besoins du développement de logiciel : mettre en œuvre des moyens pour réaliser du logiciel de qualité en respectant des contraintes de coûts et de délais.

Le *génie logiciel* est l'art de construire **industriellement** du logiciel en s'appuyant notamment sur des outils intégrés de type AGL.

"Software engineering is an engineering discipline that is concerned with all aspects of software production" [Som11]. *"Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines"* [Pre10]. Le génie logiciel est arrivé dès qu'on a eu besoin de développer du logiciel un peu conséquent : *"The name 'software engineering' was proposed in 1969 at a NATO conference to discuss software development problems— large software systems were late, did not deliver the functionality needed by their users, cost more than expected, and were unreliable"* [Som11].

Le génie logiciel couvre un large spectre, des langages de programmation à la gestion d'organisations. Il évolue en permanence, étant soumis à l'évolution des technologies et des métiers. Si on se concentre sur l'essentiel, à l'instar d'autre disciplines d'ingénierie, il s'agit d'appliquer une bonne méthode de développement et de bonnes pratiques. L'objectif majeur est de produire du logiciel de qualité, rationaliser le développement, rentabiliser les investissements. Dans ce document, nous considérons principalement les aspects techniques du projet informatique, la méthode de développement (*cf.* Section 1.1.3) [5].

1.2 Des systèmes aux modèles

Jusqu'ici nous avons traité la notion de logiciel comme si elle était indépendante de la réalité, rappelons-nous qu'il s'agit d'un objet logique qui est couplé avec un objet réel, un système, qu'on représentera par des modèles.

1.2.1 Systèmes

L'approche systémique se distingue de la pensée cartésienne pour appréhender des objets de grande complexité. Un objet complexe se caractérise par un nombre important de relations entre les éléments qui le constituent, alors qu'un objet compliqué est caractérisé par un nombre important d'éléments. L'analyse cartésienne s'applique bien au domaine du compliqué, mais mal aux domaines du complexe [LM90].

Selon Joël de Rosnay, un système est un ensemble d'éléments en interaction dynamique organisés en fonction d'un but [DR75]. Un exemple courant de système est donné par une entreprise : les éléments sont les services, les départements... les buts sont "produire", "vendre", "faire du profit"... ; l'interaction est concrétisée par la coopération interne, les relations avec la clientèle et les fournisseurs... Nous préférons la définition plus complète de Le Moigne :

5. Le lecteur trouvera les aspects gestion de projet et qualité dans [Pre10 ; Som11 ; RJB99].

> Un système est défini comme quelque chose (n'importe quoi identifiable) qui fait quelque chose (activité, fonction) et qui est doté d'une structure, qui évolue dans le temps, dans quelque chose (environnement), pour quelque chose (finalité) [LM90].

Deux types de systèmes sont à considérer : les systèmes ouverts et les systèmes fermés. Les seconds n'ont aucune interaction avec l'environnement (milieu extérieur). On ne s'intéressera ici qu'aux systèmes ouverts qui sont en relation permanente avec leur environnement. On les représente par la FIGURE 1.2 [6] avec trois niveaux d'interactions : (1) Un système effectue des transformations d'entrées en sorties (matériaux, produits,

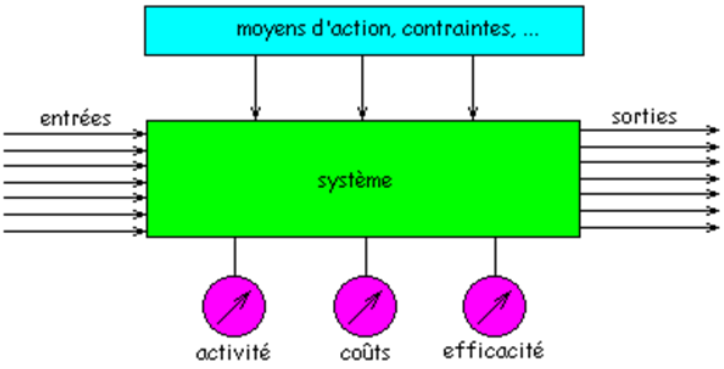


FIGURE 1.2 – Représentation abstraite d'un système

informations...) en fonction de ses finalités. (2) On peut apprécier le fonctionnement du système à partir de variables essentielles (activité, coûts, efficacité...). Le pilote du système doit pouvoir disposer d'instruments de mesure de variables essentielles : les indicateurs. (3) Le pilote doit pouvoir disposer de moyens d'action lui permettant de faire remplir ses missions au système. Par exemple, dans les systèmes de production, on trouve des *Key-Performance Indicators* (KPIs) définis par la norme ISO 22400 [ISO14] (*cf.* Chapitre 5).

Trois propriétés formelles caractérisent les systèmes : (i) la propriété de totalité (le système est composé d'éléments qui constituent un tout cohérent et indivisible. La complexité d'un système dépend autant de la variété de ses composants que de l'interaction entre ces mêmes composants), (ii) la propriété de rétroaction (on possède en entrée des informations sur les sorties, c'est le cycle d'information) et (iii) la propriété d'équifinalité (les mêmes conséquences peuvent avoir des origines différentes, contrairement aux systèmes fermés).

Noter que l'observation d'un système peut se faire par de multiples acteurs et selon de multiples points de vue. Par exemple, pour un produit donné, on trouve le client pour utiliser, l'ingénieur pour concevoir, le technicien pour fabriquer, la logistique pour les pièces ou la distribution, le service commercial pour la vente, le SAV pour la maintenance.

Comme nous l'avons indiqué ci-avant, l'approche systémique prend son intérêt dès que les systèmes deviennent complexes. Pour passer à l'échelle, tout en maîtrisant leur

6. Extraite du cours eMiage A354 proposé avec Alain Vailly et Emmanuel Desmontils `https://www.e-miage.fr/`

description, on a besoins rapidement de système de systèmes, c'est à dire des systèmes dont les éléments sont aussi des systèmes.

La composition hiérarchique permet de structurer fractalement un système en sous-systèmes. La complexité n'est plus dans le concept mais dans l'aggrégation des sous-systemes pour former un tout. On s'appuie pour cela sur l'abstraction et l'encapsulation (*cf.* Annexe C.1.2). On doit alors clairement établir les interfaces des sous-systèmes pour les composer.

1.2.2 Système informatique

Pour traiter automatiquement l'information d'un système quelconque, on a besoin d'un système informatique.

Un **système informatique** est un ensemble de moyens informatiques et de télécommunications, matériels et logiciels, ayant pour finalité de collecter, traiter, stocker, acheminer et présenter des données [Lon17].

Cette définition semble similaire à celle du logiciel. De fait elle inclut le logiciel mais aussi le matériel qui permet d'exécuter le logiciel. Pour classer les systèmes informatiques, nous utiliserons la même classification que celle des applications de la page 20.

L'informatique a intrinsèquement la nature de s'adapter à toute situation, tout domaine d'activité et donc à tout système. Elle est ainsi qualifié d'**ubiquitaire** ou *pervasive*[7]. Sans rentrer dans des considérations philosophiques, l'informatique est incontournable pour tout traitement d'information mais sujette aussi à des limites sociétales telles que le respect des personnes (*privacy*) et la sécurité de leurs données.

Dans la suite, sont abordés principalement les systèmes d'information, que j'aurai tendance à associer à tout type de système, et les systèmes temps-réels ou cyber-physiques, qu'on peut associer aux ingénieurs. Dans les deux cas, on utilisera de l'ingénierie pour les développer. Cela couvre deux des trois catégories de la page 20.

1.2.3 Ingénierie des système d'information

Plusieurs définitions de la notion de système d'information existent. Voici une synthèse extraite de [Rol86] :

"un **système d'information** (SI) est un artefact, un objet artificiel, greffé sur un objet naturel qui peut être une organisation, un processus industriel, une commande embarquée. Il est conçu pour mémoriser un ensemble d'images de l'objet réel à

7. D'un point de vue recherche, c'est un atout pour participer à des projets pluridisciplinaires.

différents moments de sa vie ; ces images doivent être accessibles par les partenaires de l'organisation qui s'en servent pour décider des actions à entreprendre dans les meilleures conditions."

Un SI est en quelque sorte une extension de la mémoire humaine qui amplifie le pouvoir de mémorisation des acteurs de l'organisation et facilite leur prise de décision. Un SI possède diverses facettes. La représentation la plus utilisée est celle des trois cycles de la FIGURE 1.3. (i) Le cycle de vie est le cheminement chronologique du système d'information : conception, réalisation, maintenance, déclin. (ii) Le cycle d'abstraction correspond à plusieurs niveaux, trois en général (Merise) : conceptuel (indépendant de l'organisation et des moyens techniques), logique/organisationnel (dépend de qui, quoi, comment, où), physique (dépend des moyens techniques). (iii) Le cycle de décision représente l'ensemble des mécanismes de décision.

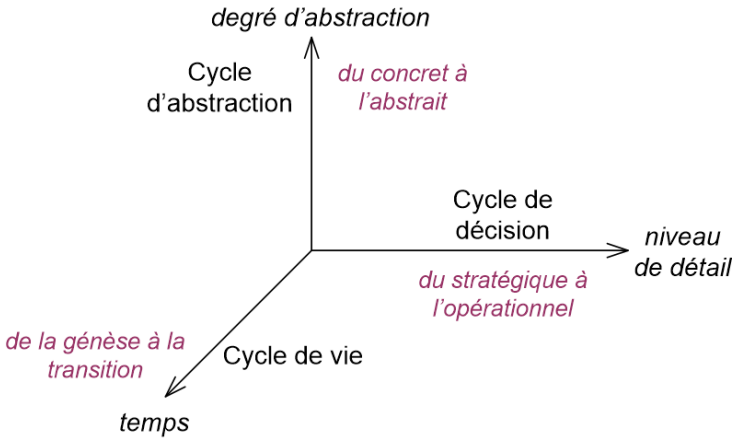


FIGURE 1.3 – Les 3 cycles du SI

Lorsque l'objet naturel est une entreprise ou une organisation, on l'appelle "l'entreprise-système" [LM90]. Ce système distingue trois composantes : (1) le système opérant qui réalise des tâches d'exécution (*e.g.* chaîne de montage d'automobiles, actions de représentants de commerce, établissement de documents administratifs), (2) le système de pilotage pour fixer les objectifs et les moyens et décider à différents niveaux hiérarchiques de l'entreprise, et (3) le système d'information qui est l'intermédiaire entre les deux précédents, chargé de véhiculer l'information interne et externe. La représentation de la FIGURE 1.4 est une bonne illustration de ce qui précède.

L'informatisation du système d'information conduit à distinguer dans la conception d'un système d'information deux niveaux d'étude différents :

— le niveau du système d'information organisationnel (SIO) qui exprime l'activité organisée associée au fonctionnement du système d'information (signification des informations, tâches humaines/informatisées), et
— le niveau du système d'information informatisé (SII) qui ne concerne que le contenu informatisé (logiciel, fichiers ou bases). *C'est le système informatique du SI.*

Une différence entre l'Analyse et Conception de Systèmes d'Information (ACSI) et le Génie Logiciel (GL) est que l'ACSI prend en compte l'impact du système d'information informatisé sur l'organisation. Elle couvre donc un spectre plus large. Néanmoins les méthodes mises en œuvre sont similaires.

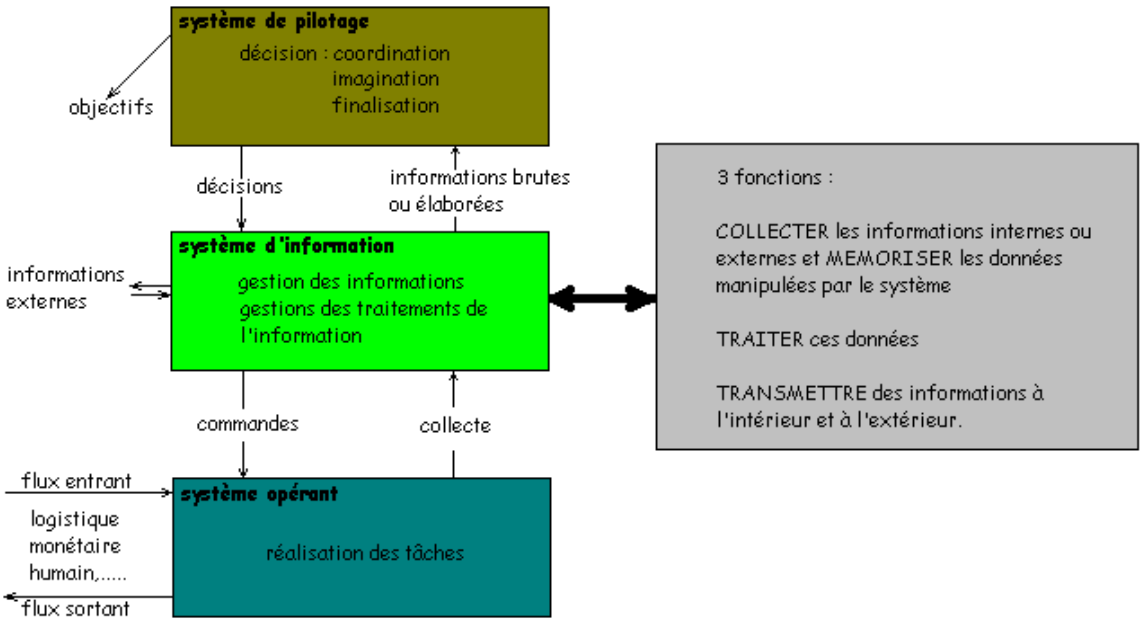


FIGURE 1.4 – Positionnement du système d'information [LM90]

Pour Nanci et Espinasse [NE01], l'ingénierie des systèmes d'information relève plus de l'ingénierie des besoins que du développement, mais pour d'autres [CRS01], on place derrière l'ingénierie des systèmes d'information tout ce qui concerne le développement logiciel des systèmes d'information, se rapprochant alors de l'ingénierie du logiciel.

1.2.4 Ingénierie Système

Autant l'informatique de gestion peut facilement être confondue avec l'ingénierie des SI autant l'informatique industrielle couvre des thématiques nombreuses et variées. On y trouve des automates industriels pour le pilotage simple du matériel, de systèmes embarqués lorsqu'il y a couplage fort entre logiciel et matériel, de systèmes temps réels lorsque la réactivité et souvent la sûreté de fonctionnement sont essentiels, notamment pour des systèmes de supervision, de robotique, de domotique, etc.

À la différence des systèmes embarqués traditionnels, un **système cyber-physique** (*Cyber-Physical System* ou CPS) à part entière est généralement conçu comme un réseau d'éléments informatiques en interaction avec des entrées et des sorties physiques au lieu de dispositifs autonomes en interaction [Lee08] [8]. *"les CPS utilisent des computations et de la communication profondément intégrée et interagissant avec les process physiques afin de produire de nouvelles capacités du système. Un CPS peut être considéré aussi bien à une petite échelle (e.g. pacemaker) qu'à de grandes échelles (un réseau national de distribution d'énergie)"* [Car16].

Dans ce document on s'intéresse plutôt aux systèmes cyber-physiques de production (CPPS) qui rassemblent les outils de gestion automatisée de production. *"Un CPPS consiste en entités et sous-systèmes autonomes et coopérants, connectés au travers d'une*

8. repris de `https://fr.wikipedia.org/wiki/Syst%C3%A8me_cyber-physique`

relation contextualisée, au sein et au travers de tous les niveaux de la production, du process aux réseaux logistiques" [Car16]. Les CCPS participent aux activités telles que la conception, la planification des ressources (matérielles, financières, ou humaines), leur ordonnancement, l'enregistrement et la traçabilité des activités de production, le contrôle des activités de production de l'entreprise.

> *A system is an artifact created by humans that consists of components or blocks that pursue a common goal that cannot be achieved by each of the single elements. A block can consist of software, hardware, persons, or any other units [Wei08].*

L'ingénierie système a la particularité de couvrir divers domaines des sciences de l'ingénieur comme l'automatisme, la mécanique, la robotique, l'électronique, les processus... C'est un peu le domaine des sciences physiques en opposition à celui des sciences naturelles et du monde du vivant. Elle est issue des secteurs aéronautique et défense dans les années 1960.

> *Systems engineering concentrates on the definition and documentation of system requirements in the early development phase, the preparation of a system design, and the verification of the system as to compliance with the requirements, taking the overall problem into account : operation, time, test, creation, cost and planning, training and support, and disposal [Wei08].*

Le génie logiciel joue un rôle d'intégration dans l'ingénierie système. Les méthodes mises en œuvre sont souvent similaires quel que soit le "génie". Comme dans les systèmes d'information (*cf.* FIGURE 1.3), on retrouve les cycles d'abstraction et de vie. Les cycles d'abstractions s'appuient sur les modèles que nous abordons maintenant.

1.2.5 Modèles

Comprendre des phénomènes, expliquer des situations, prendre divers points de vue ou construire progressivement un système sont autant de situations où on a besoin de modèles. En mécanique par exemple, on s'intéresse au mouvement des systèmes matériels et à leurs déformations. Un modèle va alors mettre en évidence les forces en jeu par des équations et la cinématique utilisera des dérivées.

> Selon Calvez [Cal90], un **modèle** (une **représentation**) est une interprétation explicite par son utilisateur de l'idée qu'il se fait d'une situation. Il peut être exprimé par des mathématiques, des symboles, des mots, mais essentiellement, c'est une description d'entités et de relations entre elles.

Un modèle est **correct** s'il permet de répondre aux questions qu'on se pose. Un modèle est **opérationnel** s'il peut être exécuté par une machine.

La recherche d'une bonne représentation lors de la résolution de problème est presque toujours une étape indispensable vers la solution [Lau86]. Il faut donc savoir changer de représentation pour mieux modéliser le problème. La *modélisation* en informatique est le passage du domaine du problème à celui de sa solution informatique. Ses difficultés sont : appréhender la sémantique du monde réel et la tranformer en signes manipulables par les ordinateurs. Le développement du logiciel est une suite de modèles, généralement donnés dans une représentation différente, se rapprochant petit à petit des concepts de la programmation, en suivant le processus qui rationalise le développement.

Pour décrire un système, on considère en général trois axes (*cf.* FIGURE 1.5) : (1) L'axe **statique** ou structurel décrit ce que le système manipule (les informations, les données, les produits...) et son organisation (composants et relations) ; (2) L'axe **dynamique** (ou comportement dynamique) décrit comment évolue le système dans le temps (causalité, contrôle, comportement dynamique, événements) ; (3) L'axe **fonctionnel** (ou comportement fonctionnel) décrit les traitements à réaliser (fonctions, opérations, services, actions).

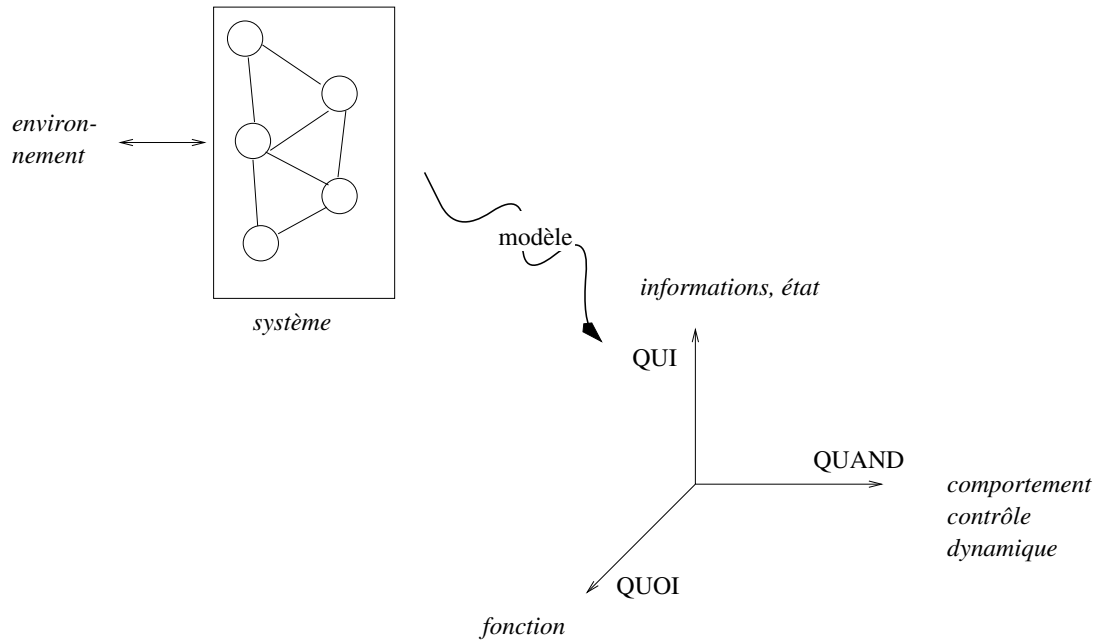


FIGURE 1.5 – Modèles d'un système d'information en trois dimensions

Les méthodes de développement sont parfois classées selon qu'elles privilégient l'un ou l'autre des aspects. Nous illustrons ces tendances dans la TABLE 1.1.

TABLE 1.1 – Décomposition d'un système par axe

statique (données) ⟵		(opérations)		⟶ **dynamique** (traitements)
Entité/ Association	Structures de Données	Langages Formels	Flots de Données	Réseaux de Petri Automates

Une **vue** est une projection sur un ou plusieurs axes. Par exemple, le modèle conceptuel des traitements dans Merise couvre les axes dynamiques et fonctionnels alors que dans

UML on dispose de diagrammes pour chaque axe. Le **point de vue** (*viewpoint*) est une présentation du modèle adaptée au rôle de l'acteur du système qui va le lire (*cf.* page 25).

A ce stade, le lecteur commence à comprendre la complexité de la terminologie. Par abus de langage, le terme **modèle** désigne souvent à la fois le résultat de la modélisation, la théorie sous-jacente et la notation utilisée. Le terme 'modèle' désigne parfois la vue, le point de vue ou le niveau d'abstraction (modèle de conception), de fait un modèle y est composé d'autres modèles. Sachant qu'un modèle sert à communiquer, nous y incluons les commentaires et explications textuelles. Par contre nous ne traitons pas de **modélisation** qui rassemble les règles et bonnes pratiques pour construire des modèles.

Dans une *keynote*, à ISD 2023, Geert Poels définit un **modèle conceptuel** par deux concepts : une **représentation** et une **abstraction** ; le concept est une généralisation d'instances par sélection. "Tous les modèles sont conceptuels avec des préoccupations communes qui les rendent inter-disciplinaires". A notre avis, on doit ajouter à ces deux concepts, ceux d'**intention** (la vision de l'auteur avec son histoire) et d'**interprétation** (la vision du lecteur avec son histoire).

Il est nécessaire à ce stade de discuter du terme abstraction, trouvé un peu à toutes les sauces dans ce document. Dans son dictionnaire encyclopédique du génie logiciel [Hab97], Henri Habrias ne s'y risque pas puisqu'il renvoie vers le non-déterminisme. *"En informatique, le concept d'abstraction identifie et regroupe des caractéristiques et traitements communs applicables à des entités ou concepts variés ; une représentation abstraite commune de tels objets permet d'en simplifier et d'en unifier la manipulation"* Wikipedia[9]. Selon le Larousse, *"l'abstraction est une opération intellectuelle qui consiste à isoler par la pensée l'un des caractères de quelque chose et à le considérer indépendamment des autres caractères de l'objet (sélection) ; son contraire est la réalité"*.

Une **abstraction** est une représentation simplifié de l'objet d'intérêt (*e.g.* un système, un logiciel). *Par exemple, un modèle est une abstraction de la réalité (contraire = concrétisation) ; une classe est une abstraction de ses instances (contraire = instanciation).* Une abstraction peut être plus abstraite qu'une autre (*cf.* annexe B.4.2) *Par exemple, une superclasse est une abstraction d'une classe (héritage, généralisation/spécialisation, sous-typage) ; un modèle est une abstraction d'un autre modèle (contraire = raffinement).* Le terme abstraction, recouvre à la fois le fait d'abstraire et le résultat de ce processus. On parlera d'abstraction/raffinement lorsqu'il s'agit d'un même object vu avec plus ou moins de détail mais d'un seul point de vue par exemple pour le logiciel dans le chapitre 4. On parlera plutôt d'alignement lorsque

9. `https://fr.wikipedia.org/wiki/Abstraction_(informatique)`

le point de vue change entre les niveaux d'abstraction, par exemple entre le métier et l'informatique dans le chapitre 6.

Dans le développement logiciel, tel que décrit dans l'annexe B, une **spécification** est un ensemble de modèles. Les spécifications y sont classées selon leur degré de formalisme : de informelles, standardisées, semi-formelles à formelles (*cf.* Section B.4.1). Il en est de même pour les modèles, qui peuvent aussi être des représentations

— **schématiques**, normalisée ou pas, pour mettre en évidence des aspects spécifiques (plans, dessin technique, coupes, shéma électrique, hydrolique, mécanique...), on peut aussi utiliser des graphiques (*charts*), des cartes mentales... c'est-à-dire tout ce qui peut aider à comprendre.
— **réalistes** pour le *design* (vue 2D, 3D, maquettes d'écrans de saisie,..).

Un programme source est un modèle formel et opérationnel du logiciel. Il reste abstrait vis-à-vis de son environnement d'exécution. La documentation d'un programme (manuel utilisateur ou manuel de référence) contient aussi des modèles pour s'abstraire du code détaillé. On utilisera avec profit les notations UML et OCL pour produire l'information dans le formalisme le plus adapté [AV13].

Plus un modèle est formel et complet, plus on pourra l'exploiter pour répondre aux questions qu'on se pose ou pour tout autre usage.

Les questions qu'on se pose sur un modèle sont de trois ordres :

1. les questions sur le système modélisé - cela dépend bien sûr du système étudié, par exemple est-ce qu'il peut tomber en panne ? est-ce qu'il est sécurisé ? est-ce qu'il est coûteux ? etc.
2. les questions sur le modèle lui-même notamment liées à sa correction : est-il cohérent, complet, lisible, complexe, autosuffisant, modulaire, vérifiable (pour les questions sur le modèle) ?
3. les questions sur la cohérence entre le modèle et le système : dans quelle mesure le modèle est représentatif du système ou dit autrement est-ce que les réponses que me donnent le modèle sont aussi celles du système.

Les questions du premier ordre se traduisent par des **propriétés** (elles même décrites par des langages) et font l'objet d'une **vérification**. Nous traitons les questions du second ordre sous l'angle de la qualité dans la section B.6 de l'annexe B, on peut y répondre par de la mesure ou des techniques de vérification. Les questions du troisième ordre sont relatives à la **validation** et abordées dans la section B.6 de l'annexe B, on peut y répondre par des lectures croisées, du maquettage, de la simulation ou du test de recette.

Nous nous intéresserons dans la suite à la vérification des modèles, mais nous verrons aussi d'autres usages que "répondre aux questions", par exemple la *transformation des modèles*.

Rappelons que pour raisonner sur des systèmes, on en fait une abstraction, ici un **modèle**. Le modèle est exprimé dans un langage (le formalisme, la notation), qui outre les concepts, définit le lexique, la syntaxe et la sémantique pour exprimer le modèle.

Pour raisonner sur des modèles, on en fait (aussi) une abstraction : un **métamodèle**. Un métamodèle est donc une abstraction d'un modèle, c'est-à-dire une image dans laquelle on ne retient que les caractéristiques utiles pour raisonner sur les modèles (comparer des modèles, transformer des modèles). Le métamodèle est exprimé dans un langage. Le langage associé au métamodèle décrit les éléments des modèles et donc des langages de modélisation.

Nous basculons dès lors dans l'ingénierie des modèles.

1.2.6 Ingénierie des modèles

L'**ingénierie des modèles** existe depuis les débuts de l'informatique, et correspond en partie au second rôle du logiciel, *le moyen de produire du logiciel*, énoncé à la page 21. On parle de compilation, d'analyse lexicale ou syntaxique, de contrôle de type et de sémantique des langages de programmation [Mey92]. Noter qu'un code source est une abstraction du code binaire, en ce sens qu'il est indépendant du support d'exécution. Dès les années 1970, on trouve réellement les prémices de l'**ingénierie des langages**, avec Lex et Yacc, des (logiciels) générateurs de compilateurs, qui permettent à chacun de définir formellement son propre langage textuel avec une sémantique opérationnelle définie par génération de code en langage C.

Les modèles se trouvent partout et tout le temps dans les processus de développement. L'**ingénierie des modèles** est apparue en tant que tel au début des annnées 2000 sous l'impulsion de l'*Object Management Group* (OMG) pour lutter contre deux problèmes :

— Cacophonie des notations. Suite de la programmation par objets dans les années 1980, une multitude de méthode d'analyse et conception à objets est apparue [AV13] avec des notations et des pratiques variées et autant d'AGL support. Cette profusion devient un problème pour les équipes de développement pour choisir les méthodes et outils adaptés mais surtout pérennes et aussi pour les éditeurs de solutions AGL. Les éditeurs d'outils de modélisation, les prestataires et les utilisateurs se sont regroupés au sein de l'OMG pour proposer un langage normalisé UML (*Unified Modeling Language*). Dans sa version 1, UML était unifié au sens où la démarche a consisté à prendre ce qu'il y avait de mieux dans les méthodes et à harmoniser la notation qui

est la même pour toutes les étapes du développement [10]. Complété par le langage de contraintes OCL (*Object Constraint Language*) et la sémantique des calculs AS (*Action Semantics*), UML devient le langage universel pour l'ingénierie des modèles.

— Erosion des modèles. Jusque-là, en dehors des méthodes formelles, il y avait une rupture entre les modèles d'analyse et conception et ceux de la programmation. Les premiers servaient essentiellement à communiquer et expliquer les spécifications. Les derniers, opérationnels, pouvaient être analysés par compilation et exécuté par génération de code. Lors de la maintenance, le fossé se creuse puisque les premiers ne sont guère maintenus. L'approche MDA (*Model Driven Approach*) prônée par l'OMG met la notion de modèle au centre du développement logiciel avec la notion de **transformation de modèles**. L'idée, à l'instar des méthodes formelles, est de raffiner progressivement les modèles en ajoutant des détails pour arriver aux modèles opérationnels. Nous la détaillons dans l'annexe C.2.5.

La transformation de modèles fait passer des "modèles pour communiquer" à l'ingénierie des modèles.

Malheureusement, cette approche idéale atteint rapidement ses limites en pratique. La raison principale est qu'un langage même aussi large-spectre ne couvre pas tous les artefacts métiers ou techniques. D'un point de vue métier, on ne traitera pas de la même manière des applications temps réels, des systèmes d'information d'entreprise, des systèmes scientifiques. Par exemple, dans une application temps réel, la gestion du temps, des interruptions et des communications nécessite de nouveaux concepts. Le langage doit alors être étendu ou prendre une sémantique particulière. D'un point de vue technique, UML ne prend pas en compte tous les artefacts de la programmation. Par exemple, pour un développement Web on utilisera des servlets, des scripts, des pages HTML... Ici aussi le langage doit être étendu ou prendre une sémantique particulière. Une autre critique fondamentale est que la diversité des concepts ne permet pas de définir une sémantique formelle. En résumé, UML n'est ni complet ni cohérent. C'est encore plus vrai avec UML 2 qui étend le nombre de diagrammes de neuf à quatorze, à ce jour, avec des recouvrements de notation qui induisent plus de redondances. Toutefois avec UML 2, la vision de l'OMG est différente, UML devient un standard abstrait, chacun personnalisant ensuite, via des profils, la notation et fixant, si possible, une sémantique précise du langage devenu spécifique et appelé DSL (*Domain Specific Language*). Par exemple, le langage SysML [Wei08]proposé par l'OMG est un profil UML dédié à l'ingénierie système. La norme porte sur la notation graphique et des règles de formation des diagrammes (*well-formedness*). La vérification de modèles est un besoin crucial [Unh05 ; Deb+10], nous y reviendrons en section 1.3. Les profils UML sont légion. Revient alors le risque de caco-

10. Ce qui pose d'ailleurs des problèmes d'identification des diagrammes à utiliser pour tel ou tel modèle. En effet, il n'existe pas de méthode standard associée à UML mais des principes communs comme nous le détaillons dans l'annexe C.2.

phonie et la jungle des années 90 ; l'histoire est un éternel recommencement [11].

L'**ingénierie des langages** permet d'instrumenter le besoin de manipuler des langages pour construire un processus personnalisé.

> Quelque part, l'ingénierie des modèles fait entrer les modèles semi-formels et visuels dans le monde de l'ingénierie des langages. Les méthodes formelles sont une forme avancée d'ingénierie des modèles dans laquelle l'espace de modélisation est restreint à une théorie bien fondée mais pour laquelle l'exploitation des modèles est plus riche du fait de la précision des modèles.

Traitement de modèles

La modélisation, qui permet de créer des modèles, peut s'appuyer sur des outils d'édition. Les traitements de modèles répondent à différents besoins.

- La **vérification** permet d'attester des (i) propriétés du système telles que non-redondance, non blocage, sûreté, (ii) propriétés du modèle telles que correction, cohérence, complétude, (iii) transformations telles que alignement, cohérence, complétude. Elle utilise des techniques de preuve, d'inférence, de compilation...
- La **validation** évalue le modèle vis-à-vis du besoin réel par des techniques de lecture, paraphrasage, tests, simulation...
- La **transformation** est un changement de représentation avec ou pas le même langage cible. On peut par exemple (i) raffiner/abstraire pour changer de niveau d'abstraction par ingénierie ou rétro-ingénierie, (ii) implanter/générer du code pour produire une version exécutable du modèle, (iii) extraire/fusionner des points de vue d'un modèle, construire un modèle par composition d'autres modèles, ou enfin (iv) changer de représentation pour communiquer.

La description des transformations peut être déclarative ou impérative. Par exemple le langage ATL [12] utilise le style déclaratif via des *règles de transformation*, le langage Kermeta [13] utilise un style impératif. Pour la transformation de modèles, l'OMG a proposé un standard appelé QVT (*Query/View/Transformation*) [Gro11a], mais cela reste un modèle de langage de transformation avec des implantations plus ou moins proches telles que M2M (Eclipse), ATL, OptimalJ, Together. Dans les méthodes formelles, on peut citer l'Atelier B [14] ou Rodin [15], des outils industriels qui permettent la preuve de modèles et la génération de code ou l'animation.

11. Encore une digression, mais elle me semble important en informatique, où le néologisme est permanent, les *buzz words* font penser que le domaine évolue très vite. Au delà de l'aspect marketing, c'est en partie vrai pour la technologie mais les principes fondamentaux évoluent bien plus lentement. Dans l'enseignement, ce sont ces principes que les étudiants doivent apprendre à maîtriser pour s'adapter aux technologies.
12. `https://eclipse.dev/atl/`
13. `https://www.kermeta.org/`
14. `https://www.atelierb.eu/`
15. `http://www.event-b.org/`

Au niveau des langages, les manipulations précédentes s'appliquent avec des métamodèles à la place des modèles.

- Pour les vérifications, on s'intéresse aux propriétés du métamodèle (cohérent, complet) ou du langage (couverture, sémantique bien fondé, ...).
- La validation n'a pas vraiment de sens au niveau meta. Par contre, on peut faire de la méthodologie et comparer les métamodèles et les langages pour faire de la classification ou de la sélection.
- la **transformation de métamodèles** permet de (i) gérer l'évolution des langages (ex : UML 1.4 → UML 1.5) ou la compatibilité (du moins lorsqu'il n'y a pas eu de rupture dans les langages comme c'est le cas pour le passage de UML 1.5 à UML 2.0), (ii) convertir un modèle d'un langage dans un autre langage (c'est l'approche la plus courante pour faire de la transformation de modèles), produire de la documentation, extraire des points de vue... (iii) composer des langages, c'est, à mon sens, la voie la plus prometteuse pour gérer la complexité car dès que le langage grossit, le métamodèle (et la grammaire si une version textuelle existe) devient ingérable, et (iv) gérer la portabilité, l'interopérabilité, les normes d'échange...

Le lecteur trouvera dans [BCW17] une bonne introduction à l'ingénierie des modèles pour le logiciel (la jungle des acronymes MD* (MDE, MDA, MDD, MBD, MBE, MDSE....) mais aussi la définition de DSL, les transformations de modèles/textes, etc).

> Basculer des modèles aux métamodèles (langages) et vice-versa n'est pas un exercice simple, notamment pour les étudiants. Si tout ce qui est dit au sujet des modèles s'applique aux métamodèles, l'inverse n'est pas vrai. Les métamodèles, étant par nature des modèles, sont décrits par un metalangage avec un méta-métamodèle. Nous sommes au cœur de l'ingénierie des langages.

Pour éviter de boucler à l'infini, en général le méta-langage est défini à partir de lui-même, on dit qu'il est *réflexif*. C'est le cas du *Meta-Object Facility* (MOF), le métalangage commun à tous les langages de l'OMG. Il est le socle de l'architecture en 4 niveaux M3-M2-M1-M0 de la Figure C.8 de l'annexe C.2. Un bel exemple de langage réflexif est le langage Smalltalk, qui permet dynamiquement de créer des classes ou des méthodes (et donc des modèles d'objets) mais aussi des métaclasses. Il est amusant de définir un langage par lui-même, nous le proposons par exemple en exercice dans [AV13] p. 335 et dont est extraite la Figure 1.6. La partie haute décrit partiellement le métamodèle Entité-Association-Propriétés de la méthode Merise. Les flèches pointillées de couleur bleue partent des concepts du modèle (partie basse de la Figure 1.6) vers un concept du métamodèle, ce sont des liens d'**instanciation** qui relie une instance à sont type. Nous n'avons pas représenté le méta-métamodèle car il est ici identique au métamodèle. Par contre nous avons complété la figure originale par un extrait des liens d'instanciation entre métamodèle et méta-métamodèle par les flèches discontinues de couleur verte.

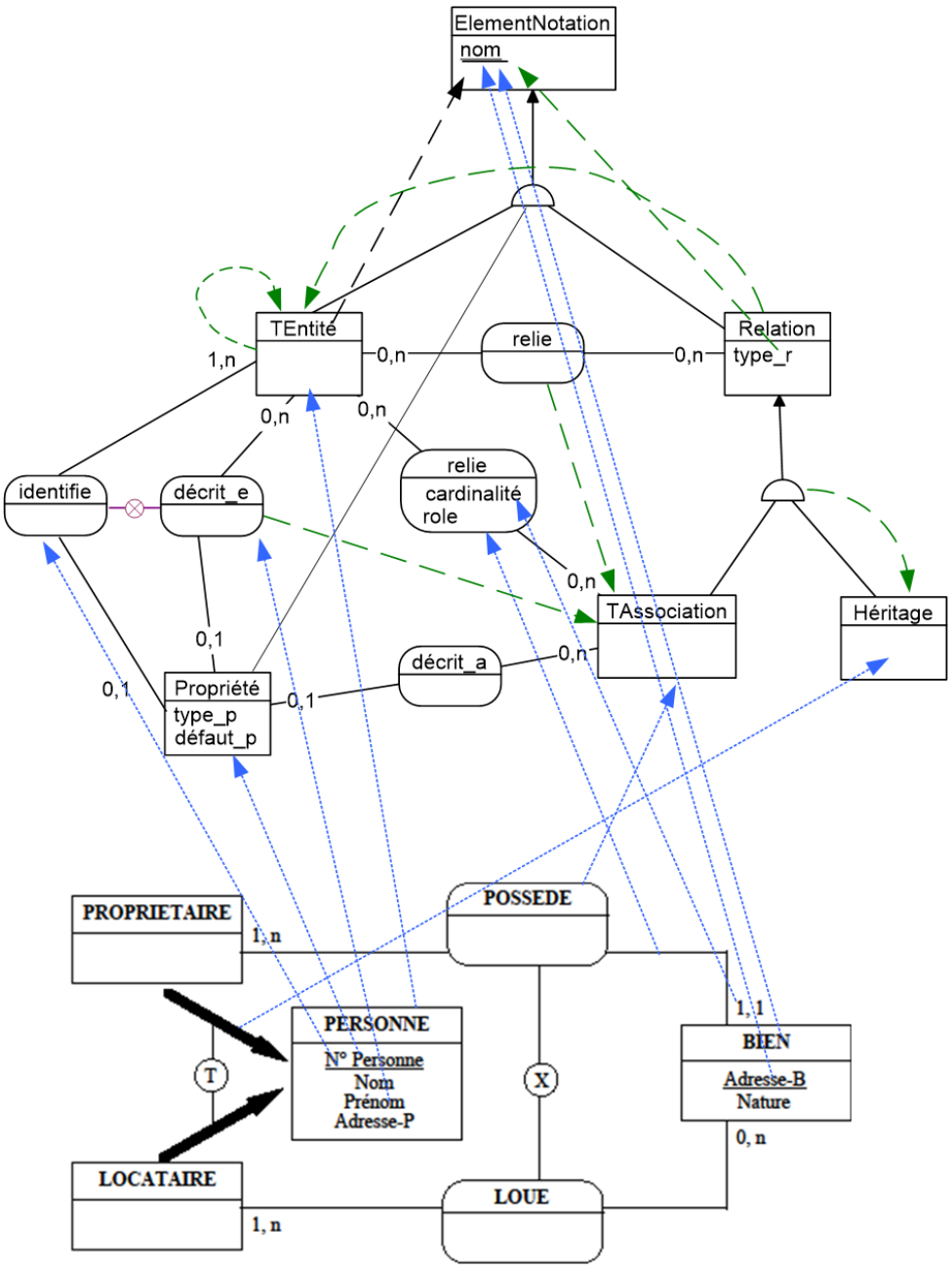


FIGURE 1.6 – Correspondance entre un modèle E-A-P et le métamodèle E-A-P

1.2.7 Rétro-ingénierie des modèles

"Reverse engineering is the process of comprehending software and producing a model of it at a high abstraction level, suitable for documentation, maintenance, or re-engineering." [RS04] La **rétro-ingénierie** dirigée par les modèles (*Model Driven Reverse Engineering* - MDRE [RS04]) vise à produire des modèles de haut niveau d'abstraction à partir de systèmes logiciels. Au delà du consensus général sur le sujet, on retrouve divers objectifs liés à la maintenance des logiciels. Nous en énumérons quelques-uns.

— Extraire les informations de la documentation de conception perdue ou obsolète.
— Comprendre une solution logicielle existante avec des documents manquants.
— Restructurer l'application (*refactoring*) pour améliorer la qualité ou suivre de nou-

velles normes de codage.
— Aligner les processus métier avec les applications existantes.
— Mettre à niveau les versions du cadre technique ou des composants techniques.
— Abstraire des composants logiciels pour les partager avec d'autres projets.
— Rendre plus générique des paramètres codés en dur par des fichiers de configuration.
— Changer la couche 'présentation' ou 'persistance' dans les applications Web n-tier.
— Changer les langages de programmation (*e.g.* de Cobol à Java)...

Nous avons illustré des scenarii dans [And19]. L'approche MDRE ciblera différents niveaux d'abstraction : de la représentation du programme aux architectures d'applications ou aux processus métiers de haut niveau. Par conséquent, différents types de modèles sont attendus avec diverses notations comme les standards MDE *de facto* tels que UML, OCL, MOF, EMF, SysML, AADL, BPMN ou des modèles personnalisés définis avec des langages spécifiques à un domaine (DSL). Les informations sources diffèrent également et peuvent inclure du code binaire, du code source, des fichiers de configuration, des programmes de tests ou des modèles de scénarios...

L'abstraction masque les détails d'implémentation. Lors du développement de systèmes logiciels, les modèles d'abstraction de haut niveau font référence à l'analyse et à la conception du système, tandis que ceux de bas niveau font référence à la mise en œuvre et au déploiement des solutions. Les relations entre les *éléments du modèle* de différentes couches sont de type *raffinement* ou *traçabilité*. Parfois, l'héritage est utilisé pour matérialiser l'abstraction entre des éléments de modèle comparables. On utilise également des **couches d'abstraction** pour représenter l'organisation d'architectures complexes. Des exemples typiques sont la pile ISO de protocoles et de services pour les télécommunications ou l'approche d'architecture de services (SOA).

1.2.8 A propos d'architecture

Le terme **architecture** est, comme le terme modèle, un terme générique, employé dans des contextes différents [16]. C'est un peu le *nec plus ultra* de l'abstraction (*holistic view* dans [Szy02]). Plus un système est complexe plus disposer d'architecture est fondamental pour comprendre/expliquer les fondements du système sans se perdre dans les détails, comme l'indique le praticien Martin Fowler, "*architecture boils down to the important stuff- whatever that is.*" [Fow02].

> Une **architecture** est un modèle abstrait d'un système qui n'est pas contredit par les modèles plus concrets. Selon [Pri12] l'architecture doir répondre aux questions 5W (*why, what, who, where, when*) et à celles du *how*. L'auteur place l'architecture dans le contexte plus large de la gestion de projet.

16. On trouvera dans le chapitre 1 de [Pri12] une discussion sur l'architecture dans les sciences de l'ingénieur.

Architectures techniques

En informatique, l'architecture concerne en premier lieu des ordinateurs et le système informatique. Les étudiants en informatique ont ces cours où on parle de codage de l'information, d'algèbre de Boole, de circuits logiques (c'est le premier niveau d'abstraction), puis de machines de Mealy ou Moore, puis d'architecture type Von Neuman avec une abstraction des composants matériels et le système de base. Puis lorsqu'on passe sur plusieurs processeurs (sans parler de multi-cœur), avec des réseaux de télécommunication, on passe aux systèmes distribués, où l'architecture représente le mode d'organisation (client/serveur, bus, étoile, anneau, maillé, pair-à-pair, n-tiers...). On parlera d'architecture centralisée si un composant a un rôle de coordination et de décentralisée si le contrôle est réparti. Dans UML, le diagramme de déploiement représente une telle répartition. L'architecture en couches permet d'organiser le logiciel en niveaux d'abstraction, là encore pour maîtriser la complexité mais aussi l'hétérogénéité. La virtualisation est un pas supplémentaire, dans le Cloud par exemple, pour s'abstraire de l'organisation matérielle, et finalement découpler matériel et logiciel pour que chacun puisse évoluer indépendemment de l'autre.

L'intergiciel (*middleware*) est la couche logicielle, et donc le logiciel, qui met en œuvre les mécanismes de communication, d'interopérabilité entre applications *e.g.* messages, appels distants, objets, transactions, composants, services. L'intergiciel a été étendu aux réseaux mobiles et à l'internet des objets (IoT) où la distance et la mobilité sont abstraits.

> Toutes ces formes d'architectures, et la liste n'est pas exhaustive, seront rangées dans la catégories `architectures techniques`, c'est-à-dire les architectures matérielles et logicielles qui mettent en œuvre les applications. L'intergiciel est, en quelque sorte, l'architecture la plus abstraite du système informatique.

Dans un projet informatique, l'analyse technique[17] consiste à produire ces architectures techniques. Un *framework* est une solution technique qui met en oeuvre une architecture, en général technique, que chacun pourra personnaliser à son besoin pour former une architecture applicative.

Architecture applicative et conception architecturale

L'architecture applicative est le plus haut niveau d'abstraction des architectures logicielles et techniques dans une vision en couche. *"Application architectures encapsulate the principal characteristics of a class of systems"* [Som11]. Le terme **architecture applicative** est plus utilisé en architecture d'entreprise ou en conception architecturale qu'en "architectures logicielles".

La conception architecturale est la première étape de la phase de conception d'un système. Pour le logiciel, c'est la conception préliminaire, appelée aussi conception générale (*cf.* l'annexe C.2). *"Architectural design is concerned with understanding how a system*

17. Etape bien mise en évidence dans le processus en 2TUP - voir l'annexeC.2.4 .

should be organized and designing the overall structure of that system. [...] You can design software architectures at two levels of abstraction, which I call architecture in the small and architecture in the large" [Som11]. On considère donc l'ensemble des applications et individuellement chacune des applications.

Pour ne pas réinventer la roue, on s'appuie sur des **patrons d'architecture** (*architectural pattern*) appelés aussi styles architecturaux. Il ne faut pas les confondre avec les **patrons de conception** (*design patterns*) qui servent à structurer les composants des applications [18]. Citons quelques exemples de patterns architecturaux [Som11 ; Pre10] : *blackboard (data store, repository), model-view-controler (MVC), pipes and filters, layered, call and return, client-server...* Ces patrons sont abordés en détail pour les applications d'entreprise, les systèmes d'information, dans [Fow02] .

Architectures logicielles

Dans son ouvrage sur les architectures logicielles [Pri12], Jacque Printz met en évidence la difficulté à définir ce concept. Il y discute la définition originelle de Garlan et Shaw [SG96] (un ensemble de composants reliés par des connecteurs) qu'il trouve restrictive et celles de Booch ou Boehm plus proches du développement logiciel avec des participants, des décisions et des invariants. De fait, en UML on pourra définir l'architecture par un diagramme de composants et compléter par des diagrammes de comportements (états-transitions, activités) et d'autres informations.

Prenons les définitions de Bass et al [BCK12] sur l'architecture logicielle et l'architecture de systèmes. *"The **software architecture** of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both"* [BCK12]. Les propriétés peuvent être structurelles, extra-fonctionelles (*cf.* Section 1.3) ou analogiques (*e.g.* patrons) et le modèle de l'architecture peut comprendre plusieurs modèles pour couvrir les axes structurels, dynamiques et fonctionels d'un système (*cf.* FIGURE 1.5) [Pre10].

*"A **system's architecture** is a representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and a concern for the human interaction with these components. That is, system architecture is concerned with a total system, including hardware, software, and humans"* [BCK12]. Dans UML, on peut les représenter avec les diagrammes de déploiement ou de composants (*cf.* l'annexe C.1.3). On notera aussi la possibilité, pour les systèmes complexes de définir des sous-systèmes. *"Another very basic concept of architecture is the decomposition of a larger system into a number of subsystems that are composed and provide this way the behavior of the overall system. We speak of a subsystem or of a component architecture"* [Bro18].

Les architectures logicielles ont été un thème de recherche actif dans les années 1990. L'intérêt des travaux de Garlan & Shaw est d'initier le raisonnement sur les architectures

18. La confusion est facile car certains patrons comme dans le MVC sont dans les deux catégories.

à composants logiciels avec des modèles plus formels. De nombreux travaux en sont issus tels que les architectures à composants *Component based software engineering* [CL02 ; Szy02 ; BCK12], les architectures à services *Service Oriented Architecture*) [Pap03 ; Erl05] mais aussi les langages de descriptions d'architecture logicielles (*Architecture Description Languages -ADL*) [Cle96]. Une comparaison des ADLs est proposée dans [MT00]. La plupart perçoivent l'architecture comme un ensemble de composants avec des connecteurs, quelques-un intègrent des propriétés dynamiques, la plupart font de la vérification et certains intègrent le raffinement ou la génération de squelettes. Peu d'entre eux intègrent les propriétés non-fonctionnelles.

Les ADLs ont l'avantage de fournir un niveau de description uniforme, abstrait et souvent vérifiable. La difficulté est de les connecter aux modèles de conception détaillée et aux applications "concrètes". Certains langages décrivent l'architecture comme une structure au dessus du code. C'est le cas de FractalADL [Lec+07] qui définit une couche composants sur un support Java appelé Julia [Bru+06] ou de Service Component Architecture (SCA) du consortium OASIS qui propose une couche service sur le code [Pai+17]. Il se pose la difficulté de la co-évolution, comme dans le *round-trip engineering* [Het10] que supportent des outils de modélisation qui permettent de remonter des éléments de code vers les modèles pour ne pas les perdre à la génération de code suivante. Le standard AADL (*Architecture Analysis and Design Language*) [19] permet de s'attaquer aux propriétés des architectures. Il peut ainsi s'intégrer dans une démarche dirigée par les modèles [FG12].

Notons enfin, la nécessité de définir précisément, à défaut de formellement les interfaces entre composants et les contrats de service pour pouvoir assembler de manière correcte et fiable les éléments d'architecture, ceci est abordé dans [Mey03 ; MAA10b].

Architecture d'entreprise

Lorsqu'on prend un contexte plus large que celui du logiciel ou même du système, tel que celui des organisations, on trouve le domaine des architectures d'entreprise.

L'urbanisation permet de cartographier et de fédérer les architectures applicatives des systèmes d'information [Lon09 ; UE10 ; RC11]. À plus haut niveau d'abstraction, on trouvera les processus métier [DM04 ; BFGM11], les *Business Process Models* (BPM) et l'architecture métier [DR12] au cœur des préoccupations de la gestion de systèmes d'information [Mor12]. L'annexe C.1.4 discute plus en détail du concept d'architecture dans le contexte UML.

L'**architecture d'entreprise** comprend tout cela et plus encore [Pep16]. *"Enterprise architecture is a description of the structure and behavior of an organization's processes, information flow, personnel, and organizational subunits, aligned with the organization's core goals and strategic direction"* [BCK12]. Elle vise à modéliser le système d'information de l'entreprise pour en contrôler l'évolution [Cap11]. L'urbanisation cible plus spécifiquement le système d'information, et la démarche d'urbanisation vise à en améliorer le

19. `http://www.aadl.info/`

fonctionnement en le rationalisant dans un "cercle vertueux de transformation et d'amélioration continue" [UE10]. Pour Y. Caseau [RC11], l'urbanisation est en premier lieu une démarche technique d'accompagnement du SI utilisant des principes simples (décomposition, découplage, intermédiation) pour répondre à des objectifs de flexibilité, de mutualisation, de maintenabilité, etc.

Archimate [20] est un langage de modélisation pour les architectures d'entreprise dans le contexte plus général du standard TOGAF (*The Open Group Architecture Framework*) d'architecture d'entreprise proposé par The Open Group. Un outil open-source appelé archi est proposé [21] même si l'approche est intégrée à différents AGL du marché.

Notons enfin l'existence de la norme ISO 42010 [ISO11], une norme pour la description des architectures de systèmss ou de logiciels. *"An architecture description language (ADL) is any form of expression for use in architecture descriptions. An ADL provides one or more model kinds as a means to frame some concerns for its audience of stakeholders. An ADL can be narrowly focused, defining a single model kind, or widely focused to provide several model kinds, optionally organized into viewpoints. Often an ADL is supported by automated tools to aid the creation, use and analysis of its models. Rapide, Wright, SysML, ArchiMate [...] are ADLs in the terms of this International Standard"* [ISO11].

1.3 Vérification de propriétés

Nous avons, depuis le début de ce chapitre, mentionné plusieurs fois la vérification et la validation (V&V) au sujet des systèmes, des modèles et des propriétés. *"V&V aims at providing a significant level of confidence in the reliability of a system"* [Deb+10]. Cette section vise à montrer la complexité, la diversité et la subjectivité du sujet de la vérification de propriétés. Précisons et éclaircissons un peu le propos car c'est un sujet important dans ce document mais aussi dans le développement de systèmes *"V&V can represent 50-80% of the total design effort"* [Deb+10].

Valider c'est contrôler que le produit, le résultat, correspond à ce qui était attendu. Vérifier c'est contrôler que le produit respecte le cahier des charges [Som11 ; Pre10 ; Gau+96]. La validation est un contrôle qui fait intervenir largement les "utilisateurs". Ainsi, dans [Unh05], l'auteur fait intervenir trois catégories de contrôles (*checks*) syntaxe/sémantique/esthétisme dont l'importance est inversée quand on parle de vérification (*syntax-first*) ou de validation (*aesthetic first*). La vérification est donc plus une "affaire" de spécialistes et nécessite une spécification [précise] [Gau+96] [22] Comme indiqué dans l'annexe B.7, les deux sont liés à la qualité des produits (les modèles, spécifications, logiciels...) ou à la qualité du processus, déterminée par des *Key Performance Indicator (KPI)*. Plus la détection de non-qualité (ou d'erreurs) est tardive, plus elle est coûteuse.

20. `https://www.opengroup.org/archimate-forum/archimate-overview`
21. `https://www.archimatetool.com/`
22. Dans son dictionnaire encyclopédique sur le génie logiciel [Hab97], Henri Habrias consacre cinq entrées à la vérification sans être générique.

La qualité est déterminée par des indicateurs (attributs, facteurs, caractéristiques) définis par aggrégation propriétés élémentaires qui peuvent être mesurées par des métriques). Elle est aussi déterminée par la vérification de propriétés.

Nous ne traitons ici que de la vérification et pour simplifier, on considère que ce qui est à vérifier est une **propriété** (QUOI) qui doit être vraie ou fausse sur un modèle (OÙ). La technique de vérification (COMMENT) permet de déterminer la valeur de la propriété à un moment donné.

Cette vision est cohérente avec

— la "vérification système" de l'ingénierie système de la page 29 (*compliance with the requirements, taking the overall problem into account : operation, time, test, creation, cost and planning, training and support, and disposal*).

— la "vérification de modèle" qui répond aux questions des trois ordres qu'on se pose sur les modèles de la page 32 et des techniques qu'on met en œuvre pour y répondre dans les transformations de modèles de la page 35.

Remarque

Nous conservons ici une définition générale de la **vérification de modèles** alors que *model checking* rassemble plutôt un ensemble de techniques autour des automates et des logiques temporelles, ou que *statistical model checking (SMC)* relève de la simulation sur un échantillon contrôlé.

Dans la section 1.2.6, nous avons vu que les modèles (OÙ) sont exprimés par des langages et que les propriétés peuvent être liées au système modélisé, au modèle voire au langage de modélisation. Pour être manipulables, les propriétés (QUOI) doivent être formalisées par un langage basé sur une **logique** (propositions, prédicats, temporelle, ...). Muni d'un système d'inférence (de raisonnement), ce langage constitue un **système formel**. Consulter [AV01b] pour une introduction aux systèmes formels et aux méthodes formelles pour spécifier des systèmes et en vérifier les propriétés. La vérification formelle des propriétés (COMMENT) se fait par des approches plus ou moins automatisées. Citons.

— **Contrôle** (le *check*) : lister les contrôles à effectuer est une approche classique des processus qualité. Unjelkar [Unh05] propose ainsi des listes de vérification [23] à faire sur les diagrammes UML.

— **Preuve** : on démontre (éventuellement sous hypothèse) que la propriété est vraie soit (i) de manière absolue (*theorem proving*) qui consiste à définir les propriétés par des théorèmes et à les prouver en utilisant un prouveur automatique qui assiste [24]

23. Encore faut-il comprendre les controles à établir, car il n'est pas toujours précisé le détail de comment le faire.

24. Le lecteur trouvera dans la partie 1 de [AV01b] un exposé détaillé avec la méthode Z basée sur la théorie des ensembles et la logique des prédicats et pour laquelle on prouve des assertions (invariant, pre/post conditions) en utilisant l'assistant de preuive Z/Eves.

et (ii) dans chaque état du (modèle du) système (*model checking*) [25].

— **Test** : on prend un échantillon des valeurs possibles pour le modèle et on montre que sur cet échantillon la propriété est vérifiée. Evidemment, la crédibilité réside dans la représentativité de l'échantillon. Le test du logiciel est abordé dans l'annexe B.7.1. On notera ici la différence entre *Model Testing* qui consiste à tester le modèle et *Model Based Testing (MBT)* qui consiste à tester le code en s'appuyant sur le modèle [AMA13 ; Mot+19]. [UPL11] propose une classification des approches de MBT, reprise dans la FIGURE 1.7.

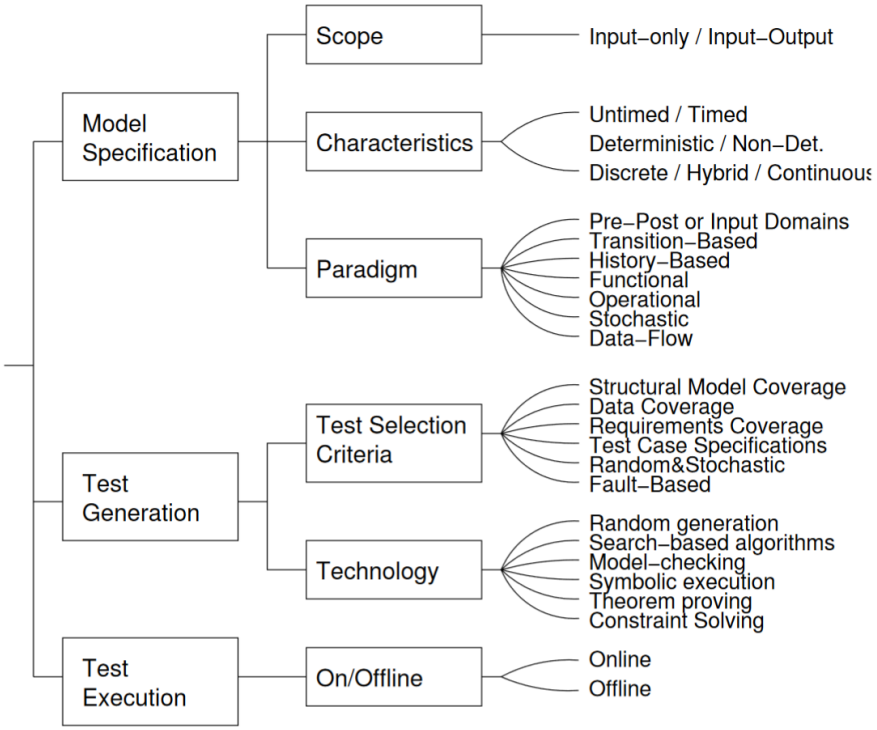


FIGURE 1.7 – Taxomonie des approches de *Model Based Testing* [UPL11]

Debabbi et al. classent les techniques de V&V en quatre catégories : informelles, statiques, dynamiques et formelles [Deb+10]. La **simulation** n'est pas une approche de vérification en tant que tel mais une technique qui s'utilise dans le test ou la preuve. Il existe de nombreuses variantes selon les logiques utilisées, on combine aussi avec des probabilités, des statistiques ou des contraintes [BK08 ; AP18 ; Del20].

Remarque

Chaque approche a ses avantages et inconvénients, certaines conviennent mieux à tel ou tel aspect du du système, on peut combiner les approches pour vérifier des propriétés très différentes [AAM17].

La classification ci-dessus distingue les techniques de vérification. Il existe une autre

25. Le lecteur trouvera dans le chapitre 5 de [AV01a] la description de propriétés pour la dynamique das systèmes et leur vérification par model checking et dans [B+99] un tour d'horizon du sujet.

classification qui distingue les **analyses statiques**, sans évaluer du système, et les **analyses dynamiques**, par évaluation du système. L'analyse dynamique implique un modèle opérationnel du système (voir p. 29). Pour une plus large couverture, certaines techniques de vérification combinent les deux [Ern03].

Propriétés et typologies

Pour un système [logiciel [26]] quelconque, on sépare les **exigences fonctionnelles** (les fonctions que doit remplir le système) et les **exigences non-fonctionnelles** (la manière de remplir les exigences fonctionnelles - la qualité de service (QoS) appelées aussi extra-fonctionnelles. *"Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. In principle, the functional requirements specification of a system should be both complete and consistent. [...]* Non-functional requirements*, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements"* [Som11]. Vu sous cet angle, pour vérifier le (modèle du) système, on peut séparer les propriétés fonctionnelles et non-fonctionnelles. On notera que les propriétés non-fonctionnelles s'appliquent à l'ensemble du système alors que les propriétés fonctionnelles sont plus "modulaires".

Remarque

Malgré leur nom, les propriétés fonctionnelles, issues des exigences fonctionnelles, s'appliquent aux trois axes d'un système (*cf.* Figure 1.5), pas seulement à l'axe des fonctions.

En croisant les taxonomies des propriétés non-fonctionnelles de [Som11] et de [AKS16], disponibles dans l'annexe B.6.4, nous nous hasardons [27] à proposer la typologie des propriétés à vérifier de la Figure 1.8. Cette hiérarchie montre la complexité de la question.

Intéressons nous aux propriétés "fonctionnelles" associées aux axes d'un système. On peut démontrer par exemple (i) la non-redondance d'informations via des analyses de dépendances (comme les formes normales de bases de données relationnelles), la cohérence des invariants par de la logique [statique] ; (ii) le non-blocage, la vivacité, l'équité ou la causalité par du *model-checking* [dynamique] [28] ; et (iii) la cohérence des traitements par des assertions (invariant, pre/post conditions) [fonctionnel]. Noter que beaucoup de propriétés sont à décrire explicitement et spécifiquement sur le système étudié, alors que les propriétés de modèles sont plus génériques (correction, cohérence, complétude).

Un certain nombre de propriétés non-fonctionnelles sont des qualités et à ce titre

26. On ne traite pas de la vérification de la partie matérielle dans ce document.

27. Nous n'avons pas trouvé de références qui traitent le sujet dans son ensemble. Ce qui ne veut pas dire que ça n'existe pas.

28. Dans [B+99], les propriétés sont classées en quatre groupes : atteignaibilité, sûreté, vivacité et équité. L'absence de blocages est à la fois de la sûreté et de la vivacité.

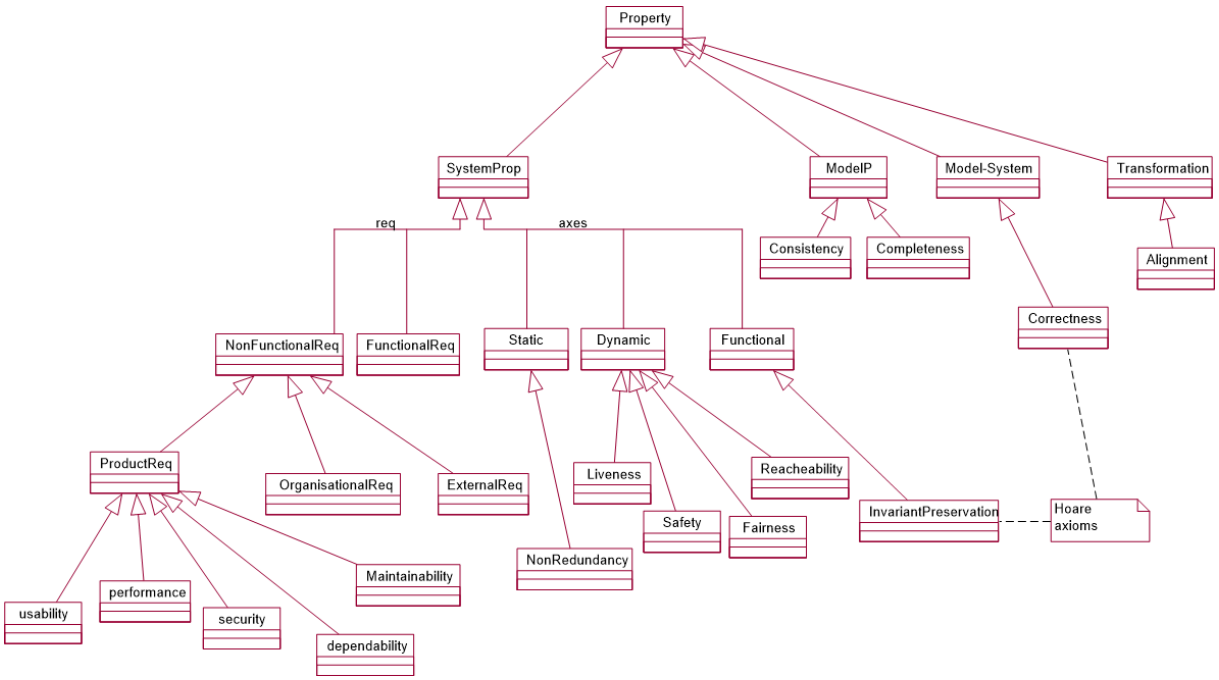


FIGURE 1.8 – Petite taxomonie des propriétés

elles sont structurées en attributs [29] et se mesurent (ou s'estiment) par des arbres de **métriques** plus qu'elles ne se vérifient [30], c'est le cas de la performance, la maintenabilité, l'évolutivité... Par contre, pour la sécurité, on utilisera plutôt la vérification alors que pour la fiabilité on utilisera les deux approches. Nous invitons le lecteur à consulter la thèse de Martin Monperrus [Mon08] sur la mesure des modèles par les modèles pour construire son propre modèle de propriétés non-fonctionnelles.

Il n'en reste pas moins que toute classification est discutable. Certaines qualités s'expriment par des propriétés fonctionnelles et non-fonctionnelles. Citons une fois de plus Sommerville au sujet de la sûreté de fonctionnement et la sécurité : *"System dependability does not just depend on good engineering. It also requires attention to detail when the system requirements are derived and the inclusion of special software requirements that are geared to ensuring the dependability and security of a system. Those dependability and security requirements are of two types : (1) Functional requirements, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks. (2) Non-functional requirements, which define the required reliability and availability of the system "* [Som11].

Revenons sur une propriété importante des modèles, définie informellement à la page 29, par "*Un modèle est **correct** s'il permet de répondre aux questions qu'on se pose*". La correction (*correctness*) est une propriété fondamentale en génie logiciel. Par exemple, un algorithme est correct vis-a-vis de sa spécification s'il produit les mêmes résultats pour les mêmes entrées et mêmes conditions. Difficile à définir de manière générique [31].

29. En qualité logicielle, on parle aussi de facteurs et critères - voir page 316.
30. Relire la discussion de la page 1.2.5.
31. Dans [Hab97], la correction est le respect d'une spécification qu'il faut prouver.

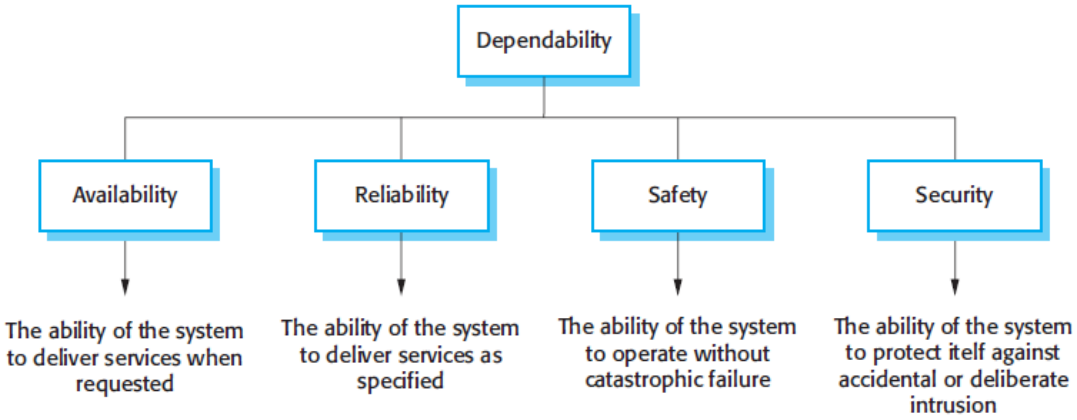


FIGURE 1.9 – Sous-taxomonie NFR - dependability [Som11]

C'est un ensemble de propriétés définies en fonction de leur contexte. Pour un modèle, on s'intéresse plus à la cohérence logique (*consistency*) et la complétude. Lorsque plusieurs vues établissent le modèle, *e.g.* les diagrammes d'UML, on vérifie que les éléments d'un diagrammes ne contredisent pas ceux d'un autre [Unh05]. Dans les méthodes formelles, elle s'applique non seulement au modèle mais aussi à son raffinement "*correct by construction*".

Bien que ce soit une évidence, il est important de rappeler que plus le modèle est précis plus la vérification donnera de la confiance sur la fiabilité du système, ainsi la correction de modèles dans l'approche MDE nécessite une étape de formalisation [GC14].

Dans le contexte des architectures, Shaw and Garlan [SG96] proposent une classification des propriétés à spécifier pour la conception architecturale (du logiciel) en trois catégories. (1) Les propriétés structurelles définissent les composants d'un système (*e.g.* modules, objets, filtres) et leurs connexions et interactions, (2) Les propriétés extra-fonctionnelles indiquent comment l'architecture de conception répond aux exigences de performance, de capacité, de fiabilité, de sécurité, d'adaptabilité et d'autres caractéristiques du système. (3) Les familles de systèmes associés définissent les propriétés de patrons reproductibles. Ainsi définies, les propriétés structurelles restent implicites car ce qui est mentionné correspond au modèle. Il est intéressant de voir des propriétés extra-fonctionnelles au niveau architectural et donc perçues comme des abstractions. L'idée de famille (ou de patron) est à mentionner non seulement pour les bonnes pratiques mais aussi pour factoriser la vérification de propriétés à un ensemble de systèmes similaires.

1.4 Conclusion

Dans ce chapitre, j'ai introduit l'environnement dans lequel se situe ma recherche, l'ingénierie logicielle pour les systèmes. Ce contexte peut paraître un peu général, mais je considère qu'à l'instar de la médecine, on a besoin de généralistes et de spécialistes. Néanmoins je ne suis pas entré dans les détails et les deux chapitres annexes B et C viennent compléter le propos.

À la lecture de ce chapitre, le lecteur aura compris qu'on se place dans le monde de

l'ingénierie du logiciel dans laquelle la connaissance ou même la science ne sont pas les seules finalités. Il s'agit d'améliorer les processus, techniques et outils mais aussi de former et diffuser. C'est bien le rôle d'un enseignant-chercheur de ne pas séparer les mondes de la recherche, de l'enseignement et des sociétés.

Le lecteur impatient pourra se dire que tout ce qui est écrit dans ce chapitre est connu. Néanmoins, tous les lecteurs de ce manuscrit ne sont pas des experts et il m'apparait aussi nécessaire, pour éviter toute confusion, de fixer les concepts du domaine de recherche.

La FIGURE 1.10 est un "modèle" simple de la terminologie de ce chapitre. Il a été généré par IRaMuTeQ [32]. Ce type de représentation met en évidence les concepts et leur poids dans le domaine mais pas les liens entre concepts.

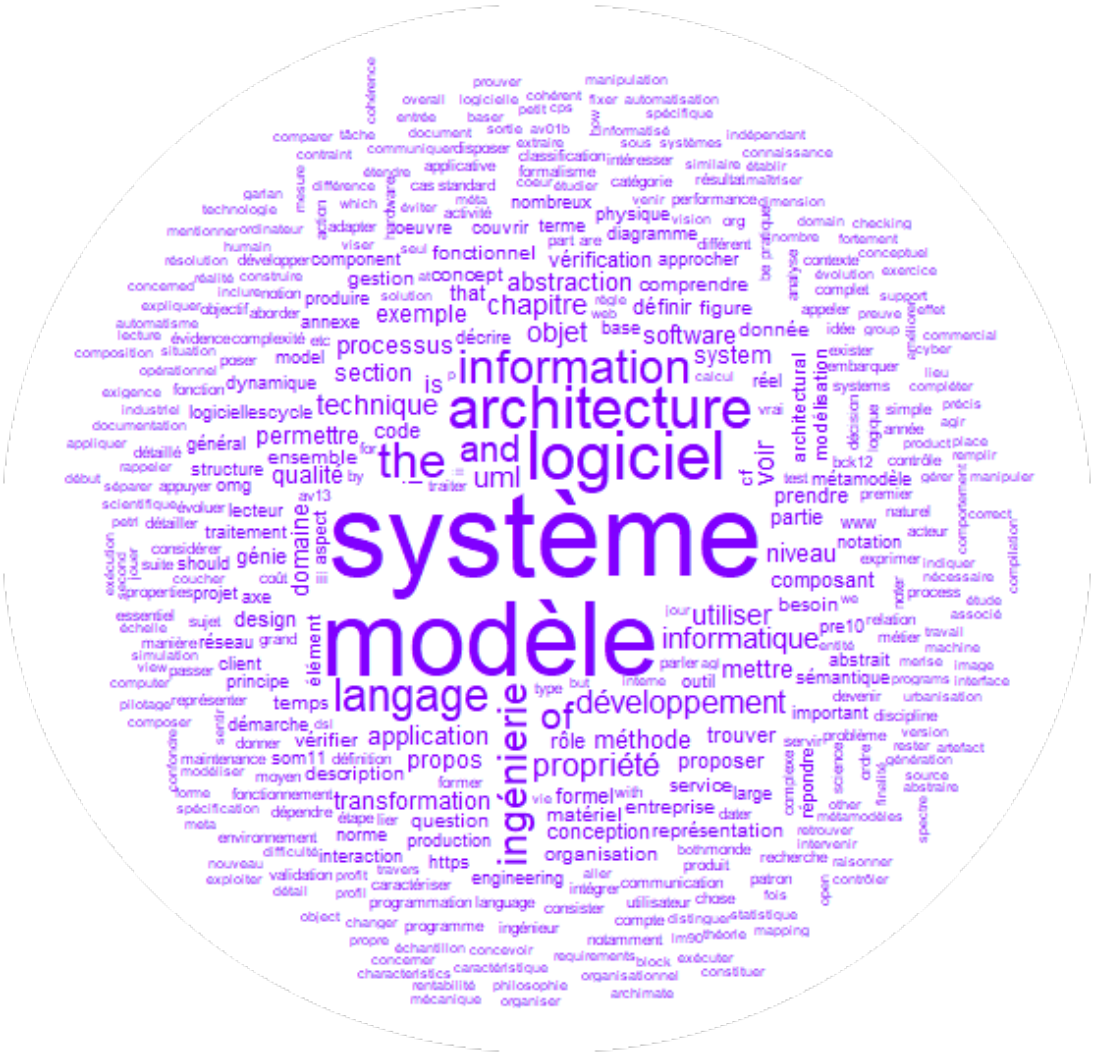


FIGURE 1.10 – Terminologie du chapitre 1

Pour établir des liens on peut représenter le domaine de recherche par une carte mentale (*mindmap*) ou par un modèle du domaine (*domain model*) via un modèle E-A-P ou un diagramme de classes UML par exemple ou bien un domaine de connaissances via une ontologie. Pour ce chapitre j'ai choisi la carte mentale de la FIGURE 1.11. Les nombres indiqués sur les nœuds representent le nombre d'occurrence des termes dans ce chapitre.

32. `http://www.iramuteq.org/`

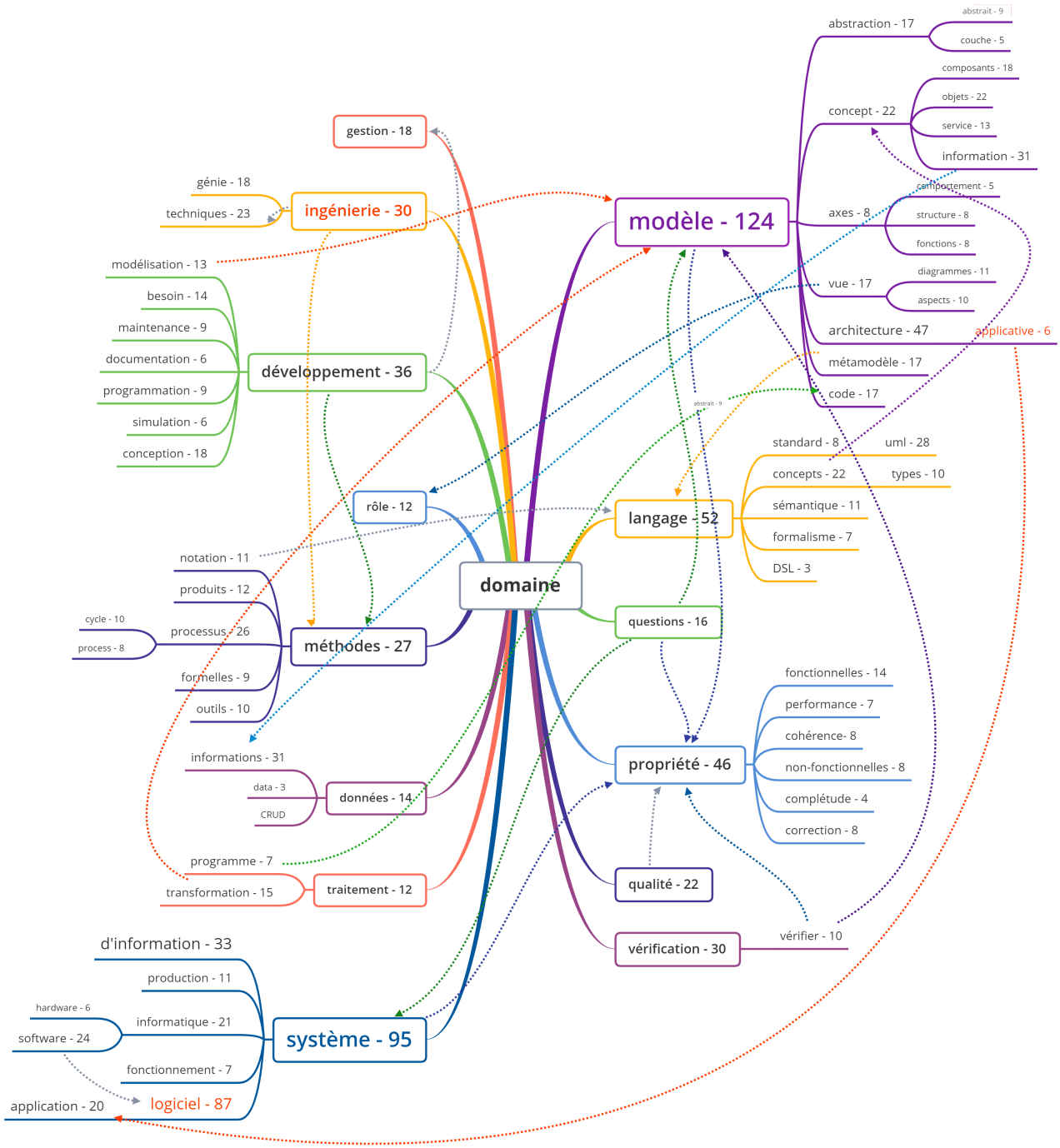


FIGURE 1.11 – Modèle simplifié du domaine du chapitre 1

Le logiciel joue un rôle prépondérant à l'ère du numérique, *"Software Is Eating the World"* [Bro18]. De par la complexité de sa modélisation, les travaux sur la modélisation des systèmes logiciels amènent à des réflexions et des pratiques qui se diffusent dans l'ingénierie système mais aussi dans d'autres domaines. *"Modeling should be an independent scientific discipline"* [CV22]. Je partage ces idées de croisement avec d'autres disciplines (pluridisciniplaire ou transdisciplinaire) [33].

33. D'où le titre de la partie dans laquelle se trouve ce chapitre.

Dans le chapitre suivant, intitulé *Mythes et réalités* je viens positionner quelques défis (*challenges*) auxquels je m'intéresse.

Post-scriptum

Ce chapitre met en évidence des acquis sur les compétences suivantes, que nous synthétiserons dans le chapitre 8.

C1

Comprendre, structurer (classifier) et expliquer le domaine de recherche sont des compétences de base pour positionner un travail de recherche.

Chapitre 2

Mythes et réalités

Chaque découverte scientifique est passionnante parce qu'elle ouvre un univers de questions. Si les questions vous angoissent, ne soyez pas scientifiques.

Boris Cyrulnik

Ce chapitre aurait pu s'appeler *Enjeux, défis, verrous scientifiques et perspectives* mais c'est trop sérieux, la recherche n'est pas toujours une science et pour être ambitieuse elle doit s'attaquer à des mythes, qu'on pourra toujours raffiner en défis dans des projets. Si la recherche doit rester cohérente et avoir du sens, il ne faut pas toujours lui donner 'un' sens (une direction) car la découverte vient de ce qu'on ne connait pas, souvent du hasard. L'innovation vient de la rupture. La recherche c'est aussi ouvrir des portes et savoir qu'on ne pourra les refermer.

2.1 Introduction

Dans le chapitre 1, nous avons posé le domaine autour des systèmes logiciels et du génie logiciel pour les mettre en œuvre, notamment en utilisant des modèles pour raisonner sur ces objets. C'est un domaine vaste avec de nombreuses problématiques. L'étude tertiaire de Khan et al. [Kha+19] fait un point sur les sujets de génie logiciel faisant l'objet d'études systématiques, dont un des objectifs est d'identifier les problèmes majeurs. Il en ressort que les études les plus nombreuses concernent la conception logicielle(48) et les modèles et méthodes (31). Il s'agit bien des thèmes sur lesquels nous nous positionnons.

Le défis en génie logiciel sont tellement nombreux que l'idée de ce chapitre n'est pas de faire une classification et de me positionner mais simplement de mentionner les sujets qui me semblent importants (réalisables ou pas) et les difficultés à les traiter. Je pars du constat principal que le développement du logiciel coûte trop cher pour explorer des pistes d'amélioration ouvrant des voies de recherche. Les enjeux dépassent donc le cadre purement scientifique.

Ce que nous appelons mythe dans ce chapitre prend parfois le sens d'idée fausse comme celles des méthodes formelles dans [Hal90 ; BH94] ou du logiciel [Pre10] (p. 21) mais aussi le sens d'objectif ultime inaccessible au sens de mes humbles recherches.

2.2 Préambule

Posons ici quelques hypothèses et principes pour faciliter le raisonnement développé dans ce chapitre.

Il existe différentes formes de logiciel et d'applications, mais du point de vue du développement nous ne ferons pas de différences.

Simplification 2.2.1 (Application/logiciel) *On ne distingue pas ici progiciels et logiciel sur mesure car les problématiques sont similaires même si elles ne s'observent pas de la même manière. On ne distinguera pas non plus les types d'application "standalone", web app, web services, mobile, infrastructure... Enfin, un logiciel, peut aussi être l'intégration de plusieurs applications.*

Rappelons que pour une application, plusieurs projets de développement et de maintenance sont nécessaire pour maintenir en condition opérationnelles de fonctionnement. Au delà du coût de maintenance (75% du coût du logiciel) *cf.* page 21, si on tient compte de la dualité besoin métier et technologie pour la conception du logiciel, *cf.* cycle "Y" de la FIGURE 1.1, il est évident que peu d'applications peuvent être considérées comme complètement nouvelles. Il y a toujours un existant technique ou métier.

Principe 2.2.1 (Développement/Evolution continue) *Il n'y a pas lieu de considérer séparément le développement et la maintenance. C'est un unique et même processus, on peut parler d'évolution continue. On utilisera simplement le terme développement, et comme indiqué dans le chapitre 1, on fait abstraction de la gestion de projet.*

Simplification 2.2.2 (Développeur) *Dans ce qui suit, développeur correspond à tous les participants du développement pas uniquement ceux qui réalisent la conception détaillée et le codage comme on peut le trouver dans les offres d'emploi.*

Simplification 2.2.3 (Prestation) *Il n'y a pas lieu de considérer séparément le développement mené en interne ou par un prestaire. Cela change l'organisation concrète du projet et des tâches (cf. Section 1.1.2) ainsi que la répartition des coûts mais pas les principes.*

Simplification 2.2.4 (Besoins) *On ne sépare pas ici besoins fonctionnels et non fonctionnels, les deux sont à mettre en œuvre pour le projet de développement.*

Les retours d'expérience sur des décennies de developpement ont développé de "bonnes pratiques" avec des principes largement acceptés. Reprenons ceux du processus unifié de la page 23, ils sont appliqués que les méthodes soient agiles ou plus traditionnelles.

Principe 2.2.2 (Grands principes de développement) *Le développement est itératif et incrémental, basé sur les cas d'utilisation et l'architecture.*

Principe 2.2.3 (Concepts de conception) *Pressman met en évidence les concepts suivants pour la conception logicielle : " Abstraction, Architecture, Patterns, Separation of concerns, Modularity, Information hiding, Functional independence (cohesion, coupling), Refinement (complementary to abstraction), Aspects, Refactoring." [Pre10] (p. 222)*

Ces concepts sont intercorrélés. Nous avons abordé la plupart des concepts dans le chapitre 1, précisons les autres. Les préoccupations (*concerns*) sont liées aux aspects et peuvent représenter les points de vue des acteurs du développement ; on notera en particulier la répartition et la concurrence des processus, la persistance des données, la sécurité et la confidentialité, les IHM, etc.

2.3 Problème fondamental

Le logiciel coûte cher, de plus en plus cher.

Selon Gartner, *"Worldwide IT spending is projected to total $4.6 trillion in 2023, an increase of 5.5% from 2022, according to the latest forecast by Gartner, Inc."* Gartner[1]. En 2021, c'était 4.1 trillion de dollars US (milliard de milliards), avec une augmentation de 8.4% en un an. Le logiciel mentionné correspond aux applications d'entreprises avec des taux de croissance toujours élevés, auxquelles il faut ajouter une partie des services (*cf.* Table 2.1).

Worldwide IT Spending Forecast (Millions of U.S. $)	2022 Spending	2022 Growth (%)	2023 Spending	2023 Growth (%)	2024 Spending	2024 Growth (%)
Data Center Systems	216,095	13.7	224,123	3.7	237,79	6.1
Devices	717,048	-10.7	684,342	-4.6	759,331	11.0
Software	793,839	8.8	891,386	12.3	1,007,769	13.1
IT Services	1,250,224	3.5	1,364,106	9.1	1,502,759	10.2
Communications Services	1,424,603	-1.8	1,479,671	3.9	1,536,156	3.8
Overall IT	**4,401,809**	**0.5**	**4,643,628**	**5.5**	**5,043,805**	**8.6**

Table 2.1 – Répartition du coût (Source : Gartner 2023)

Dans une entreprise, le nombre d'applications croît et les coûts de maintenance augmentent avec le temps. Le nombre d'applications croît parce qu'il y a toujours de nouveaux besoins (augmentation des périmètres) mais aussi du fait de la redondance (recouvrement de périmètres). La redondance est induite par le fait certaines applications sont conservées (le patrimoine applicatif ou *legacy*) même si leur périmètre est réduit et que de nouvelles applications couvrent une partie du périmètre. On a aussi le cas d'applications développées en parallèle dans des services, directions ou succursales différentes, par ignorance ou volonté d'autonomie. Mentionnons enfin le "gaspillage" ou le côté "non responsable"

1. https://www.gartner.com/en/newsroom/press-releases/2023-04-06-gartner-forecasts-worldwide-it-spending-to-grow-5-percent-in-2023

à la fois du logiciel et des ressources associées. Le logiciel étant intangible l'utilisateur est moins sensible à ce point. On constate du gaspillage, en entreprise ou dans les administrations, à développer des solutions numériques qui ne sont pas ou peu utilisées, mais qu'il faut parfois maintenir en condition opérationnelle. Les logiciels qui ne sont pas en adéquation avec les besoins des utilisateurs sont abandonnés. Ce qui coûte cher dans le logiciel c'est son développement, y compris la maintenance, qui se retrouve aussi en partie dans la catégorie 'services' [2] de la FIGURE 2.1.

Problème

Le développement du logiciel est continu sur son cycle de vie, il est donc onéreux. Il coûte trop cher.

"What are the costs of software engineering ? Roughly 60% of software costs are development costs ; 40% are testing costs. For custom software, evolution costs often exceed development costs" [Som11], p. 6. On trouve des estimations moyennes de développement sur mesure (*custom software*) sur internet [3] par exemple [4] : (i) un CMS coûte de $5,000 à $50,000, (ii) une application de santé coûte de $10,000 à $500,000, (iii) un produit SaaS coûte de $50,000 à $200,000, (iv) une application mobile coûte de $10,000 à $200,000. Ces estimations ne tiennent pas compte de la maintenance, or le logiciel doit être corrigé de ses erreurs, amélioré et adapté à de nouveaux besoins ou de nouvelles contraintes. Nous incluons tous ces coûts dans le **développement continu** (principe 2.2.1).

A priori, les coûts de maintenance corrective (hors évolution) baissent car le nombre de "bugs" diminue alors que la maintenance évolutive étant un développement, elle entraîne logiquement de nouvelles erreurs. Les deux augmentent avec le temps pour plusieurs raisons : (1) déléguer la maintenance (tierce maintenance applicative - TMA) fait perdre l'expérience et l'historique du projet, on refait les mêmes erreurs par méconnaissance (c'est une régression), (2) attendre trop longtemps pour corriger ce qui n'a pas été bien conçu induit une dette technique, (3) intégrer les évolutions implique un risque de dégrader une autre partie de l'application [5], (4) plus le temps avance, moins le logiciel est en phase avec les besoins qui l'ont motivé. La liste n'est pas exhaustive et le lecteur pourra s'inspirer des lois (Murphy, Conway, Hofstadter, Brooks...) souvent établies depuis longtemps qui s'appliquent au développement du logiciel [6] et en expliquent la dérive des coûts. Concernant l'évolution, on trouvera les lois de Lehman et al. [Leh+97].

Les coûts de maintenance sont liés à la qualité (au sens large) des logiciels mais aussi du fait des contraintes permanentes d'évolution technologiques, métier ou règlementaires. Par exemple, le coût lié à la qualité du logiciel aux US en 2018 était estimé à 2,84 trillions

2. `https://www.idc.com/getdoc.jsp?containerId=prUS46047320`
3. Les articles scientifiques sur l'estimation des coûts du logiciel vont jusqu'au début des années 2000.
4. `https://www.tpptechnology.com/en/blog/custom-software-development-costs-the-2023-guide/`
5. Plus le logiciel est évolutif, on a anticipé les cas, moins le risque est élevé mais c'est rarement le cas, car souvent, l'objectif étant de livrer un logiciel qui fonctionne avec un temps de développement limité qui n'intègre pas l'amélioration continue.
6. `https://www.timsommer.be/famous-laws-of-software-development/`

de $ (*cf.* FIGURE 2.1)[7]. La part d'échecs due à une mauvaise gestion de projet semble minime, mais cette cause est importante dans l'augmentation des coûts.

Dans un projet de développement logiciel, même s'il n'y a pas de statistiques sur le sujet[8], le coût humain est de loin le plus important dans les coûts directs. Par contre notons que ce coût augmente du fait de la rareté de la ressource face au besoin, *il y a une inflation des salaires et des coûts de recrutement élevés*. De manière conjoncturelle, les plans "Transformation numérique (ou digitale)" de la décennie 2010-2020 ont impulsé un besoin énorme de logiciel dans les entreprises et dopé les services numériques. *"Dans son rapport 2020, Evans Data Corporation*[9] *recense 26,9 millions de développeurs et de développeuses dans le monde. Cette population devrait atteindre les 45 millions d'ici 2030 alors que la demande supplante de loin l'offre dans ce domaine. Cela représente une croissance moyenne de 75% au cours de la prochaine décennie."* Post de Libeo (Québec)[10] On peut aussi parler de l'efficacité ou la rentabilité des participants au développement d'applications, défauts de formation pour des développeurs formés en un an par alternance, codage par copier-coller, compétences et apétences, efficacité des collaboration avec le télétravail, réunionite de projet, etc. Les causes sont multiples et les effets variés.

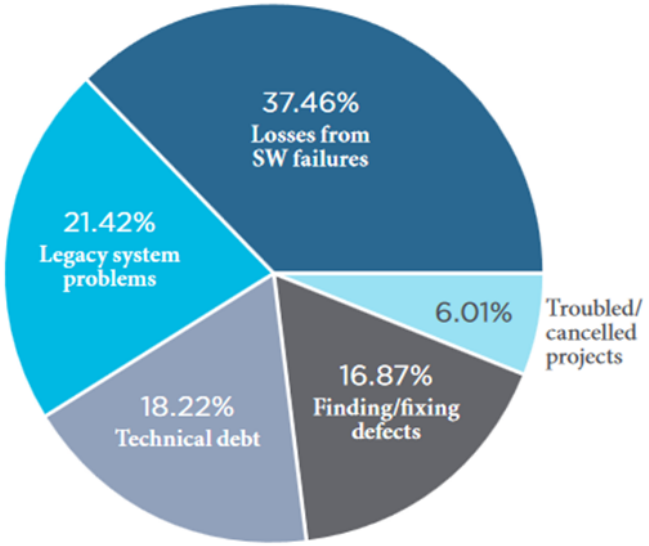


FIGURE 2.1 – Répartition des coûts liés à la faible qualité logicielle (source : CISQ)

La question est donc comment réduire les coûts de développement du logiciel tout au long de sa vie et répondre efficacement aux besoins qui le motivent. Bien qu'un seul problème global soit mentionné la question est complexe et se découpe en sous-problèmes non-indépendants, nous l'abordons dans la section suivante..

Remarque

Le problème abordé ici montre que nos travaux s'inscrivent dans la *recherche appliquée* avec des points de vue divers.

7. https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf
8. On trouve des abaques pour déterminer des coûts mais les coûts de projets informatique restent confidentiels pour les entreprises.
9. https://evansdata.com/reports/viewRelease.php?reportID=9
10. https://libeo.com/code-geeks/developpement-logiciel-stats-2021/

2.4 Analyse

Pour réduire les coûts de développement, nous choisissons d'axer l'analyse selon trois axes, on doit (1) améliorer la qualité des processus de développement pour rationaliser et réduire le coût humain, (2) améliorer la qualité des produits livrés pour réduire les coûts de maintenance, (3) adapter le développement au besoin et à la situation. Développons les axes de progression, qui ne sont pas indépendants. Les deux premiers sont basés sur la qualité telle que détaillée dans l'annexe B.6.

2.4.1 Améliorer la qualité des processus de développement

Des progrès notables ont eu lieu dans l'intégration continue et la gestion des configurations pour le déploiement, mais aussi sur la traçabilité des exigences. Au niveau du développement lui-même, les progrès ont été méthodologiques, à travers les bonnes pratiques, il y a consensus sur de grands principes *cf.* le principe 2.2.2. Les processus sont moins rigides pour éviter l'effet tunnel, la validité est revenue au centre avec les méthodes agiles, la vérifiabilité des besoins fonctionnels et non-fonctionnels est prise en compte dans les approches basées sur les tests, l'évolutivité est induite par la révision des périmètres lors des incréments. Ces pratiques réduisent le risque d'erreurs tardives.

Par contre, d'un point de vue systématisation et automatisation pour réduire l'intervention humaine, tout ou presque reste à faire. Cela nécessite de systématiser les processus pour automatiser ce qui peut l'être. Il n'existe pas de solution générique pour cela. On se heurte aussi à un facteur économique, le client ne paie pas pour l'investissement sur la rentabilité du développement mais pour les produits livrés. Le prestataire n'est pas directement incité à investir sur le sujet. Deux axes de travail sont choisis :

— Réutiliser tout ou parties de logiciel. Le cycle "Y" met en évidence la dualité technique/métier de l'application. Naïvement, le même support technique est réutilisable pour différentes applications et inversement, si la séparation est propre, on peut migrer le support technique pour une même application.

— Automatiser les tâches répétitives. Cet objectif signifie faire intervenir le moins possible l'humain et donc automatiser autant que possible. Plus précisément, l'humain doit être au cœur des phases d'ingénierie mais quasi-absent des phases répétitives. Cela nécessite de systématiser le développement, un peu comme une gamme de fabrication de produits manufacturés.

Les deux nécessitent d'améliorer la qualité des produits.

2.4.2 Améliorer la qualité des produits

Améliorer la qualité des produits, c'est essentiellement les rendre plus fiables (valides, robustes, vérifiables, intègres) et réutilisable (extensibles, compatibles, portables, ergonomiques...). (1) améliorer la fiabilité des produits livrés réduit les coûts de maintenance, et facilité la réutilisation. (2) améliorer la réutilisabilité c'est anticiper les usages et les

évolutions mais aussi la documentation. Les facteurs pour les atteindre sont l'abstraction (autodocumentation, généralité, séparation interface/mise en œuvre), la modularité (les unités de réutilisation peuvent être) plusieurs échelles -*scalability*) et la rigueur (le formalisme est indispensable pour pouvoir vérifier (vérifiabilité) et exploiter (transformer).

Les travaux sur la conception et la programmation à objets ont mis en évidence que la modularité, l'abstraction et l'encapsulation (entre autres *cf.* le principe 2.2.3) permettent de favorisent la réutilisation mais aussi de mieux appréhender les systèmes complexes par une structuration modulaire hiérarchique (holistique) [on est passé de l'approche cartésienne à l'approche systémique *cf.* la discussion de la page 24]. Dans les années 90, on s'est rendu compte que le module "objet" convient bien à la programmation mais pas trop à la conception car il est trop petit. On est passé à l'approche composant (*component-based software engineering*). Cela a inspiré fortement les travaux sur les architectures logicielles. Dans les années 2000, le développement du e-commerce favorise les applications distribuées sur Internet, le module "composant" est trop gros pour la programmation web dans laquelle on souhaite externaliser des fonctions (et les bus à objets type CORBA sont trop contraignants). On est passé à l'approche service (*Service-oriented Software Engineering*). Le hic, dans les deux cas, est que ces concepts de composant et service n'existent pas dans les langages de programmation. La structure est décrite par de la méta-information (annotations, fichiers XML) qui ne sont pas traités par le compilateur dans la sémantique du langage de programmation, mais sous forme de pré-traitements. Le codage devient compliqué et les *frameworks* incontournables.

2.4.3 Adapter le développement au besoin et à la situation

Le développement du logiciel peut coûter cher parce qu'il n'est pas en phase avec les systèmes auxquels il est associé, ce qui augmente les coûts soit parce que la méthode utilisée n'est pas adaptée. Considérons deux cas : (i) Dans les systèmes d'information, le logiciel (du système informatisé) est l'intégration de nombreuses applications (progiciels et logiciels) de générations différentes, correspondant à différentes technologies ou à des processus qui ont changé. le problème est que rapidement, le logiciel n'est pas en phase avec les processus en entreprise. Nous verrons la problématique de l'alignement entre les processus et le logiciel *Business-IT Alignment (BITA)* dans le chapitre 6 pour lesquels les enjeux sont énormes. (ii) Le logiciel est "pervasif" mais dans ses domaines d'application, on ne maîtrise pas le génie logiciel. Ce peut être le cas lorsque les modèles utilisés sont différents comme dans le monde du vivant ou la cybernétique, ou que les méthodes ou les qualités visées sont différentes comme en automatisme. Nous verrons le cas des systèmes de production industrielle dans le chapitre 7 pour lesquels les besoins en génie logiciel sont énormes.

Bien que le raisonnement présenté ici soit simpliste, il apparaît que les facteurs de qualité logicielle sont nombreux. Les solutions au problème posé page 53 ne sont donc pas simples.

2.5 Vision et propositions

La vision et les propositions visent à répondre à la question *De quoi a-t-on besoin pour améliorer la qualité des produits et processus de développement en tenant compte du contexte ?* La vision se base sur quelques principes et se frotte à quelques mythes.

2.5.1 Une affaire de modèles

Rappelons (si nécessaire) que les produits (les artéfacts du développement) ne sont pas seulement le logiciel final mais aussi les spécifications et les documentations produites tout au long du développement et que le logiciel considéré ici est un ensemble d'applications interopérantes qu'on peut trouver dans un système d'information, un système de production, etc. Néanmoins, trop de produits restent informels notamment dans les spécifications et la documentation. Des spécifications informelles ou même standardisées [*cf.* annexe B.4.1] sont inexploitables (hormis par une IA de traitement de textes), on a besoin de notations semi-formelles et formelles.

Principe 2.5.1 (Tout est modèle) *Les modèles sont incontournables pour traiter de logiciel, tous les produits du développement sont des modèles.*

Il faut donc utiliser plus de modèles dans le développement et réduire au maximum les textes non pertinents (intuitivité, autodocumentation, lisibilité). Par exemple, (i) les exigences doivent être matérialisées de telle manière qu'on puisse en partie automatiser des tests de recette, (ii) une API doit être documentée non seulement avec la liste des opérations utilisables, leurs contraintes mais aussi l'ordre dans lequel il est licite de les invoquer. Les modèles favorisent la concision des produits. Un modèle n'est utile que s'il est correct et possède de bonnes propriétés (*cf.* Section 1.3). Pour qu'il soit vérifiable et exploitable, il faut qu'il soit précis et défini avec rigueur [11]

Principe 2.5.2 (L'utilité implique la rigueur) *Les modèles doivent être le plus formel possible pour être vérifiables et exploitables (réutilisables et automatisables).*

2.5.2 Automatiser

Pour réduire l'intervention humaine, confions la construction [des produits] du logiciel à un automate.

11. J'avais déjà défendu rigueur et modularité comme des principes de base dans ma thèse [And95], j'y ajoute ici l'abstraction, et au bon niveau.

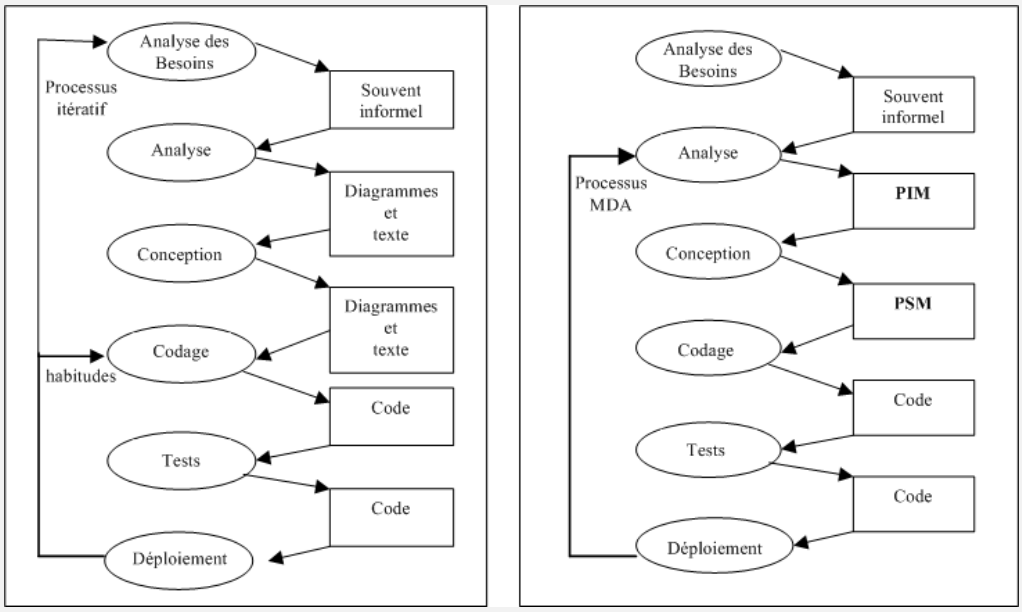
Mythe 1 - Le développement peut être confié à une Intelligence Artificielle.

Un robot automatise les tâches du processus de développement en demandant aux développeurs de prendre les décisions au fil du développement.

Cette vision est réellement un mythe au vu de la variété et de la complexité des systèmes logiciels à mettre en place. Nous pensons par contre, un peu à l'instar des lignes de produits logicielles, que des processus spécifiques peuvent être mis en place. C'est le cas des applications de gestion dont la structure est calquée sur celle des bases de données.

La pratique du développement montre différents problèmes de ruptures : (i) rupture entre les niveaux d'abstraction *e.g.* les modèles métiers vs les modèles applicatifs ou les architectures logicielles vs le code, (ii) rupture entre les étapes *e.g.* modèles des besoins vs modèles d'analyse vs modèles de conception, (iii) rupture entre le logiciel et les autres produits du développement *e.g.* la maintenance du logiciel remonte rarement aux modèles de conception.

La vision tout modèle et la continuité dans le processus de développement est en phase avec l'approche MDA *cf.* section 1.2.6 et annexe C.2.5, dont un des objectifs est d'améliorer la productivité en réduisant la distance entre les modèles et le code [KWB03] comme le montre la figure suivante (*cf.* FIGURE C.11), et rend le processus réellement itératif [a]. L'OMG s'appuie pour cela sur un langage unique UML pour tout le développement, qualifié de sans coutures (*seamless*).



Le besoin de formalisme n'est pas directement lié à MDA, d'ailleurs un des reproches d'UML est son manque de sémantique formelle. Le besoin apparait lorsqu'on veut faire de l'ingénierie et transformer les modèles comme le montre la synthèse [GC14].

a. Une autre approche est utilisée pour connecter modèles et code (on la retrouve aussi dans les architectures) et d'insérer des morceaux de code dans les modèles pour ne pas les perdre dans les versions suivantes, c'est l'approche *roundtrip engineering*.

Mythe 2 - Développer c'est raffiner.

Concevoir le développement comme un raffinement progressif à partir d'une spécification (un modèle) n'est pas envisageable en pratique du fait des différentes préoccupations de conception (principe 2.2.3).

Cette vision convenait aux approches client-serveur traditionnelles dans lesquelles les traitements sont raffinés en fonctions élémentaires et les données monolithiques. Elle fonctionne aussi pour les méthodes formelles ou la simulation, approches dans lesquelles il n'y a pas de rupture des concepts. Elle ne convient pas bien aux applications distribuées et modulaires dans lesquelles données et traitements sont regroupés ou lorsqu'il y a plusieurs aspects non orthogonaux (non indépendants). L'approche MDA de l'encart ci-dessus ne fonctionne pas sauf à considérer les approches *Executable UML* [MB02 ; Rai+04] dans lesquelles le modèle spécifique à la plateforme (PSM) vient simuler le modèle indépendant de la plateforme (PIM). L'étape clé du développpement, celle qui nécessite de la véritable ingénierie, est celle de la conception, comme l'explicite le cycle "Y" de la FIGURE 1.1.

Mythe 3 - On peut tout transformer.

La transformation de modèle ne fonctionne que lorsque les modèles sources et cibles sont proches et que le modèle source contient suffisamment d'information car la transformation infère mais n'invente pas.

Nous avons travaillé sur ce sujet dans [AT20 ; AT21], en montrant qu'il s'agit de mythes et en proposant un double processus d'abstraction et de raffinement. La thématique du développement par les modèles fait l'objet du chapitre 4. Nous le traitons aussi de manière plus secondaire dans le chapitre 6.

2.5.3 Traiter la complexité

La complexité est un grand défi posé aux ingénieurs du logiciel : *"There is one trend that is undeniable—software-based systems will undoubtedly become bigger and more complex as time passes. It is the engineering of these large,complex systems, regardless of delivery platform or application domain, that poses the "grand challenge" for software engineers."* [Pre10] (p. 821) citant Manfred Broy. C'est aussi un des défis techniques de l'ingénierie des modèles avec l'hétérogénéité et la séparation des préoccupations [Buc+20], au delà du logiciel lui-même.

Dès qu'un système est complexe, nous avons vu dans la section 1.2.5 que le modèle devait considérer plusieurs points de vue orthogonaux (aspects), qu'on peut représenter comme des modèles. Un modèle doit pouvoir être la composition (horizontalement) d'autres modèles. Dans l'idéal, on peut imaginer une description modulaire des modèles et des opérateurs de composition (composabilité). Ce n'est en général pas le cas.

Principe 2.5.3 (Décomposer pour simplifier) *Un modèle peut être composé d'autres modèles pour gérer la complexité, notamment pour couvrir des points de vue complémentaires. Il s'agit d'une composition tout-partie.*

La complexité se gère aussi verticalement en masquant des détails du modèle (abstraction). C'est le principe même des architectures. En effet, pour qu'un ensemble d'intervenants puissent intervenir, chacun dans son rôle, sur le chantier du logiciel, il est indispensable d'en partager les principes fondamentaux, qui sont inscrits dans l'architecture, qui est le plus haut niveau d'abstraction d'un modèle.

Principe 2.5.4 (Abstraire pour simplifier) *Un même modèle peut être vu à différents degrés de détail, on doit pouvoir le déplier sans perdre la logique globale.*

Dans ce manuscrit, deux types d'abstraction sont considérées : (i) celle des modules à l'intérieur des modèles (*e.g.* classes, composants, services...) qui présentent une interface permettant de d'assembler les modules ; et (ii) celles des modèles eux-mêmes qu'on peut organiser en couches d'abstraction.

Abstraction dans les modèles

Pour être réutilisables, les éléments de modèles doivent être modulaires. Nous nous positionnons dans l'approche CBSE et SOA.

Mythe 4 - *The Grand Challenge of Trusted Components.*

Comme Bertrand Meyer, nous imaginons une réutilisation réellement modulaire dans laquelle, à l'instar de l'électronique on peut constituer un circuit à partir de composants élémentaires ou pas. *A Trusted Component is a reusable software element possessing specified and guaranteed property qualities.*

Cette vision a lontemps été celle des travaux sur l'architecture logicielle. Elle est relativement similaire dans les approches composants ou services. Deux freins importants en réduisent le potentiel concret : (i) les concepts n'existent pas au niveau des langages de programmation, composants et service sont moins faciles à utiliser que les classes, objets ou types de données (voir l'encart page 57) ; (ii) la réutilisation est terminale, au sens où dans le programme, on "tire" le composant ou service avec toutes ses dépendances (boîte noire). Les interfaces de composants (ou services) réutilisables ne mentionnent pas les besoins,qui sont supposés résolus.

Principe 2.5.5 (Réutiliser c'est reconstruire un contexte favorable) *Pour pleinement utiliser un composant (ou service logiciel) il faut que son contexte d'utilisation soit modélisé dans son abstraction, on peut utiliser des interfaces, des contrats pour cela.*

Nous traitons des problèmes d'abstraction, de composition et de réutilisation à l'intérieur des modèles notamment au niveau des architectures logicielles dans le chapitre 3. En effet raisonner au niveau de l'architecture permet de traiter les problèmes fondamentaux, qu'il faut résoudre avant de se perdre dans les détails.

Abstraction entre modèles

Les modèles de niveau d'abstraction différents doivent pouvoir être connectés pour être exploités (traçabilité, raffinement, abstraction, ...). Si des transformations de modèles existent alors ce lien est calculé, sinon il faut l'inférer ou lier manuellement.

Mythe 5 - Les modèles sont des modules qu'on peut assembler.

Un modèle complexe est l'assemblage de modèles plus simples (les modules) définis par des interfaces. La combinaison de la composition et de l'abstraction permet de traiter les modèles complexes. Mais il manque l'encapsulation pour vraiment permettre le passage à l'échelle des modèles modulaire (*scalabilité*). Ce mythe se situe au niveau des langages de modélisation.

Concernant l'abstraction de modèles qui permet de les organiser en couches, nous posons les principes suivants.

Principe 2.5.6 (Lier n'est pas contraindre) *Le tissage permet de relier des éléments de différents modèles sans changer la sémantique des modèles. C'est une forme de composition qui préserve l'indépendance (tissage non-intrusif).*

Les problèmes de navigation dans les couches d'abstractions se trouvent dans le développement mais aussi de manière plus complexe dans les systèmes d'information.

Nous traitons des problèmes d'abstraction entre les modèles à travers l'ingénierie ou la rétro-ingénierie des modèles du logiciel dans les chapitres 3 et 4, des points de vue du système d'information dans le chapitre 6.

2.5.4 Etablir la confiance

Nous avons déjà abordé ce point dans le mythe 5. Il s'agit ici de tout mettre en œuvre pour s'assurer d'une certaine qualité des modèles et donc des produits du développement. L'approche est de vérifier les propriétés des produits que nous avons détaillées dans la section 1.3 du chapitre 1.

Nous traitons des problèmes de confiance dans les modèles au niveau des architectures logicielles dans le chapitre 3, des points de vue du système d'information dans le chapitre 6, et de la sécurité des applications dans le chapitre 5.

2.6 Recherche en génie logiciel

Dans cette section, nous exprimons notre vision sur la manière de mener des recherches en génie logiciel, et donc de mythes et réalités.

2.6.1 Contexte

La recherche en génie logiciel est caractérisée par les faits suivants :

— Les produits ne sont pas dissociables des processus. La manière de faire influence le résultat (loi de Conway *cf.* page 54). De même la qualité s'évalue par des critères et des métriques mais il n'y a pas un indicateur unique de performance. Bon nombre de critères ne sont pas pleinement mesurables *e.g.* la réutilisabilité ou la portabilité.

— Le pragmatisme est essentiel en génie logiciel, qui n'est pas une science exacte. Un certain nombre de travaux scientifique sont de nature empirique [SSS08]. De même, les problèmes de coûts et de qualités sont systémiques avec de nombreux paramètres. La complexité est le grand défi du GL (page 60).

— La recherche institutionnelle est largement concurrencée voire supplantée par les industriels qui fixent des standards *de facto* qui influencent les normes, spécialement pour les architectures techniques. Inversement, il faut une force de frappe importante pour mettre en œuvre les techniques et outils du GL.

— Les praticiens eux-mêmes contribuent à l'évolution des "bonnes pratiques", dans des communautés de taille variable, qui vont des groupes d'utilisateurs aux consortiums industriels. Les méthodes agiles ou la qualité des processus en sont des exemples.

— Les outils développés pour le GL sont rarement pérennes dans le temps. ne serait-ce que du fait de la dette technique qui s'accumule pour tout logiciel quel qu'il soit mais aussi du manque de support de maintenance.

— Depuis les plans "transformation digitale" vers 2010, la culture de l'innovation est souvent associée au numérique. Dans le GL, les méthodes agiles contribuent positivement à cette tendance pour la prise en compte des besoins utilisateurs. Néanmoins les fondements du GL ne sont pas à rayer de la carte. Ce phénomène de "buzz" existe aussi en recherche et influence le financement et donc l'orientation des recherches. De nos jours, "IA et apprentissage" devient incontournable dans un projet de recherche.

— A l'inverse, on peut noter un certain communautarisme, alors que les visions pluri- ou trans-disciplinaire donne une ouverture favorable au traitement des problèmes complexes. C'est encore plus vrai lorsqu'on croise l'informatique "pervasive" avec d'autres domaines scientifiques.

2.6.2 Positionnement

Mes thématiques s'inscrivent dans la classe *10011007 - Software and its engineering* de la classification ACM 2012 [12] (Figure 2.2). Au niveau du GDR-GPL [13], mes travaux

12. `https://www.acm.org/publications/class-2012`

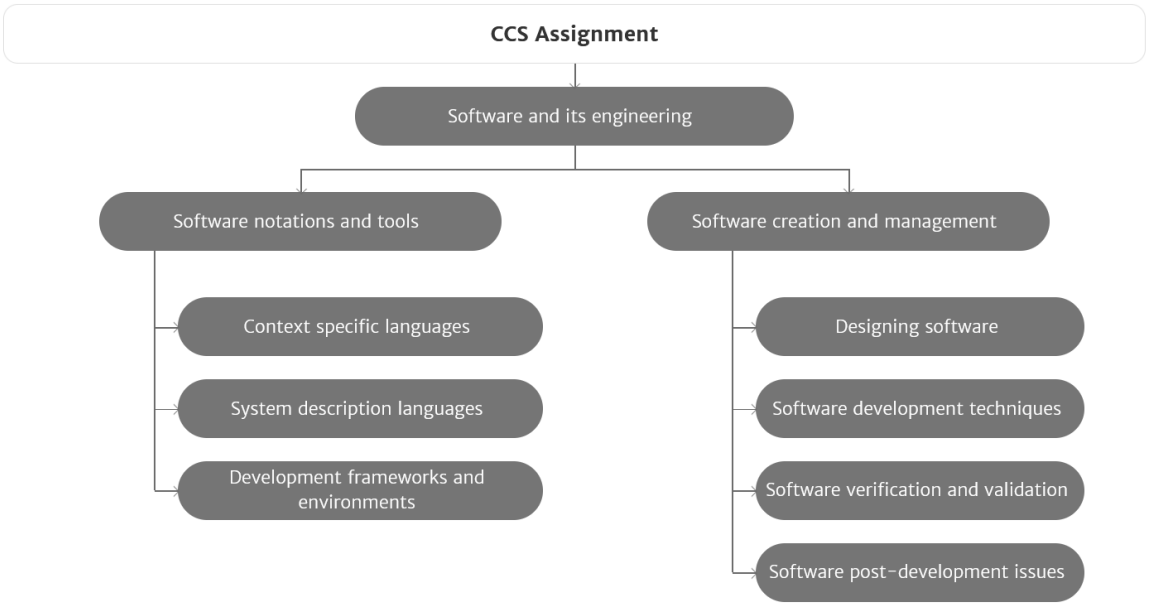


FIGURE 2.2 – The 2012 ACM Computing Classification System (Source : ACM)

s'inscrivent principalement dans trois thématiques ((i) l'ingénierie dirigée par les modèles, (ii) le développement de techniques de vérification et de validation des systèmes et (iii) l'étude des pratiques de développement y compris de maintenance logicielle) et de manière plus marginale pour la partie sécurité dans la thématique "prise en compte dans l'ensemble du cycle de vie du logiciel des propriétés telles que la sécurité, la sobriété écologique, la résilience".

2.6.3 Approche

Dans le contexte spécifique de la recherche en génie logiciel, il est difficile d'imaginer une thématique précise étudiée sur le long terme [14] pour le problème large que nous avons mentionné dans la section 2.3 et décliné ensuite. La recherche GL doit se confronter aux cas concrets. C'est ce pragmatisme qui m'anime quand à la sélection des sujets avec deux règles : (i) partir des besoins concrets pour apporter des solutions, (ii) travailler en équipe pour voir plus grand. Ma recherche est donc en partie *opportuniste* car mes rencontres influent sur les thématiques étudiées à tel ou tel moment, mais non aléatoire car je conserve le *fil conducteur* autour des modèles et le génie logiciel basé sur l'ensemble de principes énoncés dans la section 2.5. Les travaux que j'ai menés depuis 2003 se sont concentrés d'abord sur la qualité des modèles (modélisation, vérification de modèles semi-formels ou formels puis le raisonnement sur les architectures logicielles) puis celle des processus (ingénierie des modèles pour automatiser le développement). Ils sont développés dans les

13. Groupement de Recherche Génie de la Programmation et du Logiciel `https://gdr-gpl.cnrs.fr/Presentation/Motivation`, une unité de l'INS2I du CNRS qui regroupe la communauté scientifique française intéressée par le Génie Logiciel et la Programmation.

14. Cela existe pour des travaux de recherches plutôt théoriques (modèles et langages formels par exemple, sémantique...) ou des niches comme les méthodes formelles ou encore des techniques outillées assez indépendantes (outil de vérification, outil de mesure...) que d'autre peuvent utiliser.

chapitres 3, 4. La partie expérimentale est menée dans différents contextes : BITA (*cf.* Chapitre 6), production et cybernétique (*cf.* Chapitre 7)).

Dans le contexte de recherche mentionné en début de cette section, la contribution de petits groupes de chercheurs, ne peut être que modeste face aux besoins. Dans les travaux menés, pour résoudre les problèmes de génie logiciel auxquels nous nous intéressons, nous suivons une approche de recherche appliquée récurrente basée sur le triplet `<méthode, technique, outils>`.

— La méthode définit les modèles où les concepts sont définis rigoureusement et pour lesquels des propriétés peuvent être établies et leurs traitements dans des processus.
— Les techniques raffinent la méthode en systématisant les traitements de vérification ou de transformation de modèles.
— Les outils instrumentent les techniques et permettent d'expérimenter de manière opérationnelle le processus (i) modéliser, (ii) vérifier/valider, (iii) transformer.

La contribution aux axes de progression mentionnés dans la section2.4 passe aussi par la formation [15] des étudiants aux méthodes, outils et bonnes pratiques du développement. Outre les cours à l'université, cette contribution par la rédaction d'ouvrages sur le sujet [AV03c ; AV03b ; AV01a ; AV01b ; AV02 ; AV04 ; AV03a ; AV13 ; AR96].

2.6.4 Mise en œuvre

Dans la veine des mythes et réalités, la recherche suit deux logiques, celle des objectifs thématiques (problèmes, motivations et contributions) et celles des projets avec des considérations pratiques et budgétaires. Les projets permettent de dimensionner les actions. Dans une vision idéale, chacun des projets d'une organisation vient contribuer à des objectifs thématiques (stratégiques). En recherche institutionnelle financée, les chercheurs doivent monter des projets pour trouver des financements, les projets doivent être définis pour répondre soit à un appel d'offre soit à des thématiques prédéfinies qui ne sont pas celles des objectifs thématiques. La recherche opportuniste que je mentionnais précédemment, prend un nouvel axe, celui du financement qui est un paramètre supplémentaire qui souvent introduit de l'incohérence ou de la déviation. L'approche projet limite les objectifs à un sous-ensemble théoriquement atteignable, répartit l'effort sur un intervalle de temps réduit et laisse de nombreux travaux en chantier. La réalité de la recherche est un croisement entre ces deux visions, parfois antagonistes, thématiques et projets.

La vision projet se heurte à un autre problème, celui de la gestion des participants. Il faut fédérer l'équipe projet de recherche, dont les membres sont indépendants et rarement salariés (thésards, postdoc, ingénieur). Ce statut induit un fonctionnement collaboratif qui pose problèmes à la fois de synchronisation, d'investissement, de communication, etc. J'ai constaté que dans de nombreux projets, les résultats sont bien moins ambitieux que les objectifs. On peut y voir un certain cynisme en ce sens que les projets sont aussi montés pour avoir des financements.

15. Ce n'est pas encore de l'évangélisation.

2.7 Conclusion

Le génie logiciel est une thématique de recherche largement orientée vers la pratique et les problèmes à résoudre sont autant techniques qu'organisationnels. Il existe de nombreuses méthodes, approches et techniques pour développer le logiciel *tout au long de son cycle de vie* mais en pratique le développement reste très majoritairement basé sur l'humain, que ce soit en pure ingénierie ou en activités plus techniques. Le travail reste de faible qualité et la facture est élevée pour l'utilisateur final.

Les besoins en génie logiciel applicable restent donc importants et c'est encore plus vrai pour les domaines où les intervenants ne sont pas des spécialistes de logiciel (informatique temps réel, informatique scientifique parfois, et les domaines hors informatiques qui participent au développement). Je n'ai pas la prétention de tracer une *roadmap* de la recherche en génie logiciel. Les modèles me semblent incontournables comme objet d'étude des produits du développement. Dans ce chapitre, j'ai identifié quelques principes qui me semblent importants pour gérer la complexité et faciliter l'automatisation (rigueur, modularité, abstraction, composition, etc.). L'idée enfin est de tenir compte du contexte dans lequel est mené le développement pour être plus en adéquation avec le besoin et les solutions techniques. Ma recherche reste modeste, au regard des besoins et de la "concurrence", et s'appuie sur une vision pragmatique et collaborative de la recherche, en phases avec les moyens et les contingences liées aux projets.

Les pistes développées dans la suite de ce document sont les suivantes :

— raisonner sur les architectures logicielles dans le chapitre 3 car maîtriser le cadre dans lequel on travail est indispensable avant de passer au détail,
— exploiter l'ingénierie des modèles pour automatiser des tâches de développement par un processus croisé de d'abstraction technique et de raffinement fonctionnel dans le chapitre 4,
— renforcer la cohérence entre le logiciel et son contexte pour mieux maîtriser l'évolution des systèmes d'information dans le chapitre 6,
— améliorer la sécurité des applications par des mécanismes et des techniques de développement dans le chapitre 5.

Post-scriptum

Ce chapitre met en évidence des acquis sur la compétence suivante, que nous reverrons dans le chapitre 8.

C2

Identifier les problèmes, les ordonner, proposer des pistes, ordonnancer un ensemble d'activités de recherche pour structurer un travail de recherche.

Deuxième partie

Modèles pour le génie logiciel

> *The Diagram is Not the Model. The model is not the diagram. It is an abstraction, a set of concepts and relationships between them.*
>
> ERIC EVANS [Eva04]

En génie logiciel, comme dans les autres domaines d'ingénierie, il est important d'identifier les "vrais" problèmes, ceux qui présentent le plus de risques pour le projet de développement du logiciel. Ce sont les domaines métiers ou technologiques où on a le moins d'expérience, et les parties dites critiques pour une application. A l'inverse, il faut capitaliser l'expérience en automatisant les parties de processus maîtrisées. Outre les risques du projet, le logiciel présente des risques pour son utilisation, liés aux exigences non fonctionnelles du projet (voir l'annexe B.6.4).

Dans cette partie, nous explorons trois thèmes qui s'inscrivent dans la vision de la section 2.5 du chapitre 2 et ciblent les cinq mythes qui y sont énoncés :

— raisonner sur les architectures pour poser la vision commune et identifier les risques,
— automatiser partiellement le développement pour en réduire les coûts.
— assurer le bon fonctionnement et réduire les risques liés à la fiabilité et la sécurité.

Dans toutes ces situations, nous utilisons des modèles. Le premier point fait l'objet du chapitre 3, le second fait l'objet du chapitre 4, enfin le troisième point est abordé pour deux propriétés, la fiabilité dans le chapitre 3 et la sécurité dans le chapitre 5.

Chapitre 3

Raisonner sur les architectures

Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.

Eoin Woods

La notion d'architecture logicielle a été introduite dans la Section 1.2.8 du chapitre 1. Raisonner sur les architectures permet de se focaliser sur l'essentiel et ne pas se perdre dans les détails de mise en œuvre. L'architecture est un des piliers du développement (principe 2.2.2). Si les bases communes sont posées alors chacun pourra travailler sur la partie sans casser la cohésion de l'ensemble. Les architectures logicielles dont il est question ici manipulent des composants et des services logiciels, il n'y a pas un style architectural exclusif (*cf.* page 40).

Remarque

Les travaux de ce chapitre ont formé un projet scientifique structurant de l'équipe de recherche à laquelle j'appartiens depuis 2003, et notamment durant la thèse de Mohammed Messabihi [Mes11]. Ces travaux ont donné lieu à la réalisation d'un outil [AAA07c] et une extension CostoTest [AMS16]. La section 3.6 fait référence au projet Econet, que j'ai initié et coordonné de 2066 à 2008.

3.1 Introduction

La composition de modules ou composants pour former des systèmes de grande taille est une démarche éprouvée dans divers domaines scientifiques et techniques. La modularité comme principe fondamental de construction est aussi adoptée pour la construction de logiciels. Cependant le niveau d'abstraction des éléments composés est variable d'un langage (et d'une méthode) à un autre. Ainsi on peut composer des fonctions, des modules, des objets, des composants, des services, ... Le développement basé sur l'emploi de composants met l'accent sur l'approche par composition : la réutilisation et la composition de composants, élémentaires ou non, sont au cœur de cette approche ; les composants peuvent être de niveaux d'abstraction variés ; leurs contextes d'exécution peuvent être centralisés ou répartis.

Dans la pratique industrielle des composants, de nombreuses limitations existent : il

n'existe pas d'approche largement admise de construction systématique des composants ; de nombreuses approches opèrent directement au niveau de l'implantation sans référer aux niveaux importants de la spécification et de la définition des propriétés souhaitées (*e.g.* .NET [1], EJB [2]). En UML un ensemble de diagrammes couvrent la conception et l'implantation sans pouvoir assurer leur cohérence globale et l'adéquation avec le code développé. Dans les modèles à composants et services plus abstraits, tels que Wright [GA94], SOFA [BHP06], BIP [GS05] et autres, les aspects communication sont privilégiés au détriment du fonctionnel et des services. De nombreux défis restent donc à relever pour une pratique courante de cette approche. Parmi ces défis citons le besoin de langages expressifs pour la description et l'assemblage (composition) de composants, la construction de composants corrects, fiables et réutilisables, l'interopérabilité entre composants.

Dans ce contexte, qui date de 2010, nous avons entrepris l'élaboration d'une approche de construction de composants *corrects* nommée Kmelia. La proposition de Kmelia répond aux besoins suivants : (i) un modèle à composants prenant en compte des données et des services où les interfaces sont plus riches que les simples signatures des services offerts, permettant ainsi une réelle réutilisation des composants ; (ii) un modèle indépendant des plateformes d'exécution ; (iii) un langage expressif de spécification de composants, de services et de leur assemblage (un effort particulier est fourni sur la description des services requis) ; (iv) un cadre de développement où on peut élaborer des composants corrects par construction et raisonner sur leur assemblage.

Notre travail s'attaque à lever certaines des limitations de l'approche par composants et services ; nous visons à rendre systématique et ainsi à simplifier la construction par composants en utilisant les approches formelles dès les premières phases de la construction de logiciels en n'ayant pas de rupture dans la chaîne de développement/vérification. On peut ainsi construire des composants dont on peut certifier, par des preuves formelles, la correction vis-à-vis des spécifications et qu'on peut maintenir plus facilement en remontant à leurs spécifications. D'un point de vue méthodologique, notre travail vise à montrer le bien fondé de l'approche par contrats pour la conception ascendante ou descendante, et l'intérêt de la spécification précise des besoins (requis) pour mettre en œuvre le concept de *composant sur l'étagère* (COTS).

Dans ce chapitre, je présente une synthèse des travaux et résultats autour de l'approche Kmelia pour le développement par composants et services. Ces travaux ont été faits progressivement et ont donné lieu à des résultats publiés successivement [AAA06d ; AAA07b ; AAA07a ; AAA08 ; AAA11] puis enrichis d'une autre série de travaux [AMA13 ; AMS16 ; AAM17 ; Mot+19].

En dehors de son aspect synthétique, ce chapitre met en exergue la composition dans le modèle Kmelia. Les formes de composition sont classées en deux catégories : l'une

1. `https://learn.microsoft.com/fr-fr/dotnet/`
2. `https://www.oracle.com/java/technologies/enterprise-javabeans-technology.html`

verticale, qui structure en profondeur et l'autre horizontale, qui structure en largeur. La combinaison des deux permet le passage à l'échelle (*scalability*).

Dans les sections 3.2 et 3.3, je présente les principales caractéristiques du modèle à composants Kmelia, puis la composition comme concept de structuration des composants pour construire des modèles de grande taille. Le mode de composition induit en particulier des interactions et des vérifications appropriées, entre composants et services. Dans une seconde partie, le formalisme est exploité pour assurer la correction par vérification des composants et de leurs assemblages (section 3.4). Les méthodes de vérification élaborées sont implantées dans l'outil COSTO (section 3.4) dont l'extention COSTOTest intègre du test de modèles (section 3.5). La contruction des modèles par rétro-conception a fait l'objet d'un projet Egide international appelé ECONET (section 3.6). Ce chapitre se termine par un positionnement de Kmelia par rapport à d'autres approches à composants (section 3.3.6) et les perspectives d'évolutions de nos travaux.

3.2 Kmelia : un modèle à composants multiservices

Kmelia est un modèle et un langage à composants multiservices. Les services décrivent des fonctionnalités avec la possibilité d'interagir avec d'autres services.

Le modèle Kmelia partage des caractéristiques communes avec les approches à composants [AG97 ; MT00] : les composants, les services et les assemblages. Mais Kmelia se distingue d'autres approches par certaines caractéristiques spécifiques. Les composants Kmelia sont abstraits, indépendants de leur environnement et par conséquent non directement exécutables. Kmelia permet de modéliser des composants logiciels, des architectures logicielles (assemblages) avec leurs propriétés. La hiérarchisation des services et des composants est basée sur l'encapsulation et elle permet une bonne lisibilité, la flexibilité et une bonne traçabilité dans la conception des architectures [AAA06b]. Puisqu'il est abstrait, Kmelia peut servir de modèle commun pour l'étude de propriétés (interopérabilité, composabilité) de modèles à composants et services. Les modèles doivent être raffinés vers des plateformes d'exécution pour être opérationnels.

Dans la conception de Kmelia, nous avons d'abord un modèle de base [AAA06d] comme noyau simple, avec très peu de concepts (composant, service, assemblage). Ce modèle de base a été successivement enrichi avec une couche **protocole** [AAA07a] permettant de définir des enchaînements licites de services. Cette extension repose sur la spécialisation des services. Ensuite une extension aux **services partagés** (induisant des canaux multipoints) et à la communication multiple est proposée dans [AAA08]. Les données, assertions et expressions sont décrites en utilisant le langage de données introduit dans [AAM09]. Il couvre les types et opérateurs usuels (entiers, booléens, énumérations, structures, tableaux et ensembles). Les prédicats sont étiquetés *e.g.* @predName. L'utilisateur peut déclarer ses propres types dans les spécifications de composants ou dans des bibliothèques comme COCOMELIB. La version du modèle présentée ici est conforme à celle de [And+09].

3.2.1 Notations formelles et définition préalable

Nous utilisons par la suite des notations mathématiques inspirées du langage Z [AV01b]. Par exemple $X \leftrightarrow Y$ décrit l'ensemble des relations de X dans Y ($x \mapsto y$ dénote une paire (x, y) membre de la relation) ; $X_1 \uplus X_2$ est l'union disjointe de deux ensembles ; $X \nrightarrow Y$ est l'ensemble des fonctions partielles de X dans Y ; $X \rightarrow Y$ est l'ensemble des fonctions totales de X dans Y ; $X \rightarrowtail Y$ est l'ensemble des bijections partielles de X dans Y ; id est la relation identité. Si r est une relation ($r : X \leftrightarrow Y$), $\mathrm{dom}(r)$ et $\mathrm{ran}(r)$ représentent respectivement le domaine et le co-domaine de la relation r ; $r(\!|A|\!)$ est l'image relationnelle de l'ensemble $A \subseteq X$; $E \lhd r$ et $r \rhd F$ représentent la restriction respectivement du domaine et du co-domaine de la relation r avec $E \subseteq X$ et $F \subseteq Y$.

L'espace d'états représente une déclaration de variables contrainte par des types et un prédicat du premier ordre. Il est défini à la fois pour les composants et pour les services.

Définition 3.1 (Espace d'états)
Un espace d'états $\mathcal{W}$ définit un ensemble de variables contraint par un invariant : $\mathcal{W} = \langle T, V, type, Inv, Init \rangle$ où T est un ensemble de types, V un ensemble de variables, $type : V \rightarrow T$ la fonction qui associe les variables à leur type, Inv un invariant défini sur V et $Init$ l'initialisation des variables de V.

Dans la suite, on suppose $\mathcal{N}$ un ensemble fini de noms et $\mathcal{M}$ l'ensemble des noms de *messages* tel que $\mathcal{M} \subseteq \mathcal{N}$.

3.2.2 Composants dans Kmelia

Un **composant** (type) est défini par un espace d'états et un ensemble de services accessibles par leur nom. On distingue les *services offerts* qui réalisent des fonctionnalités et les *services requis* qui déclarent les besoins du composant. L'interface du composant liste les services (offerts et requis) accessibles dans le contexte du composant. Elle doit être cohérente avec la description des services : (1) les services offerts et requis ont des noms distincts et (2) les services de l'interface du composant sont un sous-ensemble des services décrits dans le composant. Formellement un composant (type) C est un quadruplet $\langle \mathcal{W}, \mathcal{I}, \mathcal{D}, \nu \rangle$ avec :

```
COMPONENT C1
  INTERFACE
    provides : <ServName list>
    requires : <ServName list>
  //espace d'etats du composant
  TYPES
        <Type Defs>
  VARIABLES
        <Var list>
  INVARIANT
        <Predicate>
  INITIALIZATION
      // affectations des variables
  SERVICES // voir detail des services
  END_SERVICES
```

Listing 1 – Structure d'un composant

- $\mathcal{W}$ l'espace d'états du composant selon la définition 3.1.
- $\mathcal{I} \subseteq \mathcal{N}$ l'interface du composant, partitioné en deux ensembles finis disjoints $\mathcal{I} = \mathcal{I}^P \uplus \mathcal{I}^R$ où ^{P} correspond aux services offerts (*provided*) et ^{R} correspond aux services requis (*required*).

— $\mathcal{D}$ est l'ensemble de descriptions de services (*cf.* Section 3.2.3) qui inclut les services offerts ($\mathcal{D}^P$) et les services requis ($\mathcal{D}^R$) en partition ($\mathcal{D} = \mathcal{D}^P \uplus \mathcal{D}^R$) conforme à celle de $\mathcal{I}$, c'est-à-dire
(1) $\text{dom}(\nu \triangleright \mathcal{D}^P) \cap \text{dom}(\nu \triangleright \mathcal{D}^R) = \emptyset$
(2) $\mathcal{I}^P \subseteq \text{dom}(\nu \triangleright \mathcal{D}^P) \wedge \mathcal{I}^R \subseteq \text{dom}(\nu \triangleright \mathcal{D}^R)$.
— ν est une fonction qui associe des descriptions aux noms de services ($\nu : \mathcal{N} \rightarrowtail\!\!\!\!\!\rightarrow \mathcal{D}$).

Portée des variables et observabilité de l'espace d'états d'un composant

Dans le but de permettre la conception et la composition des composants de façon indépendante des contextes précis d'utilisation, nous avons introduit dans Kmelia une notion d'*observabilité* de l'état des composants. En complément de l'interface publique d'un composant, son état peut être observé par des services ou des composants qui l'utiliseraient ou des composites qui l'encapsuleraient. Par conséquent, l'ensemble des variables d'état d'un composant est partitionné en deux sous-ensembles, l'un contenant les variables observables et l'autre contenant les variables non observables. Cette partition agit aussi sur la formation de l'invariant des composants.

3.2.3 Services dans Kmelia

La notion de service est centrale dans Kmelia ; les services sont au cœur de la construction des composants, de leur assemblage et des interactions entre composants. Un **service** modélise une fonctionnalité élémentaire ou complexe. Outre sa signature fonctionnelle, un service est constitué d'une interface (à l'image de l'interface d'un composant, c'est l'ensemble des services dont il dépend), d'un espace d'états, d'un contrat sous la forme de pré/post-conditions et éventuellement d'un comportement dynamique. Un **service interne** d'un composant est un service offert qui n'est pas dans l'interface du composant. Il est appelable par un autre service du même composant. Nous qualifions de **sous-service** un service offert d'un composant déclaré dans l'interface d'un autre service du même composant et appelable au sein de ce service.

```
provided aService_1
        (<param>) : <ResultType>
   Interface
     subprovides : <Serv list>
     calrequires : <Serv list>
     extrequires : <Serv list>
     intrequires : <Serv list>
   Pre <Predicate>
   Variables # espace d'etat local
      <Var list>
   Initialization
      // affectations
   Behavior
     Init <initial state>
     Final <final states>
     //eLTS
     {<transition list >}
   Post <Predicate>
End
required aService_2 ()
   // defini de la meme maniere
```

Listing 2 – Structure d'un service

Formellement un *service* s d'un composant (type) C [3] est défini par un quintuplet $\langle \sigma, \mathcal{IS}, Cont, \mathcal{W}^L, \mathcal{B} \rangle$ où

- $\sigma = \langle s, param, ptype, Tres \rangle$ est la signature du service dans laquelle s est le nom du service, $param$ un ensemble de paramètres, $ptype : param \to T$ une fonction de typage des paramètres et $res \in T$ le résultat du service ;
- $\mathcal{IS}$ est l'interface du service (*cf.* Section 3.2.3) ;
- $Cont = \langle Pre, Post \rangle$ le contrat de service comprenant Pre la pré-condition et $Post$ la post-condition ;
- $\mathcal{W}^L$ est l'espace d'états local (definition 3.1) utilisé uniquement dans le comportement dynamique $\mathcal{B}$;
- $\mathcal{B}$ est le comportement dynamique défini sous forme d'un système de transitions étendu (*cf.* Section 3.2.3).

Interface et dépendance de services

Les services sont des entités complexes, construites sur une structure de composition hiérarchique (*cf.* Section 3.3.1) et une structure de délégation (*cf.* Section 3.3.2). Ces structures apparaissent explicitement dans l'interface des services. L'interface du service $\mathcal{IS}$ est définie par un triplet $\langle \mathcal{DI}, \mu, \mathcal{W}^V \rangle$ où

- $\mathcal{DI}$ est la **dépendance du service** (*service dependency*) ; elle est composée des services dont dépend le service courant. $\mathcal{DI}$ est un 4-uplet $\langle sub, cal, req, int \rangle$ d'ensembles disjoints tels que $sub \subseteq \mathrm{dom}(\nu \triangleright \mathcal{D}^P)$ (resp. $cal \subseteq \mathrm{dom}(\nu \triangleright \mathcal{D}^R$, $req \subseteq \mathrm{dom}(\nu \triangleright \mathcal{D}^P)$, $int \subseteq \mathrm{dom}(\nu \triangleright \mathcal{D}^P)$) contient les noms des services offerts (resp. ceux requis de l'appelant, ceux requis de n'importe quel composant, les services internes) dans le cadre d'un appel à s.
- μ est un ensemble de signatures de messages $\mathcal{M} \leftrightarrow (param \times ptype)$, $param$ et $ptype$ sont définis comme dans la signature du service ;
- $\mathcal{W}^V$ est un *espace d'états virtuel* (*cf.* Section 3.2.3).

Comportement dynamique de services

Le comportement $\mathcal{B}$ d'un service s est un système de transitions étendu (*extended labelled transition system* - eLTS). L'extension porte sur l'utilisation d'annotations d'états et de transitions, référençant des sous-services offerts. Formellement $\mathcal{B}$ est un eLTS défini par un 6-uplet $\mathcal{B} = \langle S, L, \delta, \Phi, S_0, S_F \rangle$ où S est l'ensemble des états du service s, L est un ensemble d'étiquettes de transitions, δ est la relation de transition ($\delta \in S \times L \to S$), S_0 est l'état initial ($S_0 \in S$) et S_F l'ensemble fini des états finaux ($S_F \subseteq S$) et Φ est la fonction d'annotation d'états ($\Phi \in S \to sub_s$). Une *étiquette* est une combinaison d'actions ([guard] action$*$) ou bien une annotation de transitions (*cf.* Section 3.3.1). Une *action* est une expression Kmelia. Elle est qualifiée d'*action de communication* si elle implique un échange entre deux services (pour appeler ou terminer un service, pour en-

3. et par extension un service d'un composant c (instance) de type C, noté $c : C$.

voyer ou recevoir un message). Une *action élémentaire*, par exemple une affectation, est une expression qui n'implique pas d'autres services et n'utilise donc pas de canaux de communication. La syntaxe des actions de communication est inspirée du langage CSP : channel(! | ? | !! |??) message(param∗). Un *canal de communication* désigne un lien vers un service de la dépendance $\mathcal{DI}$ du service s. Ce lien est défini lors de l'assemblage (*cf.* Section 3.2.4). Parmi les noms de canaux, ___CALLER désigne le service appelant, ___SELF désigne un service du même composant, _req désigne le service requis req.

Déroulement de services

La sémantique opérationnelle d'un service est induite par son eLTS. Un déroulement d'un service est une trace de son système de transition. Lorsqu'il n'y a pas de communication, avec un autre service, les transitions se déroulent comme elles sont décrites et selon l'état du composant. Les transitions sont atomiques, mais leurs enchaînements peuvent être interrompus. Un service offert, lorsqu'il est appelé, devient actif et se déroule selon son système de transition, depuis son état initial jusqu'à son état final. Par défaut, des services d'un même composant peuvent être actifs simultanément, leur déroulement est alors entrelacé. Cependant, un service peut être décrit comme non interruptible, dans ce cas son déroulement doit atteindre l'état final avant de s'arrêter.

Service requis et espace d'états virtuel

Dans Kmelia l'accent est mis sur l'indépendance des définitions des composants : les besoins d'un composant, exprimés en termes de services requis, doivent donc être suffisamment précis et complet pour exprimer un contrat de clientèle. Un tel contrat facilite d'une part la recherche de service pouvant satisfaire ce requis et d'autre part la vérification de conformité des besoins en isolant les vérifications internes au composant des vérifications externes. Un service requis servR d'un composant C_r est une abstraction d'un service qui est offert par un autre composant *a priori* inconnu du composant C_r. Pour poser des hypothèses précises sur l'éventuel fournisseur d'un service requis, nous avons introduit la notion d'*espace d'états virtuel* $\mathcal{W}^V$. L'espace d'états virtuel caractérisant un composant C dans un service requis d'un autre composant C_r est décrit par des variables et un invariant qui sont supposés compatibles avec les variables *observables* du composant C.

3.2.4 Assemblages dans Kmelia

Les composants Kmelia sont assemblés ou composés en liant leurs services. Dans un **assemblage**, les services requis par certains composants sont liés (connectés) aux services offerts par d'autres composants. Ces liaisons, appelées **liens d'assemblage**, établissent des canaux abstraits implicites pour les communications entre services. Dans le modèle de base les canaux sont point-à-point et bidirectionnels. Il n'y a pas de restrictions à la profondeur des sous-services et sous-liens d'un assemblage. Pour plus de *flexibilité*, nous

permettons l'assemblage partiel de composants : il n'est pas nécessaire de lier tous les services offerts et tous les services requis, pourvu que ces services ne soient pas invoqués. Ainsi, on peut disposer d'un composant riche et complexe mais l'utiliser partiellement pour ne pas développer et tester un nouveau composant.

3.2.5 Un exemple de spécification en Kmelia

Illustrons à présent les notions de base du langage Kmelia à travers un cas d'étude inspiré du CoCoME (*Common Component Modelling Example*) ayant servi à comparer des modèles à composants [Rau+08]. L'application est un système de vente en caisse avec les dispositifs usuels (caisse, scanner, imprimante, lecteur de carte...). Le processus de vente inclut des fonctionnalités directement liées aux achats du client (la lecture optique des codes de produits, le paiement par carte ou en espèces, ...) et d'autres tâches telles la gestion du stock et la génération des rapports. L'énoncé original du cas comprend des diagrammes UML (cas d'utilisation, composants, classes).

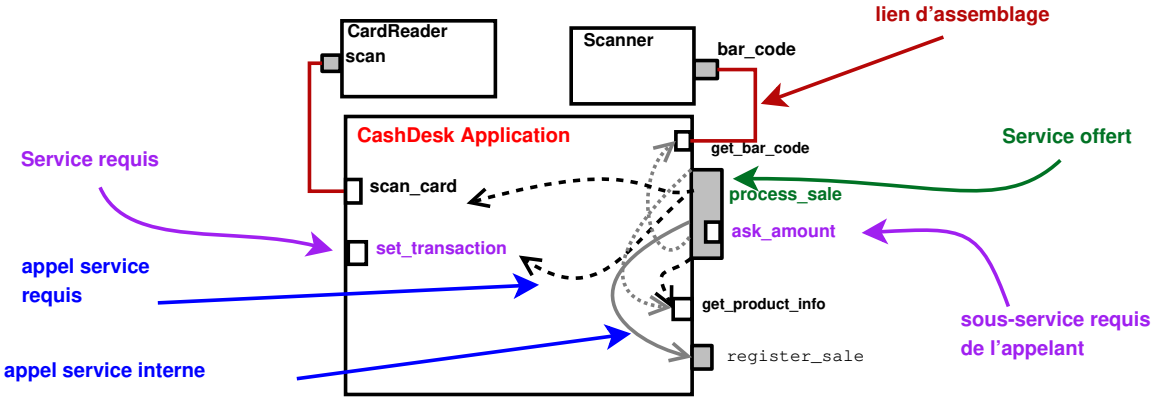


FIGURE 3.1 – Description Kmelia d'un assemblage partiel du cas CoCoME

Nous retenons de l'étude de cas un extrait de la modélisation qui illustre les principaux concepts de Kmelia. Un premier assemblage est mis en évidence dans la figure 3.1 qui se focalise sur le processus de vente `process_sale` pour lequel deux sous-services sont requis (ask_amount et bar_code). Le composant applicatif principal (CashDeskApplication) est relié à des dispositifs d'entrée/sortie, un lecteur de carte CardReader et un lecteur Scanner. Les interfaces des composants exposent des services offerts (rectangles grisés) et des services requis (rectangles blancs), éventuellement reliés par des liens d'assemblage. Par exemple le service offert scan est lié au service requis scan_card. Les flèches à l'intérieur des composants représentent les appels de service et donc leur dépendance de services *e.g.* le service process_sale. Le composant CashDeskApplication offre un service de vente process_sale. Ce dernier fait appel à d'autres services tels que ask_amount() requis de son service appelant, ou bien set_transaction, scan_card requis d'autres composants.

La syntaxe de Kmelia est illustrée dans les listings 3 et 4.

```
COMPONENT CashDeskApplication
INTERFACE
    provides : {process_sale , register_sale}
    requires : {get_product_info, set_transaction, scan_card}
USES {COCOMELIB}
TYPES
    SALE_STATE :: enum {open, close}
CONSTANTS
    isnull : Integer := 0
VARIABLES
  obs list_id : setOf Integer;
  obs state : SALE_STATE
  # ...
SERVICES #---------- services offerts ----------
provided process_sale(id : Integer) : Boolean
  //...
provided register_sale(item : OrderTO; prod : ProductTO) : Boolean
  //...
#---------- services requis ----------
required get_product_info(prod_id : Integer) : ProductTO
End
required scan_card() : String
End
required set_transaction(credit_info : String) :Boolean
End
required get_bar_code() : Integer
End
required ask_amount() : Integer
End
```

Listing 3 – Spécification Kmelia : CashDeskApplication

```
provided process_sale(id : Integer) : Boolean
Interface
    // subprovides : {}
    calrequires : {ask_amount}
    extrequires : {get_bar_code,get_product_info, set_transaction, scan_card}
    intrequires : {register_sale}
Pre
    (id in list_id) && (state = open)
Variables
    prod_id, total, amount_var, rest : Integer;
    authorisation : Boolean;
```

```
    credit_info : String; prod_info : ProductTO;
    order : OrderTO;
    payment_mode : PaymentMode
Behavior
    //−−−−−−− comportement (eLts) du service
Post
    state = open
```

Listing 4 – Spécification Kmelia : process_sale

Le Listing 3 décrit l'interface du composant CashDeskApplication, son état et ses services, prolongé par le listing 4 qui décrit le service process_sale avec une interface (la dépendance entre services), des assertions (pre/post) et un comportement (eLTS). Ce dernier est représenté par le système de transitions de la figure 3.2. Dans la description textuelle des services, une transition $((e_1,\ label),\ e_2)) \in \delta$ du eLTS du service est notée e1 −−−label−−−> e2.

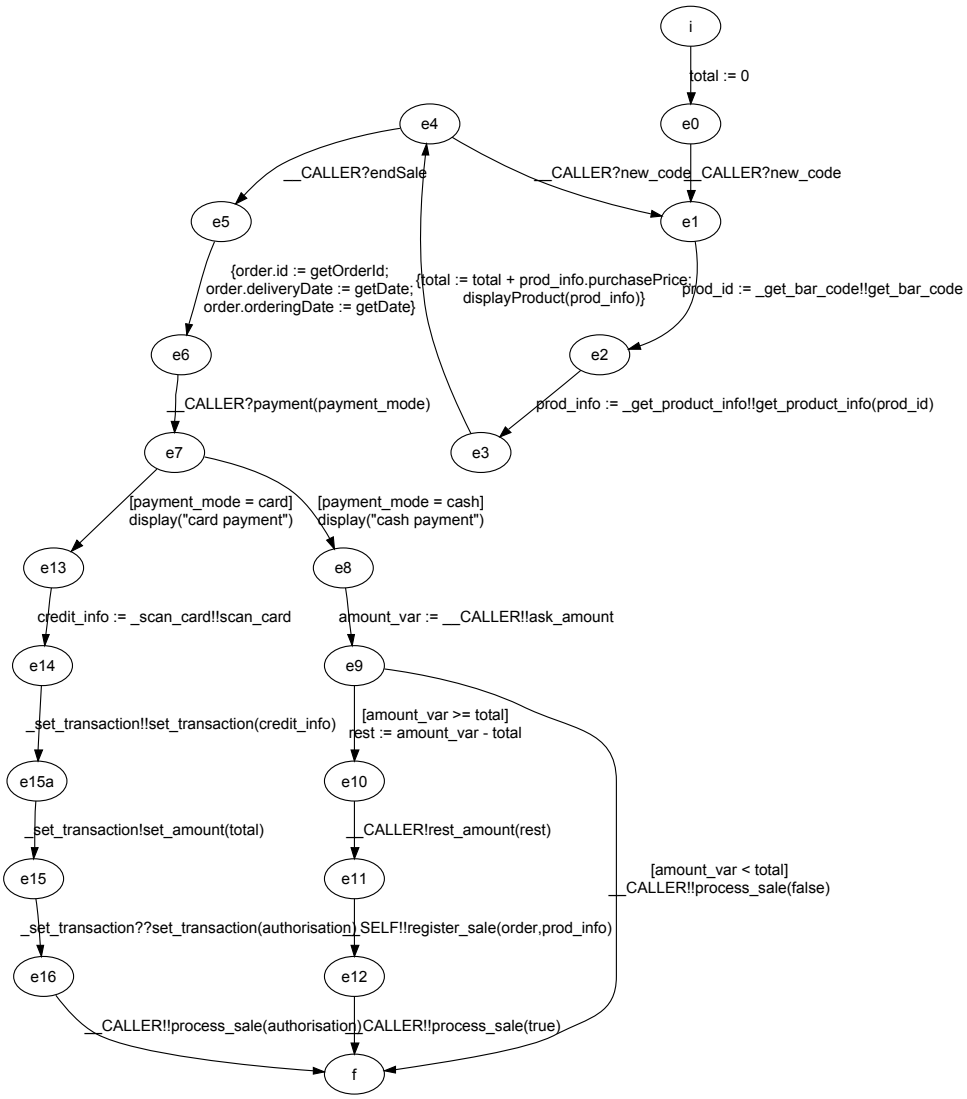


FIGURE 3.2 – Service *process_sale* (extraction `COSTO/kml2latex`)

3.3 Composition dans le modèle Kmelia

La composition est l'opération générique fondamentale de construction des composants et systèmes. Elle prend diverses formes en fonction du type de construction visée. On peut imbriquer un service/composant dans un autre (composition verticale) on peut lier un service/composant à un autre pour échanger (composition horizontale). Les deux types sont pris en compte dans Kmelia.

3.3.1 Composition verticale de services

La composition verticale de services repose sur l'ajout de *points d'expansion* dans le comportement $\mathcal{B}_s$ d'un service s qui permettent d'invoquer un (sous-)service ss dans le contexte de s et qui sont expansés lors des vérifications de compatibilité de services. Le sous-service ss doit être listé dans la dépendance subprovides de s. Les points d'expansion sont notés sous forme d'annotation d'états ou de transitions (*cf.* Section 3.2.3) selon que l'inclusion du service est *optionnelle* ou *obligatoire*.

Soit la description du Listing 5 d'un service offert (servP). subServ1 et subServ2 sont des sous-services de servP, c'est-à-dire des services offerts dans le cadre de servP. Le sous-service subServ1 peut être invoqué dans l'état e2 uniquement, il est *optionnel*, il peut être appelé par l'appelant de servP. A la fin du déroulement de subServ1, le contrôle revient dans l'état e2 ; il est possible d'itérer l'appel de subServ1. Le sous-service subServ2 lui, doit être invoqué dans l'état e1, il est *obligatoire*, (il doit être appelé par l'appelant de servP). Tout se passe comme si on descendait en profondeur sur un nœud du système de transition du service initial ; d'où le terme de composition verticale. Il y a une variante de chacun de ces deux cas de composition verticale. Une variante du cas optionnel (notée <|subServ1|>) et une du cas obligatoire (notée [| subServ1|]) qui proposent une inclusion des sous-services sans appel de service de la part de l'appelant, ni de retour de valeurs. Les systèmes de transition sont alors composés par dépliage de l'un dans l'autre au niveau des nœuds ou transitions.

```
provided servP()
Interface
   subprovides : {subServ1, subServ2}
...
Behavior
   Init  i
   Final f
{   i -- a1 --> e1,
   e1 -- [[subServ2]] --> e2,
      # obligatoire sur e1-e2
   e2 <<subServ1>>,
      # disponible dans l'etat 2
   e2 -- a2 --> e3,
   e3 -- a3 --> f
}
End
```

Listing 5 – Sous-service

Un exemple est donné avec le cas CoCoME pour la description du service de vente sell du Listing 6. Le sous-service amount (Listing 7) est invocable de manière facultative dans l'état e14.

```
provided sell ()
Interface
   subprovides : {amount}
   extrequires : {ask_sale}
Variables # local to the service
   id : Integer; mode : PaymentMode; money : Integer; sale_success : Boolean
Behavior
   Init i
   Final f
{    i -- {display("New sell, please enter your cashier identifier ");
        id := readInt() } --> e10,   # call an internal action
   e10 -- _ask_sale!!ask_sale(id) --> e11,
   e11 -- _ask_sale!start_sale() --> e12,
   e12 -- _ask_sale!end_Sale() --> e13,
   e12 -- _ask_sale!new_code() --> e12,
     i -- {display("please choose your payment mode");
           mode := readPaymentMode() } --> e10, # call an internal action
      e13 -- _ask_sale!payment(mode) --> e14,
   e14 <<amount>>,      # subservice provided in state e14 only
   e14 -- [mode = cash]_ask_sale?rest_amount(money) --> e14,
   e14 -- _ask_sale??ask_sale(sale_success) --> f
}
End
```

Listing 6 – Spécification Kmelia : Cashier:: sell

```
provided amount() : Integer
Variables # local to the service
   value : Integer
Behavior
   Init i     # initial state
   Final f    # final state
{   i -- value := readInt() --> e1,
   e1 -- ___CALLER!!amount(value) --> f }
End
```

Listing 7 – Spécification Kmelia : Cashier::amount

La composition verticale sert aussi à définir des *protocoles* dans Kmelia c'est-à-dire des enchaînements licites de services. Les protocoles sont des modes d'emploi pour les composants [AAA07a]. Les protocoles existent dans tous les modèles qui modélisent la dynamique des interactions entre composants, mais ces modèles proposent des descriptions "à plat" sans composition hiérarchique de services, comme nous l'avons expliqué dans [AAA07d].

3.3.2 Composition horizontale de services et liaison

La composition horizontale d'un service sv est basée sur la structure de délégation induite par la dépendance de services de sv (*cf.* Section 3.2.3). Un service listé dans calrequires doit être fourni par le composant lié au service sv (appelant). Un service listé dans extrequires doit être fourni par un composant lié au composant dans lequel sv est défini. Un service listé dans intrequires doit être fourni par le composant de sv. Un lien (ou liaison) est un opérateur simple de connexion qui établit en quelque sorte les canaux de communication désignés dans les services. Comparé à d'autres modèles, nous n'introduisons pas de ports ni de connecteurs (*cf.* Section 3.3.6).

Soit $\mathcal{C}$ un ensemble de composants $c_k : C_k$ pour $k \in 1..n$ avec $C_k = \langle \mathcal{W}_k, \mathcal{I}_k, \mathcal{D}_k, \nu_k \rangle$ (*cf.* Section 3.2.2). Un **lien** entre deux services est défini par un quadruplet de noms de composants et de services tel que : (1) les noms de service sont ceux de services définis dans leur composant, (2) un service n'est pas lié à lui-même.

$$BaseLink \subseteq (\mathcal{C} \times \mathcal{N} \times \mathcal{C} \times \mathcal{N})$$
$$(1) \qquad \forall (c_i, n_1, c_j, n_2) : BaseLink \bullet n_1 \in \operatorname{dom} \nu_i \wedge n_2 \in \operatorname{dom} \nu_j$$
$$(2) \qquad \forall c_i : \mathcal{C},\ n_1 : \operatorname{dom} \nu_i \bullet (c_i, n_1, c_i, n_1) \notin BaseLink$$

où $\operatorname{dom} \nu_i$ est le domaine de la fonction de nommage des services, c'est-à-dire l'ensemble des noms de services du composant C_i.

Un sous-lien est un lien défini dans le contexte d'un autre lien. La notion de sous-lien est relative à la fois à la dépendance de service sur l'appelant (ensemble *cal* de $\mathcal{DI}$) et celle des sous-services (ensemble *sub* de $\mathcal{DI}$). Il n'y a pas de dépendance circulaire entre liens (3).

$$SubLink : BaseLink \leftrightarrow BaseLink$$
$$(3) \qquad \forall (l_1, l_2) \in SubLink \bullet (l_2, l_1) \notin SubLink^*$$

où $A \leftrightarrow B$ désigne une relation entre les ensembles A et B et $SubLink^*$ est la fermeture transitive (non réflexive) de la relation $SubLink$.

Dans Kmelia les liens et sous-liens apparaissent sous forme de **liens d'assemblage** dans la composition horizontale et de **liens de promotion** dans la composition verticale de composants.

3.3.3 Composition horizontale de composants

Partant d'un ensemble de composants, la composition horizontale est l'opération qui consiste à construire un assemblage en reliant les services requis à des services offerts. L'*assemblage de composants* en Kmelia correspond à l'architecture de composants dans d'autres modèles [SG96 ; BHP06]. Les connecteurs sont simplement des liens d'assemblage (liaisons dans [Bru+06 ; BHP06]). Les spécificités dans Kmelia sont : (i) on relie des services et non des composants, des ports ou des interfaces, (ii) un lien est hiérarchique si les services associés sont composés d'autres services, (iii) le lien met en évidence un contrat

de clientèle et pas simplement une correspondance syntaxique. La structure hiérarchique de sous-liens doit être cohérente avec la composition verticale des services : les sous-services offerts dans le cadre d'un service apparaissent dans les sous-liens.

Assemblage de composants

Comme indiqué dans la section 3.2.4, un assemblage est un ensemble de composants liés sur leurs services par des *liens d'assemblage*. Formellement, un *assemblage* (type) de composants est un quintuplet $A = \langle \mathcal{C},\ alinks,\ subs,\ vmap,\ mmap \rangle$ où $\mathcal{C}$ est un ensemble de composants $c_k : C_k$ pour $k \in 1..n$, $alinks$ est un ensemble de liens d'assemblage entre services de $\mathcal{C}$, $subs$ est une relation d'inclusion de liens, $vmap$ est une fonction de correspondance de variables d'état et $mmap$ est une fonction de correspondance de messages. Nous détaillons les liens ($alinks$, $subs$) et les correspondances ($vmap$, $mmap$) dans les sections suivantes.

Liens d'assemblage Un *lien d'assemblage* entre un service n_1 d'un composant C_1 et un service n_2 d'un composant C_2 est une abstraction d'un canal de communication reliant n_1 à n_2. Chacun des services n_1 et n_2 est dans une des interfaces des composants.

$$
\begin{array}{ll}
& alinks \subseteq BaseLink \wedge \\
(1) & (\forall (c_i, n_1, c_j, n_2) : alinks \bullet c_i \in \mathcal{C} \wedge c_j \in \mathcal{C} \wedge \\
(2) & \quad ((n_1 \in \mathcal{I}_i^P \wedge n_2 \in \mathcal{I}_j^R) \vee (n_1 \in \mathcal{I}_i^R \wedge n_2 \in \mathcal{I}_j^P))) \\
& subs \subseteq SubLink \wedge \\
(3) & (\mathrm{dom}(subs) - \mathrm{ran}(subs)) \subseteq alinks \wedge \\
(4) & (\forall ((c_i, n_1, c_j, n_2) \mapsto (c_k, n_3, c_l, n_4)) \in subs \bullet c_i = c_k \wedge c_j = c_l) \wedge \\
(5) & (\forall (c_i, n_1, c_j, n_2) : \mathrm{ran}\, subs \bullet ((\nu_i(n_1) \in \mathcal{D}^P{}_i) \ \mathrm{xor}\ (\nu_j(n_2) \in \mathcal{D}^P{}_j)))
\end{array}
$$

Les contraintes ont la signification suivante. Les composants des liens sont les composants de l'assemblage (1). Il y a symétrie *requis-offert* dans un lien (2). Les sous-liens dépendent *in fine* des liens d'assemblage de plus haut niveau (3) et portent sur les mêmes composants (4). Les services offerts sont liés aux services requis (2 et 5). Les liens établissent donc des ponts de nommage explicites entre services ; pour avoir des assemblages complètement définis on doit aussi faire correspondre les espaces d'état et les messages.

Correspondances de contexte dans les liens d'assemblage Dans le cadre des liens d'assemblage, le spécifieur doit établir deux correspondances afin d'assurer la cohérence des liens et donc de l'assemblage global. Ces correspondances sont une forme d'adaptation explicite, nous avons discuté d'autres formes dans [AAA06a].

La première correspondance établit explicitement la relation entre le contexte virtuel (*cf.* Section 3.2.3) du service requis (si un tel contexte est défini) et le contexte observable du composant du service offert : c'est la correspondance de contexte. Elle illustre bien la séparation des besoins de leur satisfaction, et donc l'indépendance des composants : le service requis est défini sur un ensemble d'hypothèses (un espace virtuel) qu'on confronte

avec un composant concret (des contraintes réelles). Soit un service requis sr d'un composant cr de type CR, lié à un service offert sp d'un composant cp de type CP. Les variables de l'espace d'état virtuel (vV_{sr}) de sr sont mises en correspondance avec les variables *observables* de cp (V^O_{CP}) par une fonction totale $vmap : vV_{sr} \rightarrow exp(V^O_{CP})$ où $exp(X)$ désigne une expression sur les variables de X. Formellement, chaque lien contient une fonction $vmap : V \nrightarrow Exp(V)$ de concrétisation (ou une relation d'abstraction en raffinement).

$$
\begin{array}{ll}
& vmapVar : BaseLink \leftrightarrow V \\
& vmapExp : (BaseLink \times V) \nrightarrow exp(V) \\
(6) & \mathrm{dom}(vmapVar) \subseteq (alinks \cup subs) \wedge \mathrm{dom}(vmapExp) = vmapVar \wedge (\\
& \quad \forall (c_i, n_1, c_j, n_2) : BaseLink \mid (c_i, n_1, c_j, n_2) \in \mathrm{dom}(vmapVar) \bullet \\
(7) & \qquad vmapVar(\!|\{(c_i, n_1, c_j, n_2)\}|\!) = V^V_{\nu(n_2)} \wedge \\
(8) & \qquad (\forall v : V^V_{\nu(n_2)} \bullet var(vmapExp((c_i, n_1, c_j, n_2), v)) \subseteq V^O_{C_i}))
\end{array}
$$

où $exp(V)$ désigne l'ensemble des expressions sur V et leurs variables $var(exp(V)) = V$. Les contraintes expriment le fait que la correspondance se fait sur les liens de l'assemblage considéré (6), que toutes les variables du service requis n_2 ont une expression (7) constituée à partir des variables observables du composant de n_1 (8).

La seconde correspondance permet de relier explicitement les identifiants des messages utilisés dans les services liés : c'est la correspondance de messages (*message mapping* dans [And+09]). Les paramètres des messages doivent conserver leur ordre ; dans le cas contraire il s'agit d'un problème d'adaptation que nous avons traité dans [AAA06a]. Formellement, pour un lien donné, chaque nom de message de sr est associé à un nom de message de sp par la bijection totale $mmap : mname_{sr} \rightarrowtail\!\!\!\!\!\rightarrow mname_{sp}$.

$$
\begin{array}{ll}
& mmap : BaseLink \nrightarrow (\mathcal{M} \rightarrowtail \mathcal{M}) \\
(9) & \quad \mathrm{dom}(mmap) \subseteq (alinks \cup subs) \wedge (\\
& \quad \forall (c_i, n_1, c_j, n_2) \in \mathrm{dom}(mmap) \bullet \\
(10) & \qquad \mathrm{dom}(mmap(c_i, n_1, c_j, n_2)) = \mathrm{dom}(\mu_{\nu(n_2)}) \wedge \\
(11) & \qquad \mathrm{ran}(mmap(c_i, n_1, c_j, n_2)) = \mathrm{dom}(\mu_{\nu(n_2)}))
\end{array}
$$

Les liens sur lesquels portent les correspondances de messages sont des liens de l'assemblage considéré (9), tous les messages du service de n_1 ont une correspondance (10) et tous les messages du service de n_2 ont une correspondance (11). Chaque message a au plus une correspondance du fait de l'injection partielle. Si on souhaite une correspondance plus "lâche" alors il faut faire appel aux techniques d'adaptation.

Sous-liens d'assemblage et dépendances de services Un lien d'assemblage est structuré en concordance avec les dépendances de services (*service dependency*). Si les services établissent des dépendances de type subprovides et calrequires alors des sous-liens sont à définir dans ce cadre, de manière similaire à celle des liens. Soit l'assemblage du Listing 8. Deux composants cp et cr sont assemblés par un lien d'assemblage @la. Le service requis reqServ de cr est *réalisé* par le service offert provServ de cp. Le préfixe **p**−r indique

le sens de lecture du lien. Par la correspondance de messages, les messages nommés msg1 dans provServ sont nommés msgA dans reqServ. Par la correspondance de contexte, la variable var1 du contexte virtuel de reqServ s'exprime en fonction des variables varA, varB... de l'espace d'état observable du composant cp.

```
Assembly
 Components
  cp : CP; cr : CR
 Links // −−− liens d'assemblage −−−
  @la: p−r cp.provServ cr.reqServ
   message mapping
    msg1 = msgA
    ...
   context mapping
    cr.var1 = expr(cp.varA, cp.varB...),
    ...
   sublinks : {lasub}
  // −−−−− sous−liens −−−−−−−
    @lasub: r−p cp.subReq cr.prov
  ...
End // Assembly
```

Listing 8 – Assembly

Les sous-liens sont l'expression de la correspondance des dépendances de services. Ainsi si le service offert provServ requiert le service subReq dans sa dépendance calrequires alors un sous-lien doit être défini vers un service prov offert directement dans l'interface offerte du composant cr ou dans la dépendance subprovides du service reqServ. Les services apparaissant dans la dépendance extrequires du service offert provServ nécessitent la spécification d'un service requis du composant cp et par là même un lien vers un autre serveur (du service offert). Notons que les services apparaissant dans la dépendance intrequires du service offert provServ nécessitent la spécification d'un service offert du composant cp et aucun lien n'est explicité.

Illustration Reprenons l'exemple CoCoME. La figure 3.3 et le Listing 9 representent un assemblage partiel du système de gestion des ventes Trading System qui enrichit celui de la figure 3.1 avec les composants de gestion de persistance Inventory et d'interface Cashier et le consortium bancaire Bank.

```
Assembly
  Components
    cr  : CardReader; s : Scanner ; cda : CashDeskApplication b : Bank; c : Cashier
  Links //////////// liens d'assemblage //////////
    @code: r−p cda.get_bar_code s.bar_code
    @scan: r−p cda.scan_card cr.scan
    @trans: r−p ts.set_transaction b.transaction
  message mapping
   set_amount = transaction_amount
    @sale: p−r ts.process_sale c.ask_sale sublinks : {lamount}
    @lamount: r−p ts.ask_amount c.amount
    ...
End // assembly
```

Listing 9 – Spécification Kmelia : Assemblage du systeme CoCoMe

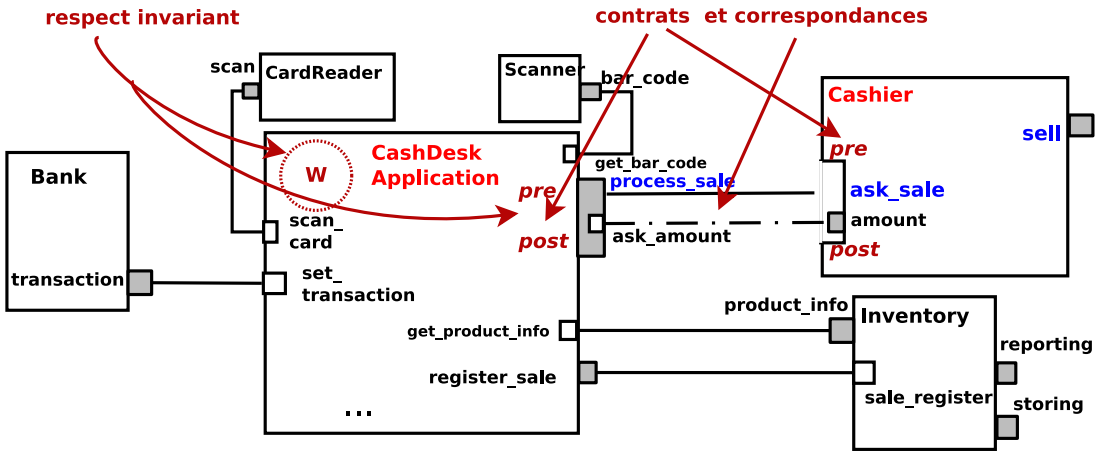


FIGURE 3.3 – Description Kmelia simplifiée d'un assemblage pour le cas CoCoME

Dans la figure 3.3, les traits pleins entre services représentent les *liens d'assemblage* qui associent des services offerts à des services requis (un service requis est « réalisé » par un service offert). Les traits discontinus sont des sous-liens. Les assertions (pré-post/conditions) des services définissent des contrats qui respectent l'invariant défini dans l'espace d'états de leur composants (ou l'espace d'état virtuel pour un service requis). Ces assertions sont reprises pour former une partie du contrat d'assemblage (en plus des compatibilités de signature, de sous-liens et d'interactions).

Interaction intercomposant

La composition horizontale induit l'interaction inter-composant. Considérons dans un premier temps des interactions basées sur des liaisons d'un service/composant à un autre service/composant. L'interaction entre deux services issus de deux composants distincts est la base de l'interaction entre composants ; elle se fait donc de pair à pair. Un service d'un composant, pendant son déroulement, appelle un autre service qui est un requis de son composant. Le déroulement se passe comme si le service requis était remplacé par le service effectif offert par un autre composant. Les deux services se déroulent en parallèle et échangent en communicant sur un **canal abstrait**. Les services interagissent *via* les **communications synchrones ou asynchrones**. C'est la composition horizontale qui définit le support de cette communication.

En se basant sur les actions de communication définies dans la section 3.2.3, une interaction est une communication synchrone établie par une paire complémentaire *envoi de message(!)-réception de message(?), appel de service(!!)-attente du démarrage de service(??), envoi du résultat de service(!!)-attente du résultat de service(??)*. La forme élémentaire des actions de communication a été étendue pour exprimer diverses autres interactions comme on le verra plus loin.

Un service pouvant communiquer avec plusieurs autres services de composants différents, l'interaction entre composants se généralise à une interaction entre plusieurs composants *via* le réseau formé par les services communicants. En effet, le déroulement d'un

service peut entraîner en cascade le déroulement et donc la communication avec d'autres services, d'où la formation d'un réseau de services communicants. Par construction, nous éliminons la formation de boucles dans les assemblages (par composition) donc les communications ne sont pas interblocantes par la présence de cycles. L'interaction globale dans un assemblage est au final une juxtaposition d'interactions de proche en proche.

3.3.4 Composition verticale de composants

La composition verticale des composants consiste en l'encapsulation d'un assemblage dans un composant, le **composite**. Ainsi une succession de *composition verticales* aboutit à une hiérarchie de composants.

Afin de faciliter la réutilisation et le passage à l'échelle, l'encapsulation d'un assemblage dans un composite masque les composants par défaut. Le composite est présenté dans son interface comme tout autre composant. Le composite peut utiliser les variables observables de ses composants directs et les services de leurs interfaces à condition de les promouvoir. L'encapsulation pouvant se faire de manière répétée, la visibilité des espaces d'états se fait de proche en proche, chaque composant déclarant ses variables observables parmi lesquelles peuvent figurer des variables promues. De la même façon, les variables et les services y compris leurs dépendances, peuvent être promus à l'interface du composant résultant de l'encapsulation.

En guise d'illustration, prenons l'architecture du système CoCoME de la figure 3.4 qui restructure l'assemblage de la figure 3.3. Les emboîtements de composants dénotent

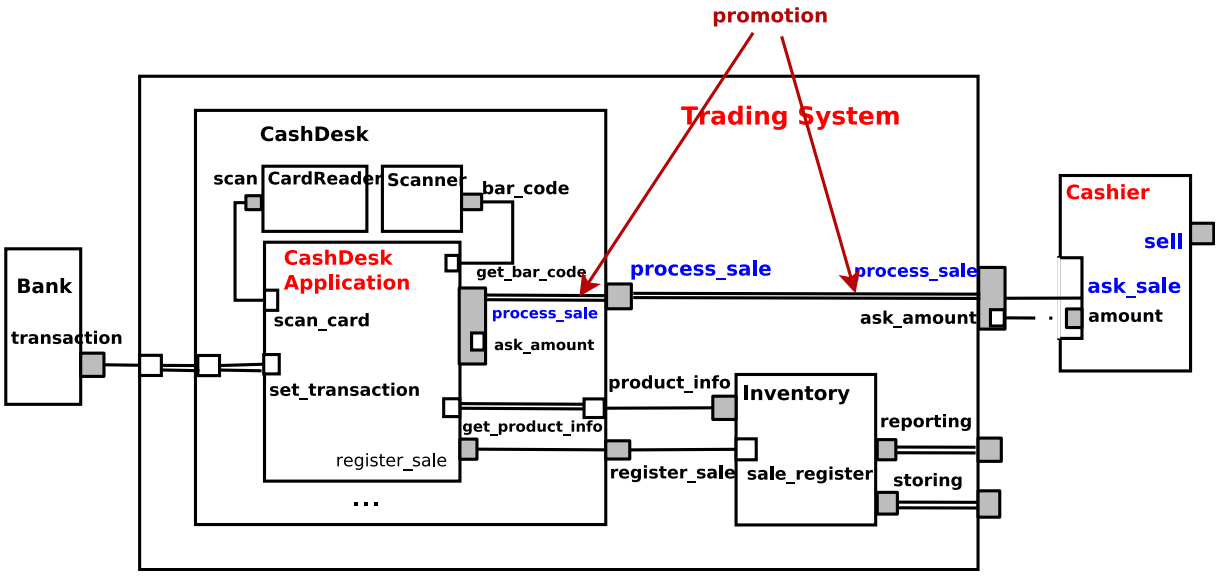


FIGURE 3.4 – Promotion de services pour le cas CoCoME

la relation de composition par encapsulation de composants ; les traits doubles sur les services sont des *liens de promotion* ; un service d'un composant est promu au niveau du composite qui le contient. Par exemple, dans le Listing 10, le service offert process_sale du composant cda : CashDeskApplication est promu au niveau du composite CashDesk, dont

il devient un point d'entrée.

```
COMPONENT CashDesk
INTERFACE
  provides : {process_sale, register_sale}
  requires : {set_transaction, get_product_info}
COMPOSITION
  Assembly
    Components
      cr : CardReader; s : Scanner ; cda : CashDeskApplication
    Links ////////////assembly links/////////
      @code: r−p cda.get_bar_code s.bar_code
      @scan: r−p cda.scan_card cr.scan
  End // assembly
  Promotion
    Links ////////////promotion links/////////
      @sale: p−p cda.process_sale SELF.process_sale sublinks : {lamount}
      @regis: p−p cda.register_sale SELF.register_sale
      @trans: r−r cda.set_transaction SELF.set_transaction
      @prod: r−r cda.get_product_info SELF.get_product_info
          ////////////promotion sublinks/////////
      @lamount: r−r cda.ask_amount SELF.ask_amount
END_COMPOSITION
```

Listing 10 – Spécification Kmelia : Composite CashDesk

La promotion de la variable state : SALE_STATE du composant cda se fait dans la déclaration de l'espace d'état du composite CashDesk par status : SALE_STATE FROM cad.state. Le renommage est facultatif. Le principe d'encapsulation est appliqué strictement : le composite n'a pas plus de droits qu'un composant client lié par assemblage ; pour modifier l'état d'un composant il doit passer par les services de ce composant.

L'indépendance composant/composite permet d'un point de vue méthodologique une conception descendante autant qu'ascendante du système à composants. L'évolution se fait alors en utilisant des règles précises de conformité de la promotion.

Interaction intra-composant

Nous précisons ici, d'une part la manière dont les services, les services internes et les sous-services d'un même composant interagissent et d'autre part la manière dont les services de composants imbriqués interagissent.

— *Composition verticale (cas des sous-services).* Considérons l'interaction entre un service et un de ses sous-services. Un service d'un composant C et ses sous-services ne partagent pas leurs variables locales, mais celles du composant C. Un service peut, lors de son déroulement, évoquer un sous-service. Le sous-service prolonge alors le déroulement de son appelant, il n'y a pas d'interaction entre eux. Un service peut

communiquer avec ses sous-services à travers les variables du composant. L'appel/-retour de service crée/termine cette interaction entre service et sous-service.

— *Composition horizontale (cas des services internes).* Il peut y avoir des interactions entre un service interne et son appelant ; les interactions sont ici basées sur les canaux de communication abstraits associés aux services. L'opérateur d'appel !! (resp. de retour ??) de service débute (resp. termine) ces interactions.

— *Promotion (cas des services de sous-composants).* Lorsqu'un composant C_e est encapsulé dans un composite C_i, pour utiliser un service de C_e, il faut le promouvoir et ensuite l'utiliser comme un service interne.

3.3.5 Composition multipartie de composants

Dans ce qui précède, nous avons considéré des assemblages simples : (1) chaque exemplaire d'un composant est nommé par une variable, (2) les liens sont exclusivement entre deux composants (de type `1-1`). L'exclusivité du lien signifie que si `n` services sont liés à un service, alors on considère réellement `n` liens indépendants. Cependant le modèle a été étendu dans [AAA08] pour autoriser des interactions multiparties. Cette extension répond à un besoin de modélisation des applications, impliquant plusieurs parties, par exemple un système de *chat* avec un modérateur et des participants.

Assemblage multiple, service partagé et liens n-aires

Dans une composition multipartie, on autorise plusieurs exemplaires d'un composant (sous forme de tableau de composants) et des *services partagés* pour permettre les liens `1-n` ou `n-1`. Un service partagé est un service utilisé simultanément, de manière transparente, par plusieurs autres services. Cette extension met en œuvre des primitives de communication multipartie (*cf.* Section 3.3.5). Un tableau de composants de même type C est déclaré par $c[n]$:C, où n est le nombre maximum d'exemplaires de composants. Chaque composant est accessible par son indice dans le tableau $c[i]$.

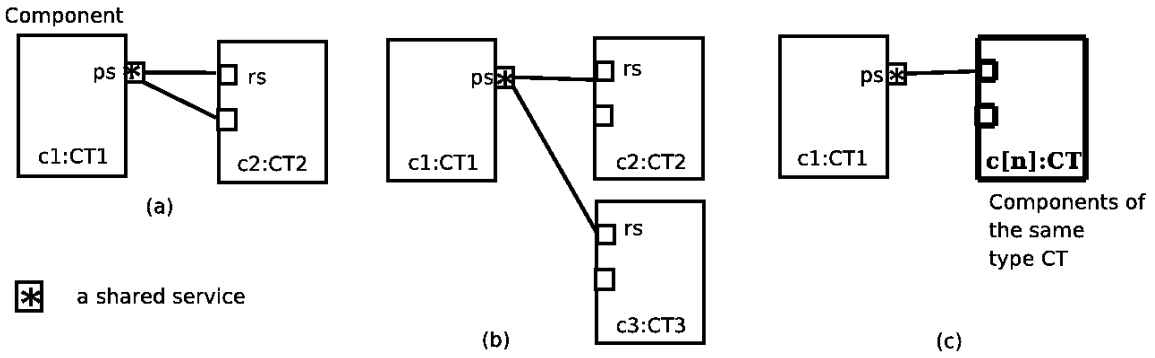


FIGURE 3.5 – Assemblages avec des services offerts partagés

Un *service offert partagé* est lié à plusieurs services requis de plusieurs composants de types quelconques (lien de type `1-n`). Par exemple un service de *chat* en conférence est invoqué par plusieurs participants. Par défaut les appelants sont rangés dans un tableau,

ils ont le même rôle dans la collaboration mais il est possible de leur faire jouer des rôles différents. Dans le *chat*, l'initiateur a des prérogatives. Les appelants n'ont *a priori* aucune connaissance les uns des autres.

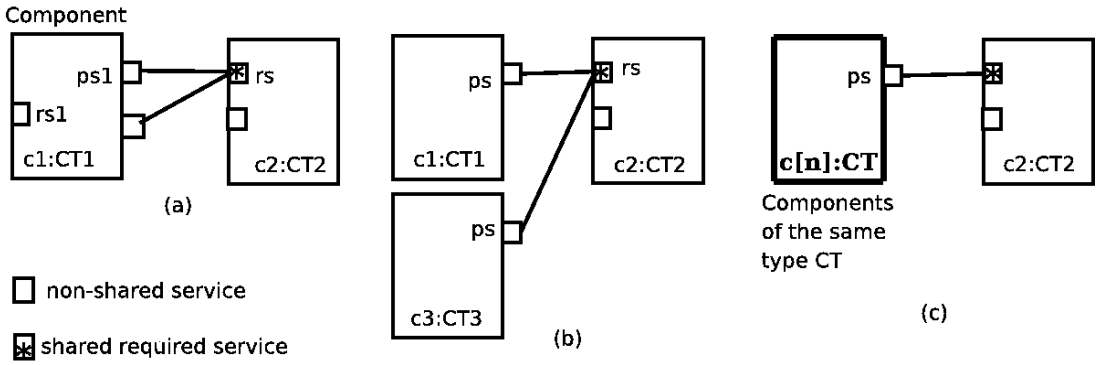


FIGURE 3.6 – Assemblage avec des services requis partagés

Un *service requis partagé* est lié à plusieurs services offerts de plusieurs composants de types quelconques (type `n-1`). Il va donc mettre en concurrence plusieurs fournisseurs. Par exemple, un service web de comparatif peut émettre des requêtes et collecter les résultats de différentes façons (le premier reçu, le meilleur selon un critère donné...).

Interaction multipartie

Lorsqu'il existe des liens d'un service d'un composant vers plusieurs services d'un ou plusieurs composants, il y a potentiellement une interaction entre plusieurs services. On est dans le cadre d'une généralisation des interactions entre services. Les communications et les primitives associées sont adaptées en conséquence. Dans ce contexte, les actions de communications prennent la forme :

channel[<selector>](! | ? | !! |??)message(param∗)

Les valeurs de <selector> sont : `ALL` pour tous les services concernés par la communication, `i` pour un service donné et `:i` pour un service quelconque.

Lorsqu'un service offert servP est partagé par plusieurs services requis servR_i, il y a interaction simultanée entre le comportement du service servP et les comportements respectifs des services ServR_i. Cette interaction se déroule comme suit :

1. les actions élémentaires de tous les services se déroulent de façon indépendante, et sont par conséquent entrelacées ;
2. les actions de communications utilisées dans les comportements des services forcent soit à une synchronisation entre tous les services (c'est le cas quand les opérateurs sont précédés de sélecteurs spécifiques `...[ALL]??...`, `...[ALL]!!...`) soit à une synchronisation entre certains services uniquement (c'est le cas avec `...[:i]?...`).

De la même manière que dans le cas précédent, lorsqu'un service requis partagé est lié à plusieurs services offerts, il y a interaction multiple simultanément ; les actions élémentaires sont entrelacées alors que celles de communication sont régies par la sémantique des opérateurs de communication.

3.3.6 Discussion

La composition en Kmelia se fait à deux niveaux. Au niveau des services, la composition verticale se fait par des points d'expansion, annotant des nœuds ou des transitions, où s'expansent les services indiqués dans l'annotation. Au niveau des composants, la composition verticale réalise des composites par un assemblage encapsulé d'autres composants tandis que la composition horizontale résulte en l'assemblage de composants ; dans les deux cas elle se fait *via* des liens de services qui peuvent ainsi interagir et par conséquent assurer l'interaction entre les composants. La liaison des services est vue comme la composition horizontale de services : c'est une composition parallèle où les services évoluent de façon indépendante ou communiquent *via* le canal abstrait établi par leur liaison.

Kmelia se situe dans la lignée des approches où l'analyse formelle est une préoccupation de premier plan (Wright [AG97], SOFA [BHP06], rCOS [LMS10], Fractal [Bru+06], BIP [GS05])que des approches qui privilégient le codage direct. Ces dernières correspondent aux propositions "industrielles" telles que EJB, .NET (mais aussi SCA [MR09]) qui sont largement utilisées dans des applications réelles de grande taille. Cependant elles ne permettent pas de raisonner formellement au niveau composant ni au niveau des compositions de composants. La composition est traitée de diverses façons selon les caractéristiques de l'approche considérée. La composition horizontale de composants se fait toujours sur un couple d'éléments :

— une entité exhibée par les composants. Cela peut être un port (Wright, BIP), une interface (Sofa, Fractal, rCOS) ou un service (Kmelia, SCA). Il peut même y avoir plusieurs niveaux comme dans UML2 où un port peut proposer plusieurs interfaces.
— une entité établissant la liaison. Cela peut être un simple lien (Sofa, Fractal, rCOS, Kmelia, SCA) ou un connecteur (Wright, BIP). Les connecteurs couvrent diverses fonctionnalités [Med+02].

Composition dans Wright. Le langage de description d'architecture Wright [AG97] constitue un modèle formel de référence pour les architectures à composants. Dans Wright, l'accent est mis sur la composition horizontale, la composition verticale étant déléguée au raffinement. Une architecture est définie par des composants reliés sur leurs ports par des connecteurs. Le comportement des ports est défini par des processus à la CSP. Les connecteurs spécifient des rôles et une coordination (*glue*), là aussi par des processus. L'analyse de la composition se ramène à l'analyse d'un ensemble de processus communicants ; elle est mise en œuvre avec l'outil FDR. L'expressivité du modèle est donc limitée à CSP.

Composition dans SOFA. Les composants SOFA élémentaires ont un comportement (*behaviour protocol*) ; les composants peuvent être composés verticalement, par encapsulation ; l'approche de structuration est descendante, c'est la méthode de conception primordiale préconisée dans SOFA ; ainsi à partir d'un composant de plus haut niveau, on descend vers les composants qui doivent y être encapsulés jusqu'aux composants de plus

bas niveaux. Le comportement des composants composites est semi-automatiquement compilé. Cette encapsulation des composants est comparable à la composition verticale dans Kmelia. La composition horizontale dans SOFA consiste en la composition parallèle des comportements (*protocols*) des composants concernés ; ils communiquent alors par échanges synchrones de messages. Aux spécificités des services près, c'est la même approche que pour la composition horizontale dans Kmelia. Des connecteurs, semi-automatiquement générés permettent de réaliser la composition horizontale des composants SOFA, un tel mécanisme n'existe pas encore dans Kmelia où les liaisons dues à la composition horizontale restent abstraites.

Composition dans Fractal. Même si les composants Fractal sont vus comme des entités à l'exécution, il est possible de les composer. Nous pouvons les examiner sous l'angle dimensionnel vertical ou horizontal. Les modèles Fractal utilisent des codes de programmes (Java par exemple) pour le comportement des composants ; cela limite les possibilités de composition horizontale. La composition par une liaison (*binding*) entre interface client et interface serveur est une composition horizontale ; dans ce cas, les composants Fractal peuvent communiquer par invocation d'opérations (de client à serveur), c'est la *primitive binding*. La composition par *composite binding* lie un nombre quelconque de composants ; elle se ramène en fait à plusieurs liaisons primitives. Dans les deux cas, l'analyse de la compatibilité comportementale n'est pas possible comme c'est le cas avec Kmelia. La composition hiérarchique par encapsulation des composants Fractal est comparable à la composition verticale par encapsulation de composants Kmelia ; en revanche nous n'avons pas identifié une encapsulation de niveau service comme le permet Kmelia.

Composition dans rCOS. La composition horizontale de composants se trouve dans rCOS sous la forme de l'union disjointe (ou composition parallèle) de composants. Cependant dans rCOS, une couche de processus donnant la dynamique des composants est ajoutée aux composants pour ordonnancer leur interaction. En effet les services des composants rCOS sont des actions élémentaires et n'autorisent pas d'interactions contrairement aux services de Kmelia. La composition verticale est assimilable à l'opération de masquage (*hiding*) de rCOS où une union disjointe par exemple se retrouve encapsulée dans un composant. Ici la promotion des services non masqués est systématique, à la différence de Kmelia ou l'encapsulation ne force pas les promotions des services non liés.

Composition dans BIP. BIP (*Behaviour, Interaction, Priority*) adopte une méthodologie à trois couches correspondant au nom BIP, pour la construction correcte des composants hétérogènes : une couche basse où on décrit des comportements sous la forme de réseau de Petri ou d'automate à états ; une couche intermédiaire contenant des connecteurs qui décrivent les interactions entre les comportements du niveau précédent. Enfin une couche haute, consacrée à la description d'un ensemble de règles de priorités pour l'ordonnancement des interactions. La composition de composants BIP est binaire et consiste

en la composition deux à deux des trois niveaux des différents composants. Le résultat est aussi un composant à trois niveaux. La composition est ainsi incrémentale. Comparée à Kmelia, BIP n'est pas multiservice ; les comportements et les assertions sont définis au niveau composant dans BIP alors qu'ils le sont au niveau service dans Kmelia. Les principes de composition sont comparables au niveau des services Kmelia ; on retrouve des principes similaires (composition horizontale avec communication synchrone), les services Kmelia et les comportements des composants BIP sont modélisés par des systèmes états-transitions communicants. La composition est identique à ce niveau ; cependant dans Kmelia les connexions des services sont plus simples - canaux abstraits supports de communication - que dans BIP où ce sont des connecteurs qui définissent le protocole de synchronisation (rendez-vous, diffusion, ...) entre des ports ainsi que le transfert de données associé. L'ordonnancement des interactions dans Kmelia est non déterministe, il n'y a pas de priorité comme dans BIP. En revanche la composition verticale dans BIP est hiérarchique, avec des règles spécifiques de calcul du modèle d'interaction pour le composant résultant ; dans Kmelia elle se résume, dans le cas des services internes, à une composition parallèle sur un canal abstrait.

Composition dans les services. La composition dans les services est un thème largement abordé, notamment à travers l'orchestration, les extensions de BPEL, la composition dynamique, ... [BBG07]. En se limitant à la composition vérifiable par des modèles formels, on s'aperçoit que les propositions sont des extensions de formalismes de la concurrence, à savoir automates, réseaux de Petri ou algèbres processus. Par exemple, dans [SB09], les auteurs formalisent des collaborations en Lotos pour donner une sémantique formelle et la vérifier avec CADP. Par leur nature, la composition de services est surtout vue d'un point de vue dynamique, illustré par le calcul d'orchestration de [ML06], et correspond à la compatibilité comportementale en Kmelia. Inversement, dans [Mil05], les contrats (au sens *design by contract*) sont pris en compte *via* les machines abstraites (à la B) mais pas le comportement dynamique. Kmelia prend en compte les deux aspects. Dans [Bro10], le contrat comporte quatre niveaux (signature, qualité de service, ontologie, comportement) mais pas le contrat par assertion. Kmelia est assez proche de l'architecture composant service (SCA) définie dans [DCL08], dans la mesure où les connexions entre composants se font sur les liens mais les auteurs ne discutent pas des contrats ni même d'une structure de composition des services.

3.4 Outillage et analyse des spécifications Kmelia

L'approche formelle adoptée pour Kmelia donne la possibilité d'analyser rigoureusement les modèles à composants construits. L'analyse se fait au niveau de chaque composant et au niveau des compositions. Dans la suite nous proposons un survol des propriétés attendues, de quelques analyses effectuées sur des composants ou assemblages Kmelia et

des méthodes d'analyse que nous avons mises au point. Nous terminons par l'instrumentation de leur vérification dans COSTO.

3.4.1 Différentes propriétés à vérifier

Le premier niveau de propriétés des spécifications Kmelia, qui inclut le langage de données et d'expressions, correspond à la correction syntaxique, la résolution de noms, le respect des portées des variables (observabilité), la correction de type, ... Une petite partie de ces règles de bonne construction a été donnée dans la description formelle du modèle sous forme de prédicats (*cf.* Section 3.2).

Ensuite les concepts (service, composant, assemblage et composite) sont analysés selon différentes facettes telles que la cohérence et la correction, l'intégrité de la communication, l'absence d'interblocage, la vivacité, l'atteignabilité, la compatibilité des liens d'assemblages ou de promotion, le respect des contrats, ...

Les composants sont analysés par rapport à des propriétés ; on peut ainsi vérifier des propriétés inhérentes aux modèles, telles que la cohérence des interfaces de composants et de services ; la terminaison des services, la préservation des invariants par les services, le respect des contrats fonctionnels de services (pré/post-conditions) par les déroulements de services, ...

Les assemblages sont analysés essentiellement sur la composabilité des liens, qui, outre la concordance des sous-liens avec les interfaces de services, se matérialise sur une échelle à quatre niveaux [4] par la cohérence des signatures, la cohérence des interfaces de services, la cohérence des contrats de services et la compatibilité comportementale (en tenant compte des correspondances d'état et de messages). La cohérence des contrats implique de prouver les implications d'assertions. La compatibilité comportementale est évaluée lien par lien, et non globalement, on évite ainsi l'explosion combinatoire [AL03], elle comprend l'évaluation des synchronisations pour déterminer la terminaison des services. On peut aussi analyser la continuité des services (toutes les dépendances d'un service offert sont-elles transitivement résolues ou pas), ...

Les composites sont analysés en tant que composant et en tant qu'assemblage ainsi que sur les propriétés relatives à la promotion : règles de visibilité, respect des contrats à travers la promotion, ...

Les interactions sont un élément fondamental dans les propriétés liées à la composition. Nous les détaillons dans la section suivante.

3.4.2 Analyse de propriétés au niveau composant

L'objectif est de vérifier la correction individuelle des composants et services avant assemblage. Pour illustrer cette analyse nous traitons ici uniquement la propriété de cohérence des composants.

4. On propose ici une solution pratique au problème épineux de l'interopérabilité de composants issus de modèles autres que Kmelia.

Cohérence des composants et des services

Pour vérifier la validité des assertions (pré/post-conditions) du contrat de service, nous avons choisi une approche classique de cohérence de l'invariant. L'invariant d'état du composant est préservé lors du déroulement des services sous hypothèse des préconditions. Les services fournis et requis ne doivent contredire l'invariant de leur composant. La vérification de cette propriété de cohérence est faite à trois niveaux d'observabilité. Voici quelques règles définies pour assurer la cohérence des services. La notation est celle de la logique du premier ordre mais en forme textuelle ; dans la suite l'expression `old(x)` dénote l'ancienne valeur de la variable `x`, nous l'étendons aux variables dans une expression.

1. Obligation de preuve pour la préservation de l'invariant par les services offerts :
   — partie observable de l'invariant Inv^O (de la même façon Pre^O désigne la précondition portant sur la partie observable) sous la condition initiale $(old(Inv^O) \wedge Pre^O)$ alors $Post^O \Rightarrow Inv^O$;
   — partie non observable ($Post^{NO}$ dénote la partie non observable de la Post-condition) sous la condition initiale $(old(Inv) \wedge Pre)$ alors $Post^{NO} \Rightarrow Inv$.
2. Invariant virtuel ($vInv$) préservé par les services requis :
   — sous la condition initiale $(old(vInv) \wedge Pre^O)$ alors $Post \Rightarrow vInv$.

La vérification effective se fait dans un environnement adapté ; les expérimentations sont menées avec le langage B [Abr96] ; l'idée est de générer des obligations de preuve en B qui sont ensuite prouvées. Le lecteur pourra trouver une présentation complète de la méthode de vérification dans [And+09].

3.4.3 Analyse de propriétés au niveau assemblage

En supposant des composants munis des propriétés attendues, l'analyse vise à déterminer statiquement les propriétés des assemblages, *e.g.* la cohérence et la composabilité.

Cohérence d'un assemblage

La cohérence d'un assemblage est primordiale pour valider son bon fonctionnement. Lier des services ne présume en rien de leur cohérence. Une bonne conception de l'assemblage doit permettre de détecter des incohérences au-delà des simples correspondances de paramètres et types. En plus de la cohérence des interfaces des services reliées pour former l'assemblage, les assertions (pré/post-conditions) et les interactions sont utilisées pour mettre en place de véritables contrats entre les services dans les assemblages. L'ensemble forme le **contrat de service multi-niveaux** [MAA10b ; BJP10]. De façon synthétique, quatre niveaux de compatibilité sont pris en compte :

1. signature : le profil des services doit être compatible ;
2. dépendances : les interfaces de composants et les dépendances de services doivent être compatibles ;

3. contrat fonctionnel : le service offert "remplit le contrat" défini par le service requis (compatibilité d'assertions) ;
4. interactions : les communications entre services doivent concorder et ne pas aboutir à des blocages.

La notions de contrat multi-niveaux permet de garantir un degré de compatibilité (ou d'*interopérabilité*) dans une vision composants hétérogènes, *i.e.* produits par divers fournisseurs avec des langages variés. Par exemple, un service Corba est défini par une interface IDL qui autorise une compatibilité au niveau 1, celle des signatures.

Composabilité

Intutivement, des services des composants sont composables si on peut les lier pour faire un assemblage de telle sorte que les services interagissent. Nous avons formellement défini la notion de composabilité dans [AAA06d]. Le modèle Kmelia nous a servi de base de raisonnement pour l'analyse de la *composabilité* sur des compositions. La composabilité se ramène à la compatibilité statique et la compatibilité comportementale.

Compatibilité statique Du fait de la description des services, la compatibilité statique lors des assemblages de composants est vérifiée de façon progressive. Une première étape de la vérification de la compatibilité statique est faite à la compilation de la spécification Kmelia par analyse des interfaces en profondeur : vérification des signatures de tous les services en jeu, concordance des liens et sous-liens, complétude des services requis pour tout service offert dans la composition (seuls sont traités les services effectivement utilisés dans la composition).

Lorsque cette première étape de compatibilité est terminée, les assertions des services liés sont utilisées pour s'assurer du respect du contrat d'assemblage. Soit un service ServR (d'un composant C) avec les assertions $PreR$ et $PostR$; soit un service offert ServP avec les assertions $PreP$ et $PostP$. Le service servP permet de satisfaire le requis ServR lorsque :

$$PreR \Rightarrow PreP \wedge PostP \Rightarrow PostR$$

De cette façon, on décrit un contrat de telle sorte que les composants qui peuvent être assemblés avec C sont ceux qui sont à même de respecter le contrat et les services du composant C qui requièrent ServR savent les conditions dans lesquelles ils obtiennent des bons résultats. En prenant en compte l'espace d'état virtuel du service requis et l'espace des variables observables du service ServP, le précédent contrat d'assemblage s'étend comme suit :

$$(Pre_R^O)_{cm} \Rightarrow Pre_P^O \wedge Post_P^O \Rightarrow (Post_R^O)_{cm}$$

avec $(Pre_R^O)_{cm}$ la précondition de ServR restreinte aux variables observables du composant qui fournit ServP ; l'indiçage avec *cm* pour *context mapping* indique qu'il peut être nécessaire d'appliquer une correspondance entre les variables du contexte virtuel et celles du contexte réel.

Pre_P^O (resp. $Post_P^O$) désigne la précondition (resp. la post-condition) de ServP restreinte aux variables observables. $(Post_R^O)_{cm}$ est la postcondition de ServP restreinte aux variables observables. Cette exigence d'analyse de cohérence au niveau des services est aussi traitée au niveau de la promotion des services requis et offerts.

Des expérimentations ont été faites sur le cas CoCoME, nous avons détecté par le biais des analyses basées sur les assertions, des insuffisances dans les spécifications Kmelia initiales. Par exemple pour la cohérence des services, dans une des opérations (newProduct qui ajoute des références dans le catalogue de produit) du composant Inventory, la post-condition était insuffisante car ne mentionnait pas de modification de la variable d'état catalog quand l'ajout d'un nouveau produit au catalogue n'était pas possible. Dans ce cas la preuve de cohérence par rapport à l'invariant échoue ; nous avons pu corriger la spécification en précisant que le catalogue était inchangé.

Compatibilité comportementale Pour l'interopérabilité dynamique, on explore l'évolution dynamique des échanges. L'interaction entre des composants se traduit par l'interaction de leurs services. Ainsi nous nous basons sur une analyse de l'interaction pair à pair de services (appelant et appelé sur une liaison). L'interopérabilité dynamique est alors assurée lorsqu'il n'y a pas de blocage dans les interactions entre les services considérés, interprétés comme des processus. Sur la base de cette similarité, nous avons formalisé la correspondance entre les services des composants et les processus puis nous avons développé deux outils de traduction de spécifications Kmelia dans des langages adaptés et outillés MEC [AAA06c] et LOTOS/CADP [AAA06d]. L'analyse de la spécification est alors faite avec ces outils. C'est la technique de *model checking* qui est la plus utilisée et la plus pratique dans ce cas.

Les expérimentations sur le cas CoCoME ont plus servi pour les analyses de cohérence des services et des assemblages que sur la partie interaction et compatibilité comportementale qui était mise au point avec d'autres études de cas [AAA06c ; AAA06d]. Le résultat des analyses est la détection de blocage dans les comportements des automates.

3.4.4 La plateforme COSTO

Les analyses de propriétés des spécifications Kmelia sont mises en œuvre dans la plateforme d'expérimentation ouverte COSTO [5], sous forme d'un ensemble de *plugins* Eclipse (figure 3.7). Dans l'architecture de COSTO Version 1, chaque outil est un *plugin*.

— un éditeur pour faciliter l'écriture de spécifications Kmelia (complétion, coloration syntaxique, marquage des erreurs, ...) ;

— un analyseur statique qui intègre des vérifications classiques (typage) mais également spécifiques à Kmelia comme la cohérence des interfaces, l'observabilité, la formation correcte des systèmes de transition, ... ;

5. COmponent Study TOolkit [AAA07c]

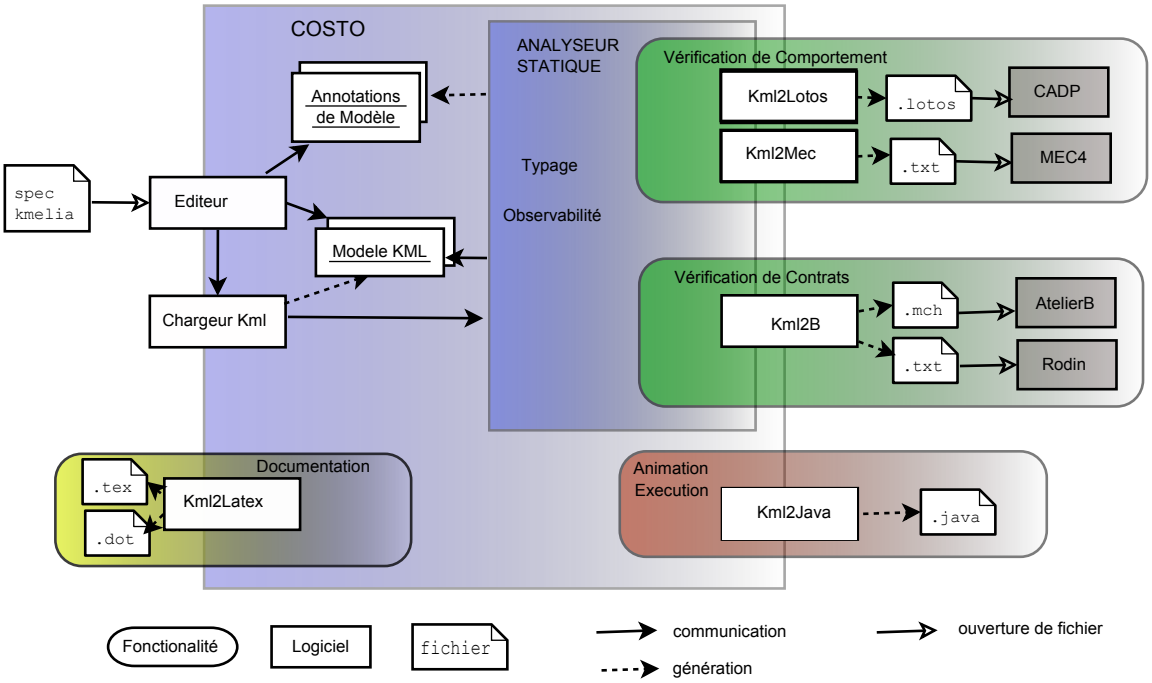


FIGURE 3.7 – Aperçu de l'architecture de COSTO Version 1

— des passerelles vers MEC et LOTOS/CADP ; le but est la vérification des interactions entre les systèmes de transition qui modélisent les services. Nous nous sommes appuyés sur des environnements existants afin d'exploiter au mieux les résultats connus en termes d'analyse de la dynamique des systèmes ;
— des passerelles vers Atelier-B et EventB/Rodin pour les preuves d'assertions et de contrats ;
— de divers utilitaires, pour présenter les spécifications sous forme graphique, pour générer des documents Latex des spécifications ;
— et d'un générateur de code Java donnant une sémantique opérationnelle restreinte et permettant l'animation de spécifications. L'animation peut se faire sous forme visuelle et interactive avec une fenêtre (*thread*) pour chaque service.

La plateforme a ensuite été enrichie d'une partie **tests** pour une meilleur couverture des contrats. En effet, les tests sont utiles car vérifier que chaque service respecte ses assertions (pre/post consitions) compte-tenu de ses interactions avec d'autres services est difficile à prouver formellement. Ce volet est détaillée dans la section 3.5.

3.5 Construction assistée de test de composants

Cette section fait référence aux travaux du projet *CostoTest*, mené avec Gilles Ardourel et Jean-Marie Mottu de 2012 à 2019 [AAM13 ; AMA13 ; AMS16 ; AAM17 ; Mot+19]. Divers projets de recherche étudiant (TER) ont permis une partie du développement et des expérimentations, nous remercions ces étudiants.

Le besoin de tester le modèle vient des limites des techniques de vérification formelle. En particulier démontrer que le déroulement d'un service est conforme à son contrat

fonctionnel (assertions de type pre/post conditions) n'est pas faisable avec les techniques mentionnées dans la section 3.4. En effet, compte-tenu de l'expressivité de Kmelia pour les services (eLTS, calculs symboliques, appels de services et communications), l'évaluation des comportements dynamiques des services par des outils de *model checking* ou de *theorem proving* se heurte aux limitations des langages associés (Mec, Lotos, Event-B...) par exemple l'explosion du nombre d'états ou les limitations du calcul symbolique. Nous donnons une sémantique opérationnelle par génération de code Java, qui permet d'exécuter Kmelia (*cf.* Section 3.4.4) et par là même nous donne un support pour le test.

> Il s'agit de test de modèles (*model testing- MT*) et non de test basé sur les modèles (*model-based testing - MBT*). C'est une différence importante que curieusement nous avons eu du mal à expliquer dans les conférences sur le test logiciel - voir par exemple le titre de [Mot+19].

L'originalité du travail est l'assistance proposée au testeur pour construire des modèles de tests, appelés harnais de test. Ces modèles étant écrits en Kmelia, on bénéficie des outils de modélisation et de vérification de COSTO.

3.5.1 Tester des modèles à composants et services

Dans l'ingénierie des modèles, la correction des modèles est essentielle. Tester au plus tôt permet de réduire le coût du processus de vérification et de validation [STW03]. Dans une vision pragmatique, l'idée est de tester la correction directement sur les modèles [GBR05]. Ainsi l'effort se concentre sur la détection en amont des erreurs "métiers" (*platform independent*), coûteuses à corriger lorsqu'elles sont repérées tardivement. Le test de modèles permet de s'affranchir des erreurs spécifiques à l'implantation et réduit la complexité globale du test [Bor+04]. Il permet une remontée des erreurs de test (*feedback*) en phase avec le langage de modélisation. Définir les tests au niveau modèle facilite leur adaptation en cas d'évolution du modèle. L'abstraction réduit la complexité du travail d'écriture des tests mais leur exécution sur des modèles reste problématique.

Pendant le test d'intégration de composants logiciels, l'encapsulation doit être préservée. Ainsi trouver où et comment fournir les données de test est difficile : pour atteindre une variable, il faut trouver les services qui la manipulent soit dans l'interface offerte du composant, soit dans des services qui en dépendent. Etablir le *contexte de chaque test* (*fixture*) d'un service impliquera une séquence d'invocation de services en plus de l'initialisation du composant. Dans [Gro04], Gross mentionne trois défis pour le test de composants : (i) tester des composants dans un nouveau contexte [6], (ii) tester sans accéder au fonctionnement interne d'un composant, et (iii) déterminer le niveau d'acceptation des tests (adéquation). Ce dernier correspond au problème C1 dans la classification des problématiques de Ghosh et al. [GM99], qui comprend aussi la génération de données de

6. Ce peut être un nouveau développement par le fournisseur ou un déploiement par les utilisateurs.

test (C2), la sélection de sous-ensembles de composants à tester (C3) et la création de séquences de test de composants (C4). Dans notre travail, nous traitons les deux premiers défis de Gross et les problèmes C3 et C4 de Ghosh, qu'il faut résoudre avant d'envisager les autres défis et problèmes.

Considérons la construction de *harnais de test* unitaire ou d'intégration pour des systèmes à composants et services, qui permettent de passer des données de test, d'exécuter les tests, et de récupérer le verdict (succès ou échec du test). La construction part d'un modèle du système, d'une *intention de test* et d'informations devant être extraites des composants sans briser l'encapsulation. Sachant que le testeur découvre l'application à tester et que les intentions de test restent informelles, il a besoin d'itérer jusqu'à ce que le niveau de précision soit atteint pour que le système soit testable. Le testeur a besoin d'assistance pour définir la forme véritable du test (où injecter les données de test, comment définir concrètement l'oracle) et déterminer la partie du système nécessaire au test.

Dans l'exemple de la Figure 3.8, le composant c:Client requiert un service offert du composant s:Serveur, via un lien d'assemblage. Cet assemblage de composants est représenté selon la notation de SCA [MR09], ce qui montre que les concepts (syntaxe abstraite) vont au delà des langages (syntaxe concrète). Nous avons étendu la notation SCA (par-

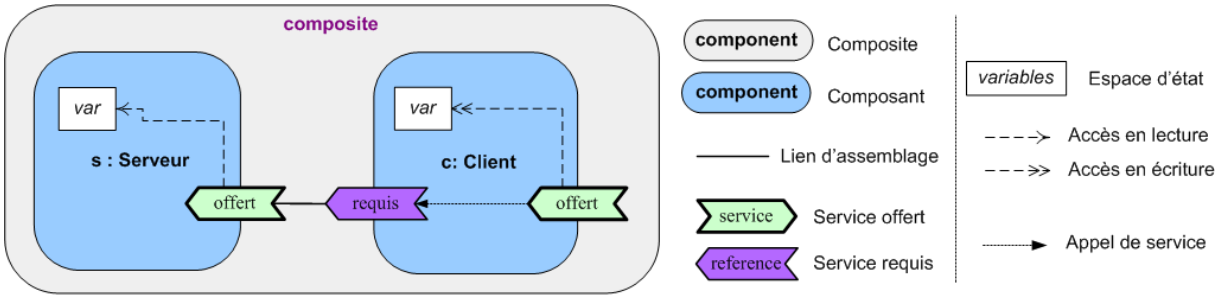


FIGURE 3.8 – Notation du modèle à composants et services

tie droite de la légende dans la Figure 3.8) pour mettre en évidence les dépendances de services et de données qui jouent un rôle prépondérant dans le contexte de test : (i) les composants ont un état formé de variables typées, (ii) les services ont un accès en lecture/écriture à ces variables, et (iii) les services offerts dépendent de services requis pour remplir leur tâche.

Illustrons ce modèle par l'exemple de `platoon` de véhicules de la Figure 3.9 [7]. Les véhicules, liés par ondes, adaptent leur état (vitesse et position) en fonction de leur prédécesseur, tout en préservant une distance de sécurité (propriété de sûreté de fonctionnement). Le pilote pilot est responsable du but. Trois (composants) véhicules sont enchaînés dans l'assemblage. Chaque composant dispose d'un service de configuration pour initialiser son état. Le service run place les véhicules dans un mode d'autorégulation qui requiert des services du véhicule précédent pour obtenir la position et la vitesse.

7. L'exemple est disponible sur `http://www.lina.sciences.univ-nantes.fr/aelos/projects/kmelia/`

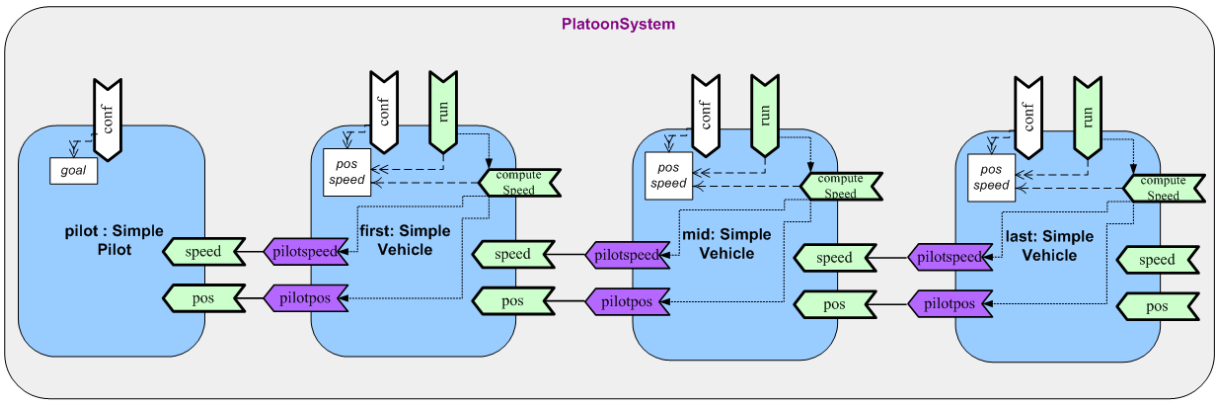


FIGURE 3.9 – Modèle à composants du système *Platoon* de véhicules

3.5.2 Test de conformité du modèle

Le *test de conformité* vise à montrer que les services des composants du système opèrent le comportement spécifié dans leur interface. Une partie du système, représenté par un modèle abstrait (noté PIM), est concernée par un test selon un objectif de test. Ici le terme d'*intention de test* (IT) soutend l'objectif du test (en langage naturel), la cible (les services à tester) et les conditions de test (les données à fournir, l'état supposé des composants, les résultats attendus dans un oracle). Nous appelons *système sous test* (SUT - *system under test*), la partie du système concernée par le test.

Pour être testé, le SUT nécessite un environnement de test permettant de configurer les composants, passer des données de test, exécuter chaque test, en récupérer les sorties, contrôler leur conformité avec un oracle, produire leur verdict. Cet environnement est appelé un *harnais de test* (HT). Sa construction est une étape nécessaire avant de pouvoir considérer les techniques de création de données de test et d'oracle (par *Model-Based Testing* - *MBT* par exemple, ce qu'on ne traite pas ici). Des classes `JUnit` sont un exemple de harnais de test pour des programmes sous test écrits en Java.

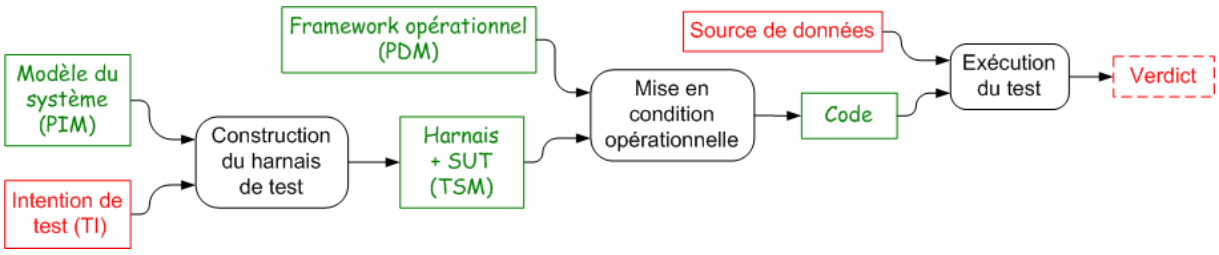


FIGURE 3.10 – Processus général de test

Notre proposition consiste à modéliser le harnais au même niveau d'abstraction que le modèle à composant du système. Le processus que nous proposons (Figure 3.10) considère en entrée le modèle à composants du système que nous étudions au niveau modèle (*Platform Independant Model* - *PIM*), ainsi que l'intention de test. La première activité consiste à produire le harnais de test comme un *modèle spécifique au test* (*Test Specific Model* - *TSM*) qui va permettre de tester le SUT (un ou plusieurs composants sous test du

système considéré). Ce niveau d'abstraction des tests implique de transformer le harnais vers une plateforme spécifique permettant l'exécution des tests et l'obtention de verdict. La deuxième activité du processus génère donc le code opérationnel des tests à partir d'une description du framework opérationnel (*Platform Description Model - PDM*).

Ce processus de test comprend deux étapes principales : la construction du harnais et sa mise en condition opérationnelle. Ces deux étapes sont décrites dans la suite de cette section. La dernière étape est l'exécution du harnais de test pour déterminer le verdict. La conception de l'oracle et des jeux de données ont une influence à ce niveau opérationnel. Si le modèle dispose de contrats fonctionnels *e.g.* pre/post conditions avec un invariant, on s'appuira sur ces informations pour inférer en partie le prédicat de l'oracle.

3.5.3 Construction du harnais de test

Comme le souligne Ghosh [GM99], il est nécessaire de pouvoir considérer différents contextes pour tester un composant à cause de l'influence des autres composants dans l'assemblage (*race conditions*). Le harnais de test est alors conçu (ou construit) en isolant les composants et services à tester, et en remplaçant tout ou partie de leurs clients (resp. de leurs serveurs) par un *driver* de test (resp. des *mocks*), puis en fournissant les sources de données de test.

Prenons en exemple le test de conformité d'une propriété de sûreté (l'écart entre deux véhicules est suffisant) du service computeSpeed(safeDistance:Integer):Integer du composant mid. Le comportement du service dépend de la distance au précédent, des *positions* et *vitesses* de lui-même et de son prédécesseur. Le *driver* de test (vtd) se charge de placer le service dans son contexte (le composant de droite sur la Figure 3.11). Ensuite le testeur fixe les besoins du composant mid pour tester le service computeSpeed, en l'occurrence des serveurs pour les services pilotspeed et pilotpos requis par computeSpeed. Deux solutions s'offrent à lui : soit il place des composants de test spécifiques (*mocks*), soit il conserve un peu du contexte applicatif (le composant first). Cette variabilité permet différents types de test dans la même approche et répond à la problématique C3 de Ghosh (cf page 98). Dans la Figure 3.11 un *mock* spécifique a été conçu car vitesse et position sont liées.

Les activités impliquées dans cette partie sont :

— Déterminer le sous-ensemble du PIM concerné. Pour s'assurer de la cohérence et de la pertinence du sous-ensemble choisi, on s'appuie sur les dépendances de services.

— Rechercher et affecter des composants et services de substitution *mocks* pour obtenir une architecture TSM satisfaisant les dépendances des services utilisés.

— Concevoir le driver de test en concrétisant l'intention de test de façon à mettre le service en condition de test : définir la configuration des composants testés, définir la séquence d'appels de service qui place ces composants dans l'état attendu, définir les paramètres à fournir aux services et préciser les données à passer aux *mocks*.

Dans la pratique du test, ces activités, souvent lourdes, relèvent de l'ingénierie de test et sont développées par les testeurs. Par exemple, si computeSpeed prenait ses données

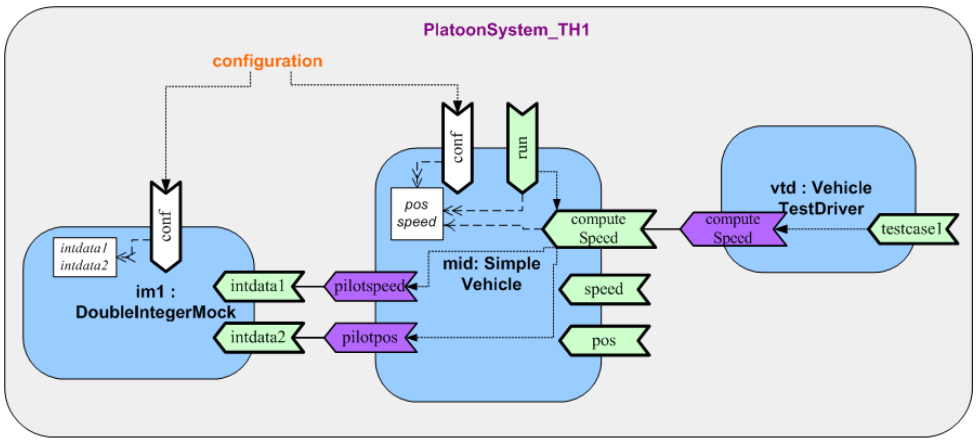


FIGURE 3.11 – Harnais de test du service computeSpeed du composant mid

sur ses paramètres d'entrée et de sortie, il s'agirait d'un (simple) appel fonctionnel et fournir les données de test serait trivial. Mais ce n'est pas le cas dans notre exemple puisque la distance du prédécesseur, sa vitesse et sa position dépendent de l'état du (composant) véhicule qui précède. De plus la simple configuration des composants ne permet pas ici de fixer ces valeurs. Les données sont fournies grâce à des appels de services : le paramètre safe distance est fourni dans la séquence d'appel du service testcase1 du pilote de test, la position et la vitesse (variables pos et speed de mid) sont fixées par le service de configuration conf de mid, la vitesse et la position du précédent sont fournis par des fonctions abstraites invoquées dans les services initdata1) et (initdata2). Trouver les services à invoquer pour mettre les composants dans un état acceptable pour le test n'est donc pas trivial. Notre objectif est d'assister le testeur durant ces activités, comme expliqué dans la section 3.5.4.

3.5.4 Mise en condition opérationnelle du harnais

Dans une approche IDM, mettre en condition opérationnelle consiste à produire un modèle adapté à la plateforme cible, notée PDM dans la figure 3.10. Les premières spécifications sont souvent trop abstraites ou incomplètes pour être exécutables. Chaque service impliqué dans le test doit être exécutable dans un environnement cohérent : ses dépendances doivent être satisfaites par des services compatibles et toutes les opérations utilisées dans l'environnement doivent être assez concrètes ou liées à une opération concrète. Dans notre vision, il s'agit donc de connecter les deux niveaux comme l'illustre la Figure 3.12 :

— Vérifier que le modèle est exécutable (décrit dans la section 3.5.4).
— Déterminer les sources de données concrètes et leur représentation.
— Etablir les *liaisons formelles de données* (*LFD*) c'est-à-dire les points d'entrée du TSM pour les données concrètes.

Cette mise en condition opérationnelle présente un coût supplémentaire à cette étape amont de conception. Cependant, les erreurs détectées aussi tôt sont moins coûteuses à corriger que celles de l'implantation des composants et services. En particulier, ces derniers

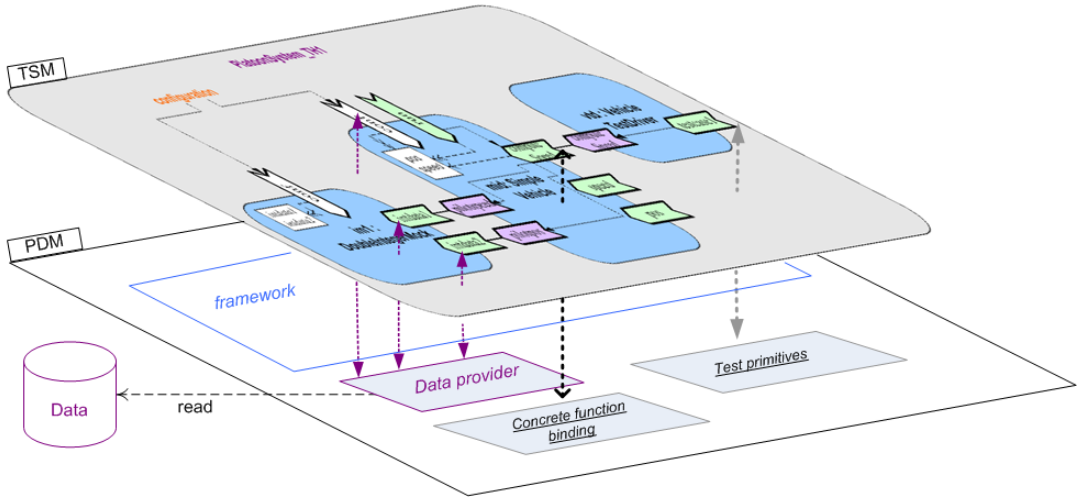


FIGURE 3.12 – Correspondance entre les données concrètes et les fonctions abstraites

peuvent être implantés sur plusieurs plateformes, multipliant les tests alors qu'une partie de leur comportement est déjà vérifiable pendant la modélisation, comme proposé ici.

La seconde proposition de ce travail est un processus général d'aide à la construction d'un harnais de test qui produise un modèle testable et exécutable à partir d'une intention de test plus ou moins détaillée. Ce processus, schématisé dans la Figure 3.13, couvre la construction du harnais et la partie *exécutabilité* de la mise en condition opérationnelle de la Figure 3.10. A partir d'un modèle du système (PIM) et d'une intention de test

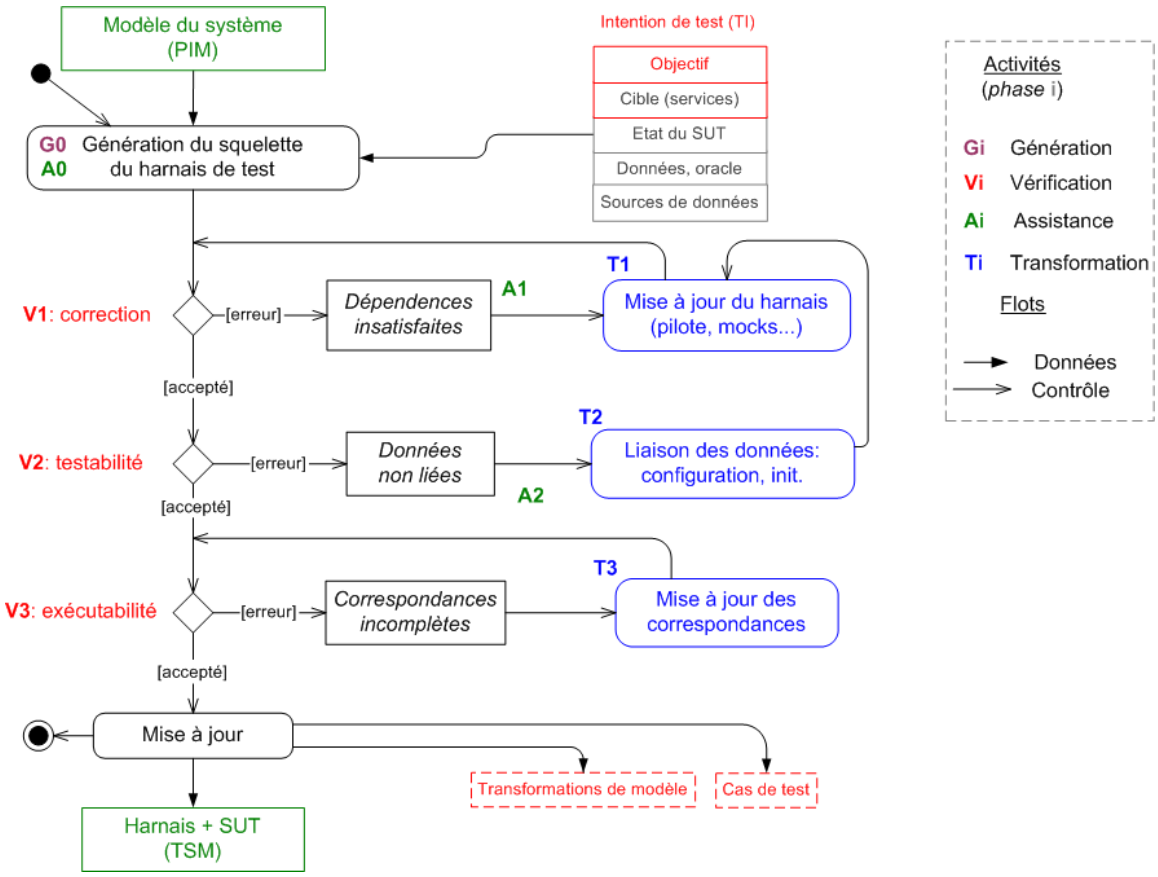


FIGURE 3.13 – Construction du harnais de test

(TI) on construit une application de test exécutable (TSM) et des informations complémentaires (cas de test, transformations). L'intention de test (introduite en section 3.5.2) est au départ relativement incomplète, elle dépend finalement de la connaissance qu'a le testeur de l'application. Elle est conçue comme une structure à contenu variable qui par raffinement donnera des cas de test comprenant les sources de données de test (*data binding*), la séquence de test du driver de test, l'expression contextualisée de l'oracle via des services observateurs (*blame assignment*).

A0-G0 Le squelette du harnais est généré par sélection de composants du PIM en fonction de l'intention de test. Les *DFS* sont extraites des services à tester (la source), si le testeur a fourni la cible.

V1 **Correction** Les propriétés de correction mentionnées dans [MAA10a], la cohérence et la complétude des dépendances de services vis-à-vis de la source sont vérifiées.

A1 A partir des signatures des services manquants (et plus si l'interface des services requis est riche) les services candidats sont inférés (soit dans la bibliothèque de composants de tests soit dans le PIM).

T1 On peut générer un pilote à partir des *DFS* et l'expression de verdict à partir de l'oracle, générer des liens d'assemblages suggérés durant **A1** en ajoutant des substituts proposés (services et composants) ou générés par défaut à partir de sources de données, remplacer des composants et services par des substituts pour réduire le périmètre de test, modifier l'appel du service de configuration des composants, y compris ceux ajoutés dans cette itération, ou attendre la détection d'erreurs.

V2 **Testabilité** Toutes les variables des *DFS* doivent être associées de manière cohérente au harnais. Les paramètres des services apparaissent naturellement dans la *séquence de test*. Les autres variables peuvent se retrouver dans la configuration, la signature des services des substituts ou dans la séquence de test.

A2 Aide à la recherche de données. Pour chaque variable non liée d'un composant, on établit la liste des services de configuration qui peuvent modifier cette variable. Pour chaque donnée externe non liée, une liste des services de substitution est proposée.

T2 Les *LFD* du service sont établies par (i) la configuration des composants en sélectionnant un service de configuration et en lui affectant une source de données, (ii) l'initialisation du test en sélectionnant un service de modification et en lui affectant une source de données, (iii) les liens d'assemblage des services requis responsables de la donnée requise et notamment en le reliant à un substitut.

V3 **Exécutabilité** Les données des composants de test, les substituts, ou la configuration des composants peuvent être spécifiés dans le modèle sous forme de fonctions abstraites de données (signature + assertion). Pour être exécutable, le harnais doit établir des *LFD* de ces fonctions.

T3 Pour chaque correspondance incomplète, on affecte une fonction concrète. Les primitives de test (assert) sont associées à des fonctions concrètes définies dans une librairie qui contient les fonctions prédéfinies pour les types de base.

L'intérêt de ce processus est méthodologique et il définit un cadre pour la production d'outils d'assistance. Certaines étapes sont automatisables, d'autres restent à la charge du testeur, y compris les problématiques non traitées ici, telle que la génération de données de test par exemple.

La contruction du harnais est décrite dans [AMA13]. Des éléments de mise en oeuvre et une expérimentation sur le cas `platoon` sont proposés dans [AAM13]. L'outil `CostoTest` est présenté dans [AMS16]. Enfin, dans [Mot+19], nous démontrons l'intérêt du test de modèles par rapport au test du code, même basé sur des modèles.

3.6 Rétro-ingénierie d'architectures

Raisonner sur les architectures est intéressant en approche descendante pour construire des applications de qualité mais tout aussi intéressant en approche ascendante pour améliorer la qualité des systèmes logiciels existants. Cette section fait référence aux travaux du projet Econet [8] financé par Egide, que j'ai initié et coordonné de 2006 à 2008 [Anq+09].

Le projet se situe dans l'aproche *Component Based Software Engineering (CBSE)* et cible en particulier le mythe 4 de la page 61, celui des composants fiables fabriqués par des tiers (*Components Off The Shelf - COTS*) à intégrer et donc réutiliser dans ses propres logiciels. Un composant est un module logiciel évolutif (unité plus grande qu'un objet) qui peut être utilisé à haut niveau d'abstraction (architecture logicielle, conception) et à bas niveau (programmes, *frameworks*). Comme nous l'avons vu dans la section 3.3.6, divers modèles sont proposés autour des composants et/ou services pour raisonner et établir des propriétés et aider à structurer le code. Par contre, raisonner sur le code existant pour en déterminer les propriétés n'est pas courant. C'est l'approche choisie dans ce projet.

L'approche CBSE reste un défi. La plupart des approches académiques privilégient les modèles abstraits, parfois proches des modèles architecturaux avec les langages de description d'architectures (ADL) avec des propriétés vérifiables telles que la sécurité et vivacité ; certains d'entre eux traitent du raffinement et de la génération de code. A l'inverse, les propositions industrielles telles que CORBA, EJB, OSGI ou .NET sont orientées implantation (à objets). Elles définissent des composants à plat (sans structures hiérarchiques) et le modèle est basé sur une infrastructure sous-jacente pour les bibliothèques de composants et la gestion des communications. Le manque d'abstraction rend difficile la réutilisation des composants. De plus, les mises en œuvre de composant ne respectent pas les normes industrielles et parfois il n'y a pas composants du tout. Contrairement à d'autres paradigmes comme l'objet, il n'y a pas de langage de programmation à composants (un langage tel que ComponentJ est une couche sur Java). En d'autres termes, il existe un écart entre les spécifications des composants (les modèles académiques) et les implémentations de composants (infrastructure industrielle ou implémentations orientées objet). La conformité entre la mise en œuvre et la spécification des composants est

8. `https://velo.wiki.ls2n.fr/doku.php?id=projects:econet:start`

rarement prouvée. Un problème majeur est alors de combler cette lacune.

— Une première solution consiste à définir des techniques de transformation de modèle afin de générer un code à partir des spécifications des composants. Cette méthode d'ingénierie *forward engineering* est similaire aux approches MDE mais reste complexe puisqu'il faut, en théorie, prouver la correction de la transformation et la conservation des propriétés *correct by construction*. Mais surtout, il existe différents frameworks et langages cibles (lire le chapitre 4 à ce sujet).

— Une autre façon consiste à se concentrer sur l'analyse du code du programme afin de comparez le code réel du composant avec sa description de haut niveau (abstraite). Cette méthode de rétro-ingénierie *reverse engineering* reste toujours une question assez ouverte dans les recherches CBSE [BHM06 ; PP07]. Ce problème est encore plus complexe que dans la première solution, pour les raisons suivantes :
  — Souvent, le code source d'un composant n'est pas disponible après son déploiement ou même non physiquement disponible dans un appel de service distant ou un service Web. Cependant, pour une industrie de composants, l'indisponibilité du code source est contractuelle : les services peuvent même être proposés sur une base de paiement à l'utilisation.
  — Dans une implémentations à objets, l'absence de structures "composant" implique de trouver des critères adéquats pour structurer les composants.
  — De nombreuses instructions et envois de messages doivent être omis pour une identification de service pertinente.
  — Il n'existe pas de modèle de composant commun (ou de norme) pour la spécification (abstraite) du composant – de nombreuses cibles pour la rétro-ingénierie.

  Les clients du service doivent interagir correctement avec les services et doivent connaître au moins l'interface mais dans la plupart des cas le comportement dynamique ou le protocole attaché aux services. À partir de là, des vérifications de compatibilité et des contrôles de cohérence peuvent être effectués pour assurer une bonne interaction ou pour éviter une utilisation erronée ou illégale des services.

Le but du projet Econet était de contribuer à l'approche rétro-ingénierie en développant des techniques d'extraction d'abstractions du code (y compris la description d'interfaces de composants) et de vérification de modèles représentatifs du code, *e.g.* pour vérifier en ligne un service bancaire sans code disponible ou vérifier qu'un composant client est compatible avec un composant implémenté. Noter que le travail est similaire pour les approches à services. Nous considérons les composants comme des conteneurs de services.

Le cœur du projet est d'établir un lien entre les codes des composants et spécifications des composants. Les avantages de l'abstraction sont de vérifier la conformité des codes des composants et de leurs spécifications, pour vérifier statiquement diverses propriétés telles que la sûreté (*safety*) et la vivacité (*liveness*). Nous avons délimité la zone de contribution sur l'expertise des partenaires du projet.

1. Le modèle source (niveau implémentation) est limité au code Java. Pour obtenir

une spécification abstraite d'un composant à partir de son code, on peut partir d'un code de base (*plain Java*) mais s'il contient des annotations ou commentaires alors la détection en sera facilitée. De même il faut délimiter l'espace de recherche et ne pas prendre en compte des bibliothèques de base du langage.

> Si un code est issu d'une approche systématique à partir de modèles et que des informations de construction ou de traçabilités sont notées dans le code sous forme d'annotations ou de commentaires, alors la détection de *patterns* est plus simple.

2. Les modèles cibles sont des modèles de composants abstraits inspirés de ceux des partenaires. Nous étudions non seulement les caractéristiques structurelles du système, mais aussi sur l'abstraction *comportementale* à partir du code Java. Le comportement est lié aux caractéristiques dynamiques et fonctionnelles (*cf.* FIGURE 1.5) des composants et services [PV02 ; AAA06d ; Pav+05]. En particulier, le comportement dynamique décrit l'évolution dynamique des composants, des connecteurs ou services (interactions). Les mécanismes utilisés pour spécifier des composants sont fondé sur différents formalismes : conception par contrat (implémentée par des assertions), spécifications algébriques, machines à états, expressions régulières, etc.

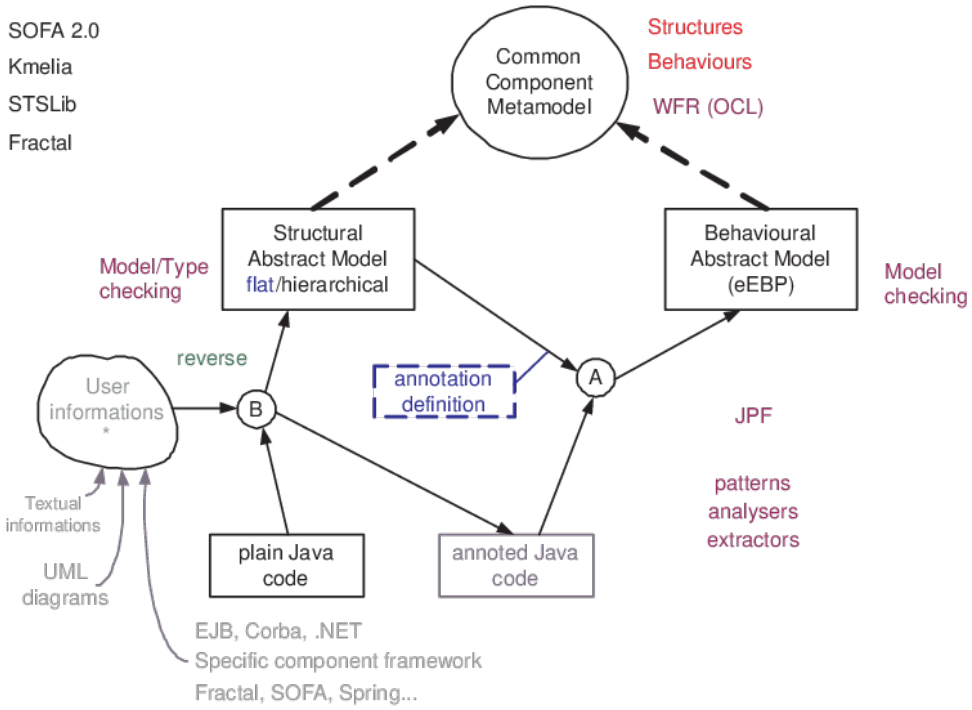


FIGURE 3.14 – Econet Architecture : final version

La FIGURE 3.14 montre l'architecture du projet. Le projet est organisé en trois parties, détaillées comme suit.

1. La partie métamodèle est partagée par les deux processus et constitue l'API de base (Application Programming Interface) pour le traitement du modèle de composant. Le métamodèle commun est une abstraction de plusieurs modèles de composants

pour obtenir un processus de réingénierie générique. Il doit gérer des connexions étroites avec le code qui implémente les applications des composants. Ces points de connexion sont représentés par des annotations dans le code Java.

2. L'objectif du processus B est de construire un modèle de composant structurel et un code Java annoté correspondant. Ces deux éléments sont des entrées du processus A. Le modèle est également une instance du métamodèle qui va contrôler sa cohérence. À partir du code Java brut et de l'interaction de l'utilisateur, le processus B doit produire un code Java annoté et un modèle de composant correspondant. Les deux résultats doivent être cohérents.
3. Le processus A extrait une spécification de comportement dynamique des composants identifiés lors du processus A à partir du code Java annoté. L'idée est donc de rendre l'ingénierie inverse la plus générale possible afin de permettre l'extraction de comportements dans n'importe quel formalisme. Pour être plus précis, les formalismes considérés sont : *enhanced behavior protocols/SOFA* [BHP06], *eLTS/Kmelia* et *STS* [Pav+05].

Les résultats du projet sont une identification des problèmes théoriques et pratiques, une architecture de la rétro-ingénierie de composants, un *plugin* Eclipse javacompExt pour le processus B et des techniques et expérimentations pour le processus A. les expérimentations ont été menées sur le cas CoCoME [9], un *contest* des modèles à composants proposé à Dagstuhl [Rau+08].

L'outil javacompExt [Anq+09] s'inscrit dans le cadre de la rétro-ingénierie de code source à objets - ici en Java (*cf.* Section 1.2.7 du Chapitre 1). L'érosion de l'architecture logicielle est un problème général dans les logiciels existants. Pour lutter contre cette tendance, les modèles et langages de composants sont conçus pour tenter de rendre explicites et automatiquement exécutoires les décisions architecturales en termes de composants, d'interfaces et de canaux de communication autorisés entre les interfaces des composants. Pour assister les architectes logiciels à abstraire leurs applications, nous explorons la possibilité d'extraire des éléments architecturaux (composants, communications, services, ...) du code source. L'outil javacompExt a été conçu sur des heuristiques pour extraire des informations sur les composants d'un code source Java. Il a été implanté en Java par Jean-Claude Royer. La capture d'écran de la FIGURE 3.15 est un exemple résultat présenté par notre outil. Les boites sombres sont des types de données, les boites claires sont des types de composants. Les flèches simples vont d'un type de composant composite à un type de sous-composant. Les flèches en pointillé illustrent une communication (c'est-à-dire un ou plusieurs appels de méthode), de l'appelant à l'appelé. La largeur de la flèche pointillée dépend du nombre de services appelés sur ce canal de communication. Une fenêtre contextuelle (en haut de la fenêtre principale sur le côté droit) sur les composants affiche les services qu'ils fournissent et nécessitent. Une fenêtre contextuelle sur la communication montre quels services sont impliqués dans cette communication.

9. `https://github.com/cocome-community-case-study`

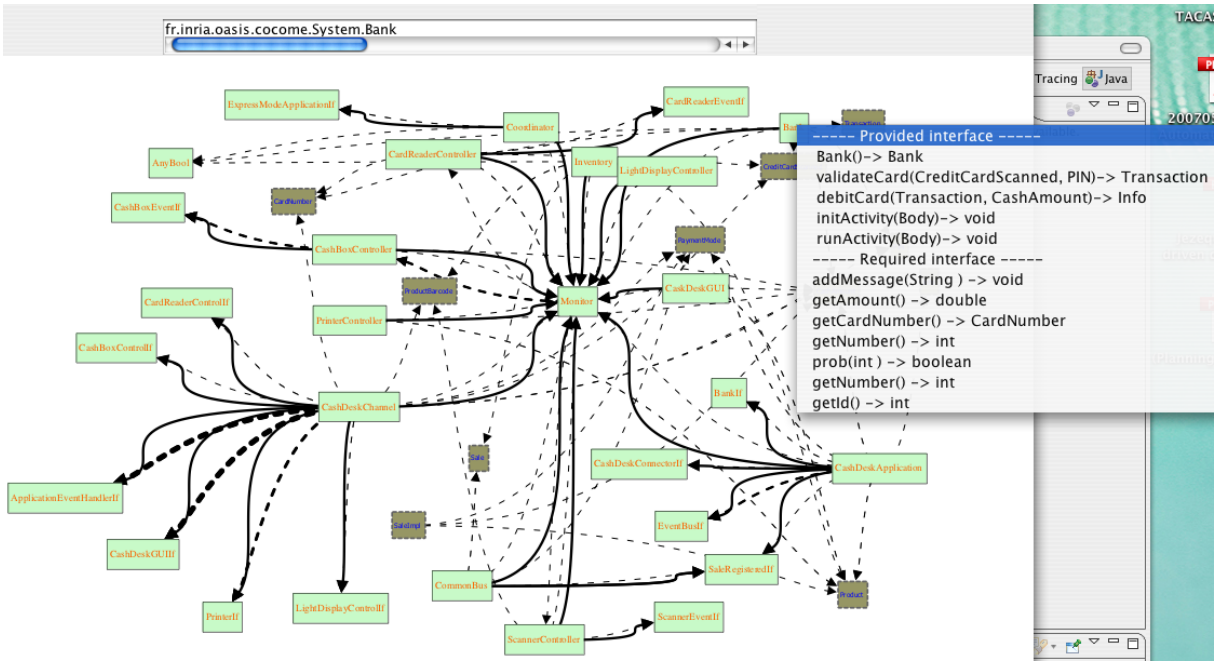


FIGURE 3.15 – Outil `javacompExt` appliqué à un sous-ensemble du cas CoCoME

Les rapports du projet sont disponibles en ligne [10].

3.7 Conclusion

Dans les travaux mentionnés dans ce chapitre, on s'attaque au Mythe 4 de la page 61 mentionnées dans le chapitre 2

Nous avons présenté les principes de la composition dans l'approche Kmelia pour la conception et le développement à base de composants. La présentation est faite sous la forme d'une synthèse des travaux et résultats autour de Kmelia qui est un modèle formel et extensible, fondé sur peu de concepts fondamentaux (service, composant, assemblage). C'est aussi un langage **hétérogène** à large spectre couvrant les aspects structurels, fonctionnels, et comportementaux et qui renforce la notion de requis facilitant une réutilisation individuelle des composants (*off-the-shelf*). Il se distingue ainsi de la plupart des modèles connexes référencés dans cet article.

Plusieurs modes de composition sont proposés pour la construction hiérarchique de composants et services. Notamment nous avons distingué la composition verticale pour une structuration en profondeur des services et composants, puis la composition horizontale pour une structuration en largeur. A partir de la composition nous avons montré les formes d'interactions admises dans Kmelia. Elles sont fondées sur des communications via des canaux abstraits à la manière de CSP de Hoare. Les aspects formels permettent de mettre en œuvre l'analyse automatisée des propriétés et la dernière partie de l'article est consacrée à l'outillage de Kmelia ; nous avons donné les principales caractéristiques de la plateforme COSTO qui accompagne Kmelia.

10. `https://velo.wiki.ls2n.fr/doku.php?id=projects:econet:start`

La vérification de propriétés est un volet essentiel qui couvre beaucoup d'explorations et de développements à faire autour de l'approche Kmelia. Parmi les pistes explorées, nous avons étudié la correction fonctionnelle des services en utilisant la méthode B, notamment afin de prouver le raffinement des assertions par les automates de services. Dans certaines approches comme Sofa ou Fractal la génération de code de l'architecture vient compléter le code métier. Kmelia permet de décrire le comportement d'un composant avec suffisamment de détail pour obtenir le code métier par raffinement ou par génération de code. Nous avons développé le raffinement de spécifications Kmelia vers des sémantiques opérationnelles par traduction dans d'autres formalismes pour la vérification des propriétés du comportement dynamique, ou par génération de code Java, qui permet l'instrumentation du code et les tests unitaires, d'intégration ou fonctionnels des modèles. La plateforme COSTO/Kmelia est intéressante du fait qu'elle couvre les activités de modélisation, de vérification de propriétés et d'exploitation des modèles. Concernant la vérification, l'originalité profonde est de couvrir les différents types de techniques : analyse statique, *model checking*, *theorem proving* et tests. Ce point a été développé dans [AAM17].

Post-scriptum

Ce travail mené au sein de l'équipe a été fédérateur. Il couvre bien les problématiques de base du génie logiciel, qui constituent pour moi un socle immuable depuis ma thèse : approches modulaires et approches formelles. Il est source d'inspiration ou de comparaison avec les travaux détaillés dans le chapitre 4 sur ingénierie logicielle dirigée par les modèles, le chapitre 5 pour la vérification de propriétés de sécurité, le chapitre 6 pour les modèles dans l'architecture d'entreprise, le chapitre 7 pour la la définition de systèmes manufacturiers reconfigurables...

Ce chapitre a mis en évidence des acquis sur les compétences suivantes, que nous reverrons dans le chapitre 8.

C2

Identifier les problèmes, les ordonner, imaginer des pistes ambitieuses, proposer des pistes réalistes, ordonnancer un ensemble d'activités de recherche pour structurer un travail de recherche scientifique sur du long terme.

C3

Organiser une recherche collaborative, détecter des compétences, définir des complémentarités et coordonner pour contribuer à un travail de recherche d'envergure.

C4

Animer des projets de recherche et fédérer les énergies en lien avec la compétence **C2** pour mener des travaux de recherche.

Chapitre 4

Développer du logiciel par les modèles

Rien ne se perd, rien ne se crée, tout se transforme.

Lavoisier

Le développement dirigé par les modèles (*Model Driven Development - MDD*) a acquis un rôle majeur dans le génie logiciel mais reste trop peu investi dans la pratique. En MDD, les modèles sont des abstractions d'implémentations et le développement est un processus de raffinement, si possible automatisé. En pratique, la distance sémantique entre les modèles logiques, indépendants d'une implantation, et les modèles d'implémentation empêche les raffinements automatiques et la génération de code. De plus, la conception logicielle est la phase du développement où la notion d'ingénierie prime : l'ingénieur logiciel doit choisir et décider quelle sera la solution la plus adaptée à la situation (en lien avec la qualité) comme le mentionne le principe 2.2.3. On notera que l'ingénierie est quelque part l'inverse de l'automatisation : moins le processus est maîtrisé, plus on a besoin d'inventer.

Ce chapitre reprend différents travaux menés sur des projets étudiants de Master, le travail est toujours en cours. Nous revisitons le schéma MDD et proposons une méthode basée à la fois sur des activités d'ingénierie et de rétro-ingénierie pour mettre en œuvre l'**ingénierie des modèles** (*cf.* section 1.2.6). L'automatisation partielle des processus de raffinement et d'abstraction passe par des transformations de modèle. L'ingénierie est structurée par des macro-transformations en couches paramétrées par les abstractions de plateformes obtenues par rétro-ingénierie. L'idée sous-jacente est que la complexité soit dans le processus de transformation et non les transformations individuelles, qui si elles sont petites peuvent être systématisées et automatisées. Une telle vision rend aussi le processus plus flexible et adaptable à chaque projet. Nous illustrons l'approche par des études de cas sur un petit système physique distribué. Ce travail vise à aider les praticiens à automatiser le MDD mais aussi à collaborer pour produire des solutions concrètes.

Remarque

La lecture de la section 1.2.6 du chapitre 1 est un prérequis à la lecture de ce chapitre, de même que la section C.2.5 de l'annexe C.
Ce chapitre reprend des éléments publiés [ALB19 ; AT20 ; AT21 ; AT24].

4.1 Introduction

Les développeurs de logiciels ont besoin de méthodes et d'outils pour concevoir et programmer des applications logicielles à partir des exigences et des spécifications d'analyse. Le développement basé sur modèle (*Model Based Development - MBD*) a émergé avec la conception structurée dans les années 1970 et a acquis un rôle de premier plan dans le génie logiciel avec l'avènement de l'approche dirigée par modèle (MDA) grâce à des normes interopérables (UML, MOF, XMI...) et des outils (modélisation, transformation de modèles...). Dans ce contexte, le MDD raccourcit le cycle de développement en se concentrant sur les abstractions et en automatisant partiellement la génération de code : décrire des modèles abstraits, les vérifier et les transformer pour faire fonctionner des applications. Selon Bran Selic [Sel08], le MDD se réduit à deux idées principales : élever le niveau d'abstraction et élever le degré d'automatisation informatique. Dans la recherche en génie logiciel, le MDD a perdu une certaine attention depuis 2010 et est devenu l'un des sujets du Model Driven Engineering (MDE) [Sel07], qui est désormais le principal domaine de contribution ; MDE comprend l'évolution du modèle, la rétro-ingénierie, l'ingénierie linguistique, etc [BCW17].

A ce jour, l'objectif n'est pas atteint malgré de nombreuses contributions techniques et d'outillage. Dans les pratiques actuelles d'ingénierie logicielle [1], même avec une méthode comme RUP [RJB99], les développeurs restent dans la vision MBD de la page 59. Les modèles servent à raisonner et communiquer en analyse et conception mais disparaissent ensuite. Il manque des propositions MDD génériques en tant que processus de transformations de modèles [DREP12], exception faite d'AUTOSAR dans le domaine automobile ou du CRUD dans les applications web de bases de données.

La question principale est *comment implémenter un processus de transformation générique à partir d'un modèle logique ?* Nous n'avons pas trouvé de réponse dans la littérature, même dans [PM07] qui introduit la compilation de modèles conceptuels. Nous pouvons l'expliquer par les observations suivantes : (i) la distance entre les modèles logiques et l'implémentation est très grande, elle inclut toutes les activités de conception de logiciels ; (ii) l'ajout d'éléments techniques (plateforme) au niveau de la conception est une activité d'ingénierie et donc difficilement automatisable ; (iii) les outils pour le processus de transformation manquent et ils se concentrent sur transformations à petite échelle. Depuis quelques années, les chercheurs se concentrent plutôt sur le support aux langages spécifiques (DSL). Nous avons besoin de techniques et outils pour gérer le décalage entre les modèles logiques et les modèles de conception afin de permettre l'automatisation. Cela correspond aux défis de l'*engineering practice* de [PMR16].

Pour automatiser l'approche MDD, et réduire la distance entre les modèles et le code nous proposons : (i) une approche systématique de la conception par un processus MDD structuré avec des macro-transformations (être systématique permet de définir les acti-

1. Sauf pour l'approche des méthodes formelles où les modèles abstraits sont affinés en modèles opérationnels avec beaucoup d'efforts sur la vérification du modèle.

vités afin de les automatiser), (ii) un processus descendant (raffinement) et ascendant (abstraction) pour aligner les modèles et les programmes par composition de modèle (la partie ascendante se focalise sur les aspects techniques), (iii) la définition de transformations systématiques au lieu de directives afin de les automatiser et (iv) des transformations de modèles pour aider le praticien grâce à des outils. Notre objectif à long terme est de fournir une assistance méthodologique aux développeurs MDD avec des processus, des techniques et des outils.

Dans ce chapitre, la section 4.2 remonte nos expériences sur la pratique du MDD pour dégager des principes d'automatisation du MDD dans la Section 4.3. Un processus MDD en co-évolution est ensuite introduit avec une approche descendante et ascendante. La section 4.4 se concentre sur les problèmes de mise en œuvre, d'outillage et d'expérimentation sur les transformations. Tout au long de ce chapitre nous ferons référence à une étude de cas, disponible en ligne [2].

4.2 Retours d'expériences

Selon la vision MDA, illustrée en page 59, le MDD est un *processus de transformation* d'un *Platform Independent Model (PIM)* à un *Platform Specific Model (PSM)* (plus) concret injectant des éléments de la *Platform Description Model (PDM)*. En amont on trouve le *Computation Independent Model (CIM)* qui reflète le modèle du domaine d'application, le modèle métier et les besoins.

Notre point de départ est PIM, le modèle logique produit lors de l'activité d'analyse. Il comprend des aspects structurels (*e.g.* classe UML), des aspects dynamiques (*e.g.* diagrammes d'état UML) et des aspects fonctionnels (*e.g.* activité UML, instructions OCL, sémantique des actions) du système modélisé. La cible est PSM, le système logiciel (code source, bibliothèques) qui implémente l'application à déployer sur les appareils. Comme le montre la Figure 1.1, la conception consiste à "tisser" le modèle logique sur l'infrastructure technique (*platform*) pour aboutir à un modèle exécutable.

Pour répondre à la question *"comment implémenter un processus de transformation générique à partir d'un modèle logique (PIM) ?"*, nous avons comparé trois approches avec différents degrés d'automatisation (ingénierie logicielle, génération de code, transformations pas à pas) sur une étude de cas[2] : un système domotique simple mis en œuvre avec des briques Lego EV3 pilotées par des applications Android [AT20]. Au-delà de sa simplicité, il s'agit d'un système distribué avec des communications réseaux plus complexes que de simples appels de méthodes.

4.2.1 Contexte

L'objectif est de mettre en place une chaîne de production de logiciels basée sur des modèles de systèmes d'automatisation distribués. Nous nous intéressons ici aux automates

2. `https://ev3.univ-nantes.fr/at-medi2021app/`

programmables disposant d'un environnement d'exécution "réel" prenant en compte les contraintes de fonctionnement, de sécurité et de performance [Rie13]. Certaines propriétés sont générales (sécurité, vivacité), d'autres sont liées à l'environnement ou au système lui-même (énergie, dangerosité, qualité de service...). Du point de vue logiciel, nous considérons au moins deux niveaux :

— le niveau modélisation, où les comportements individuels et collectifs sont décrits et les contraintes sont analysées sous forme d'image digitale (jumeau numérique). Nous pouvons utiliser des langages tels que UML [Gro11b], SysML [FMS08], Kmelia [Mot+19], AADL [FG12]...) et les outils de vérification ou de simulation associés, etc. Les modèles à ce niveau d'abstraction sont qualifiés de *logiques* dans le sens où les détails techniques ne sont pas donnés par l'analyse. Le modèle logique plongé dans un environnement technique sera appelé modèle de conception, comme l'illustre le processus Y de la FIGURE 1.1 issu de [RV11].

— le niveau opérationnel où les contrôles des appareils physiques sont implémentés. Ceci est réalisé grâce à des outils de communication basés sur des automates programmables (PLC), des robots, des capteurs et des actionneurs.

Des étapes intermédiaires peuvent être traitées pour atteindre le niveau de mise en œuvre dans l'esprit MDD [BCW17] et des lignes de produits logiciels (*Software Product Lines*) [Atk02]. En MDD, il est essentiel de s'assurer de la correction du modèle avant de démarrer le processus de transformations et de génération de code [Mot+19]. Cela réduit le coût élevé de la détection tardive des erreurs [GBR05]. Quel que soit le langage de modélisation, les modèles sont considérés comme suffisamment détaillés pour être rendus exécutables : les transformations de modèles deviennent pertinentes si les modèles contiennent suffisamment d'informations. Ce point est indispensable pour vérifier les modèles [AAM17].

Nous avons aussi besoin de rétro-ingénierie de modèles (*cf.* Section 1.2.7). Dans MDA, une étape d'ingénierie inverse peut être représentée par une transformation de modèle d'un PSM vers un PIM comme illustré par la Figure 4.1.

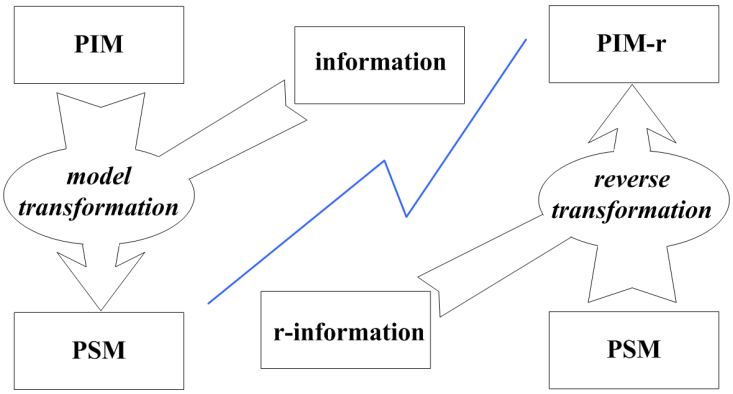


FIGURE 4.1 – Transformation inverse du modèle

Un processus de transformation est une composition de transformations de modèles, où chaque modèle intermédiaire est le PIM ou le PSM d'une autre transformation.

4.2.2 Etude de cas

Nous avons sélectionné un équipement domotique simplifié : une porte de garage comprenant les dispositifs matériels (télécommande, porte, automate, capteur, actionneurs...) et le logiciel qui pilote ces dispositifs. Le comportement du système est suffisamment simple pour être compris [3]. Le système fonctionne comme suit. Supposons la porte fermée. L'utilisateur commence à ouvrir la porte en appuyant sur le bouton `open` de sa télécommande. Il peut arrêter l'ouverture en appuyant à nouveau sur le bouton `open`, le moteur s'arrête. Sinon, la porte s'ouvre complètement et déclenche le capteur d'ouverture `so`, le moteur s'arrête. Appuyer sur le bouton `close` fermera la porte si elle est (partiellement ou complètement) ouverte. La fermeture peut être interrompue en appuyant à nouveau sur le bouton `close`, le moteur s'arrête. Sinon, la porte se ferme complètement et déclenche un capteur de fermeture `sc`, le moteur s'arrête. À tout moment, si quelqu'un déclenche un bouton d'arrêt d'urgence situé au mur, la porte se verrouille. Pour reprendre (dans le même état) l'utilisateur tourne une clé privée dans une serrure sur le mur.

Nous fournissons un modèle logique du cas d'étude en UML *e.g.* le diagramme de classes de FIGURE 4.2 ou le diagramme d'état de FIGURE 4.3 qui décrit le comportement du contrôleur de porte. Les actions sur les portes sont transmises aux moteurs par la porte

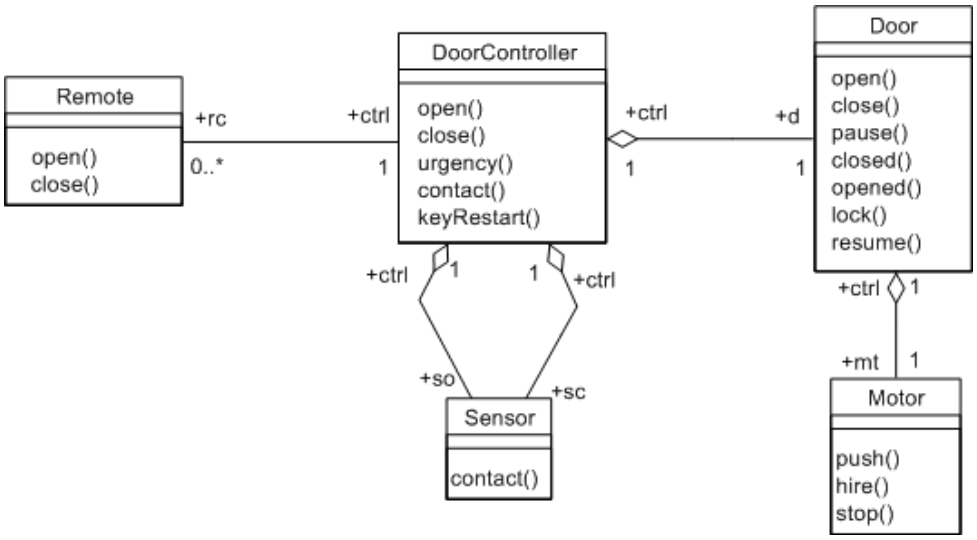


FIGURE 4.2 – Diagramme de classes - porte de garage

elle-même. La télécommande, lorsqu'elle est activée, réagit à deux événements (appui sur le bouton d'ouverture ou appui sur le bouton de fermeture) et informe ensuite le contrôleur qui interprète l'événement.

La vérification des modèles logiques peut inclure l'analyse statique, le contrôle de type, le *model checking* pour les communications, la preuve de théorèmes pour les assertions de contrats fonctionnels et les tests [AAM17]. Ces techniques nécessitent souvent la transformation des modèles vers des notations formelles. Nous supposons que les propriétés du modèle sont vérifiées avant l'implantation.

3. Voir l'énoncé de différents cas sur `https://ev3.univ-nantes.fr/projets-ev3/`.

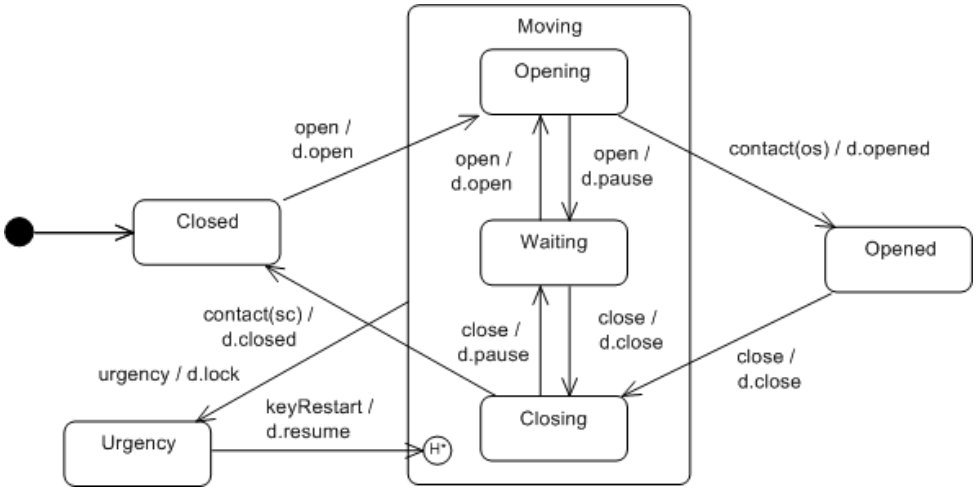


FIGURE 4.3 – Diagramme états-transitions du contrôleur de porte

4.2.3 Passage des modèles à leur implémentation

La conception logicielle est l'activité qui implémente les exigences fonctionnelles et non-fonctionnelles au sein d'une plateforme technique (*cf.* FIGURE 1.1). Il s'agit d'une activité d'ingénierie où les décisions prises affectent la qualité du résultat (*cf.* les principes de la page 52). Elle couvre les aspects complémentaires tels que la persistance, la concurrence, les interfaces humaines, le déploiement. Le résultat devra évoluer au gré des évolutions techniques ou fonctionnelles. Nous supposons dans la suite une cible technique donné. Nos expérimentations sont menées avec le contrôleur Lego EV3 (java/Lejos) et Android [4]. Le diagramme de déploiement de la FIGURE 4.4 illustre l'architecture technique choisie.

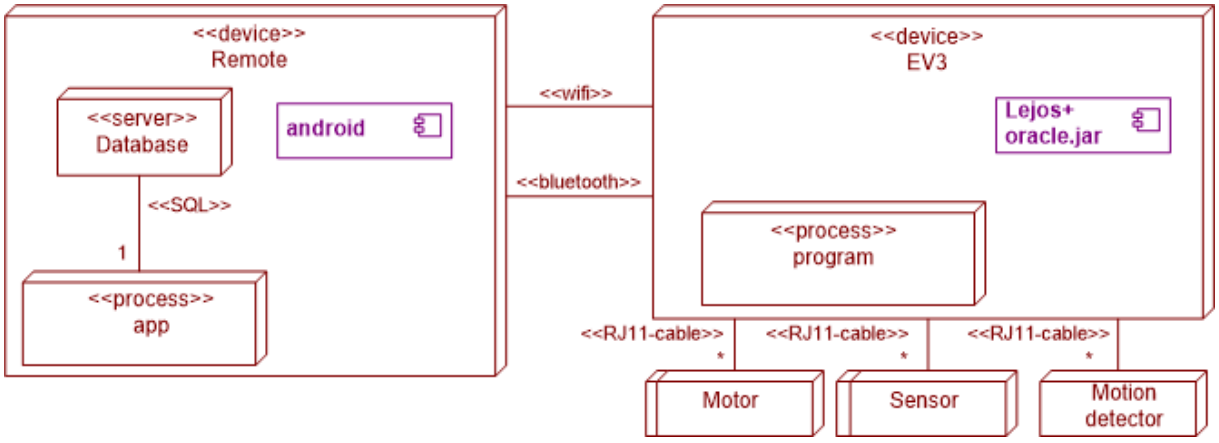


FIGURE 4.4 – Architecture technique - EV3 et Android

Il existe trois alternatives principales pour développer une application à partir d'un modèle logique (conception, codage et tests). Nous les classons par degré d'automatisation : (i) le développement *manuel*, (ii) le raffinement par étapes selon un processus de *transformation* de modèles, (iii) la *génération* partielle ou totale de code.

4. Nous fournissons de la documentation à ce sujet sur le site `https://ev3.univ-nantes.fr/` monté dans le cadre d'un projet *Fonds de Développement de la Pédagogie (FDP)* subventionné par la Faculté des Sciences de Nantes en 2020.

Nous détaillons par la suite en A) les expérimentations du développement manuel, puis évaluerons en B) la génération de code et nous terminons par discuter en C) des transformations de modèles, qui sont une approche intermédiaire.

A) Développement manuel

Le développement manuel est l'approche standard dans laquelle les ingénieurs partent d'un modèle d'analyse et conçoivent des solutions logicielles. L'étude de cas a été remise à différents groupes d'étudiants de différents cursus. Le point de départ était le modèle logique décrit sommairement dans la Section 4.2.2, la documentation sur EV3 Lejos, des exemples de tutoriels et aussi des articles comme [Han11 ; NTW04 ; PD07 ; SK98].

> Le cas a été confié à différents groupes d'étudiants qui ont réalisé différentes implantations sur des itérations différentes. Nous ne présentons ici que deux versions.

Une première version[5] a été proposé par des étudiants de Master 1 Informatique. Le diagramme de classes de la Figure 4.6 correspond à l'application EV3 qui pilote le prototype physique de la Figure 4.5. Le portail est commandé à distance par une application Android en connexion Bluetooth. Une autre version[6] produite par un groupe d'étudiants de Master 1 Miage a conduit à l'implantation du diagramme de classes de Figure 4.7.

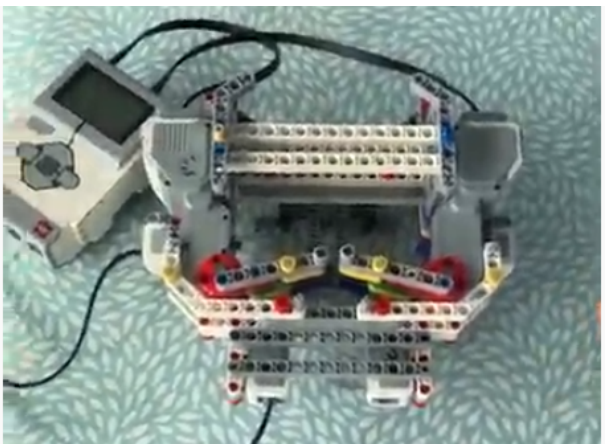


Figure 4.5 – Prototype du portail (v1)

Le code produit n'est pas nécessairement conforme au modèle logique UML initial, qui était perçu comme une référence documentaire et non un modèle contractuel. Les décisions de conception détaillée sont différentes d'un projet à l'autre. Pour la version `v1` de la Figure 4.6, les étudiants utilisé des types enum pour implémenter des machines à états tandis qu'un *state pattern* a été choisi dans la version `v2` de Figure 4.7. Le dispositif distant a également été implémenté de différentes manières en fonction de l'expérience et de la motivation des étudiants : de l'interface graphique Java Swing avec communication TCP-IP filaire avec EV3 ou de l'application Android avec connexion Bluetooth ou WiFi. Enfin, le prototype de Figure 4.5 utilise deux moteurs pour deux battants de porte alors qu'une seule porte et un seul moteur étaient spécifiés dans le modèle logique. En effet, le modèle sert à comprendre et à interpréter le cas et les étudiants ne visent pas une stricte conservation de la sémantique. Isoler les différents choix de conception est une première étape pour rationaliser le développement dans un processus de raffinement (*cf.* Section 4.3).

5. `https://github.com/demeph/TER-2017-2018`
6. `https://github.com/FrapperColin/2017-2018/tree/master/IngenierieLogicielleDomoDoor`

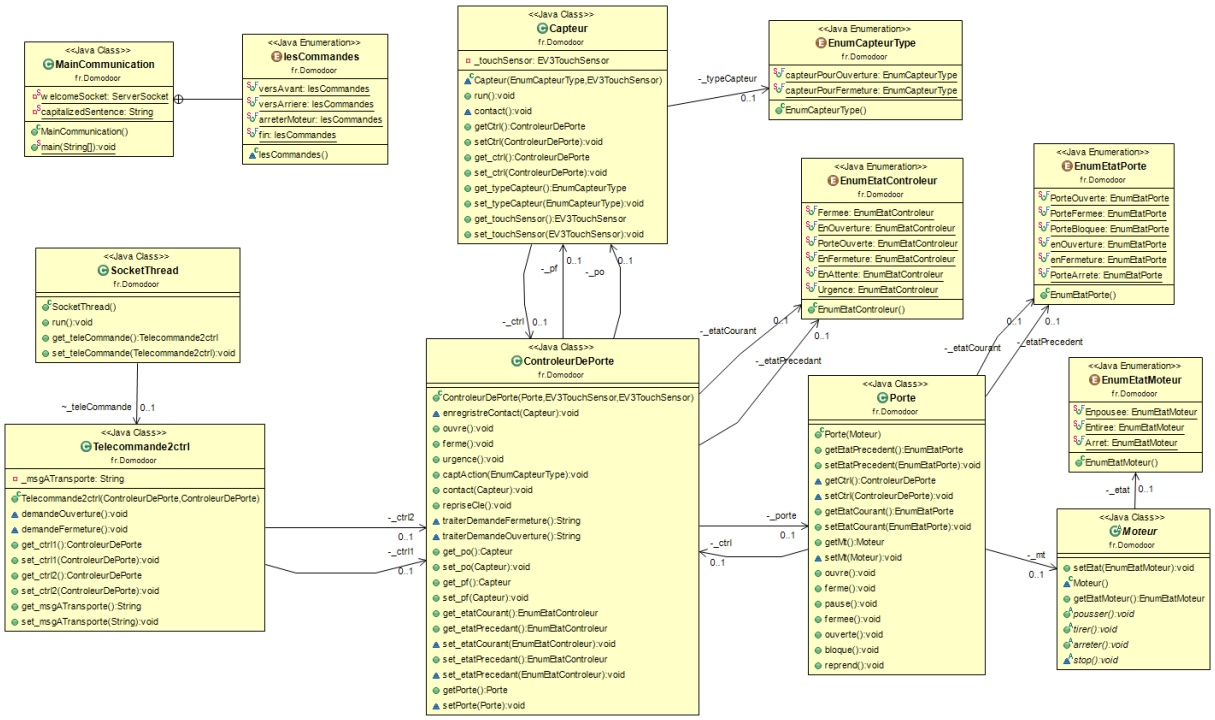


FIGURE 4.6 – Diagramme de classe de l'application (v1)

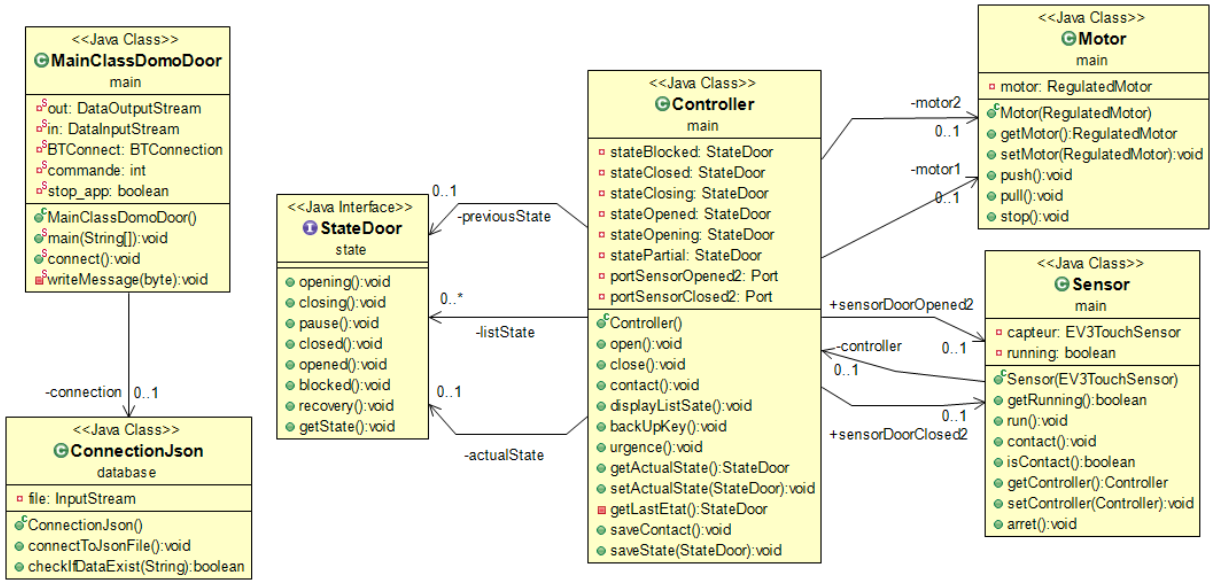


FIGURE 4.7 – Diagramme de classe de l'application portail (v2)

B) Génération de code, animation

Dans cette approche, le modèle est compilé dans un code source exécutable. Il existe deux cas d'usage principaux : soit générer une partie du code (par exemple des squelettes de classes) que le développeur complète ensuite soit exécuter le modèle pour le simuler ou l'animer. Dans le dernier cas, le modèle est qualifié d'opérationnel.

Génération de code depuis les éditeurs UML Nous avons comparé les fonctionnalités de génération de code des éditeurs UML en fonction des options de génération de

code. Notre panorama n'est certainement pas une étude exhaustive mais donne une idée des outils prototypes dans certaines grandes catégories (gratuits, écosystème Eclipse Modelling (EM), AGL payants...) et de montrer des tendances. L'interopérabilité des modèles est assurée par la norme XMI (profils XML pour UML, SysML ou MOF), ce standard existe en presque autant de versions que les langages de modélisation. Malheureusement, le standard d'échange de diagrammes est moins interopérable. Les outils permettent d'éditer des diagrammes de classes (CD) et des diagrammes de transition d'état [7] (STD) et de (re)générer du code source. Bien qu'un lien existe entre le corps des méthodes et les diagrammes d'activités, nous ne l'avons pas vu concrètement dans les outils. Généralement ils fournissent la signature de l'opération mais rarement plus sauf lorsqu'une ingénierie *round trip* est inclus qui permet d'insérer des fragments de code cible aux opérations du modèle afin de les conserver lors de la régénération du code après évolution du modèle.

— StarUML est un représentant des éditeurs UML gratuits. Chaque STD est associé à une classe dont il représente le comportement dynamique (protocole). Star UML peut exporter les diagrammes en XMI, il utilise la version UML 2.0 de l'organisme OMG (Object Management Group), ce n'est pas la dernière version UML qui est la 2.5.1. StarUML génère les parties structurelles génère les parties structurelles des DC (profils des opérations, attributs, relations entre classes) et donc pas le corps des opérations ni les STD.

— Papyrus et Modelio représentent l'écosystème *Eclipse Modeling* qui bénéficient du support EMF *open source* avec une communauté active, même si cela baisse au profit des DSL. Papyrus utilisait, pour ces expérimentations, la version UML2 5.0.0 d'Eclipse MDT (Model Development Tools) qui est conforme à la version UML 2.5 d'OMG. La génération de code de Papyrus inclut des comportements aux opérations avec un incrémental qui remplace la génération de code. De plus on peut ajouter de nouveaux générateurs en plus du C++ et du Java natifs. Mais nous n'avons pas trouvé de plugin adéquat pour les machines à états et les associations n'ont pas non plus été générées dans le code Java. Modelio inclut le *roundtrip* mais contrairement à Visual Paradigm, il fait la différence entre les méthodes gérées par Modelio et les autres. Une méthode gérée est automatiquement générée pour chaque version. Une méthode simple (non gérée) est sous la responsabilité du développeur. Dans cette catégorie d'outils, on peut aussi citer les outils d'Obeo, notamment UML Designer et Acceleo, ou le nouveau projet Polarys qui inclut Papyrus et Topcased.

— Yakindu est un outil dédié aux statecharts (STD) proposé par Itemis. Il prend en charge tous les éléments pour modéliser des diagrammes états transitions. L'inconvénient majeur est de ne pouvoir traiter qu'un STD à la fois (une seule classe), on ne traite donc pas des synchronisations entre automates, sauf à les simuler sous forme de sous-régions parallèles (état hiérarchique composite). Il est possible de créer deux sous-diagrammes parallèles dans un diagramme, ça permet de simuler le compor-

7. State-transition diagram

tement de deux diagrammes mais la génération créera qu'une seule classe pour les deux sous-diagrammes. La génération de code développe par contre toute la machinerie d'exécution de l'automate. On trouvera un exposé détaillé de l'implantation Java des STD dans le mémoire [Han11].

— Visual Paradigm et Rational représentent la catégorie des outils payants. Visual Paradigm est très riche en standards et fonctionnalités. Il prend en charge la génération de diagrammes de classes et de transitions d'état dans le code source Java mais également en C++ ou VB.net. Sa fonctionnalité *round-trip Engineering* synchronise le code et le modèle. Nous n'avons pas eu accès au code généré pour estimer l'effort de programmation nécessaire pour ajouter une communication entre les machines à états. Après modification d'une classe générée, nous pouvons mettre à jour le modèle UML des diagrammes de classes. Les modifications apportées sont rajoutées dans le diagrammes de classes. IBM Rational Rhapsody, inspiré des auteurs d'OMT, principale contribution à UML, apparaît comme l'outil le plus complet. Le code est mis à jour automatiquement dans une vue parallèle du modèle. On peut éditer le code directement, les diagrammes resteront synchronisés. Là encore nous n'avons pas eu accès au code généré pour estimer la partie programmation manuelle.

La plupart des outils ne sont pas liés à un seul langage, par exemple Modelio, Papyrus ou Visual Paradigm intègrent d'autres standards OMG comme SysML, BPMN. Par conséquent, certains outils peuvent avoir des notations ésotériques pour certains éléments du modèle. Plusieurs outils couvrent un périmètre plus large que la modélisation de systèmes, couvrant des parties de l'architecture d'entreprise.

La Table 4.1 résume quelques caractéristiques des générateurs des outils. La ligne MOM (Message Oriented Middleware) indique la présente d'une implantation distribuée des objets qui échangent par envoi de messages et signaux. Nous n'avons pas retenu ici la possibilité de traiter le temps (réel) comme par exemple avec le profil MARTE. La licence d'utilisation peut être de type OpenSource, Free, Commercial. Nous appelons *API Mapping* une fonctionnalité qui permet de rattacher des éléments de modèle à des éléments prédéfinis dans des bibliothèques, des *frameworks*... C'est différent du *round-trip Engineering* qui contient le corps des méthodes définies dans les classes.

Table 4.1 – Comparaison d'éditeurs de modèles à génération de code

	Star UML	Papyrus	Yakindu	Modelio	VisualParadim	IBM rational rhapsody
UML - XMI	2.0	2.5	-	2.4.1	2.0	2.4.1
DC	√	√	-	√	√	√
STD	-	-	one only	√[1]	√	√
Operations	-	incremental	-	RoundTrip	RoundTrip	√
Round-trip	-	override	-	√	√	√
MOM	-	-	-	-	-	-
API Mapping	-	-	-	-	-	-
Licence[d]	F, C	O	F, C	O	C	C

[1]Extension possible `http://www.sinelabore.com/doku.php?id=wiki:landing_pages:modelio`

La TABLE 4.1 illustre le fait que, à notre connaissance, aucun outil ne traite encore clairement du problème des middlewares de communication (hétérogènes) ou du *mapping* vers les fonctionnalités techniques (haut niveau pour les architectures, bas niveau pour les frameworks et bibliothèques) sauf intégration dans un contexte très restreint comme Java, .NET, REST... Visual Paradigm peut intégrer des modèles de déploiement dans le cloud. IBM Engineering Systems Design Rhapsody est plutôt dédié à la conception détaillée. Certains outils proposent des fonctionnalités de persistance (*e.g.* mapping de relations objets ou SQL) que nous n'avons pas retenues ici puisque nous nous concentrons sur l'automatisation. Notez que lors de nos expérimentations, nous avons utilisé Papyrus pour générer des diagrammes de classes en Java.

UML exécutable Générer du code depuis UML pour une architecture technique donnée ou même un *framework* donné reste encore réservé à des cas simples comme la génération d'applications CRUD (Create, Read, Update, Delete) sur des bases de données relationnelles simples. Plus précisément il faut que le *framework* technique soit générique et complet mais aussi que les modèles soient simples [AV13]. On peut aussi animer ou exécuter des spécifications, qui sont alors qualifiées d'opérationnelles.

Dans tous les cas, le pré-requis est d'avoir des modèles complets sur la structure du système (diagrammes de classes et composants), son comportement dynamique (diagrammes états-transitions) et son comportement fonctionnel (diagrammes d'activités). Les diagrammes ne suffisent pas à préciser la sémantique. On doit en plus ajouter des contraintes écrites en OCL [CG12] (langage déclaratif pour les assertions de type invariant et pre/post-conditions) ou bien du pseudo-code écrit dans un langage conforme à la sémantique des actions en UML. Le concept d'action est présent de manière abstraite dans les diagrammes d'activités ou états-transitions.

La sémantique des actions (*Action Semantics*) est définie par un méta-langage des calculs depuis la version UML 1.4. Il s'agit d'un véritable langage de programmation mais aucune syntaxe concrète n'était proposée en standard (et donc pas de compilateur !). Dès UML 1.4, des syntaxes concrètes étaient proposées dans les outils proposant une version exécutable d'UML, notamment pour le temps-réel :

- Le langage *Action Specification Language (ASL)* est implanté dans l'outil iUMLLite de Kennedy-Carter (Abstract Solutions) supportant xUML [Rai+04].
- Le langage *BridgePoint Action Language (AL)* (et les dérivés aussi SMALL, OAL, TALL) proposé par Balcer & Mellor et implanté dans l'outil xtUML de Mentor Graphics [MB02].
- Le langage *Kabira Action Semantics (Kabira AS)* proposé par Kabira Technologies (acquis par Tibco, TIBCO Business Studio).
- Le sous-ensemble d'actions de la norme de télécommunication SDL [BHS91], qui existe aussi sous la forme d'un profil UML.

On trouve aussi le langage *Platform Independent Action Language (PAL)* proposé par Pathfinder Solutions, ou SCRALL [CSS09] qui propose une représentation graphique. Une

syntaxe concrète est aussi proposée dans [PP08]. La sémantique des actions est maintenant intégrée à la notation et UML exécutable est un profil. Seulement quinze ans après il n'y a toujours pas de sémantique reconnue et de syntaxe concrète [8]. Ces efforts ont conduit à une sémantique pour un sous-ensemble de modèles UML exécutables, appelée `fUML` (*Semantics of a Foundational Subset for Executable UML*) [OMG18], muni cette fois d'une syntaxe concrète normalisée `Alf`. Une implantation de référence existe [9].

Nous n'avons pas expérimenté l'approche "exécutable" puisque les outils associés sont payants ou anciens. De toutes façons, notre objectif n'est pas d'exécuter ou de simuler des modèles UML mais de concevoir des applications.

C) Développement par transformations de modèles

La génération de code est en général limitée et ne permet pas réellement de "concevoir" l'application en intégrant les aspects orthogonaux comme la concurrence, la persistance ou les IHM. Les transformations de modèles permettent plus de souplesse et d'intervention dans le processus de développement.

> À notre connaissance, il n'y a aucune proposition du processus MDD en tant que processus de transformations de modèles.

Nous recensons ici un certain nombre de considérations à prendre en compte :

— Il est illusoire de vouloir générer automatiquement du code sans avoir des modèles UML riches et détaillés.

— La génération de code elle-même n'est pas envisageable comme unique étape de transformation, du fait de la distance sémantique entre le modèle logique et la cible technique, composées d'aspects orthogonaux mais corrélés, appelés *domaines* (*e.g.* persistance, IHM, contrôle, communications, entrées/sorties) sur lesquels le modèle initial doit être "tissé".

— La conception est par nature une activité d'ingénierie, liée à l'expérience des concepteurs *cf.* page 117. On ne peut industrialiser un processus que si tous les rouages sont connus avec précision.

— La pratique nous a montré que les transformations étaient efficaces lorsque les modèles étaient proches sémantiquement *e.g.* diagramme de classe et modèle relationnel pour la persistance.

— Dans un système complexe on trouve des règles contradictoires et des oublis. Travailler avec des transformations simples permet de les rendre plus cohérentes, complètes et vérifiables et donc réutilisables dans des compositions de transformations.

— Il y a différentes manière de spécifier les transformations. Selon l'étude [Kah+19], les outils peuvent être classés en trois catégories *Model-to-Model (M2M), Model-*

8. `http://modeling-languages.com/uml-action-language-omg-journey/`
9. `http://modeldriven.github.io/fUML-Reference-Implementation/`

to-Text (M2T) and Text-to-Model (T2M). L'approche M2M est la plus courante et se subdivise en quatre approches : relationelle/déclarative (*e.g.* ATL [10]), impérative/opérational (*e.g.* Kermeta [11]), à base de graphes (*e.g.* Groove [12]), ou hybride (*e.g.* la famille de standards QVT [Kur08]). L'approche M2T est incontournable pour la génération de code avec différentes techniques connues (visiteurs, *template*, hybride).

1) Transformation UML pour la simulation ou génération de code Dans [AV13], nous avons consacré un chapitre complet à la conception avec des transformations de modèles. Nous avons défini des **transformations systématiques** vers ou depuis les bases de données ou vers des programmes à objet (Java, Smalltalk). Une transformation systématique est spécifiée par un ensemble de règles de transformations.

La transformation isomorphe [13] des diagrammes structurels comme le diagramme de classe en programmation à objets ou en base de données ne pose de difficultés profondes [14] du fait de la proximité sémantique des modèles sources et cibles de la transformation. Par contre la transformation de modèles dynamiques (diagrammes états-transition ou diagrammes d'activités) et fonctionnels (décrits en OCL ou sémantique des actions) est bien plus ardue car soit le modèle cible n'existe pas vraiment (les automates ou les flots de données n'existent pas dans les langages de programmation) soit il dépend d'un *framework* technique (une archive jar java par exemple). Pour illustrer cette complexité, nous relatons, dans [ALB19], quelques expériences des étudiants avec ATL pour la transformation d'automates en programmation à objets. Ces expérimentations ne résolvent pas le problème.

2) Transformation UML pour la conception logicielle Cette opération est complexe du fait de son côté "ingénierie" : on invente, on construit, on assemble en tenant compte de l'environnement technique cible. Comme indiqué en début de cette section C), nous n'avons pas trouvé de travaux à ce sujet. C'est donc l'objet de notre proposition présentée dans la Section 4.3.

4.2.4 Bilan

Nous résumons ici les retours d'expérimentation donnés dans [AT20] :

1. En ingénierie, nous avons observé les groupes d'étudiants s'entraîner sur l'étude de cas. Le code de l'application a toujours été une interprétation libre de la spé-

10. `https://www.eclipse.org/atl/`
11. `http://diverse-project.github.io/k3/`
12. `https://groove.sourceforge.net/groove-index.html`
13. On entend ici que ce qui est généré existait déjà sous une forme éventuellement différente mais il n'y a pas d'invention de nouvelles choses, elles sont inférées. Par exemple, une classe est transformée en une relation de base de donnée relationnelle ou en classes d'un programme à objets.
14. Certains points peuvent être un peu compliqués comme le passage de l'héritage multiple à l'héritage simple.

cification PIM, ce qui n'était pas considéré comme une abstraction à affiner et à laquelle se conformer. Les préoccupations de conception (persistance, GUI, concurrence, qualité) n'ont pas été priorisées par les étudiants, bien qu'elles s'influencent mutuellement.

2. Les générateurs de code UML n'exploitent généralement pas toutes les informations du modèle (UML ou SysML, OCL, Action Semantics). Les outils commerciaux offrent des solutions plus puissantes mais pas accessibles et difficilement comparables. Les outils de type *Executable UML*, notamment ceux incluant `fUML` [OMG18] et `Alf` sont intéressants à des fins de simulation ou d'animation (spécifiques à la plate-forme) mais pas à des fins de conception MDD.
3. Dans l'approche des transformations pas à pas, les expérimentations portaient sur le raffinement des machines à états et des messages UML en Java par des transformations ATL. En pratique, les étudiants n'ont pas réussi à construire un processus rigoureux et automatisé en raison de la complexité des transformations individuelles et du processus de transformation.

De plus, les préoccupations croisées (*cross cutting concerns* ou aspects orthogonaux) ne sont pas faciles à tisser. Par exemple, le *middleware* de messages et le moteur des machines à états ont une influence mutuelle. Cela rend la génération de code pilotée par modèle [MJ13] des solutions très tubulaires par rapport aux transformations de modèle.

Dans [AT20] nous avons esquissé un processus général de transformation en quatre étapes qui résume les aspects à considérer pour affiner la conception logicielle. Ces macro-transformations utilisent des informations de configuration. Dans la suite du document, nous revisitons ce processus et fournissons des détails sur le processus de conception de transformation par étapes.

4.3 Un processus MDE revisité

Pour automatiser le processus MDD, nous posons un ensemble de principes issus de nos observations :

P1 La transformation de modèle peut déduire de nouveaux éléments mais ne peut pas les créer à partir de rien. La qualité des entrées influence la qualité des sorties. Une étape préliminaire consiste à vérifier la qualité et les propriétés du PIM telles que la cohérence et l'exhaustivité.

P2 La conception logicielle prend en compte les aspects transversaux, tels que la persistance, l'interface graphique, le contrôle de concurrence, les communications, les entrées/sorties, sur lequel le modèle logique doit être "tissé". La distance sémantique entre PIM et PSM nécessite un véritable processus de transformation et ne peut être atteinte seulement par génération de code.

P3 La conception logicielle, en tant qu'activité d'ingénierie, est liée à l'expérience des développeurs. Un processus ne peut être automatisé que si toutes les activités sont

connues avec précision. Nous avons besoin d'une définition systématique des activités de conception automatisables en tant que transformations.

P4 En pratique, les transformations de modèles ne sont efficaces lorsque les modèles source et cible sont sémantiquement proches *e.g.* diagramme de classes ↔ modèle relationnel pour la persistance. Les petites transformations sont plus faciles à vérifier, à composer et à réutiliser. La complexité (l'intelligence) n'est pas dans la transformation atomique mais dans le processus de transformation..

P5 Une transformation composite est un processus composé hiérarchiquement d'autres transformations (plus simples) [15]. Les transformations de modèles atomiques sont de préférence des compositions de modèles telles que le tissage ou le mapping [Cla11] pour favoriser la traçabilité (éléments & décisions).

P6 La modélisation et la rétro-ingénierie sont des activités d'abstraction, les transformations de conception sont des activités de raffinement [16]. Notez bien que la rétro-ingénierie ne s'applique pas ici aux applications (PSM) mais aux plateformes (PDM), ce qui est moins compliqué car on ne mélange pas le métier et l'applicatif. Un processus de conception équilibré mélange des abstractions vers le PDM et des raffinements du PIM au PSM.

Sur la base de ces principes, nous proposons un processus en co-évolution : raffinement (descendant) par transformations détaillé dans la Section 4.3.1 et abstraction (ascendante) de l'architecture technique détaillé dans la Section 4.3.2.

4.3.1 Un processus de conception graduel

La conception descendante est définie par 4 étapes que l'on qualifiera de macro-transformations comme illustré par la FIGURE 4.8. Le modèle du système est affiné par une séquence de PIM (le premier est le modèle logique) et le modèle technique représente une séquence de PDM. Chaque macro-transformation est une transformation composite qui répond soit à une étape de conception, soit à une étape de programmation.

T1 La transformation de déploiement structure le PIM en applications (sous-systèmes) et les affecte aux nœuds de déploiement du PDM. Les liens entre les nœuds sont annotés par des protocoles de communication. Les sous-systèmes sont des éléments structurels du PIM tels que des classe UML, des composants ou des packages. Le concepteur peut structurer l'architecture de l'application avec des composants et décrire leurs API. Si le PIM comprend des diagrammes de composants et de déploiement pour une conception préliminaire, T1 reprendra le *mapping* existant.

T2 La transformation MOM (Message Oriented Middleware) affine les communications d'objets. Pour chaque protocole, les envois de messages UML sont instrumentés dans les frameworks MOM donnés par le PDM. Par exemple, un envoi de message peut

15. Dans un processus séquentiel, le PSM d'une transformation devient le PIM de la transformation suivante.

16. Ce n'est pas du *roundtrip* mais on peut l'utiliser pour aligner le modèle avec le code.

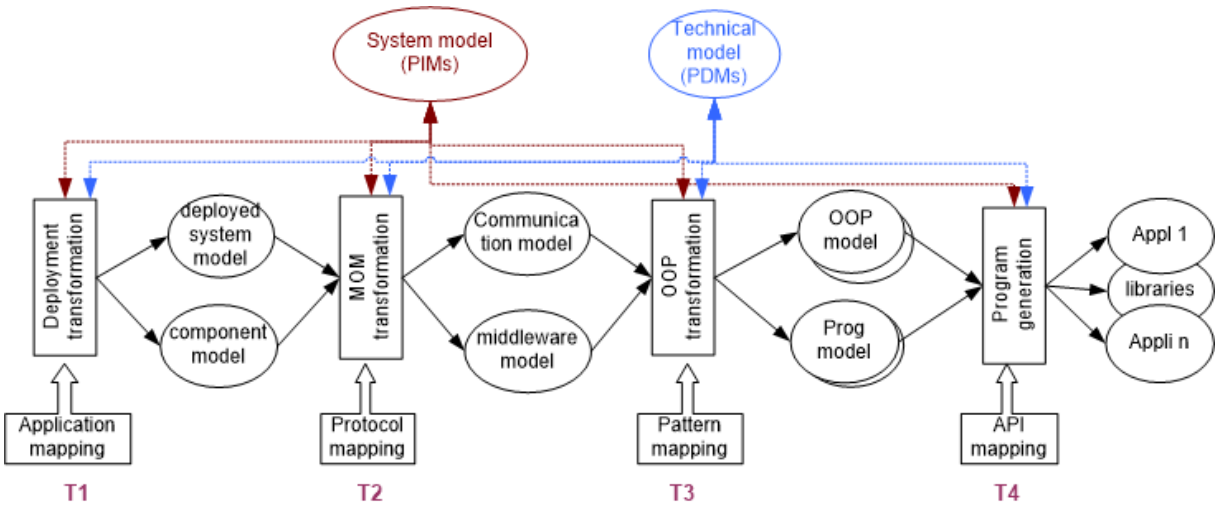


FIGURE 4.8 – Processus global de transformation

être un appel de méthode dans le langage cible (Java, C++ ou C#). Les appels de procédure à distance (RMI), les communications sans fil ou les communications TCP-IP nécessitent des modèles de transformation de haut niveau.

T3 La transformation Programmation Orientée Objet affine les concepts UML en modèles POO qui en général n'incluent pas nativement ces concepts. Ce problème épineux est détaillé dans le chapitre 7 de [AV13]. En particulier, T3 affine les diagrammes d'états-transitions (resp. diagrammes d'activités, héritage multiple, associations...) dans les concepts POO. La persistance est obtenue par une transformation spéciale vers les modèles DAO (Data Access Object) fournis dans le PDM.

T4 La transformation du programme prétraite la génération de code en générant du code à partir de modèles de bas niveau (*implementation blue print*) et en faisant correspondre les éléments du modèle aux bibliothèques prédéfinies des *frameworks* techniques. Rappelons que le cadre technique est la mise en œuvre du PDM.

Les aspects de conception tels que l'interface graphique ou la sécurité ne sont pas pris en compte ici, nous supposons qu'ils sont développés manuellement ; mais des modèles orthogonaux pourraient aider. Tous les paramètres de configuration des transformations et toutes les décisions doivent être stockés pour pouvoir rejouer le processus de transformation de manière itérative de cas d'évolution du modèle du système. Nous discuterons de l'implantation dans la section 4.4.

Une description détaillée et illustrée des quatre macro-transformations est proposée dans [AT24].

Il s'agit d'une approche générique dont les transformations dépendent à la fois du PIM et du PDM. Puisque le PDM est une entrée pour les transformations, nous avons besoin de vues différentes (abstraites) sur ce PDM. Nous discuterons de ce point dans la section 4.3.2.

4.3.2 Un processus d'abstraction

Dans la section 4.3.1, nous avons mentionné l'importance des informations sur l'aspect technique de la conception (les *frameworks*, bibliothèques, *middleware*...) dans les transformations de modèle. Cette information est présentée différemment selon le degré d'abstraction, le *Platform Description Model (PDM)* est le niveau architectural. L'idée est est proposer un modèle technique pour chaque macro-transformation comme le montre la Figure 4.9. L'approche MDD est alors un processus de transformations de modèles en co-évolution : abstraction (ascendante) du PDM (partie droite de Figure 4.9) et raffinement (descendant) par transformations (partie gauche de Figure 4.9). Les PSM (intermédiaires) sont liés aux modèles PDM par mapping de modèles.

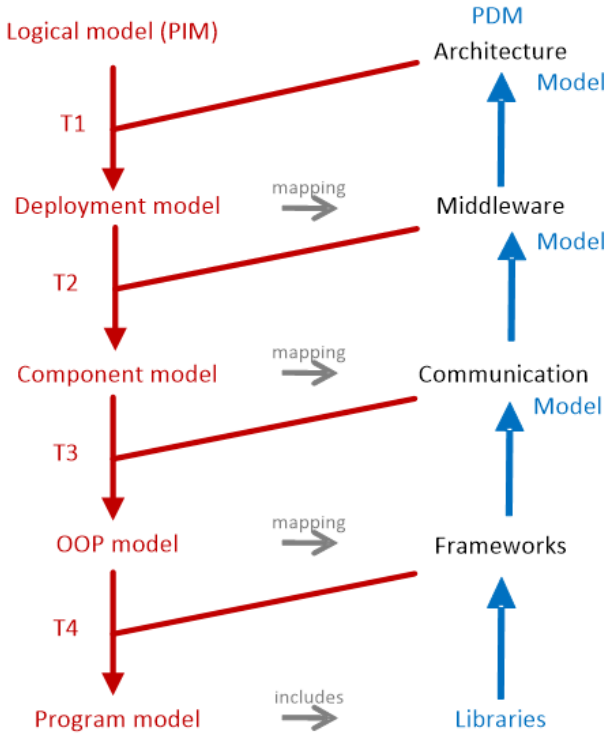


Figure 4.9 – Macro-transformation process

Les frameworks d'implémentation contiennent des archives de code mais pas de modèle (PDM). On peut combler ce manque en extrayant un PDM du code source du framework par Model Driven Reverse Engineering (MDRE), comme illustré par (partie droite de Figure 4.9). MDRE cible différents niveaux d'abstraction, de la représentation du programme aux architectures d'applications ou aux processus métier de haut niveau (*cf.* Section 1.2.7). MDRE est donc lui-même un processus de transformation [And19].

En MDE, écrire des transformations de modèles n'est pas trivial même si les méta-modèles source et cible sont généralement connus. Trouver l'abstraction dans MDRE est un problème encore plus difficile car il manque des directives et des informations de traçabilité [RFZ17 ; RS04]. Les informations sources diffèrent également et peuvent inclure du code binaire, du code source, des fichiers de configuration, des programmes de tests ou des modèles de scénarios... Une telle diversité rend l'activité RE difficile à appliquer. Heureusement, nous ne ciblons pas des applications complètes mais des frameworks techniques.

Dans notre cas (*cf.* Figure 4.9) les couches d'abstraction sont : le modèle de programme, le modèle POO, le middleware et le modèle de distribution. Par souci de simplicité, la Figure 4.9 montre uniquement quelques modèles (PIM-PDM) par niveau mais rappelez-vous que plus vous progressez dans le code, plus vous avez de domaines en parallèle. Dans chaque couche, les modèles sont écrits avec diverses notations de modélisation. Des standards de facto tels que UML, OCL, MOF, EMF, SysML, AADL, BPMN ou des modèles personnalisés définis avec des langages spécifiques à un domaine (DSL). Nous

utilisons les mêmes notations que dans le processus de conception pas à pas car ce sont les entrées de ce processus.

La prise en charge des outils de retro-ingénierie existe à bas niveau, mais monter en abstraction reste une tâche d'ingénierie [And19]. Les langages intermédiaires tels que Knowledge Discovery Metamodel (KDM) [Bru+14] sont utiles à bas niveau car il s'agit d'un modèle d'abstraction détaillée du code. Dans MDRE, plus on on abstrait, plus il est difficile de trouver des abstractions [Pep+16b]. Ici aussi, nous soutenons l'idée que rationaliser en concevant de petites transformations puis les composer hérarchiquement donne des résultats concrets. Nous illustrerons notre approche par un exemple dans la Section 4.5.

4.4 Implantation et expérimentation

Le processus de FIGURE 4.8 est abstrait et générique. Pour pouvoir automatiser des transformations de modèles, nous avons besoin (i) d'une définition systématique des transformations (section 4.4.1) et (ii) de l'outillage pour les implémenter (section 4.4.2).

4.4.1 Définition systématique des transformations

Nous définissons un sous-ensemble de transformations systématiques. Les détails sur les spécifications de transformation, l'implémentation d'ATL ou Papyrus et leur application à l'étude de cas sont détaillés dans l'annexe web[2]. Par exemple, T4 vise à unifier les éléments du modèle et la mise en œuvre (code source). Tous les éléments du modèle ne sont pas générés de toutes pièces, certains existent déjà, parfois sous une nature différente, dans le modèle technique (*cf.* FIGURE 4.9). Nous préconisons l'*API Mapping* pour associer des éléments de modèle à des éléments prédéfinis dans des bibliothèques ou des *frameworks*. Dans cette section, nous discutons du *mapping* des classes de conception (et des opérations) avec des classes de code source prédéfinies.

Pour simplifier, nous nous concentrons sur les classes comme éléments du modèle, mais cela devrait être étendu aux packages, aux types de données, aux opérations, etc.

Mapping d'API

Dans notre étude de cas, les capteurs et actionneurs existent déjà au niveau du code dans la bibliothèque Lejos. Par souci de simplicité, nous considérons qu'un élément de modèle correspond à une implémentation mais qu'une implémentation peut correspondre à plusieurs éléments de modèle (relation 1-N). Lorsque les éléments du modèle et de l'implémentation ne correspondent pas, les développeurs refactorisent généralement le modèle pour qu'il converge. Le processus de cartographie comprend trois activités :

1. *Détecter* les candidats à l'implémentation dans les bibliothèques avec si possible des taux de correspondance. Différents éléments du modèle sont pris en compte comme la classe, l'attribut, l'opération... Nous sommes ici confrontés à deux problématiques :

le niveau d'astraction et la correspondance de modèle. Fondamentalement, le modèle et les éléments de mise en œuvre ne sont pas comparables et nous avons besoin d'un modèle du cadre de mise en œuvre. Les éléments du modèle ne sont pas indépendants *e.g.* , les opérations sont dans des classes regroupées en packages, la manière d'agencer les éléments du modèle influence le processus de mise en correspondance.

2. *Sélectionner* l'implémentation adéquate des éléments du modèle (classe, attribut, opération) et lier les éléments du modèle aux éléments d'implémentation par une composition de modèle (non intrusive) [Cla11].
3. *S'adapter* à la situation. Une fois qu'un lien de correspondance est établi, cela implique souvent de revoir la conception. L'adaptation est le mécanisme de base pour lier les PIM aux PDM selon deux stratégies.
   — *Encapsuler et déléguer*. Les classes du modèle encapsulent les classes d'implémentation (modèle Adapter). L'avantage est de garder une traçabilité sur les API de base. L'inconvénient est la multiplication des classes à maintenir.
   — *Remplacer* les classes de modèle par les classes d'implémentation. La transformation doit remplacer les déclarations de type mais aussi les messages envoyés. Les avantages et les inconvénients sont inverse de ceux de l'encapsulation.

   Le remplacement est possible lorsque les classes ont une structure et un comportement similaires, y compris les types primitifs de modélisation. Dans tous les autres cas, le patron Adapter réalise les adaptations multi-fonctionnalités :
   — Attribut : nom, adaptation du type, valeur par défaut, visibilité...
   — Références (rôle) : nom, adaptation du type, valeur par défaut, visibilité...
   — Opération : nom, paramètres (ordre, défaut), adaptation du type...
   — Protocole : machine à états pour la classe du modèle mais pas pour la classe d'implémentation.
   — Composition : une classe est implantée par plusieurs classes du code.
   — Raffinement de la communication : les communications MOM sont distribuées dans différentes classes.
   — Couches d'API : classer les méthodes pour réduire la dépendance.
   — Principes de conception : améliorer la qualité selon les principes de conception SOLID, IOC, DRY, KISS... *cf.* principes 2.2.3.

Nous illustrons T4 par un exemple à la page 31 de l'annexe Web[2].

4.4.2 Outillage

Une implémentation rationnelle combine des outils de transformation de modèleset des fonctionnalités de génération de code. A bas niveau, la transformation du modèle serait de type *model mapping* ou *weaving*, alors qu'à plus haut niveau on trouve des liens de traçabilité ou de la restructuration. Lors des expérimentations, nous avons utilisé ATL [17] pour écrire des transformations car son style basé sur des règles prend en charge à la

17. `https://www.eclipse.org/atl/` `--/acceleo/` `--/papyrus/`

fois les transformations de modèle à modèle (M2M) et de modèle à texte (M2T), ce qui est pratique pour notre processus de macro-transformation (*cf.* FIGURE 4.8). Acceleo[17]est pratique pour réaliser les transformations de type M2T (model-to-text). La génération de code pour la transformation M2T a été implantée à l'aide de Papyrus[17]pour les transformations UML-OOP-SI vers Java et les transformations ATL personnalisées. Dans MDRE, nous avons utilisé Modisco et AgileJ comme illustré par un exemple à la page 36 de l'annexe Web[2]. *Knowledge Discovery Metamodel (KDM)* [Bru+14] est un standard pour la représentation de systèmes logiciels à bas niveau car c'est une représentation modèle qui conserve des informations très détaillées. Discutons maintenant de quelques problèmes ouverts de mise en œuvre et de configuration :

— Qualité d'entrée Comme mentionné au début de la Section 4.3, la qualité du modèle d'entrée influence la qualité des transformations. Ce sujet déborde du cadre de ce chapitre mais nous en avons discuté dans [AAM17]. L'animation et la simulation de modèles aident à valider les entrées. Lorsque l'environnement technique est maîtrisé, la transformation peut tisser le modèle avec le *framework* pour le rendre exécutable.

— Zoologie de DSL ou Profils Chaque macro-transformation (T1, T2, T3, T4) gère différents modèles écrits avec différents langages. Nous avons supposé ici uniquement les profils UML par souci de simplicité, mais le choix des langages (standards ou DSL) est vraiment un challenge pour la communauté MDE, tant du point de vue théorique que des outils. Là encore, l'idée de combiner de petites choses pour en faire de grandes, s'applique. Il est plus simple de définir des petits langages DSL et de les composer pour "construire" la sémantique.

— Parallélisme et processus de transformation De par sa simplicité, la FIGURE 4.8 masque la multiplicité des sous-modèles. Chaque transformation composite est un processus de transformation. En particulier, les macro-transformations (T1, T2, T3, T4) sont composites. Plus on progresse dans le processus, plus lance des transformations en *parallèle* $T1 \mapsto T2^n \mapsto T3^{n \times m} \mapsto T4^{n \times m \times p}$. Au début, T1 fonctionne avec une seule application globale. T2 s'applique à chaque sous-système de T1. T3 s'applique à chaque composant de T2. T4 s'applique à chaque classe de T3. Cependant l'ordre dans lequel sont traitées les transformations atomiques influencera le résultat final. Ce problème a été discuté dans le contexte de la vérification de modèle dans [AA05].

— Traçabilité Puisque les modèles sont obtenus par transformation, nous supposons que des liens de traçabilité sont insérés pour retracer l'origine des éléments du modèle à partir de PIM et/ou PDM *e.g.* au moyen d'annotations de modèle. Une originalité est de tracer depuis le PDM, ce qui n'est pas habituel en MDD.

— Itération Par essence, ce processus est génératif mais les transformations ne sont pas entièrement automatisées. Le choix de conception doit être fait par le concepteur du logiciel et une interaction de l'utilisateur est requise pour les transformations qui ne peuvent pas être automatisées. Rappelons que la partie GUI est considérée comme développée séparément. La collaboration homme/machine est à définir lorsque nécessaire. Une approche *Round trip* aiderait à stocker les informations utilisateur lors

de l'itération sur le processus de transformation.

— Support à la transformation Les expérimentations montrent qu'aucun outil de transformation n'était la panacée, notamment parce que divers styles de transformation sont impliqués *e.g.* synthèse, extraction, mappage, refactoring [Kar+06]. Le lecteur trouvera dans [Kah+19] une étude d'outils de transformation.

4.5 Expérimentations et illustration

Nous décrivons ici des expérimentations sur les transformations : la génération de code avec ATL dans la Section 4.5.1, un mapping d'API dans la Section 4.5.2 et une transformation ascendante avec une abstraction du *framework* Lejos dans la Section 4.5.3.

4.5.1 Transformation de code source avec ATL

Nous illustrons ici un exemple de l'approche par transformation *Model-To-Text* (M2T) qui génère du code source à partir des modèles UML issus de la transformation T3. à partir de profils UML pour la programmation à objets (diagrammes de classes détaillés pour les opérations). Pour analyser le modèle XMI et générer le code Java, nous avons défini un moteur de transformation ATL composé d'un ensemble de règles de sous-transformation. Nous classons ces règles en deux familles :

1. *Générer la structure du code source* Dans les transformations M2T, ATL fournit le concept d'assistants (méthodes) pour analyser le modèle XMI. Chaque assistant génère un morceau de code conforme à la syntaxe Java. L'assistant ATL (*helper*) de Figure 4.10 organise le processus d'analyse-génération en appelant des sous-règles.

```
helper context MM!Model def : GenerateJavaCode() : String =
    let  classes : MM!Class = self.ownedType->select(c | c.oclIsTypeOf(MM!Class)) in
       '/* \n'+
       ' * Automatically generated Java code with ATL \n'+
       '   Authors: Mohammed TEBIB & Pascal Andre \n'+
       ' */ \n'+
        classes->iterate(it; Class_Code: String = ''|Class_Code
          + thisModule.getImport(it.name) + '\n'
          + it.visibility +' class '+ it.name
          + if it.hasBehavior(it) then ' implements ' + 'I'+it.name+'StateMachine ' else '' endif
          + '{\n  '
          +'  //attributes \n'
          +'   ' + it.GenerateAttributes(it)
          +'\n\n    //methods \n'
          +'   ' + it.GenerateMethods(it)
          + '\n} \n'
          + it.GenerateInterfaces(it)
        )
;
```

Figure 4.10 – Transformation ATL pour les classes

— L'assistant GenerateClasses() analyse chaque classe présente dans le modèle XMI (UML-Java) et génère la structure du code de la classe Java. Cet assistant en appelle d'autres pour générer le code complet : (i) GenerateAttributes()

pour générer les attributs correspondant à chaque classe, (ii) GenerateMethods() pour générer uniquement la signature de chaque méthode, cet assistant pourrait être étendu dans le futur pour générer le corps de la méthode à partir du diagramme d'activité associé, et (iii) GenerateInterfaces() pour générer les interfaces modélisées si elles existent.

— L'assistant GenerateAttributes() analyse toutes les classes et génère toutes les informations liées aux attributs : visibilité, nom et type (voir FIGURE 4.11). Si la classe correspondante a un comportement décrit par STD, l'assistant génère également un attribut privé référençant l'état de l'objet courant.

```
--A method to generate the attributes of a given class
helper context MM!Class def : GenerateAttributes(x:MM!Class) : String =
   let  attributes : MM!Property = x.ownedAttribute->
            select(a | a.oclIsTypeOf(MM!Property)) in
               attributes->iterate(it; att: String = ''| att + '  '
            + thisModule.addAdapterAttributes(x.name)  + '\n'
            + it.visibility + ' '
            + if it.isStatic.toString()='false' then ' ' else 'static  ' endif
            + it.type + ' '
            + it.name + ';'
            + '\n  '
);
```

FIGURE 4.11 – Transformation ATL pour les attributs

— L'assistant GenerateMethods() génère la signature de méthode : visibilité, type renvoyé, nom et paramètres (voir FIGURE 4.12). Pour respecter la spécification du modèle *State*, cet assistant génère également une méthode appelée setState() pour initialiser l'attribut d'état correspondant.

```
--A method to generate the methods of a given class
helper context MM!Class def : GenerateMethods(x:MM!Class) : String =
 let  methods : MM!Operation = x.ownedOperation->
    select(a | a.oclIsTypeOf(MM!Operation)) in
      methods->iterate(it; att: String = ' '|  att + '  '
       + thisModule.mappingmethods(x.name, it.name) + '\n'
       + it.visibility
       + if(it.isStatic.toString()='true') then 'static '
         else ' ' endif
       + if(it.isAbstract.toString()='true') then 'abstract '
         else ' ' endif
       + if(it.type.toString()<>'OclUndefined') then
          if(it.type.toString().substring(1, 3)='<un') then
           it.type.toString().substring(11, it.type.toString().size())
          else if (it.type.toString().substring(1, 3)='IN!') then
                it.type.toString().substring(4, it.type.toString().size())
               else '' endif
          endif
         else 'void' endif
       + '  '+it.name + '('
       + '){\n'
       + '     }\n   '
    )
;
```

FIGURE 4.12 – Transformation ATL pour les méthodes

FIGURE 4.13 montre une partie des fichiers Java générés pour le cas d'étude.

FIGURE 4.13 – Exemples de classes générées par la transformation ATL

4.5.2 Transformation de code source par *mapping*

Dans l'étude de cas (*cf.* Section 4.2.2), les étudiants ont choisi d'implémenter la classe Motor par classe lejos.robotics.RegulatedMotor. Le faire par transformation en conception de bas niveau nécessite (a) de trouver des classes concrètes candidates et, une fois trouvées, (b) de relier les caractéristiques de la classe de conception aux caractéristiques de la classe concrète.

a) Correspondances candidates Pour chaque classe du modèle de conception *e.g.* Motor, l'objectif est de trouver des classes d'*implementation* candidates dans le *framework* vers lesquelles établir un lien. Un prérequis est d'avoir à disposition un modèle du cadre ou d'en établir un s'il n'en existe pas encore. A titre d'exemple, nous listons dans TABLE 4.2 les classes et candidats du modèle de conception. Pour l'exemple simple de la classe Motor, la TABLE 4.2 montre qu'il n'est pas facile de détecter quelle classe candidate pourrait être la bonne. Nous ne recherchons certainement pas de cartographie automatisée mais des algorithmes d'exploration (*mining*) pour détecter les candidats en fonction des noms (classe, attributs, opérations), l'utilisateur devra décider de la classe à associer. Si un PDM existe (*cf.* Section 4.5.3), une recherche automatisée sera plus facile à mettre en œuvre (que dans une API texte comme Javadoc).

TABLE 4.2 – Classes candidates à une liaison

Modèle	candidats Lejos	Décision	Commentaire
Motor	<<abstract>> Motor		La classe Motor contient 3 instances de 'regulated motors'.
	EV3LargeRegulatedMotor	Installé	En fait, cela dépend du matériel installé.
	42 candidats		Autres classes ou interfaces avec "*motor*.java"
	lejos.hardware.motor package		11 classes ou interfaces avec "*motor*.java" sur 13

Les classes d'implémentation proviennent de la bibliothèque Lejos (voir Section 4.5.3).

b) Adaptation Pour simplifier la description des attributs de mapping et leur injection dans le précédent moteur de transformation ATL, nous avons choisi de les représenter sous forme d'un fichier de propriétés contenant la liste des attributs de mapping. Suivant la spécification ATL tout fichier d'entrée doit avoir un format XMI et respecter une description définie par son méta-modèle. Dans l'exemple de la FIGURE 4.14, la classe (modèle) Motor délègue, via un adaptateur généré, ses appels de méthode au EV3Large RegulatedMotor.

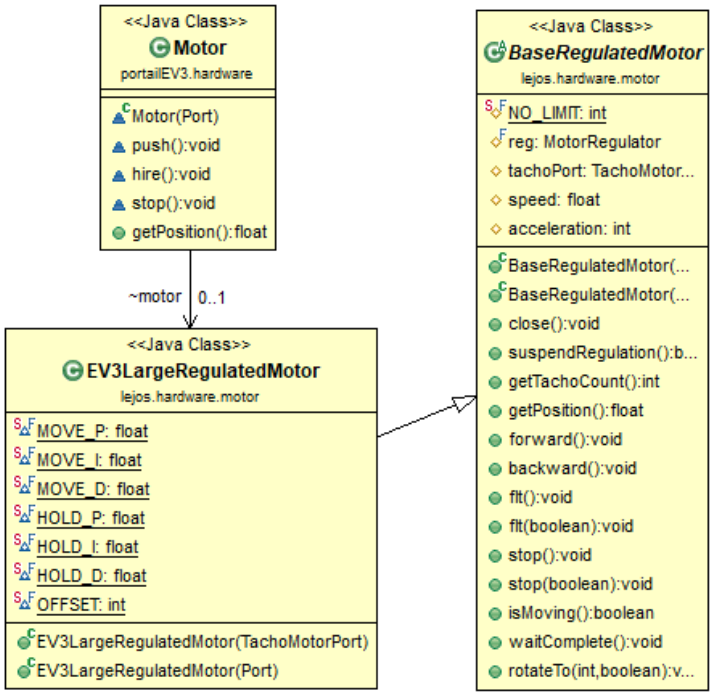


FIGURE 4.14 – Liaison de la classe `Motor` par adaptation

Consultez les détails et les résultats de la transformation aux pages 33 à 35 de l'annexe Web [18]. La transformation ci-dessus a fonctionné pour les correspondances directes basées sur le nom mais les développeurs devront sans doute coder des parties d'adaptations plus complexes.

4.5.3 Example : Rétro-ingénierie de bibliothèques `Lejos`

Dans notre étude de cas, nous utilisons le *framework* Lejos [19]. Pour abstraire un PDM Lejos, nous sommes partis de l'archive `ev3classes-src.zip` de la bibliothèque EV3. Noter que les autres bibliothèques Android/Java sont standard. Les métriques, extraites par le plugin o3smeasure, sont données dans la FIGURE 4.15.

Des expérimentations ont été menées avec `Papyrus`, `Modisco` et `AgileJ`. `Papyrus` a permis d'effectuer de l'ingénierie inverse [20] des classes individuelles mais pas des packages. Dans le cadre d'un projet papyrus, l'application de la commande `Java>Reverse` sur les

18. `https://aelos.ls2n.fr/at-medi2021app/`
19. Lejos est un système d'exploitation complet basé sur un *framework* Oracle JVM.
20. `https://wiki.eclipse.org/Java_reverse_engineering`

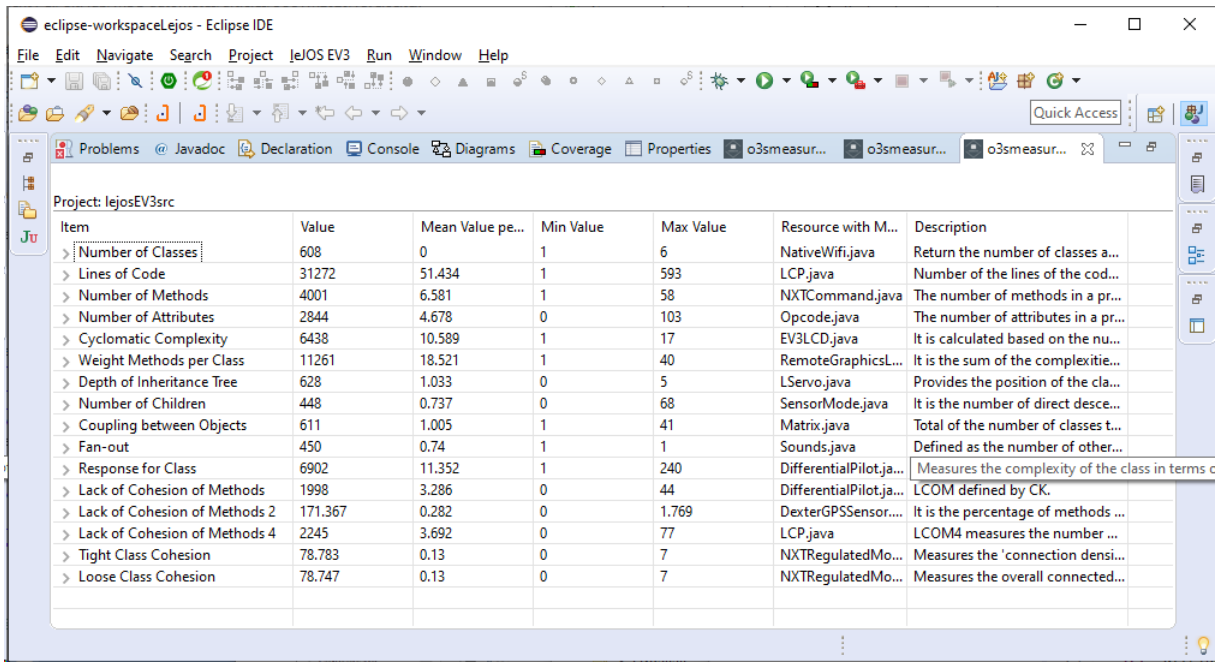


FIGURE 4.15 – Métriques de la bibliothèque de classes Lejos EV3

éléments du modèle lejosEV3src échoue sauf pour les classes. Même pour une classe, les méthodes n'étaient pas incluses.

Avec `Modisco` [Bru+14], la découverte UML à partir du code Java est composée de deux transformations (Java vers KDM / KDM vers UML). Malheureusement, le second n'est plus disponible dans la distribution Eclipse Modeling, mais reste disponible dans le dépôt git de Modisco. Encore une fois, nous avons été confrontés à deux problèmes de compatibilité ATL : les règles paresseuses ne sont pas autorisées en mode *refining* et le modèle distinct ... foreach est également interdit dans ce cas. Le mode *refining* a un modèle de sortie similaire à celui d'entrée ; il est adapté aux restructurations. De plus, les méthodes n'ont pas été reconnues en tant qu'éléments de modèle dans KDM.

Avec `AgileJ`[21], la rétro-conception du code Java en diagrammes de classes UML est simple. La FIGURE 4.16 montre un extrait du résultat de l'application de la rétro-conception sur la bibliothèque Lejos à l'aide d'AgileJ/structureviews. D'un point de vue visuel, on constate qu'il calcule de nombreuses relations entre classes par rapport à d'autres outils comme `ObjectAid` (*cf.* FIGURE 4.6 ou FIGURE 4.7). Les dépendances, induites par ces relations, peuvent être utilisées dans l'algorithme de *mapping* si on détecte d'autres classes logiques (pas seulement des classes d'implantation). En résumé, le problème n'est pas simple. Noter qu'`AgileJ` inclut un outil de filtrage utile qui identifie des patrons de conception, des structures spécifiques au projet et les conventions habituelles de programmation Java. Ces filtres contrôlent la manière dont les diagrammes de classes sont remplis et réduisent le bruit dans les informations présentées.

Dans cette expérimentation, l'unité de travail est la "classe". Pour chaque classe du modèle *e.g.* Motor, l'objectif est de trouver des classes d'implémentation candidates dans

21. https ://marketplace.eclipse.org/content/agilej-structureviews

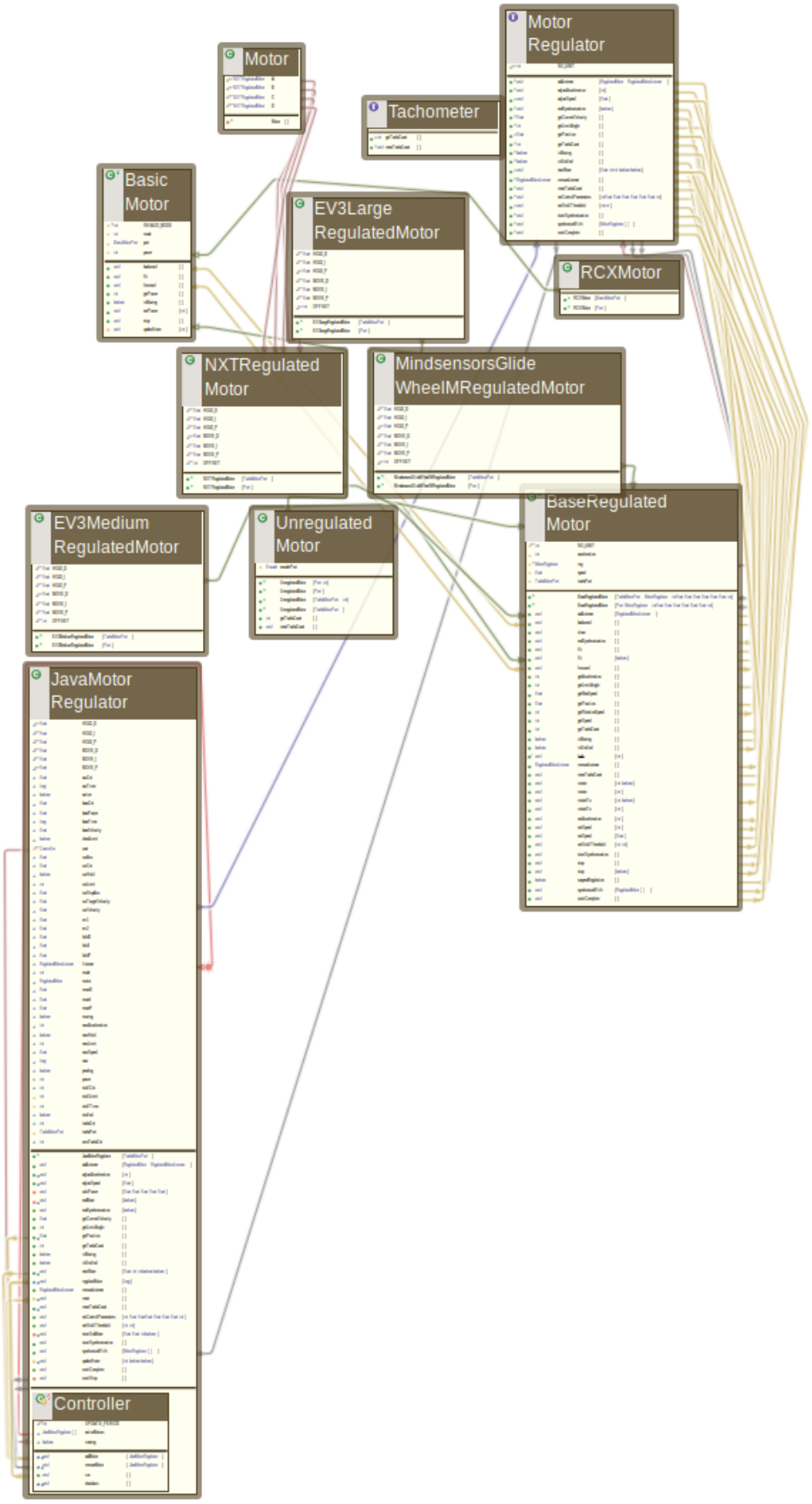


FIGURE 4.16 – Application du processus RE sur le package *Motor* de la bibliothèque Lejos

le modèle du *framework*. Le processus MDRE vise à fournir des abstractions candidates au *mapping*. Afin de réduire le nombre de candidats à comparer, nous appliquons les heuristiques simples suivantes : (i) se concentrer sur les fichiers sources Java (479 parmi les éléments KDM), (ii) sélectionner uniquement les interfaces (160) et les classes abstraites (19), car généralement les *framework*s sont structurés pour évoluer, (iii) rechercher des correspondance de noms (textes) (iv) ou mieux des correspondances de modèles (en tenant compte des attributs, des opérations et des références). Ces heuristiques peuvent être implantées sous forme de requêtes en `Modisco`. Des stéréotypes ou annotations spécifiques sont utiles pour séparer les classes de modèles dans le cas d'un traitement itératif.

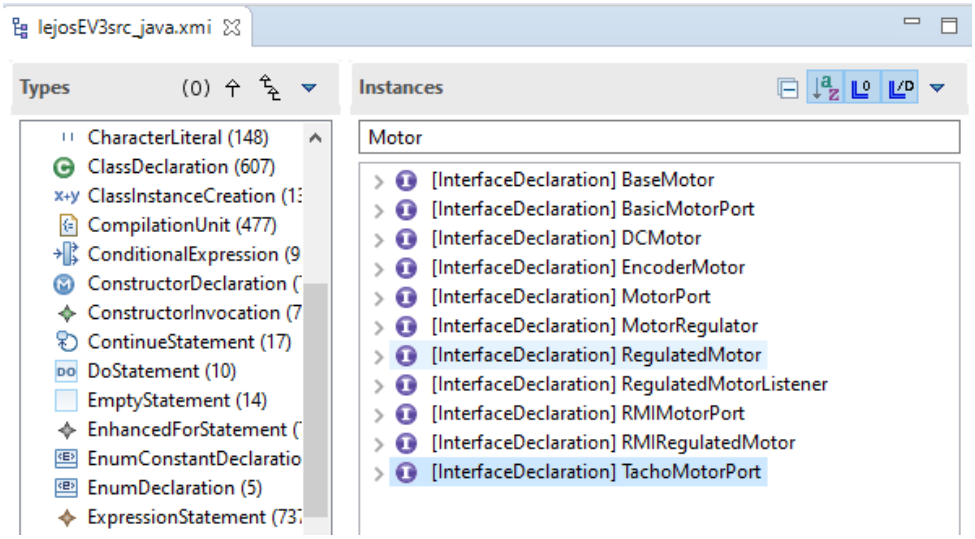


FIGURE 4.17 – Découverte d'interfaces avec Modisco

L'outil de filtrage d'`AgileJ` (*cf.* FIGURE 4.18) est suffisamment puissant pour réduire le bruit des éléments structurels clés. Une fois le filtre appliqué, il modifie le contenu de l'écran *e.g.* afficher toutes les interfaces ou afficher les classes abstraites. Dans l'exemple de la classe Motor, la correspondance de chaînes fournit 11 interfaces et la classe abstraite

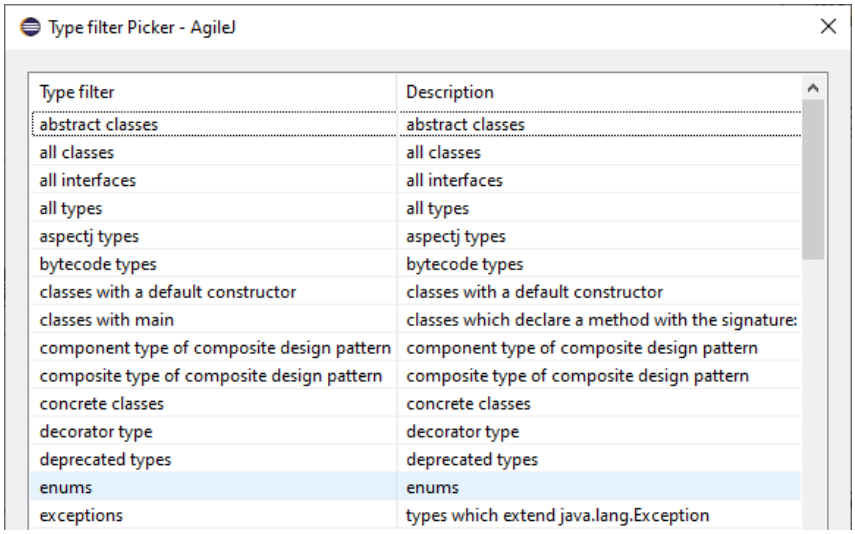


FIGURE 4.18 – Processus de filtrage avec AgileJ

BasicMotor. Il s'agit d'un ensemble raisonnable pour trouver des correspondances d'API potentiels (*pick and adapt*). Notez qu'`AgileJ` fournit des informations visuelles et interactives tandis que `Modisco` permet de personnaliser les requêtes et les transformations. D'autres expérimentations sont nécessaires avec `Papyrus RE` qui est toujours en phase de contribution. D'autres expérimentations sur MDRE peuvent être trouvées dans [And19] qui montrent la complexité du processus.

4.6 Conclusion

Dans les travaux mentionnés dans ce chapitre, on s'attaque aux mythes 2 (page 60) et 3 (page 60) mentionnées dans le chapitre 2.

Les expérimentations montrent que le développement logiciel reste une activité d'ingénierie (très coûteuse et très complexe). En pratique, les coûts sont réduits par les développeurs en capilisant sur les développements similaires en termes de métiers ET de technologie. Les modèles, s'ils sont utilisés au niveau architectural et dans les phases amont de conception, disparaissent dans les phases aval, limitant leur intérêt dès qu'on entre en phases de maintenance. Il existe des techniques et des outils de modélisation et de transformation de modèles qui peuvent s'appliquer à des situations spécifiques mais pas (encore) au développent dans son ensemble. Les expérimentations montrent aussi le potentiel de l'approche mais beaucoup de travail reste à faire pour mettre en œuvre les macro-transformations. La combinaison de différents langages de transformations sera nécessaire pour exploiter leur puissance.

Pour automatiser, il faut rationaliser, la contribution actuelle de ce travail est une vision du développement dans laquelle les modèles jouent un rôle fondamental. Nous proposons ici (i) une approche systématique de la conception par un processus MDD structuré avec des macro-transformations (être systématique permet de définir les activités afin de les automatiser, les macro-transformation correspondent aux grandes décisions de conception), (ii) un processus descendant et ascendant pour aligner les modèles logiques sur les modèles de cadre de mise en œuvre (l'originalité est que la rétro-conception se focalise uniquement sur l'aspect technique du développement pas le code de l'application), (iii) des définitions de transformations systématiques pour les transformations atomiques et (iv) l'implantation de certains d'entre elles.

Nous sommes convaincus que les éditeurs de logiciels devraient investir davantage dans l'automatisation des logiciels que dans le développement de logiciels. La contribution actuelle est un premier pas vers la question difficile posée dans l'introduction. L'idée n'est pas de proposer une méthode applicable à toute situation mais un canevas générique à spécialiser à un domaine spécifique ou une technologie spécifique. Dans ce second cas, on se rapproche de la vision **low code**, car on peut considérer que les abstractions des *frameworks* techniques fournissent des éléments pour un langage de plus haut niveau.

Des limites (*validity threats*) demeurent dans cette recherche empirique, principalement

en raison de la complexité du problème : (i) Si nous avons expérimenté quelques transformations, nous sommes encore loin d'un processus complet et nous sommes convaincus que tout ne sera pas automatisable. Prosaïquement, plus on est abstrait moins c'est automatisable, plus on distribue sur le web plus les transformations sont compliquées, les parties IHM sont à traiter à part (DSL possible), la persistance s'appuie sur des frameworks dédiés... (ii) La rétro-conception des frameworks, n'est effective sur T3, et encore elle est essentiellement structurelle, il faudrait remonter plus haut pour une abstraction complète des PDM. Sur notre cas d'étude, il faudrait aussi abstraire une partir d'Android pour la partie mobile, on voit vite le besoin de délimiter des contextes raisonnables. (iii) L'écart entre la définition d'une transformation et son automatisation est important pour les transformations composites, ce qui entraîne des problèmes de faisabilité, en particulier pour les transformations de haut niveau. Nous avons décrit une certaine applicabilité des transformations de bas niveau uniquement. (iv) Pour valider une approche, il faut l'évaluer sur de nombreux cas. La couverture des cas pratique la généricité nécessite de l'appliquer à différents types d'applications (web, mobile...) ce qui n'est pas encore le cas. En résumé, nous dessinons ici une vision, et non un manuel de conception ni un outil CASE, qui peut fournir des pistes de recherche à suivre. En outre, d'un point de vue pédagogique, nous avons trouvé ce travail utile pour expliquer la conception du logiciel aux étudiants de manière rationnelle.

Il reste un travail énorme pour obtenir une boîte à outils MDD. Un projet de recherche collaboratif serait une réponse efficace à ce défi. Nous avons commencé à implémenter une couche de communication pour affiner l'envoi de messages dans les communications Bluetooth et Wifi. Les travaux futurs visent à compléter la couverture à partir de transformations de bas niveau (*e.g.* T4) et en parallèle monter en abstraction sur le modèle PDM au niveau framework. En effet le processus de FIGURE 4.9 est organisé comme une pile telle que les transformations de bas niveau peuvent fonctionner indépendamment de celles de haut niveau. Une perspective à court terme est de continuer à écrire des transformations individuelles (petites étapes) qui seront composées hiérarchiquement jusqu'à atteindre des macro-transformations. Un enjeu particulier est de combiner des préoccupations transversales lors des transformations ainsi que des modèles multi-vues en intégrant des techniques de transformations orientées aspect. Une perspective à long terme consiste à fournir des lignes directrices de conception plus systématiques pour les étapes T2 et T1, basées sur les langages de conception architecturale.

Post-scriptum

Ce chapitre a mis en évidence des acquis sur les compétences suivantes, que nous reverrons dans le chapitre 8.

C2

Identifier les problèmes, les ordonner, imaginer des pistes ambitieuses, proposer des pistes réalistes, ordonnancer un ensemble d'activités de recherche pour structurer un travail de recherche scientifique sur du long terme.

C3

Organiser une recherche collaborative, détecter des compétences, définir des complémentarités et coordonner pour contribuer à un travail de recherche d'envergure

.

Chapitre 5

Améliorer la sécurité des applications

Security is a more challenging problem than safety.

Ian Sommerville [Som11]

Dans le chapitre 1 à la page 45, nous avons introduit les notions de propriététs fonctionelles et non-fonctionnelles. Jusqu'ici, je me suis surtout intéressé aux propriétés fonctionnelles, notamment dans la partie vérification du chapitre 3. Dans ce chapitre, j'aborde une catégorie de propriétés non-fonctionnelles, celle relative à la sécurité. Sommairement, garantir la sécurité c'est garantir l'intégrité d'un système, de son usage et de ses données. On doit éviter toute intrusion et tout détournement. C'est un véritable défi du fait que la sécurité recouvre le matériel ET le logiciel. Elle doit aussi prendre en compte la notion de contexte d'utilisation. Il faut non seulement protéger le système, garantir les accès par identification et authentification, mais aussi filtrer les entrées/sorties par exemple en crytant les communications. La contribution proposée reste très modeste au vu de l'immensité du champ.

Remarque

Ce chapitre reprend principalement les travaux de la thèse d'Abdramane Bah [Bah20] et les travaux en cours de la thèse de Mohammed El Amin Tébib.

5.1 Introduction

La sécurité est un casse-tête pour les responsables de production et les RSSI. On se doit de protégéer les données et leur confidentialité mais ainsi assurer le fonctionnement normal des applications. La sécurité est un vaste domaine qui va du matériel aux applications en passant par les infrastructures de communication. Mais souvent elle n'est pas pleinement pris en compte par les développeurs, car elle n'est pas considéré comme prioritaire vis-à-vis de l'adéquation au besoin. La sécurité a toujours fait partie du cursus des informaticiens mais souvent considéré de manière séparée du développement logiciel. Les consciences changent depuis quelques années car les risques sont accrus du fait non seulement du côté pervasif du numérique mais aussi de l'intégration des services entre sociétés. Les travaux

présentés ici restent à une échelle de contribution très modeste. L'idée est bien souvent de définir des modèles de sécurité et de vérifier des propriétés sur ces modèles.

Nous avons abordé deux thématiques dans nos travaux :

— la composition de services (web) pour mettre en place des services de haut niveau pose des problèmes de sécurité lorsque les services appelés sont distants, sur d'autres serveurs et que des droits d'accès sont nécessaires à l'exécution (implicite) de ces services qu'on peut qualifier de tiers car on ne les invoque pas directement. Nous avons étudié ce problème dans la thèse d'Abdramane Bah [Bah20] pour laquelle on se place dans une fédération de serveurs.

— la vérification de sécurité des applications mobiles, notamment autour des droits d'accès, car les smartphones contiennent énormément de données personnelles et les utilisateurs peu avertis sur la dangerosité de telle ou telle application téléchargée. Nous prenons le point de vue du développeur pour intervenir lors du développement des applications, les développeurs ne sont pas non plus des spécialistes des failles de sécurité du code qu'ils écrivent et ou des frameworks qu'ils utilisent.

5.2 Sécurité des services Web dans des fédérations

Cette section introduit les différents aspects des problèmes de contrôle d'accès que nous traitons dans cette problématique ainsi que les solutions que nous proposons.

5.2.1 Interopérabilité et sécurité

Les organisations ont besoin de collaborer en partageant des informations et des ressources pour atteindre leurs objectifs communs et respectifs. Cependant, les systèmes d'information des organisations sont naturellement indépendamment les uns des autres. Par conséquent, ils sont hétérogènes en termes de structure, de plateforme d'exécution ou de fournisseurs de logiciels. Une des solutions au problème d'interopérabilité des systèmes d'information est l'*l'architecture orientée services (SOA)*. Elle est implémentée avec les technologies des services web qui sont basées sur les protocoles standard d'Internet tels que HTTP. Avec SOA, les systèmes d'information deviennent accessibles à distance via des *services* entre les organisations partenaires. Dans ce contexte d'interopérabilité, la sécurité des accès et la sécurité mutuelle des systèmes d'information deviennent une exigence cruciale pour le partage des services entre les organisations.

Le premier besoin est une relation de collaboration et de confiance qui permet de restreindre les communications et le partage d'informations uniquement entre les organisations partenaires. Dans ce cercle de confiance, chaque organisation représente un **domaine de sécurité**, entité unique et autonome d'administration délimitée par des frontières de sécurité. Il existe trois modèles de collaboration entre des domaines autonomes : le modèle *point-à-point*, le modèle *étoile* et la **fédération**. La fédération est le modèle le plus flexible et dynamique. En outre, la fédération préserve l'autonomie des domaines notamment en

termes de sécurité. Une fédération est une collection de domaines autonomes qui s'associent pour mettre en commun et partager de manière sécurisée tout ou une partie de leurs systèmes d'information. Les domaines membres d'une fédération adhèrent à des règles, des politiques et des protocoles communs de collaboration.

Lorsque les services sont partagés entre les domaines d'une fédération, on parle de **fédération de services** ou *fédération SOA*. Ces services restent sous le contrôle de leur domaine. Une fédération de services permet de composer des services au-delà des frontières de sécurité des domaines pour créer de nouvelles applications et services composites. Dans une fédération de services, les communications sont entre les applications (*application-to-application*) et se font à travers les services web. Ce sont des applications modulaires conçues selon les principes de bonne conception logicielle tels que la forte cohérence et le faible couplage. Ils sont décrits par des contrats standards qui leurs permettent d'interagir à la volée avec les consommateurs de services. Cependant, la gestion individuelle de la sécurité des services représente un obstacle majeur à la fédération des services de différents domaines dans la mesure où chaque service est sécurisé de manière autonome dans un domaine de la fédération.

Le deuxième besoin de sécurité pour l'interopérabilité entre les domaines est le contrôle d'accès aux services partagés. Les utilisateurs de ces services ne sont pas connus dans tous les domaines. Pour sécuriser et contrôler l'accès aux services, les domaines peuvent jouer deux rôles distincts : fournisseur d'identités (*Identity Provider - IdP*) et fournisseur de services (*Relying Party - RP*). Les fournisseurs de services représentent les domaines qui partagent leurs services avec la fédération et les fournisseurs d'identités sont ceux qui consomment ces services. Un domaine peut être fournisseur de services et/ou fournisseur d'identités. Le **contrôle d'accès** (ou contrôle des permissions d'accès) comporte deux étapes : l'authentification et l'autorisation. Les utilisateurs sont authentifiés auprès de leurs domaines pour pouvoir accéder aux services des fournisseurs de services. Le contrôle est délicat dans une fédération de services : les services peuvent être consommés dans plusieurs domaines par la composition de services et un service partagé composite repose sur d'autres services fournis par d'autres domaines de la fédération. Chaque service a ses exigences de contrôle d'accès publiées via son contrat de services afin de permettre l'accès sécurisé à distance à ses informations.

Dans la section suivante, nous allons introduire les concepts fondamentaux du contrôle d'accès dans une fédération SOA, puis les difficultés auxquelles nous nous attaquons.

5.2.2 Contrôle d'accès dans une fédération SOA

Dans une fédération SOA, chaque domaine sécurise l'accès à ses services avec ses propres modèles et mécanismes de contrôle d'accès. Les services et les utilisateurs peuvent appartenir à des domaines distincts. Par conséquent, l'accès aux services se fait entre les domaines. Cependant, les services partagés entre les domaines doivent être accessibles uniquement aux utilisateurs autorisés de la fédération. Le contrôle d'accès est donc réparti

entre les domaines de la fédération. L'authentification des utilisateurs est déléguée à leurs propres domaines, tandis que l'autorisation des utilisateurs (*i.e.* l'octroi des permissions d'accès) et le contrôle des permissions d'accès restent sous le contrôle des fournisseurs de services. Nous appelons **contrôle d'accès fédéré**, cette distribution de l'authentification et de l'autorisation entre les domaines d'une fédération. Les difficultés à résoudre sont l'autorisation des utilisateurs à accéder aux services des domaines et la délégation de l'authentification des utilisateurs à leurs propres domaines.

Autorisation des utilisateurs

Seuls les utilisateurs autorisés doivent pouvoir accéder aux services. L'autorisation consiste à accorder des **permissions d'accès** aux utilisateurs en définissant des **politiques de contrôle d'accès**. Ces dernières sont définies avant le contrôle d'accès qui est effectué lors de l'accès aux services. Il existe plusieurs modèles d'autorisation dans la littérature ([HRS12] [Hu+14]) tels que les modèles basés sur les rôles (*Role-Based Access Control* - RBAC), les modèles basés sur les attributs (*Attribute-Based Access Control* - ABAC). Pour permettre l'accès à ses services offert, un domaine fournisseur de services doit explicitement autoriser les utilisateurs des autres domaines dans sa politique de contrôle d'accès. L'autorisation des *utilisateurs externes* consiste à leur accorder des permissions d'accès sur la base de leurs attributs d'autorisation (*i.e.* les privilèges dont ils disposent dans leurs propres domaines). Comme les utilisateurs ne sont connus que dans leur domaine, les autres domaines de la fédération n'ont aucune information pour leur accorder des permissions d'accès adéquates. Par ailleurs, chaque domaine accorde des permissions d'accès aux utilisateurs externes en utilisant son propre modèle et ses attributs d'autorisation afin d'être indépendant des autres domaines de la fédération.

La difficulté principale d'autorisation des utilisateurs dans une fédération est l'hétérogénéité des modèles et des attributs d'autorisation des domaines. Les autres difficultés sont liées à SOA telles que le couplage faible et la composition de services. Ces difficultés sont détaillées dans la Section 2.3 du Chapitre 2 de [Bah20]).

Authentification des utilisateurs

L'authentification des utilisateurs est une étape préliminaire au contrôle d'accès. Elle consiste à associer l'utilisateur à une identité numérique connue. Dans une fédération, l'identité des utilisateurs est connue uniquement dans leurs propres domaines (fournisseurs d'identités - IdP) qui sont les seuls capables de les authentifier. Dans une fédération SOA, le contrôle d'accès est également orienté services. L'authentification est effectuée par des *services de sécurité* notamment le service d'authentification. Chaque domaine possède ses propres services de sécurité. Les services offerts par un domaine ne font confiance qu'à ses services de sécurité. Ainsi, pour consommer un service dans un autre domaine, l'utilisateur est authentifié par son domaine et obtient un justificatif d'authentification appelé *jeton de sécurité* qui contient ses attributs d'autorisation locaux. Ce jeton est

envoyé aux domaines fournisseurs de services. Et sur la base des attributs d'autorisation contenus dans le jetons, les décisions d'accès (*e.g.* autorisé, refusé) sont déterminées. La confiance entre les domaines est le fondement de ce processus d'authentification pour accéder aux services.

Il existe des normes telles que *WS-Federation* [WS-F], Shibboleth [Kal08] qui assurent la gestion de confiance et formalisent le processus d'authentification entre les domaines. Dans le cadre de SOA, les services n'ont pas d'interface utilisateur. Le processus d'authentification est donc transparent aux utilisateurs. De ce fait, *WS-Federation* (qui est basée sur les normes de sécurité des services web telles que *WS-Trust, WS-Security*) est la norme adaptée aux environnements orientés services. Elle permet aussi aux domaines d'avoir des mécanismes hétérogènes d'authentification. Cependant, les normes de fédération d'authentification y compris *WS-Federation* supposent l'existence d'un modèle et un ensemble d'attributs d'autorisation communs à tous les domaines de la fédération. Cela rend difficile l'authentification des utilisateurs dans une fédération SOA où les domaines ont différents modèles et attributs de contrôle d'accès.

Contrôle des permissions d'accès aux services

Bien que l'authentification établisse l'identité des utilisateurs et fournisse leurs attributs d'autorisation, elle ne permet pas de déterminer si l'utilisateur possède les permissions d'accès aux services. Le contrôle des permissions d'accès est la phase qui permet de vérifier si les attributs contenus dans le jeton de sécurité de l'utilisateur sont autorisés dans la politique de contrôle d'accès des domaines fournisseur de services. Dans SOA, ces attributs sont exigés par les services. En effet, chaque service a ses propres exigences de contrôle d'accès qui sont définies dans sa politique de sécurité. Les appels de services doivent être sécurisés conformément à cette politique de sécurité. La phase de contrôle d'accès consiste à vérifier d'abord si les appels de services sont dignes de confiance (*i.e.* provenant des domaines de la fédération) ; ensuite si les appels sont sécurisés conformément à la politique de sécurité du service et enfin si les attributs d'autorisation contenus dans le jeton de sécurité répondent aux exigences de contrôle d'accès du service. Cependant, l'hétérogénéité des modèles et des attributs d'autorisation des domaines engendrent aussi l'hétérogénéité des exigences de contrôle d'accès des services. Par conséquent, les services des domaines sont hétérogènes en termes de contrôle d'accès. Ce qui représente un obstacle à l'accessibilité et à la composition des services des domaines de la fédération.

Comme pour les modèles et les attributs d'autorisation, chaque domaine possède ses propres mécanismes de contrôle d'accès. Il existe cependant une architecture standard spécifiée par la norme *XACML* [XAC3] pour l'évaluation des décisions d'accès aux services dans SOA. Cependant, le partage des services entre les domaines engendre deux points d'accès aux services : à l'intérieur du domaine et depuis l'extérieur du domaine. En effet, les services offerts par un domaine sont des services utilisés à l'intérieur de ce domaine. De ce fait, les appels de services viennent aussi bien de l'intérieur que de l'extérieur

des domaines. Bien que les exigences de sécurité pour les appels internes peuvent être différentes de celles des appels externes, ces deux types d'appels de services doivent être traités par le même mécanisme existant de contrôle d'accès.

Défis et exigences

La fédération de services est confrontée à des défis majeurs. (i) Hétérogénéité des modèles d'autorisation d'accès. Chaque domaine précise ses politiques de contrôle d'accès sur ses propred attributs d'autorisation *e.g.* un rôle. Lorsque les domaines utilisent des attributs d'autorisation avec des sémantiques incompatibles, l'accès aux services est soit empêché, soit accordé à des personnes non autorisées. (ii) Un domaine peut appartenir à différentes fédérations ou collaborer avec un pair. En aucun cas, le contexte ne doit interférer avec le contexte local. (iii) Composition des services fédérés. La composition de services doit prendre en compte le contrôle d'accès de chaque service et donc l'hétérogénéité des attributs d'autorisation de domaine.

La fédération sécurisée des services de domaines indépendants doit répondre aux exigences suivantes :

— Authentification fédérée unique (*Single federated sign-on*). Un utilisateur doit pouvoir s'authentifier une fois auprès de la fédération puis utiliser les services pour lesquels il dispose d'une autorisation valide.
— Autorisation décentralisée. Les utilisateurs doivent obtenir les autorisations de leurs domaines et accéder aux services sur la base de ces autorisations.
— Autonomie des domaines. Chaque domaine contrôle l'accès à ses services.
— Adaptation dynamique à la croissance de la fédération. Les mécanismes de contrôle d'accès du domaine ne devrait pas nécessiter d'efforts de maintenance importants lors des changements d'autorisation dans la fédération.
— Confidentialité des informations de sécurité interne. Les attributs d'autorisation sont des informations sensibles et ne doivent pas être divulguées au-delà des limites du domaine.

Dans [Bah20], nous proposons trois contributions pour répondre à ces besoins : une *méthode d'autorisation* et un *mécanisme de contrôle d'accès fédéré* basé sur le *mapping* d'attributs pour assurer l'interopérabilité entre les modèles et les attributs d'autorisation des domaines et une *méthode de fédération de services* pour assurer l'interopérabilité des services en termes de contrôle d'accès. Ces contributions reposent sur une nouvelle architecture de fédération de domaines spécifique à SOA que nous avons définie. Elle sert de fondement pour le partage sécurisé des services entre les domaines ainsi que l'autorisation des utilisateurs et le contrôle d'accès des services partagés.

Dans la suite, nous présentons les définitions (Section 5.2.3) puis l'architecture de fédération de domaines (Section 5.2.4), la méthode de fédération et d'autorisation des utilisateurs (Section 5.2.5), la méthode de mise à disposition de services dans la fédération (Section 5.2.6), le mécanisme de contrôle d'accès fédéré basé sur le *mapping* d'attributs

(Section 5.2.7) et enfin la compositon de services fédérés (Section 5.2.8). Ces sections s'inspirent du chapitre 4 de [Bah20] et de [Bah+19 ; Bah+20b].

5.2.3 Fédération de domaines et de services

Pour assurer l'interopérabilité entre les modèles et les mécanismes de sécurité des domaines, nous définissons une nouvelle architecture de fédération. Cette architecture sert de fondation pour : (i) l'autorisation flexible et dynamique des utilisateurs entre les domaines ; (ii) la fédération sécurisée des services des domaines ; (iii) l'interopérabilité entre les mécanismes de contrôle d'accès des domaines. La suite définit formellement les domaines et la fédération puis présente l'architecture de la fédération.

A) Domaine de sécurité et services

Un domaine de sécurité (ou simplement un domaine) est une unité unique d'administration de la sécurité. Il comprend une architecture orientés services (SOA) composée d'un ensemble d'utilisateurs (ou consommateurs de services), un ensemble de services, un registre de services, une politique de sécurité qui régit l'accès des utilisateurs aux services et un ensemble de services de sécurité qui appliquent la politique de sécurité.

Définition 5.1 (Domaine) *Un domaine d_i est défini par un tuple $\langle U_i, S_i, R_i, SP_i, SS_i \rangle$ où U_i est l'ensemble d'utilisateurs ; S_i est l'ensemble de services ; R_i est le registre de service qui contient un sous-ensemble des services de S_i **publiés** ; SP_i est la politique de sécurité et SS_i est l'ensemble des services de sécurité de d_i.*

Comme illustré sur la FIGURE 5.1, la politique de sécurité du domaine SP_i régit l'accès des utilisateurs aux services S_i et elle est mise en œuvre par les services de sécurité SS_i.

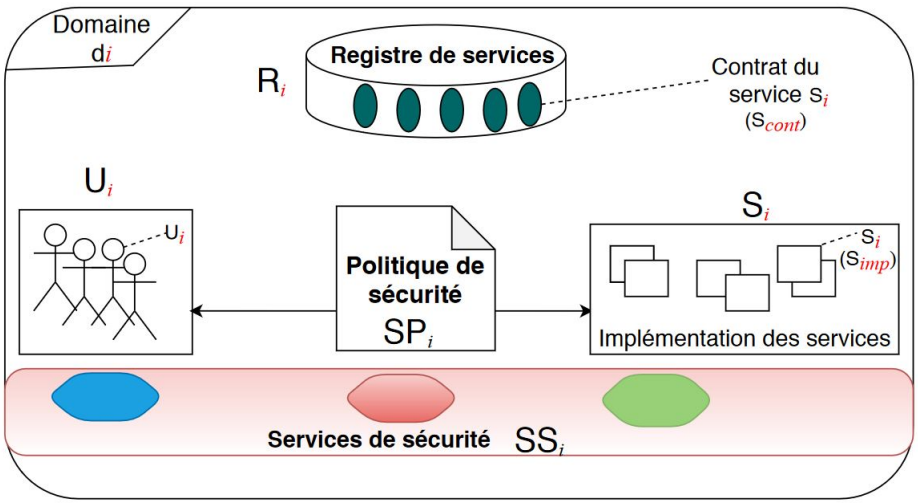


FIGURE 5.1 – Composants d'un domaine de sécurité (source : [Bah20], p. 68)

Définition 5.2 (Utilisateur) *Un utilisateur u_i de U_i est caractérisé par ses attributs ; $u_i = \langle uID, uAT \rangle$ où uID est l'ensemble de ses attributs d'identité (e.g. son nom) et uAT est l'ensemble de ses attributs d'autorisation tels que son rôle (e.g. enseignant).*

Un service est une unité logique de traitement, autonome et auto-descriptive qui permet d'accéder à distance aux informations et aux fonctionnalités métiers d'un domaine. Un service comprend deux parties séparées [DVL13 ; Ber+10] : un contrat de service s_{cont} et une implémentation de service s_{imp}. Le contrat de service (*cf.* Section 3.2.3 du Chapitre 3) décrit les fonctionnalités offertes par le service ainsi que sa politique de sécurité (*e.g.* les exigences de sécurité). L'implémentation réalise le contrat de service.

Définition 5.3 (Service) *Un service est un tuple $s = \langle I, P, Edp \rangle$ où I est l'interface du service, P sa politique de sécurité et Edp l'adresse sur laquelle le service reçoit des invocations de messages ; cette adresse s'appelle endpoint.*

— L'interface de service décrit l'ensemble des opérations fournies par le service et les protocoles utilisés pour y accéder. La politique de sécurité du service décrit les capacités (par exemple, les algorithmes de cryptage pris en charge) et les exigences de sécurité du service ; par exemple, les parties du message d'appel qui doivent être chiffrées, les exigences de protection des messages sous forme d'attributs (a_i) ou *termes* (combinaison des attributs telles que $a_i \; OR \; (a_j \; AND \; a_k)$).

— Pour simplifier la politique de sécurité du service, les exigences de protection sont abstraites sous forme d'attributs d'autorisation ou de termes $l_r \in R[X]$ où X est l'ensemble des attributs utilisés pour exprimer les exigences R de la politique de sécurité P ; d'où la notation $s = \langle I, P_{R[X]}, Edp \rangle$.

Les permissions d'accès aux services sont définies et contrôlées à l'aide de la politique de sécurité du domaine (définition 5.1).

Définition 5.4 (Politique de sécurité) *La politique de sécurité du domaine $_i$ est définie par $SP_i = \langle AT, AR \rangle$ où AT est l'ensemble des attributs d'autorisation de d_i et AR est l'ensemble des règles d'autorisation basé sur les attributs de AT (par exemple, les règles décrivent quels attributs a_k sont autorisés à accéder à un service s_i : $\{(s_i, a_k)\}$).*

Les services de sécurité SS_i (définition 5.1) incluent le service d'authentification nommé *service de jeton local* (LTS), le service d'autorisation (ATS) et l'intercepteur $Interceptor$. Ces services de sécurité assurent le contrôle d'accès des services. Détaillons leur role.

Les services effectuent des actions pour un utilisateur (*e.g.* humain, application) [Pap08]. Le contrôle d'accès assure que seuls les utilisateurs autorisés ont accès aux services conformément à la politique de sécurité du domaine SP. Considérons par exemple, un service de transfert d'argent d'une banque (TRANSFERTSERVICE) qui *débite* un premier compte bancaire d'un montant donné pour *créditer* un second compte du même montant. La politique de sécurité de la banque pour ce service décrit quel utilisateur dans la banque (employés, clients) a accès au service transfertService et dans quelles conditions.

Les services de sécurité implémentent le contrôle d'accès selon une architecture XACML [Fer+16]. Le service de sécurité *Interceptor* représente le *policy enforcement point* (PEP) d'XACML et le service d'autorisation *ATS* représente le *policy decision point* (PDP). Pour accéder à un service, l'utilisateur est authentifié auprès du service d'authentification (LTS) (*cf.*FIGURE 5.2, étape 1) qui délivre un jeton de sécurité. Ce jeton est utilisé pour invoquer le service (FIGURE 5.2, étape 2).

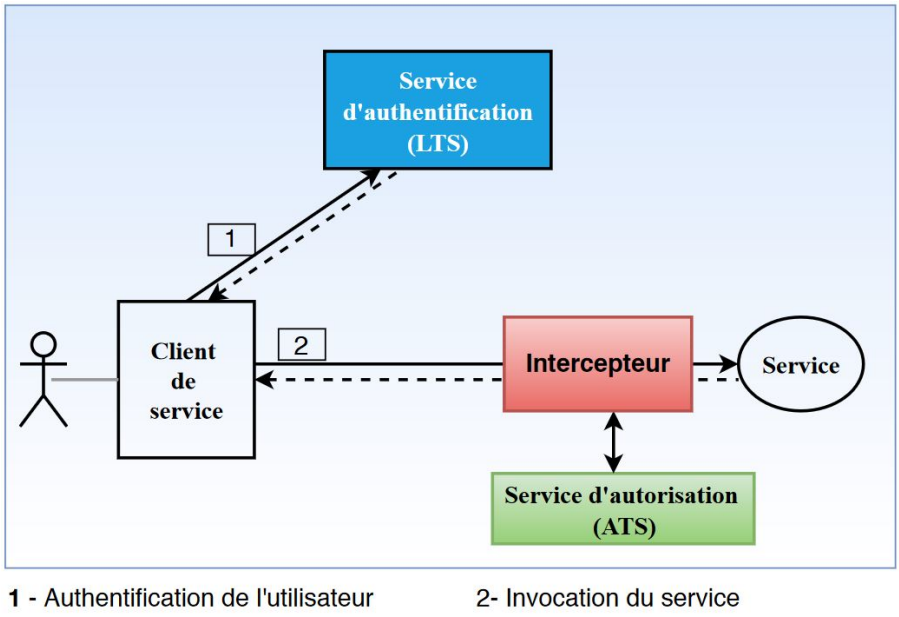


FIGURE 5.2 – Processus de contrôle d'accès dans un domaine (source : [Bah20], p. 70)

Le service de sécurité *Interceptor* intercepte tous les messages d'appel adressés aux services d'un domaine ; il a plusieurs composants incluant une fille d'attente *CallQueue*. Dans la suite, nous utilisons la notation $Interceptor{\downarrow}CallQueue$ pour indiquer la sélection de la file d'attente. Le service *Interceptor* joue trois rôles : service de vérification de la sécurité des messages, service de vérification de la confiance et le point d'application des décisions de contrôle d'accès (PEP). Il valide les messages adressés aux services conformément à leurs politiques de sécurité. Il extrait les attributs d'autorisation de l'utilisateur du jeton de sécurité intégré dans le message d'appel et demander une décision d'accès au service d'autorisation *ATS*. Ce dernier vérifie si les attributs de l'utilisateur disposent des permissions nécessaires pour exécuter le service conformément aux politiques de sécurité du domaine (SP_i). Ce processus de contrôle d'accès est pris en compte lors du partage des services dans une fédération afin de minimiser les dépendances entre les domaines.

B) Fédération

Une fédération F est un ensemble de domaines autonomes qui adhèrent à des règles et des politiques de gouvernance communes qui régulent les interactions entre eux. Les administrateurs des domaines impliqués s'engagent à définir et à suivre les règles de sécurité communes et les accords de médiation. La fédération crée un environnement de confiance pour le partage nsécurisé des services entre domaines. Nous considérons

une fédération comme un domaine, à la différence que ses utilisateurs et ses services proviennent des domaines qui la composent.

Définition 5.5 (Fédération) *Une fédération F de n domaines $d_i = \langle U_i, S_i, R_i, SP_i, SS_i \rangle$ par le tuple $F = \langle U_f, S_f, R_f, SP_f, SS_f \rangle$ où*
U_f est l'union des utilisateurs de ses domaines ($U_f = \bigcup_{i=1}^{n}(U_i)$) ; SP_f est la politique de sécurité de la fédération et SS_f les services de sécurité de la fédération.

Cette définition est uniforme avec la définition 5.1 du domaine. Initialement S_f et R_f sont vides à la création de la fédération. En conséquence, SS_f et SP_f sont également vides. Dans la section 5.2.6, nous montrons comment S_f and R_f sont construits en promouvant les services de domaines au niveau de la fédération.

5.2.4 Architecture de la fédération

Dans une fédération classique [Kal08], chaque domaine joue un des rôles suivants : fournisseur d'identité ou (*Identity Provider* - IdP) et fournisseur de services ou (*Relying Party* - RP). Les IdPs gèrent et administrent l'identité et les attributs des utilisateurs. Les RPs gèrent les services et leur sécurité. Le contrôle d'accès dans la fédération est également réparti entre ces rôles (IdP et RP). Les IdPs authentifient les utilisateurs et les RPs autorisent ou non l'accès aux services sur la base de cette authentification. Pour faciliter l'établissement de la confiance entre les IdPs et les RPs, une fédération peut comporter également une autorité de la fédération qui gère l'adhésion et la sortie des domaines dans la fédération.

Dans une fédération SOA, nous considérons les domaines comme des fournisseurs de services (RP) et des fournisseurs d'identités (IdP) à la fois. La définition globale des attributs de contrôle d'accès entre les domaines et de leurs sémantiques permet de déléguer l'authentification des utilisateurs à leurs domaines. L'interopérabilité sémantique entre les modèles et les attributs d'autorisation des domaines permet de préserver leurs mécanismes de contrôle d'accès.

Pour gérer le lien entre domaine et fédération, nous ajoutons un nouveau rôle dans la fédération : **le médiateur global de contrôle d'accès** (*Global Access Control Mediator - GACM*). Comme son nom l'indique, le GACM sert de médiateur d'interopérabilité pour le contrôle d'accès entre les domaines. Les domaines ne sont pas interopérables parce qu'ils ont des modèles, des attributs et des mécanismes de contrôle d'accès hétérogènes. Le GACM n'est pas un intermédiaire de contrôle d'accès, ni un intermédiaire d'autorisation des utilisateurs. Le GACM ne définit aucune politique de contrôle d'accès et n'évalue aucune décision d'accès aux services des domaines. L'interopérabilité est réalisée de façon *ad-hoc*. Les domaines inter-opèrent en établissant des correspondances (ou *mapping*) entre leurs modèles et attributs de contrôle d'accès. Le GACM sert principalement établir des politiques *mapping* pour le contrôle d'accès des domaines. Le rôle GACM est peut

être assumé par un domaine existant, par l'autorité de la fédération ou par un nouveau domaine. Cependant, le GACM en tant que autorité de la fédération est plus judicieux.

Définition 5.6 (GACM) *Nous définissons le GACM en tant que domaine comme un tuple : $GACM = \langle U_f, S_f, R_f, SP_f, SS_f \rangle$ où U_f est l'union des utilisateurs de ses domaines ($U_f = \bigcup_{i=1}^{n}(U_i)$) ; S_f l'ensemble des services partagés par les domaines ; R_f le registre de services de la fédération, SP_f est la politique de sécurité de la fédération et SS_f les services de sécurité de la fédération.*

Le GACM représente la fédération pour les domaines. Initialement S_f et R_f sont vides à la création de la fédération. En conséquence, SS_f est également vide.

Le modèle ABAC est choisi comme pivot pour le GACM car il permet une interopérabilité entre les modèles de contrôle d'accès (*e.g.* RBAC, MAC) [HRS12]. Les concepts des autres modèles sont interprétés comme des attributs. Le GACM définit un ensemble d'attributs d'autorisation standards pour la fédération appelés **attributs fédérés**. Nous définissons la politique de sécurité de la fédération comme suit : $SP_f = \langle AF \rangle$ où AF est l'ensemble des attributs fédérés. SP_f ne comporte pas de règles d'autorisation car le GACM ne définit pas de politique de contrôle d'accès.

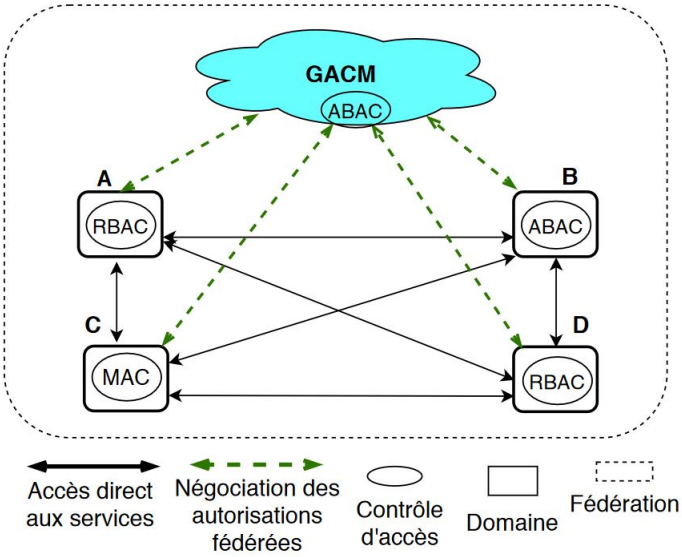


FIGURE 5.3 – Architecture de fédération proposée (source : [Bah+19], [Bah20], p. 74)

L'autorisation des utilisateurs entre les domaines est assurée par l'intermédiaire du GACM (flèche en pointillés sur la FIGURE 5.3). Les appels de services se font directement entre les domaines (flèche nettes sur la FIGURE 5.3). La gestion des services et de leur sécurité reste sous le contrôle des domaines. Le GACM sert uniquement à l'interopérabilité de contrôle d'accès entre les domaines. Les attributs fédérés permettent de déléguer l'authentification des utilisateurs à leurs domaines.

Le choix des attributs fédérés, la gestion des conflits relève de l'administration de la fédération et se fait sur des règles de gestion organisationnelles. Le comité qui gère

la fédération, constitué entre autres de représentants de chacun des domaines décide de quels seront les attributs retenus et de leurs caractéristiques associées.

Le contrôle d'accès à l'intérieur des domaines est toujours réalisé avec les attributs locaux et les mécanismes de contrôle d'accès existants. Le pont entre le contrôle d'accès à l'intérieur des domaines et entre les domaines est établie par des politiques de *mapping* d'attributs. Dans la section suivante, nous présentons notre méthode d'autorisation des utilisateurs basée sur le *mapping* d'attributs.

5.2.5 Méthode de fédération et d'autorisation des utilisateurs

La fédération des utilisateurs consiste à définir des attributs d'autorisation communs à tous les domaines pour le contrôle d'accès. A cause de leur autonomie de fonctionnement, les domaines ne peuvent abandonner leurs modèles de contrôle d'accès au profit d'un modèle global. Par conséquent, pour fédérer les utilisateurs des domaines, nous avons besoin définir des correspondances (ou *mapping*) entre les attributs d'autorisation des domaines afin d'assurer l'interopérabilité entre les modèles et les attributs de contrôle d'accès existants des domaines.

Le GACM ne définit pas de politiques de contrôle d'accès au domaine et n'accorde pas d'autorisations d'accès aux utilisateurs. Il représente l'autorité de la fédération servant de pont entre les domaines. L'objectif principal du *mapping* d'attributs pour les domaines est de comprendre les attributs d'autorisation de chacun afin de déterminer les autorisations d'accès locales pour les attributs d'autorisation externes. Cependant, la fédération évolue ; de nouveaux domaines le rejoignent avec de nouveaux attributs d'autorisation et d'autres partent. Le *mapping* d'attributs doit s'adapter dynamiquement à l'évolution des changements d'attributs de fédération et d'autorisation dans les domaines. Pour atteindre ces objectifs et éviter les fuites d'informations de sécurité, le GACM définit les attributs d'autorisation de la fédération, les *attributs fédérés*, indépendamment de ceux des domaines. Les attributs fédérés sont publics et compréhensibles par tous les domaines.

Nous définissons des *mapping* entre les attributs fédérés et les attributs d'autorisation de domaine à deux niveaux comme le montre la figure 5.4 :

— Au niveau du GACM : les domaines négocient une seule fois avec le GACM, les *mappings* entre leurs attributs d'autorisation et les attributs fédérés. Ce premier *mapping*, appelé *federated mapping* est enregistré dans le GACM et une copie est déposée dans les domaines ;

— Au niveau du domaine : chaque domaine définit localement les *mappings* entre les attributs fédérés et ses attributs d'autorisation. Il est appelé *domain mapping*.

Les interactions entre domaines sont ensuite effectuées à l'aide d'attributs fédérés. Grâce aux attributs fédérés, les domaines peuvent accorder des autorisations d'accès à tous les autres domaines de la fédération sans connaître leurs attributs d'autorisation locaux. De ce fait, les domaines peuvent accéder aux services les uns des autres malgré l'hétérogénéité de leurs attributs d'autorisation. L'avantage offert par notre approche aux

utilisateurs est d'obtenir les autorisations d'accès dans d'autres domaines en fonction de leurs attributs d'autorisation d'origine.

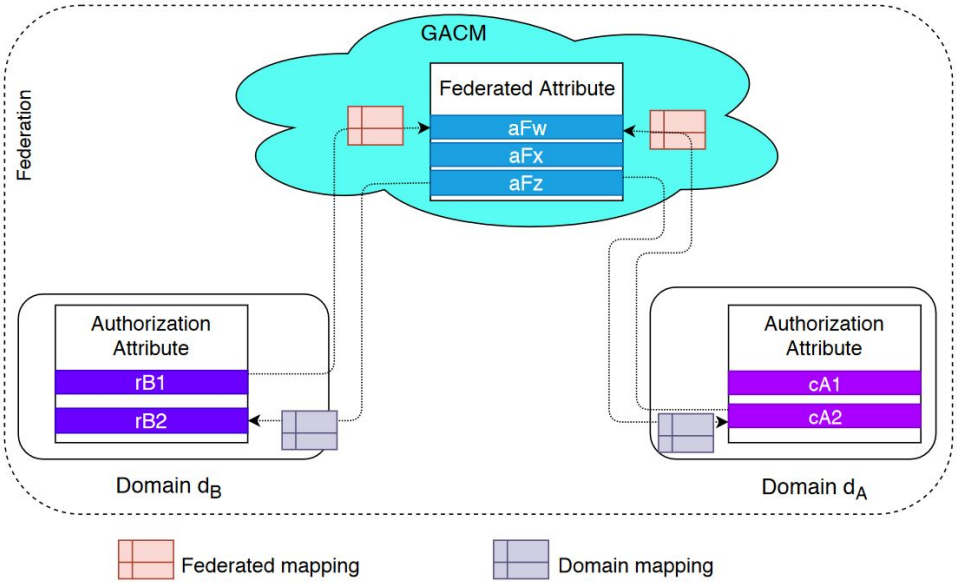


FIGURE 5.4 – *Mapping* des attributs d'autorisation de domaine (source : [Bah+19])

Le *mapping* d'attributs sert principalement de solution d'interopérabilité entre les attributs d'autorisation hétérogène des domaines. Cette hétérogénéité des attributs d'autorisation se présente sous trois aspects : conceptuelle, de définition et sémantique. L'hétérogénéité conceptuelle est liée au modèle de contrôle d'accès (*e.g.* RBAC, MAC, ABAC). Par conséquent, les attributs d'autorisation des domaines peuvent être des rôles (RBAC), des niveaux de sécurité (MAC) etc... L'hétérogénéité de définition se manifeste lorsque les domaines utilisent le même modèle de contrôle d'accès, mais des définitions différentes des attributs d'autorisation. Par exemple, deux universités *U1* et *U2* utilisant un modèle RBAC peuvent avoir les définitions suivantes des rôles : pour U1,les rôles sont *enseignant, etudiant, administrateur* et pour U2 les rôles sont *enseignant-permanent, ater, doctorant, etudiant, administrateur*. L'hétérogénéité sémantique concerne la signification des attributs d'autorisation. Dans l'exemple ci-dessus, le sens des rôles *administrateur* de U1 et U2 peut être différent. En outre, un administrateur de U1 et U2 peut avoir des privilèges différentes sur les services de U2.

Définition des politiques de mapping

Notre méthode d'autorisation soulève une question essentielle qui est : comment définir les politiques de *mapping* fédérés (PMF) et de domaines (PMD) ?

Le *mapping* d'attributs consiste à définir des relations entre les attributs d'autorisation de deux domaines. Ces relations peuvent être des relations d'équivalence [PJIZ18], d'égalité [LTL10] ou des relations de correspondances [LTL10] entre les noms et les valeurs des attributs [LTL10][ZYW12]. En effet, les attributs ont des noms et des valeurs. Les relations d'équivalence permettent de créer des relations un-à-un (1 :1) entre les attributs.

Par exemple, un attribut A (*e.g. rôle=manager*) du domaine d_1 équivaut à un seul attribut B (*e.g. job-role=administrateur*) du domaine d_2. De ce fait, elles résolvent les problèmes de *synonyme* et d'*homonyme* (*e.g. role=mere* et *role=maire*) entre les attributs. En outre, les relations d'équivalence établissent une relation entre les attributs dans un sens comme dans l'autre. Par exemple, si un attribut A équivaut à un attribut B alors B équivaut à A. Cependant, les relations d'équivalence ne sont pas applicables lorsque les domaines ont des noms d'attributs différents (*e.g.* rôle (RBAC) et niveau de sécurité (MAC)).

Par contre, les relations de correspondances ce problème d'hétérogénéité des noms d'attributs et permettent d'établir des relations un-à-plusieurs (1 :n), plusieurs-à-un (n :1) en plus des relations un-à-un (1 :1). Les relations de correspondances sont des relations d'implication, donc à sens unique. Par exemple, un attribut A d'un domaine d_i peut correspondre à un attribut B d'un domaine d_j, mais réciproquement, l'attribut B du domaine d_j ne correspond pas à l'attribut A du domaine d_i. Le principal avantage des politiques de correspondance est qu'elles ne peuvent pas être utilisées à la fois pour consommer des services hors d'un domaine et pour autoriser l'accès aux services de ce même domaine.

Ainsi, nous avons choisi de définir les politiques PMF et PMD avec des relations de correspondance entre les attributs locaux des domaines et les attributs fédérés du GACM. L'objectif est d'assurer que les politiques de *mapping* PMF et PMD soient à sens unique. Nous décrivons les politiques de *mapping* à l'aide des règles logiques [LTL10].

Authentification et tiers de confiance

Le contrôle d'accès aux services repose sur les attributs d'autorisation des utilisateurs revendiqués par un tiers de confiance. Chaque domaine possède son propre mécanisme d'authentification appelé *local token service* (LTS). Le LTS authentifie les utilisateurs et émet un jeton de sécurité signé par le certificat de sécurité du domaine. Les services d'un domaine ne sont accessibles qu'avec un jeton de sécurité émis par le LTS du domaine.

Afin d'établir la confiance entre les domaines, nous introduisons dans le GACM un mécanisme d'authentification spécialisé appelé *federated token service* (FTS) pour l'authentification de domaine. Nous identifions les domaines et les GACM avec les certificats à clé publique. Les certificats de sécurité des domaines sont transmis au GACM qui transmet à son tour son certificat aux domaines. Les domaines s'authentifient auprès du GACM avec les tokens de sécurité signés avec leurs certificats de sécurité. En réponse, le FTS délivre les jetons de sécurité signés par le certificat de sécurité du GACM. Par conséquent, les domaines de la fédération se font confiance via les *jetons de sécurité fédérés*.

Pour accéder à un service (S_B) du domaine B (d_B) depuis un domaine A (d_A), l'authentification de l'utilisateur (U_A) s'effectue avec les étapes suivantes :

1. le LTS de d_A authentifie U_A et délivre un jeton de sécurité (ST_A) signé avec le certificat de sécurité d_A (1.a - flèche pointillée dans la FIGURE 5.5) ;
2. d_A s'authentifie auprès du FTS à l'aide de ST_A et obtient pour le compte de U_A, un jeton de sécurité fédéré (ST_F) signé avec le certificat GACM (1.b) ;

3. le consommateur du service utilise ST_F pour obtenir un jeton de sécurité (ST_B) de d_B signé avec le certificat de sécurité d_B. La signature ST_F prouve que d_A et U_A appartiennent à la fédération et sont dignes de confiance (1.c).
4. enfin, S_B est appelé au nom de U_A avec ST_B (1.d).

Les attributs d'autorisation contenus dans le ST_B étant spécifiques à d_B, sont utilisés pour le contrôle d'accès de S_B.

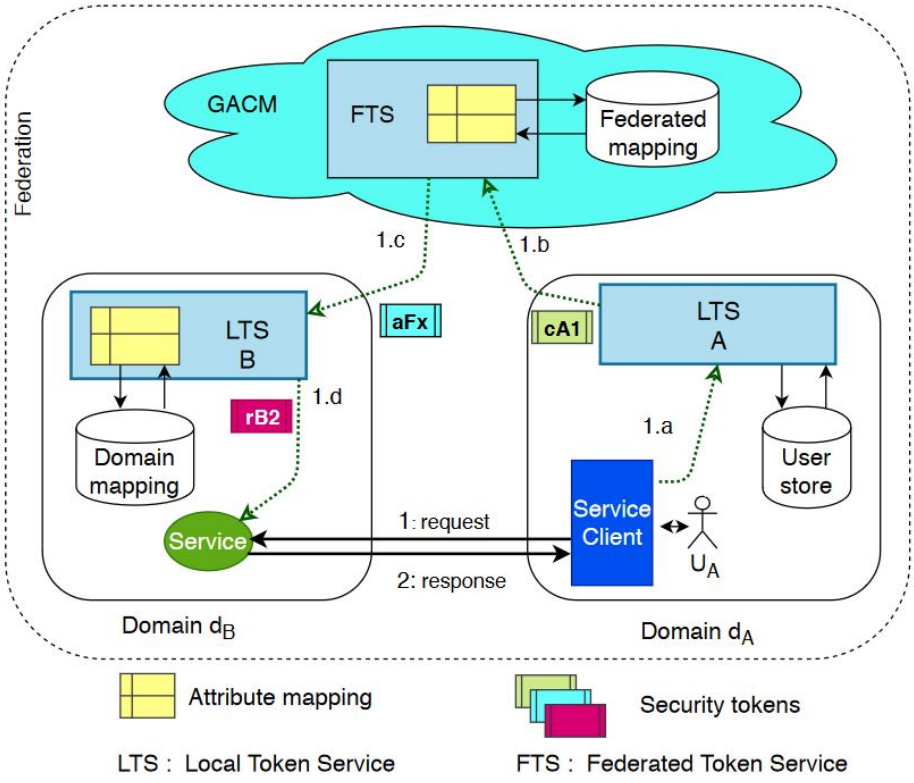


FIGURE 5.5 – Séquence d'authentification unique fédérée (source : [Bah+19])

Autorisation

Le jeton de sécurité utilisé pour appeler un service doit contenir les attributs d'autorisation du domaine fournissant ce service. L'utilisateur dispose dans un premier temps des attributs d'autorisation de son domaine qui doivent être successivement liés aux attributs fédérés et aux attributs d'autorisation du domaine cible lors du processus d'authentification. Dans la FIGURE 5.5, pour obtenir l'autorisation de U_A, l'attribut d'autorisation de U_A ($cA1$) est utilisé par le FTS pour calculer l'attribut fédéré aFx correspondant à $cA1$. L'aFx envoyé à d_B, permet au LTS de d_B de calculer l'attribut d'autorisation $rB2$ correspondant à aFx. Ce dernier permet enfin d'accéder au service ciblé par U_A.

5.2.6 Mise à disposition de services dans la fédération

Les services partagés par les domaines doivent être visibles en un seul endroit au niveau de la fédération pour faciliter leur découverte et leur composition au sein de la fédération. A ce niveau, le contrat de service, et notamment les exigences de contrôle

d'accès au service, doit être précisé par les attributs d'autorisation de la fédération. Nous appelons **promotion de service** la redéfinition des exigences de contrôle d'accès aux services avec les attributs fédérés. Elle doit être transparente pour les consommateurs des services partagés. Notre méthode consiste à définir un registre global de services au niveau du GACM dans lequel les domaines promeuvent les services qu'ils souhaitent partager à partir de leurs registres locaux. La fédération de services avec notre méthode comporte trois étapes : (i) initialisation de la fédération ; (ii) création d'un registre de services pour la fédération ; (iii) promotion de services.

i) Initialisation de la fédération

Supposons n domaines $d_i = \langle U_i, S_i, R_i, SP_i, SS_i \rangle$, $(i = 1 \cdots n)$ qui souhaitent collaborer en fédérant leurs services respectifs. Outre les accords de collaboration, la fédération $F = \langle U_f, S_f, R_f, SP_f, SS_f \rangle$ est créée par les étapes suivantes :

1. *Mise en place du médiateur de contrôle d'accès (GACM)* Le GACM (cf. section 5.2.4) représente l'autorité de la fédération en matière d'interopérabilité sécurisée entre les domaines. Cette étape consiste à choisir ou créer le domaine qui jouera le rôle du médiateur de contrôle d'accès dans la fédération. Le GACM comprend les services de sécurité pour le *mapping* entre les domaines. Initialement, nous définissons un service de sécurité dans le GACM nommé *service de jeton fédéré FTS* pour l'évaluation des politiques de *mapping* fédérés (PMF) et la gestion de confiance entre les domaines.
2. *Fédération des utilisateurs des domaines* En raison de l'hétérogénéité de leurs attributs d'autorisation, les domaines accèdent aux services des uns et des autres en utilisant les attributs fédérés. Les attributs fédérés servent à décrire les privilèges d'accès des utilisateurs de manière compréhensible et non ambiguë dans toute la fédération. En ce sens, les attributs fédérés fédèrent les attributs d'autorisation des domaines et par la même occasion les utilisateurs des domaines. La fédération des utilisateurs consiste à définir des *mappings* entre les attributs d'autorisation des domaines via notre méthode d'autorisation (*cf.* Section 5.2.5). Pour fédérer les utilisateurs des domaines, d'une part, les domaines négocient avec le GACM afin d'établir les correspondances entre leurs attributs d'autorisation et les attributs fédérés ; d'autre part, ils établissent localement des correspondances entre les attributs fédérés et leurs attributs d'autorisation. Les utilisateurs fédérés sont initialisés avec $\bigcup_{i=1}^{n}(U_i)$ qui représente l'ensemble des utilisateurs de tous les domaines. Cependant, l'appartenance des utilisateurs fédérés au GACM est implicite. Le GACM ne stocke pas les utilisateurs des domaines.

On note par $d_i \sqsubset F$ qu'un domaine d_i est membre de la fédération F. La fédération n'a pas encore de services, $S_f = \{\}$. Pour faciliter la découverte et l'utilisation de services partagés entre domaines, nous avons besoin d'un registre de services au niveau de la fédération : le *registre fédéré (de service)* est le registre R_f de la fédération F. Il est ajouté au GACM, et est vide à ce stade.

ii) Promotion des services des domaines

Pour promouvoir un service s dans la fédération F (*cf.* FIGURE 5.6), un contrat de service s_{fcont} est créé à partir de son contrat de service local $s_{cont} = \langle I, P_{R[AT]}, Edp \rangle$. Les exigences AC $R[AT]$ du contrat de service existant doivent être redéfinies à l'aide des attributs fédérés $af_j \in AF$ pour créer les exigences de contrôle d'accès du contrat de service fédéré.

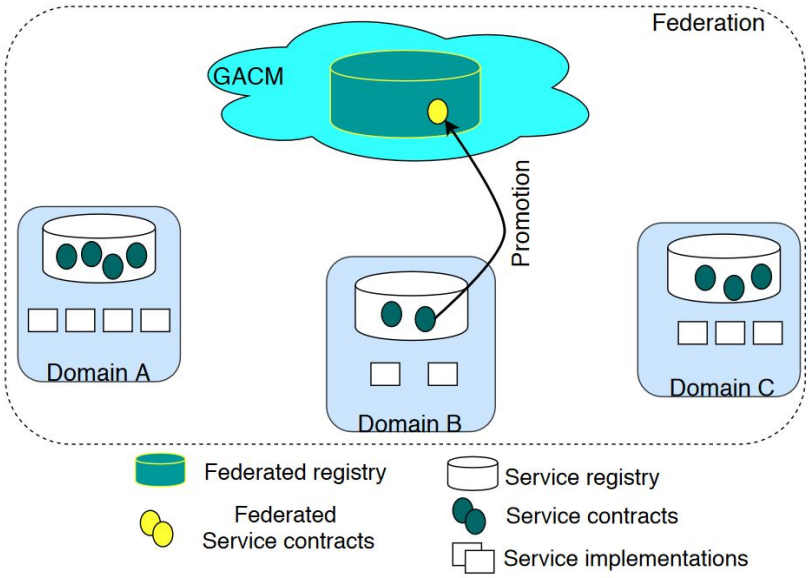


FIGURE 5.6 – Promotion du service au niveau de la fédération (source : [Bah+20b])

Considérant qu'une exigence AC est un *terme* t_r construit avec les attributs d'autorisation de domaine $at_i \in AT$, l'exigence AC pour la fédération aboutit à transformer les attributs de domaine at_i en t_r avec les attributs fédérés af_j. Cela se traduit par des exigences AC $R[AF]$ pour la politique de service fédérée $P_{R[X]}$. Le remplacement des attributs d'autorisation dans les termes est basé sur les politiques de *mapping* fédérés représentées par la fonction de *mapping* m entre les attributs d'autorisation locaux du domaine AT et les attributs fédérés AF. Ces *mappings* entre les attributs sont définis dans les domaines en tant que fonctions $m : AT \rightarrow AF$ soit un ensemble de couples $\{(at_i, af_j)\}$.

Chaque exigence de contrôle d'accès du nouveau contrat de service est obtenue en transformant chaque terme t_r trouvé dans le contrat de service existant avec les mappings m ; ce qui résulte en $R[AF]$ un ensemble de termes construits avec AF. Le service s possède alors deux contrats de service : un contrat local de service utilisable uniquement à l'intérieur du domaine d_i et un contrat fédéré du service utilisable à l'extérieur du domaine d_i. Le contrat fédéré du service s est $s_{fcont} = \langle I, P_{R[AF]}, Edp \rangle$.

Le contrat fédéré s_{fcont} du service est ensuite publié dans le registre fédéré du GACM afin que d'autres domaines puissent y accéder (cf. Fig. 5.6). Étant donné que le service et son contrat sont interchangeables, le contrat local s_{cont} et le contrat fédéré s_{fcont} du service s peuvent être considérés comme deux services distincts. Le service s à l'intérieur du domaine d_i devient le service s_f dans la fédération. Le service s est interopérable en

termes de sécurité uniquement à l'intérieur du domaine d_i. Alors que le service s_f est interopérable dans la fédération. Ainsi, la promotion de services permet d'accéder et de composer les services de différents domaines au delà de leurs frontières de sécurité.

Le registre fédéré contient les services s_f promus par les domaines de la fédération ($\exists d_i \sqsubset F \wedge s_f \in S_i$) ; ils sont appelés des *services fédérés* au lieu des *services promus* afin d'être alignés avec les attributs fédérés : $R_f = \{s_f, ...\}$. L'implémentation d'un service fédéré se trouve dans le domaine qui le fournit : les appels aux services fédérés sont gérés par les domaines contenant leurs implémentations.

Promouvoir un service $s_i = \langle I, P_{R[AT_i]}, Edp \rangle$ d'un domaine $d_i = \langle U_i, S_i, R_i, SP_i, SS_i \rangle$ où $SP_i = \langle AT_i, AR_i \rangle$ avec $AT_i = \{at_u\}\, u \in 1..q \wedge q \in \mathbb{N}$, $AR_i = \{(s_u, at_v)\}\, u, v \in \mathbb{N}$ dans la fédération $F = \langle U_f, S_f, R_f, SP_f, SS_f \rangle$, entraîne l'exécution des étapes suivantes :

Préliminaire : Définir une fonction de *mapping* m entre les attributs d'autorisation AT_i du domaine d_i et les attributs fédérés AF_j ; $m : AT_i \rightarrow AF_f$. Soit s_a est un ensemble d'attributs. La fonction $map(m, s_a)$ donne un ensemble d'attributs s'_a.

S 1 : Copier le contrat local $\langle I, P_{R[AT_i]}, Edp \rangle$ du service s_i à partir du registre de service R_i ;

S 2 : Isoler les exigences de contrôle d'accès $R[AT_i] = \{t_1, t_2, ...\}$ de la politique d'accès $P_{R[AT_i]}$; chaque exigence de contrôle d'accès spécifiée avec les attributs d'autorisation locaux $at_u \in AT_i$ représente un terme t_r ;

S 3 : Transformer $R[AT_i]$ en exigences de contrôle d'accès fédéré $R[AF_f]$ en appliquant la fonction de *mapping* m sur l'ensemble des termes $R[AT_i]$ pour changer les attributs $at_u \in AT_i$ par les attributs fédérés correspondants $af_j \in AF_f$: $map(m, R[AT_i])$;

S 4 : Créer le nouveau contrat fédéré du service $s_{fcont} = \langle I, P_{R[AT_f]}, Edp \rangle$ avec les exigences de contrôle d'accès fédéré $R[AT_f]$;

S 5 : Publier le contrat fédéré du service s_{fcont} dans le registre fédéré R_f.

La règle suivante définit formellement la promotion des services de domaine dans la fédération :

$$\frac{\begin{array}{c} F = \langle U_f, S_f, R_f, SP_f, SS_f \rangle \\ d_i = \langle U_i, S_i, R_i, SP_i, SS_i \rangle \qquad s_i = \langle I, P_{R[AT_i]}, Edp \rangle \\ d_i \sqsubset F \wedge s_i \in S_i \qquad R[AF_f] = map(m, R_i) \\ s_j = \langle I, P_{R[AF_f]}, Edp \rangle \end{array}}{F = \langle U_f, S_f, R_f \cup \{s_j\}, SP_f, SS_f \rangle} (promotion, d_i, s_i)$$

Après les étapes de promotion ci-dessus, le registre fédéré qui était vide au départ, contient maintenant un service fédéré s_f visible et accessible à tous les domaines de la fédération, $R_f = \{s_f\}$; de là, le processus est incrémental.

5.2.7 Mécanisme de contrôle d'accès fédéré

Le contrôle d'accès fédéré repose sur des politiques de contrôle d'accès qui définissent les permissions d'accès des utilisateurs aux services. Les mécanismes existants de contrôle d'accès fédéré des services [WS-F] ne prennent pas en charge l'évaluation des politiques de *mapping*. Nous proposons dans cette section un mécanisme de contrôle d'accès fédéré basé sur le *mapping*. Ce mécanisme permet d'évaluer les politiques de *mapping* définies entre les domaines lors de l'accès aux services.

Sémantique des appels de services dans la fédération

Les services sont fédérés pour faciliter leur découverte et leur utilisation dans la fédération. Le GACM sert d'interface entre les consommateurs de services et les fournisseurs de services. Les services fédérés sont considérés comme fournis par la fédération représentée par le GACM. Lorsque les services fédérés sont invoqués au sein de la fédération, il est important que l'accès soit aussi transparent que possible pour les utilisateurs de la fédération. Nous regroupons donc les services en catégories et les appels du service sont gérés en fonction de ces catégories.

Classification des services Tous les services d'un domaine ne sont pas partagés au niveau de la fédération. Nous distinguons deux catégories de services : (i) les *services locaux* et (ii) les *services fédérés*. Les services locaux ne sont publiés que dans le registre de services des domaines ; ils sont partagés uniquement à l'intérieur d'un domaine. Les exigences de contrôle d'accès d'un service local sont spécifiées avec les attributs d'autorisation de son domaine. Les services fédérés sont publiés dans le registre de services de la fédération. Les exigences de contrôle d'accès des services fédérés sont spécifiées avec les attributs fédérés, compréhensibles par tous les domaines. Cependant, il existe deux points de vue pour les services fédérés. Du point de vue des domaines (consommateurs de services), un service fédéré est fourni par la fédération. Du point de vue des domaines (fournisseurs de services), un service fédéré est un service local partagé au niveau de la fédération. La fédération de services facilite l'utilisation et la composition des services de différents domaines en termes de sécurité. De plus, la fédération de services est transparente pour leurs consommateurs car elle ne modifie pas les règles d'appel des services.

Règles d'appel des services Nous avons défini formellement les règles qui régissent les appels des deux catégories de services entre les domaines dans [Bah+20b]. Dans les règles sémantiques suivantes, la fonction **localToken**(s_i, u_i, s_j, ss_i) est utilisée pour obtenir un jeton de sécurité local st_i émis par un service de sécurité ss_i au nom d'un utilisateur u_i pour appeler un service s_j à partir d'un service s_i ; **domainToken**(s_i, tk_i, s_j, ss_j) est utilisés à partir d'un service s_i d'un domaine d_i pour demander un jeton de sécurité tk_j requis par un service s_j à un service de sécurité ss_j d'un domaine d_j de la fédération F au nom d'un utilisateur d'un domaine d_i authentifié par son jeton de sécurité tk_i.

A titre d'exemple, illustrons le cas des *Appels de services fédérés.* Lorsqu'un service s_j d'un domaine d_j est un service fédéré, alors le jeton de sécurité utilisé pour appeler s_j est obtenu du service de sécurité ss_f de la fédération.

$$\frac{\begin{array}{c} F = \langle U_f, S_f, R_f, SS_f, SP_f \rangle \\ d_i = \langle U_i, S_i, R_i, SS_i, SP_i \rangle \quad d_i \sqsubset F \\ d_j = \langle U_j, S_j, R_j, SS_j, SP_j \rangle \quad d_j \sqsubset F \\ u \in U_i \quad s_i \in S_i \quad s_j \in S_j \quad ss_i \in SS_i \\ s_j \in R_f \quad s_j = \langle I_j, P_{R[AT_j]}, Edp_j \rangle \\ tk_i = \mathsf{localToken}(s_i, u, s_j, ss_i) \\ ss_f \in SS_f \quad tk_f = \mathsf{domainToken}(s_i, tk_i, s_j, ss_f) \end{array}}{\mathsf{CallAttempt}(s_i, u, s_j) \rightsquigarrow \mathsf{SecureCall}(s_i, tk_f, Edp_j)} (interdomainFederatedServCall)$$

5.2.8 Composition de services fédérés

La composition des services doit prendre en compte la gestion de la confiance et le contrôle d'accès de chaque service fédéré qui appartiennent à différents domaines de la fédération. Le contrôle d'accès de la composition de service se fait au niveau du service composite et au niveau des services composés. Cela crée deux problèmes supplémentaires : (1) la spécification des exigences de contrôle d'accès du service composite et (2) l'authentification unique fédérée entre le consommateur du service composite (demandeur initial), le service composite et les services composés. Ces problèmes sont traités en considérant deux scénarios : (i) invocation des services composés au nom du service composite et (ii) invocation des services composés au nom du demandeur initial.

Dans le premier scénario, le contrôle d'accès du service composite s'effectue comme dans n'importe quel service fédéré. Les exigences de contrôle d'accès du service composite sont indépendantes de celles des services composés. Pour invoquer un service composé, le service composite suit les étapes d'authentification décrites à la page 154. La composition du service avec ce scénario est illustrée dans la figure 5.7. Nous supposons que les exigences du service composite sont l'union de celles des services composés.

Dans le deuxième scénario, les exigences de contrôle d'accès du service composite dépendent de celles des services composés qui peuvent être différentes d'un service à l'autre. Les services composés nécessitent un jeton de sécurité contenant les attributs d'autorisation de leurs domaines. Le consommateur du service composite doit fournir un jeton de sécurité qui satisfait à ces exigences.

1. Premièrement, le *token store* est introduit au niveau du service composite pour stocker le jeton de sécurité du demandeur initial (ST_{init}). Le service composite doit transmettre le ST_{init} pour appeler les services composés. Mais le ST_{init} contient les attributs d'autorisation du domaine qui fournit le service composite.
2. Deuxièmement un nouveau processus d'authentification est lancé en utilisant le demandeur ST_{init} afin d'avoir les attributs d'autorisation du domaine du service

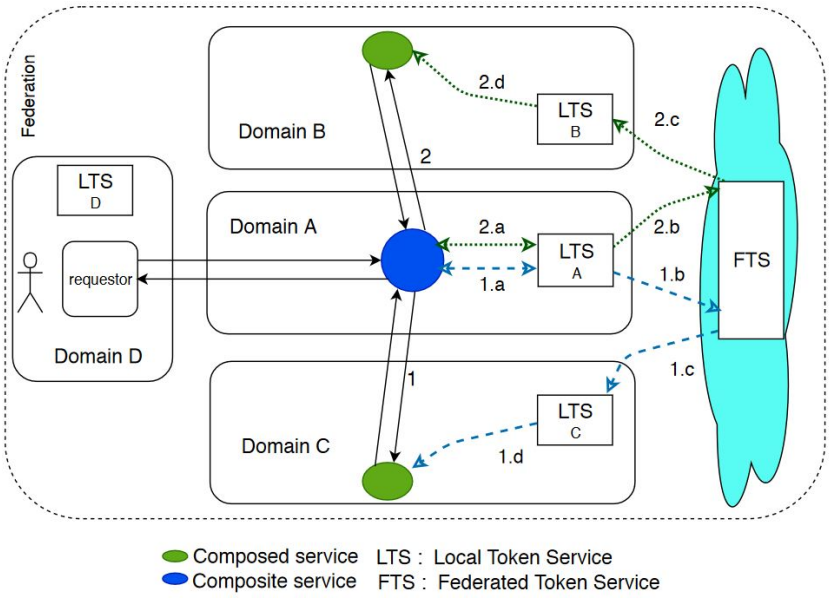


FIGURE 5.7 – Services composés invoqués au nom du service composite (source : [Bah+19])

composé correspondant à ceux contenus dans ST_{init}.

Le service composite doit transmettre le jeton de sécurité du demandeur initial (ST_{init}) pour appeler les services composés. Au niveau du service composite, ST_{init} contient les attributs d'autorisation du domaine qui fournit le service composite. Les services composés nécessitent un jeton de sécurité contenant les attributs d'autorisation de leurs domaines. Pour disposer de ces jetons de sécurité, un nouveau processus d'authentification doit être exécuté à l'aide du ST_{init} afin d'avoir les attributs d'autorisation correspondant à ceux contenus dans ST_{init}. La figure 5.8 illustre la composition du service avec ce scénario.

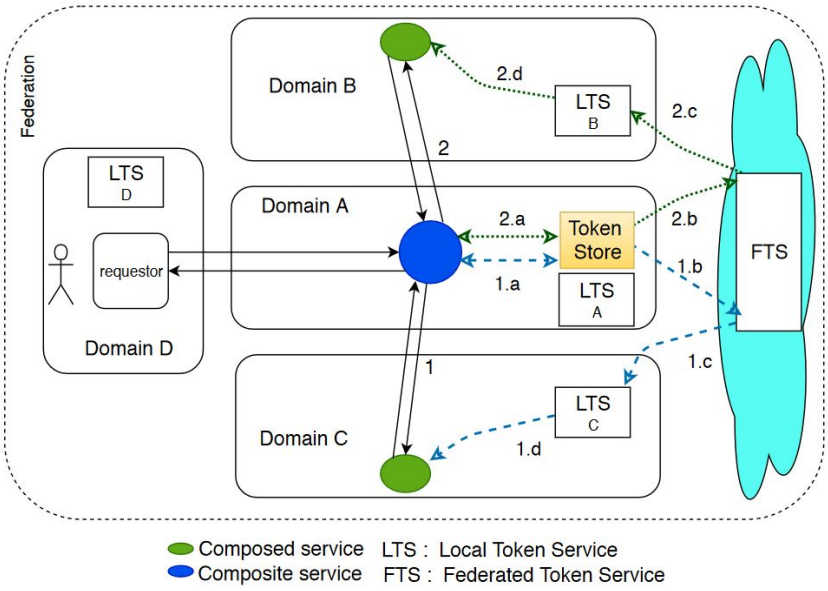


FIGURE 5.8 – Invocation de services composés au nom du demandeur initial (source : [Bah+19])

5.2.9 Mise en œuvre du contrôle d'accès fédéré

La mise en œuvre est basée sur la norme de fédération d'authentification *WS-Federation*. Le contrôle d'accès fédéré consiste à authentifier l'utilisateur dans son domaine (IdP) et à accompagner le message d'invocation du jeton d'authentification (*e.g.* assertion SAML) pour la décision d'accès dans le domaine du service (RP). Les politiques de contrôle d'accès des domaines (RP) sont définies sur leurs propres attributs locaux (*cf.* FIGURE 5.2). Les utilisateurs dans les domaines IdPs possèdent aussi des attributs locaux de ces domaines. Les attributs fédérés servent non seulement de moyens de correspondances entre les attributs des domaines IdP et RP mais aussi à déléguer l'authentification des utilisateurs aux IdPs. Pour ce faire, les politiques de *mapping* établies entre les domaines doivent être évaluées lors de l'accès aux services afin de convertir les attributs locaux en attributs fédérés et inversement. Afin de découpler le contrôle d'accès inter-domaine du contrôle d'accès intra-domaine, nous avons choisi d'évaluer les politiques de *mapping* PMF et PDM lors de la phase d'authentification.

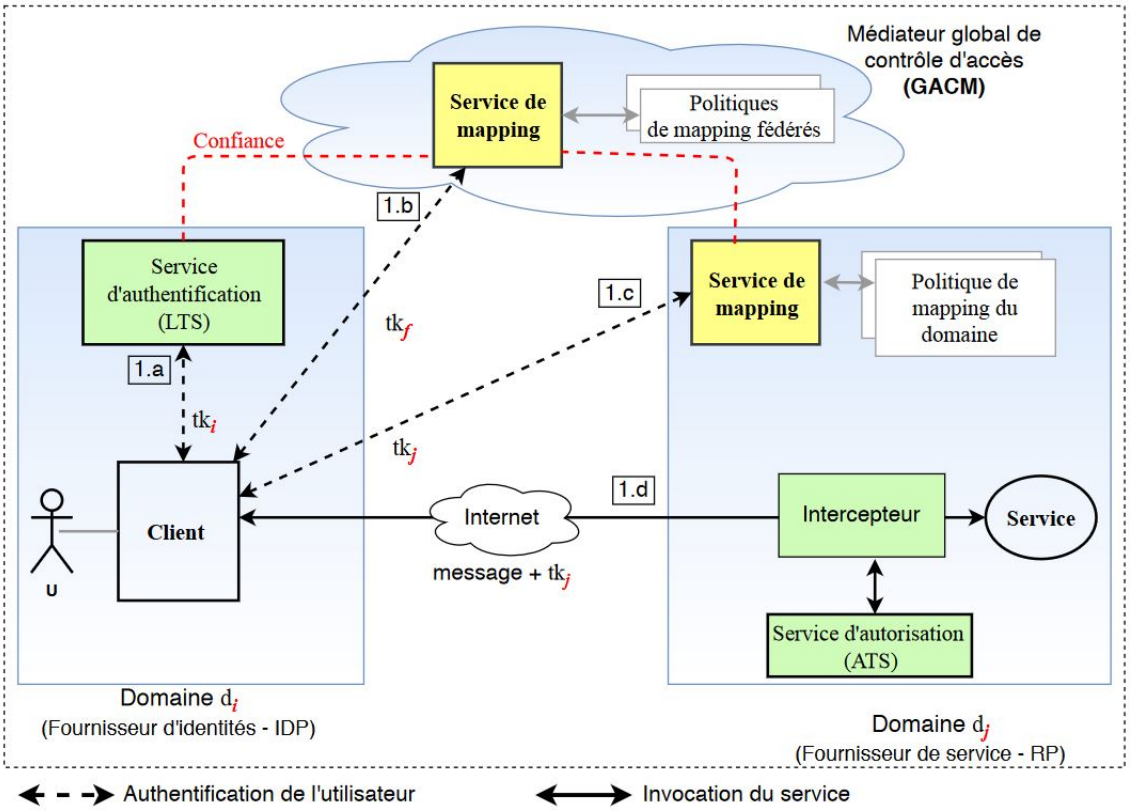


FIGURE 5.9 – Processus de contrôle d'accès fédéré (source : [Bah20], p. 95)

La FIGURE 5.2 illustrait le processus de contrôle d'accès dans un domaine, nous l'étendons au contrôle d'accès fédéré dans la FIGURE 5.9. Lorsqu'un utilisateur U d'un domaine d_i accède à un service s d'un domaine d_j, les étapes suivantes sont exécutés :

1. l'utilisateur U est authentifié localement auprès du service d'authentification de d_i qui délivre un jeton de sécurité local tk_i contenant les attributs locaux de l'utilisateur (1.a).

2. Avec le jeton tk_i, l'utilisateur est authentifié auprès du service de *mapping* du GACM qui délivre à partir de tk_i un jeton fédéré tk_f contenant les attributs fédérés de l'utilisateur. Ces attributs fédérés correspondent aux attributs locaux contenus dans tk_i (1.b).
3. Avec le jeton tk_f, le client de service demande un jeton local tk_j auprès du service de *mapping* de d_j. Les attributs locaux contenus dans tk_j sont obtenus à partir des attributs fédérés de tk_f. La signature numérique de tk_j prouve que U appartient à la fédération (1.c).
4. Finalement, le service s est appelé au nom de l'utilisateur U avec le jeton tk_j.

Les attributs d'autorisation contenus dans le tk_j étant spécifiques à d_j, ils sont utilisés pour évaluer la décision d'accès au service s.

L'évaluation des politiques de *mapping* est déléguée à un service de sécurité dédié afin de : (i) découpler le mécanisme de *mapping* d'attributs du mécanisme de contrôle d'accès des domaines (IdPs et RPs) et (ii) assurer la sécurité des *mapping* fédérés. Les services de *mapping* facilitent la maintenance des mécanismes de contrôle d'accès des domaines. Un service de *mapping* est un service de sécurité qui fournit la fonction de conversion d'attributs. Ce service est ajouté au contrôle d'accès d'un domaine lors de son adhésion à la fédération et est retiré lorsque le domaine quitte la fédération. Le service de *mapping* n'intervient que lors de l'accès aux services partagés avec la fédération.

5.2.10 Bilan

Les contributions de ces travaux sur le contrôle d'accès fédéré des services couvrent trois aspects : l'autorisation des utilisateurs, la délégation de l'authentification des utilisateurs à leurs domaines et le contrôle des permissions d' accès. Concernant l'autorisation des utilisateurs, nous avons proposé une méthode d'autorisation basée sur la technique de *mapping* d'attributs afin d'assurer l'interopérabilité des modèles et des attributs contrôle d'accès des domaines. Le mécanisme de contrôle d'accès fédéré permet d'assurer le couplage faible entre les domaines en termes de contrôle d'accès. Enfin, la méthode de fédération de services assure l'interopérabilité des services en termes de contrôle d'accès *i.e.* visibilité et accessibilité des services à différents niveaux de sécurité. Ces contributions ont fait l'objet de deux publications ([Bah+19] et [Bah+20b]).

Une mise en œuvre de la proposition par les technologies des web services est proposée dans [Bah20] avec les technologies de services Web telles que SOAP, WSDL et les normes de sécurité associées. Le contrôle d'accès des services Web s'appuie sur la composition de plusieurs normes de sécurité telles que WS-Security, WS-Trust, WS-Federation, SAML, XACML [AV16 ; SWS07]. La dépendance entre couches (et normes) est difficile à gérer car les solutions existantes de sécurité des services Web ne sont pas intégrées et n'implémentent, de manière cohérente, que certaines couches à la fois. C'est ce qui a rendu difficile l'implémentation de la solution, qui s'appuie principalement sur *Glassfish Metro RI*[1] du

1. Pile de services Web Metro, `https://javaee.github.io/metro/`

serveur d'application *open source* `Glassfish`[2]. L'implantation prend en compte les *mapping* et la conversion des attributs contenus dans les jetons de sécurité, le mécanisme de promotion de service ainsi que celui de fédération de services.

5.3 Sécurité des applications, aider les développeurs

Cette section vise à présenter les travaux menés dans le cadre de la thèse de Mohammed El Amin TEBIB, démarrée en octobre 2020 en co-tutelle avec l'Université Grenoble Alpes et co-encadrée avec Oum El Kheir AKTOUF, et Mariem Graa (rejoint l'encadrement en 2021). La thèse vise l'assistance au développement d'applications mobiles sécurisées.

| La présentation est succincte du fait que la thèse est en cours.

Malgré divers travaux de recherches menés pendant la dernière décennie sur la sécurité des applications Android, les problèmes de sécurité persistent. En 2020 et début 2021 les statistiques restent très significatives en termes de nombre de failles de sécurité liés à ce type d'applications (1148 failles, dont 791 liées aux privilèges) selon le CVE[3]. Une autre étude expérimentale récente [Sco+19] effectuée sur des applications Android open source (574 référentiels github) a montré que les problèmes liés aux autorisations sont toujours un phénomène fréquent dans les applications Android. Ce qui conduit a des fuites de sécurité très connues comme l'escalade de privilèges et l'exploitation des données privées. Les autorisations sur Android sont manipulées par les développeurs à travers le concept de permissions. et sont accordés par l'utilisateur pendant l'exécution de l'application. Une permission sera définie dans le code d'application (dans un fichier de configuration qui s'appelle manifest.xml) pour donner l'accès à une application d'utiliser une ressource exposée par le système, ou alors un composant d'application à interagir avec un autre composant. Pour manipuler ces permissions (connaître leur signification, comment les utiliser et pour quels objectifs), Google à mis à disposition des développeurs une documentation officielle expliquant leur utilisation[4]. Cependant, en raison des changements continus du nombre et des spécifications des autorisations, cette documentation devient rapidement obsolète et difficilement lisible pour les développeurs (qui sont souvent des développeurs tiers - c'est-à-dire non spécialistes de la sécurité), ce qui entraîne différents problèmes et risques de sécurité.

Dans [Teb+21a], nous avons étudié le problème des permissions d'accès, clé de voute des droits d'accès dans le système Android. Les attaques liées à ces permissions constituent un problème de sécurité fréquent dans l'environnement Android. En raison d'une mauvaise utilisation des privilèges, les attaquants usurpent les droits de l'application et exécutent actions malveillantes. La plupart des solutions de défense existantes sont possibles du point de vue des utilisateurs finaux, *e.g.* utiliser un anti-virus et définir finement

2. Pile de services Web Metro, `https://javaee.github.io/glassfish/`
3. `https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224`
4. `http://developer.android.com/sdk/apidiff/30/change`

les configurations du système. Mais pour cela il faut qu'il soit averti non pas des risques mais de comment s'en prémunir, ce n'est pas le cas pour les millions d'utilisateurs, qui acceptent des requêtes par ignorance ou simplement pour avancer dans l'utilisation. Il en est souvent de même pour les développeurs, en effet dans les formations, la sécurité reste un parent pauvre, par rapport aux autres besoins, notamment fonctionnels ou d'efficacité. Nous prenons le point de vue des développeurs car la sécurité devrait être un souci de conception du logiciel. Dans cette approche, nous proposons un modèle abstrait des applications et proposons un ensemble de règles pour vérifier les propriétés de sécurité. Nous proposons une vérification formelle dont une limite évidente est la validité, il faut qu'elle corresponde à l'application. Nous proposons un outil appelé `PrivDroid` [5]. Cet outil effectue une rétro-ingénierie de modèle Ecore (voir le chapitre 4) à partir des applications. Les règles sont alors implantées sous forme de transformations de modèles ATL. Ainsi implantée, la vérification peut être formelle pour une partie des règles mais ne suffit pas, une évaluation dynamique est nécessaire pour éviter des règles trop strictes.

Dans cette thèse, l'objectif est d'aider les développeurs à détecter de les comportements indésirables liés aux autorisations d'accès qui compromettent le l'intégrité des applications et la confidentialité de leurs données. Le périmètre d'exploration de la thèse a été précisé dans une étude systématique [Teb+22 ; Teb+23a] qui met en évidence les types de vulnérabilités, les techniques de vérification et leur positionnement dans le cycle de développement sécurisé du logiciel (*Secure Software Development Life-cycle*). L'étude comparative montre la limite des outils actuels et les besoins. Une première réponse est donnée dans [Teb+23b] qui présente une version de l'outil `PrivDroid` qui intègre des *code smells* pour détecter des failles de sécurité de type *Privilege Escalation*. L'outil est fonctionnel et disponible sur GitHub [6] mais bien sûr il n'est pas complet, le travail continue. La thèse sera soutenue en décembre 2024.

5.4 Conclusion

Je ne suis pas un spécialiste de la sécurité informatique mais reste sensible à cette exigence de tout système informatique, que j'aborde du point de vue des modèles et de leur vérification, à savoir un ensemble de propriétés à vérifier.

Les deux travaux présentés rapidement ici ont été menés dans le cadre de deux thèses er restent des contributions limités au vu du spectre que couvre la sécurité informatique. La première thèse touche au domaine de la sécurité des réseaux et communications qui est très complexe à cause de la variété et l'hétérogénéité des normes. Elle propose des mécanismes à intégrer dans les fédérations de domaines de sécurité. La second touche directement au génie logiciel pour aider les développeurs. Les travaux sur les applications mobiles

5. En 2021, l'outil est appelé `PermDroid` dans [Teb+21a]. Nous nous sommes rendus compte plus tard qu'un outil homonyme était apparu en 2022 [YZS22], nous avons contacté les auteurs et nous avons décidé de renommer l'outil en `PrivDroid` pour éviter des confusions, même si l'antériorité de publication était nôtre.

6. `https://github.com/tebmed/privdroid`

prennent tout leur sens compte-tenu du côté envahissant (ou pervasif) de ces applications pour des millions de gens. La difficulté majeure est liée à l'évolution permanente des logiciels systèmes et applicatifs qui font qu'à la fois des vuilnérabilités sont corrigées et de nouvelles apparaissent en permanence.

Ces travaux montrent la pertinence d'une approche modèle pour vérifier la sécurité des applications. Les approches proposées restent pragmatiques et répondent au besoin de techniques et outils pour aider au développement des applications, c'est bien du génie logiciel.

Post-scriptum

Ce chapitre a mis en évidence des acquis sur les compétences suivantes, que nous reverrons dans le chapitre 8.

C2

Identifier les problèmes, proposer des pistes, organiser les projets et fédérer les énergies sont autant de compétences à maîtriser pour mener des travaux de recherche.

C3

Organiser une recherche collaborative, détecter des compétences, définir des complémentarités et coordonner pour contribuer à un travail de recherche d'envergure.

C5

Investir de nouveaux domaines, collaborer avec des chercheurs d'autres spécialité implique de se comprendre et se faire comprendre, les apports croisés permettent d'innover dans les solutions des problèmes identifiés.

Troisième partie

Modèles en contexte pluridisciplinaire

Breakthrough innovation occurs when we bring down boundaries and encourage disciplines to learn from each other.

Talent Economics : The Fine Line Between Winning and Losing the Global War for Talent
Gyan Nagpal

Dans cette partie III, le génie logiciel (comme représentant de l'informatique) n'est pas unique, d'autres disciplines sont concernées. Dans cette partie, nous explorons des travaux pluridisciplinaires.

Nous aborderons l'interdisciplinarité avec la gestion et les systèmes d'information dans le chapitre 6 et avec la gestion de production et la cyber-physique dans le chapitre 7. Nous terminerons cette introduction par quelques expériences pluridisciplinaires avec les Sciences humaines et sociales (SHS), notamment le projet ONECAD avec des géographes, des gestionnaires et des collectivités territoriales.

Contribution à la géographie et sociétés

Les travaux relatés ici se placent dans le cadre l'évaluation de la capacité d'accueil, une préoccupation des élus dans la gestion de leur territoire, qui est devenue une obligation depuis puisqu'elle est est inscrite dans la Loi Littoral (art. L146-2 du code de l'Urbanisme [7]) depuis 2000. *"Évaluer la capacité d'accueil et de développement consiste alors à mesurer si l'accueil d'habitants, de touristes et d'activités supplémentaires envisagé par la collectivité locale est compatible avec le capital de ressources dont elle dispose et les objectifs qu'elle porte pour son territoire littoral. Il ne s'agit pas de déterminer mécaniquement la capacité d'accueil à partir des caractéristiques du territoire, mais d'une évaluation participative et partenariale"* [Cha+09]. La question est intrinsèquement pluridisciplinaire et elle a été traitée scientifiquement avec des géographes, des gestionnaires, des juristes mais aussi des collectivités territoriales (politique) et ses services de l'état (administration). La réflexion a été lancée en 2006, sur les territoires littoraux, et a aboutit à une méthode d'évaluation en 2009 (Cahier n°2 [Pot+09]) et un guide pratique en 2010 (Cahier n°3 [Cha+10]) Nous sommes arrivés en 2010 pour la seconde phase des travaux qui vise à mettre en place un outil numérique pour mettre en œuvre la méthode.

L'application ONECAD constitue l'aboutissement de plusieurs années de travail. La mise en place d'une méthode d'évaluation de la CAD a en effet nécessité 7 années d'échanges entre plusieurs laboratoires de l'Université de Nantes, les services de l'État (DREAL [8], DDTM 44 [9], 85, 56, 64), des bureaux d'études, des élus,... Aux termes de ces années, les retours d'expériences ont montré la nécessité de réaliser un outil numérique

7. `https://www.legifrance.gouv.fr/codes/article_lc/LEGIARTI000006814912`
8. Direction régionale de l'environnement, de l'aménagement et du logement
9. Direction Départementale des Territoires et de la Mer de Loire-Atlantique

au service de cette méthode, pour faciliter son utilisation et favoriser sa diffusion auprès des acteurs de la gestion des territoires, qu'ils soient littoraux ou non.

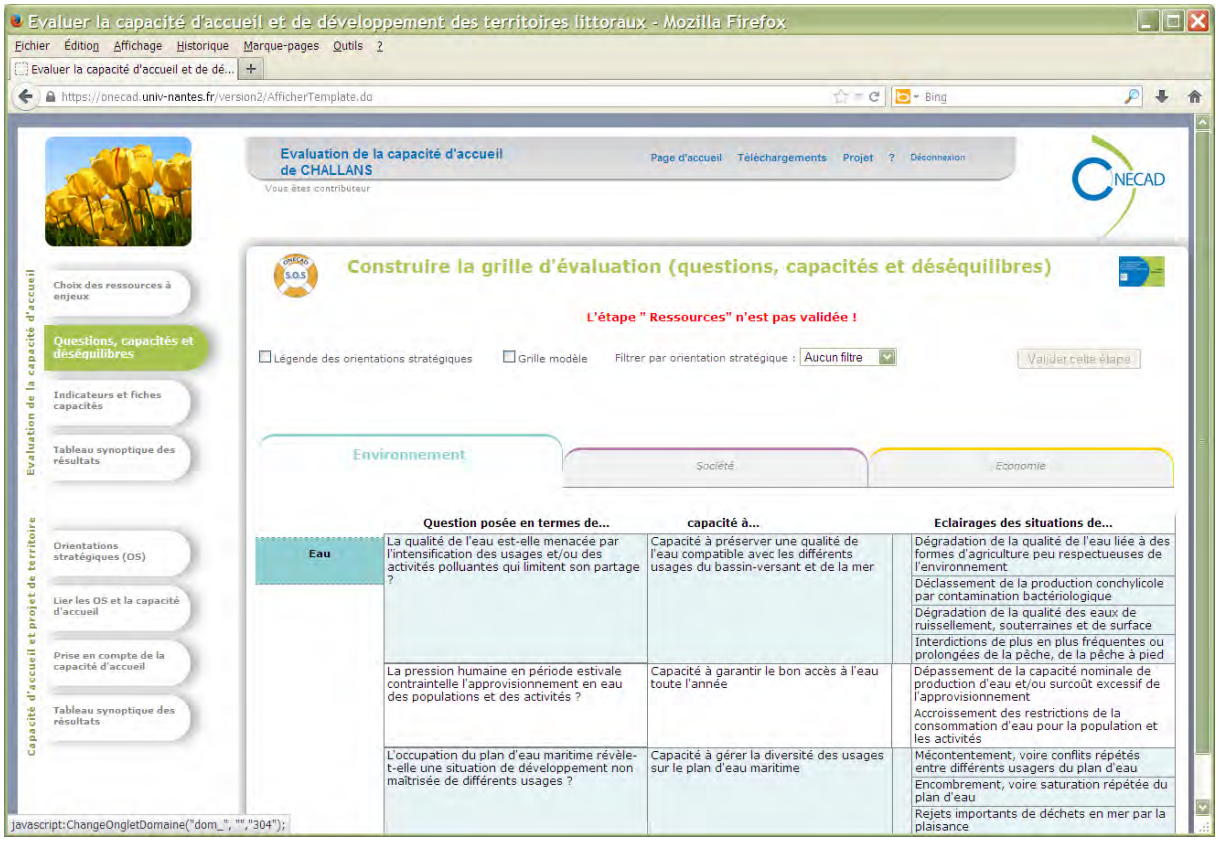
ONECAD est destiné aux praticiens des territoires en cours d'élaboration de documents de planification ou d'aménagement (SCoT [10], PLU [11], DTA [12], SMVM [13]...), bureaux d'études, élus, services de l'État. L'application s'accompagne d'un guide de l'utilisateur [Cha+09] [14], qui vise à accompagner et faciliter l'utilisation de l'application. En complément des nombreuses aides disponibles dans l'application, ce guide permet d'éclairer sur le plan pratique et théorique l'ensemble du déroulé de l'évaluation au fil des avancées de l'utilisateur. Il facilite également le lien entre l'outil numérique et la méthode d'évaluation elle-même.

L'outil numérique ONECAD a un autre objectif : celui du partage des connaissances. Présenté sous forme d'une plate-forme numérique en ligne, il offre une aide individualisée, territoire par territoire, grâce à un identifiant et un mot de passe personnel. Mais elle propose aussi, à ceux qui le souhaitent, de mettre à la disposition de chaque praticien les évaluations réalisées par d'autres territoires en garantissant l'anonymat des dépôts. Chacun peut décider de déposer ce qu'il souhaite. Évaluation après évaluation, la plate-forme peut ainsi s'enrichir, constituer un levier pour optimiser le temps de travail et devenir un lieu de partage des expériences menées par les différents acteurs. Ces conditions sont indispensables pour assurer une gestion pérenne et concertée des territoires sous forte pression humaine.

Quatre versions du prototype ont été développées, livrées et mise en ligne sur un serveur de l'université de Nantes [15]. Quatre campagnes de test ont été programmées. Les tests techniques ont été menées en partenariat avec le PIAC, le Centre de prestations et d'ingénierie Informatiques du MEDDTL [16]. Les recettes ont été menées avec des utilisateurs couvrant le panel des intervenants (collectivités, bureaux d'étude, services de l'état). Une version finale a été développée (*cf.* FIGURE III), elle devait être hébergée sur le site du ministère de l'écologie et de l'environnement, partenaire du projet, qui préconise un socle technique (partie droite du "Y" de la FIGURE 1.1). L'application était conforme aux exigences et de construction et de sécurité. Au final, suite à des changements de direction, et donc de stratégie au ministère, l'application n'est pas mise à disposition des collectivités par le ministère. Par contre, un transfert a été lancé avec la société Operis [17], qui propose des solutions numériques aux collectivités.

Le projet a été financé sur différents fonds. Le gros du développement s'appuie un

10. Schéma de Cohérence Territoriale
11. Plan Local d'Urbanisme
12. Directives territoriales d'aménagement
13. Schéma de mise en valeur de la mer
14. `https://www.researchgate.net/publication/329104327_Capacite_d'accueil_des_territoires_littoraux_guide_de_l'utilisateur_ONECAD/references`
15. `https://onecad.univ-nantes.fr/`
16. Ministère de l'Écologie, du Développement Durable des Transports et du Logement, [nom de l'époque]
17. `https://web.operis.fr/`



III.1 Ecran de l'application ONECAD 2.0

financement obtenu en réponse à un appel à projets de la Fondation de France « Quels littoraux pour demain ? » [18] sur 3 ans, avec un financement pour un ingénieur de développement. Outre le pilotage du développement, j'ai pu appliquer l'idée d'un développement agile qui s'adapte à la fois à des évolutions métiers et des contraintes techniques fluctuantes. On peut finalement dire qu'il s'agit de mettre en place un processus métier avec des technologies Web. Ce qui nous amène au problème d'alignement traité dans le chapitre 6. Divers projets parallèles ont été menés pour des études complémentaires, comme l'interface et le parcours utilisateur (*User eXperience - UX*) avec des étudiants en communication ou l'outil ONECAD-Modl, mené avec des étudiants de Miage, qui est destiné aux experts d'évaluation pour proposer des grilles initiales de questionnement. On rejoint ici la thématique de **l'ingénierie des modèles**. Le rapport final du projet est disponible en ligne [19]. Le travail a été présenté en conférence [CPA14] et fait l'objet d'un chapitre d'ouvrage [CPA16].

Contributions GL au droit

Je me suis intéressé au droit dans deux collaborations. La première avec Henri Habrias pour démontrer le bien-fondé des méthodes formelles dans l'écriture des textes de lois pour les rendre cohérents. Nous avons travaillé sur le cas des constitutions de noms de familles

18. `https://www.fondationdefrance.org/images/pdf/2023_AAP_FDF/Projets_Littoral_FdF_2011-2022.pdf`
19. `https://www.researchgate.net/publication/313853122_Outil_numerique_d'evaluation_de_la_capacite_d'accueil_et_de_developpement_d'un_territoire_sous_pression`

pour en faire une ontologie formelle.

Dans [AH06], nous avons utilisé le formalisme de la théorie des ensembles comme base pour spécifier l'attribution de noms. A première vue, l'utilisation d'un langage formel nous obligeait à être précis et à préciser profondément les énoncés de l'ontologie. L'écriture d'assertions formelles nous permet de détecter les problèmes ontologiques habituels (*e.g.* les conflits de confusion, de passage à l'échelle et de dénomination), les articles de loi redondants et ambigus (*e.g.* noms de base des parents, adoption) ou les éléments manquants (*e.g.* la combinaison de quatre noms, les noms passés et présents). Vérifier les types et les assertions améliore la confiance dans l'ontologie, nous avons trouvé plusieurs incohérences avec l'Atelier-B. De plus, utiliser le raffinement pour prouver une version exécutable de l'ontologie améliore le processus de validation. Lors de la spécification, nous avons également essayé de modéliser l'essence de l'ontologie et pas seulement le résultat final (visible) (*e.g.* en modélisant la parentalité, le mariage plutôt que les noms eux-mêmes). Dans [AH06] mais aussi dans [HA04], nous explorons l'incrémentalité et le phasage dans un cycle de vie, par exemple ici les règles de noms évoluent selon les événements, enfants, mariage, divorce...

Parmi les applications informatiques que nous rencontrons dans notre vie quotidienne, certaines présentent une caractéristique spécifique à l'interface entre les domaines informatique et juridique : le coût du dysfonctionnement est à la charge de l'usager car celui-ci ne peut prouver de la réalité de ce dysfonctionnement. Cette absence de preuve matérielle est une conséquence de la dématérialisation des traces de fonctionnement. Par ailleurs les traces logicielles sont détenues par l'organisme qui détient ou gère l'application informatique. Une telle application est nommée "Système Numérique Inéquitable (SNI)" [ED19] car le risque est supporté par l'utilisateur (partie juridiquement faible) et non par l'organisme gestionnaire (partie juridiquement forte). Dans de tels systèmes, l'usager n'est pas toujours en mesure de prouver qu'il a raison lors d'une remise en cause. Les exemples sont nombreux, usage non frauduleux d'un service de transport (bicloo, tram, bus...), remise de devoirs sur un service pédagogique (Madoc), dépôt d'un dossier sur une plateforme (concours, impôts...). Par exemple un usager d'un vélo de la ville de Nantes (Bicloo) qui rend le vélo qu'il a emprunté ne dispose pas de preuve de cette restitution (pas de ticket qu'il pourrait produire). En cas de défaillance de l'application informatique et de disparition du vélo, il sera tenu pour responsable et perdra la caution. Avec Chantal Enguehard, nous avons travaillé sur l'apport des méthodes formelles pour les problèmes de certification dans ce type de systèmes. Nous avons travaillé avec un groupe de Master sur ces systèmes appelés alors *Système à Risques Inéquitable (SRI)* dans un projet TER en 2013 qui a donné lieu à la création d'un site web [20]. Nous nous sommes en particulier intéressés au vote électronique qui ne fournit pas les mêmes garanties (propriétés) qu'un vote matérialisé. Nous avons soumis différents projets.

— Un projet AAP interdisciplinaire "Analyse des Systèmes Numériques Inéquitables

20. `https://chasseurdesri.wordpress.com/`

(SNI)" en 2014 a été déposé auprès de l'université avec des chercheurs de Droit et changement social (UMR 6297) et du Centre Nantais de sociologie (EA 3260).

— Le projet ANR DIVA (DIspositifs de Vote Automatisé) en 2012 (AAP blanc version 30 pages), associant politistes, juristes et informaticiens, propose une analyse pluridisciplinaire et comparée des dispositifs de vote automatisé. Il repose sur l'hypothèse selon laquelle les modalités pratiques du vote ne constituent pas un simple voile, sans effet sur les pratiques et les procédures électorales, mais qu'elles participent au contraire à façonner les manières de faire et de penser les élections. Les partenaires sont l'UMR Triangle (porteur du projet) avec des politistes de Lyon (Lyon 2, IEP, ENS) et Saint-Etienne (Université Jean Monnet), l'Institut Louis Favoreu-GERCJ, composante de l'UMR 6201 DPCDIDE, qui vient apporter utilement son expérience sur les questions électorales et le LINA [21] (avec un membre de l'ESIEA de Laval pour la partie sécurité et un membre du LSV Cachan pour la cryptographie).

— Le projet M4E (Modalités Électorales : Évaluation des Évolutions Électroniques) en 2013 (AAP générique version 6 pages), a pour objectif de produire une étude sur le vote électronique en France fondée sur l'observation des élections et de leur encadrement juridique. Les partenaires sont le LINA (porteur du projet), le Centre de recherches et d'études sur les droits fondamentaux (CREDOF) en droit public à l'Université de Paris Ouest-Nanterre, le département de sciences politiques de l'Université de Nanterre, le Groupe d'Etudes et de Recherche Interdisciplinaire en Information et Communication GERiiCO de l'Université de Lille 3 et un expert-Conseil en sécurité juridique et technique des NTICs.

Ces projets n'ont pas été sélectionnés pour financement.

Contributions GL à la patrimonialisation

A la Faculté des Langues et Cultures Étrangères de nantes (FLCE), des collaborations ont été lancées sur deux sujets : la transition numérique et la patrimonalisation.

Sous l'impulsion de Joël Massol, professeur d'Allemand, responsable de la recherche en LEA, une action a été lancée en mai 2014 sur l'entreprise à l'ère numérique et suivie d'une inscription dans un projet RFI régional. Nous avons démarré un groupe de travail pluridisciplinaire sur "La transition numérique des PME-PMI : une étude comparative et pluridisciplinaire". L'objet d'étude était : la transition numérique des PME-PMI dans différents pays (Allemagne, France, Italie, Espagne, Royaume-Uni etc.). L'objectif était de produire une étude comparative et pluridisciplinaire de cette transition numérique dans différents pays et d'en tirer les premiers enseignements des différents cas nationaux. La thématique de recherche se rattachait principalement à deux champs de recherche : celui de l'économie numérique et celui de l'évaluation des politiques publiques. J'ai mené une étude sur la transition numérique en France, notamment avec les plans étatiques,

21. Le LS2N résulte essentiellement de la fusion des ex-UMR IRCCyN (Institut de Recherche en Communications et Cybernétique de Nantes) et LINA (Laboratoire d'Informatique de Nantes Atlantique).

dont l'objectif est la transformation numérique de l'économie française avec un soutien particulier pour les TPE/PME. J'étais aussi en charge des outils collaboratifs du projet. L'initiative s'est éteinte fin 2016 par manque de contributions. Néanmoins, j'ai pu reprendre ma contribution (une synthèse du domaine et des références bibliographiques et webographiques) pour alimenter l'état de l'art dans le projet Orange EDF que je mentionne dans la Section 8.3.3 du chapitre 8.

J'ai aussi collaboré avec Géraldine Galeote, professeur d'Espagnol, sur deux propositions de projets pluridisciplinaires ANR dont Géraldine était responsable scientifique.

— En 2019, le projet PRC HISOC *Heritage and Identities in Societies in Conflict (XIXth-XXIth centuries)* propose d'interroger, grâce à une approche transdisciplinaire et transnationale, les processus de patrimonialisation dans une configuration sociétale très peu étudiée, les sociétés multiculturelles en conflit (entendu dans le sens d'antagonismes). L'innovation du projet ne réside pas seulement dans la problématique qu'il pose mais également dans la méthode spécifique qui va devoir être élaborée pour que les spécialistes de diverses disciplines puissent faire converger leurs recherches vers un objectif commun. Le consortium comprenait 22 chercheurs d'institutions françaises et 7 chercheurs étrangers. La complémentarité des membres se situe au niveau des domaines de spécialité (histoire, anthropologie, droit, science politique, économie, marketing, linguistique, littérature) mais aussi des aires géographiques étudiées (Espagne, Grande-Bretagne, Allemagne et France). Le caractère transdisciplinaire et transnational du projet ainsi que la multidimensionnalité des processus de patrimonialisation et du concept de patrimoine nécessitent la formation d'un consortium étoffé. Le numérique est perçu comme un outil essentiel dans la collaboration, d'un point de vue utilitaire avec des bases de données, mais aussi sur le potentiel des modèles pour les inférences et déductions par exemple.
— En 2020, le projet HEMEN *Emotions patrimoniales dans les nations européennes - Heritage Emotions in European Nations* est une révision du projet HISOC dans laquelle, on se propose d'analyser les émotions patrimoniales en tant que moteur des processus d'identification nationale, dans les sociétés européennes, en convoquant un dialogue entre le centre et la périphérie. Le consortium est structuré autour d'une collaboration transdisciplinaire de 5 laboratoires nantais [le CRINI (Centre de Recherches sur la Nation, les Identités et l'Interculturalité), le CHRIA (Centre de Recherche en Histoire Internationale et Atlantique), le LEMNA (Laboratoire d'Economie et de Management de Loire-Atlantique), le LPPL (Laboratoire de Psychologie des Pays de Loire) et le LS2N (Laboratoire des Sciences du Numérique de Nantes)] et 5 autres laboratoires français (Brest, Bordeaux, Perpignan, Paris/Sorbonne et Strasbourg) et des chercheurs étrangers. Pour la partie numérique, l'analyse des données vient compléter le côté utilitaire du stockage structuré du projet HISOC.

La qualité de ces projets a été reconnue mais ils n'ont pas été sélectionnés pour financement, limitant la progression dans ce domaine.

Chapitre 6

Aligner le logiciel aux processus métiers de l'entreprise

Le secret du bonheur, c'est l'alignement entre ce que vous pensez, ce que vous dites et ce que vous faites.

Mahatma Gandhi

Pour être efficace, que ce soit individuellement ou dans une organisation, la cohérence est indispensable entre la réflexion, la décision et les actions qui en découlent. Nous avons vu dans la Section 1.2.3 du Chapitre 6 que dans une entreprise, le système d'information est l'intermédiaire entre les décisionnaires, les opérationnels et l'extérieur (clients, partenaires). Il faut que les comportements des uns et des autres soient cohérents pour que "l'entreprise-système" fonctionne correctement. La cohérence pour les systèmes d'informations (SI) vise à mettre en correspondance, au sein de l'entreprise, la vision stratégique du métier et la vision opérationnelle de l'informatique, de préférence via le prisme de l'urbanisation des SI (*cf.* page 41). Le problème de la cohérence est continu dans le temps puisque les systèmes évoluent en permanence et de manière souvent indépendante. La distance sémantique entre les points de vue rend difficile la mesure de l'impact de l'évolution du parc applicatif vis-à-vis des processus métiers ou des technologies. Au delà de modèles abstraits les architectes ont besoin de sonder les différentes couches du système d'information et quelque part d'aligner les concepts à travers ces couches. Nous proposons une approche pragmatique pour rapprocher les points de vue et aider à évaluer l'impact de restructurations sur l'évolution du parc applicatif. Une fois alignés les modèles des deux points de vue, des mesures estiment la qualité de l'alignement. L'approche présentée est mise en œuvre par des transformations de modèles et expérimentée sur des cas concrets.

Remarque

Ce chapitre reprend principalement les travaux menés durant la thèse de Jonathan Pepin [Pep16 ; Pep+18a] et mentionne les travaux en cours de la thèse d'Ali Benjilany. Ces travaux peuvent aussi s'appliquer dans le projet ONECAD mentionné en début de la Partie III.

6.1 Introduction

En théorie des organisations, on parle d'*alignement* ou alignement stratégique, lorsque les moyens mis en œuvre par une entreprise sont cohérents avec sa stratégie et contribuent efficacement à sa compétitivité [HV93]. Dans un système d'information (SI), la cohérence entre l'organisation et le système informatique qui implante ses processus automatisés est fondamentale du point de vue de la qualité du SI. On parle alors d'alignement entre le métier et l'informatique support, appelé couramment *Business/IT alignment (BITA)*. Cet alignement contribue à la performance de l'organisation [TS07 ; UL13 ; RSP17]. Face à la concurrence, les entreprises raccourcissent le cycle de décision et exigent une forte réactivité du SI alors que le cycle de maintenance et de renouvellement du parc applicatif, souvent hétérogène, est plus long du fait de contraintes budgétaires ou organisationnelles. Cet alignement est un défi lancé depuis plus de 20 ans avec l'article fondateur [HV93] et qui reste d'actualité [Col+15]. Il a évolué au cours du temps notamment avec l'émergence de l'**architecture d'entreprise** (ou *urbanisation*).

La démarche d'architecture d'entreprise permet de faire évoluer le système d'information d'une entreprise au rythme des stratégies à appliquer pour faire progresser l'activité de l'entreprise. Des plans de transitions (ou plans de migrations) se mettent en place selon un cycle itératif permettant l'évolution de sous-ensembles ciblés du système d'information [Lan13]. La **cartographie** vise à obtenir une image fidèle de l'état actuel des sous-ensembles du système d'information. Elle est constituée de *points de vue* du système d'information, notamment les points de vue des domaines *métier* et *informatique*. Les points de vue ont des objectifs, des méthodes, des acteurs, des représentations ou des pratiques variées. Les points de vue sont complémentaires, couvrant des aspects différents (composition), ou similaires mais avec des niveaux de détail différents (abstraction). Par exemple, il est nécessaire de structurer les différentes sources, support de la cartographie, hétérogènes par nature. D'une part, la stratégie et le fonctionnement de l'entreprise sont renseignés à travers des documents le plus souvent informels et dans le meilleur des cas à travers une modélisation métier décrite dans un langage plus ou moins formalisé et normé. D'autre part, le patrimoine applicatif est composé de programmes et d'infrastructures physiques. Sans cartographie applicative, la documentation est constituée au mieux de modèles d'architecture. Le cas échéant le code source est un point d'entrée possible, mais sa densité nécessite une abstraction méthodique des concepts techniques par détection ou filtre. Il peut être difficile de garder ces points de vue cohérents au fil des **maintenances**, des *changements technologiques* (mobilité, géolocalisation, objets connectés...) et des *évolutions stratégiques et organisationnelles* (lois, marché concurrentiel, actionnariat, économie collaborative, préoccupations environnementales...). L'évolution des points de vue du SI est un problème complexe et un enjeu important pour l'entreprise. Elle permet à l'entreprise d'être à la fois *performante et flexible*. Pour suivre et piloter l'évolution du SI, la démarche d'architecture d'entreprise est un projet qui mobilise un temps conséquent et du personnel en nombre ce qui peut *engendrer des coûts* importants.

Les travaux présentés ici s'inscrivent dans les étapes de cartographie et d'identification de la trajectoire du SI. Aligner de bout en bout chaque élément du SI, du code binaire déployé jusqu'à la stratégie d'entreprise, reste encore une gageure. Notre objectif est de proposer une *méthode pragmatique d'alignement* pour les systèmes d'information existants qui nécessitent des évolutions ; l'approche est guidée par les modèles. L'alignement *Business/IT* n'est qu'une étape de la démarche d'architecture d'entreprise et d'urbanisation [UL13 ; AGT13] mais elle est cruciale du point de vue de l'évolution du SI. Dans le cycle de la méthode de développement d'architecture (*Architecture Development Method, ADM*) du cadre d'architecture TOGAF (*The Open Group Architecture Framework*) présenté par la FIGURE 6.1, nos travaux assistent à l'accomplissement des étapes notées B, C, D et E. Notre méthode contribue à l'alignement entre les visions métier et technique (*Business/IT*) pour maintenir le patrimoine applicatif en phase avec l'évolution des métiers. Nous avons exposé la méthode dans [Pep+15a] et son exploitation des modèles dans [Pep+16b]. Les aspects techniques ont été détaillés dans [Pep+18b]

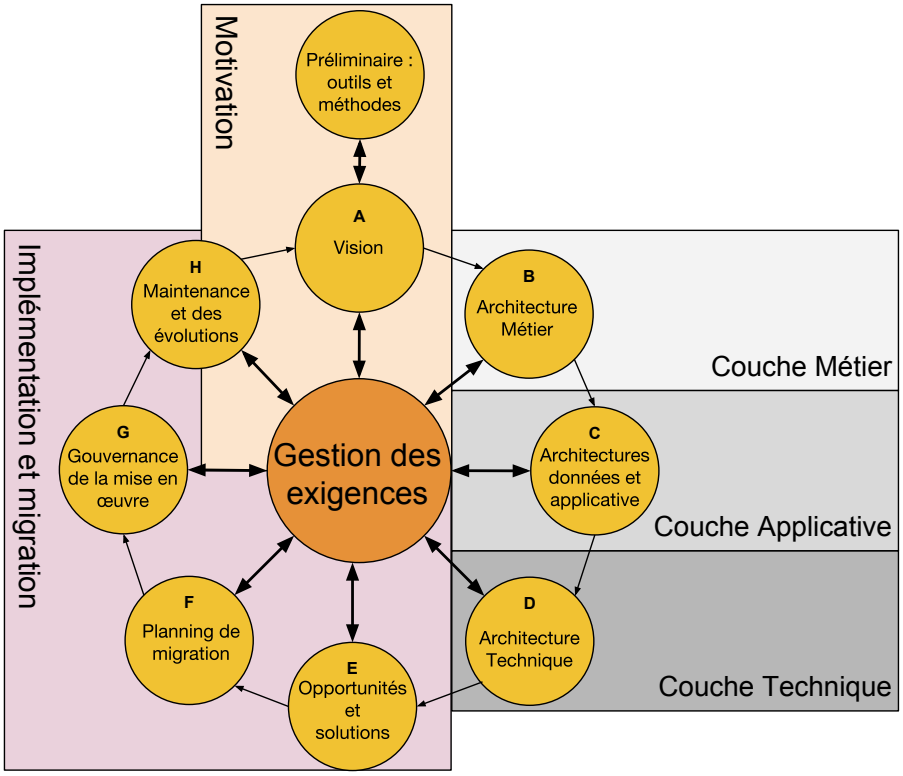


FIGURE 6.1 – Le cycle ADM du cadre d'architecture d'entreprise TOGAF [Pep+18a]

Dans ce chapitre, nous résumons les contributions et nous discutons des extensions pour un alignement sur l'ensemble des couches de l'architecture d'entreprise. La section 6.2 expose notre approche opérationnelle de l'alignement des visions métier et technique. Cette approche est orientée modèles et le cœur du problème concerne les vues métier, applicative et fonctionnelle. La cartographie est non-intrusive par tissage. L'évaluation de l'alignement est discutée en Section 6.3. L'approche est outillée et a été expérimentée sur plusieurs cas concrets (Section 6.4). Son extension haute vers la stratégie et basse vers l'infrastructure est discutée dans la Section 6.5.

6.2 Une approche pragmatique au cœur de l'alignement

Notre objectif est de définir une méthode d'alignement qui s'applique aux systèmes d'information existants (*legacy systems*) qui nécessitent des évolutions. Nous ne considérons que la dimension alignement *Business-IT* de [HV93]. En effet, au sens large, l'alignement couvre des aspects différents. La méthode GRAAL par exemple distingue trois dimensions : sociale (l'entreprise), physique (infrastructure) et symbolique (logiciel) [Lan13]. Nous avons réduit le périmètre au problème de l'alignement entre la vision métier et la vision informatique parfois déformée par le spectre de la vision architecturale des urbanistes.

Dans une vision globale, cet alignement BITA est interprété comme une ligne de traçabilité traversant les couches (partie gauche de la FIGURE 6.2) mais comme indiqué en introduction aligner de bout en bout chaque élément du SI, du code source déployé jusqu'à la stratégie d'entreprise, reste encore une gageure. L'expérimentation et la pratique des cas industriels nous ont montré que cette vision classique du SI est *idéaliste* : les entreprises n'ont pas toutes le même niveau de **maturité** ni la capacité à s'investir dans un projet d'architecture du SI. Ainsi, certaines entreprises n'abordent que l'essentiel du SI avec les vues applicative et fonctionnelle, tandis que d'autres poussent l'exercice jusqu'aux couches de processus métier et technique. De plus un SI est rarement cartographié en entier, le travail d'architecture commence lorsque surviennent des problèmes : techniques, de maintenance ou de coût. Lorsque les problèmes deviennent majeurs et ne peuvent plus être réparés par l'application de pansement (*Patch*), un audit complet est nécessaire pour redéfinir une vision stratégique du SI par la cartographie de l'existant. Ainsi, l'architecture va s'attaquer à *une ou plusieurs applications* mises en cause dans l'entreprise et non à l'intégralité du SI.

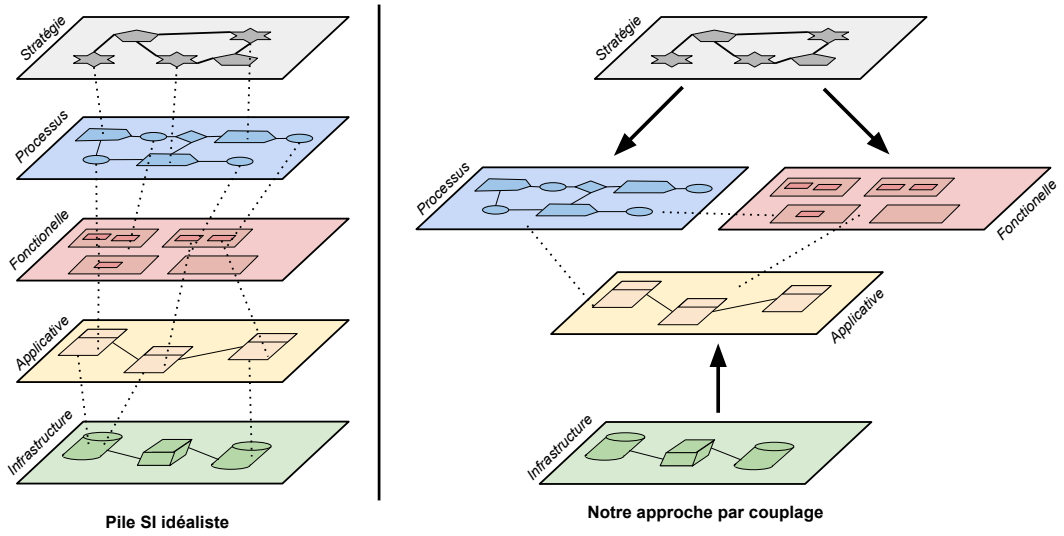


FIGURE 6.2 – Les couches du SI : approche idéale vs. approche pragmatique [Pep+16b]

En pratique, le but est surtout de rapprocher les modèles et de mettre en évidence les forces et faiblesses pour des scénarios d'évolution. Dans [Pep+16b] nous avons présenté une vision opérationnelle, illustrée à droite sur la FIGURE 6.2. Cette vision du SI met en évidence un **couplage** des différentes couches et non une traçabilité qui traverse les couches. Le cœur de l'alignement est formé d'un triplet de modèles BPM, App et Fun exprimant respectivement les points de vue métier, applicatif et fonctionnel. La traçabilité est mise en œuvre par des liens sémantiques variés entre ces points de vue (ou couche). Le point de vue métier décrit les processus métier du SI. Le point de vue application représente les applications du SI sous forme de composants et services logiciels. Le point de vue fonctionnel représente l'urbanisation du système d'information *e.g.* en zone/îlots/quartiers. Selon le niveau de maturité, la préoccupation, ou l'importance de l'entreprise, on disposera ou pas des points de vue métier et fonctionnel. L'approche est *flexible* (agile) car elle s'adapte aux éléments disponibles ; elle n'oblige pas à avoir à la fois la couche fonctionnelle et la couche processus ; Au cours du cycle d'architecture d'entreprise, il sera toujours possible d'ajouter des points de vue à l'alignement, sans avoir besoin d'une couche intermédiaire comme considérée dans la représentation de la pile idéaliste (à gauche FIGURE 6.2). Les liens sémantiques expriment des relations de raffinement ou de correspondance pour les données et les traitements. Les méta-modèles de ces points de vue et de leur alignement sont proposés dans [Pep16]. Les éléments principaux sont représentés dans la FIGURE 6.3 avec les liens entre ces éléments.

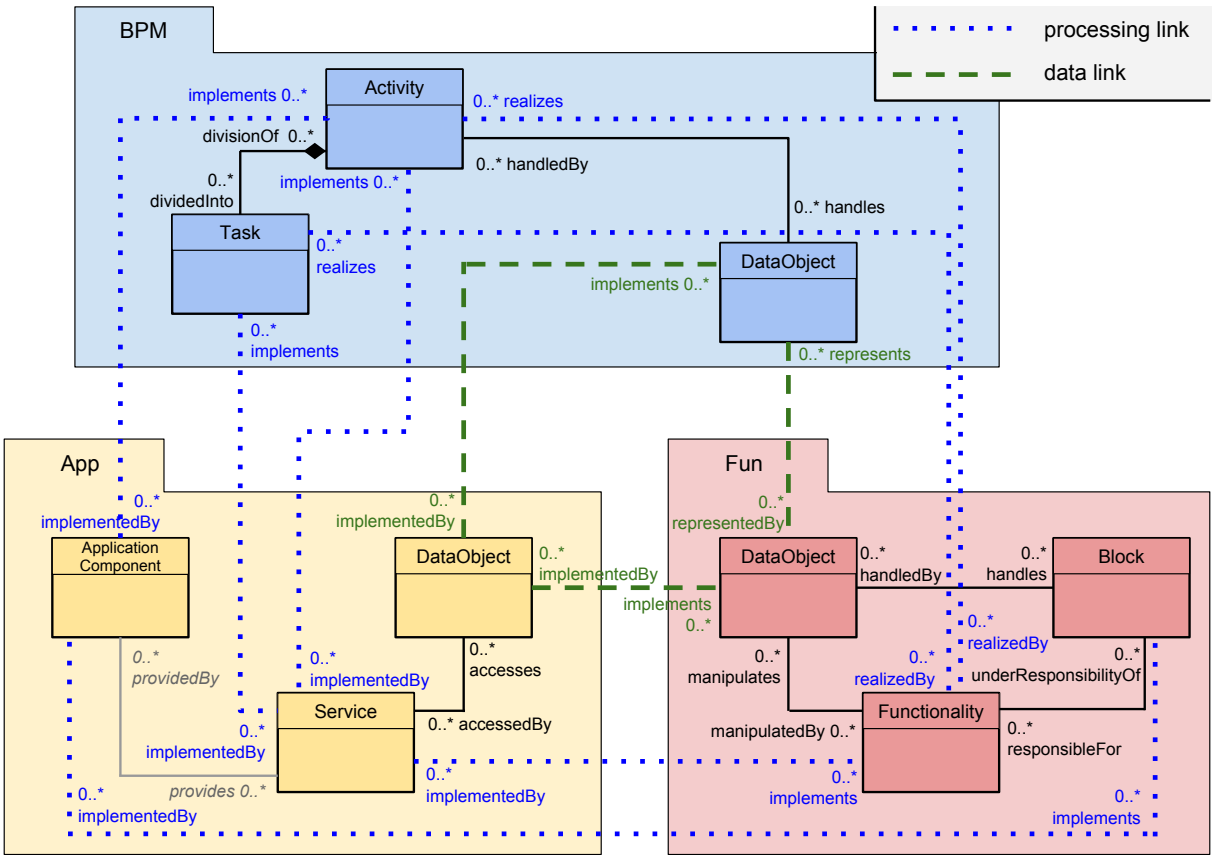


FIGURE 6.3 – Définition de l'alignement au niveau méta-modèle [Pep+16b]

Le *rapprochement des points de vue* se fait en concrétisant la stratégie par des modèles de processus métier (*top-down*) et en masquant les détails d'implantation par un processus d'abstraction ou de rétro-ingénierie (*bottom-up*) jusqu'à un niveau acceptable pour un "langage commun". Le langage commun est alors défini comme un tissage entre les langages *i.e.* une correspondance entre leurs concepts. La FIGURE 6.4 illustre ce processus global. A ce stade, nous faisons abstraction de la vision stratégique et la vision métier est

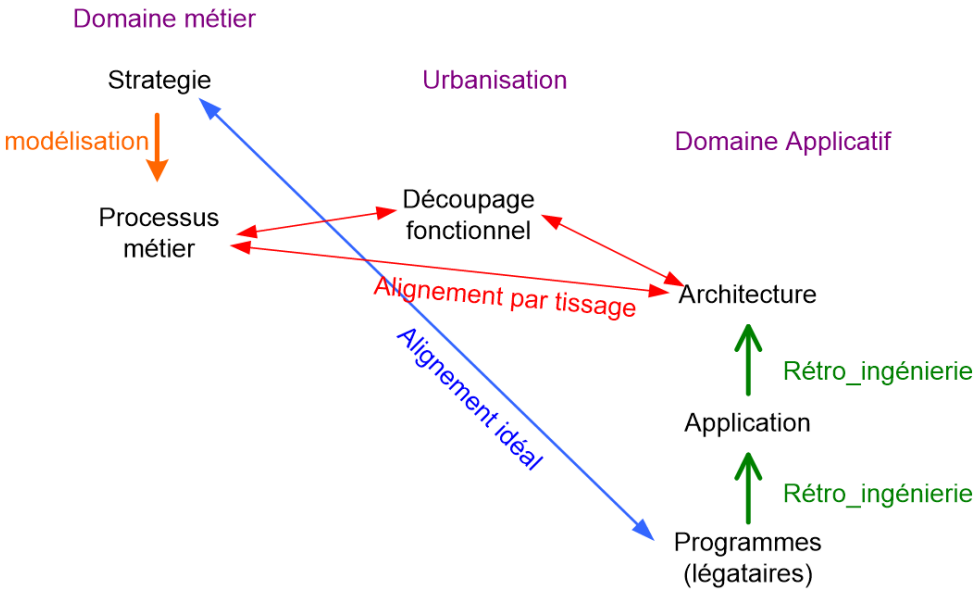


FIGURE 6.4 – Processus global d'alignement BITA

instrumentée par des modèles de processus métiers et fonctionnels. Nous y reviendrons en section 6.5.2. A l'autre bout de la chaîne, le code déployé implique une maîtrise de l'infrastructure pas toujours disponible, nous y reviendrons en section 6.5.1. L'alignement s'inscrit dans la première étape du travail de l'architecte dans le cycle ADM (FIGURE 6.1) pour établir la cartographie du SI (B et C). Le *processus d'alignement* vise à alimenter les modèles BPM, App et Fun de la FIGURE 6.3 en fonction des données disponibles dans l'organisation puis à les aligner concrètement.

Le processus concret est illustré par la FIGURE 6.5. Pour obtenir le modèle applicatif, nous avons proposé un processus de remontée en *abstraction* depuis le code source des applications (rétro-ingénierie). Une série de transformations `S1.x` permet d'abstraire les concepts techniques pour ne conserver que les éléments architecturaux. Pour obtenir les modèles métiers, c'est l'inverse, on doit matérialiser la stratégie d'entreprise, on parle de *concrétisation* `S2` ou de raffinement. Si la documentation métier est inexistante ou partielle, les différents acteurs (analystes métier, analystes fonctionnels et utilisateurs finaux) doivent définir les modèles fonctionnels et processus à partir de leurs savoirs et connaissances du fonctionnement et de l'organisation de l'entreprise. Les processus ont été implantés sous Eclipse avec Modisco [Bru+14] et Mia-transformation [1]. Les détails sont fournis dans le chapitre 5 de [Pep16]. Pour aligner des modèles obtenus (applicatifs, métiers

1. Mia-transformation est un logiciel de transformation de modèles qui fait partie de la suite Mia-Studio éditée par Mia-Software, qui fait partie de Sodifrance, partenaire de la thèse de Jonathan Pépin.

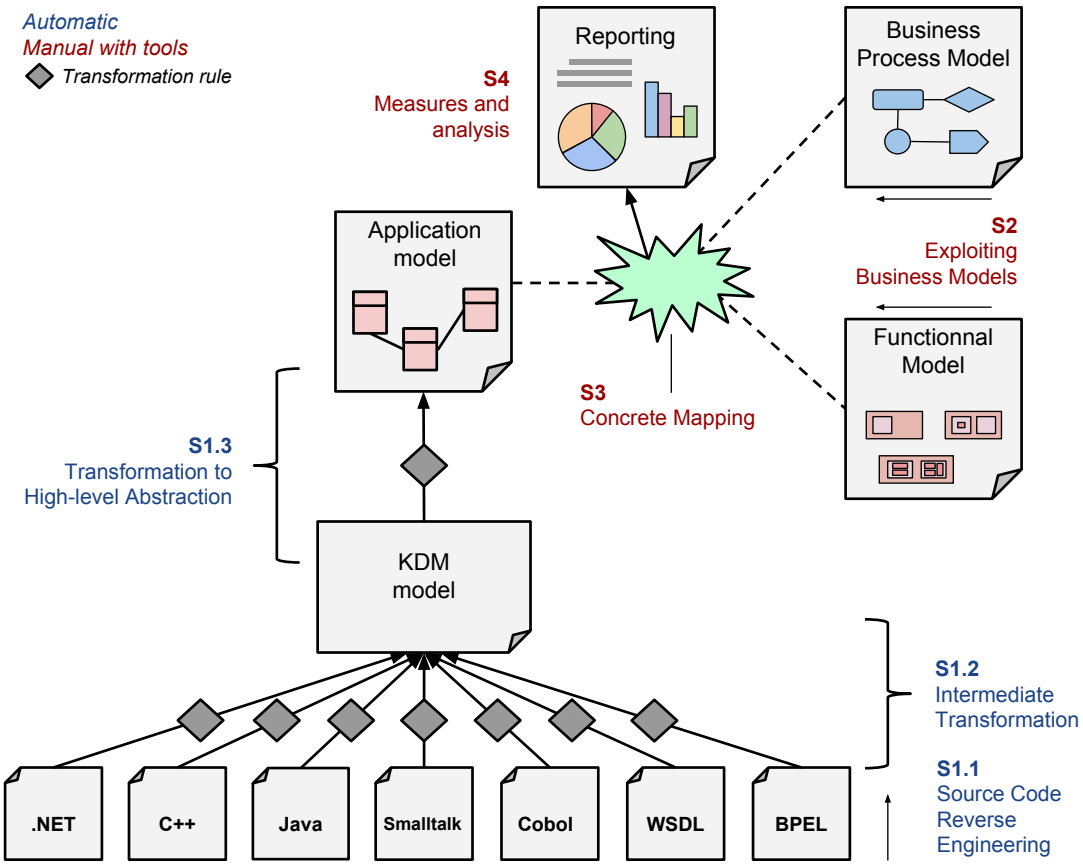


FIGURE 6.5 – Processus concret d'alignement BITA [Pep+15b]

et fonctionnels), nous proposons une solution non-intrusive S3 de *tissage* d'implémentation de facettes permettant l'extension virtuelle de méta-modèles. Nos propositions forment une méthode *outillée* présentée dans [Pep+18c] et qui utilise des standards de l'ingénierie des modèles. Nous avons notamment contribué au framework EMF Facet. Les résultats sont présentés dans un tableau de bord S4, disponible en page 150 de la thèse [Pep16].

Notre méthode est **générique** et **compatible** avec les principaux cadres d'architecture abordés précédemment. Il n'est pas nécessaire d'adopter un cadre unique, mais recommandé de prendre les **bonnes pratiques** adaptées à la situation dans différents cadres. Les modèles sont construits et alignés pour être analysés et répondre aux interrogations des décideurs. Nous abordons ce point dans la Section 6.3.

6.3 Maîtriser les évolutions du SI par analyse de dépendances

La cartographie et l'alignement des points de vue sont les étapes préliminaires d'un processus d'évolution du SI tel que celui de la FIGURE 6.1. L'architecte d'entreprise doit être capable d'identifier les éléments du SI à faire évoluer et de mesurer les impacts sur les différentes couches (ou points de vue). Les points de vue étant transverses (partie droite

de la FIGURE 6.2), l'alignement que nous avons proposé à partir de liens sémantiques permet à l'architecte d'*interroger le modèle i.e.* quels composants applicatifs implémentent telle fonctionnalité ? Quels processus métiers sont impactées par des modifications de composants applicatifs ou techniques ? L'objectif des travaux est de fournir aux urbanistes des outils d'assistance à l'évolution.

L'évaluation de l'alignement vise à détecter des problèmes de cohérence ou de complétude entre couches. Nous avons dans un premier temps travaillé sur une mesure de la qualité de l'alignement avec un indicateur global permettant d'étalonner la cohérence globale du système d'information. Nous n'avons pas trouvé de calcul d'un agrégat pertinent. L'analyse détaillée nous a convaincu que ce n'était ni la voie, ni le besoin. Ce n'est pas la voie car trouver un référentiel de bonne qualité de l'alignement se révèle ardu et finalement trop complexe. Ce n'est pas le besoin car l'objectif des architectes n'est pas d'aboutir après plusieurs itérations à un "super" alignement final, mais plutôt d'établir un constat à un moment donné avec des pistes d'améliorations puisque le système d'information évolue en permanence. L'évaluation peut se faire par observation/modification, par une analyse structurelle ou par des requêtes spécifiques. Dans la suite nous présentons trois techniques : l'outil de tissage (Section 6.3.1), l'analyse par requêtes (Section 6.3.2) et l'analyse de dépendances (Section 6.3.3).

6.3.1 Observer et modifier l'alignement du SI par tissage

L'alignement de modèles réalisé à l'aide de l'approche de la section 6.2 permet une navigation bi-directionnelle, *montante et descendante* entre les points de vue du SI. Un éditeur arborescent compatible avec le framework Eclipse EMF permet de charger les modèles et les liens sémantiques d'alignement permettent la navigation. Nous proposons un éditeur de tissage pour créer, consulter, modifier ou supprimer les liens d'alignement entre les trois points de vue exhibés dans la section 6.2. Cet éditeur est donc un navigateur dans l'alignement. La FIGURE 6.6 illustre cet éditeur sur un cas de simulation d'assurance “incendie, accidents et risques divers” (I.A.R.D.). A droite, le point de vue applicatif comprend différents services et fonctions et le point de vue fonctionnel met en évidence le bloc Adhérent d'un contrat IARD. A gauche le tissage relie le bloc fonctionnel *Simulation contrat* avec les services du modèle applicatif et notamment le service *validerSimulationContrat*.

Naviguer et modifier le tissage ne suffit pas pour évaluer. D'une part, la navigation arborescente même assistée par des filtres de recherche est peu pratique pour les modèles volumineux. D'autre part, elle ne répond qu'aux seuls types de questions posées précédemment. Nous souhaitons en poser d'autres *e.g.* quels sont les éléments qui ne sont pas alignés correctement ? Afin de pallier ce problème, nous proposons dans les sections suivantes deux types d'analyse pour accompagner la prise de décision des évolutions à appliquer au SI : une analyse par requête et une analyse par matrice de dépendances.

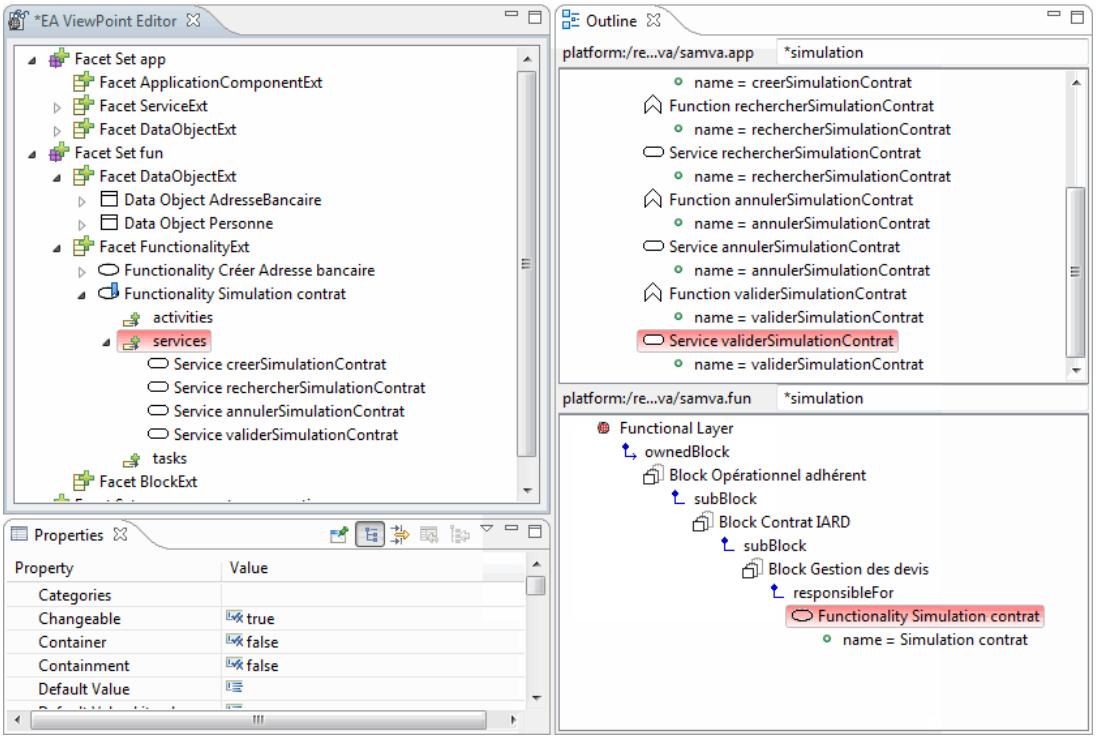


FIGURE 6.6 – Editeur de tissage pour la navigation entre modèles [Pep+18a]

6.3.2 Mesurer l'alignement du SI par requêtes

Le travail d'alignement est proportionnel au volume d'informations cartographiant les points de vue du SI. L'architecte doit pouvoir suivre son avancement et vérifier sa cohérence, mais aussi mesurer l'alignement pour fournir aux décideurs des indicateurs guidant à la fois l'analyse de l'état actuel, la détection d'anomalies et la valorisation de scénario de projections futures.

La mesure de l'alignement reste inexplorée [AGT16]. Les travaux de Simonin [Sim09] se focalisent sur la couche fonctionnelle (urbanisation). D'autres traitent surtout de la couche métier (voire stratégique) avec le logiciel. Ils incluent la notion d'acteurs, qui a du sens du point de vue métier mais pas du point de vue architecture logicielle. Aversano et al. [AGT10] traitent de la couverture des processus métiers par le logiciel, mais ne présentent pas les modèles du logiciel. Ils ont proposé des métriques pour l'*alignement fonctionnel* dans [AGT16] mais pour des modèles de bas niveaux (activités et classes UML). Nous avions défini des métriques d'indice de couplage et de cohérence similaires mais leur intérêt restait difficile à appréhender car l'alignement est multiforme. Rolland et Etien [ER05] alignent les processus métiers avec des modèles UML via des ontologies mais sans parc applicatif. Thévenet et al. [TS07] s'intéressent à l'alignement stratégique et l'évolution, sans lien direct avec le code.

Nos indicateurs sont classés en quatre catégories : couverture, cohérence, densité d'alignement concret (entre les modèles métiers, applicatifs et fonctionnels) et couverture du code final. Le *framework* Eclipse EMF permet d'interroger les modèles par requêtes écrites dans le langage OCL. Nous avons étendu OCL pour que les requêtes soient compatibles

avec EMF Facet, utilisé pour le tissage entre modèles. Les requêtes servent à mesurer les indicateurs d'alignement. Ainsi, une première mesure possible est la complétude de l'alignement illustrée par la FIGURE 6.7 : le nombre d'éléments alignés sur le nom d'éléments alignables des points de vue processus et applicatif, selon la définition de l'alignement (FIGURE 6.3) entre points de vue au niveau méta-modèle.

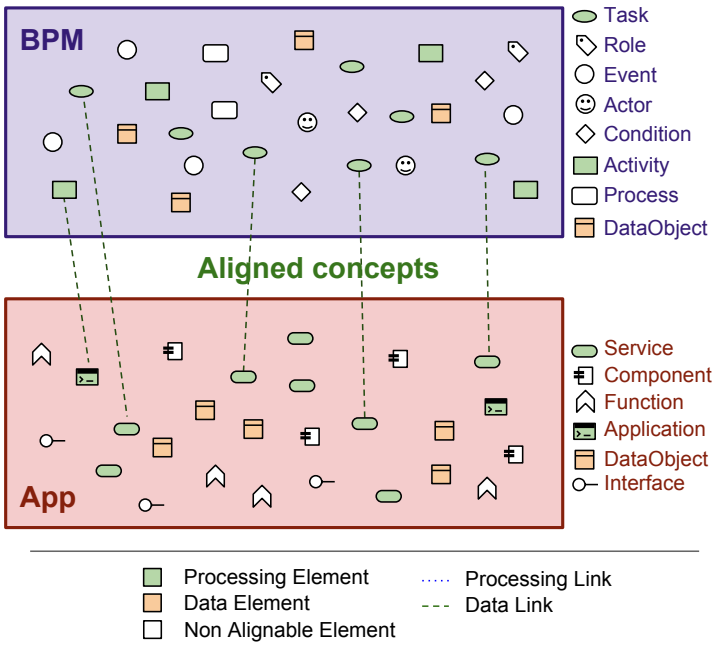


FIGURE 6.7 – Complétude Activités de Processus/Service applicatif [Pep+18a]

Les concepts sont typés, ce qui autorise des analyses par préoccupation (*concerns*) *e.g.* données et traitements sont distingués. Les requêtes permettent la vérification de cohérence entre traitements et données. Si un élément de traitement d'un point de vue A qui manipule des éléments de données, est aligné avec un élément de traitement d'un autre point de vue B, alors les éléments de données du point de vue A doivent être alignés aux éléments de données du point de vue B et réciproquement. La requête du Listing 11 liste les blocs ayant un alignement incohérent de données avec des composants applicatifs.

```
BlockAppComponentNonConsistent:
context FunctionnalLayer
fun::Block.allInstances()→select(applicationComponents→notEmpty())→symmetricDifference(
  fun::Block.allInstances()→select(blk |
    blk.applicationComponents.provides.accesses.dataObjectsFun.handles→includes(blk)))
```

Listing 11 – Calcul des oublis entre Bloc Fonctionnels et Composants applicatifs

L'analyse par requête de l'alignement détecte des oublis et incohérences et de les corriger à chaque itération permettant un réel suivi qualité en continu du SI. Des règles d'urbanisme spécifiques au SI peuvent aisément être implémentées en OCL et compatibles avec les liens d'alignement.

6.3.3 Identifier les dépendances entre points de vue

Un SI doit avoir des qualités de cohérence forte et couplage faible, bien connues en conception logicielle [Par72]. Le SI au fil des évolutions peut devenir intriqué. L'analyse de dépendances répond ainsi à des problématiques très diverses. Du point de vue applicatif, elle permet à l'architecte de mettre en évidence (i) les zones de couplage fort candidates à une restructuration (*refactoring*) (ii) les impacts en cas de débranchement ou de remplacement de composants applicatifs. Du point de vue processus métiers, elle permet d'identifier les dépendances entre activités mais aussi de déterminer les composants applicatifs impactés par une réorganisation métier. Outre le calcul des dépendances, l'architecte a besoin de représentations visuelles des dépendances (à grande échelle) pour se projeter dans l'évolution.

Techniquement, les modèles alignés représentent un graphe composé de nœuds (les instances des types) et d'arcs (les relations entre instances). Ce qui permet d'appliquer des algorithmes issus de la théorie des graphes. Les graphes peuvent se représenter par une matrice, et plus précisément une matrice d'adjacence. Les nœuds sont représentés entête de chaque ligne (l) et colonne (c), une intersection entre une ligne et une colonne est la présence d'une dépendance entre deux nœuds. Cette matrice est appelée *Dependency Structure Matrix* (DSM) [EB12]. Il existe donc $l \times c$ possibilités de représenter le graphe en matrice, selon l'ordre de lecture de chaque nœuds. De ce constat quantitatif, nous nous sommes posés la question du meilleur ordre pour représenter notre graphe d'alignement et ainsi de faciliter la lecture de l'architecte pour visualiser les dépendances et notamment identifier les groupes de dépendance. Nous avons étudié les algorithmes de regroupements (*clustering*) [Sch07] pour choisir le plus adapté à la nature de nos travaux. Nous avons retenu l'algorithme Markov Cluster (MCL) [Don00] qui présente l'avantage de ne pas nécessiter de connaître à l'avance le nombre de clusters mais l'inconvénient de s'appliquer naturellement à un seul point de vue à la fois.

La matrice correspondant à un alignement des points de vues (processus, fonctionnel, applicatif) est multi-domaine, *Multiple-Domain Matrix* (MDM) [Bar+10]. L'application d'un algorithme de regroupement sur une matrice MDM est plus délicate. Notre contribution est de proposer d'appliquer cet algorithme de trois manières différentes sur l'alignement : (1) un regroupement global qui s'applique sur toute la matrice mélangeant les domaines, (2) un regroupement intra-domaine qui ne s'applique qu'à l'intérieur de chaque domaine et (3) un regroupement inter-domaine qui s'applique à l'intersection entre deux domaines. Cette variante permet à l'architecte de visualiser précisément par quels liens les points de vue sont en dépendance. Les algorithmes de regroupement sont déterministes et fournissent donc un résultat identique quelle que soit l'exécution d'un jeu de valeurs.

La Figure 6.8 donne un aperçu du *clustering* de notre outil pour le même exemple que la Figure 6.6. Il s'agit du calcul de regroupement MCL inter-domaine pour l'alignement entre le point de vue fonctionnel et le point de vue applicatif. Les lignes et colonnes représentent les concepts du modèle applicatif (en jaune) ou fonctionnel (en rose). La

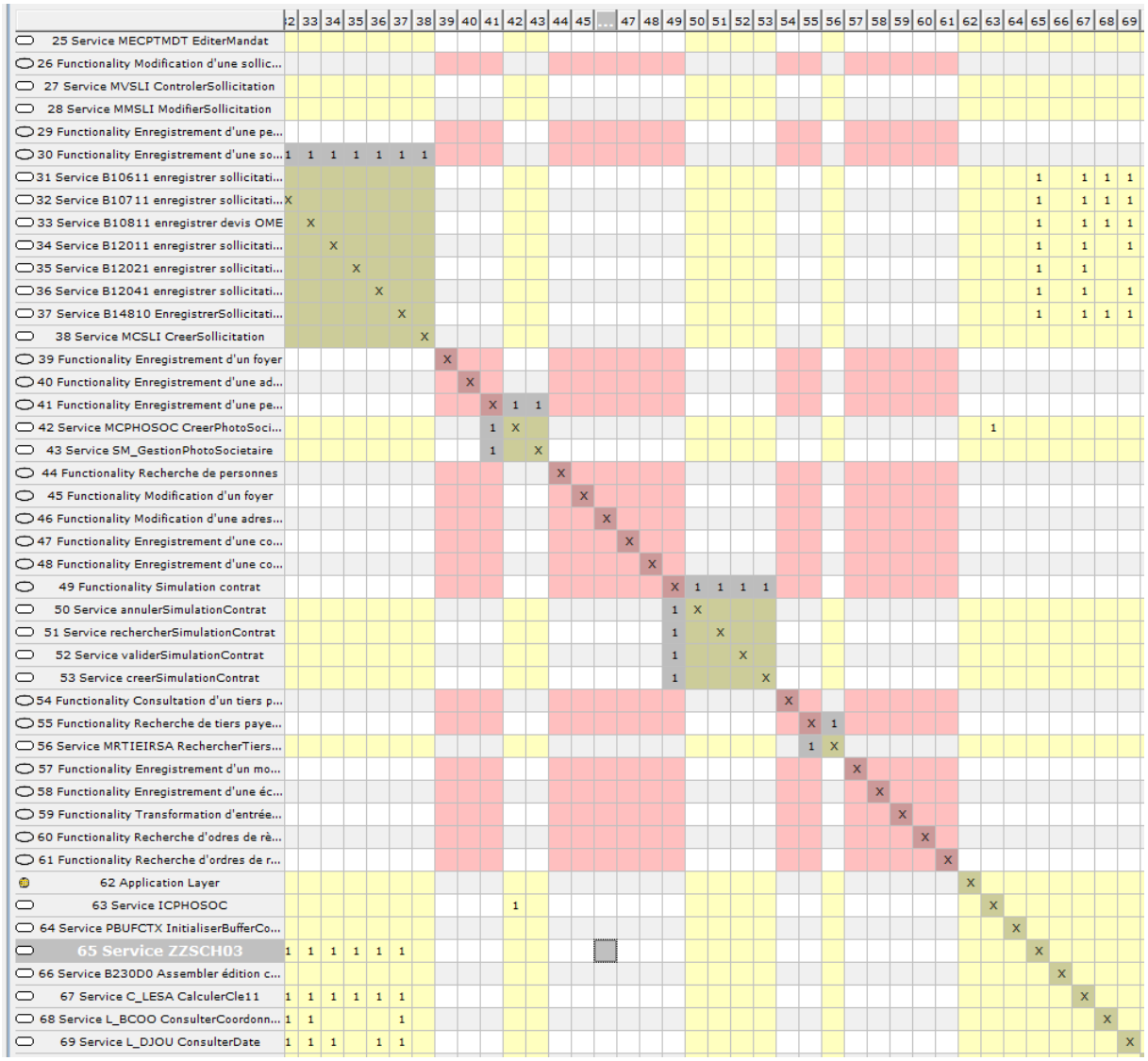


FIGURE 6.8 – Matrice de dépendances entre modèles [Pep+18a]

ligne diagonale montre que chaque concept est lié à lui-même.

Les points intéressants sont les sous-matrices plus pleines comme celles de la ligne 30 et suivantes qui mettent en évidence un cluster de services, concernant ici les sollicitations clients. Ce cluster de services (lig/col 31-37) est aussi en dépendance applicatif-applicatif avec des services utilitaires d'accès aux données et d'édition (lig/col 65-69) On voit un couplage applicatif-fonctionnel sur la gestion des photos (lig/col 41-43), sur la simulation des contrats (lig/col 49-53).

6.4 Expérimentations

Dans un premier temps, nous avons testé la méthode (principes, démarche, outils) sur des cas simples et imaginaires, comme celui qui a servi à illustrer nos propos dans les sections précédentes. Nous avons eu ensuite l'opportunité de tester notre démarche sur trois cas d'études concrets, provenant de sociétés d'Assurance Mutuelle françaises

que nous nommerons SAMM, SAMI et SAMUT. Chaque étude possède ses spécificités et couvre tout ou partie des modèles du processus de la section 6.2. Il ne s'agissait pas de réaliser un cas de bout en bout et le tissage reste partiel. L'expérimentation a pour but de vérifier la faisabilité de notre démarche : pertinence des modèles vis-à-vis de la pratique en entreprise, adéquation de la démarche, automatisation des transformations et de leur enchaînement, pertinence du tissage et de sa mise en œuvre, application à des modèles volumineux. Nous résumons ci-après les expériences.

Le cas SAMM est composé *i)* d'un code source complet écrit en Java avec 33 400 classes (3 400 000 lignes de codes) et *ii)* d'un référentiel d'entreprise sous la forme d'un portail HTML exporté depuis le logiciel MEGA EA [2]. Le but de ce scénario était de valider que notre méthode permet d'exécuter l'analyse par clustering d'un alignement réel à l'aide de notre matrice de dépendance. Le référentiel d'entreprise contient 360 diagrammes de processus métier couvrant la totalité du SI. Le volume de l'application est conséquent et représente un véritable défi à traiter. Ce cas d'étude nous a permis de tester notre approche sur un code source conséquent. Le chargement des modèles obtenus par rétro-ingénierie a été un défi pour nos outils. La transformation a été facilitée par la présence d'une nomenclature des concepts qui se retrouve à différents niveaux et par là même montre de bonnes pratiques de codage, à quelques exceptions près. Néanmoins, nous regrettons ne pas avoir eu accès au source original du référentiel MEGA pour réaliser un tissage complet. Nous avons du nous contenter du portail HTML.

Le cas SAMI est composé uniquement d'un référentiel MEGA dont le source est disponible. Le but de ce scénario était de valider que notre méthode permettait bien d'exécuter une remontée en abstraction d'un code applicatif réel et conséquent. Nous avons extrait les différents concepts pour peupler à l'aide d'une transformation nos différents modèles (App, BPM, Fun) afin de vérifier la couverture et la compatibilité des concepts. Le cas présente 625 composants, 11894 services, 18 blocs fonctionnels, 131 fonctionnalités, 167 processus et 268 activités.

Le cas SAMUT n'avait aucune représentation métier. Le but de ce scénario était de valider que notre méthode permet de construire un alignement à partir de zéro en constituant les modèles et d'aider l'architecte d'entreprise à visualiser les dépendances. Les objets de données ont été extraits par rétro-ingénierie d'une base de données hiérarchique et les composants applicatifs de procédures stockées. Un architecte à ensuite modélisé et identifié des blocs fonctionnels et nous avons alors pu réaliser le tissage entre les blocs et les composants applicatifs. Le SI comporte 12 blocs, 1045 composants et 669 objets de données. Ce cas d'étude a permis de tester notre méthode de tissage, et isoler des composants qui étaient orphelins (rangés dans aucun bloc).

Ces trois cas ont des propriétés particulières, les supports sources sont hétérogènes et représentatifs de la disparité de maturité des SI. Ils ont mis en évidence la souplesse de notre méthode qui peut s'adapter à chaque étape. La contrepartie est d'enrichir l'écosys-

2. `http://www.mega.com/fr/solution/business-architecture`

tème des modèles et transformations pour chaque nouveau cas rencontré.

6.5 Extensions de l'alignement opérationnel

Dans les sections précédentes, nous nous sommes attachés à réduire le fossé entre domaine métier et informatique. Nous discutons ici de l'extension de l'alignement aux couches infrastructure et stratégie en vue de couvrir l'ensemble des vues du SI.

6.5.1 Alignement d'infrastructure

Dans nos cas d'étude, nous avons constaté que les entreprises maintenaient une cartographie technologique composée des informations sur les serveurs, systèmes, réseaux et caractéristiques physiques. Cette cartographie permet à la direction des systèmes d'information (DSI) de réaliser une intervention de maintenance ciblée. Par exemple, en cas de panne d'un service applicatif, il est nécessaire de retrouver rapidement les coordonnées du logiciel incriminé : IP, droits d'accès, site physique ou cloud, etc. Conserver une cartographie technologique actualisée est une forte préoccupation. Une panne entraînant la cessation d'une partie de l'activité de l'entreprise ayant un coût lié au temps d'immobilisation, la résolution doit-être la plus rapide possible pour limiter les pertes. Nous avons établi un premier état de l'art qui met en évidence que certaines cartographies technologiques mélangent tout les aspects, tandis que d'autres distinguent les concepts de déploiement et des concepts d'infrastructure.

Déploiement Ce point de vue a pour but de décrire comment les applications sont réparties au niveau virtuel. La vue applicative décrit les composants, fonctions ou objets de donnée, mais pas leur nature technologique. Par exemple, un ou plusieurs composants écrits en Java peuvent être encapsulés dans des archives *jar*, *war* ou *ear*et déployés dans un serveur d'application J2EE (Glassfish, JBoss, Apache Tomcat...) ; un ou plusieurs objets de données sont stockés dans une table de base de données (MySQL, Postgre, Oracle, Neo4j, MongoDB...).

Infrastructure Ce point de vue a pour but de décrire le matériel où sont installés les logiciels, ainsi que le réseau et les interfaces d'échanges d'informations. Ce point de vue modélise la situation géographique du matériel (pays > ville > site > bâtiment > salle > baie > unité) et peut couvrir plusieurs sites distants.

Archimate®[3] distingue le niveau technologique (intergiciel, composants et services des applications) et le niveau physique (équipements matériels, de réseaux physiques...) de la FIGURE 6.9. Le niveau technologique peut être qualifié de virtuel.

La Direction Interministérielle des Systèmes d'Information et de Communication (DISIC)[4] de l'État français propose un seul point de vue infrastructure (FIGURE 6.10) qui couvre le niveau physique.

3. `http://pubs.opengroup.org/architecture/archimate3-doc/`
4. `https://references.modernisation.gouv.fr`

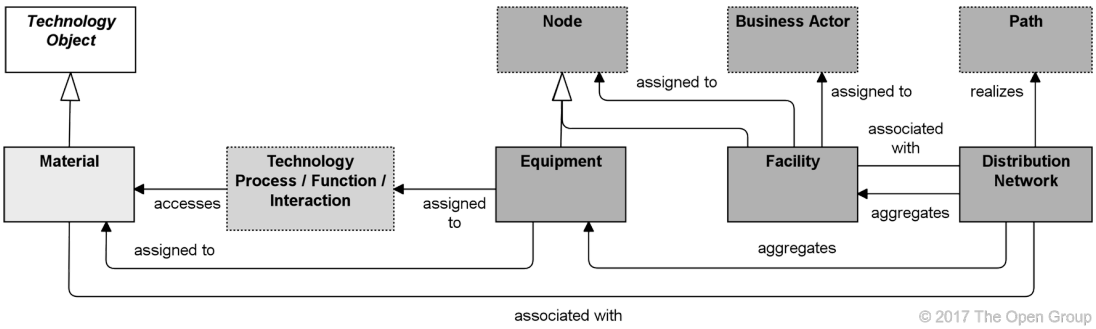


FIGURE 6.9 – Le méta-modèle physique de ArchiMate 3.0.1[3]

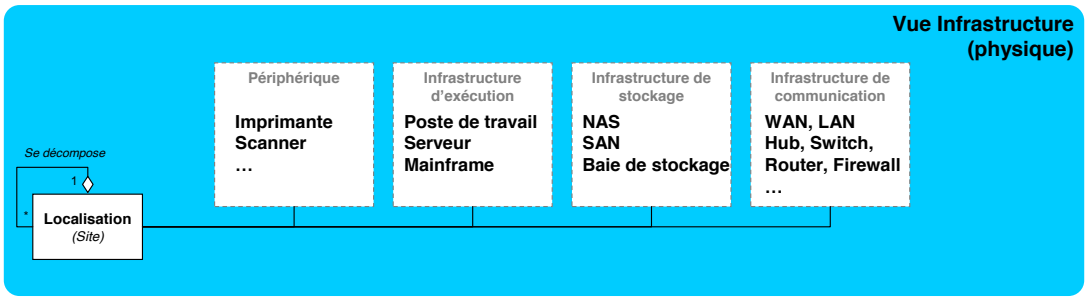


FIGURE 6.10 – Les concepts simplifiés de la Vue Infrastructure selon DISIC[4]

D'autres études de méta-modèles d'infrastructure doivent étoffer notre état de l'art et nous permettre de créer une extension générique de notre définition de l'alignement au niveau méta-modèle qui doit rester indépendante d'une solution spécifique.

6.5.2 Alignement stratégique

L'alignement stratégique est une thématique qui a généré une grande production scientifique depuis l'article fondateur d'Henderson et Venkatraman [HV93]. Cela s'explique non seulement par l'intérêt croissant du sujet mais aussi plus prosaïquement par le fait que le problème relève à la fois des domaines de la gestion et de l'informatique. Nous restreignons notre domaine d'investigation aux approches incluant des modèles, notamment autour de l'architecture d'entreprise. Le problème se résume alors à disposer de modèles pour la couche stratégie et la couche processus métiers et de modèles pour l'alignement entre ces couches conformément à l'approche de la Figure 6.2. Nous disposons déjà de langages pour la couche métier.

L'étude bibliographique met en évidence les éléments suivants :

— Il existe de nombreux langages et techniques de modélisation de la stratégie ou de maturité [UL11 ; AGT16 ; RSP17], ce qui ne facilite pas la mise en pratique pour les entreprises et la conception d'outils d'aide.

— Certaines approches s'appliquent à la conception (*design time*) [TS07] en établissant des ponts entre couches (traçabilité), l'ingénierie des exigences y joue un rôle prépondérant [TS07 ; UL11]. D'autres visent la rétro-conception en annotant les modèles métiers d'indicateurs exploités pour relier à des concepts de la couche stratégie.

— La couche stratégique peut elle-même être décomposée en plusieurs couches notamment en séparant par exemple une approche basée sur les *buts* (*goal modeling languages* [UL11 ; DBB11], i* [PGA08]) et celle basée sur la *valeur* (VMDL [RP13], e^3-value [PGA08]). Une approche telle que *Process Goal Alignment* (PGA) permet de relier les deux niveaux [RSP17].
— La jonction entre les couches métier et stratégique peut se faire en annotant les processus métiers d'informations utiles à l'alignement avec la stratégie. Morrisson et al. [Mor+12] proposent ainsi un calcul d'alignement sur les buts basés sur ces annotations. Ullah et Lai [UL11] proposent de relier les processus métiers aux buts.
— Plus récemment, le *Business Motivation Model* (BMM) de l'OMG contient des éléments pour spécifier la stratégie et d'autres éléments pour relier le niveau stratégie au niveau métier [Bha17 ; HP14].

Par la suite, nous envisageons une mise en œuvre d'une version générique de BMM qui a l'avantage de s'intégrer avec nos standards de modélisation que ce soit TOGAF, UML ou EMF.

6.6 Conclusion

Les travaux mentionnés dans ce chapitre sont en phase avec mes enseignements de conception de systèmes d'information en Miage. Nous apprenons aux étudiants à développer des applications d'informatique de gestion qui répondent aux besoins des organisations. L'alignement BITA est une manière d'évaluer l'adéquation entre les applications et l'entreprise. Durant la thèse de Jonathan Pepin, nous avons proposé une méthode pragmatique au problème d'alignement des points de vue métier et applicatif s'insérant dans la démarche d'urbanisation des architectures d'entreprise. Notre méthode est basée sur une proposition de modèles intermédiaires génériques, un rapprochement des points de vue et un alignement par tissage de concepts comparables. Le rapprochement est rendu possible par une abstraction progressive du code en architecture applicative à base de composants et services. L'approche est outillée dans le cadre d'Eclipse EMF et a été expérimentée sur des cas réels d'entreprises clientes de Sodifrance, partenaire de la thèse, permettant d'éprouver la viabilité de l'approche. L'application à des applications de taille importante fut un défi mais les modèles obtenus par rétro-ingénierie ont pu être chargés par les outils développés.

Même s'il peut être amélioré sur divers points, nous estimons que le résultat, doit permettre aux architectes de cartographier, vérifier, qualifier, quantifier l'alignement actuel d'un système d'information (*As-Is*). L'étape suivante est de travailler sur les scénarios d'évolution, qu'elles soient technique ou organisationnelle, et de mesurer l'impact des changements. L'objectif est alors de proposer méthodes et outils permettant d'évaluer les différents scénarios de situations futures (*To-Be*) en termes de coûts pour aiser les décideurs dans leur stratégie. Il s'agit là d'un besoin fort des architectes.

Post-scriptum

Nos travaux sur l'alignement BITA sont aussi au cœur de la thèse d'Ali Benjilany démarrée en Septembre 2021 sous la direction de Dalila Tamzalit que je co-encadre avec Hugo Brunelière. La thèse se focalise sur l'alignement opérationnel entre les couches métiers et applicatives. L'originalité est de traiter différents points de vue, notamment les fonctions, les données, la sécurité, la confidentialité, etc. Un premier résultat est une étude systématique de l'alignement opérationnel entre les couches métiers et application, publiée à ISD 2023 [And+23] et enrichie en chapitre pour un volume Springer LNISO à paraître en juillet 2024. Dans cette étude nous comparons 44 approches sur cinq critères : (i) les modèles des couches, (ii) les liens entre couches, (iii) la définition et l'exploitation de l'alignement, (iv) l'outillage et (v) les applications. Nous mettons en évidence les limites et suggérons des pistes de travail. Une première proposition de méthode pour l'alignement opérationnel a été présentée à la conférence ICEIS 2024 [Ben+24b] et sélectionné pour un chapitre d'un volume Springer LNBIP. Elle a été complétée par une évaluation par anti-patterns d'alignement à Inforsid 2024 [Ben+24a]. Une proposition multi-évaluation (métriques, règles, anti-patterns) de l'alignement sera présentée à ISD 2024. Nous sommes aussi en lien avec des entreprises et collectivités pour la mise en pratique sur des cas du quotidien.

Ce chapitre a mis en évidence des acquis sur les compétences suivantes, que nous reverrons dans le chapitre 8.

C2

Identifier les problèmes, les ordonner, imaginer des pistes ambitieuses, proposer des pistes réalistes, ordonnancer un ensemble d'activités de recherche pour structurer un travail de recherche scientifique sur du long terme.

C3

Organiser une recherche collaborative, détecter des compétences, définir des complémentarités et coordonner pour contribuer à un travail de recherche d'envergure.

C5

Investir de nouveaux domaines, collaborer avec des chercheurs d'autres spécialité implique de se comprendre et se faire comprendre, les apports croisés permettent d'innover dans les solutions des problèmes identifiés.

Chapitre 7

Rationaliser les systèmes de production

The factory of the future will have only two employees, a man and a dog. The man will be there to feed the dog. The dog will be there to keep the man from touching the equipment.

Carl Bass, Autodesk CEO

Le domaine qui nous intéresse ici est celui des systèmes cyber-physiques de production (Cyber-Physical Production Systems – CPPS). C'est un domaine réellement **pluridisciplinaire**. On le constate dans la structuration du CNU [1] [2]

- Le domaine est principalement centré sur le thème Sciences et le groupe 9 du CNU autour des sciences physiques et les sections 60 à 63. Evidemment la section 61 Génie informatique, automatique et traitement du signal est centrale à l'idée d'automatisation dans les systèmes cyber-physiques (CPS).
- Les "systèmes de production" incluent des thématiques comme la gestion de production, l'organisation, le pilotage, le management issues des Sciences de gestion et du management (Section 06 du groupe 2 du thème Droit, économie et gestion du CNU). Les objectifs sont la réduction des coûts l'amélioration de la qualité et la diminution des délais. La Gestion de Production Assistée par Ordinateur (GPAO) désigne de manière un peu désuette l'usage de l'informatique pour aider dans la réalisation de la production.
- L'automatisation est intimement liée à l'informatique. Le domaine CCPS est doublement couvert par l'informatique industrielle (Section 61) et l'informatique (Section 27 du groupe 5 mathématique et informatique) avec en particulier l'informatique de gestion qui se focalise sur les systèmes d'information d'entreprise dont fait partie la production.

L'ingénierie système, que nous avons introduit dans la Section 1.2.4 du Chapitre 1, joue un rôle-clé dans la mise au point des systèmes de production. Progressivement, le numérique

1. Conseil National des Universités
2. C'est moins clair dans la classification ACM 2012 où *Industry and manufacturing* est classée comme une branche de *Operations research*, on trouve *Computer systems organization> Embedded and cyber-physical systems* mais aussi des éléments dans pleins de domaines comme *hardware*, *hardware*, *applied computing*...

a révolutionné le monde industriel en proposant une intégration complète des services et métiers de l'entreprise. Le logiciel joue désormais un rôle clé dans le domaine CPPS.

Remarque

Ce chapitre reprend principalement les travaux menés en collaboration avec Olivier Cardin depuis 2017 sur le pilotage des systèmes de production.

7.1 Introduction

Mon appétence pour le monde industriel et des systèmes de production est assez ancienne ; j'ai fait mes premières armes avec Franck Barbier [Bar91], autour d'un pilotage par kanbans [ABR95] et nous avons poursuivi avec un projet de collaboration régional appelé CIM-Anjou. Ce projet est un projet étalé sur plusieurs années, de 1995 à 1999. Il a été financé par l'Etat, la Région, le Département, la Ville d'Angers, ainsi que par des fonds européens (fonds FEDER). Les partenaires étaient : l'ISERPA (Institut Supérieur d'Enseignement et de Recherche en Production Automatisée, Angers.), l'IMA (Institut de Mathématiques Appliquées, Angers), l'IRIN [3] (Institut de Recherche en Informatique de Nantes), le LIUM (Laboratoire d'Informatique de l'Université du Maine, Le Mans) et le Centre Régional d'Innovation et de Transfert de Technologies (CRITT) Productique. L'objectif principal du projet CIM-Anjou était d'apporter de nouveaux outils d'ordonnancement industriel aux entreprises sous la forme d'un Atelier de Conception et Réalisation de logiciel d'Ordonnancement Industriel. Ce projet a abouti à une valorisation industrielle en 1998 avec GFI Informatique sous l'impulsion du CRITT Productique.

Bien plus tard, lorsque nous cherchions des domaines d'application pour COSTO/K-melia (*cf.* Chapitre 3), Marianne Huchard [4] nous avait suggéré la robotique. En l'absence de cas concrets, je me suis donc intéressé au robots Lego EV3 en lançant des actions enseignement/recherche sur le développement logiciel dans cet environnement [5]. C'est aussi à cette époque que j'ai travaillé sur le projet Orange EDF que je mentionne dans la section 8.3.3 du chapitre 8 et rencontré Olivier Cardin lors d'une rencontre inter-équipe du LS2N en 2017. Nous avons démarré une collaboration en 2017 sur un article d'application du génie logiciel aux systèmes de production [AC17] puis encadré avec Olivier le stage de Mohammed El Amin Tébib au LS2N de janvier 2018 à juillet 2018. Ce stage financé par le projet Orange EDF portait sur l'apport de l'ingénierie des modèles dans une évolution continue du système de contrôle de production et a donné lieu à une publication [TAC18]. Nous avons poursuivi chaque année sur d'autres sujets du domaine. Je relate ces travaux dans la suite de ce chapitre après avoir introduit quelque peu les systèmes de production.

3. Ex LS2N
4. Professeur au LIRMM
5. `https://ev3.univ-nantes.fr/`

7.2 Les systèmes de production manufacturiers

L'objectif de cette section est d'introduire brièvement le vocabulaire du domaine de la production automatisée. Le lecteur trouvera dans [Car16] une introduction aux systèmes cyber-physiques (*Cyber-Physical Production Systems - CPS*) et une contribution à la conception, l'évaluation et l'implémentation de systèmes de production cyber-physiques (*Cyber-Physical Production Systems – CPPS*). *"Cyber-Physical Production Systems are systems of systems of autonomous and cooperative elements connecting with each other in situation dependent ways, on and across all levels of production, from processes through machines up to production and logistics networks, enhancing decision-making processes in real-time, response to unforeseen conditions and evolution along time"* [Car19]. Les bénéfices attendus de l'utilisation des CPPS sont l'optimisation des process de production (capacités, environnement, adaptation...) et des ressources (minimisation), la personnalisation des produits, le recentrage des process de production sur l'humain [Car19] Les systèmes cyber-physiques de production couvrent maintenant l'ensemble des domaines pour que la production soit intégrée au fonctionnement global de l'entreprise et couvre des services allant de la commande au client jusqu'à la logistique de livraison. Il s'agit d'un prérequis pour satisfaire le besoin de réactivité de la production à la demande. On trouvera dans [Goe13] l'intégration des systèmes de production dans l'informatique d'entreprise, avec notamment le problème de l'alignement BITA (*cf.* Chapitre 6) incluant la production. Cet ensemble constitue la vision **Industrie 4.0** dans laquelle l'informatique, au sens large, est réellement prégnante ou envahissante (*pervasive*) puisqu'elle est impliquée dans tous les processus. La FIGURE 7.1 met en évident la montée en complexité des systèmes de production en CPPS par intégration des technologies de l'information et de la communication.

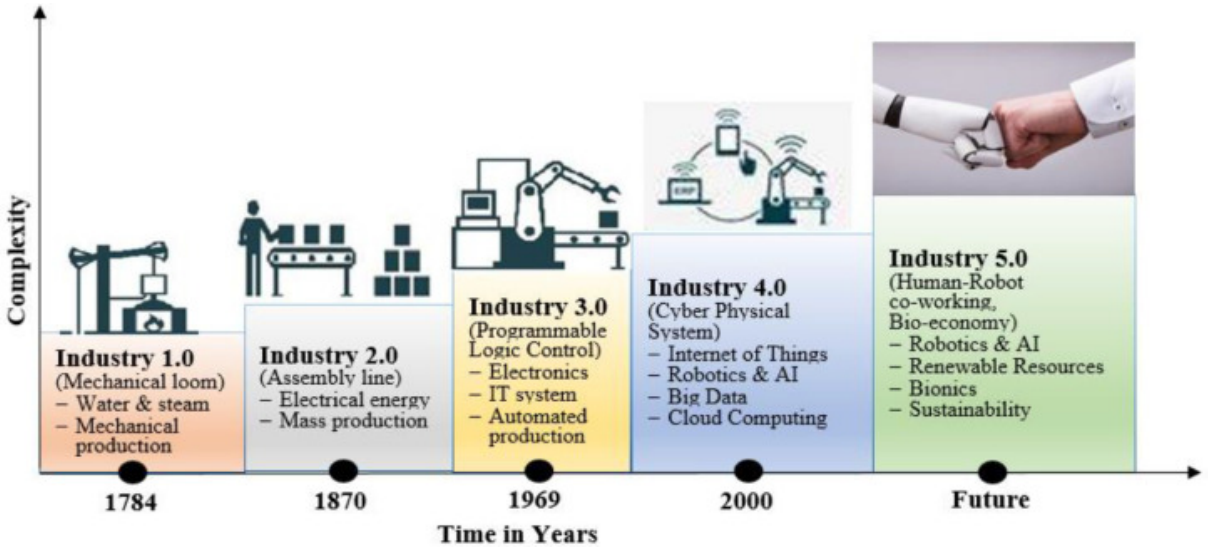


FIGURE 7.1 – Générations de systèmes de production [Mez+23]

L'informatique joue un rôle croissant dans les CPPS. *"Cyber-physical production systems (CPPS), relying on the latest and foreseeable further developments of computer*

science (CS), information and communication technologies (ICT), and manufacturing science and technology (MST) may lead to the 4th industrial revolution, frequently noted as Industrie 4.0" [Mon+16]. Les termes *digitalisation* et *virtualisation* désignent pour la partie informatique, que ce soit au niveau d'un atelier (*jumeau numérique*[6] ou *Digital twin*) ou d'une entreprise (cloud). La dualité physique/digitale d'un CPPS est illustrée dans la FIGURE 7.2.

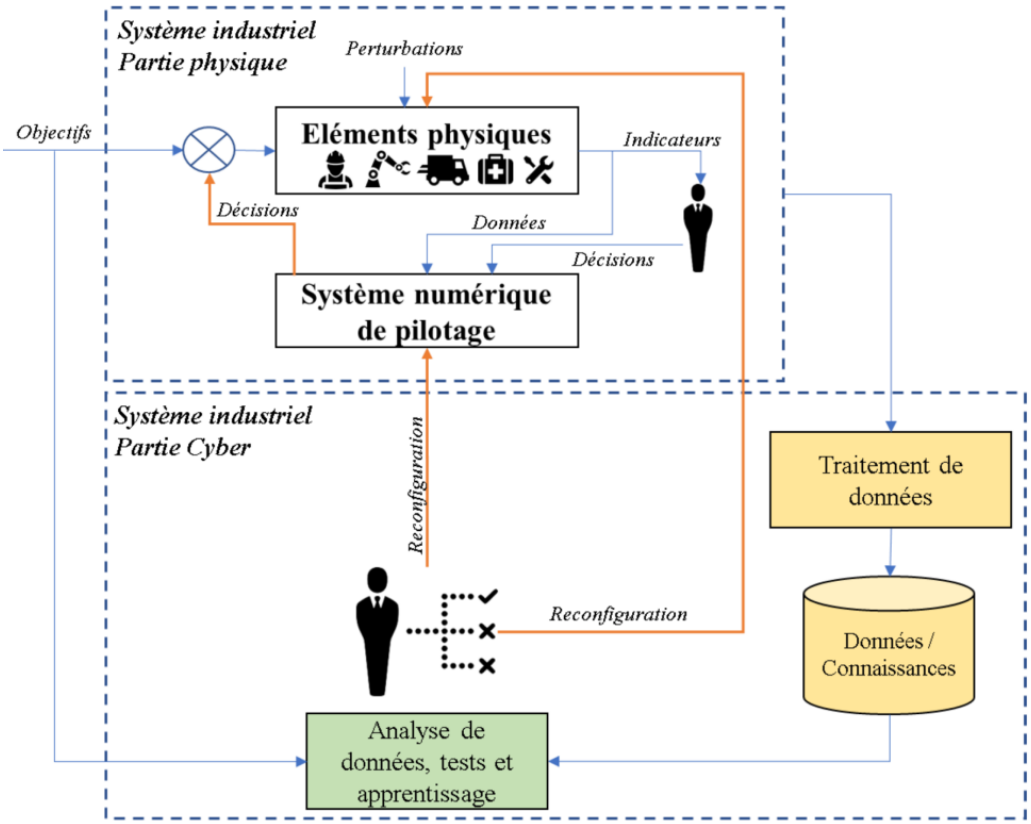


FIGURE 7.2 – Schématisation globale d'un système industriel cyberphysique [CDT23]

Smart production ou *smart manufacturingSmart manufacturing* sont aussi employés pour indiquer que le contrôle (digital) prend en compte les éléments du contexte. Bien qu'industrielle, la production considérée est manufacturière, c'est-à-dire de transformation des biens, y compris la réparation et l'installation d'équipements industriels, selon l'INSEE[7].

Il faut reconnaitre que la terminologie est foisonnante dans le domaine et le lexique n'est pas facile à appréhender, d'autant qu'il y a des traductions français-anglais qui génèrent des variantes lexicales. Les appellations ont évolué dans le temps.

Dans les années 80, on parlait de *Computer Integrated Manufacturing (CIM)*, nommés sur wikipedia en "fabrication intégrée par ordinateur" ou production intégrée par ordinateur, qu'il ne faut pas confondre avec fabrication assistée par ordinateur

6. Ce terme couvre de nombreux usages et de nombreuses interprétations [JM20]
7. `https://www.insee.fr/fr/metadonnees/definition/c1934`

(FAO) (en anglais, Computer-Aided Manufacturing ou CAM), qui vise à créer un programme de commande numérique ou bien avec Conception Assistée par Ordinateur (CAO), qui se focalise sur la modélisation géométrique et la simulation. L'acronyme CFAO regroupe les deux lorsque la modélisation permet de produire un fichier de commande. La FAO se focalise sur la machine (ressource) tandis que le CIMcouvre toute l'usine avec des ramifications dans l'ERP au delà de la seule production (on y trouve donc la CFAO, mais aussi la GPAO (*cf.* page 193), les équipements de stockage et de transport, etc.

Dans la littérature on trouve aussi beaucoup le terme "systèmes de contrôle" (*Control system*), associé ou pas à la production (*manufacturing*). Souvent il désigne l'organisation centrale (le cœur du pilotage) du système de production. Les quatre formes de base du contrôle sont illustrées dans la FIGURE 7.3 inspirée de [DBW91] : centralisé, hiérarchique pur, hiérarchique communiquant, hétérarchique..

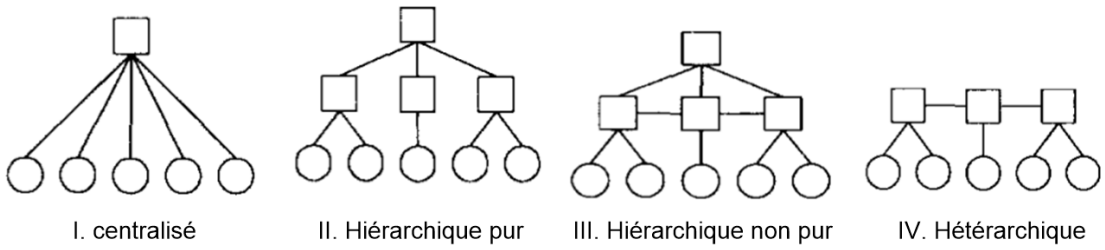


FIGURE 7.3 – Les différentes formes de contrôle de systèmes de production [DBW91]

Revenons sur le terme "*Manufacturing system*". Nous utiliserons en français le terme "*système de production*" ou plus rarement le terme "*système manufacturier*" moins courant [8] ; le terme CPPS couvre lui très large. Si "*Manufacturing system*" est rarement employé tel quel dans la littérature [9], ses sous-catégories le sont bien plus, notamment parce qu'elles sont associées aux révolutions dans l'organisation des usines et des ateliers pour suivre la demande des clients. Prenons quelques exemples inspirés de [Par+22 ; CF21].

1. La production artisanale (*Craft Manufacturing*), effectuée majoritairement grâce à des processus manuels, délivre des produits personnalisés à la demande du client mais avec un coût élevé.
2. Un *Dedicated Manufacturing system (DMS)* est conçu dans l'objectif de fabriquer ou d'assembler un produit particulier à une cadence élevée. Ils correspondent à la 2e révolution industrielle (*cf.* FIGURE 7.1).
3. Dans un *Flexible Manufacturing System (FMS)*, l'objectif est de produire la plus grande variété possible de produits et de s'adapter à la personnalisation de la demande. Cela est possible grâce à la commande numérique ; l'informatique est le levier de la 3e révolution industrielle (*cf.* FIGURE 7.1).

8. On le trouve dans la thèse récente d'Alexandre Parant [Par+22]. La traduction de "*Manufacturing system*" est aussi système de fabrication.

9. Il n'est pas défini dans wikipedia par exemple, mais c'est le titre du livre [Chr13].

4. Un *Reconfigurable Manufacturing system (RMS)* permet non seulement de s'adapter à la varité de la demande mais aussi aux aléas de production (variabilité de l'offre) et de consommation (variabilité de la demande). Les parties physiques et logicielles sont reconfigurables pour ajuster la production aux besoins réels, on est passé dans le monde cyberphysique (CPS) avec une influence croisée du logiciel et du matériel mais aussi des systèmes de gestion de production et des systèmes d'information, c'est le monde complexe de l'industrie 4.0 (*cf.* FIGURE 7.1).

La FIGURE 7.4 montre les différentes générations de systèmes production en fonction de la demande du marché. La période "mass production" correspond aux DMS, celle de "mass customization" correspond aux FMS et la période après 2000 correspond aux RMS. On passe d'une variété artisanale à une variété industrielle en passant par la standardisation, la boucle est bouclée.

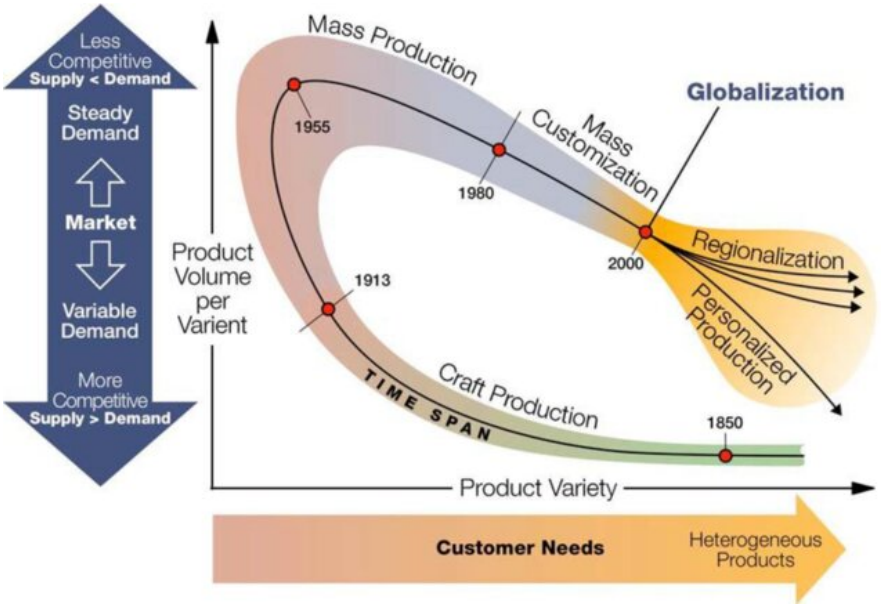


FIGURE 7.4 – Evolution des systèmes de productions [Kor10]

La spécificité des RMS est d'être reconfigurable sur la partie matérielle et/ou logicielle (*cf.* FIGURE 7.5). Ce n'est pas suffisant, les systèmes de contrôle doivent eux-mêmes être modulaires et ouverts pour s'adapter. L'objectif ultime de RMS est d'utiliser une approche systémique dans la conception du processus de fabrication pour la reconfiguration simultanée de (1) l'ensemble du système, (2) du matériel de la machine et (3) du logiciel de contrôle. Le paradigme RMS permet aux reconfigurations d'atteindre une évolutivité rentable [Kor+99].

	Fixed Machine Hardware	**Reconfigurable Hardware**
No software	Manual machines, Dedicated mfg. lines (DML)	—
Fixed control software	CNC machines, robots, Flexible mfg. systems	Modular CNC machines
Reconfigurable software	Modular, open-architecture controller	RMS
System configuration rules & economic modeling		

FIGURE 7.5 – Axes de reconfiguration [Kor+99]

Dans [Bi+08] on trouve un scénario pour de tels systèmes qui utilisent des matières premières comme entrées et fournissent des produits comme sorties (*cf.* FIGURE 7.6), on peut reconfigurer le logiciel et le matériel, comme signalé par Koren, mais aussi modifier les machines, les fonctionnalités, le processus et le transport, etc. Les caractéristiques

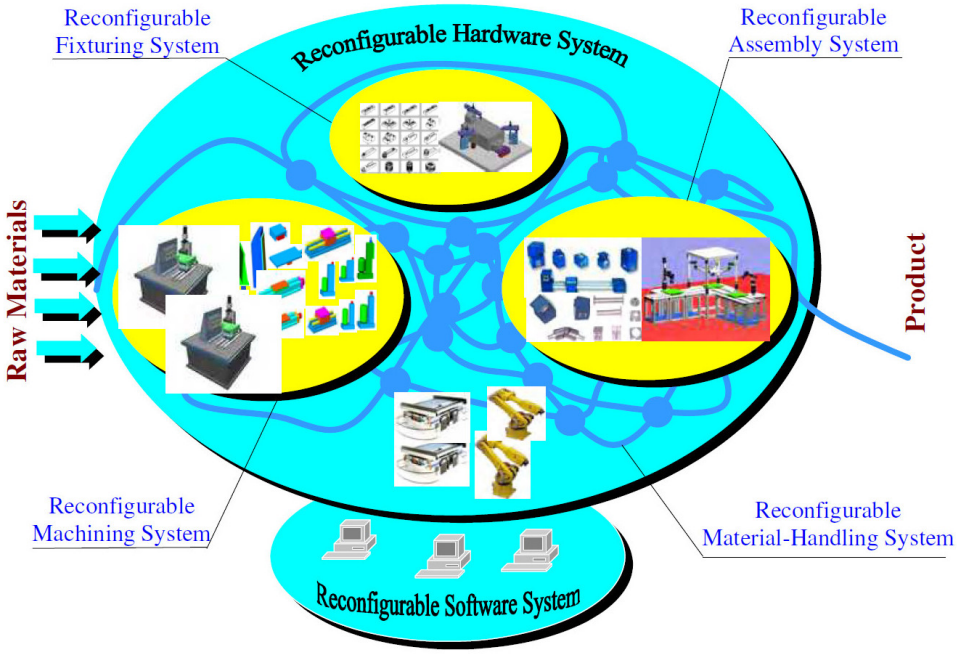


FIGURE 7.6 – Scénarios de reconfiguration des RMS [Kor+99]

du RMS incluent la "modularité", l'"évolutivité", l'"intégrabilité", la "convertibilité" et la "diagnosticabilité" [Bi+08]. La modularité implique que les éléments logiciels et matériels soient modularisés. L'évolutivité signifie que le système est évolutif en termes de volume de produit. L'intégrabilité signifie que le système et les composants du système sont conçus à la fois pour une intégration facile et pour l'introduction future de nouvelles technologies. La convertibilité permet un changement rapide entre les produits existants et une adaptabilité rapide du système pour les produits futurs. La diagnosticabilité est capable d'identifier rapidement les sources des problèmes de qualité et de fiabilité qui surviennent dans les grands systèmes.

La FIGURE 7.7 propose une classification des caractéristiques précédentes des RMS. Ces caractéristiques permettent d'identifier à quelles fins le système doit être reconfiguré. Dans la plupart des cas, un besoin de reconfiguration intervient lorsque le système doit s'adapter aux variations de la demande et/ou de l'environnement. Cette adaptation peut concerner soit le changement de la fonction d'une machine, soit l'ajout d'un nouveau produit, soit la modification d'un produit ou alors l'ajustement du volume de production.

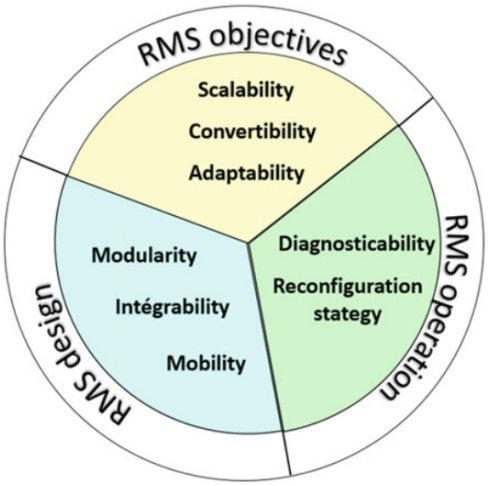


FIGURE 7.7 – Classification des caractéristiques des RMS [CF21]

On trouve aussi d'autres dérivés de "*Manufacturing system*", tels que :

— *Holonic Manufacturing System (HMS)* désigne une organisation modulaire de type acteur, donc collaborative et distribuée (Systèmes Multi-Agents Holoniques - SMAH).

La particularité est une structuration hiérarchique basée sur la notion d'agrégation (composition). Un holon peut être composé d'autres holons. Un des modèles de référence est PROSA [Van+98], basé sur le triplet *order holons, product holons, and resource holons*. L'intérêt des HMS est leur faculté à s'auto-organiser.

— *Intelligent Manufacturing System (IMS)* est une appellation générique pour des systèmes intégrant humains, machines et processus d'une manière optimisée. C'est aussi le nom d'un programme industriel [10]. Dans les IMS, les capteurs jouent un rôle important pour prendre des informations sur le contexte et donc réagir au mieux aux aléas de la production. *Smart Manufacturing System (SMS)* (*cf.* page 196) est une variante des IMS [TO23].

Les catégories ne sont pas exclusives car les sources sont variées, ainsi que le montre *"Manufacturing in the Age of Human-Centric and Sustainable Industry 5.0 : Application to Holonic, Flexible, Reconfigurable and Smart Manufacturing Systems"* le titre de [TO23]. Selon [Kus18] les SM se basent sur 6 piliers (*manufacturing technology and processes, materials, data, predictive engineering, sustainability and resource sharing and networking*) et les technologies numériques couvrent les *CPS, internet of things, cloud computing, service-oriented computing, artificial intelligence and data science*. Les tendances du SM y sont présentées en dix conjectures autour de ces technologies. On retiendra *digitalisation, modelling, Enterprise dichotomy, connectivity and interoperability, Standardisation and collaboration, Cybersecurity and safety*, liées à nos thématiques.

Nous avons vu jusqu'ici les systèmes de production dans leur ensemble, mais les travaux de recherche concernent aussi les briques qui constituent ces systèmes. La norme ISA 95 [ISA10] (*cf.* Section 7.3.1) met en évidence ces briques. La partie gauche de la FIGURE 7.8 [11] indique les composants et standards impliqués dans les couches 2 [*Super-*

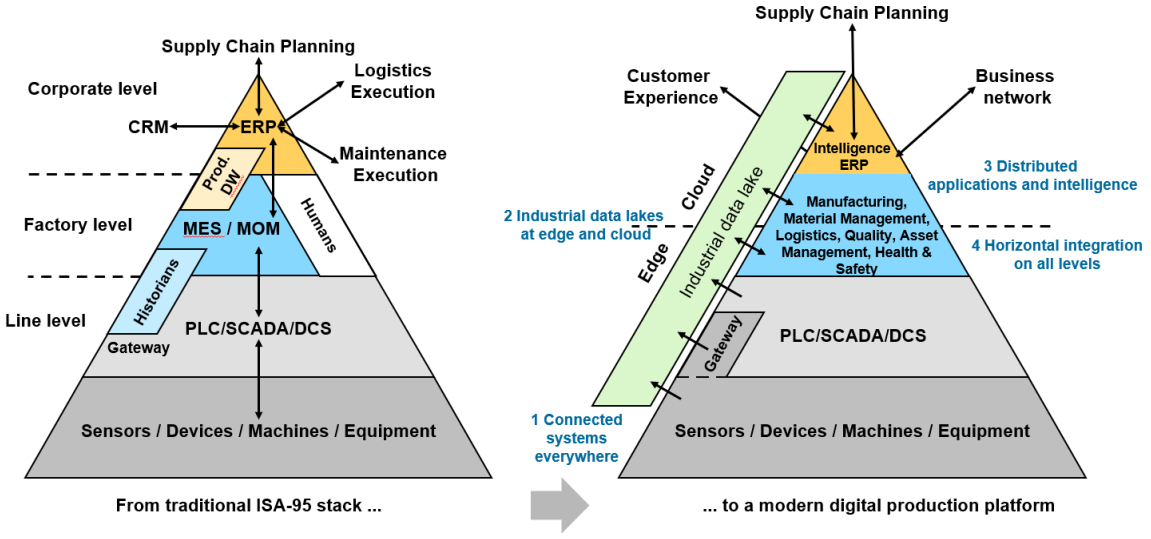


FIGURE 7.8 – Niveaux et composants (source : SAP[11])

10. https://www.ims.org/
11. https://blogs.sap.com/2023/04/05/industry-4.0-and-the-standard-isa-95-the-pathway-to-revolutionizing-the-energy-industry/

visory Control And Data Acquisition (SCADA), Programmable Logic Controllers (PLC), Distributed Control Systems (DCS) and Building Automation Systems (BAS)] et 3 [*Manufacturing Execution Systems (MES), Manufacturing Operations Management (MOM)*]. Les échelles de temps de réponse varient d'un niveau à l'autre : µs ou ms (niv 0), sec (niv 1), min (niv 2), heure (niv 3), jour/mois (niv 4).

Dans les travaux relatés dans ce chapitre, nous nous intéressons principalement à la conception des HMS et aux RMS et donc au niveau 3 (atelier, usine) qui gère le pilotage des composants du niveau 2 (ligne de production). Nous sommes aussi guidés par la modularité et l'intégrabilité de ces systèmes ainsi que par l'applicabilité des propositions (*cf.* page 206). Dans le projet ANR RODIC (*cf.* page 221), un lien est fait avec les ERP du niveau 4 par les (KPI). On retrouve alors le problème de l'alignement BITA (*cf.* Chapitre 6) processus de production dans les processus de l'entreprise ; il est décrit dans [Goe13].

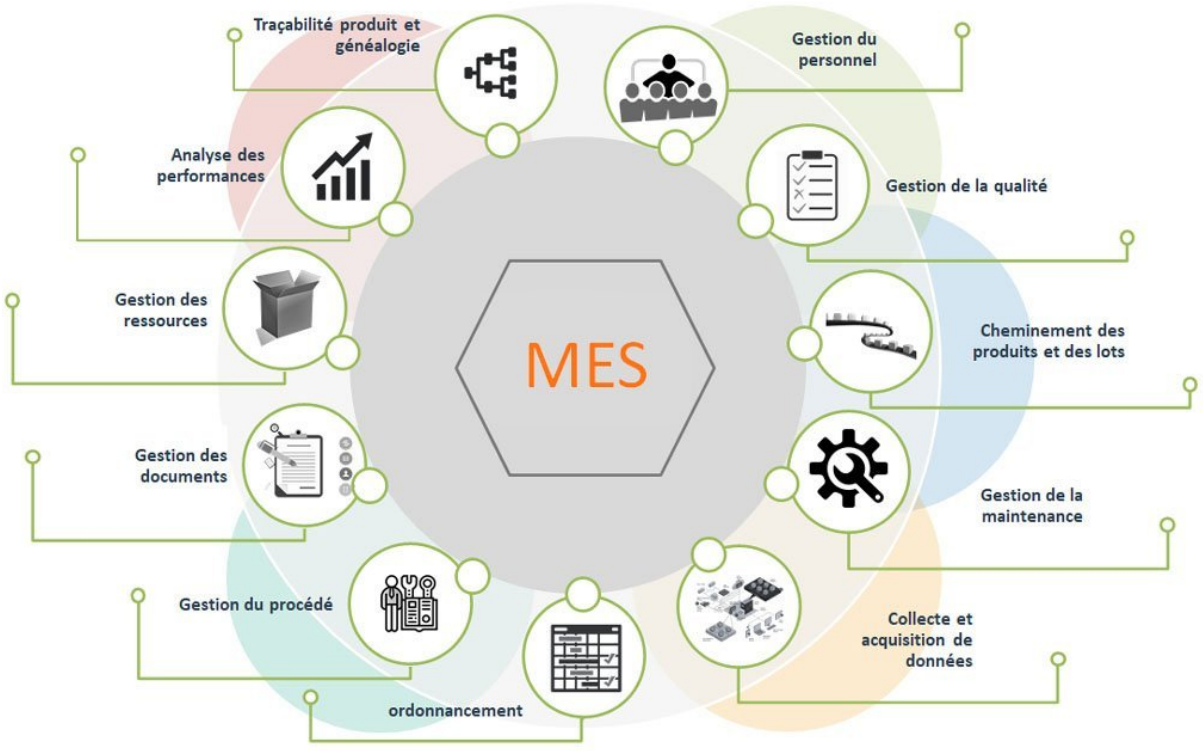


FIGURE 7.9 – Les 11 fonctions principales d'un MES[12]

Les MES ont été définis par l'assocation MESA International (Manufacturing Enterprise Solutions Association). En 2000, la norme ANSI/ISA-95 fusionne ce modèle avec le modèle de référence Purdue (PRM). *"Les systèmes d'exécution de la fabrication ou "Manufacturing Execution Systems" (MES) assurant l'intégration entre les ERP et les systèmes de production automatisés par la mise en œuvre de fonctions de gestion de l'information technique et par un effort de standardisation de leurs services et systèmes d'informations respectifs. Les principales fonctions de ces systèmes sont l'acquisition et la collecte de*

données en temps réel, la surveillance de la production, la planification et le suivi de la production, l'analyse de la performance." [Goe13] D'un point de vue simplifié de 2007 dans [Kle07], le MES est le niveau intermédiaire entre l'ERP de l'entreprise et les automatismes et correspond à la gestion de production. La FIGURE 7.9 [12] met en évidence les fonctions du MES.

Les systèmes MOM répondent aux fonctionnalités de production critiques suivantes [13] : qualité, sécurité, fiabilité, efficacité et conformité réglementaire. La norme ISA-95 Partie 3 définit les activités qui se produisent dans les systèmes de gestion des opérations de production tels que : production, maintenance, qualité, manutention et gestion de stocks, support. Les activités support incluent la gestion de la sécurité, des informations, de la configuration, de la documentation, de la conformité réglementaire et des incidents/écarts. Les systèmes MOM d'aujourd'hui permettent aux fabricants de standardiser et d'optimiser les processus dans toute l'entreprise. La FIGURE 7.10 [14] met en évidence les fonctions du MOM. Le modèle d'activité MOM sert de pont entre les processus métier de haut niveau définis au niveau de l'entreprise (niveau 4) et les actions spécifiques menées en atelier au niveau de la gestion des opérations de fabrication (niveau 3).

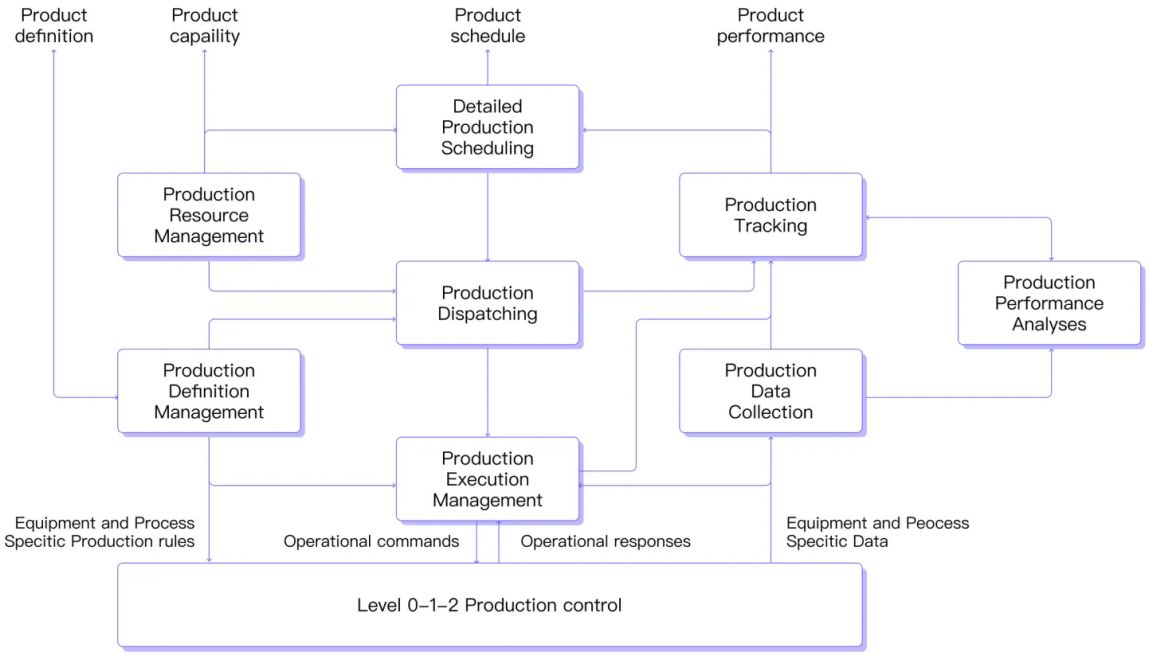


FIGURE 7.10 – Fonctions du MOM[14]

L'atelier (*Shop floor*) est le lieu de la production manufacturière des produits, en distinction avec les activités administratives de gestion. On y trouve des ressources (machines, opérateurs, transporteurs...). Le contrôle d'atelier est l'une des fonctions du contrôle de production (*manufacturing control*) qui surveille le déroulement des activités de production *e.g.* lorsque le produit est traité, assemblé, inspecté, etc. Il gère aussi les stocks ma-

12. https://cqpm.fr/le-pilotage-de-la-production-le-logiciel-mes/
13. https://www.plm.automation.siemens.com/global/en/our-story/glossary/isa-95-framework-and-layers/53244
14. https://emqx.medium.com/exploring-isa95-standards-in-manufacturing-1325093a0f06

nuels (stockage) ou automatisés (magasin). Il se situe au bas du niveau 3 dans la norme ISA-95 comme le montre la FIGURE 7.11, qui précise aussi de manière utile l'évolution dans le temps du contrôle Shop-Floor *Modern SFC = MES + SCADA* [TP13].

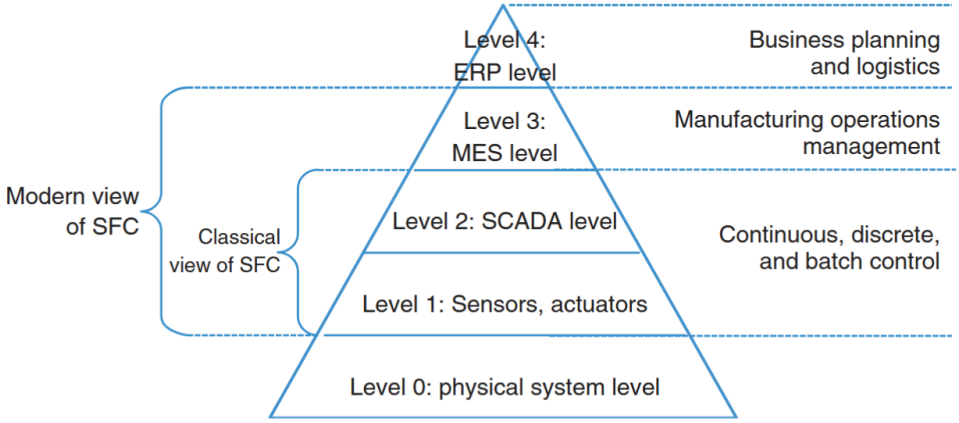


FIGURE 7.11 – Evolution du contrôle Shop-Floor selon [TP13]

L'ordonnancement des activités est une fonction clé et utilise des algorithmes de recherche opérationnelle Les modèles les plus connus sont ceux d'une machine unique, de machines parallèles, d'un atelier à cheminement unique (*Flow Shop*) ou d'un atelier à cheminement multiple (*Job Shop*). Cet aspect a été développé dans le projet CIM-Anjou (*cf.* page 194).

7.3 Contributions aux systèmes de production manufacturiers

Nous balayons dans cette section quelques contributions au domaine : la modélisation pour les architectures de référence en Section 7.3.1, l'apport du génie logiciel dans la construction d'un MES en Section 7.3.2, la gestion de l'hétérogénéité dans l'interface niveau 2-3 en Section 7.3.3 et les RMS avec le projet Rodic en Section 7.4.

7.3.1 Architectures de référence et leur modélisation

Les architectures de référence et les standards sont incontournables pour la mise en œuvre des systèmes de production en entreprise, compte-tenu de la complexité et l'hétérogénéité des systèmes. Nos contributions aux *architectures de référence* (RA) comprennent (I) une étude systématique sur l'applicabilité des RAs, (II) des modèles de RAs qui définissent des règles pour les mises en œuvre de systèmes manufacturiers. Détaillons.

I) Les architectures de référence et leur applicabilité

Il existe une multitude solutions propriétaires, standards de facto et normes pour les systèmes de production. Dans [Kai+23], nous avons recensé et analysé 78 propositions de modèles qualifiés d'**architectures de référence (RA)**.

Les premières propositions ont émergé dans les années 80 avec *Computer Integrated Manufacturing (CIM)* (*cf.* page 196) et notamment deux modèles qui couvrent toute l'entreprise (PERA et CIMOSA) et inspirent la norme ANSI/ISA-95 (IEC 62264). A cette époque, la performance s'appuie sur l'efficacité via les travaux sur l'optimisation de l'ordonnancement ds tâches. Les deux décennies suivantes, qui rappelont-le correspondent aux systèmes flexibles, ont fait émerger des standards tels que les systèmes holoniques (PROSA, ADACOR, HBCA...). Ces solutions distribuées rendent les systèmes pus résilients aux aléas. Ils facilitent aussi l'intégration des systèmes et le passage à l'échelle. Au cours des 25 dernières années, une série de nouvelles technologies ont été développées pour contribuer à la conception de systèmes de production où l'information joue un rôle fondamental. Elles incluent par exemple les services, les jumeaux numériques (*digital twins*), les agents ou les *blockchains*. En particulier, les services jouent un rôle clé pour l'intégration de systèmes très hétérogènes par nature (variété de fournisseurs et d'interfaces).

La norme ISA-95 [ISA10] est une pierre d'angle dans les architectures de référence. Elle traite les différents domaines opérationnels qui supportent la production (logistique, qualité, maintenance) pour assurer la continuité des processus à travers les applications concernées : ERP (Enterprise resource planning), systèmes de contrôle (SCADA, API/-PLC, SNCC/DCS...), MES (Manufacturing execution systems), mais aussi LIMS (Laboratory Information Management Systems), WES (Warehouse Execution Systems, LES (Logistics Execution Systems), CMMS (Computerized Maintenance Management System)... ISA-95 prend en compte les interactions du développement de l'entreprise avec le domaine opérationnel (PLM - Product Lifecycle Management).

ISA-95 propose une intégration verticale de la production et de l'entreprise en 5 niveaux (*cf.* Figure 7.12). Le niveau 0 définit les processus physiques réels (*Production process*). Le niveau 1 définit les activités impliquées dans la détection et la manipulation des processus physiques (*Sensing and Manipulating*). Le niveau 2 définit les activités de surveillance et de contrôle des processus physiques (*Monitoring & Supervising*). Le niveau 3 définit les activités du flux de travail pour produire les produits finaux souhaités (*Manufacturing Operations Management and Control*, *Dispatching production, detailed production schduling, reliability assurance, etc.*). Le niveau 4 définit les activités commerciales nécessaires à la gestion d'une opération de fabrication (*Business Planning & logistics*, *Plant Production Scheduling*, *Operational Management*, etc.).

Cette structuration hiérarchique va généralement de pair avec l'organisation géographique : de la ressource d'un poste de travail à tous les sites de l'entreprise. Les niveaux bas gèrent les ressources avec des modes de fonctionnement variés (continu, discret, répétition, batch...). Concrètement, la norme se focalise uniquement sur l'interface entre les niveaux 3 et 4. Elle va aussi de pair avec les composants et standards impliqués dans les couches tels que MES, MOM, PLC, SCADA (*cf.* Figure 7.8). Noter que la partie droite de la Figure 7.8 met en évidence des éléments pertinents pour normes récentes décrites ci-après.

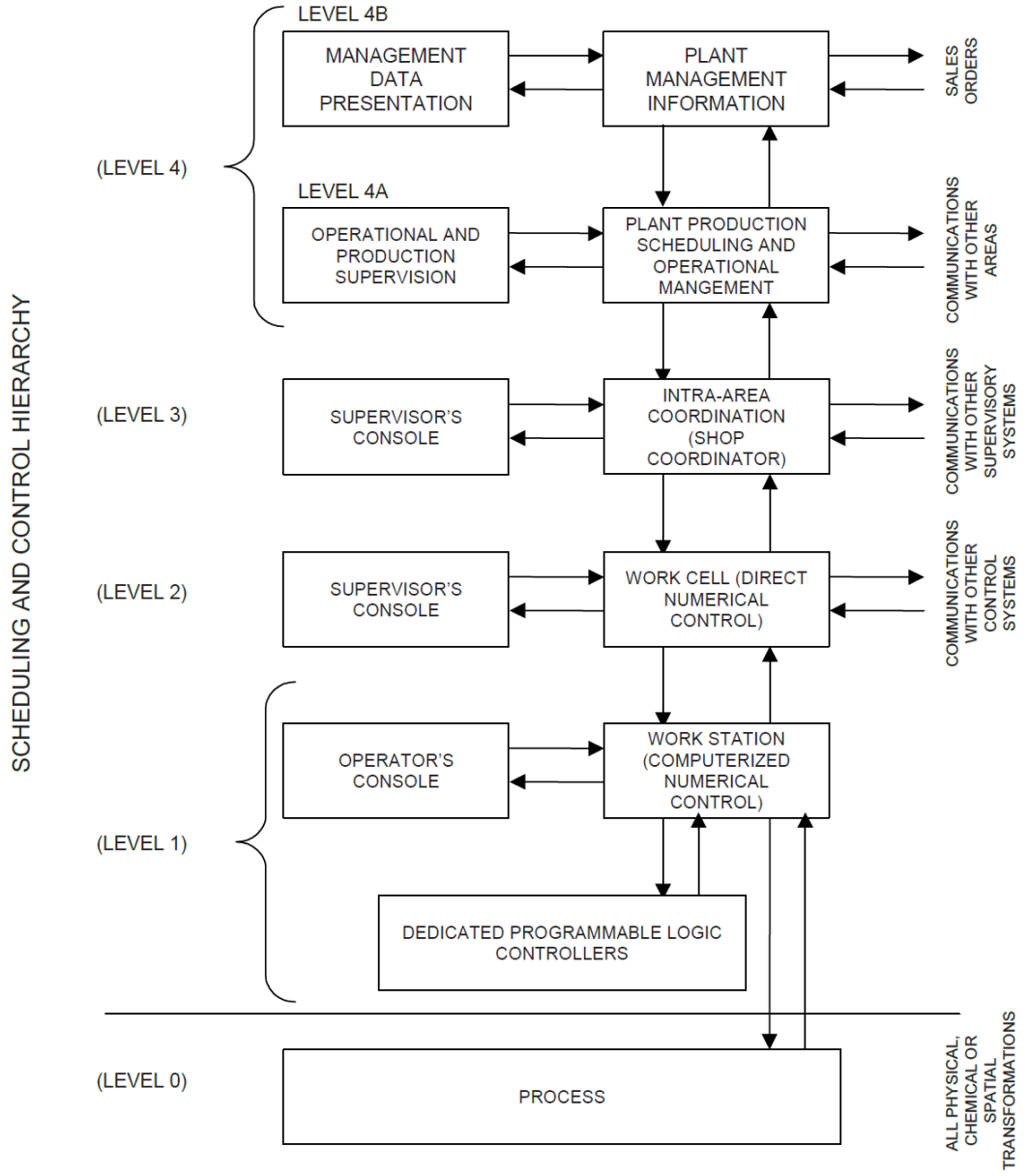


FIGURE 7.12 – Les niveaux d'intégration verticale [ISA10]

La norme ISA 95 est à la base de NIST SME (*Smart Factory Ecosystem*) et RAMI 4.0 (Industry 4.0) qui visent en plus une intégration horizontale en interconnectant divers systèmes d'information, par exemple, grâce à une orientation service. L'écosystème SME reprend les normes pertinentes pour les systèmes de production numérique. Ce modèle sépare fonctions commerciales, de production et de produit à travers des cycles de vie dédiés et la décentralisation.

Dans [Kai+23], nous avons utilisé le terme générique "***digital manufacturing***" pour désigner l'ensemble des domaines d'application de l'informatique pour la production (*manufacturing process*). Cinq domaines sont considérés *Smart manufacturing & Industry 4.0*, *Internet of Things (IoT)*, *Manufacturing Control*, *Cyber-Physical Systems (CPS)*. Les progiciels de gestion (ERP, SCM..) se trouvent dans la première catégorie. La FIGURE 7.13

propose une classification de ces domaines avec des architectures de référence associées.

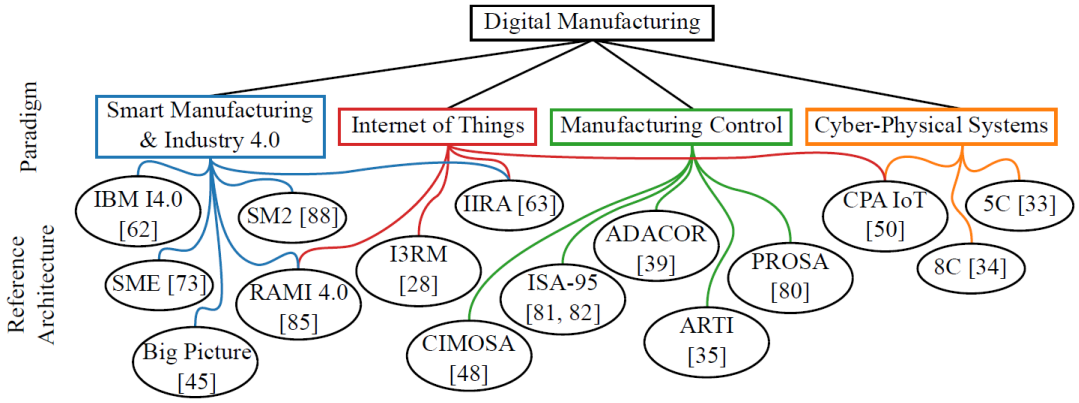


FIGURE 7.13 – Relations entre RA et domaines du *digital manufacturing* [Kai+23]

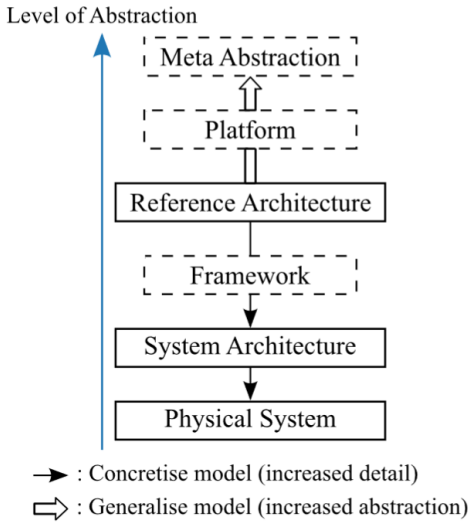


FIGURE 7.14 – Niveaux de modélisation [Kai+23]

Dans une perspective de modélisation, nous avons classé les approches en trois niveaux (1) meta architecture de référence comprenant des standards et normes assez génériques, (2) model architecture système avec des principes et des éléments de conception, et (3) system système physique avec une implantation générique. Nous avons aussi ajouté d'autres types de modèles présents dans la littérature : les plateformes, qui combinent plusieurs architectures, et les frameworks qui mettent en œuvre des architectures de référence. La FIGURE 7.14 organise ces formats par niveaux d'abstraction. Les méta-modèles permettent de définir les architectures de références, si un même méta-modèle est utilisé les RA deviennent comparables voir combinables par transformation de modèles. La figure 4 de [Kai+23] montre les relations de précédence entre les RA. Notons que la norme ISA 95 est plus intéressante en pratique que les évolutions récentes NIST SME et RAMI 4.0 car elle propose des modèles détaillés sous forme de diagrammes UML, c'est aussi le cas d'architectures de référence comme Prosa ou Adacor. Ce point est un des critères de l'applicabilité des architectures de référence, une des contributions de [Kai+23].

L'applicabilité d'une RA est sa capacité à être adoptée et appliquée dans la pratique. Nous définissons deux mesures qualitatives pour évaluer l'applicabilité : (1) alternatives de développement, indiquant la simplicité des directives de conception fournies par l'architecture de référence et (2) variété des systèmes résultants, décrivant dans quelle mesure l'architecture de référence peut être appliquée. Nous avons classé les 36 RAs sélectionnées (sur 78 approches recensées pour l'étude systématique) en dix catégories (*cf.* FIGURE 7.15) et classé ces catégories sur une échelle d'applicabilité : Low (E, H, I), D, Medium (A, B, F), High (C, G, J). Les RA notées en gras sont des normes ; celles en italique sont des

	Network-based		Layered-based	
	Operation	Enterprise	Operation	Enterprise
Abstract	ARTI [35] *PROSA* [80] QHAR [83] A	-	5C [33] BCPS [46] Cisco IoT [47] CPS-A IM [49] E	8C [34] ARADF [41] *CIMOSA* [48] DTaaS [53] IVRA [64] iI&I [13] RMSF [6] SLA [89] H
Constrained	ADACOR² [36] *ADACOR* [39] HCBA [60] B	NIST SOA [75] D	AIOTI HLA [42] **ITU-IoT** [66] **IoT RA** [67] RAMEC [84] WSO2 IoT [95] F	**Big Picture** [45] *IIRA* [63] I3RM [28] ISMA [65] KSTEP [4] SME [73] **PERA/ISA-95** [81, 82] **RAMI 4.0** [85] **SM2** [88] I
Concrete	WoT [94] *ADACOR/JADE** [104] *PROSA/Erlang** [105] C	-	Shoestring [7, 52] G	IBM Industry 4.0 [62] *IIRA/Thingsboard** [108] J

FIGURE 7.15 – Classification des architectures de référence en 10 catégories [Kai+23]

standards *de facto* pour une approche ou une technologie spécifique. Sans surprise, les RAs qui couvrent large laissent les développeurs face aux problèmes d'intégration. Néanmoins les RAs, lorsqu'elles sont bien structurées permettent de définir des règles de compatibilité. On peut le voir, le passage d'une vision très abstraite indiquant ce qu'on imagine nécessaire, à un système opérationnel intégré est difficilement envisageable. Par exemple, RAMI 4.0 propose une architecture cubique intégratrice mais les détails s'appuient sur d'autres RA. Il faut raffiner ces architectures jusqu'à avoir des modèles "blueprint" qui donnent vraiment les clés pour concevoir un système de production et plus prosaïquement la partie centrale que constitue le MES. Une piste est donc de formaliser les standards de manière modulaire avec des RAs spécifiques à leur domaine, des interfaces claires et réalisable par des architectures de référence ouvertes.

II) Modélisation dans les architectures de référence

En réponse au besoin énoncé dans la section précédente, nous avons étudié la modélisation d'architectures de référence dans [AC22]. Nous avons mis en évidence les risques liés à des modélisations trop floues qui laissent le concepteur devant de multiples problèmes à résoudre. Nous avons illustré de telles erreurs sur des architectures de référence de type *Holonic Manufacturing System (HMS)* (*cf.* page 199) dont la particularité est une structuration hiérarchique basée sur la notion d'agrégation (ou de composition). Un des modèles

de référence est PROSA [Van+98], basé sur le triplet *order holons, product holons, and resource holons*. Nous avons alors montré l'intérêt de représenter les architectures de référence par des modèles modulaires rigoureux inspirés des patrons de conception. (*cf.* Section 1.2.8 du Chapitre 1)

Prenons l'exemple de la description des produits utilisés dans un processus de fabrication. C'est un exemple typique d'agrégation car les produits de base proposent une structure et des services clairement différents de ceux des produits composites. Les modèles existants ne distinguent pas (i) les instances de produits des types de produits (les instances ont des agrégations plus simples), (ii) la cohésion des classifications, et (iii) la cohésion avec les autres concepts, notamment lors du raffinement des concepts. La Figure 7.16 montre une structuration en patron qui met en évidence ces variations. Pour être pragmatique nous avons également ajouté la notion de container[15] pour stocker les instances de produits (et non les types de produits...). Nous avons étendu les conteneurs au cas de produits fragiles afin d'illustrer concrètement la problématique de la spécialisation des schémas. Un schéma est une structure sur laquelle on peut appliquer l'héritage par exemple. Cela nécessite des macro-définitions : les flèches bleues peuvent être calculées à partir de cet héritage de schémas. Figure 7.16 montre bien la complexité croissante de la combinaison de motifs et l'intérêt des raccourcis de modélisation de notation mais aussi le besoin de règles de combinaison écrites sous forme d'instructions formelles par exemple en OCL.

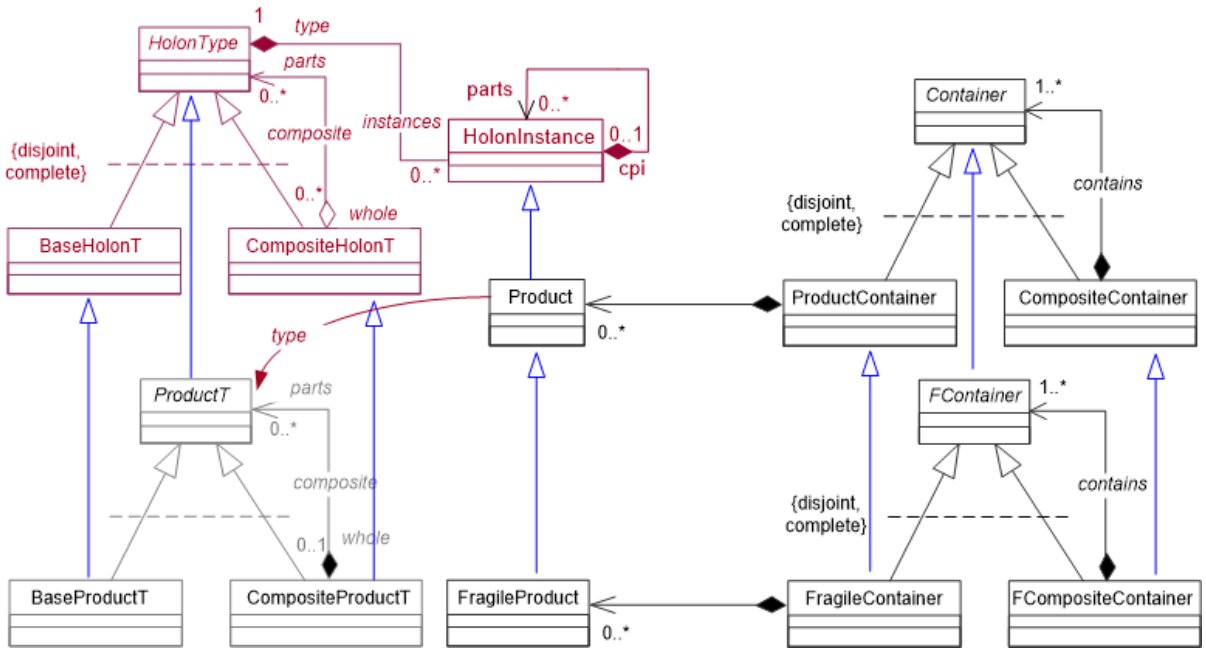


Figure 7.16 – Patrons d'aggrégation de produits [AC22]

Nous avons donc proposé des règles méthodologiques de modélisation et un catalogue de patrons de conception pour de tels systèmes [AC22]

Dans [Der+23], nous avons mis en œuvre ces principes pour des architectures de

15. Il est considéré comme une ressource de stockage.

contrôle distribuées hiérachiquement en réseaux d'un point de vue physique et digital. Ce type d'organisation implique un système de décision à chaque niveau de manière à être modulaire et évolutive tant sur les demandes issues du système de décision que de l'organisation du réseau pour faire face aux aléas. Pour répondre à ce besoin, nous avons proposé le Framework GARCIA (*Generic Aggregation model for Reconfigurable holonIc Control Architecture*). Le *framework* GARCIA s'applique de préférence sur des entités cyber-physiques en réseau. Dans ce contexte, deux types de réseaux coexistent : le réseau virtuel, reliant les cyber-entités représentant les dispositifs physiques (ressources, produits, ...), ainsi que le réseau physique, qui est le réseau de communication reliant les équipements physiques (appareils physiques) qui peuvent communiquer directement entre eux. La FIGURE 7.17 montre une structuration en patron qui met en évidence ces variations. Ce modèle de type jumeau numérique est divisé en trois domaines : numérique, physique et communication (les régions délimitées sur la FIGURE 7.17).

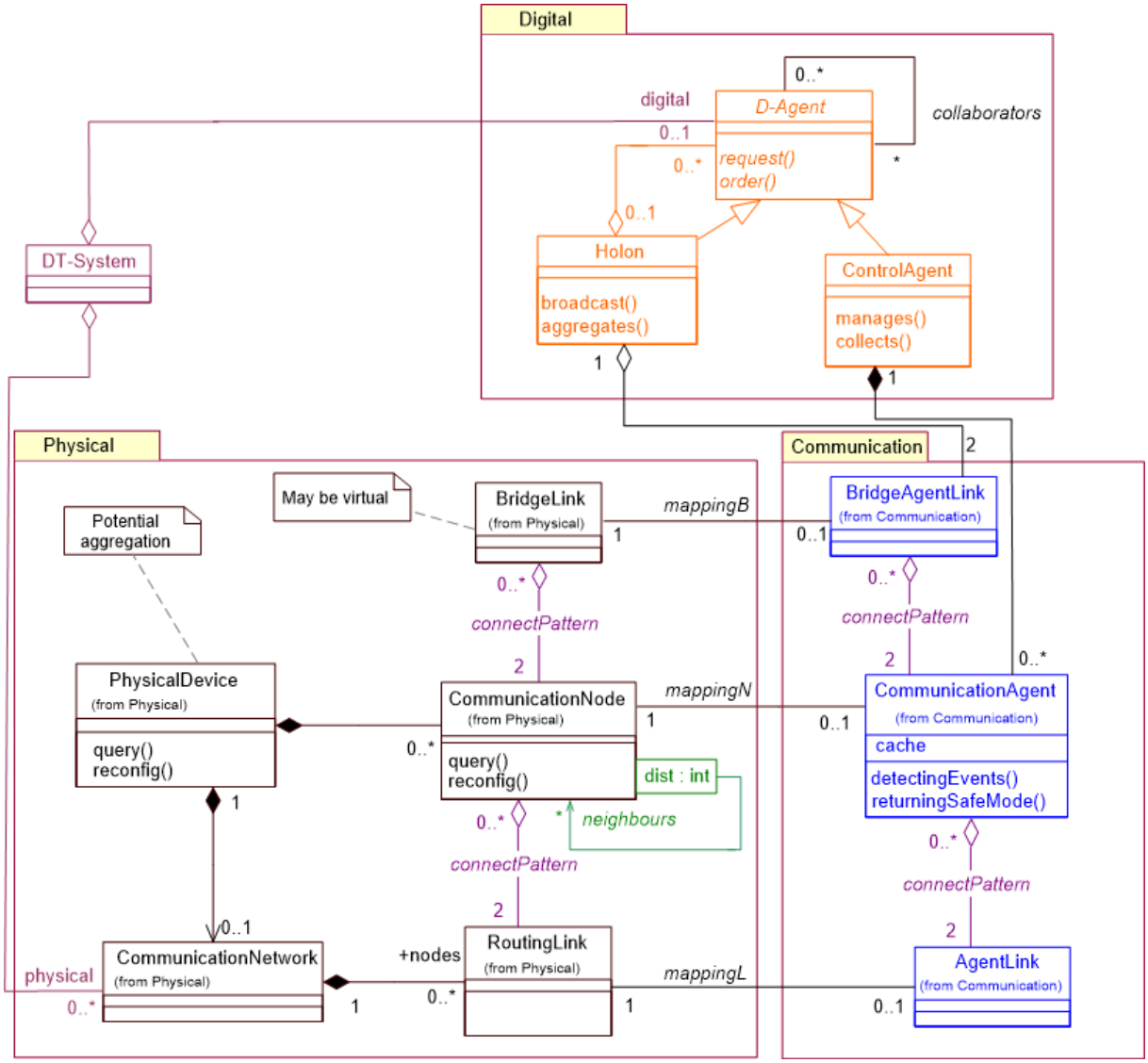


FIGURE 7.17 – Modèle de l'architecture de contrôle [Der+23]

— Le domaine physique regroupe tous les objets pouvant contenir des entités intel-

ligentes. Un PhysicalDevice décrit un objet capable de détecter et de communiquer des données. Il peut s'agir d'un système complet (flotte d'objets) ou d'un objet unique dans lequel sont insérés des capteurs (produit intelligent). Un nœud CommunicatingNode décrit les appareils situés à l'intérieur du dispositif physique, générant des données et prenant des décisions. Il peut s'agir d'une unité informatique, d'un périphérique réseau ou d'un nœud de capteur. Un réseau de communication comprend un ensemble de nœud de communication liés qui peuvent échanger des données/informations ou discuter des décisions à prendre.

— Le domaine digital regroupe toutes les entités décisionnelles. Il s'agit d'un domaine complexe qui peut être lié au système d'information de l'entreprise, par ex. stockés dans un système basé sur le cloud. Nous montrons ici uniquement les classes de bas niveau de la partie numérique.

— Le domaine de la communication regroupe toutes les entités représentant l'interface entre le domaine physique et le domaine numérique. Il permet de gérer l'indépendance matériel/logiciel. Un CommunicationAgent est un jumeau numérique (*digital twin*) de chaque CommunicatingNode existant en tant que PhysicalDevice. Il surveille et contrôle son jumeau.

Nous avons appliqué GARCIA à un exemple d'atelier de production (*cf.* FIGURE 7.18). La principale caractéristique de cet exemple est la nécessité pour le système de reconfigurer dynamiquement son contrôle en raison de son évolution. Le système global est constitué de deux sous-systèmes de 5 machines, capables d'exécuter diverses activités de fabrication. Dans cette étude de cas, l'atelier est censé être reconfigurable, dans le sens où chaque machine et chaque chariot peuvent être transférés d'un sous-système à un autre en fonction des exigences de production. Chaque élément de l'atelier est ainsi équipé

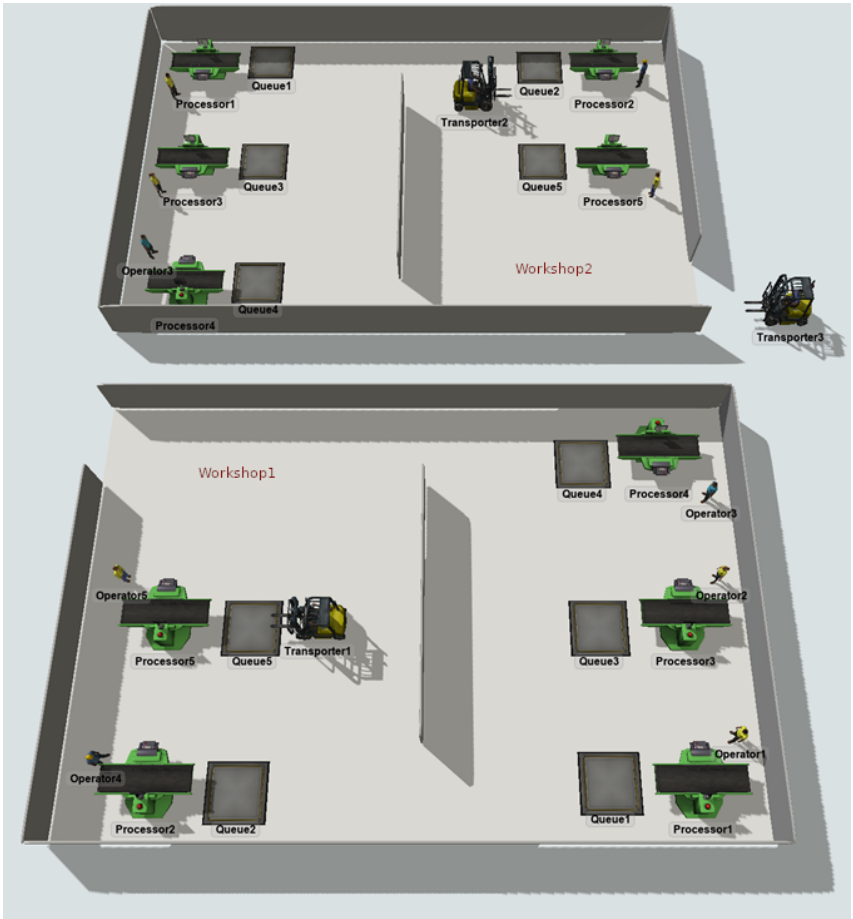


FIGURE 7.18 – Atelier de production [Der+23]

de dispositifs de communication, connectés à un réseau de communication sans fil. A l'intérieur de ces sous-systèmes, le transfert des produits est effectué par des chariots autonomes, un dans le premier sous-système, deux dans le second. Les chariots reçoivent en temps réel les missions de transport calculées par le système de contrôle, et envoient régulièrement en retour divers accusés de réception au système de contrôle.

GARCIA est "concrétisé" pour l'étude de cas en ajoutant des sous-classes aux classes génériques HCA. La partie numérique (en orange en bas de la FIGURE 7.19) est composée d'agents de contrôle des ressources individuelles et de trois types de holons pour contrôler les processus (ProductionHolon), le transfert de produits par chariots (TransferHolon) et la production (MS–System). Là encore, les choses ont été simplifiées pour être plus lisibles.

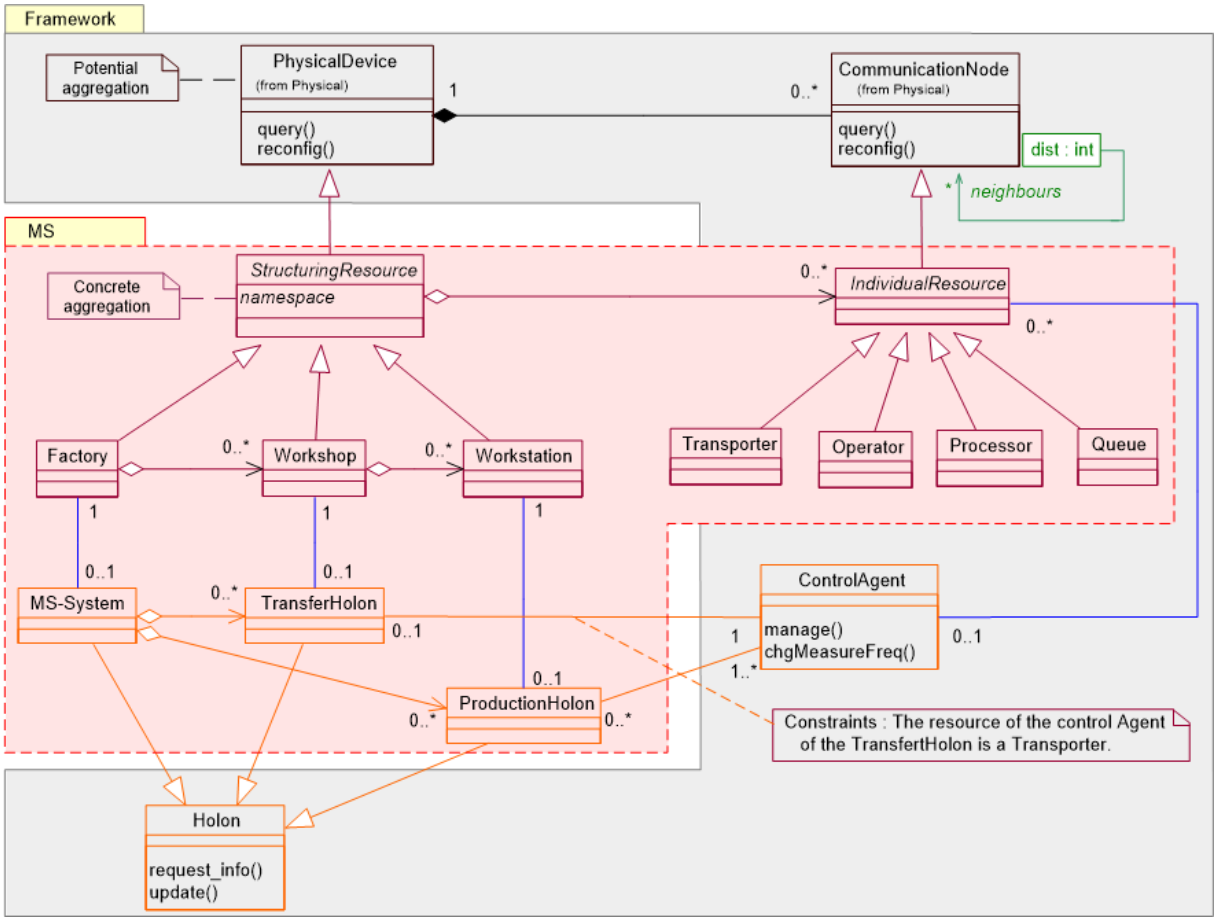


FIGURE 7.19 – Application de GARCIA à un atelier de production [Der+23]

Le domaine d'application ne se limite pas à la fabrication, et nous avons aussi appliqué GARCIA dans le cadre du projet français ANR McBIM, développant le concept de matériaux communicants pour l'industrie de la construction [DD23].

7.3.2 Vers une approche GL pour les HMS

Dans cette section on s'intéresse au MES, c'est-à-dire au niveau 3 de la norme ISA 95 (*cf.* page 204). Dans [AC17], nous avons posé les principes d'une approche génie logiciel pour les systèmes de contrôle de production. Plus particulièrement, dans le contexte de

l'Industrie 4.0, les systèmes de (cyber)production entrent dans le monde des services qui constitue une unité adéquate pour faire correspondre les systèmes virtuels et physiques y compris le *Cloud computing*, le *Big data*, l'Internet des objets (IoT) et la mobilité. La coordination et le contrôle d'un système aussi complexe par le biais d'acteurs ou de composants nécessite des méthodes et des techniques pour concevoir, vérifier et déployer les services, éventuellement de manière dynamique, faisant de l'ingénierie de services une approche incontournable pour développer nouvelle génération de systèmes de cyber-fabrication. En nous inspirant du travail mené sur Kmelia (*cf.* Chapitre 3), nous avons proposé un processus de développement basé sur des services sûrs, c'est-à-dire sur le processus *modélisation / vérification / exploitation* tel qu'illistré par la FIGURE 7.20. Les services sont définis de manière contractuelle à plusieurs niveaux [MAA10b] : syntaxique, structurel, fonctionnel, comportemental (dynamique et interactions) et qualité de service (QoS). La vérification couvre l'analyse statique, la vérification du comportement dynamique et celle du comportement fonctionnel d'un modèle hétérogène. L'exploitation vise ici à insérer les services dans des bibliothèques pour être invoqués par le MES.

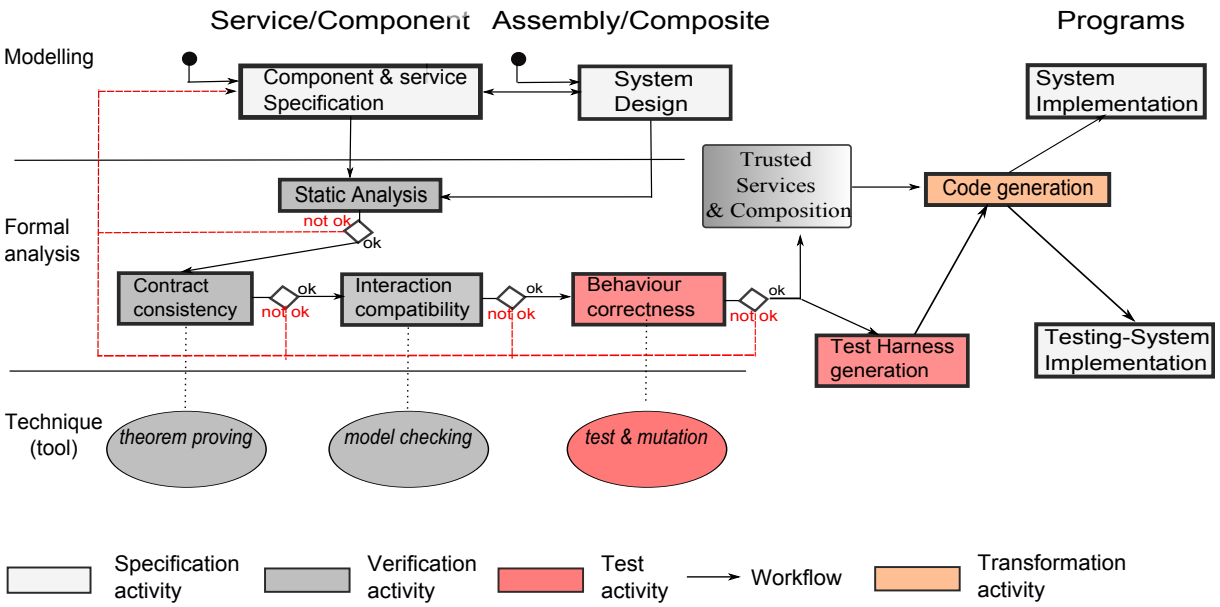


FIGURE 7.20 – Spécification, Vérification and Implantation de services [AC17]

Dans [TAC18] nous poussons plus loin la logique en imaginant la construction du système de contrôle par l'ingénierie des modèles (IDM) (*cf.* Section 1.2.6, page 33). Dans ce travail, nous nous sommes intéressés aux HMS (*cf.* page 199), qui présentent l'avantage d'une structure naturellement distribuée. Pour montrer l'intérêt de l'approche nous avons comparé la vision MDE avec la vision traditionnelle matérialisée par une application mise en œuvre sur une chaîne de production physique. Cette application implante un HMS orientée service appelé *SoHMS* et développé durant la thèse de Francisco Gamboa Quintanilla [GQ15]. Cette approche est déjà "moderne" car c'est un HMS développé en Java avec une vision service.

Le système est implanté sur le cas Sofal, une ligne d'assemblage installée à l'IUT de

Nantes à Carquefou, et schématisée par la FIGURE 7.21, qui est une version modifiée de la source [GQ15] dans laquelle nous faisons apparaître la possibilité de basculer soit sur le pilotage réel de Sofal soit en mode d'émulation sur Rockwell Arena. L'application logicielle est constituée de trois parties : une interface homme-machine (HMI) couplée au logiciel de contrôle (SoHMS) et la communication vers les dispositifs CPS (VLC). L'interface de communication avec Arena est gérée par la couche Visual Basic pour Applications à l'aide de TCP sockets [16].

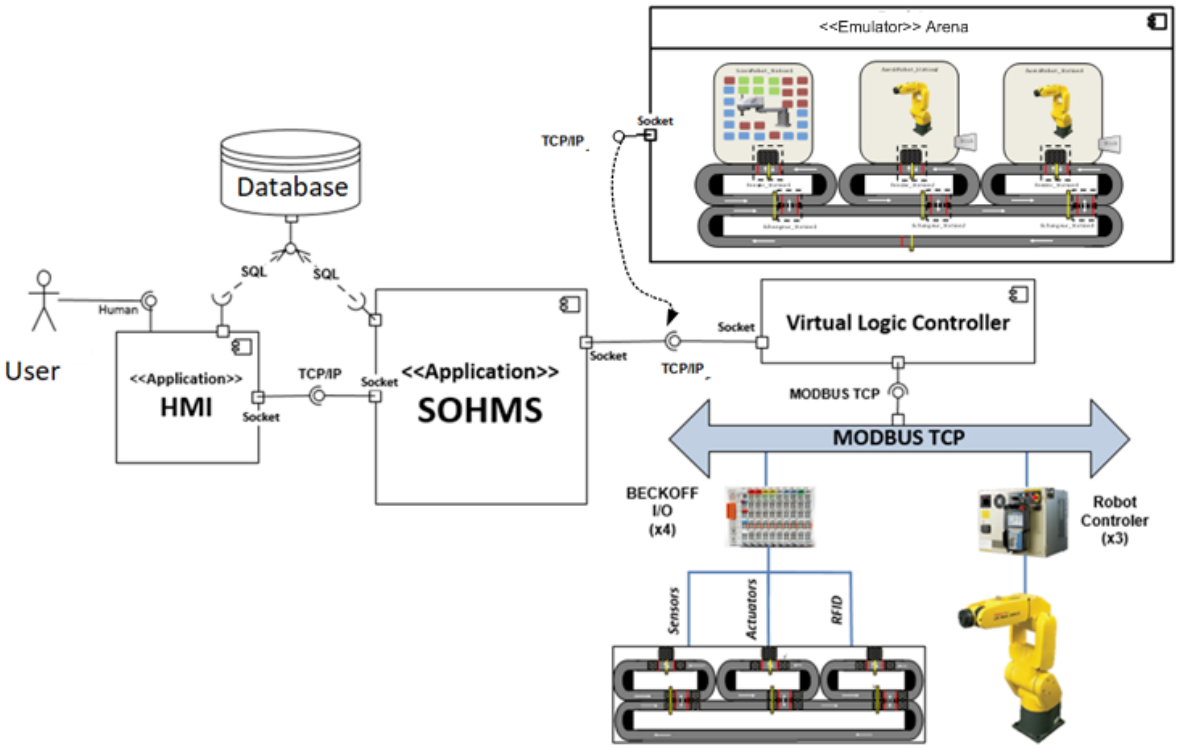


FIGURE 7.21 – Architecture de la ligne d'assemblage Sofal [AC17]

Ce qui nous intéresse ici n'est pas le détail de l'application, qu'on trouvera expliqué dans [GQ15], mais sa construction. Les métriques des deux programmes HMI et SoHMS comptaient 160 classes, 1240 méthodes et 14802 lignes de code. Ce qui en fait un code intéressant du point de vue génie logiciel. Dans [TAC18] nous avons mentionné un certain nombre de problèmes, du point de vue logiciel, tels que le manque d'abstraction et de modularisation menant à des problèmes de maintenance (lisibilité, évolutivité, non-localité, manque de réutilisation...). On y trouve de nombreuses redondances et duplication de code, parfois dues à des confusions de type classe/instance (*cf.* FIGURE 7.16), un mélange entre le code spécifique à l'application Sofal et les concepts plus généraux comme les hiérarchies de holons, la difficulté à vérifier des propriétés du système, à tester le système, etc. L'application SoHMS actuelle ne fournit pas de modèle dans lequel nous pourrions appliquer des techniques de vérification (démonstration de théorème ou vérification de modèle) et garantir les exigences accomplissement tel que l'exactitude, la qualité de service, la cohérence, etc. Les systèmes de fabrication actuels étant très diversifiés selon les

16. Sans faire du *teasing*, cette information sera utile pour la suite ;-))

exigences mécaniques ou physiques, tout changement peut avoir un effet négatif voire bloquant sur le logiciel de contrôle. Par exemple, ajouter, modifier ou supprimer un élément CPS composant de l'atelier de production peut bloquer la production. L'évolution et la maintenance de composants de la chaine de production sont des tâches très délicates en terme d'impact et elles nécessitent une expertise de logicielle.

Notre proposition de restructuration logicielle vise deux objectifs : (i) améliorer la qualité de l'architecture logicielle (modularité, extensibilité, variabilité) et (ii) améliorer le processus de construction (abstraction, réutilisation, testabilité, reconfigurabilité).

Architecture des applications Au lieu d'applications monolithiques, imbriquées et spécifiques (partie gauche de la FIGURE 7.22), nous ciblons des applications modulaires, indépendantes et paramétriques (partie droite de la FIGURE 7.22). L'objectif est d'être le plus modulaire possible afin de réduire le couplage entre les différents composants, et réduire les critères de spécificité d'un logiciel de production à un contexte précis (aménagement, commandes, ressources, etc.).

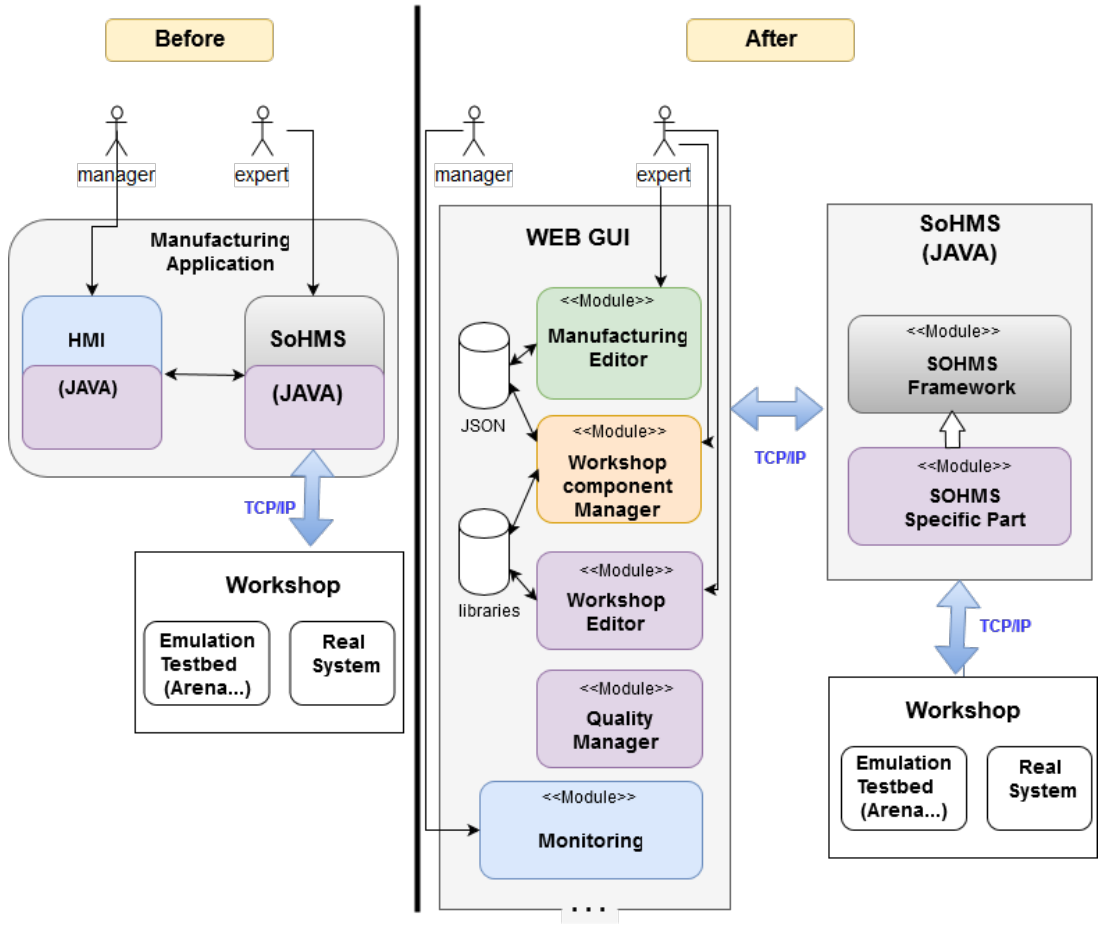


FIGURE 7.22 – Architecture d'un HMS évolutif [TAC18]

Le principal problème est de séparer la partie générique (le fonctionnement d'un atelier quelconque), de la partie spécifique qui dépend des ressources de l'atelier et organisation à mettre en place (ici pour Sofal). Le second problème est de séparer les fonctionnalités de l'atelier (ressources et flux) des caractéristiques de fabrication (commandes et produits).

La couche IHM est séparée en trois applications : la définition du système, la configuration et le suivi. Elle inclut les interactions qu'un expert peut effectuer sur un système de production donné : définir les différents scénarios (*Manufacturing editor*), monitorer les scénarios en cours et définir le plan de l'atelier. Techniquement, une couche d'interface graphique Web permet d'être indépendant des systèmes d'exploitation et des dispositifs physiques, mais aussi de connecter les applications au Cloud orienté services ou à l'IoT avec des plateformes de type Scada. L'application SoHMS pilote le système de production instancié avec les éléments de configuration choisis. Le moteur générique du contrôleur de production (*SoHMS framework*) inclut la définition HMS, les algorithmes de planification et les propriétés. La partie spécifique contient toutes les fonctionnalités spécifiques à un atelier instance (ressources, paramètres, configurations...). Enfin la couche physique correspond à différents besoins : émulation avec Arena ou Flexim par exemple, gestion des tableaux de bord et le véritable atelier. Le réseau de communication entre les couches doivent être compatibles avec différentes implémentations (sockets, messages, souvenirs partagés...) en fonction de l'infrastructure cible (vous y reviendrons dans la Section 7.3.3).

Processus de construction L'idée est d'utiliser une approche dirigée par les modèles qui soit itérative. Fondamentalement, l'application SoHMS existante (partie gauche de la Figure 7.23), est un ensemble de programmes Java qui est développé, testé et maintenu au niveau du code, même si une documentation UML existe. Les développeurs doivent avoir une certaine expertise, non seulement pour savoir où intervenir mais aussi comment corriger sans "tout casser" ou introduire de nouveaux bogues. Un suppport industriel d'intégration continue avec tests automatisés de non-régression peut toujours être mis en place, ce n'est pas une originalité ici. Nous avons construit des modèles abstraits par rétro-ingénierie pour dégager des éléments du *SoHMS framework*.

La nouvelle vision du processus de construction (partie droite de la figure Figure 7.23) installe une sorte de ligne de produits logiciels où l'on conçoit sur l'atelier de production avant de le mettre en place. Les bibliothèques HMS sont définies via des composants et des services avec leur implémentation, de telle sorte que la construction de systèmes de contrôle de fabrication est plutôt une activité d'intégration par assemblage plutôt qu'une activité de programmation. L'ingénierie des modèles permet de travailler avec des concepts abstraits et vérifier les propriétés attendues sans s'encombrer des détails inutiles de l'implémentation. Ceux-ci sont, rappelons-le (*cf.* Section 4.1 du Chapitre 4), qualifiés de modèles indépendants de la plateforme (PIM) dans l'esprit MDE alors que les programmes actuels sont réellement modèles spécifiques à la plate-forme (PSM). En effet, la généralisation/spécialisation permet de cibler divers dispositifs et systèmes physiques fournis par divers fournisseurs industriels. La partie droite de la Figure 7.23 montre les trois couches d'abstraction. Abstraction et séparation des préoccupations sont des facteurs clés la qualité de la conception logicielle.

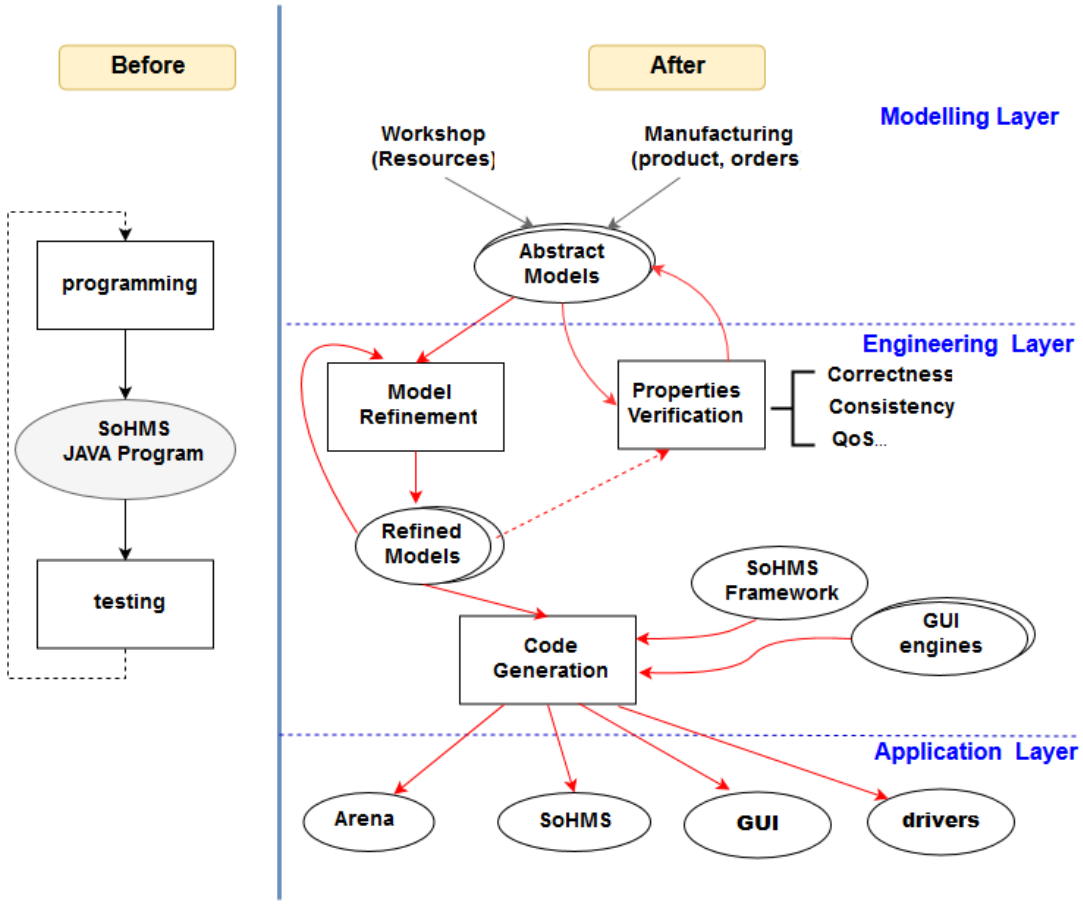


FIGURE 7.23 – Processus de construction dirigé par les modèles [TAC18]

7.3.3 Gérer l'hétérogénéité des communications

Dans cette section on s'intéresse à la communication entre le MES et les dispositifs CPS, c'est-à-dire à l'interface niveau 3 - niveau 2 de la norme ISA 95 (*cf.* page 204).

La multiplication des canaux d'information et la montée en puissance des CPS font que les communications deviennent un enjeu crucial dans les systèmes automatisés, et donc les systèmes de production notamment avec les jumeaux numériques mais aussi tous les capteurs distants. Les systèmes et réseaux de communication sont l'épine dorsale des systèmes de production modernes (IoT, Edge computing). Cisco [17] prévoyait que le trafic IP mondial fera plus que tripler pour atteindre 4.8 zettaoctets en 2022, que les communications entre machines (M2M) passerait de 30 à 50 % en 2023, et que les échanges réseaux basse consommation (IoT) passerait de 2,5 à 14,4 %. Dès lors l'interopérabilité des systèmes est un enjeu fondamental, qui se traite au niveau des couches basses applicatives (les logiciels systèmes & réseaux et les matériels sont considérés comme des paramètres ici). Nous l'avons vu dans les sections précédentes, le MES, et les jumeaux numériques en particulier, sont basés sur des systèmes distribués extensibles qui échangent par API de services via des middleware de communication de type *Remote Procedure Call (RPC), Object Request Brokers ORB (ORB), Message Oriented Middlewares (MOM), Enterprise*

17. Cisco VNI : Cisco Visual Networking Index : Forecast and Trends, 2018–2023. Cisco White Papers (2020)

Application Integration (EAI) framework, Enterprise Service Bus (ESB)...

Comme pour l'interface niveau 3 - niveau 2 de la norme ISA 95 (*cf.* page 204), une vision service unifiée (SOA) facilite l'interopérabilité, et il "suffirait" de prendre un intergiciel (*middleware*) service ou un protocole comme MQTT pour le résoudre. En pratique ce n'est pas le cas, les systèmes de production sont un empilement hérité (*legacy*) de solutions hétérogènes qu'on ne peut rayer pour des raisons (historiques, de compatibilité ascendante, de modularité, de coûts, d'organisation...). En raison de la diversité des ressources, des fournisseurs et des applications existantes, les moyens de communication sont hétérogènes et une seule technologie ne peut généralement pas suffire. La maintenance (logicielle) des systèmes de production introduit des perturbations lorsque du remplacement des appareils ou des ressources défaillants ou cassés par de nouveaux dispositifs (parfois livrés par de nouveaux fournisseurs), ou le remplacement de composants logiciels pour faire face à la dette technique des applications existantes (*e.g.* changement de contrôleurs)... L'évolution du logiciel est continue tout au long du cycle de vie du système et peut introduire une hétérogénéité qui aura un impact sur le l'intergiciel de communication. La maintenance associée devient délicate lorsque les instructions de communication (messages, signaux, variables...) sont diffus et fusionnés dans le code de l'application ; la restructuration du code (*refactoring*) est compliquée.

La FIGURE 7.24 met en évidence deux exemples d'interfaces de communication dans des HMS. Le cas de gauche de la FIGURE 7.24, extrait de [Jov+14], se situe dans la fabrication de pneumatiques. Le cas de droite de la FIGURE 7.24, reprend l'exemple Sofal de la section précédente (*cf.* FIGURE 7.21). Tous deux présentent une connexion avec la contrepartie simulation du système réel.

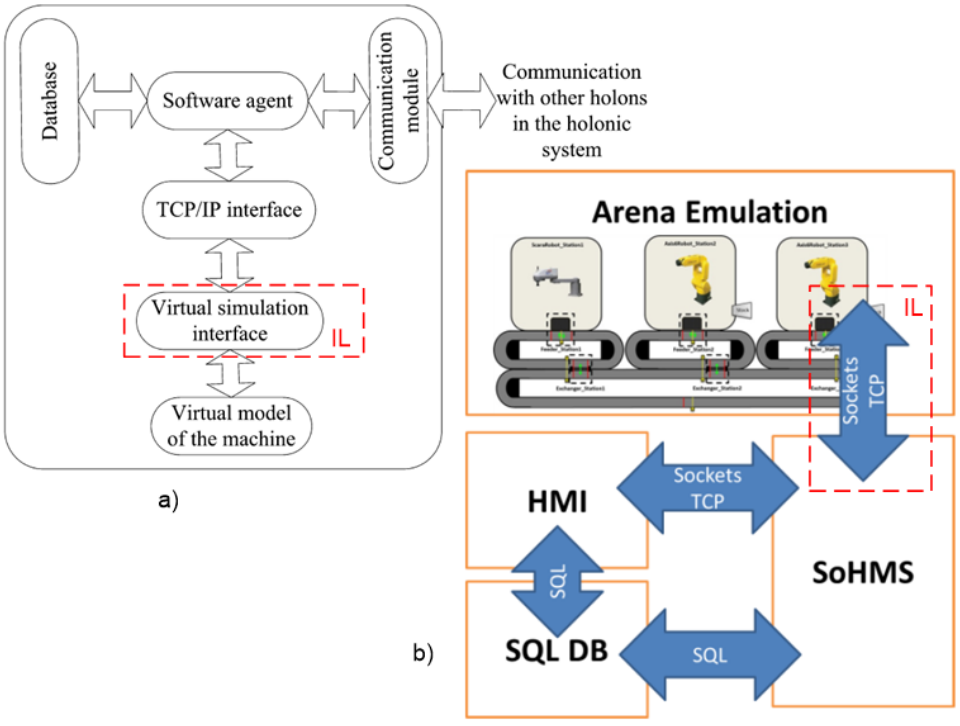


FIGURE 7.24 – Couches communication HMS [AAC19]

Dans de telles architectures, trois classes de communication peuvent être différenciées :

— intra-holons Les holons s'exécutent en parallèle et échangent des informations selon un protocole qui doit être standardisé. Par exemple, un holon responsable de la gestion d'une ressource doit être capable à la fois de négocier avec d'autres holons pour établir la planification de cette ressource, et en même temps de déclencher et de surveiller l'activité de la ressource elle-même suite à cette planification. Évidemment, ces tâches ne sont pas exécutées sur le même horizon temporel, et sont donc exécutés en parallèle. Les solutions de communication sont souvent basées sur des tableaux de variables partagées ou par files d'attente de messages.

— inter-holons Même si les holons ont des comportements indépendants, ils sont destiné à collaborer et à négocier pour atteindre les objectifs de la holarchie, en général par échange de messages, signaux et variables partagées. De nombreux protocoles ont déjà été élaborés, notamment le *Contract Net Protocol (CNP)*. Cependant, cette tendance est censée évoluer avec les holons intégrés, où certains holons pourraient être physiquement séparés des systèmes de contrôle.

— externe Les systèmes de contrôle communiquent également avec d'autres applications, *e.g.* une interface utilisateur-machine, une base de données, certains automates, etc. Cette communication implique plusieurs protocoles liés aux contraintes logicielles ou matérielles. La liste est longue, mais elle peut être organisée en trois classes : (i) protocoles orientés Web ou IoT, tels que UDP, TCP, MQTT ou HTTP ; (ii) protocoles de réseaux locaux industriels, par ex. Modbus TCP ou Profibus ; (iii) protocoles de messagerie OPC DA, OPC-UA ou SOAP, Sockets TCP ou OTP.

La probabilité d'avoir plusieurs protocoles dans la même application est assez élevée, notamment dans le *plug-and-product* [MLC15] ou les RMS où divers équipements et interfaces sont branchés et déconnectés tout au long le cycle de vie du système de contrôle. Lors de l'intégration d'un jumeau numérique dans l'architecture, cette fonctionnalité est tout à fait obligatoire, En résumé, on a besoin d'un outil de communication multi-protocole.

Dans [AAC19] nous proposons une architecture de communication hétérogène qui s'inspire des principes de bonne conception logicielle (*cf.* page 53) et notamment de la séparation des préoccupations. Nous y avons comparé différents styles architecturaux et proposé une architecture mixte dans laquelle apparaît un composant chargé de traiter en souplesse l'hétérogénéité et nommé *Multi-Protocol Communication Tool* (MPCT). Ce composant est illustré par un hexagone sur la Figure 7.25.

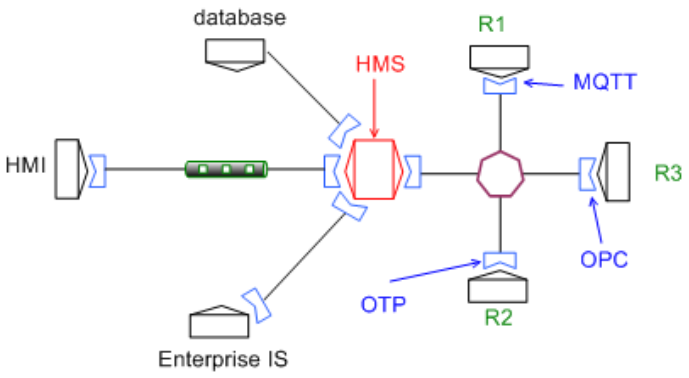


Figure 7.25 – Architecture de communication mixte pour HMS [AAC19]

La Figure 7.26 montre la nouvelle architecture Sofal qui est un compromis entre simplicité, adaptabilité et hétérogénéité. La partie gauche reste telle quelle car à la fois la persistance (JDBC) et le lien avec l'IHM (TCPSocket) sont des communications de type point à point avec envoi asynchrone de messages avec des files d'attente par boîtes aux lettres. La partie droite de la Figure 7.26 permet d'ajouter de nouvelles ressources avec d'autres protocoles que TCPSocket, à condition de développer les connecteurs de communication correspondants. Un langage de message de ressource (RML) définit l'interface entre SOHMS et les ressources.

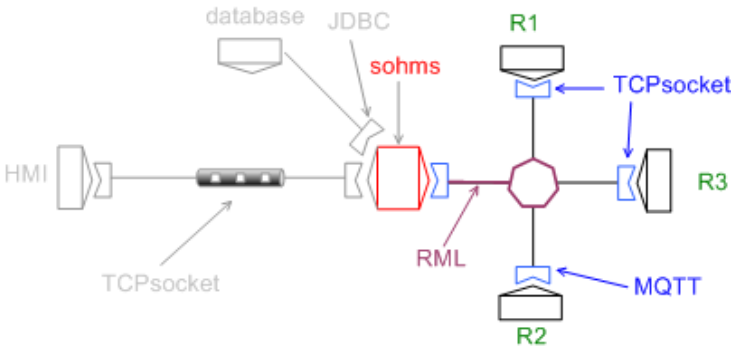


Figure 7.26 – Illustration du MCPT pour Sofal [AAC19]

Dans [ACA20] nous avons proposé une implantation pour cet outil MPCT, développé durant un projet de Master 2 Alma. La Figure 7.27 montre l'architecture du MPCT dans son contexte. MPCT assure la communication avec les ressources hétérogènes et un seul protocole avec HMS, un peu comme un multiplexeur/démultiplexeur. Le MPCT gère dynamiquement les évolutions de resources via les ports Ti par le ProtocolHandler selon le référentiel de configuration selon une architecture *hub-and-spoke* architecture [AZZ15]. Les transmissions effectives se font via le composant ProtocolHandler qui se charge de convertir les messages (bibliothèques d'adapteurs). De nouveaux protocoles peuvent être créés dynamiquement via la ProtocolFactory.

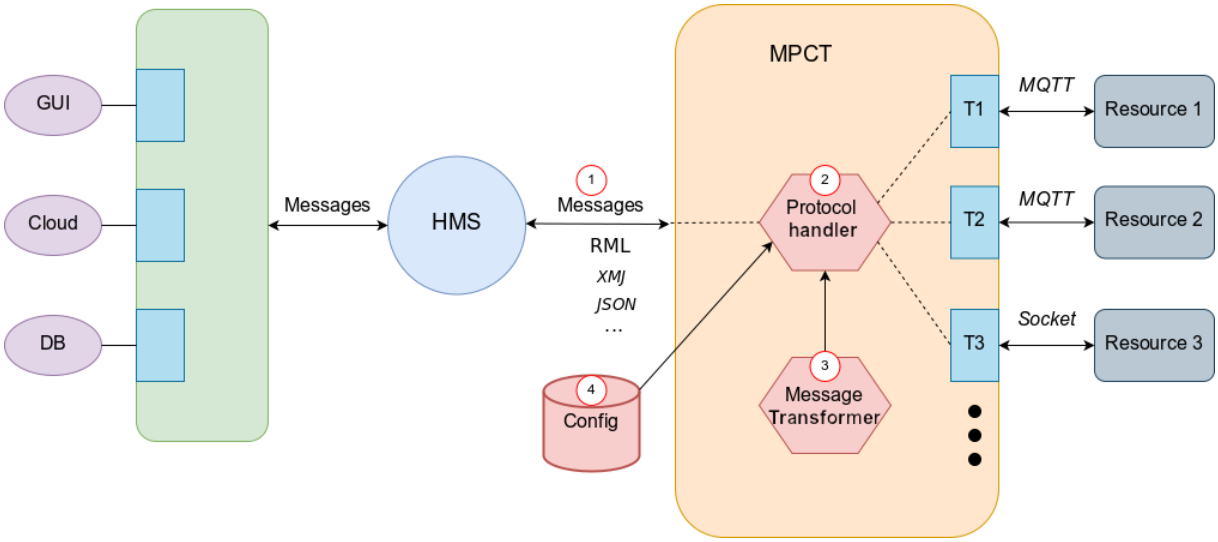


Figure 7.27 – Contexte et Architecture MPCT [ACA20]

MPCT est implémenté en Java avec des patrons de conception (*e.g. Factory, Adaptor, Composite State, Proxy, Mediator* et *Facade*) pour favoriser la maintenance mais aussi la réutilisation. Des optimisations peuvent améliorer les performances *e.g.* lorsque plusieurs

ressources utilisent la même instance MQTT, nous les regroupons dans un seul adaptateur de communication. MPCT peut être instancié autant de fois que nécessaire, par exemple comme intergiciel pour les communications de ressource à ressource directes (hors HMS). Les expérimentations ont validé l'approche et le POC en utilisant MQTT (broadcast) et TCP sockets (point à point).

7.4 Conclusion

L'informatique industrielle est marquée par le pragmatisme et le besoin de solutions techniques fiables et robustes. Ces solutions prennent du temps à mettre au point et sont parfois trop rigides (limiter la prise de risque, rester sur des solutions qui fonctionnent, personnaliser le logiciel, etc.) et qui deviennent inadaptées au contexte qui au contraire fluctue de plus en plus rapidement (concurrence, émergences technologiques, variabilité de la demande *time to market*...). La transformation numérique des entreprises et sa déclinaison dans l'industrie (usine du futur, industrie 4.0, ...) sont incontournables et rendent le génie logiciel incontournable car le logiciel est ubiquitaire et les solutions empiriques ne peuvent suivre le rythme des corrections et des évolutions.

Les travaux présentés rapidement ici montrent la pertinence d'une approche GL pour mettre en œuvre les systèmes de contrôle de production, d'autant que ces derniers sont désormais en lien direct avec les systèmes d'information des entreprises. Les approches proposées restent pragmatiques et répondent au besoin de techniques et outils pour aider au développement des applications, dans l'esprit du génie logiciel. Il y a beaucoup à faire, les challenges sont passionnants.

Post-scriptum

Le travail enclenché depuis 2017 se poursuit en partie sur le projet ANR *Rapid reOnfiguration of manufacturing systems : a model-baseD software engineering and human Interaction Coupled approach* (RODIC) [18], piloté par Olivier Cardin. se situe dans le contexte des RMS (*cf.* page 198) et vise à aider les opérateurs à sélectionner la configuration à mettre en place, selon des indicateurs de performances choisis (l'humain est au centre de la décision). Le projet doit aboutir à un outil opérationnel qui permet de définir des configurations d'ateliers de production, de les paramétrer en scénarios qui permettent une simulation et une évaluation de performance.

Le *Work Package* dont j'ai la responsabilité est en charge des modèles de configurations. Ces modèles sont définis par des DSLs que nous étudions dans la thèse de Yasmina Dali Youcef, co-encadrée avec Olivier Cardin et Erwan Bousse démarrée en juin 2023. La particularité est le côté fractal des éléments en jeu dans le fonctionnement d'un atelier de production et la variété des paramètres des RMS. Nous avons mis en évidence un premier modèle de référence [AC24]. Nous avons étudié le cas particulier des indicateurs de performance (KPI) qui proviennent du système d'information (architecture d'entreprise) [AG24] mais surtout leur applicabilité dans les systèmes concrets car en pratique ils sont noyés dans le code des programmes MES.

Ce chapitre a mis en évidence des acquis sur les compétences suivantes, que nous reverrons dans le chapitre 8.

C3

Investir de nouveaux domaines, collaborer avec des chercheurs d'autres spécialité implique de se comprendre et se faire comprendre, les apports croisés permettent d'innover dans les solutions des problèmes identifiés.

18. `https://rodic.ls2n.fr/`

Quatrième partie

Modèles de compétences

Easy is not a option..No days off..Never Quit..Be Fearless..Talent you have Naturally..Skill is only developed by hours and hours of Work.

FaceBook post from Jan 28, 2014
Usain Bolt

Dans cette partie IV, constituée du seul chapitre 8, je reviens sur mes interrogations initiales de la page 11 au sujet de l'habilitation à diriger les recherches, à savoir quelles sont les compétences attendues et les ai-je ? Pour poser le raisonnement, et c'est un fil conducteur dans ce manuscrit, je passerai par des modèles, qui sont ma contribution au sujet. J'essayerai ensuite d'instancier pour trouver des éléments de réponse à mes interrogations initiales.

La question n'est pas seulement de savoir si on est apte à faire quelque chose mais si on sait le faire. *"La formation révèle l'aptitude, et le terrain révèle la compétence."* Patrice Aimé Agossou [19]

19. `https://tinyurl.com/agossou`

Développer des compétences pour diriger des recherches

Aucun de nous, en agissant seul, ne peut atteindre le succès.

Discours d'investiture, 10 mai 1994
Nelson Mandela

Ce chapitre est un peu atypique au vu des HDR consultées. Il vient de la réflexion que j'ai mené sur l'HDR, l'idée que je m'en fais et mes capacités et aspirations sur le sujet. Cette thématique est aussi en phase avec les nouvelles habilitations en enseignement [APC] et de suivi des doctorants à Nantes Université.

L'approche de formation et évaluation par les compétences vient de la formation par l'apprentissage et la formation continue où on essaie de certifier les capacités à connaître ou à faire telle ou telle chose, que ce soit pour les certificats de langues, certificats d'études, certification qualité, certification Microsoft, certification C2i devenue PIX... En général, l'approche de mise en situation est plus qualitative que quantitative, sauf lorsqu'on industrialise la certification par des questionnaires QCM, comme pour le PIX ou les certifications d'entreprises (Microsoft, Cisco...). C'est une approche plutôt cartésienne (on va structurer les acquis en matrices de compétences sur des thématiques données) alors qu'en éducation traditionnelle l'approche est plutôt systémique (l'évaluation est plus globale sur un ensemble de sujets corrélées).

A la Miage de Nantes, nous avons intégré la notion de compétences depuis de nombreuses années et habilitations, mais cette notion était décorellée de l'évaluation quantitative. L'approche par compétences (APC) telle qu'elle est menée désormais, et qui a pu être expérimentée sur le C2i, convient à l'évaluation de connaissances d'une formation mais pas à l'enseignement et à l'éducation. Les savoirs, savoirs-faire et savoirs-être ne se mesurent pas simplement et les évaluations numériques couvrent bien au delà des matrices de compétences limitées, réductrices et formatées.

Les compétences en recherche vont bien au delà des contributions scientifiques, qu'on met en avant dans les publications. Elles couvrent aussi le savoir-faire, le savoir-être, l'éthique, le travail collaboratif, la gestion (humaine, de projet, des risques, d'équipe...), ou encore la culture d'amélioration continue et d'innovation. Il me paraît pertinent pour un chercheur de réfléchir à la pratique de la recherche et le savoir-faire en ces termes.

Ma contribution dans ce chapitre est une synthèse autour des compétences en recherche, présentée sous forme d'un référentiel, sur lequel je m'appuie ensuite pour une évaluation subjective mais argumentée de compétences développées en pratique.

8.1 Introduction

Au delà des thématiques et des défis scientifiques à relever, diriger des recherches c'est diriger des projets [1] à plus ou moins long terme, diriger des personnes, communiquer avec des tutelles et des partenaires, transférer des connaissances vers l'enseignement ou la société, établir des stratégies. Il y a donc des compétences de management et de communication à maîtriser. Diriger des recherches implique des compétences qu'on peut répertorier de manière classique en trois catégories :

— savoirs et connaissances des domaines scientifiques et techniques (domaines, problèmes et défis, pistes de solutions et contributions),
— savoir-faire (gestion d'activités ou de projets de recherche, management, etc.),
— savoir-être (gestion humaine, communication, transmission, pilotage, déontologie, curiosité, innovation...).

Nous avons placé la déontologie dans le savoir-être mais cela pourrait être une catégorie à part entière, car ne pas réfléchir aux conséquences de ses recherches est dangereux. La science, tout comme la liberté d'expression, n'autorise pas tout et n'excuse pas tout. Le génie logiciel n'est pas le domaine où les enjeux déontologiques et les conséquences humaines sont les plus aigus *e.g.* automatiser le développement ne va pas mettre tout les développeurs au chômage. Les enjeux humains et sociétaux sont plus cruciaux en IA ou en cybernétique. Noter tout de même le cas particulier du volet "*dependability and security*" où, même si la responsabilité en jeu porte plus sur le projet de développement (métier) que la manière de développer (GL), les défauts non détectés du logiciel peuvent avoir des conséquences matérielles, financières ou humaines.

Ce chapitre vise à discuter de ces points, les analyser en les illustrant, le cas échéant, par des retours d'expérience. Néanmoins comme tout travail rigoureux, cela commence par un état de l'art, que je présente dans la section 8.2. Je distingue trois trois rôles : le chercheur, l'encadrant et le directeur de recherche. Pour chacun, j'identifie les éventuels référentiels de compétences et en l'absence de propositions je suggère un référentiel. J'effectue ensuite un bilan de compétences en collectant les compétences qui parsèment ce manuscrit que je complète par une auto-évaluation illustrée sur mon parcours dans la section 8.3.

1. Les projets dont il est question, sont de deux types : les réponses à des appels à projets avec un volet financier mais aussi tout simplement l'organisation de travaux collaboratifs, qu'on appellera activités de recherche.

8.2 Référentiels de compétences

Avertissement

Pour des raisons pratiques de rédaction, comme pour les textes légaux, j'utiliserai souvent le masculin dans la formulation.

Nous distinguons le référentiel du chercheur, qui doit savoir chercher des réponses à des problèmes dans un espace délimité, de celui du directeur, qui mène les recherches doit savoir où on va et comment on y aller (ensemble). Entre le chercheur et le directeur, se trouve l'encadrant, qui est un chercheur (confirmé) dirigent des étudiant(e)s en thèse. J'explore donc le spectre des compérences sur ces trois rôles que peut prendre une personne dans la recherche. Ces rôles sont des niveaux de maturité, en ce sens qu'un directeur est aussi un encadrant qui est un chercheur.

8.2.1 Référentiels de compétences du chercheur

L'approche compétence, qui s'applique progressivement dans l'enseignement, se diffuse aussi à la formation par la recherche. Il existe depuis 2019, un référentiel des compétences attendues des titulaires du diplôme de doctorat, qui est défini par l'arrêté du 22 février 2019 [2]. *La délivrance du doctorat certifie la capacité à produire des connaissances scientifiques nouvelles de haut niveau ainsi que l'acquisition et la maîtrise de blocs de compétences communs à l'ensemble des docteurs et liés à leur formation par la recherche.* Son objectif n'est pas directement d'évaluer les docteurs mais *de favoriser le recrutement des docteurs par les employeurs des secteurs de la production et des services.* Le référentiel des compétences attendues pour un docteur (et par extension pour un chercheur) est composé de 6 blocs de compétences[2], que je rappelle ici pour y faire référence ensuite.

- *Bloc 1 - Conception et élaboration d'une démarche de recherche et développement, d'études et prospective*
  - *disposer d'une expertise scientifique tant générale que spécifique d'un domaine de recherche et de travail déterminé ;*
  - *faire le point sur l'état et les limites des savoirs au sein d'un secteur d'activité déterminé, aux échelles locale, nationale et internationale ;*
  - *identifier et résoudre des problèmes complexes et nouveaux impliquant une pluralité de domaines, en mobilisant les connaissances et les savoir-faire les plus avancés ;*
  - *identifier les possibilités de ruptures conceptuelles et concevoir des axes d'innovation pour un secteur professionnel ; -apporter des contributions novatrices dans le cadre d'échanges de haut niveau, et dans des contextes internationaux ; -s'adapter en permanence aux nécessités de recherche et d'innovation au sein d'un secteur professionnel.*
- *Bloc 2 - Mise en œuvre d'une démarche de recherche et développement, d'études et prospective*
  - *mettre en œuvre les méthodes et les outils de la recherche en lien avec l'innovation ;*
  - *mettre en œuvre les principes, outils et démarches d'évaluation des coûts et de financement d'une démarche d'innovation ou de R&D ;*

2. `https://www.legifrance.gouv.fr/loda/id/JORFTEXT000038200990`

  - *garantir la validité des travaux ainsi que leur déontologie et leur confidentialité en mettant en œuvre les dispositifs de contrôle adaptés ;*
  - *gérer les contraintes temporelles des activités d'études, d'innovation ou de R&D ;*
  - *mettre en œuvre les facteurs d'engagement, de gestion des risques et d'autonomie nécessaire à la finalisation d'un projet R&D , d'études ou d'innovation.*
- *Bloc 3 - Valorisation et transfert des résultats d'une démarche R&D, d'études et prospective*
  - *mettre en œuvre les problématiques de transfert à des fins d'exploitation et valorisation des résultats ou des produits dans des secteurs économiques ou sociaux ;*
  - *respecter les règles de propriété intellectuelle ou industrielle liées à un secteur ;*
  - *respecter les principes de déontologie et d'éthique en relation avec l'intégrité des travaux et les impacts potentiels ;*
  - *mettre en œuvre l'ensemble des dispositifs de publication à l'échelle internationale permettant de valoriser les savoirs et connaissances nouvelles ;*
  - *mobiliser les techniques de communication de données en "open data" pour valoriser des démarches et résultats.*
- *Bloc 4 - Veille scientifique et technologique à l'échelle internationale*
  - *acquérir, synthétiser et analyser les données et informations scientifiques et technologiques d'avant-garde à l'échelle internationale ;*
  - *disposer d'une compréhension, d'un recul et d'un regard critique sur l'ensemble des informations de pointe disponibles ;*
  - *dépasser les frontières des données et du savoir disponibles par croisement avec différents champs de la connaissance ou autres secteurs professionnels ;*
  - *développer des réseaux de coopération scientifiques et professionnels à l'échelle internationale ;*
  - *disposer de la curiosité, de l'adaptabilité et de l'ouverture nécessaire pour se former et entretenir une culture générale et internationale de haut niveau.*
- *Bloc 5 - Formation et diffusion de la culture scientifique et technique*
  - *rendre compte et communiquer en plusieurs langues des travaux à caractère scientifique et technologique en direction de publics ou publications différents, à l'écrit comme à l'oral ;*
  - *enseigner et former des publics diversifiés à des concepts, outils et méthodes avancés ;*
  - *s'adapter à un public varié pour communiquer et promouvoir des concepts et démarches d'avant-garde.*
- *Bloc 6 - Encadrement d'équipes dédiées à des activités de recherche et développement, d'études et prospective*
  - *animer et coordonner une équipe dans le cadre de tâches complexes ou interdisciplinaires ;*
  - *repérer les compétences manquantes au sein d'une équipe et participer au recrutement ou à la sollicitation de prestataires ;*
  - *construire les démarches nécessaires pour impulser l'esprit d'entrepreneuriat au sein d'une équipe ;*
  - *identifier les ressources clés pour une équipe et préparer les évolutions en termes de formation et de développement personnel ;*
  - *évaluer le travail des personnes et de l'équipe vis à vis des projets et objectifs.*

A l'école doctorale *Mathématiques et Sciences et Technologies du numérique, de l'Information et de la Communication (MaSTIC)*, le rapport d'avancement soumis au Comité de Suivi Individuel (CSI) contient depuis 2023 un volet "auto-évaluation des compétences". Pour leur Comité de Suivi Individuel (CSI), les doctorant(e)s sont invité(e)s à noter et à dater, au fil de l'année, les formations suivies, leurs réalisations, leurs publications et communications, et plus généralement tout ce qui atteste leurs compétences.

Remarque

Cette description des compétences est assez énigmatique pour les doctorant(e)s qui ne savent guère comment remplir leur auto-évaluation de compétences.

Il faut maîtriser son domaine de recherche (bloc 4) et son périmètre (bloc 1), définir et évaluer ses contributions (bloc 2), imaginer l'usage (bloc 3), communiquer ses résultats ou son expertise (bloc 5) et encadrer une équipe (bloc 6).

Remarque

Le référentiel est ambitieux en ce sens qu'il couvre l'ensemble des situations que peut rencontrer un(e) doctorant(e), mais chaque doctorant n'est pas forcément en situation de monter en compétence sur tous les blocs. Le bloc 6, en particulier, correspond sans doute à l'encadrement de personnels techniques, par exemple dans les sciences expérimentales ou les études de terrain en Sciencs Humaines et Sociales. Interprété comme un encadrement de recherche, il sort, à mon avis, du périmètre du rôle "chercheur" et correspond plus à des compétences du directeur de recherche (*cf.* Section 8.2.3).

Certaines compétences se retrouvent sur plusieurs blocs, comme la déontologie de recherche et l'éthique, pour lesquelles des formations sont obligatoires, ou l'innovation et la R&D. On note que certaines compétences sont évaluables par la formation et pas forcément par la pratique. Le bloc 4 se mélange un peu avec le bloc 1 ; on aurait pu séparer maîtrise continue du domaine thématique (bloc 1 et les trois premières compétences du bloc 4) et positionnement dans le domaine (réseautage, collaborations du bloc 4).

Remarque

Le référentiel est structuré sur le savoir-faire. La connaissance apparait dans certains blocs. Inversement le docteur doit produire des connaissances. Par contre le savoir-être est ignoré. Je l'aborderai dans l'encadrement collectif dans la Section 8.2.2.

Au fil de mes encadrements, j'ai recensé informellement un ensemble de compétences attendues pour le docteur, appliqué au génie logiciel. J'indique pour chacune la correspondance avec les blocs de compétences du référentiel (les blocs 3 valorisation et 6 encadrement ne sont pas couverts dans mes attentes) : (i) maîtriser son périmètre de recherche (savoir de quoi on parle) [blocs 1 & 4], (ii) contribuer au domaine (trouver des idées originales) [bloc 2] (iii) implémenter et expérimenter (savoir développer et tester sur des cas d'étude) [bloc 2], (iv) présenter ses recherches (savoir communiquer) [bloc 5]. Pour chaque compétence, on peut évaluer sur une échelle le niveau de compétence [3].

Les compétences du référentiel s'appliquent à tout chercheur et donc implicitement à

3. En pratique, j'ai constaté qu'au moins deux compétences à un niveau satisfaisant sont nécessaires pour que la thèse puisse avancer.

ceux qui les encadrent. L'encadrant doit couvrir d'autres compétences puisqu'il doit faire monter en compétences les doctorants.

8.2.2 Référentiels de compétences d'un encadrant

J'utilise le terme générique **encadrant** pour désigner les personnes qui participent à l'encadrement d'une thèse et donc à faire monter en compétences le(la) doctorant(e). Dans la charte du doctorant de Nantes Université [4], le terme utilisé est *le(s) directeur(s) de thèse*. Dans le document de suivi individuel (CSI) de l'Ecode doctorale MaSTIC [5], quatre termes sont utilisés *Directeur de thèse, Co-directeur, Co-encadrant, Autre co-encadrant*, les règles de composition étant définies dans le règlement intérieur [6]

I) Etude de l'existant

> Je n'ai pas trouvé de référentiel de compétences pour l'encadrant de thèse. Cependant, la fonction est similaire à celle d'un tuteur en apprentissage.

Sur le site du ministère de l'éducation nationale, une fiche donne le rôle du maître d'apprentissage [7]. *Le maître d'apprentissage doit posséder des compétences professionnelles mais aussi des qualités pédagogiques. [...] Il assure la formation pratique de l'apprenti et l'accompagne vers l'obtention de son diplôme. il forme l'apprenti sur son temps de travail, se rend disponible pour répondre aux questions de l'apprenti et s'assurer de son intégration.*

Sur le site du ministère de l'emploi on trouve un référentiel de compétences [8] pour la *Certification relative aux compétences de maître d'apprentissage / tuteur* (*cf.* Table 8.1), qui finalement reste au niveau organisationnel ; il décrit des activités.

Table 8.1 – Certification relative aux compétences de maître d'apprentissage[8]

DC	Domaines de compétences	N°	Compétences professionnelles
1	**Accueillir et faciliter l'intégration de l'apprenti/alternant**	1	Préparer l'arrivée de l'apprenti/alternant dans l'entreprise
		2	Accueillir l'apprenti/alternant à son arrivée dans l'entreprise
		3	Faciliter l'intégration de l'apprenti/alternant durant la période d'essai
2	**Accompagner le développement des apprentissages et l'autonomie pro**	4	Suivre le parcours avec le centre de formation
		5	Organiser le parcours au sein de l'entreprise
		6	Accompagner l'apprenti/alternant dans son parcours d'apprentissage
3	**Participer à la transmission des savoir-faire et à l'évaluation professionnelle des apprentissages**	7	S'appuyer sur des situations de travail pour développer l'apprentissage
		8	Guider la réflexion de l'apprenti/alternant sur ses activités professionnelles et d'apprentissages
		9	Evaluer les acquis des apprentissages en situation de travail

Le Conseil national des universités (CNU) a publié en 2021 un rapport sur la Concer-

4. https://www.univ-nantes.fr/recherche-et-innovation/doctorat/formulaires-lies-a-la-these-de-doctorat
5. https://ed-mastic.doctorat-paysdelaloire.fr/
6. https://ed-mastic.doctorat-paysdelaloire.fr/medias/fichier/ri-mastic-2023_1682584001370-pdf
7. https://travail-emploi.gouv.fr/IMG/pdf/apprentissage-infographie-role_du_maitre.pdf
8. https://travail-emploi.gouv.fr/IMG/pdf/referentiel_de_competences_matu.pdf

tation sur le recrutement des enseignants-chercheurs [9], qui contient avec des préconisations au sujet des HDR et notamment *"Définir les compétences transversales attestées par l'HDR, en regard des quatre objectifs portés par l'arrêté du 23 novembre 1988"*. Enfin l'HCERES reprend en 2022 le référentiel CITE [10] niveau 8 de l'UNESCO, pour l'inscrire dans la politique des établissements pour une formation pour et par la recherche. Il concerne donc plus la formation doctorale que l'encadrement.

A l'ED MaSTIC, ni le règlement intérieur[6] ni la charte du doctorat des Pays de la Loire[4] ne stipulent d'éléments au sujet des rôles et compétences de l'encadrement. Il est simplement mentionné *la déontologie inspirant les dispositions réglementaires en vigueur et les pratiques déjà expérimentées dans le respect de la diversité des disciplines et des établissements* et *la direction de thèse doit dégager le caractère novateur du projet dans le contexte scientifique et s'assurer de son actualité*. Ces éléments couvrent partiellement le bloc 1 de compétences du référentiel du chercheur de la section 8.2.1. *La direction de thèse doit définir et rassembler les moyens à mettre en œuvre pour permettre la réalisation du travail dans de bonnes conditions*. Plus que des compétences attendues, on y trouve donc des règles administratives (sortes de droits et devoirs utiles en cas de conflit), organisationnelles ou déontologiques *e.g.* l'intégrité scientifique.

L'éducation nationale[7] mentionne que le maître d'apprentissage *peut encadrer 2 apprentis et 1 redoublant au maximum*. A l'université, le nombre maximum d'encadrants par thèse est défini par le conseil de chaque ED. L'arrêté du 25 mai 2016 [11] fixe le cadre national de la formation et les modalités conduisant à la délivrance du diplôme national de doctorat fixe les conditions. A l'ED MaSTIC, le règlement intérieur[6] stipule les points suivants. *Un minimum de 40% d'encadrement est attribué à la direction de doctorat qui s'engage à y consacrer une part significative de son temps. Le pourcentage majoritaire d'encadrement doit être attribué à la direction. Le pourcentage minimum pour une co-direction ou un co-encadrement est porté à 30%. Une direction ou une co-direction de doctorat HDR ne peut encadrer en même temps plus de 6 doctorants ou doctorantes dans la limite de 300%.*

Littérature Pour terminer, notons un article intéressant *"Encadrer des thèses : d'abord, ne pas nuire."* en deux volets : *(1) État d'un champ de recherche [HHLGN22a], (2) Diriger c'est enseigner [HHLGN22b]*. Nous avons tous connu ou vécu des situations de dérives d'encadrements dans notre parcours universitaire. Du simple abus de pouvoir au harcèlement, certaines situations deviennent récurrentes ou des habitudes dans tel ou tel domaine ou avec telle ou telle personne. Cet article collectif récent est une contribution réflexive sur les pratiques pédagogiques d'encadrant(e)s de thèse, en géographie humaine, en France.

— Le premier volet ouvre une réflexion sur le vocabulaire (directeur · ice, encadrant · e,

9. https://tinyurl.com/yc79nw5d
10. https://www.hceres.fr/sites/default/files/documents/DEI/fr_2022-dei_referentiel_doctorat_etranger.pdf
11. https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000032587086

patron · ne, supervisor…), les textes réglementaires, la littérature scientifique existante, et l'importance qu'il y a d'inscrire la thèse dans une trajectoire professionnelle. Les auteurs insistent sur le lexique, qui est important pour établir la relation entre doctorant(e)s et encadrant(e)s, entre le rapport hiérarchique et la soumission, l'autonomie et l'obéissance, l'épanouissement et le conditionnement. Le respect mutuel est important dans la relation de pouvoir dont il faut éviter les abus. On y trouve des références au management (et à ses abus) et à la pédagogie. Les auteurs mentionnent trois styles d'encadrement : *" Un premier style d'encadrement correspond à un faible niveau d'engagement (laissez-faire), soit une direction de thèse peu structurée, peu accompagnante, dans un faible niveau d'interaction. Le modèle pastoral est centré sur les besoins du · de la doctorant · e et son soutien, mais est moins orienté vers des tâches et les objectifs. Le style directorial met de côté ces questions d'interactions non immédiatement relatives aux tâches de la recherche pour se concentrer sur les livrables, les soumissions, les objectifs à court terme dans une pratique relevant plus du micro-management. Le modèle contractuel enfin s'adresse à des candidat · es particulièrement motivé · es et autonomes mais l'encadrant · e exerce à la fois une gestion par compétences, une mise en oeuvre des objectifs et des relations interpersonnelles d'ordre professionnel de haute intensité. La relation est explicitée en amont sur les droits et devoirs de chacun · e : c'est un contrat social, un pacte scientifique et professionnel."* [HHLGN22a] Les auteurs rappellent le lien avec l'apprentissage de compétences, de manières de faire, de penser et d'être (*hard skills, soft skills*) mais aussi avec le mentorat, pour inspirer sans contraindre.

— Le second volet est plus une réflexion sur les pratiques. On y trouve le besoin de pédagogie dans *"Notre conviction est qu'encadrer, c'est d'abord enseigner"* [HHLGN22b] que nous évoquons de manière plus large à travers l'apprentissage à la page 234. Les auteurs propose un référentiel des tâches (pas de compétences), en les reliant à l'éthique et aux responsabilités qu'elles engagent. Les tâches suivantes sont discutées : 1 Dire oui, dire non, 2 Trouver un financement, 3 Construire le projet, 4 Installer, 5 Communiquer et réagir, 6 Accompagner, 7 Former, 8 Écrire, 9 Socialiser, 10 Un · e mentor pour longtemps. Les propos sont illustrés par des situations concrètes et inspirées de la réalité.

Finalement, la réflexion menée semble porter plus sur faire en sorte que le doctorat ne se passe pas mal alors que nous développons plus les compétences à développer par les encadrants pour que ça se passe bien.

Remarque

Un enseignant-chercheur est missionné pour des activités d'enseignement, de recherche et d'administration, dont fait partie l'encadrement. Durant la thèse, on est est formé à la recherche mais pas à l'enseignement ou à l'encadrement. Pour ces deux dernières missions, on apprend par la pratique "sur le tas".

II) Proposition

En l'absence de référentiel pour l'encadrant, je propose donc une réflexion sur le sujet, qui pourrait ultérieurement être déclinée en compétences plus fines. Pour la présenter, prenons le petit modèle de classes de la FIGURE 8.1.

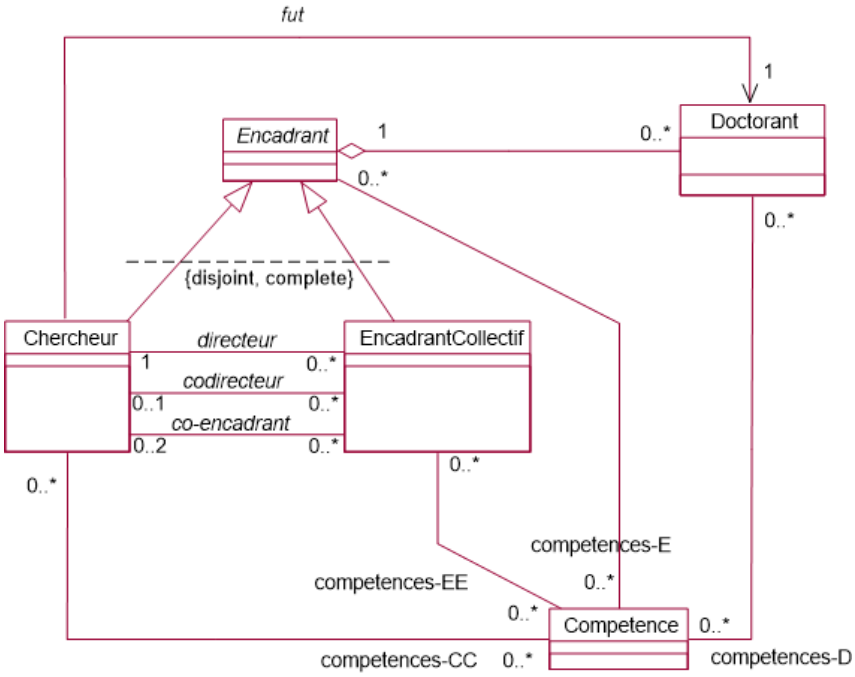


FIGURE 8.1 – Modèle statique des compétences (diagramme de classes UML)

Les compétences competences–D du Doctorant sont celles du référentiel du chercheur de la section 8.2.1. On les suppose acquises par le Chercheur, qui fut auparavant Doctorant. Les compétences de l' Encadrant sont clairement des compétences de *management*. Inversement, on n'est pas encore dans les compétences de pilotage du directeur de recherche qui gère un ensemble de recherches (*cf.* Section 8.2.3) comme un manager en entreprise qui gère un portefeuille de projets. L'échelle est donc plus réduite car on considère le seul projet de thèse[12] et j'omets volontairement les aspects financements de la thèse. Plus précisément, l'encadrant(e) doit à la fois en faire moins qu'un chef de projet (de thèse), un seul projet est géré mais il doit en faire plus car l'objectif est **moins de faire faire au doctorant des tâches qu'apprendre au doctorant à les faire**, l'encadrant est un tuteur, il ne fixe pas le chemin mais aide à le trouver.

Comme le montre la FIGURE 8.1, il y a deux cas bien distincts en termes de compétences d'encadrement : le cas d'un encadrement individuel et l'encadrement collectif. Dans le modèle, Encadrant est une classe abstraite qui a deux sous-classes distinctes. L'encadrement collectif doit posséder de nouvelles competences–EE de travail en équipe pour parvenir aux competences–E de l'encadrement individuel. De plus chaque chercheur doit

12. On trouve le cas particulier de thèses financées sur projet, pour lequel le projet de thèse doit se combiner avec le projet de recherche, et les thèses Cifre pour lesquels les enjeux de l'entreprise interfèrent aussi avec le projet de thèse.

avoir des competences–CC de chercheur collectif pour travailler en équipe, qui complètent les competences–E d'un encadrant individuel.

> Définir des contraintes ensemblistes, en OCL par exemple, sur les différents ensembles de compétences n'a pas vraiment de sens ici car on parle de type de compétences différentes. Il faudrait définir des sous-classes de Competence.

Encadrement individuel (tutorat)

Me basant sur ma propre expérience, j'estime, de prime abord, que l'encadrant(e) doit posséder deux qualités : **pédagogie** et **empathie**. En effet le rôle est d'accompagner le ou la doctorant.e dans son parcours, pas de diriger au sens strict du terme, il n'y a pas de chef mais plutôt un tutorat. Noter qu'il existe un dispositif complémentaire à l'université de Nantes, dit de mentorat, qui est un accompagnement indépendant et non intrusif sur le déroulement de la thèse. Le savoir-être prime sur le savoir-faire de gestion de projet (découpage des activitées, ordonnancement, répartition...). Il s'agit de compétences de management des personnes et d'enseignement. Détaillons quelques competences–E d'Encadrement.

E1 (Pédagogie)

Transmettre le savoir-faire pour arriver à une autonomie dans l'organisation.

L'idée est de mettre en situation pour faire réfléchir à comment faire les choses. Cette compétence couvre différentes facettes. L'encadrant doit non seulement donner des exemples mais aussi montrer l'exemple. L'accompagnement dans les premières réalisations peut se faire en co-construction. On doit alors guider le doctorant à organiser son activité sans définir et planifier précisément.

E2 (Empathie)

S'adapter au doctorant, comprendre son caractère ou son fonctionnement trouver ce qui fait avancer.

La transmission n'est pas un processus systématique. Elle s'adapte aux qualités et compétences du doctorant. Certains aiment avancer en autonomie, savent la direction à suivre, d'autres pas. Le doctorant a des affinités pour certains types d'activités (analyser, réfléchir, expérimenter, tester, présenter les travaux), le déroulement en tient compte. J'ai suivi des doctorants très différents. Certains, indépendants, avancent seuls et rendent compte de leur avancement ; d'autres ont besoin de rendez-vous réguliers. Certains doctorants ont besoin d'être accompagnés. L'empathie n'est pas le laisser-faire. S'adapter c'est alors trouver la forme de discours pour avancer (encourager, fixer des objectifs, évaluer les résultats, challenger...), refaire différemment, changer d'orientation.

E3 (Introspection)

Réfléchir au déroulement de la thèse et à son approche d'encadrement.

Toutes les thèses ne vont pas à la même vitesse et tous les doctorants ne butent pas sur les mêmes difficultés. Les sujets sont tous différents. Au delà de cette variété, une manière de procéder convient à certains mais pas à d'autres. L'encadrant doit se remettre en cause en cas d'échec ou de difficulté, analyser les causes pour savoir si le problème vient du sujet, du doctorant ou de la façon de procéder. Sur certaines thèses, nous avons dû improviser en fonction des échecs (ou succès).

E4 (Agilité)

Adapter son approche en continu en fonction des résultats et de la montée en compétences.

Si une méthode ne convient pas, on essaie une autre. Par exemple on suggère telle ou telle activité en autonomie, si elle échoue on peut la faire ensemble et collaborer. On peut aussi alterner les activités de découverte, d'analyse, de proposition, de production en fonction de l'avancée. La difficulté est réelle quand la montée en compétence ne se fait pas ou tellement lentement qu'elle met en péril l'avancement.

A l'impossible nul n'est tenu, face à un doctorant non motivé, on peut déployer différentes stratégies pour le même résultat décevant, le CSI est alors une évaluation objective intéressante pour juger de la situation.

La gestion de projet, l'évaluation de charge, la planification sont des savoirs-faire opérationnels importants mais ne constituent pas pour moi des compétences à atteindre dans le rôle "encadrant" car c'est difficilement répétable, car chaque sujet est différent et chaque doctorant est différent. Il n'y a pas non plus un processus global systématique pour aborder les problèmes d'un encadrement de thèse qui garantisse la réussite. De même, gérer le temps est important dans le déroulement d'une thèse, mais là encore, c'est la rigueur du suivi qui importe plus qu'une compétence spécifique : la compétence `RE4` prime. Finalement les blocs 1 et 2 du Référentiels de compétences du chercheur de la section 8.2.1 couvrent ce point.

Notons enfin, que ces compétences lorsqu'elles sont appliquées, prémunissent en grande partie des problèmes de discrimination, de harcèlement et d'éthique. Si l'éthique fait partie de la formation doctorale, la lutte contre les harcèlements et inégalités sont l'affaire de tous et des formations doctorales sont proposées pour les doctorants et les encadrants.

Encadrement collectif (équipe)

L'encadrement collectif donne un rapport de forces inverse à celui d'une gestion de projets : plusieurs personnes dirigent une personne. L'idée est de travailler ensemble pour un objectif commun, la réussite de la thèse. Dans l'encadrement collectif, on souhaite atteindre les competences–E de manière collaborative, avec malgré tout des rôles plus ou moins formels dans la prise de décision comme directeur, co-directeurs et co-encadrants. Le directeur doit être rigoureux dans les procédures administratives mais plus s'effacer (en tant que chef) dans la collaboration. Je ne considère pas ici les cas où le directeur n'a qu'un rôle administratif, lié à son statut HdR par exemple.

Me basant sur ma propre expérience, j'estime que l'encadrement collectif doit posséder trois qualités : **cohérence**, **complémentarité** et **complétude**. L'objectif est de pouvoir, non pas parler d'une seule voix mais de tenir le même discours, à variante près. Si les encadrants ne sont pas d'accord, le doctorant se trouve tiraillé et désorienté. La complétude signifie que l'ensemble des compétences de chacun couvre les compétences attendues pour l'encadrement. La complémentarité signifie qu'enlever un encadrant diminue le spectre des compétences. Le collectif doit disposer des compétences d'Encadrement en Equipe competences–EE.

EE1 (Collaboration)

Savoir organiser le travail d'équipe.

Cette compétence collective va influer sur l'organisation de l'encadrement, la distribution des rôles, la structure hiérarchique ou collaborative des processus de décision et processus opérationnels. Elle est liée à l'expérience et l'histoire de chacun, aux compétences et caractères individuels, aux égos, etc. Chacun doit trouver un rôle à sa mesure et il faut pouvoir répartir des rôles différents. Cela devient une compétence du groupe quand elle émerge des compétences individuelles `CC1` et `CC2`.

EE2 (Union)

Pouvoir décider ensemble, de manière solidaire et démocratique.

Cette compétence est relative au pilotage ou la stratégie. On doit fixer le cap, prévoir, anticiper et décider. Si les points de décisions sont émergents alors ils sont moins remis en cause.

EE3 (Ouverture d'esprit)

Pouvoir intégrer des différences.

Détaillons quelques Compétences de Collaboration des chercheurs competences–CC amenant aux compétences d'encadrement collectif competences–EE.

CC1 (Collaborateur)

Pouvoir travailler en équipe. Savoir s'écouter, se comprendre, s'adapter, s'accepter, se faire confiance.

Cette compétence regroupe diverses compétences de savoir-être qui mènent à une certaine unité d'action dans la réalisation des compétences `E1` à `E4` en particulier pour la communication. On retrouve ici aussi la bienveillance mais aussi la vigilance.

CC2 (Initiative)

Pouvoir prendre sa place, se situer, suggérer.

Si la compétence `CC1` se focalise sur les relations, la compétence `CC2` se focalise sur l'action, on analyse, on propose, on agit. Le chercheur doit être actif dans le groupe.

CC3 (Flexibilité)

Pouvoir s'adapter à chaque interlocuteur.

Cette compétence, qui va de pair avec `EE3`, est indispensable dans les collaborations pluridisciplinaires et/ou internationales.

Remarque

Plus la taille de l'équipe est importante, plus il est difficile d'atteindre les compétences collectives à cause de la variété des compétences et des caractères individuels.

Notons enfin que les écoles doctorales et le ministère progressent dans le sens d'une meilleure définition et un meilleur suivi des encadrements des thèses. La réflexion, dans ce cas, vise aussi un objectif supplémentaire, celui d'éviter des problèmes humains durant la thèse tels que mentionnés dans [HHLGN22b].

Les compétences mentionnées dans cette section, s'appliquent à tout encadrant(e) et donc implicitement à ceux qui dirigent des travaux de recherches. Mais le directeur doit aller plus loin puisqu'il doit manager les équipes, les thématiques et les projets.

8.2.3 Référentiels de compétences d'un directeur de recherche

Le rôle de directeur de recherche est de piloter et gérer les recherches, il doit donc avoir des compétences de **management**.

I) Etude de l'existant

Le management est un domaine vaste et complexe. Le lecteur trouvera dans [BHHM19], un tour d'horizon très accessible et dans [Juë20] des fiches pratiques pour piloter une

équipe. On *manage*[13] des entreprises, des organisations, des associations, des projets, ou plus spécifiquement les ressources, la qualité, la performance, l'innovation, les systèmes d'information... Dans [BHHM19], il n'y a pas une définition mais trois définitions qui couvrent différents aspects autour de l'organisation, les activités et les ressources, dont les individus. Les fonctions principales du *manager* se regroupent en cinq catégories : (1) prévoir et planifier, (2) organiser, (3) diriger et être un leader, (4) coordonner et (5) contrôler. On considère habituellement trois niveaux de management, le niveau **stratégique** (cadres dirigeants ou *top management*), le niveau **tactique** (cadres intermédiaires ou *middle management*) et le niveau **opérationnel** (cadres fonctionnels ou *operational management*). Les auteurs de [BHHM19] définissent des proportions de types de compétences associées (*cf.* FIGURE 8.2). Un cadre intermédiaire, tel qu'un directeur de recherche, doit couvrir tous les domaines de compétences. Les connaissances et compétences y sont classées en 4 catégories : 1. analytiques, 2. humaines, 3. techniques, 4. interculturelles. Les catégories 1 & 3 sont de type *hard skills* (savoirs et savoirs-faire), les catégories 2 & 4 *soft skills* (savoirs-être). *"Les compétences d'un manager s'affinent et s'élargissent avec l'expérience"* [BHHM19].

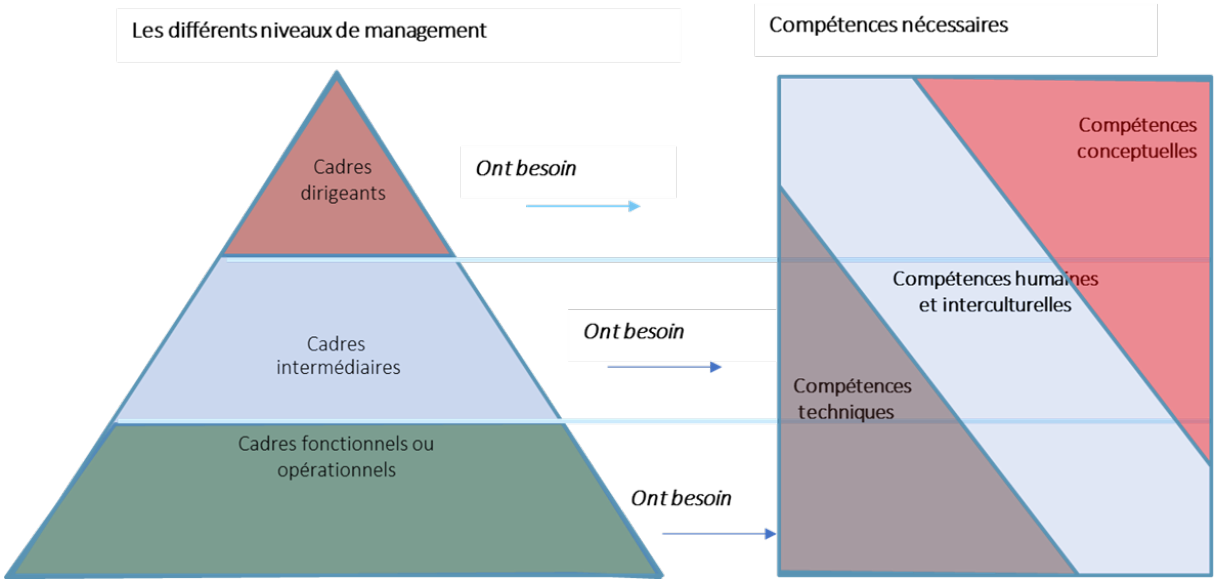


FIGURE 8.2 – Compétences et hiérarchies du management [BHHM19]

En termes de référentiels de compétences du management, il semble y avoir autant de référentiels que d'organisation et une bonne pratique pour les entreprise est de définir un référentiel personnalisé qui prend du sens. *"Le référentiel de compétences managériales est à la fois un outil et une méthodologie d'évaluation des compétences des managers. Il est élaboré, le plus souvent, par le service des ressources humaines. Ce référentiel permet de définir les compétences stratégiques pour l'entreprise et de les mesurer, de partager les besoins en compétences au sein des parties prenantes internes et de renforcer la culture*

13. J'utiliserai aussi le terme "gérer" mais il est plus restrictif que "manager" notamment dans le rôle de pilotage des personnes et de la communication. Des différences sur la terminologie se trouvent sur le Web.

d'entreprise" , Le figaro [14]. L'exemple d'évaluation de compétence de la FIGURE 8.3 [15] montre que la liste des compétences peut être longue si on entre dans les détails.

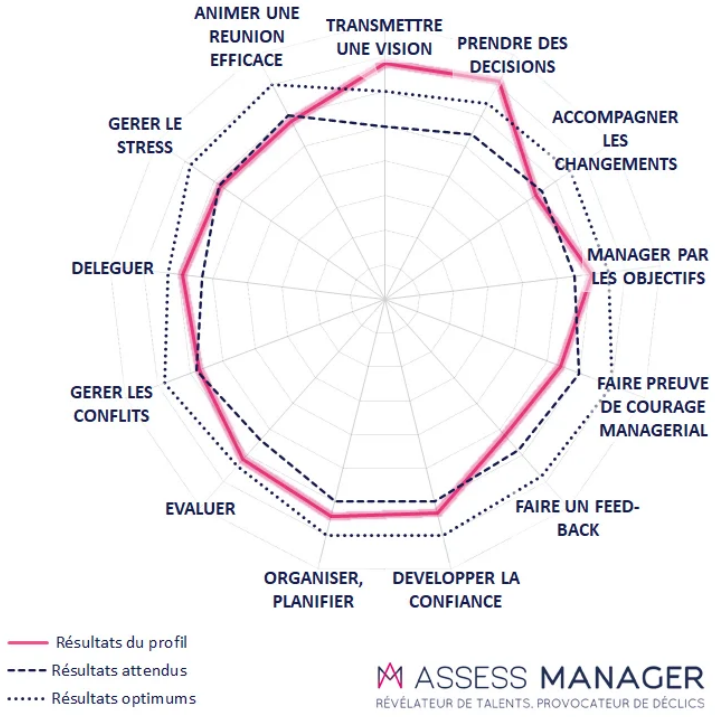


FIGURE 8.3 – Radar des compétences managériales[15]

Au niveau RNCP, la fiche RNCP36493 "Manager des entreprises et des organisations" [16], propose une certification pour les chefs d'entreprise sur de nombreuses compétences organisées en quatre blocs de compétences communs à l'ensemble des parcours ainsi que d'un bloc de spécialisation à choisir parmi 3. Les compétences recouvrent surtout les activités de gestion d'entreprise dans son contexte économique, financier et légal.

Du point de vue institutionnel, on trouve un référentiel sur le site du Centre national de la fonction publique territoriale [17] qui détaille les activités/compétences en savoir-faire et savoirs pour les cadres de FPT. Curieusement, le savoir-être n'est pas traité, alors qu'il est souvent vu comme prioritaire dans les entreprises. Sur le site du ministère de la transformation publique, se trouve un référentiel de pratiques managériales dans un environnement complexe [18]. Il regroupe les savoir-faire et pratiques par fonction managériale (pilote, animateur, contributeur, coach, leader) avec une grille d'évaluation qui mènent à un graphique radar du type de la FIGURE 8.3.

14. https://emploi.lefigaro.fr/evolution-professionnelle/guide-de-l-evolution-professionnelle/1147-competences-manageriales-liste-et-exemple/
15. extrait de https://www.assess-manager.com/referentiel-de-competences-en-management
16. https://www.francecompetences.fr/recherche/rncp/36493/
17. https://www.cnfpt.fr/evoluer/lemploi-fpt/repertoire-metiers/referentiel-managementencadrement/national
18. https://www.modernisation.gouv.fr/files/2022-05/ReferentielsPratiquesManageriales MaturiteEquipe.pdf

Ce petit tour d'horizon montre que l'existant est riche mais il n'y pas un référentiel directement applicable à la situation du directeur de recherche. Cette richesse est par contre une source d'inspiration pour produire et valider un référentiel.

II) Proposition

En l'absence de référentiel pour le directeur de recherche, je propose donc une réflexion sur le sujet, qui pourrait ultérieurement être déclinée en compétences plus fines.

Si on considère un laboratoire de recherche comme une entreprise, le niveau *stratégique* est celui de la direction du laboratoire, le niveau *tactique* est celui des équipes (ou groupes d'équipes) et le directeur de recherche apparait plutôt au niveau *opérationnel* du management (*cf.* page 238). Le périmètre de management considéré est "diriger les recherches et des chercheurs" et non diriger un laboratoire avec des personnels administratifs et techniques. Le directeur de recherche (DR) est plus *manager* fonctionnel que hiérarchique (sauf pour les doctorants, les personnels techniques ou les personnels contractuels d'un projet). Il doit avoir une vision globale sur ses thématiques et la décliner dans des activités opérationnelles de recherche, dont l'encadrement de thèse n'est qu'un exemple. On peut classer ces activités simplement en deux catégories : les sujets et les projets. Ces derniers impliquent une gestion des ressources (humaines, financières, techniques), une planification dans le temps, une gestion financière, etc.

Proposons quelques *groupes de compétences DR*. Pour éviter des confusion, je mentionnerai "groupe de recherche" plutôt que "équipe de recherche" qui relève souvent de l'organisation structurelle d'un laboratoire.

DR1 (Stratégie de recherche)

Savoir structurer l'activité de recherche en sujets et projets. Savoir positionner cette activité dans un écosystème plus large.

Pour diriger des recherches il faut savoir où on va et comment on y va. Cette compétence est relative aux choix thématiques, à la priorisation des sujets et à la structuration des activités pour chaque sujet. Outre la connaissance et la maîtrise du domaine (on retrouve ici de manière plus exigente les compétences des Blocs 1 & 2, mais aussi la partie démarche du Bloc 6 du référentiel du doctorant). Des sous-compétences y sont associées telles que la capacité d'innovation, l'opportunisme, l'ambition.. Une partie des sujets donne lieu à des projets, qu'il faut monter en collaboration avec d'autres chercheurs auprès de financeurs.

Le second point relève de la maîtrise de l'environnement (acteurs majeurs, courants, ...), de l'habilité **politique** avec des compétences de négociation et d'association.

DR2 (Visibilité)

Savoir connaître et faire connaître son groupe de recherche (réseau, rayonnement).

Cette compétence est indispensable pour établir des collaborations de recherche. Elle se développe avec l'expérience par les conférences, les échanges, les commissions, les groupes thématiques, etc. Il faut s'informer et tenir informer la communauté. Bien sûr cela dépend des thématiques, l'écosystème varie d'une thématique à l'autre. Comme tout réseau social, cette compétence est exponentielle, plus on est connu plus on est invité. On retrouve aussi ici des compétences, à un niveau plus exigeant, des Blocs 3, 4 & 5 du référentiel du doctorant.

DR3 (Encadrement)

Pour chaque sujet de recherche, être capable d'organiser le travail du groupe.

On retrouve ici des compétences de travail collaboratif (`CC1` à `CC3`) et management d'équipe (`EE1`, `EE2`) car les sujets et même les projets de recherche sont essentiellement collaboratifs que ce soit pour la direction ou les *work packages* d'un projet de recherche multi-équipes. On retrouve aussi ici des compétences, à un niveau plus exigeant, du Bloc 6 du référentiel du doctorant.

DR4 (Gestion de projet)

Pour chaque projet de recherche, être capable de piloter le projet et gérer les ressources.

On trouve ici un bloc de compétences original par rapport aux compétences du doctorant ou de l'encadrant. De la génèse à l'achèvement du projet, le porteur de projet ou le *steering committee* doivent gérer les aspects scientifiques, techniques, organisationnels, humains, temporels, légaux... Ilustrons cela par les compétences attestées de la fiche RNCP23805 - Manager de projet [19].

Le Manager de Projet doit être capable d'exercer sa responsabilité de manière totalement autonome (en respect avec l'organisation de son lieu de travail) dans les activités suivantes :

- *I Aligner l'exécution du projet sur les objectifs de ses commanditaires / Piloter et coordonner les différents acteurs / Traduire les besoins exprimés en actions concrètes pour l'équipe projet / Valider les livrables du projet / Organiser le contrôle qualité du projet*
- *II Cadrer les actions de l'équipe projet / Planifier / Allouer les ressources nécessaires / Maîtriser l'échéancier du projet / Gérer les risques du projet*
- *III Etablir le budget prévisionnel du projet et le projeter dans le temps / Maîtriser les couts / Elaborer la stratégie achat du projet / Piloter la sélection des fournisseurs et les négociations*
- *IV Recruter l'équipe projet / Animer et motiver l'équipe projet / Assurer la cohésion de l'équipe et la coopération entre ses membres / Suivre, piloter la performance et gérer les compétences de l'équipe projet*

19. `https://www.francecompetences.fr/recherche/rncp/23805/`

V Identifier les parties prenantes du projet et instaurer un dialogue avec elles / Assurer leur engagement / Elaborer la stratégie de communication du projet / Assurer la diffusion de l'information relative au projet

La gestion des risques est aussi un point important dans la gestion des projets (recrutement, dysfonctionnement, malentendus, désaccords, *egos*, engagement, difficultés techniques....).

DR5 (Coordination de projets)

Etre capable de piloter un portefeuille de projets et en gérer les ressources transverses.

Gérer plusieurs projets implique une macro-coordination qui n'est pas simple car chaque projet a un contexte différent, des interlocuteurs différents, des attentes différentes, un calendrier différent, des ressources différentes mais parfois partagées notamment au niveau des participants. L'idéal est d'avoir un pool de sujets et projets complémentaires dans une thématique donnée. Ce n'est pas donné à tous les HdR de pouvoir se construire un tel environnement.

8.2.4 A propos des compétences d'évaluation

Bien que mentionnée un peu partout dans ce chapitre, un de mes rapporteurs remarquait, à juste titre, qu'"*il est d'ailleurs étonnant qu'évaluer soit une compétence qui n'apparait pas explicitement dans sa liste de compétences attendues, alors qu'elle est évidemment présente tout au long du mémoire, de son activité de recherche, de développement et de projet*". Cela nécessite donc un développement, surtout si on traite des compétences pour l'(auto)évaluation des compétences (sic !).

J'avais envisagé l'évaluation plus comme une activité que comme une compétence à l'instar de la remarque de la page 229 au sujet de la formulation ambiguë des blocs de compétence du doctorant entre activités et compétences pour le faire. Dans ce chapitre, le terme évaluation est surtout lié à l'évaluation des compétences mais l'évaluation est omni-présente dans la vie des chercheurs, quel que soit le rôle tenu. Le chercheur est évalué et évalue de manière individuelle et collective en permanence. On évalue les travaux, les résultats, les publications, les activités, les performances, les projets, les équipes, les laboratoires, les instituts... *Il faut donc être compétent pour le faire.*

EVAL

Etre capable d'évaluer objectivement, honnêtement et de manière juste quelque chose ou quelqu'un.

La compétence **évaluation** est à la fois transversale aux différents rôles et fractale sur l'objet évalué (activité, résultats, projets, coûts...), le contexte d'évaluation ou de décision (avancement, concours, individuel/collectif...) et les enjeux (label, financement, nomina-

tion, publication...). Elle fait référence à d'autres compétences *e.g.* Bloc 6 du docteur, compétence 9 du maître d'apprentissage, compétence E2 de l'encadrant, etc. L'évaluation passe toujours par le crible de ses propres exigences et de ce point de vue on dira qu'elle est juste si on n'exige pas plus des autres que de soi-même. On rejoint les compétences liées à l'éthique, aux conflits d'intérêt, aux conséquences pour soi-même et pour les autres. L'ajouter au référentiel n'est donc pas simple, il faudrait le faire pour chaque rôle et décliner sur les objets/personnes à évaluer. Je ne l'ai pas fait ici pour ne pas compliquer le discours.

8.2.5 Synthèse

Les différentes compétences abordées dans cette section sont additives, plus on couvre large, plus on doit avoir des compétences, comme l'illustre la Figure 8.4 en correspondance avec Figure 8.1.

Noter que toutes ces compétences identifiées constituent plutôt des blocs de compétences qu'il serait nécessaire d'affiner en compétences plus fines. Nous ne l'avons pas fait ici car l'idée est de réfléchir au sujet, notamment pour structurer mon bilan de compétences, mais pas de fournir un référentiel complet pour une certification.

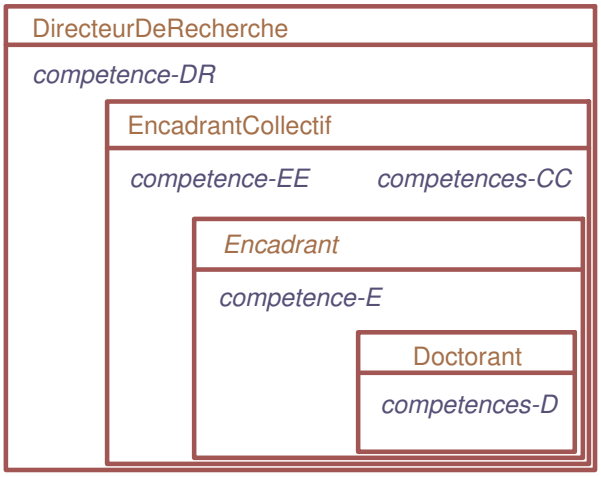


Figure 8.4 – Compétences globales

8.3 Compétences

Dans cette section, l'idée est de faire un retour d'expérience (REX) à travers le filtre des compétences de la section précédente. Il s'agit donc d'un *bilan subjectif de compétences*. C'est un exercice personnel qui ne présume en rien d'une évaluation par des pairs. La section est organisée en trois temps : synthèse du référentiel global, auto-évaluation, retours d'expérience.

8.3.1 Référentiel global

Je synthétise ici les compétences de la section de la section 8.2, qui rassemblées forment un référentiel global.

Bloc 1 - Conception et élaboration d'une démarche de recherche et développement, d'études et prospective

Bloc 2 - Mise en œuvre d'une démarche de recherche et développement, d'études et pros-

pective

Bloc 3 - Valorisation et transfert des résultats d'une démarche R&D, d'études et prospective

Bloc 4 - Veille scientifique et technologique à l'échelle internationale

Bloc 5 - Formation et diffusion de la culture scientifique et technique

Bloc 6 - Encadrement d'équipes dédiées à des activités de recherche et développement, d'études et prospective

E1 Transmettre le savoir-faire pour arriver à une autonomie dans l'organisation (pédagogie).

E2 S'adapter au doctorant, comprendre son caractère ou son fonctionnement trouver ce qui fait avancer (empathie).

E3 Réfléchir au déroulement de la thèse et à son approche d'encadrement (introspection).

E4 Adapter son approche en continu en fonction des résultats et de la montée en compétences (agilité).

EE1 Savoir organiser le travail d'équipe (collaboration).

EE2 Pouvoir décider ensemble, de manière solidaire et démocratique (union).

EE3 Pouvoir intégrer des différences (ouverture d'esprit).

CC1 Pouvoir travailler en équipe. Savoir s'écouter, se comprendre, s'adapter, s'accepter, se faire confiance (Collaborateur).

CC2 Pouvoir prendre sa place, se situer, suggérer (initiative).

CC3 Pouvoir s'adapter à chaque interlocuteur (flexibilité).

DR1 Savoir structurer l'activité de recherche en sujets et projets. Savoir positionner cette activité dans un écosystème plus large. (stratégie de recherche, politique).

DR2 Savoir connaître et se faire connaître (réseau, visibilité).

DR3 Pour chaque sujet de recherche, être capable d'organiser le travail d'équipe (compétences du bloc 6).

DR4 Pour chaque projet de recherche, être capable de piloter le projet et gérer les ressources (Gestion de projet).

DR5 Etre capable de piloter un portefeuille de projets et en gérer les ressources transverses (Coordination de projets).

EVAL Etre capable d'évaluer objectivement, honnêtement et de manière juste quelque chose ou quelqu'un (fractal).

8.3.2 Auto-évaluation

Commençons par collecter et rappeler les compétences informelles synthétisées tout au long du manuscrit, pour les placer dans les référentiels de la Section 8.2.

C1 Comprendre, structurer (classifier) et expliquer le domaine de recherche sont des compétences de base pour positionner un travail de recherche innovant.
⇒ *contribue aux compétences Bloc 1, Bloc 4*

C2 Identifier les problèmes, les ordonner, imaginer des pistes ambitieuses, proposer des pistes réalistes, ordonnancer un ensemble d'activités de recherche pour structurer un travail de recherche scientifique sur du long terme.
⇒ *contribue aux compétences Bloc 2*

C3 Organiser une recherche collaborative, détecter des compétences, définir des complémentarités et coordonner pour contribuer à un travail de recherche d'envergure.
⇒ *contribue aux compétences Bloc 6, DR1*

C4 Animer des projets de recherche et fédérer les énergies en lien avec la compétence C2 pour rationaliser les travaux de recherche.
⇒ *contribue aux compétences DR1, DR3*

C5 Investir de nouveaux domaines, collaborer avec des chercheurs d'autres spécialité implique de se comprendre et se faire comprendre, les apports croisés permettent d'innover dans les solutions des problèmes identifiés.
⇒ *contribue aux compétences CC1, CC2, CC3*

C6 Expliquer et justifier les travaux, les étendre à de nouveaux domaines, transférer les compétences, diffuser les savoirs sont indispensables à la valorisation du travail de recherche.
⇒ *contribue aux compétences Bloc 3*

C7 Agir dans des communautés thématiques, des groupes de recherches, des réseaux en lien avec la compétence C1 permet d'étendre les périmètres et innover.
⇒ *contribue aux compétences Bloc 5, DR2*

Les chapitres du manuscrit décrivent les travaux scientifiques, il est naturel que les compétences d'encadrement E* et EE* n'apparaissent pas explicitement. Elles seront illustrées dans la section 8.3.3. La partie gauche de la FIGURE 8.5 est un rappel, extrait de la FIGURE 1, de la réflexion initiale sur l'HDR, focalisé sur le sujet des compétences. Le contenu de ce chapitre a mis en évidence une réflexion plus poussée sur le sujet. Je me

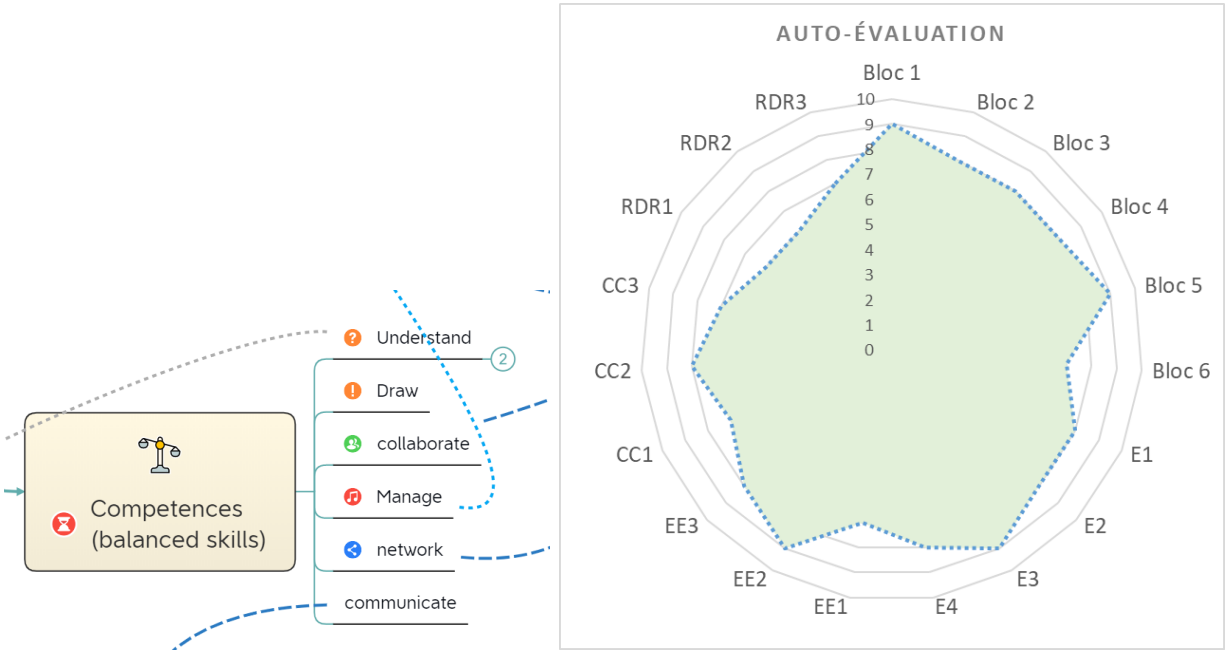


FIGURE 8.5 – Evaluation de compétences HDR

suis auto-évalué sur le référentiel global de compétences décrit ci-dessus. Le radar de la partie droite de la FIGURE 8.5 évalue quantitativement le niveau estimé. Une marge de progression existe, notamment sur les compétences de type DR*.

Comme nous le demandons aux étudiants dans leur présentations de projets ou de stage, il faut être capable de justifier les niveaux auto-attribués (preuves, métriques, justifications...). C'est une partie "réflexive" de la compétence EVAL et elle fait l'objet de la section suivante.

8.3.3 Retours d'expérience

Pour commenter l'auto-évaluation, j'aurais pu suivre la même organisation que le référentiel de la section 8.2 (chercheur/encadrant/directeur), ou structurer par catégorie de compétences (savoirs/savoirs-faire/savoirs-être). J'ai choisi une approche thématique, plus transversale, par type d'activité : contributions scientifiques et techniques, collaborations, encadrer, travailler en mode projet...

A) Contributions scientifiques et techniques

Je me considère comme un *Docteur généraliste en Génie Logiciel et Systèmes d'information*[20] et non un spécialiste pointu d'un sous-thème précis. J'estime avoir une bonne maîtrise du périmètre autour du génie logiciel, des modèles et de la vérification (Blocs 1& 4, DR1) du fait de la variété des sujets abordés mais aussi parce que ce sont des sujets que j'ai pu enseigner. J'ai toujours aimé lire, analyser, classifier les références de la littérature. Plus récemment, je l'ai abordé d'un point de vue plus méthodologique dans [Kai+23 ; Teb+23a ; And+23] et dans une formation doctorale dispensée depuis 2023.

J'ai toujours aimé réfléchir aux problèmes rencontrés, trouver des solutions et j'ai donc développé des compétences d'analyse/proposition en intégrant la notion de risque, cela s'applique à tout types de problèmes (techniques, organisationnels, humains) et donc de compétences de savoir-faire (compétences Bloc 2, EE2, EE3, EE4).

D'un point de vue technique, je peux participer aux développements sans être expert du sujet, et aime me frotter au côté expérimental de la validation des propositions (compétence Bloc 3). Mes publications de type proposition ou contribution contiennent une partie mise en œuvre et expérimentation. Les développements prennent du temps et il faut que ce soit du temps régulier avec des montées en charge pour de nouvelles fonctionalités. Avoir des financements pour des développeurs est un atout important. On retombe sur le financement de la recherche. La difficulté avec les expérimentations est de trouver un terrain adapté avec des cas si possible concrets. Nous avons lontemps eu ce problème avec Kmelia, jusqu'à l'application aux processus manufacturiers. Nous le rencontrons à nouveau sur l'alignement BITA dans lequel le patrimoine applicatif doit être significatif.

Comprendre et expliquer allant de pair, la communication est indispensable (Blocs

20. `https://www.linkedin.com/in/pascal-andr%C3%A9-bb7696a/`

4& 5). La rédaction n'a jamais été un obstacle, cependant l'expérience et les expériences (bonnes ou mauvaises) m'ont permis de monter en compétence à ce sujet, et aussi de l'expliquer dans des cours. Savoir à qui on s'adresse, quel est le message à faire passer, s'avoir s'adapter au lecteur, savoir dimensionner la communication sont des compétences qui s'apprennent au cours du temps. Faire passer des articles en conférence est un art, surtout lorsqu'on n'est pas une célébrité... Etant timide, la communication orale m'a obligé à développer la structure du discours et à préparer plus soigneusement les interventions. L'improvisation est aussi plus délicate en anglais. Ici aussi l'expérience m'a permis de mieux comprendre comment construire et faire passer les messages. Ce n'est pas parfait, mais ça reste efficace, notamment par l'usage du schéma comme support au discours.

La manière d'atteindre ces contributions ou leur organisation fait appel à d'autres compétences que nous verrons dans la suite.

B) Collaboration, coopération

Je ne conçois pas de contribution scientifique et technique significatrice qui soit uniquement individuelle. Le résultat est plus probant lorsqu'on coopère ou collabore. La différence entre coopération et collaboration est que dans le second cas, chacun procède comme il l'entend [21], pour le reste on travaille sur un objectif partagé, en jouant des rôles de réalisateur et de décideur. En recherche institutionnelle, on collabore car il n'y a pas vraiment de relation hiérarchique entre les chercheurs sauf pour les doctorants, post-doctorants, stagiaires, ingénieurs et techniciens dans des recherches financées (bourses, projets,...) ou les étudiants dans les projets tutorés. De fait on est amené à travailler en équipe, et pour cela se mettre d'accords sur les objectifs, les moyens, les plannings, les publications... Concrètement, je travaille en équipes pédagogiques (Miage, LEA, eMiage : pour les formations, les modules ou les projets pédagogiques) et de recherche (équipe, thèmes, projets, échanges, rencontres, manifestations...). En recherche universitaire, la collaboration volontaire est le seul mode de travail collectif (en équipe). Les formes et cadres de coopération et collaborations sont très divers : nombre et types de partenaires (universitaire, industriel), types de contribution (étude, synthèse, méthode, produit), type de contrat (volontariat, collaboration, prestation, transfert...). Une Cifre ou un contrat industriel sont souvent centrés autour d'un besoin plus ou moins formalisé, avec finalement deux rôles, le commanditaire et le prestataire ; la collaboration de recherche est de type négociation en ce sens que tout le monde, et surtout le commanditaire doit être satisfait (gagnant ?). Dans un projet émergent entre chercheurs, il y a du *leadership* mais pas d'autorité, il faut convaincre mais décider ensemble. Dans tous ces cas, le consensus est de mise. Il faut savoir être à l'écoute, comprendre, analyser, être force de proposition, convaincre. Le porteur (et son équipe) joue un rôle essentiel dans l'ambiance du projet et la relation entre participants, néanmoins chacun doit aussi s'adapter et prendre sa place. Dans un projet avec financeur institutionnel ou privé, le porteur (la porteuse) du projet

21. `https://interpole.xyz/?CooperationOuCollaborationQuellesDifferen`

joue un rôle majeur dans la relation et doit savoir prendre du recul pour rendre compte (*reporting*). Prenons quelques exemples vécus.

Collaboration mono-équipe et ou mono-thème Le projet Kmelia (*cf.* Chapitre 3) est un projet collaboratif d'une même équipe de recherche. Sans support financier, il est basé sur le volontariat : chacun y adhère ou pas, les décisions sont discutées et adoptées ensemble. Son horizon est à moyen terme (depuis 2005) et comprend des sous-projets (projets étudiants, thèses, développements...), des groupes de travail thématiques (langage, vérification, développement, ...). L'avancée dépend non seulement de l'investissement de chacun mais aussi de la synchronisation des périodes d'activités, des relations humaines. D'un point de vue individuel, il faut faire preuve d'engagement, de solidarité, de ténacité mais aussi de tolérance, de patience, d'acceptation de choix ou de frustration quand d'autres ne suivent pas. J'ai été impliqué sur l'ensemble des activités de ce projet.
On trouve le même type de collaboration avec des chercheurs d'autres équipes mais sur un thème similaire. La localisation sur plusieurs sites ne favorise pas les échanges informels. En plus d'être disponible, il faut être mobile.

Collaboration multi-équipe Le LS2N est multisite, multi-établissement et multi-tutelle, il donne la possibilité de multiples collaborations. La difficulté est de se rencontrer et se connaître. Dans le cadre du thème transverse « Entreprise du futur » du LS2N [22], sous la coordination de Dalila Tamzalit, nous avons répondu à une demande d'Orange Labs [23] pour mieux orienter ses activités d'innovation sur le marché Entreprise. Pour ce projet `OrangeEdF`, Le LS2N a été sollicité pour un accompagnement sur l'automatisation et la transformation possible de la vie de l'entreprise et ses processus métiers clés au regard de l'accélération conjointe de l'innovation des technologies du numérique (Big Data, IoT, IA, ...). Ce travail collectif démarré en octobre 2016 a donné lieu à de nombreuses réunions et échanges avec différents collègues des équipes du thème transverse. Il s'est terminé en décembre 2017, deux rapports (un état de l'art et un document de perspective scientifique) ont été livrés et une présentation effectuée. Un financement a été dégagé pour le stage de Mr Tebib (voir la rubrique encadrement page 251). Les qualités importantes ici étaient d'être force de proposition, de prendre en charge des parties, de devancer les besoins, de suggérer pour aider la porteuse de projet, qui risque l'inertie du fait du nombre de (potentiels) participants, du manque d'implication de certains...
Dans le projet CIM-Anjou (*cf.* Section 7.1 du Chapitre 7), les partenaires sont issus de différents laboratoires et institutions de la région Pays de la Loire. Avoir un financement a permis, non seulement de recruter un ingénieur pour des développements, mais de financer des réunions de travail qui ont soudé les équipes. On trouve le même type de collaboration avec des chercheurs de laboratoires de villes différentes. Il faut être encore plus disponible et mobile, même si la visio-conférence (post Covid) est devenue un outil indispensable.

22. `https://www.ls2n.fr/theme/edf/`
23. `https://lelab.orange.fr/`

Collaboration internationale Dans une collaboration de recherche internationale, les participants sont issus du même domaine mais de culture différente. J'ai collaboré avec des chercheurs en Europe, Afrique et moyen-orient, Australie. La recherche d'affinités thématiques et la prise d'initiative sont importantes pour lancer de telles collaborations. Les échanges sont un plus. A l'inverse le temps qui passe met à mal les collaborations. Dans le projet `ECONET` (*cf.* Section 3.6 du Chapitre 3), les participants sont issus du même domaine mais de culture différente. L'échelle de temps est plus étirée, chaque chose prend plus de temps, et on doit être vigilant à l'interprétation des propos. Dans le projet ECONET, j'ai remarqué que la distance et l'éloignement induisent plus d'indépendance de décision des partenaires, la direction réclame plus de coordination.

Collaboration pluri-disciplinaire Dans un projet pluridisciplinaire (*cf.* Partie III), on doit s'adapter à une ou plusieurs autres disciplines avec des pratiques de recherche parfois très éloignées de celles de nos thématiques. Il faut faire preuve d'écoute et d'empathie, suggérer sans vouloir imposer sa vision, faire émerger une solution hybride qui mêle les acquis de chacun, savoir séparer l'essentiel et le secondaire, se remmettre en cause encore et toujours, ne pas laisser faire si on n'est pas convaincu du bien fondé...

J'ai été co-responsable scientifique dans un projet de partenariat avec `Ifremer` [Pel+99 ; Pel+21]. Le contrat portait sur la modélisation de pêcheries de 1998 à 2000. Il s'agissait de construire un simulateur de pêcheries complexes. La modélisation a été menée avec un groupe d'étudiant du DESS Génie Informatique de Nantes et le développement a été conduit selon une méthode propre basée sur UML et réalisé en Java, par des stagiaires issus de l'équipe de DESS. Le développement a ensuite été repris par la société Cogitec, partenaire du projet puis CodeLutin sous l'appellation ISIS-Fish. La collaboration est à la fois plus directive vis-à-vis des étudiants et plus stratégique ou partenariale avec le commanditaire.

Dans le projet `ONECAD` (*cf.* page 168), j'ai collaboré avec des chercheurs en géographie ou gestion, avec des utilisateurs finaux (collectivités locales et territorales, bureaux d'études, services ministériel). Chacun a un point de vue différent sur le projet. Il faut faire preuve de pédagogie pour expliquer, d'écoute pour comprendre et converger. Le management visuel, le prototypage ou la démonstration ont été nécessaires pour donner du sens aux mots. Travailler avec des spécialistes des IHM permet de prendre en compte le point de vue des utilisateurs finaux et l'usage numérique dans le(s) métier(s). Réduire les discussions techniques en sous comité avec le ministère a permis de comprendre les exigences techniques sur le code, la documentation la vérification et de s'aligner avec les pratiques pour le déploiement. On adapte son discours à son interlocuteur

Collaboration et compétences Ces projets m'ont permis de développer les compétences Blocs 1, 2 & 4, CC1, CC2, CC3 et collectivement les compétences EE1, EE2, EE3. la réussite des collaborations est liée à l'engagement individuel, à l'adhésion forte aux projets, le respect mutuel et le respect de ses engagements - les participants doivent donc

avoir, au moins en partie, les compétences CC1, CC2. Les égos sont un frein au travail collaboratif. Les projets ECONET et ONECAD m'ont plus particulièrement aidé à développer les compétences CC3, EE3 et ONECAD le Bloc 6 pour les personnels. Ces projets, le montage de projets non acceptés pour financement et les rencontres hors-projets avec d'autres chercheurs m'ont permis de monter en compétences sur `Bloc 1, CC1, CC2, DR1, DR2`.

C) Pilotage (produit, projet)

Toutes les collaborations ne sont pas de type projet et ne nécessitent pas de compétences de pilotage. Lorsque la collaboration porte sur un produit, une prestation spécifique, la collaboration doit suivre une gestion de projet plus précise. Non seulement des compétences de gestion de projet sont nécessaires (compétences DR3) pour gérer ressources, coûts, délais, structuration, répartition, planification...) mais il faut redoubler d'efforts pour faire adhérer les participants à la vision et gérer la(les) collaboration(s) (compétences EE1, EE2). L'intérêt individuel doit se fondre dans l'intérêt collectif pour conserver la motivation initiale.

J'ai constaté des échelles de temps différentes, liées notamment au caractère individuel, à la culture, aux responsabilités assumées, aux méthodes de travail. Ces dernières sont aussi influencées par les domaines de recherche (*e.g.* sciences dures, sciences du vivant, sciences humaines et sociales) ou la localisation (entreprise, université, centre de recherche). Une adaptation est donc nécessaire (compétences CC1, CC3).

Dans le projet Ifremer (*cf.* page 249), il y avait trois rôles dans le projet commanditaire (MOA), MOE et prestataire. Notre équipe était l'intermédiaire MOE. Les compétences de compréhension du besoin d'un côté et compréhension des contraintes techniques de l'autre sont assez classique en gestion de projet informatique.

La situation était plus complexe le projet ONECAD (*cf.* page 168). Sans tomber dans les clichés, les scientifiques structurent de manière plus rigoureuse l'organisation, le cadencement et la validation des activités. Pour planifier, il faut que les étapes de décisions ne soient pas remises en cause à court terme, il faut aussi anticiper les risques et bien poser les hypothèses. Une approche agile dans laquelle, la remise en cause intervient seulement au cycle suivant dans une approche itérative et incrémentale, permet de gérer de manière rationnelle à la fois le projet et les incertitudes. La négociation permet d'affiner les contrats lorsqu'une partie-prenante a un rôle de prestataire vis-à-vis d'une autre.

Dans le projet Orange EDF (*cf.* page 248), l'organisation était à la fois plus compliquée du fait du nombre de participants mais plus simple pour la production car un sous-groupe gère la cohérence globale, et un espace localisé était proposé pour les contributions locales.

C'est plus difficile quand il y a des dépendances, c'est-à-dire que la contribution de l'un influe sur la contribution de l'autre. Par exemple, dans le projet `CIM-Anjou` (*cf.* page 194), les modèles d'ateliers de notre partie servaient de support aux modèles de calcul d'ordonnancement du partenanire angevin.

C'est encore plus difficile quand il y a des dépendances croisées. Dans le projet RODIC (*cf.* page 221), les modèles de configurations du WP3 sont instanciés dans l'IHM du WP1, qui définit des scénarios de simulation du WP2 qui se basent sur des modèles de configuration du WP3. Il faut non seulement des compétences de structuration des activités mais aussi de définition des interfaces (contrats) entre WP et de négociation de périmètres.

Dans tous ces projets, les compétences de gestion sont nécessaires pour suivre le projet, donner le tempo, fixer et suivre les objectifs communs, les compétences de pilotage concernent la compréhension des attentes et des exigences, la gestion des risques, la maîtrise des points de décision et d'évolution (compétences DR1, DR3, RD3, CC1, CC3, EE1, EE2, EE3). Il faut savoir proposer, avancer, se remettre en cause, négocier. Le savoir-être (compétences CC1, CC2, CC3) est un élément clé pour éviter les mésententes et les désaccords, limiter les "égo", trouver les consensus.

D) Encadrement individuel et collectif

Comme indiqué à la page 247, la majorité de mes encadrements sont collectifs par conviction. Non seulement la pluralité apporte une meilleure couverture des connaissances et des compétences (union) mais elle permet une réduction des risques en nivelant les compétences *e.g.* un caractère fort ou une compétence élevée sont un biais dans la décision collective (cohésion).

Mon expérience d'encadrement individuel est principalement associée à l'encadrement de projets et stages de recherche ou dans les thèses dans lesquelles j'étais encadrant principal. J'ai toujours favorisé la mise en situation d'autonomie et responsabilité des étudiants au détriment parfois des résultats. L'objectif était de faire monter les participants en compétence[24]. Outre la remise en cause en cas de retard, échec, blocage, je me suis appliqué à avancer sur les compétences E1, E2, E3, E4 par exemple en changeant la stratégie sur le suivi, entre réunions fréquentes et espacées, planifiées ou spontanées, en direct au tableau ou à base de présentation, avec des attentes précises sur des actions très cadrées (le plus tard possible) ou au contraire suggestives et non coercitives, etc. L'utilisation systématique de schémas pour les discussions facilite la communication mais aussi la mémorisation des idées (compétence Bloc 5. Parfois l'idée est de faire ensemble lorsqu'il est difficile de verbaliser les concepts, encadrer c'est accompagner, encadrer c'est enseigner ; on retrouve les idées développées page 231. Toutes ces stratégies ou tactiques sont liées à l'attitude, les motivations et attentes, le caractère des personnes encadrées. La *success story* arrive lorsque la transmission du tuteur au tutoré va au delà du savoir pour entrer dans le savoir-faire puis le savoir-être (*être sur la même longueur d'onde* après avoir règlé chacun la longueur d'onde).

Mon expérience d'encadrement collectif est liée aux projets et stages de recherche, de personnels contractuels sur des projets, de thèse. Ma vision de l'encadrement ne change pas

24. Hors référentiel officiel à l'époque, donc celles de mon référentiel informel de la page 229.

par rapport à l'encadrement individuel mais elle doit s'adapter à celle des co-endadrants. Il faut pour cela réappliquer encore et toujours la méthode PDCA (*Plan Do Check Act*) issue du management qui forment un cycle itératif qui se veut vertueux "Roue de Deming" pour l'amélioration continue, qu'on peut retrouver dans les compétences EE1, EE2, EE3 en considérant que pour EE1 l'important est aussi de montrer l'exemple (apprentissage par analogie) et le chemin. Par contre, le fonctionnement du collectif d'encadrement a une influence sur l'avancement et son ressenti. C'est à ce niveau que j'ai pu développer les compétences CC1, CC2, CC3. Les qualités individuelles sont à la fois le *Ying* et le *Yang* en fonction du dosage. Par exemple, une personne dynamique est à la fois moteur sur l'activité mais peut aussi inhiber les autres par son influence.

Tous les encadrements de stages de Master ont donné lieu à des publications, moins des deux tiers des projets TER ont aussi donné lieu à une publication ou élément de publication, le reste n'était pas exploitable. J'ai co-encadré deux thèses avec un financement institutionnel, une thèse Cifre, deux thèses en co-tutelle, une thèse en bourse indépendante et une thèse sur projet. Trois thèses sont achevées avec succès, une a été arrêtée par démission au bout d'un an et demi (manque de motivation et de résultats) et trois sont encore en cours. Les co-encadrement étaient composés deux ou trois personnes, et à chaque fois l'expérience était différente, même quand un co-encadrant était le même. Au final, l'expérience se fait avec dix personnes différentes, huit chercheur(se)s et deux encadrants entreprise. La disponibilité des doctorant(e)s était variable : une seule résidence (1 thèse), alternance de sites sur la semaine (4 thèses), alternance bi-annuelle en co-tutelle (1 thèse) et une thèse en cotutelle se fait entièrement à distance (visio). De même les co-encadrants sont sur le même site (2), sur des sites localement différents (5) ou éloignés (3). Les modalités varient, évidemment en fonction de la distance, sur le même site, les rencontres sont bien plus spontanées et ouvertes. La progression des outils de travail à distance est un vrai plus pour l'encadrement dispersé dans le temps et l'espace (git, cloud, drive, mattermost, slack, overleaf, trello, zoom, discord, teams, meet, ...) mais les obstacles arrivent quand tous ne veulent les utiliser. La principale qualité dans l'encadrement est la cohérence. Elle est mise à mal lorsque les encadrants donnent des indications différentes voire contraire, lorsque l'implication n'est pas à la hauteur des exigences, lorsque les choix communs ne sont pas respectés, lorsque le système de valeur est différent. Dans une partie des encadrements, certains encadrants restent en retrait, notamment en termes d'initiative (3 cas vécus). Les cas qui posent problème sont des visions différentes de la stratégie (2 cas vécus). Le plus dur à accepter est lorsque la bonne solution n'est pas choisie dans telle ou telle étape majeure. Les facteurs d'échec de l'encadrement mais aussi de thèse sont connus : démotivation ou absence, incompréhension, autoritarisme, entêtement. De là naissent les tensions. L'incompétence ne mène pas forcément à l'échec mais diminue drastiquement la qualité. Dans mon expérience, les Comités de Suivi de Thèse (CSI) ont un apport positif certain sur la relation étudiant/encadrant mais n'influent pas vraiment sur les relations de co-encadrement.

Agilité

L'agile tour [25] est présent à Nantes depuis le 14 octobre 2009, depuis j'y assiste presque tous les ans à quelques conférences et ateliers parmi les nombreux proposés chaque année. Depuis 2010, je fais intervenir des spécialistes d'agilité dans les enseignements en Miage. C'est une bonne source d'inspiration, non seulement pour conduire les projets informatique, la planification ou l'estimation de charge, mais aussi pour gérer les projets en général, animer les réunions, ritualiser les collaborations et encore mieux travailler dans l'esprit de l'entreprise libérée. En pratique, nous avons constaté en rétrospective du projet ONECAD intitulée "Un peu d'agilité dans un projet de recherche pluridisciplinaire" et exposée en séminaires en 2024, que nous avions été cohérents avec le manifeste agile [26]. En effet, nous avons itérant de manière incrémentale sur les activités par lots -3 versions en 2 ans- (*cf.* FIGURE 8.6), en révisant le périmètre fonctionnel, en plaçant les utilisateurs au centre de l'application notamment pour l'ergonomie et la recette, en séparant l'expertise de l'utilisation (2 applications), etc.

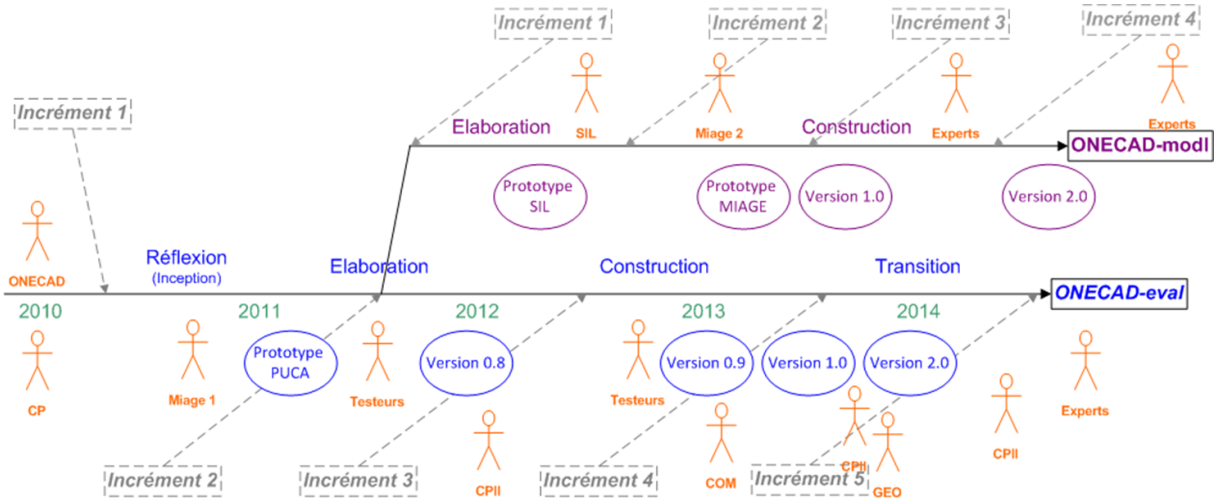


FIGURE 8.6 – Evolution du projet ONECAD - Agilité

De manière générale, contrairement à ce qu'on pourrait imaginer, l'agilité n'est pas facile à appliquer au monde de la recherche universitaire [27] de part la variété des activités des enseignants chercheurs, de la multiplicité des projets (enseignement ou recherche), de la variabilité des équipes, de la dispersion géographique des projets nationaux ou internationaux et tout simplement du côté novateur de la recherche avec les risques associés qui fait qu'on n'applique pas la même organisation à chaque projet. Même sur un seul projet de recherche, il est difficile de mobiliser les acteurs. De fait un projet pluridisciplinaire ou pluri-géographique est majoritairement un projet fractal et c'est l'agilité à l'échelle

25. `http://www.agiletour.org/`
26. Parmi les 12 principes du manifeste Agile on trouve : l'interaction avec les personnes plus que les processus et les outils, un produit opérationnel plus qu'une documentation pléthorique, la collaboration avec le client plus que la négociation de contrat, la réactivité face au changement plus que le suivi d'un plan.
27. Nous avions eu quelques réunions à ce sujet avec Agile Garden.

qu'il faut mettre en place avec un framework comme SAFe (*Scaled Agile Framework*) par exemple. D'autres écueils sont le manque d'adhésion et d'habitude dans cette démarche, le manque de culture projet au sens de la gestion de projet et finalement l'absence de rôles tels que le *Scrum Master*, le *Product Owner* dans un rôle reconnu qui sont des acteurs clés dans une organisation agile. Nous avons pu mettre en place des pratiques agiles dans les cours et formations (*ice breakers*, *roti*...) et les projets étudiants (kanban, rituels, poker planning...) mais on est très loin du compte en termes de potentiel. Il reste de la marge de manœuvre qui devra commencer par l'aculturation.

Réseaux

Des séminaires de laboratoire aux conférences internationales, il y a de nombreuses opportunités de rencontrer des interlocuteurs, des chercheurs dans les conférences ou les groupes thématiques (GDR, GT...), des industriels dans les groupes thématiques (ADL-Ouest, ON-AQL,...), dans les projets menés, dans les contacts des contacts, les réseaux sociaux (LinkedIn, X...). Je ne vais pas lister ici ces opportunités me concernant, le lecteur trouvera un échantillon dans ma page personnelle [28]. Dans ces occasions, il faut avoir le contact facile, et saisir les opportunités. Cela nécessite aussi de prendre le temps et de susciter les rencontres, se déplacer... On retrouve le problème de la disponibilité.

Pour les groupes, je fais partie de différents groupes de travail au niveau national dans le cadre du GDR de programmation puis du GDR ALP et maintenant le GDR GPL. Les groupes auxquels j'ai été affilié via mon équipe de recherche sont COSMAL (Composants Objets Services : Modèles, Architectures et Langages), RIMEL, MFDL (Méthodes Formelles dans le Développement Logiciel), IDM (Ingénierie Dirigée par les Modèles). Actuellement, nous intervenons sur GT-IDM et GT-GL_Sec du GdR GPL, GT-Verif du GdR IM et méthodes formelles pour la sécurité dans le GdR Sécurité via la thèse de M. Tébib. Je reconnais avoir une participation en dent de scie à ces groupes, selon les thématiques et le temps disponible et à ce sujet j'apprécie la visio-conférence pour les réunions ou journées car les déplacements sont parfois longs et coûteux.

Autre exemple, j'ai été impliqué dans deux axes de la filière Ouest Numérique de la région PDL, créée en 2011, l'axe Qualité Logicielle et l'axe Ingénierie des Modèles. Je suis particulièrement investi dans le premier (AQL) à travers la responsabilité de la plateforme collaborative, l'organisation de réunions, la responsabilité d'actions structurantes et du groupe formation. J'ai participé à l'élaboration de la feuille de route et des fiches actions, l'élaboration des fiches de synthèse, et la construction de l'enquête régionale qui a donné lieu au BaroQL, un indicateur de la qualité du logiciel dans la région [29].

Certaines collaborations sont issues de rencontres locales (projets CIM-Anjou, Ifremer, Onecad, Rodic), d'autres de conférences internationales (suivi de thèses, publications collaboratives, projet Econet).

28. `http://pagesperso.ls2n.fr/~andre-p/fr/links.html`
29. `http://www.a2jv.fr/resultats-du-baroql-sur-la-qualite-logicielle-en-pays-de-la-loire/`

Ces différents contextes m'ont permis de développer les compétences DR1, DR2 en plus de celles du chercheur.

Recherche de financement

Le réseau favorise les rencontres. L'étape suivante est de "transformer l'essai". Un certain nombre de soumissions de projets (collaborations, cifre, projets, contrats, transferts) n'ont pas abouti et d'autres n'ont pas été lancés par manque de temps ou d'énergie. Il y a une marge de progression sur ce sujet.

Un point à noter ici est que la vision des thématiques et financements est éminemment politique et fonctionne à une vitesse bien plus rapide que celle des progrès scientifique et techniques. On le constate avec les appels nationaux et internationnaux. Il y a aussi des modes et des *buzz words* qui doivent figurer dans les réponses pour être étudiées, c'est le cas des approches services il y a quelques années ou de l'intelligence artificielle ou l'informatique durable actuellement. Il y a un déphasage du à des échelles de temps peu compatibles.

Le logiciel est un domaine où la recherche privée a plus d'influence que la recherche institutionnelle. Les GAFAM et les gros éditeurs de solutions telles que Oracle, HP, IBM, SAP sont des acteurs clés dans le génie logiciel parce qu'il disposent et proposent des outils pour développer des applications logicielles. En termes de recherche appliquée, leur force de frappe rend la concurrence difficile sur certains sujets et impose les visions. La collaboration aussi est difficile du fait d'un cadencement très différent des activités, sujets et projets. La recherche institutionnelle est marquée par les appels à projets de l'Europe. Le programme pour une Europe numérique (DIGITAL) [30] est un nouveau programme de financement de l'Union axé sur l'intégration de la technologie numérique aux entreprises, aux citoyens et aux administrations publiques. Il vise à fournir un financement stratégique pour relever les défis d'une Europe plus verte et plus numérique, en soutenant des projets dans cinq domaines clés de capacité : (i) super-calcul, (ii) intelligence artificielle, (iii) cybersécurité, (iv) compétences numériques avancées et (v) garantir une large utilisation des technologies numériques dans l'ensemble de l'économie et de la société, y compris par l'intermédiaire de pôles d'innovation numérique. D'une certaine manière, la recherche privée et publique imposent les thématiques dans un rythme plus rapide que celui des enseignants-chercheurs. Par exemple, on est passé de l'approche à objets aux composants puis aux services et micro-services sans garantir les avantages de l'une par rapport à l'autre. Cele peut engendrer une certaine fuite en avant dans les activités du checheur ou du directeur de recherche. Il est alors intéressant de trouver des sujets de niches ou des contributions très innovantes qui pourront aider à développer de nouvelles collaborations.

30. https://www.horizon-europe.gouv.fr/programmes-europeens-de-financement-de-projets-hors-horizon-europe-29927

Direction de recherche

Diriger des recherches c'est rendre cohérent un ensemble de recherches et de projets. La direction de recherche est souvent liée à la structuration dans une organisation avec deux processus aggréger (centraliser) et répartir (décentraliser) et relève de la stratégie de recherche. Une approche idéale pour le faire est de prendre un thème qu'on décompose en sous-thèmes, chaque sous-thème constitue un projet de collaboration (financé ou pas). La cohérence est obtenue lorsque ces sous-projets sont complémentaires pour constituer une contribution d'envergure. Ce peut être une vision d'équipe de recherche qui réalise fidèlement la compétence DR1. Une autre approche de recherche, de type expertise, consiste à développer une solution outillée pour résoudre un problème générique et ensuite décliner cette solution dans différents contextes ou sur différents problèmes. L'expertise est un facteur-clé pour la participation dans des projets collaborations.

Ma stratégie de recherche personnelle est plus liée aux rencontres et opportunités. La collaboration démarre sur de petites actions qui prennent du volume vers des projets. Les personnes les plus difficiles à convaincre sont celles des entreprises pour lesquels les retours sur investissement doivent être rapides ; les projets partent d'un besoin, qui doit être précis. Si les travaux de la partie II restent assez cohérents sur mon domaine de recherche, le génie logiciel, ceux de la partie III sont plus dispersés. Nous reverrons ce point dans le chapitre 8.4.

J'ai une vision plutôt collégiale de la direction de recherche, applicable quel que soit le domaine (science, humanités, entreprise, associatif, politique...). Les principes de fonctionnement collectif doivent être simples, pédagogiques et compréhensibles par tous. Par exemple, "la même règle s'applique à tous", "tout le monde paie la taxe et la taxe sert à tous", "travailler plus pour gagner plus" sont faciles à comprendre. Si leur application est honnête et juste, le collectif fonctionne bien.

8.4 Conclusion

Dans ce chapitre, l'objectif était non pas de fournir un référentiel d'évaluation des chercheurs HDR mais de réfléchir au métier en termes de compétences pratiques et savoir s'il est adapté à soi.

Pour mener à bien ses activités un directeur de recherche doit développer un profil complet du point de vue scientifique, organisationnel et humain avec des compétences de savoirs, savoirs-faire, savoir-être. Il doit de plus être commercial et gestionnaire pour financer ses activités de recherche ; il doit mettre en place un réseau de partenaires chercheurs et industriels pour cela. À mon avis, seule une organisation en équipe(s) permet d'atteindre globalement ces objectifs, surtout lorsqu'on est chercheur à mi-temps.

Un autre aspect à prendre en compte est l'exploitabilité des résultats de recherche, et on peut constater que l'immense majorité des travaux de recherche et des publications

a un impact mineur [31] sur les progrès scientifiques et techniques, d'où le sous-titre de ce manuscrit, *Rester modeste et avancer ensemble*.

Post-scriptum

C'est un processus sans fin, qu'on appelle de manière moderne l'amélioration continue. De fait, la compétence ici est de pouvoir prendre du recul pour analyser le passer, se remettre en cause et proposer des améliorations pour le futur.

> *Ce sont vos choix, M. Potter, qui déterminent qui vous êtes, bien plus que vos compétences.*
>
> Harry Potter à l'école des sorciers
> J. K. Rowling

31. Ce n'est pas qu'un manque d'ambition, c'est lié fortement à l'évaluation de la recherche et la politique de publication.

PERSPECTIVES

Les travaux mis en avant dans ce manuscrit s'inscrivent dans une problématique assez large de génie logiciel, avec des enjeux multiples (déclinaison des coûts de développement/maintenance). Ces travaux présentés sont bien entamés mais loin d'être achevés. Toutes les perspectives ne peuvent être suivies, par manque de temps, de moyens ou de priorité. J'ai organisé la suite de ces travaux en trois axes de recherche. Le premier est un axe horizontal autour de la composition de langages (de modélisation) pour couvrir différents aspect complémentaires. Le deuxième est un axe vertical au cœur de la conception logicielle qui vise à construire une application spécifique à partir d'un modèle et d'un framework générique. Le troisième axe se focalise sur l'architecture d'entreprise dans laquelle les modèles sont perçus de différents points de vue qu'il faut pouvoir satisfaire (pas combiner contrairement à l'axe 1). La FIGURE 8.7 représente ces axes et les domaines associés. Les axes 1 & 2 contribuent à l'axe 3. Les travaux sur la sécurité et les projets pluridisciplinaires continuent, mais ne constituent pas un axe majeur.

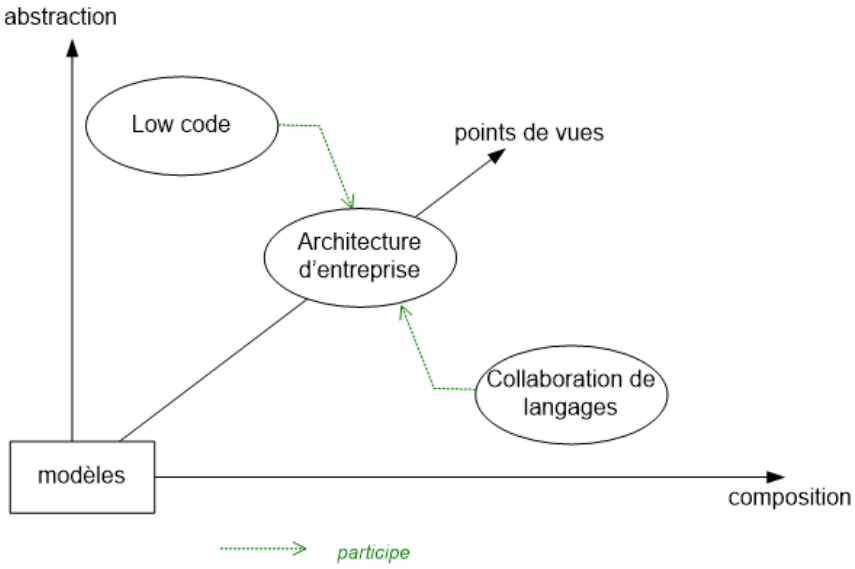


FIGURE 8.7 – Perspectives de recherche

La vision générale reste la même et le processus est itératif et centré sur les modèles : (i) on définit et alimente les modèles (modélisation ou rétro-ingénierie), (ii) on définit et vérifie des propriétés sur les modèles (vérification, validation), (iii) on exploite les modèles selon des cas d'usage dédiés (transformation). L'objectif reste aussi de proposer des approches applicables en pratique. D'un point de vue opérationnel, le cap à franchir est de découper en projets sponsorisés.

Axe 1 - Collaboration de langages

Des langages comme UML ou Kmelia sont large spectre et couvrent les axes de description structurels, dynamiques et fonctionnels d'un système (*cf.* FIGURE 1.5). L'avantage est, comme pour la programmation, de disposer d'un seul langage et un compilateur suffit. Les inconvénients majeurs sont d'une part de donner une sémantique cohérente à l'ensemble des concepts du langage avec des techniques de vérification associées (cela se fait souvent par projection) et d'autre part de maintenir et faire évoluer ces langages et leur environnement. A l'inverse, travailler avec des langages dédiés ou DSL est plus simple car chacun se focalise sur un sous-ensemble cohérent de concepts (aspect, un point de vue, axe...). Il est plus facile de vérifier les modèles et de les exploiter par transformation. La difficulté est de combiner ces langages pour obtenir un modèle global du système. Ce n'est pas nouveau, nous avions cette problématique appelée (i) intégration de paradigmes dans l'équipe GL-MSF puis SHES de l'IRIN dans les années 2000 [Sal03] puis (ii) intégration de méthodes de spécification et développement dans l'équipe dans l'équipe COLOSS du LINA. Le lecteur trouvera dans [Att07], un exposé argumenté du sujet. Plusieurs approches y sont envisagées (1) combinaison de paradigmes *e.g.* LOTOS, Raise ; (2) extension d'un langage par d'autres aspects *e.g.* Timed-CSP, Z+CSP ; (3) intégration forte et homogène, souvent dans un langage plus abstrait *e.g.* Event Calculus. Trois aspects principaux sont considérés pour la stratégie d'intégration multiparadigme (contrôle, données+comportement, temps) avec plongement dans des domaines sémantiques.

Verrou 1

Le défi scientifique est de définir des langages de modélisation expressifs, combinables et évolutifs dont les modèles puissent être vérifiés ou exécutés.

D'un point de vue abstrait (concepts, modèle, propriétés), intégrer les langages est difficile, à l'instar des langages formels [Att07]. La composition des langages ne doit pas donner un nouveau langage car il serait bien trop complexe à définir et tous les aspects n'intéressent pas tout le monde. Inversement, un langage pivot peut se projeter sur tel point de vue pour la modélisation ou la vérification mais la cohérence doit être assurée et le langage doit être universel. Kmelia permet la vérification par projection mais ne couvre pas tous les aspects (*e.g.* le temps ou les calculs restent très limités). UML distingue la représentation interne des concepts et les points de vues en diagrammes. Le méta-modèle UML [Gro11b ; Gro11c] définit des centaines de concepts, qui même groupés par thème, sont difficiles à appréhender et à vérifier. La principale critique d'UML est son manque de sémantique formelle. On a besoin de langages expressifs mais cohérents et vérifiables.

D'un point de vue concret (langages, compilateurs, vérification), on souhaite réutiliser au maximum les langages existants et les outils associés, notamment pour la vérification, car développer un langage et les outils associés est chronophage. Dans Kmelia, l'extraction d'un point de vue produit une spécification dans un langage dédié dont on utilise les outils

pour la vérification. L'approche, efficiente, permet de prouver des propriétés spécifiques mais pas de garantir totalement la cohérence de l'ensemble. Par contre le langage n'est pas assez expressif pour l'algorithmique usuelle. L'idée est de réutiliser à la fois les langages et écosystèmes associés (compilateurs, vérification, transformation) par un système d'API, dit autrement de concevoir **modulairement** les langages comme on conçoit les modules logiciels avec des interfaces et des contrats. La composition de modèles a déjà été étudiée par exemple dans la thèse de M. Clavreul [Cla11] mais à ma connaissance c'est une thématique émergente dans les DSL [32].

L'étude et l'application de cette perspective d'ingénierie des langages (*cf.* page 36) est incluse dans les travaux engagés dans la thèse de Yasmina Dali Youcef du projet RODIC/WP3 (*cf.* page 221) mais aussi dans le projet IRGA/Castav avec l'université de Grenoble. Dans le premier contexte, plusieurs points de vue d'un atelier de fabrication doivent être cohérents (physique, contrôle, KPI...). Dans le second, on définit des langages pour les applications mobiles et d'autres pour les contextes d'exécution, l'ensemble permet de vérifier des propriétés de sécurités tenant compte de l'usage. Le problème de manipulation de langages divers est implicite dans l'alignement BITA entre les couches des architectures d'entreprise. Par contre l'objectif n'est pas de composer mais de tisser des liens pour cartographier et aligner l'ensemble des points de vue.

Axe 2 - Développement *Low-code*

L'objectif du developpement *Low-code* est de permettre à des non-specialistes de génie logiciel de développer et maintenir certains types d'applications [DR+22]. L'approche, basée sur des *Low-Code Development Platforms (LCDPs)* [BF21], est prometteuse en termes de marché selon le cabinet Gartner et interprétée par les praticiens comme la réduction de l'effort de codage [Luo+21]. En pratique, les logiciels produits accumulent souvent de gros volumes de code complexe, plus difficile à maintenir que dans le développement traditionnels, du fait de la non prise en compte des bonnes pratiques d'ingénierie logicielle [Let21].

Verrou 2

Le défi scientifique et technique est de proposer des solutions d'assistance au développement, qui réduisent l'effort de développement et de maintenance, sans nuire à la qualité.

Mon objectif n'est pas de fournir une plateforme mais une démarche MDD générique qui se décline sur des environnements cibles. Mon point de vue sur le *Low-code* est similaire à celui de Di Ruscio et al. : *"Low-code development and model-driven engineering : Two sides of the same coin ?"* [DR+22]. Cet axe s'inscrit donc dans les perspectives ouvertes par les travaux du chapitre 4. Le principe de base est celui de l'abstraction des plateformes techniques conjuguées avec le raffinement des modèles logiques. Nous n'imaginons pas

32. Différent de *megamodel* qui est un modèle de modèles et de relations entre eux [HSG12].

à ce stade partir des modèles d'analyse, complètement indépendants de toute mise en œuvre, mais de la conception générale. Deux voies, relativement indépendantes, sont donc à explorer : (i) abstraire progressivement les bibliothèques représentant les *frameworks* d'implantation en *Platform Description Model (PDM)* et (ii) raffiner la conception en structurant le processus selon l'impact des décisions de conception et en respectant des principes de bonne conception. Détaillons.

L'abstraction de PDM, par rétro-ingénierie, est utile non seulement pour construire les PDM pour le raffinement mais aussi, et tout simplement, pour mieux documenter l'architecture du framework, le code à réutiliser et les API associées, mieux étudier l'intégration des parties de code personnalisées à insérer, etc. Par exemple, de nombreuse API fournissent simplement des signatures d'opération, alors qu'un protocole d'usage sous forme d'automate donne une idée de l'ordre dans lequel appeler ces opérations. Enfin, un modèle de documentation détaillé permet d'interroger le PDM pour trouver un fournisseur de services à partir de requêtes détaillée. On retrouve l'idée de contrat de service multi-niveaux de Kmelia (*cf.* Section 3.4 du Chapitre 3) ; l'interface s'interroge non seulement sur la signature mais toutes les facettes du contrat de service [MAA10b ; BJP10]. Abstraire du code est aussi l'objet du projet `ComparCode`. Ce travail, mené avec des étudiants, vise à comparer des applications au niveau du code et des modèles pour détecter des similarités. Les applications sont multiples : détection de plagiat, vérification de licences, recherche de patterns, factorisation de code, etc.

Le processus de raffinement a été tracé dans la Section 4.3 du Chapitre 4. La première phase est d'en systématiser les étapes, ce qui est une contribution pour l'enseignement du développement aux étudiants. La seconde phase est d'automatiser les parties systématiques par des transformations de modèles. Nous l'avons fait de manière expérimentale et échantillonnée mais la couverture est loin du compte. Sans surprise, plus on remonte en abstraction, plus l'automatisation est délicate. La notion d'architecture logicielle (*cf.* Chapitre 3) conserve son importance pour fixer les contraintes globales et faire en sorte que le raffinement d'un module n'introduise pas des incohérences sur les autres modules. Une analogie naïve pour les frameworks est celle des circuits intégrés qui définissent le fonctionnement de base sur lequel on peut enficher des composants techniques et fonctionnels. Un résultat attendu est aussi de répondre à une question ouverte dans le chapitre 1 : comment trouver quel service (offert) peut satisfaire une service requis ? L'hypothèse sous-jacente est de savoir définir les services requis par un contrat de service assez large (un appel d'offre), auquel plusieurs fournisseurs pourront répondre. Le processus est alors le suivant : (1) définir le contrat de service requis, (2) chercher les services candidats dans le PDM, et (3) adapter le contrat de service pour satisfaire client et fournisseur.

D'un point de vue concret et à court terme, le framework lejos est ciblé car est à la fois limité (écrit en Java uniquement) et complet (c'est un système d'exploitation). D'autres situations semblent intéressantes comme le raffinement de processus métiers en micro-service dans le cadre de l'alignement BITA de systèmes d'information.

Axe 3 - Architecture d'entreprise

Dans le monde des Systèmes d'Information, l'architecture d'entreprise rassemble les disciplines qui raisonnent sur l'entreprise et son système d'information pour en améliorer la qualité (*cf.* Section 1.2.8 du Chapitre 1). En pratique, elle est l'apanage des architectes, et plus l'organisation est grande plus ils sont nombreux et spécialisés avec une orientation soit technique soit fonctionnelle [33] : architectes d'entreprise, architectes des systèmes d'information, architectes fonctionnels, architectes applicatifs, architectes techniques, architectes des systèmes informatiques, urbaniste, etc. Ces architectes travaillent sur des modèles des systèmes réels, qui en sont une interprétation. Avoir une vision plus abstraite permet d'éviter de s'engluer dans la complexité des systèmes existants mais à l'inverse on peut fausser le raisonnement si le modèle d'architecture est déconnecté de la réalité. C'est souvent le cas. Cartographier l'existant pour avoir une vue à jour est chronophage. Remonter les évolutions techniques ou fonctionnelle en continu implique un engagement des équipes du terrain, qui n'en n'ont pas un intérêt direct. Il en résulte que les modèles traités sont hors sols au bout d'un certain temps, leur suivi par les architectes est manuel. Un des besoins cruciaux mentionnés par les architectes applicatifs est d'alimenter les modèles architecturaux par les données du terrain. Il faut donc créer des outils qui puissent se connecter au système d'information mais surtout il faut en faire bien plus en termes d'abstraction pour masquer les détails. C'est ce dernier point qui pose le plus de problèmes comme nous l'avons montré dans le chapitre 6. Le second besoin pour l'alignement BITA est de rendre compatible la vision flux des processus métiers avec la vision urbanisation des applications sur un point de vue fonctionnel, sécurité, données, géographique...

Verrou 3

Le défi scientifique et technique de l'urbanisation est double : premièrement remonter les abstractions issues des infrastructures et des applications déployées à un niveau architectural et deuxièmement décliner des points de vue sur l'alignement en fonction de responsabilités (fonctionnel, RSSI, Data, RGPD...).

L'objectif est d'améliorer les techniques de rétro-ingénierie pour déterminer les abstractions nécessaires au niveau de la couche applicative. On doit passer d'une portée lexicale de type paquetages et dépendances, qui produit un modèle statique, à une structuration plus comportementale en isolant les collaborations dans des composants. L'interface de ces composants, à l'instar de Kmelia doit préciser les services requis en plus des services offerts. Sur ce point, nous pouvons enrichir les techniques proposées dans le projet ECONET (*cf.* Section 3.6 du Chapitre 3) en les adaptant au contexte. Le contexte inclut les pratiques appliquées au système d'information ciblé. C'est difficile car une solution logicielle est le reflet de sa construction (loi de Conway - page 54), si la construction est systématique, sa

33. Fiche ROME M1802 et M1806 notamment pour les métiers ou RNCP31471 pour les compétences de la Miage.

déconstruction par rétro-ingénierie peut alors l'être. Inversement plus il y a d'exceptions, plus il est difficile d'identifier des patterns. Cette abstraction comportementale vise le point de vue fonctionnel. On doit procéder de même avec les autres points de vue choisis *e.g.* données, RGPD... Un dernier challenge sera alors de croiser les points de vue pour qualifier le niveau d'alignement (on retrouve la composition non-intrusive des modèles) et comparer la situation actuelle avec les scénarios futurs (on retrouve la comparaison de modèles). Concernant l'outillage on retrouve des similarités avec l'axe *Low-code* de la section 8.4.

Pour la mise en pratique nous avons eu plusieurs réunions avec des architectes de collectivités locales et d'entreprises, le besoin est là mais la conversion en projets de recherche n'est pas simple. Les démarches et échanges continuent. La piste la plus prometteuse est celle d'une collaboration avec un éditeur d'ERP pour qui on a des modèles applicatifs de l'ERP à croiser avec des modèles métiers des clients. Le cas est très intéressant car le périmètre n'est aussi large que celui des autres interlocuteurs (DSI de collectivités).

Montée en compétences

Dans le bilan auto-évalué de compétences (*cf.* Section 8.3 du Chapitre 8), les compétences de chercheur et d'encadrement m'ont paru couvertes par des expériences passées. Par contre, j'ai mentionné comme points d'amélioration, les compétences relatives à la structuration de la recherche en projets (`DR1`), le pilotage de groupe au delà de l'encadrement (`DR3`), la valorisation des travaux et le transfert de compétences (`Bloc 3`) et la visibilité dans les communautés (`DR2`). Les projets ANR (axe 1) et les collaborations entreprise (axe 3) sont l'occasion de tester différentes idées même si la marge de manœuvre reste mince à ce stade. Sur les travaux de l'axe 2, les échanges passés n'ont pas permis de convaincre mes interlocuteurs d'aller plus en avant sur le sujet. Les pistes sont nombreuses mais il faut investir plus de temps pour les concrétiser.

Concernant la diffusion des savoirs (`Bloc 5`), les travaux permettent d'actualiser les cours de génie logiciel dispensés (un peu par moi mais aussi par beaucoup d'autres) en Miage, je les mets à profit dans le parcours ISI (Ingénierie des Systèmes d'Information) devenu DSN (Développement des Systèmes Numériques) dans la dernière habilitation. Ces enseignements étant à vocation applicative, l'intervention de professionnels est systématique dans chque module de génie logiciel à la Miage de Nantes. Notons que les cas d'étude du chapitre 4 servent aux projets des étudiants de Master Miage. Ceux du chapitre 6 ne le sont pas par souci de confidentialité, mais de nouveaux cas pourraient l'être. La diffusion vers la société se fait à travers ma participation à des groupes de travail, mais là encore il est important de libérer du temps pour ces actions.

BIBLIOGRAPHIE

[AA05] Pascal ANDRÉ et Gilles ARDOUREL, « Domain Based Verification for UML Models », in : *Workshop on Consistency in Model Driven Engineering C@Mode'05*, sous la dir. de Ludwik KUZNIARZ et al., nov. 2005, p. 47-62, URL : http://www.ipd.bth.se/consistencyUML/CoMoDE-2005/.

[AAA06a] Pascal ANDRÉ, Gilles ARDOUREL et J. Christian ATTIOGBÉ, « Adaptation for Hierarchical Components and Services », in : *Proceedings of the Third International Workshop on Coordination and Adaption Techniques for Software Entities, WCAT@ECOOP 2006, Nantes, France, July 4, 2006*, sous la dir. de Steffen BECKER et al., t. 189, ENTCS, Elsevier, 2006, p. 5-20.

[AAA06b] Pascal ANDRÉ, Gilles ARDOUREL et J. Christian ATTIOGBÉ, « Spécification d'architectures en Kmelia : hiérarchie de connexion et composition », in : *1er Conférence francophone sur les Architectures Logicielles (CAL 2006), 4-6 September 2006, Nantes, France*, sous la dir. de Mourad Chabane OUSSALAH et al., Hermes Science, 2006, p. 101-118.

[AAA06c] Pascal ANDRÉ, Gilles ARDOUREL et Christian ATTIOGBÉ, « Vérification d'assemblage de composants logiciels Expérimentations avec MEC », in : *6e conférence francophone de MOdélisation et SIMulation, MOSIM 2006*, Rabat, Maroc : Lavoisier, avr. 2006, p. 497-506, ISBN : 2-7430-0892-X.

[AAA06d] J. Christian ATTIOGBÉ, Pascal ANDRÉ et Gilles ARDOUREL, « Checking Component Composability », in : *Software Composition - 5th International Symposium, SC@ETAPS 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*, sous la dir. de Welf LÖWE et Mario SÜDHOLT, t. 4089, LNCS, Springer, 2006, p. 18-33.

[AAA07a] Pascal ANDRÉ, Gilles ARDOUREL et J. Christian ATTIOGBÉ, « Defining Component Protocols with Service Composition : Illustration with the Kmelia Model », in : *Software Composition - 6th International Symposium, SC@ETAPS 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, sous la dir. de Markus LUMPE et Wim VANDERPERREN, t. 4829, LNCD, Springer, 2007, p. 2-17, DOI : 10.1007/978-3-540-77351-1_2.

[AAA07b] Pascal ANDRÉ, Gilles ARDOUREL et J. Christian ATTIOGBÉ, « Protocoles d'utilisation de composants : spécification et analyse en Kmelia », in : *Actes des journées Langages et Modèles à Objets, LMO'07. Toulouse,*

France, 27-29 mars, sous la dir. d'Isabelle Borne et al., Hermès Lavoisier, 2007, p. 19-34.

[AAA07c] Pascal André, Gilles Ardourel et Christian Attiogbé, « A Formal Analysis Toolbox for the Kmelia Component Model », in : *ProVeCS 2007 - Satellite Event of TOOLS Europe*, Zurich, Switzerland : ETH Research Report –, 2007, to appear, url : http://lina.atlanstic.net/prov/.

[AAA07d] Pascal André, Gilles Ardourel et Christian Attiogbé, « Protocoles d'utilisation de composants, Spécification et analyse en Kmelia », in : *13e Conférence Francophone sur les Langages et Modèles à Objets*, Hermès Sciences Publications - Lavoisier, 2007, p. 19-34, isbn : 978-2-7462-1806-2.

[AAA08] Pascal André, Gilles Ardourel et J. Christian Attiogbé, « Composing Components with Shared Services in the KmeliaModel », in : *Software Composition - 7th International Symposium, SC@ETAPS 2008, Budapest, Hungary, March 29-30, 2008. Proceedings*, sous la dir. de Cesare Pautasso et Éric Tanter, t. 4954, LNCS, Springer, 2008, p. 125-140, doi : 10.1007/978-3-540-78789-1_9.

[AAA11] Pascal André, Gilles Ardourel et J. Christian Attiogbé, « Kmelia, un modèle abstrait et formel pour la description et la composition de composants et de services », in : *Tech. Sci. Informatiques* 30.*6* (2011), p. 627-658, doi : 10.3166/tsi.30.627-658.

[AAC19] Pascal André, Fawzi Azzi et Olivier Cardin, « Heterogeneous Communication Middleware for Digital Twin Based Cyber Manufacturing Systems », in : *Proceedings of the 9th Workshop on Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future, SOHOMA 2019, Valencia, Spain, October 3-4, 2019*, sous la dir. de Theodor Borangiu et al., t. 853, Studies in Computational Intelligence, Springer, 2019, p. 146-157.

[AAL18a] Pascal André, J. Christian Attiogbé et Arnaud Lanoix, « Modelling and Analysing the Landing Gear System : a Solution with Event-B/Rodin », in : *CoRR* abs/1803.05647 (2018), arXiv : 1803.05647, url : http://arxiv.org/abs/1803.05647.

[AAL18b] Pascal André, J. Christian Attiogbé et Arnaud Lanoix, « Systematic Construction of Critical Embedded Systems Using Event-B », in : *New Trends in Model and Data Engineering - MEDI 2018 International Workshops, DETECT, MEDI4SG, IWCFS, REMEDY, Marrakesh, Morocco, October 24-26, 2018, Proceedings*, sous la dir. d'El Hassan Abdelwahed et al., t. 929, Communications in Computer and Information Science, Springer, 2018, p. 200-216, doi : 10.1007/978-3-030-02852-7_18.

[AAL20] Pascal André, J. Christian Attiogbé et Arnaud Lanoix, « A tool-assisted method for the systematic construction of critical embedded systems using Event-B », in : *Comput. Sci. Inf. Syst.* 17.*1* (2020), p. 315-338, doi : 10.2298/CSIS190501042A.

[AAM09] Pascal André, J. Christian Attiogbé et Mohamed Messabihi, « Correction d'assemblages de composants impliquant des interfaces paramétrées », in : *3e Conférence francophone sur les Architectures Logicielles, CAL 2009, Nancy, France, 24-25 Mars, 2009*, sous la dir. d'Olivier Zendra et Antoine Beugnard, t. L-4, Revue des Nouvelles Technologies de l'Information, Cépaduès-Éditions, 2009, p. 33-46, url : http://editions-rnti.fr/?inprocid=1000835.

[AAM10] Pascal André, Gilles Ardourel et Mohamed Messabihi, « Component Service Promotion : Contracts, Mechanisms and Safety », in : *Formal Aspects of Component Software - 7th International Workshop, FACS 2010, Guimarães, Portugal, October 14-16, 2010, Revised Selected Papers*, sous la dir. de Luís Soares Barbosa et Markus Lumpe, t. 6921, Lecture Notes in Computer Science, Springer, 2010, p. 145-162, doi : 10.1007/978-3-642-27269-1_9.

[AAM11] Pascal Andre, Gilles Ardourel et Mohamed Messabihi, « Vérification de contrats logiciels à l'aide de transformations de modèles Application à Kmelia », in : *7ièmes Journées sur l'Ingénierie Dirigée par les Modèles*, sous la dir. d'Ileana Ober, Actes des 7èmes Journées sur l'Ingénierie Dirigée par les Modèles, Ileana Ober, Toulouse, France, juin 2011, url : https://hal.science/hal-01147192.

[AAM13] Pascal Andre, Gilles Ardourel et Jean-Marie Mottu, « Assistance au test de modèles à composants et services », in : *2ème Conférence en IngénieriE du Logiciel*, ISBN 978-2-905267-89-4, Nancy, France, avr. 2013, p. 13-28, url : https://hal.science/hal-00823319.

[AAM17] Pascal André, J. Christian Attiogbé et Jean-Marie Mottu, « Combining Techniques to Verify Service-based Components », in : *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017*, sous la dir. de Luís Ferreira Pires, Slimane Hammoudi et Bran Selic, SciTePress, 2017, p. 645-656, doi : 10.5220/0006212106450656.

[AAS04] Pascal André, Gilles Ardourel et Gerson Sunyé, « The Bosco Project - A JMI-Compliant Template-based Code Generator », in : *Proceedings of the ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering, Nice, France, July 1-3, 2004*, ISCA, 2004, p. 157-162.

[Abr84] Jean-Raymond Abrial, « Spécifier ou comment matérialiser l'abstrait », in : *Technique et Science Informatique* 3.*3* (1984), p. 201-219.

[ABR95] Pascal André, Frank Barbier et Jean-Claude Royer, « Une expérimentation de développement formel à objets », in : *Techniques et Sciences Informatique* 14.*8* (1995), p. 973-1005.

[Abr96] Jean-Raymond Abrial, *The B-Book Assigning Programs to Meanings*, ISBN 0-521-49619-5, Cambridge University Press, 1996, isbn : 0-521-49619-5.

[AC17] Pascal André et Olivier Cardin, « Trusted Services for Cyber Manufacturing Systems », in : *Service Orientation in Holonic and Multi-Agent Manufacturing - Proceedings of SOHOMA 2017, Nantes, France, October 19-20, 2017*, sous la dir. de Theodor Borangiu et al., t. 762, Studies in Computational Intelligence, Springer, 2017, p. 359-370, doi : 10.1007/978-3-319-73751-5_27.

[AC22] Pascal André et Olivier Cardin, « Aggregation Patterns in Holonic Manufacturing Systems », in : *Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future*, sous la dir. de Theodor Borangiu et al., Cham : Springer International Publishing, 2022, p. 3-15, isbn : 978-3-030-99108-1.

[AC24] Pascal André et Olivier Cardin, « A Core Reference Model for Applicable Reconfigurable Manufacturing Systems », in : *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*, sous la dir. de Theodor Borangiu et al., Cham : Springer Nature Switzerland, 2024, p. 507-519, isbn : 978-3-031-53445-4.

[ACA20] Pascal André, Olivier Cardin et Fawzi Azzi, « Multi-protocol Communication Tool for Virtualized Cyber Manufacturing Systems », in : *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future - Proceedings of SOHOMA 2020, Paris, France, 1-2 October 2020*, sous la dir. de Theodor Borangiu et al., t. 952, Studies in Computational Intelligence, Springer, 2020, p. 385-397.

[AG24] Pascal André et Virginie Goepp, « A Framework for Defining Customised KPI in Manufacturing Systems », in : *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*, sous la dir. de Theodor Borangiu et al., Cham : Springer Nature Switzerland, 2024, p. 309-320, isbn : 978-3-031-53445-4.

[AG97] Robert Allen et David Garlan, « A Formal Basis for Architectural Connection », in : *ACM Transactions on Software Engineering and Methodology* 6.*3* (juill. 1997), p. 213-249.

[AGT10] Lerina Aversano, Carmine Grasso et Maria Tortorella, « Measuring the Alignment Between Business Processes and Software Systems : A Case Study », in : *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, New York, NY, USA : ACM, 2010, p. 2330-2336, isbn : 978-1-60558-639-7.

[AGT13] L. Aversano, C. Grasso et M. Tortorella, « A Literature Review of Business/IT Alignment Strategies », en, in : *Enterprise Information Systems*, sous la dir. de José Cordeiro, Leszek A. Maciaszek et Joaquim Filipe, LNBIP 141, Springer, jan. 2013, p. 471-488, isbn : 978-3-642-40653-9, 978-3-642-40654-6, (visité le 07/02/2014).

[AGT16] Lerina Aversano, Carmine Grasso et Maria Tortorella, « Managing the alignment between business processes and software systems », in : *Information and Software Technology* 72 (2016), p. 171 -188, issn : 0950-5849, doi : https://doi.org/10.1016/j.infsof.2015.12.009.

[AH06] Pascal André et Henri Habrias, « Application d'ontologies formelles au droit », in : *SDC'06 Workshop on Ontologies et textes juridiques, OTJ'06*, Nantes, France, 2006, p. 30-39.

[AKS16] Nida Afreen, Asma Khatoon et Mohd. Sadiq, « A Taxonomy of Software's Non-functional Requirements », in : *Proceedings of the Second International Conference on Computer and Communication Technologies*, sous la dir. de Suresh Chandra Satapathy et al., New Delhi : Springer India, 2016, p. 47-53, isbn : 978-81-322-2517-1.

[AL03] P. C. Attie et D. H. Lorenz, *Establishing Behavioral Compatibility of Software Components without State Explosion*, rapp. tech. NU-CCIS-03-02, College of Computer et Information Science, Northeastern University, 2003.

[ALB19] Pascal Andre et Yannis Le Bars, « Conception assistée de contrôleurs d'automates depuis des modèles UML », in : *MSR 2019 - 12ème Colloque sur la Modélisation des Systèmes Réactifs, Nov 2019, Angers, France*, Angers, France, nov. 2019, url : https://hal.archives-ouvertes.fr/hal-02431942.

[AMA13] Pascal André, Jean-Marie Mottu et Gilles Ardourel, « Building Test Harness From Service-based Component Models », in : *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2013, co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, October 1st, 2013*, sous la dir. de Frédéric Boulanger, Michalis Famelis et Daniel Ratiu, t. 1069, CEUR Work-

shop Proceedings, CEUR-WS.org, 2013, p. 11-20, URL : https://ceur-ws.org/Vol-1069/04-paper.pdf.

[AMS16] Pascal ANDRÉ, Jean-Marie MOTTU et Gerson SUNYÉ, « COSTOTest : a tool for building and running test harness for service-based component models (demo) », in : *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, sous la dir. d'Andreas ZELLER et Abhik ROYCHOUDHURY, ACM, 2016, p. 437-440, DOI : 10.1145/2931037.2948704.

[And+00] Pascal ANDRÉ et al., « An algebraic view of UML class diagrams », in : *Actes des journées Langages et Modèles à Objets, LMO'2000. Mont Saint-Hilaire, Québec, Canada, 25-28 janvier*, sous la dir. de Christophe DONY et Houari A. SAHRAOUI, Hermès, 2000, p. 261-276.

[And+09] Pascal ANDRÉ et al., « Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies », in : *Proceedings of the 6th International Workshop on Formal Aspects of Component Software, FACS@FMWeek 2009, Eindhoven, The Netherlands, November 2-3, 2009*, sous la dir. de Sun MENG et Bernhard SCHÄTZ, t. 263, Electronic Notes in Theoretical Computer Science, Elsevier, 2009, p. 5-30, DOI : 10.1016/j.entcs.2010.05.002.

[And+10a] Pascal ANDRÉ et al., « Contract-based Verification of Kmelia Component Assemblies using Event-B », in : *FESCA @ Etaps2010*, Paphos, Greece, mars 2010, p. 1, URL : https://hal.science/hal-00483755.

[And+10b] Pascal ANDRÉ et al., « Using Event-B to Verify the Kmelia Components and Their Assemblies », in : *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, sous la dir. de Marc FRAPPIER et al., t. 5977, Lecture Notes in Computer Science, Springer, 2010, p. 410, DOI : 10.1007/978-3-642-11811-1_43.

[And+17] Pascal ANDRE et al., « Un outil d'assistance à la construction de tests de modèles à composants et services », in : *16èmes journées AFADL Approches Formelles dans l'Assistance au Développement de Logiciels*, Montpellier, France, juin 2017, URL : https://hal.science/hal-01628306.

[And19] Pascal ANDRÉ, « Case Studies in Model-Driven Reverse Engineering », in : *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*, sous la dir. de Slimane HAMMOUDI, Luís Ferreira PIRES et Bran SELIC, SciTePress, 2019, p. 256-263.

[And+21] Pascal Andre et al., « Raffinement de protocoles de communication par transformation de modèle », in : *13ème Colloque sur la Modélisation des Systèmes Réactifs (MSR'21)*, CNAM, Paris, France, sept. 2021, URL : https://hal.science/hal-04203186.

[And+23] Pascal André et al., « A Review of Core Operational Business-IT Alignment », in : *31st International Conference on Information Systems Development (ISD 2023)*, sous la dir. de Miguel Mira da Silva et al., Conference proceedings, Lisbon, Portugal : ICT / Association for Information Systems, août 2023, URL : https://aisel.aisnet.org/isd2014/proceedings2023/modelling/1/.

[And95] Pascal André, « Methodes formelles et a objets pour le developpement du logiciel : etudes et propositions », Thèse de doctorat dirigée par BEZIVIN, J. Sciences appliquées Rennes 1 1995, thèse de doct., 1995, 1 vol. (268 p.) URL : http://www.theses.fr/1995REN10052.

[Anq+09] Nicolas Anquetil et al., « JavaCompExt : Extracting Architectural Elements from Java Source Code », in : *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, sous la dir. d'Andy Zaidman, Giuliano Antoniol et Stéphane Ducasse, IEEE Computer Society, 2009, p. 317-318, DOI : 10.1109/WCRE.2009.53.

[AP18] Gul Agha et Karl Palmskog, « A Survey of Statistical Model Checking », in : *ACM Trans. Model. Comput. Simul.* 28.*1* (jan. 2018), ISSN : 1049-3301, DOI : 10.1145/3158668, URL : https://doi.org/10.1145/3158668.

[APC] Pôle Pédagogie Nantes université, *L'approche par compétences à la Faculté des sciences et des techniques Guide à l'intention des enseignants et enseignants-chercheurs*, Version 2, 2022, URL : https://sciences-techniques.univ-nantes.fr/medias/fichier/guide-apc-2022v2-241022vimp_1667836679299-pdf.

[AR92] Pascal André et Jean-Claude Royer, « Optimizing Method Search with Lookup Caches and Incremental Coloring », in : *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1992, Vancouver, British Columbia, Canada, October 18-22, 1992*, sous la dir. de John R. Pugh, ACM, 1992, p. 110-126, DOI : 10.1145/141936.141947.

[AR96] Pascal André et Jean-Claude Royer, « Ingénierie objet : Concepts et techniques », in : *8 : Ingénierie objet : Concepts et techniques*, ISBN 2-7296-0642-4, InterEditions/Masson, 1996, chap. Spécifications formelles dans le développement à objets, p. 271-314, ISBN : 2-7296-0642-4.

[ARR00] Pascal André, Annya Romanczuk et Jean-Claude Royer, « Checking the Consistency of UML Class Diagrams Using Larch Prover », in : *Rigorous Object-Oriented Methods, ROOM 2000, York, UK, 17 January 2000*, Workshops in Computing, BCS, 2000, URL : http://ewic.bcs.org/content/ConWebDoc/4207.

[AT20] Pascal André et Mohammed El Amin Tebib, « Refining Automation System Control with MDE », in : *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, sous la dir. de Slimane Hammoudi, Luís Ferreira Pires et Bran Selic, SCITEPRESS, 2020, p. 425-432, DOI : 10.5220/0009147804250432.

[AT21] Pascal André et Mohammed El Amin Tebib, « More Automation in Model Driven Development », in : *Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings*, sous la dir. de J. Christian Attiogbé et Sadok Ben Yahia, t. 12732, Lecture Notes in Computer Science, Springer, 2021, p. 75-83, DOI : 10.1007/978-3-030-78428-7_7.

[AT24] Pascal André et Mohammed El Amin Tebib, « Assistance in Model Driven Development Toward an automated transformation design process », in : *Complex Syst. Informatics Model. Q.* 38 (2024), to appear, p. 1-46.

[Atk02] C. Atkinson, *Component-based Product Line Engineering with UML*, Addison-Wesley object technology series, Addison-Wesley, 2002, ISBN : 9780201737912.

[Att07] Christian Attiogbé, « Contributions aux approches formelles de développement de logiciels : Intégration de méthodes formelles et analyse multifacette », Habilitation à diriger des recherches, Université de Nantes, sept. 2007, URL : https://theses.hal.science/tel-00481602.

[Aud09] Laurent Audibert, *UML 2 : De l'apprentissage à la pratique*, Info +, Editions Ellipses, 2009, ISBN : 9782729852696.

[AV01a] Pascal André et Alain Vailly, *Conception de systèmes d'information ; Panorama des méthodes et des techniques*, t. 1, Collection Technosup, ISBN 2-7298-0479-X, Editions Ellipses, 2001, ISBN : 2-7298-0479-X.

[AV01b] Pascal André et Alain Vailly, *Spécification des logiciels ; Deux exemples de pratiques récentes : Z et UML*, t. 2, Collection Technosup, ISBN 2-7298-0774-8, Editions Ellipses, 2001, ISBN : 2-7298-0774-8.

[AV02] Pascal André et Alain Vailly, *Exercices corrigés de conception logicielle ; Modélisation des Systèmes d'Information par la pratique*, t. 3, Collection Technosup, ISBN 2-7298-1289-X, Editions Ellipses, 2002, ISBN : 2-7298-1289-X.

[AV03a] Pascal André et Alain Vailly, *Exercices corrigés en UML ; Passeport pour une maîtrise de la notation.* T. 5, Collection Technosup, ISBN 2-7298-1725-5, Editions Ellipses, 2003, ISBN : X.

[AV03b] Pascal André et Alain Vailly, « Partie 2 de l'ouvrage collectif Piloter les technologies de l'informatique et des télécoms », in : Editions WEKA, nov. 2003, chap. 6 : Etudes de cas : Développement de logiciel avec UML, 56 pages.

[AV03c] Pascal André et Alain Vailly, « Partie 7 de l'ouvrage collectif Piloter les technologies de l'informatique et des télécoms », in : Editions WEKA, nov. 2003, chap. 10 : Développement de logiciel avec UML, 59 pages.

[AV04] Pascal André et Alain Vailly, *Exercices corrigés en langage Z ; Les spécifications formelles par la pratique*, t. 4, Collection Technosup, ISBN 2-7298-1942-8, Editions Ellipses, 2004, ISBN : 2-7298-1942-8.

[AV13] Pascal André et Alain Vailly, *Développement de logiciel avec UML2 et OCL ; cours et exercices corrigés*, t. 6, Collection Technosup, ISBN 9782729883539, Editions Ellipses, 2013, p. 384, ISBN : 9782729883539.

[AV16] Aruna S. et VIT Vellore, « Security in Web Services- Issues and Challenges », in : *International Journal of Engineering Research and Technology (IJERT)* 09.*09* (sept. 2016), DOI : http://dx.doi.org/10.17577/IJERTV5IS090245.

[AZZ15] Yu An, Yu Zhang et Bo Zeng, « The reliable hub-and-spoke design problem : Models and algorithms », in : *Transportation Research Part B : Methodological* 77 (2015), p. 103 -122, ISSN : 0191-2615, DOI : https://doi.org/10.1016/j.trb.2015.02.006, URL : http://www.sciencedirect.com/science/article/pii/S0191261515000260.

[B+99] Béatrice Bérard et al., *Vérification de logiciels : techniques et outils du model-checking*, Vuibert Informatique, ISBN 2-7117-8646-3, Vuibert, 1999, ISBN : 2-7117-8646-3.

[Bah+19] Abdramane Bah et al., « Federation of Services from Autonomous Domains with Heterogeneous Access Control Models », in : *Information and Cyber Security - 18th International Conference, ISSA 2019, Johannesburg, South Africa, August 15, 2019, Proceedings*, sous la dir. d'Hein S. Venter et al., t. 1166, Communications in Computer and Information Science, Springer, 2019, p. 83-98.

[Bah20] Abdramane Bah, « Interopérabilité et sécurité des systèmes d'information : application aux systèmes de gestion de l'éducation », Thèse de doctorat dirigée par Attiogbé, ChristianKonaté, Jacqueline et André, Pascal Informatique Nantes 2020, thèse de doct., Université de Nantes (Unam) et Université des Sciences Techniques et Technologiques de Bamako (Mali), 2020, URL : http://www.theses.fr/2020NANT4028.

[Bah+20a] Abdramane Bah et al., « Service Promotion in a Federation of Security Domains », in : *Colloque Africain sur la Recherche en Informatique et en Mathématiques Appliquées*, Thiès, Senegal, oct. 2020, URL : https://hal.science/hal-02909605.

[Bah+20b] Abdramane Bah et al., « Service Promotion in a Federation of Security Domains », in : *ARIMA J.* 34 (2020), p. 5, DOI : 10.46298/arima.6757.

[Bah+23] Ivan Baheux et al., « DroidSecTester : Towards context-driven modelling and detection of Android application vulnerabilities », in : *IWSF & SHIFT Workshop, Co-located with the 34th International Symposium on Software Reliability Engineering (ISSRE 2023), October 9 – 12, Florence, Italy*, Florence, Italy, oct. 2023, p. xx, URL : https://www.shiftworkshop.net/.

[Bal06] H. Balzert, *UML 2 compact*, Compact (Paris. 2006), Eyrolles, 2006, ISBN : 9782212117530.

[Bar+10] J. Bartolomei et al., « Analysis and applications of design structure matrix, domain mapping matrix, and engineering system matrix frameworks », in : (2010), URL : http://ardent.mit.edu/real_options/Real_opts_papers/Jennifer%20mini%20thesis.pdf.

[Bar13] Dominique Barjot, « Existe-t-il un modèle français de l'ingénierie ? », in : *Entreprises et histoire* 71.*2* (2013), p. 6, DOI : 10.3917/eh.071.0006, URL : https://doi.org/10.3917/eh.071.0006.

[Bar91] Franck Barbier, « Une approche objet dédiée à la fonction production d'une entreprise manufacturière : application à la gestion de production », Thèse de doctorat dirigée par Haurat, Alain Informatique Chambéry 1991, thèse de doct., 1991, 1 vol. (209 p.) URL : http://www.theses.fr/1991CHAMS001.

[BBG07] M.H. ter Beek, A. Bucchiarone et S. Gnesi, « Formal Methods for Service Composition », in : *Annals of Mathematics, Computing & Teleinformatics* 1.*5* (2007), p. 1-10.

[BCK12] Len Bass, Paul Clements et Rick Kazman, *Software Architecture in Practice*, 3rd, Addison-Wesley Professional, 2012, ISBN : 0321815734.

[BCW17] Marco Brambilla, Jordi Cabot et Manuel Wimmer, *Model-Driven Software Engineering in Practice : Second Edition*, 2nd, Morgan & Claypool Publishers, 2017, isbn : 1627057080, 9781627057080.

[Ben+24a] Ali Benjilany et al., « Détection d'anti-patrons d'alignement dans les SI : Vers une approche automatisée », in : *INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID 2024)*, Nancy, France, mai 2024, url : https://hal.science/hal-04557692.

[Ben+24b] Ali Benjilany et al., « Towards a link mapping and evaluation approach for Core Operational Business-IT Alignment », in : *26th International Conference on Enterprise Information Systems (ICEIS 2024)*, Angers, France, avr. 2024, url : https://hal.science/hal-04447879.

[Ber+10] Elisa Bertino et al., *Security for Web Services and Service-Oriented Architectures*, Springer, 2010, isbn : 978-3-540-87741-7.

[BF21] Alexander C. Bock et Ulrich Frank, « Low-Code Platform », in : *Business Information Systems Engineering* 63.*6* (nov. 2021), 733–740, issn : 1867-0202, doi : 10.1007/s12599-021-00726-8, url : http://dx.doi.org/10.1007/s12599-021-00726-8.

[BFGM11] Marie Bia-Figueiredo, Yves Gillette et Chantal Morley, *Processus métiers et S.I. - Gouvernance, management, modélisation - 3e édition : Gouvernance, management, modélisation*, 3e éd., Management des systèmes d'information, Dunod, 2011, isbn : 9782100562169.

[BH94] Jonathan P. Bowen et Michael G. Hinchey, « Seven More Myths of Formal Methods », in : *FME'94 Symposium*, sous la dir. de Naftalin. Maurice, Tim Denvir et Miquel (Eds) Bertran, t. 873, LNCS, Barcelona, Spain : Springer Verlag, oct. 1994, p. 105-117.

[Bha17] Prithvi Bhattacharya, « Modelling Strategic Alignment of Business and IT through Enterprise Architecture : Augmenting Archimate with BMM », in : *Procedia Computer Science* 121 (2017), CENTERIS 2017, p. 80 -88, issn : 1877-0509.

[BHHM19] Thierry Burger-Helmchen, Caroline Hussler et Paul Muller, *Management*, Vuibert, 2019.

[BHM06] Tomas Barros, Ludovic Henrio et Eric Madelaine, « Model-Checking Distributed Components : The Vercors Platform », in : *International Workshop on Formal Aspects of Component Software (FACS'06)*, Prague : Electronic Notes in Theoretical Computer Science (ENTCS), sept. 2006.

[BHP06] Tomas Bures, Petr Hnetynka et Frantisek Plasil, « SOFA 2.0 : Balancing Advanced Features in a Hierarchical Component Model », in : *SERA '06 : Fourth IC on Software Engineering Research, Management and Applications*, IEEE Computer Society, 2006, p. 40-48, isbn : 0-7695-2656-X, doi : http://dx.doi.org/10.1109/SERA.2006.62.

[BHS91] Ferenc Belina, Dieter Hogrefe et Amardeo Sarma, *SDL with Applications from Protocol Specification*, The BCS Practitioner, ISBN 0-13-785890-6, Prentice Hall, 1991, isbn : 0-13-785890-6.

[Bi+08] Z. M. Bi et al., « Reconfigurable manufacturing systems : the state of the art », in : *International Journal of Production Research* 46.*4* (fév. 2008), 967–992, issn : 1366-588X, doi : 10.1080/00207540600905646.

[BJP10] Antoine Beugnard, Jean-Marc Jézéquel et Noël Plouzeau, « Contract Aware Components, 10 years after », in : *Proceedings International Workshop on Component and Service Interoperability, WCSI 2010, Málaga, Spain, 29th June 2010*, sous la dir. de Javier Cámara, Carlos Canal et Gwen Salaün, t. 37, EPTCS, 2010, p. 1-11, doi : 10.4204/EPTCS.37.1.

[BK08] Christel Baier et Joost-Pieter Katoen, *Principles of model checking*, MIT Press, 2008, isbn : 978-0-262-02649-9.

[BM88] Didier Banos et Guy Malbosc, *Merise Pratique, Tome 1 : Les points clés de la méthode*, EAN 9782212039009, Editions Eyrolles, 1988, isbn : ASIN : B0014JJPUW.

[BMD10] Leila BOUBAKER, Leila MELLAL et Mébarek DJEBABRA, « Modèle DIC (Données ? Informations ? Connaissances) Outil support pour le développement des mémoires projets », in : *La Revue des Sciences de Gestion* 243-244.*3* (2010), p. 153, doi : 10.3917/rsg.243.0153, url : https://doi.org/10.3917/rsg.243.0153.

[Boe82] Barry W. Boehm, « Les facteurs du coût logiciel », in : *Technique et Science Informatique* 1.*1* (1982), p. 5-24.

[Boe88] Barry W. Boehm, « A spiral model of software development and enhancement. », in : *IEEE Computer* 21.*5* (mai 1988), p. 61-72.

[Bor+04] Marc Born et al., « Model-driven development and testing - a case study », in : *First European Workshop on MDA with Emphasis on Industrial Application*, Twente Univ., 2004, p. 97-104.

[BR85] Barry W. Boehm et Rony Ross, « La gestion de projets logiciels selon la théorie W : une étude de cas », in : *Revue Génie Logiciel et Systèmes Experts* 15 (sept. 1985), p. 4-20.

[Bra88] Gilles Bracon, « La spécification de logiciels dans l'industrie », in : *BIGRE + GLOBULE* 58 (jan. 1988), IRISA-AFCET, p. 12-19.

[Bro10] Antonio Brogi, « On the potential advantages of exploiting behavioural information for contract-based service discovery and composition », in : *Journal of Logic and Algebraic Programming* (mars 2010), issn : 15678326, doi : 10.1016/j.jlap.2010.01.001, url : http://dx.doi.org/10.1016/j.jlap.2010.01.001.

[Bro18] Manfred Broy, « The Leading Role of Software and Systems Architecture in the Age of Digitization », in : *The Essence of Software Engineering*, sous la dir. de Volker Gruhn et Rüdiger Striemer, Springer, 2018, p. 1-23, doi : 10.1007/978-3-319-73897-0_1, url : https://doi.org/10.1007/978-3-319-73897-0_1.

[Bru+06] Eric Bruneton et al., « The FRACTAL component model and its support in Java », in : *Software Practice and Experience* 36.*11-12* (2006), p. 1257-1284, doi : 10.1002/spe.767.

[Bru+14] Hugo Bruneliere et al., « MoDisco : a Model Driven Reverse Engineering Framework », in : *Inf. Softw. Technol.* 56.*8* (2014), p. 1012-1032.

[Buc+20] Antonio Bucchiarone et al., « Grand challenges in model-driven engineering : an analysis of the state of the research », in : *Software and Systems Modeling* 19.*1* (jan. 2020), p. 5-13, doi : 10.1007/s10270-019-00773-6, url : https://doi.org/10.1007/s10270-019-00773-6.

[Cal90] Jean-Paul Calvez, *Spécification et conception des systèmes : une méthodologie*, Masson, mai 1990.

[Cap11] J. Capirossi, *Architecture d'entreprise*, Collection Management et informatique, Hermes Science Publications, 2011, isbn : 9782746229808, url : http://books.google.co.ma/books?id=IxX_2ySnZgcC.

[Car16] Olivier Cardin, « Contribution à la conception, l'évaluation et l'implémentation de systèmes de production cyber-physiques », Habilitation à diriger des recherches, Université de nantes, déc. 2016, url : https://theses.hal.science/tel-01443318.

[Car19] Olivier Cardin, « Classification of cyber-physical production systems applications : Proposition of an analysis framework », in : *Computers in Industry* 104 (jan. 2019), p. 11 -21, doi : 10.1016/j.compind.2018.10.002, url : https://hal.science/hal-01900263.

[CDT23] O. Cardin, W. Derigent et D. Trentesaux, *Digitalisation et contrôle des systèmes industriels cyber-physiques : Concepts, technologies et applications*, G - Reference,Information and Interdisciplinary Subjects Series, ISTE Editions Limited, 2023, isbn : 9781789480856, url : https://books.google.fr/books?id=D3_ZEAAAQBAJ.

[CF21] Erica Capawa Fotsoh, « Contribution à la reconfiguration des lignes de production : définition et démarche de choix de configurations alternatives. », Theses, Université de Nantes (UN), FRA., oct. 2021, URL : https://theses.hal.science/tel-03789836.

[CG12] Jordi Cabot et Martin Gogolla, « Object Constraint Language (OCL) : A Definitive Guide », in : *Formal Methods for Model-Driven Engineering*, sous la dir. de Marco Bernardo, Vittorio Cortellessa et Alfonso Pierantonio, t. 7320, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, p. 58-90, ISBN : 978-3-642-30981-6, DOI : 10.1007/978-3-642-30982-3_3, URL : http://dx.doi.org/10.1007/978-3-642-30982-3_3.

[Cha+09] Céline Chadenas et al., *Pour une meilleure adéquation entre pression humaine et ressources littorales : évaluer la capacité d'accueil du territoire*, 2, 2009, p. 57-88, URL : http://cahiers-nantais.fr/index.php?id=778.

[Cha+10] Céline Chadenas et al., *Evaluer la capacité d'accueil et de développement des territoires littoraux : guide pratique 2e édition*, oct. 2010, p. 104, URL : https://www.researchgate.net/publication/313861085_Evaluer_la_capacite_d'accueil_et_de_developpement_des_territoires_littoraux_guide_pratique_2e_edition/citations.

[Chr13] G. Chryssolouris, *Manufacturing Systems : Theory and Practice*, Mechanical Engineering Series, Springer New York, 2013, ISBN : 9781475722130.

[CL02] Ivica Crnkovic et Magnus Larsson, éd., *Building Reliable Component-Based Software Systems*, Artech House publisher, 2002, ISBN : ISBN 1-58053-327-2, URL : http://www.mrtc.mdh.se/index.phtml?choice=publications&id=0411.

[Cla11] Mickaël Clavreul, « Model and Metamodel Composition : Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques », thèse de doct., Université Rennes 1, 2011.

[Cle96] Paul C. Clements, « A Survey of Architecture Description Languages », in : *IWSSD '96 : Proceedings of the 8th International Workshop on Software Specification and Design*, Washington, DC, USA : IEEE Computer Society, 1996, p. 16, ISBN : 0-8186-7361-3.

[Col+15] Tim Coltman et al., « Strategic IT alignment : twenty-five years on », in : *Journal of Information Technology* 30.*2* (juin 2015), p. 91-100, ISSN : 1466-4437, DOI : 10.1057/jit.2014.35.

[CPA14] Céline Chadenas, Patrick Pottier et Pascal André, « Evaluer la capacité d'accueil des territoires littoraux. », in : *26èmes Journées scientifiques, la Société d'Ecologie Humaine (SEH), Habiter le littoral. Enjeux écologiques et humains contemporains*, 16 au 18 octobre 2014, Marseille, France : Observatoire Hommes-Milieux (OHM), 2014, p. 10.

[CPA16] Céline Chadenas, Patrick Pottier et Pascal André, « Evaluer la capacité d'accueil des territoires littoraux. Questions de méthode », in : *Habiter le littoral. Enjeux contemporains*, sous la dir. de Robert Samuel et Melin Hélène, Presses Universitaires de Provence, 2016, p. 359-374, isbn : 9791032000847, url : https://presses-universitaires.univ-amu.fr/habiter-littoral-0.

[CRS01] Corine Cauvet et Camille Rosenthal-Sabroux, *Ingénierie des systèmes d'information*, Hermès Science, 2001, isbn : 978-2-7462-0219-1.

[CSS09] Anis Charfi, Artur Schmidt et Axel Spriestersbach, « A Hybrid Graphical and Textual Notation and Editor for UML Actions », in : *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, Berlin, Heidelberg : Springer-Verlag, 2009, p. 237-252, isbn : 978-3-642-02673-7, doi : 10.1007/978-3-642-02674-4_17, url : http://dx.doi.org/10.1007/978-3-642-02674-4_17.

[CV22] Jordi Cabot et Antonio Vallecillo, « Modeling should be an independent scientific discipline », en, in : *Software and Systems Modeling* (août 2022), issn : 1619-1374, doi : 10.1007/s10270-022-01035-8, url : https://doi.org/10.1007/s10270-022-01035-8 (visité le 13/10/2022).

[DBB11] Karim Doumi, Salah Baïna et Karim Baïna, « Modeling Approach for Business IT Alignment », in : *Proceedings of ICEIS 2011, Volume 4, Beijing, China*, 2011, p. 457-464.

[DBW91] David M Dilts, Neil P Boyd et HH Whorms, « The evolution of control architectures for automated manufacturing systems », in : *Journal of manufacturing systems* 10.*1* (1991), p. 79-93.

[DCL08] Zuohua Ding, Zhenbang Chen et Jing Liu, « A Rigorous Model of Service Component Architecture », in : *Electr. Notes Theor. Comput. Sci.* 207 (2008), p. 33-48.

[DD23] William Derigent et Michael David, « SADHoA- A Switching-Agnostic Dynamic Holonic Architecture », in : *13th International Workshop on Service-Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future (SOHOMA2023)*, Annecy , France, sept. 2023.

[Deb+10] Mourad Debbabi et al., *Verification and Validation in Systems Engineering : Assessing UML/SysML Design Models*, Springer, 2010, isbn : 9783642152276.

[Del20] Benoit Delahaye, « Modeling and Verification of Systems with Uncertainties », Habilitation à diriger des recherches, Université de Nantes, déc. 2020, url : https://hal.science/tel-03636882.

[Der+23] William Derigent et al., « Generic Aggregation Model for Reconfigurable Holonic Control Architecture – The GARCIA Framework », in : *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future*, sous la dir. de Theodor Borangiu, Damien Trentesaux et Paulo Leitão, Cham : Springer International Publishing, 2023, p. 407-422, isbn : 978-3-031-24291-5.

[DGD05] Dragan Djuric, Dragan Gasevic et Vladan Devedzic, « Ontology Modeling and MDA », in : *Journal of Object Technology* 4.*1* (2005), p. 109-128.

[DM04] Bernard Debauche et Patrick Mégard, *BPM, Business Process Management : pilotage métier de l'entreprise*, Management et informatique, Hermes Science Publications, 2004, isbn : 9782746208520.

[Don00] Stijn van Dongen, « Graph Clustering by Flow Simulation », en, PhD Thesis, University of Utrecht, mai 2000, isbn : ISBN 90–393–2408–5.

[DR12] Philippe Desfray et Gilbert Raymond, *TOGAF en pratique : Modèles d'architecture d'entreprise*, 1re éd., Management des systèmes d'information, Dunod, 2012, isbn : 9782100581016.

[DR+22] Davide Di Ruscio et al., « Low-code development and model-driven engineering : Two sides of the same coin ? », in : *Software and Systems Modeling* 21.*2* (jan. 2022), 437–446, issn : 1619-1374, doi : 10.1007/s10270-021-00970-2, url : http://dx.doi.org/10.1007/s10270-021-00970-2.

[DR75] Joël De Rosnay, *Le macroscope. Vers une vision globale*, Seuil, 1975, isbn : 9782021186444.

[DREP12] Davide Di Ruscio, Romina Eramo et Alfonso Pierantonio, « Model Transformations », in : *Formal Methods for Model-Driven Engineering : 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, sous la dir. de Marco Bernardo, Vittorio Cortellessa et Alfonso Pierantonio, Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 91-136, isbn : 978-3-642-30982-3.

[DVL13] Lonneke Dikmans et Ronald Van Luttikhuizen, *SOA Made Simple : Discover the true meaning behind the buzzword that is "Service Oriented Architecture"*, Birmingham, UK : Packt Pub, 2013, isbn : 978-1-84968-417-0.

[EB12] Steven D. Eppinger et Tyson R. Browning, *Design Structure Matrix Methods and Applications*, Anglais, Cambridge, Mass : MIT Press, 2012, isbn : 978-0-262-01752-7.

[ED19] Chantal Enguehard et Anaïs Danet, « Les Systèmes Inéquitables Numériques (SIN) », in : *Journées Réseaux de l'Enseignement et de la Recherche JRES2019*, Dijon, France, déc. 2019, url : `https://hal.science/hal-02434239`.

[ER05] Anne Etien et Colette Rolland, « Measuring the fitness relationship », English, in : *Requirements Engineering* 10.*3* (2005), p. 184-197, issn : 0947-3602, doi : `10.1007/s00766-005-0003-8`, url : `http://dx.doi.org/10.1007/s00766-005-0003-8`.

[Erl05] Thomas Erl, éd., *Service-Oriented Architecture - Concepts, Technology, and Design*, Boston, MA, USA : Prentice Hall, 2005, isbn : 0-13-185858-0.

[Ern03] Michael D Ernst, « Static and dynamic analysis : Synergy and duality », in : *WODA 2003 : ICSE Workshop on Dynamic Analysis*, 2003, p. 24-27.

[Eva04] Eric Evans, *Domain-Driven Design - Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2004.

[Fer+16] David Ferraiolo et al., « Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC) », in : *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control - ABAC '16*, ACM Press, 2016, p. 13-24, isbn : 978-1-4503-4079-3.

[FG12] Peter H. Feiler et David P. Gluch, *Model-Based Engineering with AADL : An Introduction to the SAE Architecture Analysis & Design Language*, 1st, Addison-Wesley Professional, 2012, isbn : 0321888944.

[FMS08] Sanford Friedenthal, Alan Moore et Rick Steiner, *A Practical Guide to SysML : Systems Modeling Language*, San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008, isbn : 0123743796, 9780080558363, 9780123743794.

[Fow02] Martin Fowler, *Patterns of Enterprise Application Architecture*, Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002, isbn : 0321127420.

[Fow04] Martin Fowler, *UML 2.0*, Campus Press Reference, ISBN 2-7440-1713-2, Pearson Education France, 2004, p. 201, isbn : 2-7440-1713-2.

[GA94] D. Garlan et R Allen, « Formalizing Architectural Connection », in : *Proceedings of the 16th ICSE*, IEEE Computer Society Press, 1994, p. 71-80.

[Gau+96] Marie-Claude Gaudel et al., *Précis de génie logiciel*, Enseignement, ISBN 2-225-85189-1, Masson, 1996, p. 160, isbn : 2-225-85189-1.

[GBR05] Martin Gogolla, Jörn Bohling et Mark Richters, « Validating UML and OCL models in USE by automatic snapshot generation », in : *Software and Systems Modeling* 4.*4* (2005), p. 386-398, issn : 1619-1366.

[GC14] Carlos A. Gonzalez et Jordi Cabot, « Formal verification of static software models in MDE : A systematic review », in : *Information and Software Technology* 56.*8* (2014), p. 821-838, issn : 0950-5849, doi : https://doi.org/10.1016/j.infsof.2014.03.003, url : https://www.sciencedirect.com/science/article/pii/S0950584914000627.

[GG96] George Gardarin et Olivier Gardarin, *Le Client-Serveur*, ISBN 2-212-08876-0, Eyrolles, 1996, isbn : 2-212-08876-0.

[GM99] Sudipto Ghosh et Aditya P. Mathur, « Issues in Testing Distributed Component-Based Systems », in : *In First International ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.

[Goe13] Virginie Goepp, « Modèles et Méthodes pour l'Alignement des Systèmes d'Information en Entreprises Industrielles », Habilitation à diriger des recherches, INSA de Strasbourg, 2013, 1 vol. (174 p.) url : http://www.theses.fr/1991CHAMS001.

[GQ15] Francisco Gamboa Quintanilla, « Couplage des Architectures Holonique et Orientée-Services pour la Conception de Systèmes de Production Agiles », Thèse de doctorat dirigée par Castagna, Pierre et Cardin, Olivier Automatique et informatique appliquée, thèse de doct., Université Nantes-Angers-Le Mans-COMUE, 2015, 1 vol. (163 p.) url : http://www.theses.fr/2015NANT2078.

[Gro04] Hans-Gerhard Gross, *Component-based Software Testing With Uml*, SpringerVerlag, 2004, isbn : 354020864X, 9783540208648.

[Gro11a] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/-View/Transformation, version 1.1*, rapp. tech., http://www.omg.org/spec/QVT/1.1/ : Object Management Group, jan. 2011.

[Gro11b] Object Management Group, *The OMG Unified Modeling Language Specification, version 2.4.1*, rapp. tech., UML 2.4 Superstructure Specification available at http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF : Object Management Group, août 2011.

[Gro11c] Object Management Group, *The OMG Unified Modeling Language Specification, version 2.4.1*, rapp. tech., UML 2.4 Infrastructure Specification available at `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF` : Object Management Group, août 2011.

[GS05] Gregor Gossler et Joseph Sifakis, « Composition for component-based modeling », in : *Sci. Comput. Program.* 55.*1-3* (2005), p. 161-183, URL : `papers/Goessler-Sifakis-05.pdf`.

[Gue94] Nicolas Guelfi, « Les réseaux algébriques hiérarchiques : un formalisme de spécifications structurées pour le développement de systèmes concurrents », PhD Thesis, Université Paris XI Orsay, sept. 1994.

[GV03] Claude Girault et Rüdiger Valk, *Petri nets for systems engineering - a guide to modeling, verification, and applications*, Springer, 2003, ISBN : 978-3-540-41217-5, URL : `http://www.springer.com/computer/swe/book/978-3-540-41217-5`.

[HA04] Henri Habrias et Pascal André, « Précondition et invariant - l'écriture de contraintes mal adaptées à un paradigme de spécification », in : *Actes du XXIIème Congrès INFORSID, Biarritz, France, 25-28 mai, 2004*, 2004, p. 387-403.

[Hab97] Henri Habrias, *Dictionnaire encyclopédique du génie logiciel*, Masson, 1997.

[Hal90] A. Hall, « Seven myths of formal methods », in : *IEEE Software* 7.*5* (1990), p. 11-20.

[Han11] Morten Olav Hansen, « Exploration of UML State Machine implementations in Java », mém. de mast., Norway : University of Oslo, fév. 2011.

[Hay92] Ian Hayes, éd., *Specification Case Studies*, 2e éd., International Series in Computer Science, ISBN 0-13-832544-8, Prentice-Hall, 1992, ISBN : 0-13-832544-8.

[Het10] Thomas Hettel, « Model round-trip engineering », thèse de doct., Queensland University of Technology, 2010, URL : `https://eprints.qut.edu.au/32082/`.

[HHLGN22a] Myriam Houssay-Holzschuch, Renaud Le Goix et Camille Noûs, « Encadrer des thèses : d'abord, ne pas nuire. (1) État d'un champ de recherche », in : *EchoGéo 59* (fév. 2022), DOI : `10.4000/echogeo.22889`, URL : `https://doi.org/10.4000/echogeo.22889`.

[HHLGN22b] Myriam Houssay-Holzschuch, Renaud Le Goix et Camille Noûs, « Encadrer des thèses : d'abord, ne pas nuire. (2) Diriger c'est enseigner », in : *EchoGéo 60* (juin 2022), DOI : `10.4000/echogeo.23589`, URL : `http://journals.openedition.org/echogeo/23589`.

[Hoa72] Charles A.R. Hoare, « Proof of correctness of Data Representations », in : *Acta Informatica* 1 (1972), p. 271-281.

[Hod91] Ralph Hodgson, « The X-Model : A Process Model for Object-Oriented Software Development », in : *Actes des quatrièmes journées internationales sur le Génie logiciel et ses applications*, Toulouse, déc. 1991, p. 713-728.

[HP14] Knut Hinkelmann et Alex Pasquini, « Supporting Business and IT Alignment by Modeling Business and IT Strategy and Its Relations to Enterprise Architecture », in : *Enterprise Systems Conference, ES 2014, Shangai, China, August 2-3, 2014*, IEEE, 2014, p. 149-154, doi : 10.1109/ES.2014.65.

[HRS12] Khalid Hafeez, Qasim Rajpoot et Awais Shibli, « Interoperability among access control models », in : *2012 15th International Multitopic Conference (INMIC)*, Islamabad, Punjab, Pakistan : IEEE, déc. 2012, p. 111-118, isbn : 978-1-4673-2252-2.

[HSB99] Brian Henderson-Sellers et Franck Barbier, « Are UML's Aggregation Kinds Meaningfull ? », in : *L'Objet* 5.*3-4* (déc. 1999), p. 21-47.

[HSG12] Regina Hebig, Andreas Seibel et Holger Giese, « On the Unification of Megamodels », en, in : *Electronic Communications of the EASST* (2012), Volume 42 : Multi-Paradigm Modeling 2010, doi : 10.14279/TUJ.ECEASST.42.704.713, url : http://journal.ub.tu-berlin.de/eceasst/article/view/704/713.

[Hu+14] Vincent C. Hu et al., *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*, rapp. tech. NIST SP 800-162, National Institute of Standards et Technology, jan. 2014, url : http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf.

[HV93] J. C. Henderson et N. Venkatraman, « Strategic Alignment : Leveraging Information Technology for Transforming Organizations », in : *IBM Syst. J.* 32.*1* (jan. 1993), p. 4-16, issn : 0018-8670.

[ISA10] ISA, *ISA95 IEC/ISO62264 Enterprise - Control System Integration Part1-Part2*, 2010, url : https://www.isa.org/products/ansi-isa-95-00-01-2010-iec-62264-1-mod-enterprise.

[ISO11] ISO/IEC/IEEE, « Systems and software engineering – Architecture description », in : *ISO/IEC/IEEE 42010 :2011(E) (Revision of ISO/IEC 42010 :2007 and IEEE Std 1471-2000)* (jan. 2011), p. 1 -46, doi : 10.1109/IEEESTD.2011.6129467.

[ISO14] ISO, *ISO22400 : 2—Automation Systems and Integration—Key Performance Indicators (KPIs) for Manufacturing Operations Management—Part. 2 : Definitions and descriptions*, 2014.

[Jac+92] Ivar Jacobson et al., *Object-Oriented Software Engineering : A Use Case Driven Approach*, ISBN 0-20-154435-0, Addison Wesley, 1992, isbn : 0-20-154435-0.

[JM20] N. Julien et É. Martin, *Le jumeau numérique : De l'intelligence artificielle à l'industrie agile*, Dunod, 2020, isbn : 9782100814039.

[Jov+14] Marko Jovanović et al., « Virtual approach to holonic control of the tyre-manufacturing system », in : *Journal of Manufacturing Systems* 33.*1* (2014), 116–128, issn : 0278-6125, doi : 10.1016/j.jmsy.2013.07.005.

[Juë20] Rémi Juët, *La boîte à outils du manager-4e éd. : 51 fiches pratiques pour piloter son équipe*, Dunod, 2020.

[Kah+19] Nafiseh Kahani et al., « Survey and classification of model transformation tools », in : *Software and Systems Modeling* 18.*4* (2019), p. 2361-2397.

[Kai+23] Jan Kaiser et al., « A review of reference architectures for digital manufacturing : Classification, applicability and open issues », in : *Comput. Ind.* 149 (2023), p. 103923, doi : 10.1016/j.compind.2023.103923.

[Kal08] Jyri Kallela, « Federated identity management solutions », in : *TKK T-110.5190 Seminar on Internetworking*, 2008.

[Kar+06] G. Karsai et al., « A Taxonomy of Model Transformation », in : *Electronic Notes in Theoretical Computer Science* 152 (2006), Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), p. 125 -142, issn : 1571-0661.

[Ket+98] Nasser Kettani et al., *De Merise à UML*, ISBN 2-212-08997-X, Eyrolles, 1998, isbn : 2-212-08997-X.

[Kha+19] Muhammad Uzair Khan et al., « Landscaping systematic mapping studies in software engineering : A tertiary study », in : *Journal of Systems and Software* 149 (2019), p. 396-436, issn : 0164-1212, doi : https://doi.org/10.1016/j.jss.2018.12.018, url : https://www.sciencedirect.com/science/article/pii/S0164121218302784.

[Kle07] J. Kletti, *Manufacturing Execution System - MES*, Springer Berlin Heidelberg, 2007, isbn : 9783642080647.

[Kor10] Yoram Koren, *The global manufacturing revolution : product-process-business integration and reconfigurable systems*, John Wiley & Sons, mai 2010, isbn : 9780470618813, doi : 10.1002/9780470618813.

[Kor+99] Y. KOREN et al., « Reconfigurable Manufacturing Systems », in : *CIRP Annals* 48.*2* (jan. 1999), p. 527-540, ISSN : 0007-8506, DOI : 10.1016/S0007-8506(07)63232-6, URL : https://www.sciencedirect.com/science/article/pii/S0007850607632326 (visité le 16/09/2023).

[Kru95] Philippe KRUCHTEN, « The 4+1 View Model of Architecture », in : *IEEE Softw.* 12.*6* (nov. 1995), p. 42-50, ISSN : 0740-7459, DOI : 10.1109/52.469759, URL : http://dx.doi.org/10.1109/52.469759.

[Kur08] Ivan KURTEV, « State of the Art of QVT : A Model Transformation Language Standard », in : *Applications of Graph Transformations with Industrial Relevance*, sous la dir. d'Andy SCHÜRR, Manfred NAGL et Albert ZÜNDORF, Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 377-393, ISBN : 978-3-540-89020-1.

[Kus18] Andrew KUSIAK, « Smart manufacturing », in : *International Journal of Production Research* 56.*1-2* (2018), p. 508-517.

[KWB03] Anneke KLEPPE, Jos WARMER et Wim BAST, *MDA Explained : The Model Driven Architecture : Practice and Promise*, 1re éd., Object Technology Series, ISBN 0-321-19442-X, Addison-Wesley, 2003, p. 192, ISBN : 0-321-19442-X.

[Lan05] Kevin LANO, *Advanced Systems Design with Java, UML and MDA*, 1re éd., Computer Science, ISBN 0-7506-6496-7, Elsevier, 2005, p. 378, ISBN : 0-7506-6496-7.

[Lan13] Marc M. LANKHORST, *Enterprise Architecture at Work - Modelling, Communication and Analysis (3. ed.)* The Enterprise Engineering Series, Springer, 2013, ISBN : 978-3-642-29650-5.

[Lau86] Jean.-Louis LAURIÈRE, *Résolution de problèmes par l'Homme et la machine*, Eyrolles, 1986.

[Lec+07] Matthieu LECLERCQ et al., « Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset », in : *29th International Conference on Software Engineering (ICSE'07)*, 2007, p. 209-219, DOI : 10.1109/ICSE.2007.82.

[Lee08] Edward A. LEE, *Cyber Physical Systems : Design Challenges*, rapp. tech. UCB/EECS-2008-8, EECS Department, University of California, Berkeley, jan. 2008, URL : http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html.

[Leh+97] M.M. LEHMAN et al., « Metrics and laws of software evolution-the nineties view », in : *Proceedings Fourth International Software Metrics Symposium*, 1997, p. 20-32, DOI : 10.1109/METRIC.1997.637156.

[Let21] Timothy C. Lethbridge, « Low-Code Is Often High-Code, So We Must Design Low-Code Platforms to Enable Proper Software Engineering », in : *Leveraging Applications of Formal Methods, Verification and Validation*, Springer International Publishing, 2021, 202–212, isbn : 9783030891596, doi : 10.1007/978-3-030-89159-6_14, url : http://dx.doi.org/10.1007/978-3-030-89159-6_14.

[LM90] Jean-Louis Le Moigne, *La modélisation de systèmes complexes*, Dunod, 1990, isbn : 9782100043828.

[LMS10] Zhiming Liu, Charles Morisset et Volker Stolz, « rCOS : Theory and Tool for Component-Based Model Driven Development », in : *Proc. of FSEN 2009*, t. 5961, LNCS, Springer, 2010, p. 62-80.

[Lon09] Christophe Longépé, *Le projet d'urbanisation du SI : Cas concret d'architecture d'entreprise*, 4e éd., InfoPro. Management des systèmes d'information, Dunod, 2009, isbn : 9782100528837.

[Lon17] Jacques Lonchamp, *Introduction aux systèmes informatiques, Architectures, composants, mise en œuvre*, 1re éd., Infosup, ISBN 978-2-10-075944-6, Dunod, 2017, p. 416, isbn : 978-2-10-075944-6.

[LTL10] Yi-Hong Long, Zhi-Hong Tang et Xu Liu, « Attribute mapping for cross-domain access control », in : *2010 International Conference on Computer and Information Application*, Tianjin, China : IEEE, déc. 2010, p. 343-347, isbn : 978-1-4244-8598-7, doi : 10.1109/ICCIA.2010.6141607.

[Luo+21] Yajing Luo et al., « Characteristics and Challenges of Low-Code Development : The Practitioners' Perspective », in : *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '21, ACM, oct. 2021, doi : 10.1145/3475716.3475782, url : http://dx.doi.org/10.1145/3475716.3475782.

[MAA10a] Mohamed Messabihi, Pascal André et Christian Attiogbé, « Multilevel Contracts for Trusted Components », in : *Proceedings International Workshop on Component and Service Interoperability*, t. 37, EPTCS, 2010, p. 71-85.

[MAA10b] Mohamed Messabihi, Pascal André et J. Christian Attiogbé, « Multilevel Contracts for Trusted Components », in : *Proceedings International Workshop on Component and Service Interoperability, WCSI 2010, Málaga, Spain, 29th June 2010*, sous la dir. de Javier Cámara, Carlos Canal et Gwen Salaün, t. 37, EPTCS, 2010, p. 71-85, doi : 10.4204/EPTCS.37.6.

[MAA10c] Mohamed Messabihi, Pascal André et J. Christian Attiogbé, « Preuve de cohérence de composants Kmelia à l'aide de la méthode B », in : *4e Conférence francophone sur les Architectures Logicielles, CAL 2010, Pau, France, 9-11 mars 2010*, sous la dir. de Khalil Drira, t. L-5, Revue des Nouvelles Technologies de l'Information, Cépaduès-Éditions, 2010, p. 105-117, url : http://editions-rnti.fr/?inprocid=1000902.

[MB02] Stephen J. Mellor et Marc J. Balcer, *Executable UML : A Foundation for Model-Driven Architecture*, 1re éd., Object Technology Series, ISBN 0-201-74804-5, Addison-Wesley, 2002, p. 416, isbn : 0-201-74804-5.

[Med+02] Nenad Medvidovic et al., « Modeling software architectures in the Unified Modeling Language », in : *ACM Trans. Softw. Eng. Methodol.* 11.*1* (2002), p. 2-57, issn : 1049-331X, doi : http://doi.acm.org.gate6.inist.fr/10.1145/504087.504088.

[Mel+04] Stephen J. Mellor et al., *MDA Distilled*, 1re éd., Object Technology Series, ISBN 0-201-78891-8, Addison-Wesley, 2004, p. 176, isbn : 0-201-78891-8.

[Mes11] Mohamed El-Habib Messabihi, « Contribution à la spécification et à la vérification des logiciels à base de composants : enrichissement du langage de données de Kmelia et vérification de contrats », Thèse de doctorat dirigée par Attiogbé, Christian et André, Pascal Informatique Nantes 2011 - 2011NANT2017, thèse de doct., 2011, 1 vol. (173 f.) url : http://www.theses.fr/2011NANT2017/document.

[Mey03] B. Meyer, « The Grand Challenge of Trusted Components », in : *Proceedings of 25th International Conference on Software Engineering*, IEEE Computer Society, 2003, p. 660-667, url : citeseer.ist.psu.edu/meyer03grand.html.

[Mey85] Bertrand Meyer, « On Formalism in Specifications », in : *IEEE Software* 2.*1* (jan. 1985), p. 6-26.

[Mey89] Bertrand Meyer, « The New Culture of Software Development : Reflections on the Practice of OOD », in : *Proceedings of TOOLS'89*, sous la dir. de Jean Bezivin et Bertrand Meyer, Paris, nov. 1989.

[Mey92] Bertrand Meyer, *Introduction to the Theory of Programming Languages*, International Series in Computer Science, ISBN 0-13-498502-8, Prentice Hall, 1992, isbn : 0-13-498502-8.

[Mey97] Bertrand Meyer, *Object-oriented Software Construction*, 2e éd., International Series in Computer Science, ISBN 0-13-629155-4, Prentice Hall, 1997, isbn : 0-13-629155-4.

[Mez+23] Tsegay T. Mezgebe et al., « Intelligent manufacturing eco-system : A post COVID-19 recovery and growth opportunity for manufacturing industry in Sub-Saharan countries », in : *Scientific African* 19 (2023), e01547, ISSN : 2468-2276, URL : https://www.sciencedirect.com/science/article/pii/S2468227623000066.

[Mil05] Nikola Milanovic, « Contract-Based Web Service Composition Framework with Correctness Guarantees », in : *ISAS*, sous la dir. de Miroslaw Malek, Edgar Nett et Neeraj Suri, t. 3694, Lecture Notes in Computer Science, Springer, 2005, p. 52-67, ISBN : 3-540-29103-2.

[MJ13] Abid Mehmood et Dayang N. A. Jawawi, « Aspect-oriented model-driven code generation : A systematic mapping study », in : *Inf. Softw. Technol.* 55.*2* (2013), p. 395-411, DOI : 10.1016/j.infsof.2012.09.003.

[ML06] Manuel Mazzara et Ivan Lanese, « Towards a Unifying Theory for Web Services Composition », in : *WS-FM*, sous la dir. de Mario Bravetti, Manuel Núñez et Gianluigi Zavattaro, t. 4184, Lecture Notes in Computer Science, Springer, 2006, p. 257-272, ISBN : 3-540-38862-1.

[MLC15] Eduardo Cardoso Moraes, Herman Augusto Lepikson et Armando Walter Colombo, « Developing Interfaces Based on Services to the Cloud Manufacturing : Plug and Produce », in : *Industrial Engineering, Management Science and Applications 2015*, Lecture Notes in Electrical Engineering, Springer Berlin Heidelberg, 2015, 821–831, ISBN : 978-3-662-47199-9, DOI : 10.1007/978-3-662-47200-2_86, URL : http://link.springer.com/chapter/10.1007/978-3-662-47200-2_86.

[Mon08] Martin Monperrus, « La mesure des modèles par les modèles », Thèse de doctorat dirigée par Jézéquel, Jean-Marc Informatique Rennes 1 2008, thèse de doct., 2008, 1 vol. (143 p.) URL : http://www.theses.fr/2008REN1S085.

[Mon+16] L. Monostori et al., « Cyber-physical systems in manufacturing », in : *CIRP Annals* 65.*2* (2016), p. 621-641, ISSN : 0007-8506, DOI : https://doi.org/10.1016/j.cirp.2016.06.005, URL : https://www.sciencedirect.com/science/article/pii/S0007850616301974.

[Mor12] Chantal Morley, *Management d'un projet Système d'Information - Principes, techniques, mise en oeuvre et out : Principes, techniques, mise en oeuvre et outils*, 7e éd., Management des systèmes d'information, Dunod, 2012, ISBN : 9782100580460.

[Mor+12] Evan D. Morrison et al., « Strategic Alignment of Business Processes », in : *Service-Oriented Computing - ICSOC 2011 Workshops*, sous la dir. de George Pallis et al., Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 9-21, ISBN : 978-3-642-31875-7.

[Mot+19] Jean-Marie Mottu et al., « Shall We Test Service-Based Models or Generated Code ? », in : *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, sous la dir. de Loli Burgueño et al., IEEE, 2019, p. 493-502, doi : 10.1109/MODELS-C.2019.00078.

[MR09] Jim Marino et Michael Rowley, *Understanding SCA (Service Component Architecture)*, 1st, Addison-Wesley Professional, 2009, isbn : 0321515080, 9780321515087.

[MR93] José Morejon et Jean-René Rames, *Conduite de projets informatiques.* Principes et techniques s'appuyant sur la méthode MERISE, ISBN 2-7296-0457-X, InterEditions, 1993, isbn : 2-7296-0457-X.

[MT00] N. Medvidovic et R. N. Taylor, « A Classification and Comparison Framework for Software Architecture Description Languages », in : *IEEE Transactions on Software Engineering* 26.*1* (jan. 2000), p. 70-93.

[NE01] Dominique Nanci et Bernard Espinasse, *Ingénierie des systèmes d'information : Merise*, Vuibert informatique, Vuibert, 2001, isbn : 9782711786749.

[Ner92] Jean-Marc Nerson, « Applying Object-Oriented Analysis and Design », in : *Communications of the ACM* 35.*9* (sept. 1992), Special Issue on Analysis and Modelling in Software Development, p. 63-74.

[NTW04] Iftikhar Azim Niaz, Jiro Tanaka et Key Words, « Mapping Uml Statecharts To Java Code », in : *in Proc. IASTED International Conf. on Software Engineering (SE 2004*, 2004, p. 111-116.

[OMG18] OMG, *Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.4*, rapp. tech., Object Management Group, déc. 2018, url : https://www.omg.org/spec/FUML/1.4/.

[Pai+17] Hye-young Paik et al., « Service Component Architecture (SCA) », in : *Web Service Implementation and Composition Techniques* (2017), p. 203-250.

[Pap03] M. P. Papazoglou, « Service-Oriented Computing : Concepts, Characteristics and Directions », in : *WISE*, IEEE Computer Society, 2003, p. 3-12, isbn : 0-7695-1999-7, url : http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1254461.

[Pap08] Michael P. Papazoglou, *Web Services : Principles and Technology*, Harlow : Pearson/Prentice Hall, 2008, isbn : 978-0-321-15555-9.

[Par+22] Alexandre Parant et al., « Système Cyber-Physique de Production Modulaire », in : *Journées des Démonstrateurs en Automatique*, Angers, France, 2022, url : https://hal.science/hal-03766854.

[Par72] D. L. Parnas, « On the Criteria to Be Used in Decomposing Systems into Modules », in : *Commun. ACM* 15.*12* (déc. 1972), p. 1053-1058, ISSN : 0001-0782, DOI : 10.1145/361598.361623, URL : http://doi.acm.org/10.1145/361598.361623.

[Pav+05] Sebastian Pavel et al., « A Java Implementation of a Component Model with Explicit Symbolic Protocols », in : *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, t. 3628, Lecture Notes in Computer Science, Springer-Verlag, 2005, p. 115-125, ISBN : 3-540-28748-5.

[PD07] Romuald Pilitowski et Anna Dereziñska, « Code Generation and Execution Framework for UML 2.0 Classes and State Machines », in : *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, sous la dir. de Tarek Sobh, Dordrecht : Springer Netherlands, 2007, p. 421-427, ISBN : 978-1-4020-6268-1.

[Pel+21] Dominique Pelletier et al., « Spatial Processes and Management of Marine Populations », in : sous la dir. de G. H. Kruse et al., University of Alaska Sea Grant, AK-SG-00-04, 2021, chap. A conceptual model for evaluating the impact of spatial management measures on the dynamics of a mixed fishery, p. 53-66, URL : https://repository.library.noaa.gov/view/noaa/38513/noaa_38513_DS1.pdf.

[Pel+99] Dominique Pelletier et al., « Developing a computer-based simulation model to explore spatial and seasonal management measures in a mixed fishery », in : *19ème Lowell Wakefield Symposium*, Anchorage, Alaska, oct. 1999.

[Pep+14] Jonathan Pepin et al., « Alignement de modèles métiers et applicatifs : une approche pragmatique par transformations de modèle », in : *Actes de CIEL 2014, Troisième Conférence en Ingénierie du Logiciel*, -, 2014, URL : http://ciel2014.i3s.unice.fr/Ciel2014_fichiers/ActesCiel2014.pdf.

[Pep+15a] Jonathan Pepin et al., « A Method for Business-IT Alignment of Legacy Systems », in : *ICEIS 2015 - Proceedings of the 17th International Conference on Enterprise Information Systems, Volume 3, Barcelona, Spain, 27-30 April, 2015*, sous la dir. de Slimane Hammoudi, Leszek A. Maciaszek et Ernest Teniente, SciTePress, 2015, p. 229-237, DOI : 10.5220/0005351502290237.

[Pep+15b] Jonathan Pepin et al., « Alignement de modèles pour l'évolution de patrimoines applicatifs », in : *Journée de travail du groupe RIMEL du GdR GPL*, Lille, France, déc. 2015.

[Pep16] Jonathan Pepin, « Architecture d'entreprise : alignement des cartographies métiers et applicatives du système d'information », Theses, Université de Nantes (Unam), déc. 2016, URL : https://hal.science/tel-01763202.

[Pep+16a] Jonathan Pepin et al., « A Facet-based Model Mapping Method for EA Alignment and Evolution », in : *Proceedings of the CAiSE'16 Forum, at the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016), Ljubljana, Slovenia, June 13-17, 2016*, sous la dir. de Sergio España, Mirjana Ivanovic et Milos Savic, t. 1612, CEUR Workshop Proceedings, CEUR-WS.org, 2016, p. 153-160, URL : https://ceur-ws.org/Vol-1612/paper20.pdf.

[Pep+16b] Jonathan Pepin et al., « An Improved Model Facet Method to Support EA Alignment », in : *Complex Syst. Informatics Model. Q.* 9 (2016), p. 1-27, DOI : 10.7250/csimq.2016-9.01.

[Pep+18a] Jonathan Pepin et al., « Alignement des points de vue du système d'information, une approche pragmatique », in : *Actes du XXXVIème Congrès INFORSID, Nantes, France, May 28-31, 2018*, 2018, p. 125-140, URL : http://inforsid.fr/actes/2018/INFORSID2018Pepin.pdf.

[Pep+18b] Jonathan Pepin et al., « Definition and Visualization of Virtual Metamodel Extensions with a Facet Framework », in : *Model-Driven Engineering and Software Development - 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018, Revised Selected Papers*, sous la dir. de Slimane Hammoudi, Luís Ferreira Pires et Bran Selic, t. 991, Communications in Computer and Information Science, Springer, 2018, p. 106-133, DOI : 10.1007/978-3-030-11030-7_6.

[Pep+18c] Jonathan Pepin et al., « Virtual Extension of Meta-models with Facet Tools », in : *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018*, sous la dir. de Slimane Hammoudi, Luís Ferreira Pires et Bran Selic, SciTePress, 2018, p. 59-70, DOI : 10.5220/0006547100590070.

[PGA08] Vincent Pijpers, Jaap Gordijn et Hans Akkermans, « Business strategy-IT alignment in a multi-actor setting : A mobile e-service case », in : *Proceedings of the 10th International Conference on Electronic Commerce 2008*, sous la dir. de Dieter Fensel et Hannes Werthner, t. 342, ACM, 2008.

[PJIZ18] D. Preuveneers, W. Joosen et E. Ilie-Zudor, « Policy reconciliation for access control in dynamic cross-enterprise collaborations », in : *Enterprise Information Systems* 12.*3* (2018), p. 279-299, issn : 1751-7575, 1751-7583.

[PM07] Oscar Pastor et Juan Carlos Molina, *Model-driven architecture in practice - a software production environment based on conceptual modeling*, Springer, 2007, isbn : 978-3-540-71867-3, url : http://www.springerlink.com/content/978-3-540-71867-3.

[PM18] Klaus Pohl et Andreas Metzger, « Software Product Lines », in : *The Essence of Software Engineering*, sous la dir. de Volker Gruhn et Rüdiger Striemer, Springer, 2018, p. 185-201, doi : 10.1007/978-3-319-73897-0_11, url : https://doi.org/10.1007/978-3-319-73897-0_11.

[PMR16] Richard F. Paige, Nicholas Matragkas et Louis M. Rose, « Evolving models in Model-Driven Engineering : State-of-the-art and future challenges », in : *Journal of Systems and Software* (2016), issn : 0164-1212, url : //www.sciencedirect.com/science/article/pii/S0164121215001909.

[Pot+09] Patrick Pottier et al., *Capacités d'accueil et de développement des territoires littoraux, approches et méthodes*, jan. 2009, p. 92, url : https://www.researchgate.net/publication/281879310_Capacites_d'accueil_et_de_developpement_des_territoires_littoraux_approches_et_methodes.

[PP05] Dan Pilone et Neil Pitman, *UML 2.0 in a Nutshell (In a Nutshell (O'Reilly))*, O'Reilly Media, Inc., 2005, isbn : 0596007957.

[PP07] Pavel Parízek et František Plášil, « Modeling Environment for Component Model Checking from Hierarchical Architecture », in : *Third International Workshop on Formal Aspects of Component Software (FACS 2006)*, t. 182, Electronic Notes in Theoretical Computer Science, Elsevier B.V., 2007, p. 139-153.

[PP08] Isabelle Perseil et Laurent Pautet, « A Concrete Syntax for UML 2.1 Action Semantics Using +CAL », in : *Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, ICECCS '08, Washington, DC, USA : IEEE Computer Society, 2008, p. 217-221, isbn : 978-0-7695-3139-7, doi : 10.1109/ICECCS.2008.34, url : http://dx.doi.org/10.1109/ICECCS.2008.34.

[Pre10] Roger Pressman, *Software Engineering : A Practitioner's Approach*, 7e éd., New York, NY, USA : McGraw-Hill, Inc., 2010, isbn : 0073375977, 9780073375977.

[Pri12] Jacques PRINTZ, *Architecture logicielle - Concevoir des applications simples, sûres et adaptables*, 3e éd., Etudes et développement, Dunod, 2012, ISBN : 9782100583423.

[PV02] F. PLASIL et S. VISNOVSKY, *Behavior protocols for Software Components*, IEEE Transactions on SW Engineering, 28 (9), 2002., 2002, URL : citeseer.ist.psu.edu/plasil02behavior.html.

[Rai+04] Chris RAISTRICK et al., *Model Driven Architecture with Executable UML*, ISBN 0-521-53771-1, Cambridge University Press, 2004, ISBN : 0-521-53771-1.

[Rau+08] Andreas RAUSCH et al., éd., *The Common Component Modeling Example : Comparing Software Component Models*, t. 5153, LNCS, Heidelberg : Springer, 2008.

[RC11] Gérard ROUCAIROL et Yves CASEAU, *Urbanisation, SOA et BPM : Le point de vue du DSI*, 4e éd., InfoPro, Management des systèmes d'information, Dunod, 2011, ISBN : 9782100566365.

[RFZ17] C. RAIBULET, F. Arcelli FONTANA et M. ZANONI, « Model-Driven Reverse Engineering Approaches : A Systematic Literature Review », in : *IEEE Access* 5 (2017), p. 14516-14542, ISSN : 2169-3536.

[Rie13] L. RIERSON, *Developing Safety-Critical Software : A Practical Guide for Aviation Software and DO-178C Compliance*, Taylor & Francis, 2013, ISBN : 9781439813683.

[RJB98] James RUMBAUGH, Ivar JACOBSON et Grady BOOCH, *The Unified Modeling Language User Guide*, Object-Oriented Series, ISBN 0-201-57168-4, Addison-Wesley, 1998, p. 512, ISBN : 0-201-57168-4.

[RJB99] James RUMBAUGH, Ivar JACOBSON et Grady BOOCH, *The Unified Software Development Process*, Object-Oriented Series, ISBN 0-201-57169-2, Addison-Wesley, 1999, p. 516, ISBN : 0-201-57169-2.

[Rol86] Colette ROLLAND, « Introduction à la conception des systèmes d'information et panorama des méthodes disponibles », in : *Revue Génie Logiciel* 4 (juin 1986), p. 7-62.

[RP13] Ben ROELENS et Geert POELS, « Towards an Integrative Component Framework for Business Models : Identifying the Common Elements Between the Current Business Model Views », in : *Proceedings of the CAiSE'13 Forum*, t. 998, CEUR Workshop Proceedings, Valencia, Spain : CEUR-WS.org, 2013, p. 114-121.

[RS04] S. RUGABER et K. STIREWALT, « Model-driven reverse engineering », in : *IEEE Software* 21.*4* (juill. 2004), p. 45-53, ISSN : 0740-7459, DOI : 10.1109/MS.2004.23.

[RSP17] Ben Roelens, Wout Steenacker et Geert Poels, « Realizing strategic fit within the business architecture : the design of a Process-Goal Alignment modeling and analysis technique », in : *Software & Systems Modeling* (jan. 2017), issn : 1619-1374.

[Rum+96] James Rumbaugh et al., *Modélisation et conception orientés objet*, t. 1, ISBN 2-225-84684-7, Edition française revue et augmentée, Masson, 1996, isbn : 2-225-84684-7.

[RV11] P. Roques et F. Vallée, *UML 2 en action : De l'analyse des besoins à la conception*, Architecte logiciel, Eyrolles, 2011, isbn : 9782212082241.

[Sal03] Gwen Salaün, « Contributions à l'intégration de langages pour la spécification formelle et la vérification de systèmes complexes », Thèse de doctorat dirigée par Oussalah, Mourad-Chabane Informatique Nantes 2003, thèse de doct., 2003, 194 p. url : http://www.theses.fr/2003NANT2023.

[SB09] Gwen Salaün et Tevfik Bultan, « Realizability of Choreographies Using Process Algebra Encodings », in : *IFM*, sous la dir. de Michael Leuschel et Heike Wehrheim, t. 5423, Lecture Notes in Computer Science, Springer, 2009, p. 167-182, isbn : 978-3-642-00254-0.

[Sch07] Satu Elisa Schaeffer, « Graph clustering », in : *Computer science review* 1.*1* (2007), p. 27-64.

[Sco+19] Gian Luca Scoccia et al., « Permission Issues in Open-Source Android Apps : An Exploratory Study », in : *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, p. 238-249, doi : 10.1109/SCAM.2019.00034.

[Sel07] Bran Selic, « From Model-Driven Development to Model-Driven Engineering », in : *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ECRTS '07, USA : IEEE Computer Society, 2007, p. 3, isbn : 0769529143, doi : 10.1109/ECRTS.2007.16, url : https://doi.org/10.1109/ECRTS.2007.16.

[Sel08] Bran Selic, « Personal reflections on automation, programming culture, and model-based software engineering », in : *Automated Software Engineering* 15.*3* (déc. 2008), p. 379-391, issn : 1573-7535, doi : 10.1007/s10515-008-0035-7.

[SG96] M. Shaw et D. Garlan, *Software Architecture : Perspective on an Emerging Discipline*, Prentice Hall, 1996.

[Sim09] Jacques Simonin, « Conception de l'architecture d'un système dirigée par un modèle d'urbanisme fonctionnel », thèse de doct., Université de Rennes 1, jan. 2009.

[SK98] Martin Schader et Axel Korthaus, « Modeling Java Threads in UML », in : *The Unified Modeling Language – Technical Aspects and Applications*, sous la dir. de Martin Schader et Axel Korthaus, Physica-Verlag, Heidelberg, 1998, p. 122-143.

[Som11] Ian Sommerville, *Software Engineering*, 9 th, Addison-Wesley, oct. 2011.

[SSS08] Forrest Shull, Janice Singer et Dag I. K. Sjøberg, éd., *Guide to Advanced Empirical Software Engineering*, Springer, 2008, isbn : 9781848000438, doi : 10.1007/978-1-84800-044-5, url : https://doi.org/10.1007/978-1-84800-044-5.

[STW03] Graeme Shanks, Elizabeth Tansley et Ron Weber, « Using ontology to validate conceptual models », in : *Commun. ACM* 46.*10* (oct. 2003), p. 85-89, issn : 0001-0782, doi : 10.1145/944217.944244.

[SWS07] Anoop Singhal, Theodore Winograd et Karen A. Scarfone, *SP 800-95. Guide to Secure Web Services*, rapp. tech., Gaithersburg, MD, United States : National Institute of Standards & Technology, 2007.

[Szy02] Clemens Szyperski, *Component Software : Beyond Object-Oriented Programming*, ISBN 0-201-74572-0, Addison Wesley Publishing Company/ACM Press, 2002, isbn : 0-201-74572-0.

[TAC18] Mohammed El Amin Tebib, Pascal André et Olivier Cardin, « A Model Driven Approach for Automated Generation of Service-Oriented Holonic Manufacturing Systems », in : *Service Orientation in Holonic and Multi-Agent Manufacturing - Proceedings of SOHOMA 2018, Bergamo, Italy, June 11-12, 2018*, sous la dir. de Theodor Borangiu et al., t. 803, Studies in Computational Intelligence, Springer, 2018, p. 183-196, doi : 10.1007/978-3-030-03003-2_14.

[Teb+21a] Mohammed El Amin Tebib et al., « Assisting Developers in Preventing Permissions Related Security Issues in Android Applications », in : *Dependable Computing - EDCC 2021 Workshops - DREAMS, DSOGRI, SERENE 2021, Munich, Germany, September 13, 2021, Proceedings*, sous la dir. de Rasmus Adler et al., t. 1462, Communications in Computer and Information Science, Springer, 2021, p. 132-143, doi : 10.1007/978-3-030-86507-8_13.

[Teb+21b] Mohammed El Amin Tebib et al., *Preventing Permissions Security Issues in Android : a Developer's Perspective*, Journée thématique du GT SSLR 2021 sur la sécurité des réseaux, Poster, mai 2021, url : https://hal.science/hal-03259639.

[Teb+22] Mohammed El Amin Tebib et al., « IDE Plugins for Secure Android Applications Development : Analysis & Classification Study », in : *SECURWARE 2022 : The Sixteenth International Conference on Emerging Security Information, Systems and Technologies*, Lisbonne, Portugal, oct. 2022, URL : https://hal.science/hal-03865020.

[Teb+23a] Mohammed El Amin Tebib et al., « A Survey on Secure Android Apps Development Life-Cycle : Vulnerabilities and Tools », in : *International Journal On Advances in Security* 16.*1 & 2* (2023), p. 54-71, URL : https://hal.science/hal-04181107.

[Teb+23b] Mohammed El Amin Tebib et al., « PrivDroid : Android Security Code Smells Tool for Privilege Escalation Prevention », in : *IIEEE CyberSciTech/ DASC/ PICom/ CDBCom 2023, 14-17 Nov*, Abu Dhabi, UAE, nov. 2023, p. xx, URL : https://icnetlab.org/cyber-science2023/dasc/.

[TO23] Chris Turner et John Oyekan, « Manufacturing in the Age of Human-Centric and Sustainable Industry 5.0 : Application to Holonic, Flexible, Reconfigurable and Smart Manufacturing Systems », in : *Sustainability* 15.*13* (2023), ISSN : 2071-1050, DOI : 10.3390/su151310169, URL : https://www.mdpi.com/2071-1050/15/13/10169.

[TP13] Damien Trentesaux et Vittaldas V. Prabhu, « Introduction to Shop-Floor Control », in : *Wiley Encyclopedia of Operations Research and Management Science*, John Wiley & Sons, oct. 2013, p. 1-9, DOI : 10.1002/9780470400531.eorms1082, URL : https://uphf.hal.science/hal-03630945.

[TRC91] Hubert Tardieu, Arnold Rochfeld et René Coletti, *La méthode Merise, Tome 1 : Principes et outils*, ISBN 2-7081-1106-X, voir aussi les tomes 2 et 3, Editions d'Organisation, 1991, ISBN : 2-7081-1106-X.

[TS07] Laure-Hélène Thevenet et Camille Salinesi, « Aligning IS to Organization's Strategy : The InStAl Method », in : *Advanced Information Systems Engineering, 19th International Conference, CAiSE 2007, Trondheim, Norway, June 11-15, 2007, Proceedings*, 2007, p. 203-217.

[UE10] Club URBA-EA, *Urbanisme des SI et gouvernance : Bonnes pratiques de l'architecture d'entreprise*, 1re éd., InfoPro. Management des systèmes d'information, Dunod, 2010, ISBN : 9782100553075.

[UL11] Azmat Ullah et Richard Lai, « Modeling Business Goal for Business/it Alignment Using Requirements Engineering », in : *Journal of Computer Information Systems* 51.*3* (2011), p. 21-28.

[UL13] Azmat Ullah et Richard Lai, « A Systematic Review of Business and Information Technology Alignment », in : *ACM Trans. Manage. Inf. Syst.* 4.*1* (avr. 2013), 4 :1–4 :30, issn : 2158-656X, (visité le 07/02/2014).

[Unh05] Bhuvan Unhelkar, *Verification and Validation for Quality of UML 2.0 Models*, Wiley Series in Systems Engineering and Management, John Wiley & Sons, 2005, isbn : 9780471727835.

[UPL11] Mark Utting, Alexander Pretschner et Bruno Legeard, « A taxonomy of model-based testing approaches », in : *Software Testing, Verification and Reliability* 22.*5* (avr. 2011), p. 297-312, doi : 10.1002/stvr.456, url : https://doi.org/10.1002/stvr.456.

[Van+98] Hendrik Van Brussel et al., « Reference architecture for holonic manufacturing systems : PROSA », in : *Computers in Industry* 37.*3* (1998), p. 255-274, issn : 0166-3615, url : https://www.sciencedirect.com/science/article/pii/S016636159800102X.

[Wei08] Tim Weilkiens, *Systems Engineering with SysML/UML : Modeling, Analysis, Design*, The MK/OMG Press, Elsevier Science, 2008, isbn : 9780123742742.

[Whi11] Jon Whittle, « What do 449 MDE Practitioners Think About MDE ? », in : *EESSMod*, sous la dir. de Michel R. V. Chaudron et al., t. 785, CEUR Workshop Proceedings, CEUR-WS.org, 2011.

[Wir93] Martin Wirsing, « Développement de logiciel et spécification formelle », in : *Technique et Science Informatique* 12.*4* (1993), p. 413-431.

[WS-F] *Web Services Federation Language (WS-Federation) Version 1.2. OASIS Standard*, OASIS, mai 2009, url : https://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html.

[XAC3] *eXtensible Access Control Markup Language (XACML) Version 3.0 - OASIS Standard*, OASIS, jan. 2013, url : https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[YZS22] Shuaihao Yang, Zigang Zeng et Wei Song, « PermDroid : automatically testing permission-related behaviour of Android applications », in : *ISSTA '22 : 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, sous la dir. de Sukyoung Ryu et Yannis Smaragdakis, ACM, 2022, p. 593-604, doi : 10.1145/3533767.3534221.

[ZYW12] ShaoMin Zhang, HongBian Yang et BaoYi Wang, « Realization Distributed Access Control Based on Ontology and Attribute with OWL », in : *Advances in Electronic Engineering, Communication and Management Vol.1*, t. 139, Springer Berlin Heidelberg, 2012, p. 583-588, isbn : 978-3-642-27286-8, doi : 10.1007/978-3-642-27287-5_94.

Annexes

Annexe A

Publications

Liste de publications depuis 2006

In English

ISD 2024 (August 2024) - ISD chapter 2023 (to appear Springer LNISO July 2024) Inforsid 2024 [Ben+24a] - ICEIS 2024 [Ben+24b] - CSIMQ 2024 [AT24]
IWSF & SHIFT Workshop (ISSRE 2023) [Bah+23] - Sohoma 2023 [AC24 ; AG24] ISD 2023 [And+23] - International Journal On Advances in Security, 2023 no 1&2 [Teb+23a] Computers in Industry Vol. 149 [Kai+23]
Sohoma 2022 [Der+23] - Secureware 2022 [Teb+22]
Sohoma 2021 [AC22] - SERENE 2021 [Teb+21a] - MEDI 2021 [AT21]
ARIMA Journal Vol. 34 [Bah+20b] - Sohoma 2020 Springer [ACA20] - CARI 2020 [Bah+20a] Modelsward 2020 [AT20] - ComSIS Journal Volume 17, Issue 1 [AAL20]
Modevva 2019 IEEE [Mot+19] - ISSA 2019 Springer [Bah+19] - Sohoma 2019 Springer [AAC19] Modelsward 2019 [And19]
MEDI/Remedy 2018 Springer [AAL18b] - CCIS Journal Springer [Pep+18b] Sohoma 2018 Springer [TAC18] - Modelsward 2018 [Pep+18c] - arXiv-2018 [Pep+18c]
Sohoma 2017 Springer [AC17] - Amaretto 2017 [AAM17]
CSIMQ Journal Vol 9 [Pep+16b] - ISTTA 2016 Demo [AMS16] + poster CAISE 2016 Forum [Pep+16a] + poster
ICEIS 2015 [Pep+15a]
Modevva 2013 [AMA13]
ENTCS 2010 [AAM10] - Fesca 2010 [And+10a] - ABZ 2010 [And+10b] - WCSI 2010 [MAA10b]
FACS 2009 [And+09] - FACS 2009 [And+09] - WCRE 2009 [Anq+09]
SC 2008 [AAA08]
SC 2007 [AAA07a] - ProveCS 2007 [AAA07c]
SC 2006 [AAA06d] - WCAT 2006 [AAA06a]

En Français

MSR 2021 actes (fr) [And+21]
MSR 2019 [ALB19]
Inforsid 2018 [Pep+18a]
AFADL 2017 [And+17]
PUV 2016 [CPA16]
RIMEL 2015 [Pep+15b]
SEH 2014 [CPA14] - CIEL 2014 [Pep+14]
CIEL 2013 [AAM13]
TSI 2011 [AAA11] - IDM 2011 [AAM11]
CAL 2010 [MAA10c]
CAL 2009 [AAM09]
LMO 2007 [AAA07b]
CAL 2006 [AAA06b] - Mosim 2006 [AAA06c]

Quelques présentations récentes

Abdis 2022 (en)	https://adbis2022.polito.it/doctoral-consortium/
http://pagesperso.ls2n.fr/~andre-p/download/ADBIS_DC_2022_Ali_Benjilany.pdf	
Ressi 2022	https://ressi2022.sciencesconf.org/program
http://pagesperso.ls2n.fr/~andre-p/download/ressi22.pdf	
Inforsid 2022 (forum)	http://inforsid.fr/forum_jcjc.php
http://pagesperso.ls2n.fr/~andre-p/download/Inforsid_2022_Forum_JCJC_Benjilany_Ali.pdf	
GDR GPL 2021	Poster
http://pagesperso.ls2n.fr/~andre-p/download/GDRGPL2021Posters_paper_2.pdf	
Journée GT IDM 2021	Article
http://pagesperso.ls2n.fr/~andre-p/download/.pdf	
GT SSLR 2021	[Teb+21b], article, poster
http://pagesperso.ls2n.fr/~andre-p/download/PosterDoctoral2021-Mohammed.pdf	
http://pagesperso.ls2n.fr/~andre-p/download/gt2021.pdf	
http://pagesperso.ls2n.fr/~andre-p/download/PosterDoctoral2021-Mohammed.pdf	

Livres et ouvrages

1. Développement de logiciel avec UML2 et OCL Cours et exercices corrigés [AV13]
2. Conception de Systèmes d'Information Volume 5 [AV03a]
3. Conception de Systèmes d'Information Volume 4 [AV04]
4. Conception de Systèmes d'Information Volume 3 [AV02]
5. Conception de Systèmes d'Information Volume 2 [AV01b]
6. Conception de Systèmes d'Information Volume 1 [AV01a]
7. Développement de logiciel avec UML [AV03c]
8. Etudes de cas : développement de logiciel avec UML [AV03b]
9. Ingénierie objet [AR96]

Annexe B

Le développement du logiciel

Cette annexe, reprend en partie le chapitre 1 de [AV01a] et le complète. Elle présente les grandes lignes du développement du logiciel et du génie logiciel.

B.1 Les méthodes de développement du logiciel

Le développement du logiciel est une suite d'étapes qui permet d'obtenir un ensemble de programmes (le logiciel) qui automatise un système d'information. Le développement du logiciel est une partie de ce qu'on appelle communément la gestion de projet. L'activité d'une entreprise, au sens large, concerne un ensemble de projets. Un projet est la réponse à un besoin de l'entreprise, il est caractérisé par l'expression du besoin, la définition d'une ou plusieurs solutions techniques et la gestion des ressources mise en œuvre pour réaliser le projet. La gestion de projet au sens large (faisabilité, moyens humains et financiers, planification, impact sur l'organisation automatisée, etc.) sort du cadre de notre contexte. Le lecteur trouvera avec profit un tour d'horizon de la gestion de projet et du développement du logiciel dans le génie logiciel [Som11 ; Pre10].

B.1.1 Système d'information

Plusieurs définitions de la notion de système d'information existent. Voici une synthèse extraite de [Rol86] :

> "un **système d'information** (SI en abrégé) est un artefact, un objet artificiel, greffé sur un objet naturel qui peut être une organisation, un processus industriel, une commande embarquée. Il est conçu pour mémoriser un ensemble d'images de l'objet réel à différents moments de sa vie ; ces images doivent être accessibles par les partenaires de l'organisation qui s'en servent pour décider des actions à entreprendre dans les meilleures conditions."

Un SI est en quelque sorte une extension de la mémoire humaine qui amplifie le pouvoir de mémorisation des acteurs de l'organisation et facilite leur prise de décision.

Sous cette dénomination se cache en fait une grande variété de systèmes informatiques. Nous distinguons les catégories d'applications suivantes :

— Les systèmes d'information en gestion sont caractérisés par un stockage important d'informations et de nombreux traitements en consultation, mise à jour de ces informations *e.g.* gestion de la clientèle, du stock, traitement de la langue. Les systèmes

actuels intègrent une composante réseau (accès distant) qui intervient plus dans l'implantation du système que dans sa modélisation.

— Le calcul sciencifique se caractérise par de nombreux calculs *e.g.* simulation, météo, imagerie. Une base d'information peut servir à stocker les informations de base ou les résultats mais intervient assez peu dans le calcul lui-même. La difficulté principale est la description de modèles mathématiques par des modèles informatiques. Un critère important est la rapidité du traitement.

— L'informatique temps réel se caractérise par une réactivité très forte du système d'information *e.g.* pilotage automatique d'avion, surveillance de centrale nucléaires. Nous rangeons, un peu arbitrairement, sous cette dénomination l'informatique embarquée (automatismes), le contrôle de processus (fabrication, surveillance), les applications où le temps joue un rôle majeur, les réseaux informatiques. L'interface entre le système d'information et son objet naturel (le processus) prend ici une grande importance. Elle inclue des aspects matériels non négligeables : capteurs pour récupérer les informations, des actionneurs pour piloter objet naturel, bus de communications, etc. [Cal90] propose une caractérisation des systèmes et un classement des systèmes sur les disciplines connexes à l'informatique industrielle.

Nous estimons que ces trois catégories représentent, assez grossièrement, les différents types de systèmes d'information que nous sommes susceptibles de modéliser ici.

B.1.2 Modélisation

Selon [Cal90], un **modèle** est une interprétation explicite par son utilisateur de la compréhension d'une situation, ou plus exactement de l'idée qu'il se fait d'une situation. Il peut être exprimé par des mathématiques, des symboles, des mots, mais essentiellement, c'est une description d'entités et de relations entre elles. Le terme **représentation** est aussi utilisé. Ainsi, modéliser revient à élaborer une vue partielle, plus ou moins abstraite de l'existant. Un modèle est correct s'il permet de répondre aux questions qu'on se pose. Un modèle est **opérationnel** s'il peut être exécuté par une machine.

La recherche d'une bonne représentation lors de la résolution de problème est presque toujours une étape indispensable vers la solution [Lau86]. Il faut donc savoir changer de représentation pour mieux modéliser le problème. La *modélisation* en informatique est le passage du domaine du problème à celui de sa solution informatique. Ses difficultés sont : appréhender la sémantique du monde réel et la tranformer en signes manipulables par les ordinateurs.

Le développement est une suite de modèles, généralement donnés dans une représentation différente, se rapprochant petit à petit des concepts de la programmation. Pour rationaliser ce développement, on utilise des méthodes.

B.1.3 Méthode de développement

Une méthode est une technique de résolution de problèmes [Lau86]. Le terme de **méthode** recouvre plusieurs notions. Dans la littérature, le terme **méthodologie** est souvent, et à tort, synonyme de méthode. Une méthode est à la fois une philosophie dans l'approche des problèmes, une démarche ou un fil conducteur dans la résolution, des outils d'aide et enfin un formalisme ou des normes.

Par exemple, dans la méthode Merise [TRC91] la philosophie est fortement inspirée des bases de données relationnelles avec une dualité données/traitements, de la programmation structurée et des réseaux de PETRI. Nous aurons l'occasion d'y revenir tout au long de cet ouvrage. Merise propose une double démarche : par niveau d'abstraction (conceptuel, organisationnel, physique ou opérationnel) et par étapes (études préalables, détaillées, fonctionnelles, techniques et mise en production). Le terme abstraction est synonyme d'éloignement vis-à-vis des considérations matérielles et logicielles (*cf.* Section 1.2.5) page 31. Les étapes servent à baliser le développement et exiger des résultats intermédiaires. Elles sont étudiées dans la section B.3.2. Les formalismes Entité/Association, réseaux de PETRI sont utilisés. Des outils comme les analyseurs, générateurs, imprimeurs peuvent être mis en place.

Classification

Une première classification, largement reconnue, est celle de ROLLAND [Rol86]. Elle distingue deux grandes classes de méthodes de la conception : les méthodes **cartésiennes** et les méthodes **systémiques**.

Les méthodes cartésiennes suivent le principe "*Diviser pour règner*". Pour étudier un phénomène, on le divise en éléments simples, on étudie séparément les éléments puis on réunit à nouveau les éléments. Par exemple, le processus de conception sera décomposé en phases, elles-mêmes découpées en étapes et ainsi de suite. Au niveau le plus bas, on a une liste de tâches à accomplir, de documents résultats à produire. Le processus est *linéaire*. Les méthodes cartésiennes associent en général la démarche par étapes à une approche fonctionnelle du système d'information, c'est-à-dire aux fonctions qu'il doit assurer (informations à produire). On passe de l'énoncé des fonctions à l'expression de la solution technique. Dans l'approche fonctionnelle, le système d'information est considéré comme un processus de traitement de l'information (boîte noire) qui mémorise, traite, met en forme et communique les informations. La réalisation du processus se fait par une démarche fonctionnelle descendante, inspirée de la programmation structurée.

Pour les méthodes systémiques, la démarche n'est pas essentielle, ce qui prime, c'est la compréhension du système d'information en tant que système s'insérant dans l'ensemble des autres systèmes de l'organisation. Le système d'information est couplé à la fois au système opérationnel de l'organisation dont il est une représentation et au système de décision dont il est le support. On met l'accent sur l'aspect global, la mise en évidence d'éléments et de relations entre ces éléments. L'approche est conceptuelle et met en valeur

différents niveaux d'abstraction par des modèles formels riches sémantiquement.

Certaines méthodes s'appliquent au cycle complet de développement, d'autres prennent en charge certaines étapes. Nous allons voir dans les sections qui suivent les différents aspects d'une méthode, à savoir la philosophie, la démarche, les outils et le formalisme. Notons pour terminer que la méthode n'est qu'un outil de gestion de projet et que le développement dépend non seulement de l'application à traiter mais aussi de critères de gestion : on doit planifier, constituer des équipes, organiser, contrôler et diriger. Des théories ont été développées sur ce sujet [BR85].

Dans la suite de ce chapitre, nous nous focaliserons principalement sur deux aspects, les **modèles de représentation** (philosophie et formalisme) et les **processus de développement** (démarche), pour analyser et comparer les méthodes de développement.

B.2 Les modèles de représentation

Par abus de langage, dans ce qui suit le terme **modèle** désignera à la fois le résultat de la modélisation, la théorie sous-jacente et la notation utilisée. Voici quelques modèles utilisés dans le développement du logiciel.

Table B.1 – Approches théoriques du développement de SI

Théorie des systèmes	Analyse modulaire des systèmes	MELESE LE MOIGNE
Théorie des ensembles	LCP/LCS Programmation structurée	WARNIER DIJKSTRA
Concepts de base de données	Objet/Relation Relationnel Codasyl	CHEN, TARDIEU CODD BACHMANN
Processus	Réseaux	PETRI
Evénements - Procédures	Corig	CGI, Pac, Ariane...
Résultats - Données	Minos	Sema, Atlas, Sligos...
Objet	OOD, OOA	MEYER, BOOCH, OMT, UML

Dans un système d'information, trois aspects sont souvent mis en évidence : ce que le système manipule (les informations, les données), quand il les manipule (causalité, contrôle, comportement dynamique, événements) et comment il les manipule (les opérations, les fonctions, le comportement fonctionnel). Cette triade acceptée par tous les concepteurs de systèmes d'information est décrit dans un espace à trois dimensions : celle des informations, celle des fonctions et celle des comportements (dynamiques).

La dimension des informations correspond aux données manipulées, mémorisées par le système. Les fonctions du système représentent ce qu'il fait, l'ordre dans lequel celles-ci sont exécutées étant son comportement. Etablissons un parallèle avec un programme : les informations sont les variables, les opérandes. Les fonctions sont les opérateurs et l'algorithme le comportement.

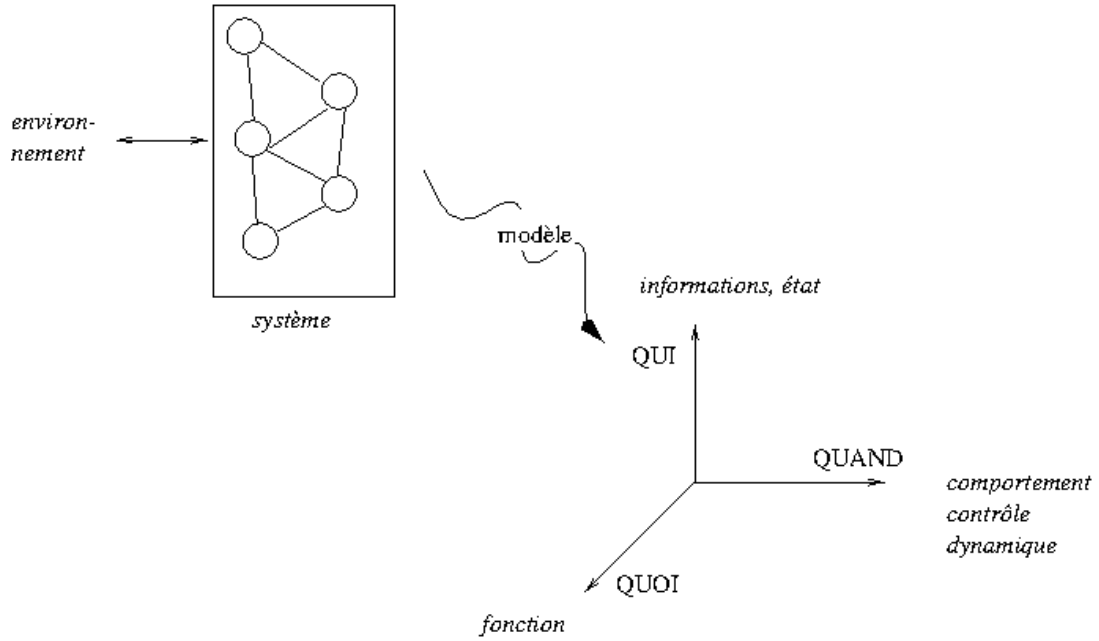


FIGURE B.1 – Modèles d'un système d'information en trois dimensions

Les méthodes de développement sont parfois classées selon qu'elles privilégient l'un ou l'autre des aspects. Nous illustrons ces tendances dans la TABLE B.2.

TABLE B.2 – Décomposition d'un système d'information

statique (données)		(opérations)		**dynamique** (traitements)
←				→
Entité/ Association	Structures de Données	Langages Formels	Flots de Données	Réseaux de PETRI Automates

Trois courants majeurs ont dominé les méthodes d'analyse et de conception :

• l'approche fonctionnelle dans laquelle le système est perçu en termes de fonctions et sous-fonctions munies d'une interface (e.g. JSD)

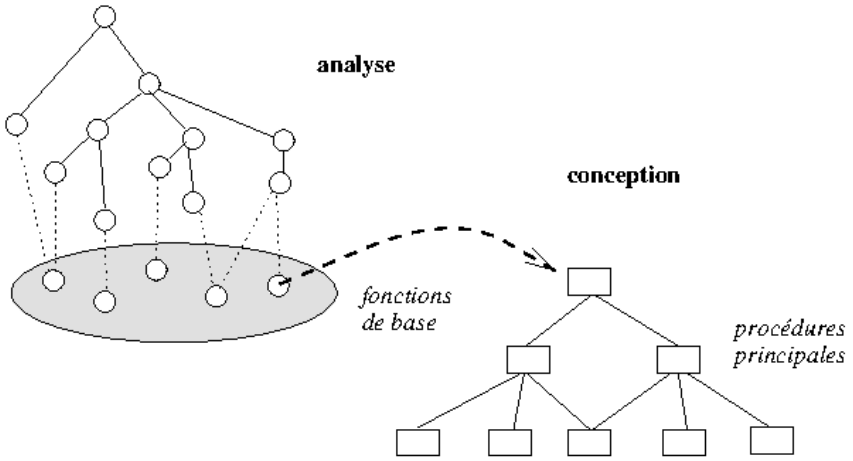


FIGURE B.2 – Décomposition fonctionnelle

• l'approche flots de données dans laquelle on exprime la transformation des données (e.g.

SA (DE MARCO), SADT (ROSS), SART (WARD et MELLOR/HATLEY-PIRBHAI), ESML (BOEING))

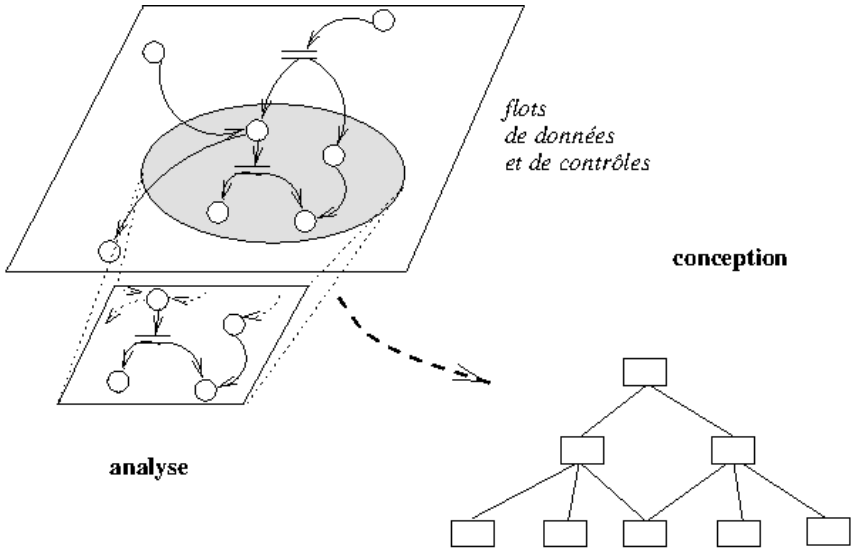


FIGURE B.3 – Décomposition flot de donnée

• l'approche modèle de donnée pour laquelle l'accent est mis sur la partie statique du système d'information (e.g. Entité-Relation (CHEN), NIAM (VERHAIJEN))

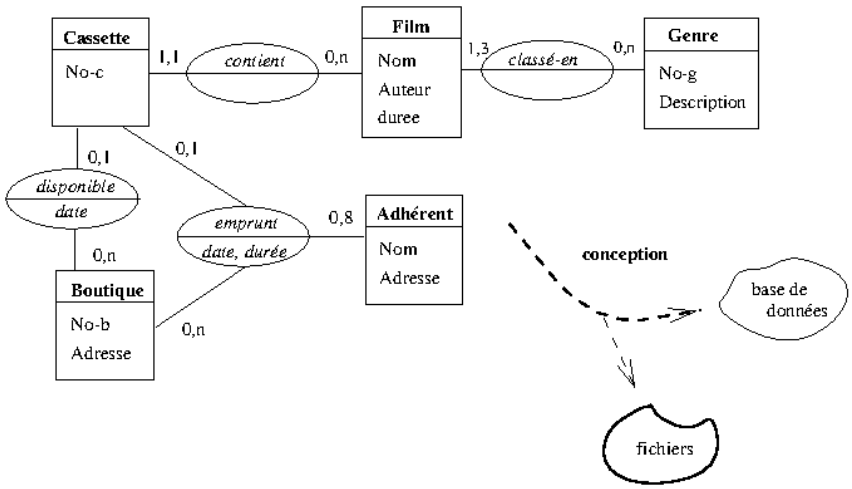


FIGURE B.4 – Décomposition modèle de donnée

De manière générale, nous pouvons affirmer que les théories utilisées pour chaque axe sont cohérentes et permettent de démontrer des propriétés propres à ces axes. Par contre, les propriétés dépendant de plusieurs axes sont bien plus difficiles à démontrer. En pratique, ces axes sont rarement independants dans une application informatique. Par exemple, une fonction accédant à une donnée peut dépendre de l'état dans lequel se trouve le système. Ces aspects de la conception de systèmes fait l'objet de recherches importantes sous le titre d'**intégration de méthodes**. L'approche à objets (voir le tome 2 de cet ouvrage) est un mélange des différents courants. Elle est implicitement une des approches de l'intégration de méthodes.

B.3 Les processus de développement

De même que les systèmes d'information ont un cycle de vie propre (gestation, conception, exploitation, maintenance, mort), le développement du logiciel suit un processus.

B.3.1 Contexte d'un développement

Le cycle de développement d'un projet s'inscrit dans le cadre plus vaste des activités d'une organisation [Cal90]. Cette activité concerne un ensemble de projets (non disjoints). Un projet est caractérisé par des objectifs et un déroulement. Les phases d'un projet sont :

- l'**expression du besoin**. On exprime les objectifs et les principales contraintes.
- la **définition du projet**. Une étude de faisabilité permet d'évaluer les objectifs et contraintes en fonctions d'hypothèses. Ensuite, on sélectionne une approche globale en considérant tous les aspects du développement sans réaliser le projet.
- l'**organisation**. Lorsque le projet est accepté (objectifs, coûts et délais sont fixés). On planifie le développement du projet.
- le **développement**. Cette phase est étudiée précisément dans la section B.3.2.
- l'**achèvement**. On vérifie la conformité du projet par rapport aux objectifs (tout au long du développement).

B.3.2 Etapes du développement

Le processus de développement d'un système d'information couvre l'intégralité de son cycle de vie. [Rol86] et [Cal90] proposent cinq étapes (même si des variantes existent, elle sont surtout des différences de terminologie) : analyse des besoins, analyse conceptuelle, conception technique, réalisation, maintenance. Dans les méthodes à objets, le développement du logiciel est souvent découpé en quatre activités *bien séparées* : analyse des besoins, analyse et conception, conception détaillée et programmation, tests et intégration. Nous retenons la décomposition suivante :

1. l'**analyse des besoins** définit les services du système, ses contraintes et ses buts en consultant les utilisateurs du système. Une étude d'opportunité peut être menée pour savoir si le système est réalisable et donner une approximation de la rentabilité de ce système.
   Synonymes : analyse préalable, *user requirements analysis*.
2. l'**analyse** est la construction d'un modèle (une spécification) du système à partir de l'analyse des besoins. A partir de ce modèle on peut proposer plusieurs scénarii et réaliser une étude de faisabilité.
   Synonymes : spécification des besoins, analyse préalable, étape conceptuelle, analyse conceptuelle, conception préliminaire, modélisation conceptuelle, *analysis, user requirement specification, requirements engineering*.
3. la **conception** est une proposition de solution au problème spécifié dans l'analyse. Elle définit la solution retenue par prise en compte des caractéristiques logiques

d'usage du futur système d'information et des moyens de réalisation, humains, techniques et organisationnels. Elle définit les principes pour les préoccupations de conception (persistance, concurrence, IHM, distribution). On distingue souvent la conception système et la conception détaillée. La première a pour objectif de donner l'architecture globale du systèmes (i.e. les différentes parties) et la seconde décrit chaque partie du système. Cette spécification du logiciel reste indépendante de tout moyen de réalisation.

Synonymes : étape fonctionnelle, analyse fonctionnelle, analyse organique, étape logique, conception technique, *global design, system design, detailed design, design engineering*.

4. la **réalisation** et le **test** produisent la solution exécutable en termes de programmes. Pour les logiciels complexes, on distingue l'implantation des différentes parties et leur validation par des tests unitaires, et l'implantation du système complet par **intégration** des parties et tests systèmes. On vérifie ainsi que les spécifications des besoins sont satisfaites.

   Synonymes : codage, implantation, mise en œuvre, programmation, *coding, implementation, program testing*.
5. l'**installation** du logiciel règle les problèmes de mise en place dans l'organisation.
6. la **maintenance** adapte la solution conceptuelle aux changements organisationnels et aux évolutions technologiques (évolutive). Elle corrige aussi les erreurs accumulées dans les phases précédentes (curative). Le processus de maintenance est un processus itératif dont l'activité répétée est un cycle de développement complet (de la spécification à la réalisation).

Cette formation suit les grandes lignes d'une démarche de développement, celle décrite par la norme DOD-STD-2167A, selon les auteurs.

B.3.3 Cycles de développement

Plusieurs cycles ont été proposés pour organiser ces tâches. Trois grandes catégories sont retenues :

— les modèles linéaires (modèle de la "cascade", modèle en "V") dans lesquels les différentes étapes ci-dessus sont réalisées tour à tour. On commence la suivante une fois la précédente achevée. En cas d'erreur, on revient sur l'étape précédente. On parle aussi d'approche descendante. Dans le modèle en "V", on insiste sur une séparation entre la construction des diverses spécification et leur validation a posteriori (tests unitaires, tests d'intégration, etc) ainsi que sur le niveau d'abstraction (utilisateur/architecture/implantation). Ce sont les modèles de référence [BR85 ; Som11 ; Pre10].

— les modèles contractuels sont une suite de contrats entre client et fournisseurs. Les méthodes formelles en sont un bon exemple, elles permettent la transformation de spécifications.

— les modèles itératifs ou à spirale [Boe88] permettent un développement incrémental notamment par le prototypage. Ces modèles intègrent des notions de risques à évaluer, de spécifications partielles et de résultats intermédiaires. Le prototypage devient une technique de modélisation. Ce modèle est évolutif par nature mais rend difficile la planification et il doit éviter les redondances.

Les modèles mixtes comme le modèle “X” [Hod91] s’inspirent de plusieurs styles. Les modèles linéaires ne dépendent pas de la technologie utilisée, or le processus de développement en dépend par nature. Le modèle “X” prend en compte le modèle objet. Deux cycles inversés sont en fait décrits, l’un pour une activité de synthèse d’un nouveau système et l’autre pour une activité d’acquisition de systèmes et de composants en vue de les réutiliser.

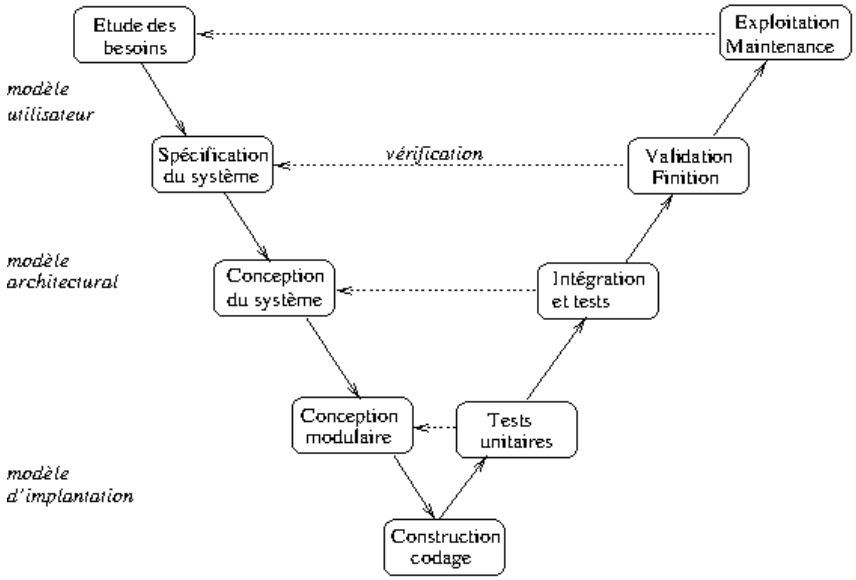


FIGURE B.5 – Un exemple de cycle de vie : le modèle “V”

B.4 Les spécifications

A chaque étape correspond un document résultat, qu’on nomme souvent produit ou spécifica-tion. Une **spécification** est un ensemble de modèles. La spécification initiale est le cahier des charges et la spécification finale est le programme (logiciel). On parle ainsi de spécification des besoins, spécification détaillée, spécification fonctionnelle ou encore spécification formelle. Par convention, le terme *spécification* désignera de façon générique les documents résultant d’une étape dans le processus de développement (*deliverables* en anglais). Le terme *spécification du logiciel*, quant à lui, désignera la description plus ou moins détaillée du comportement attendu du logiciel.

B.4.1 Types de spécification

On peut classer les spécifications selon leur forme ou leur degré de formalisme. Par exemple [Bra88] propose :

— les spécifications informelles, en langue naturelle, rédigées sans contraintes de forme,
— les spécifications standardisées, toujours en langue naturelle, mais avec une structure, un format et des règles précises (notations, glossaire, index, historique de modification...),
— les spécifications semi-formelles, qui utilisent un langage de spécification textuel ou graphique, langage doté d'une syntaxe précise et d'une sémantique assez faible permettant certains contrôles et une automatisation de certaines tâches (ex : Merise, SADT, SA-RT [AV01a]), UML [AV13]),
— les spécifications formelles, exprimées dans un langage à syntaxe et sémantique précises, construites sur une base théorique solide et permettant des validations automatisées.

Bien que la dernière forme soit préconisée, ce sont les trois premières qui sont utilisées dans l'industrie. Il est vrai que les notations semi-formelles graphiques sont plus accessibles au spécialiste du domaine de l'application. Mélanger plusieurs formalismes est intéressant s'il y a intégration ou du moins correspondance entre ces formalismes.

B.4.2 Relations entre spécifications

Il est intéressant de pouvoir manipuler les spécifications comme des objets mathématiques quelconques, c'est-à-dire de pouvoir les comparer par des relations, les transformer par des fonctions, les structurer par des opérateurs. Nous nous limiterons dans ce paragraphe à la description des relations entre spécifications et non pas à la construction de la spécification. On distingue deux sortes de relations, une relation horizontale et une relation verticale.

Relation horizontale

La manipulation d'une spécification complexe exige la décomposition en sous-spécifications. La sous-spécification est un morceau de la spécification globale. On affine ce lien de **structuration** en trois relations :

— **complémentaire** : Chaque sous-spécification prend en compte un aspect (un modèle) différent du système à modéliser : c'est la structuration multi-modèle. Une corrélation est donnée qui assure la complémentarité et la cohérence globale. Par exemple, une spécification classique d'analyse et conception structurée comprend un modèle pour les données, un modèle pour les traitements. Les traitements agissent sur les données.
— **inclusion** : Chaque sous-spécification est une partie de la spécification globale et n'a de sens que vis-à-vis de la spécification globale. C'est une structuration hiérarchique

au sens englobant/englobé. La sous-spécification décrit plus en détail un morceau de la spécification dont elle dépend. Le critère de décomposition est souvent simple et mono-valué : le temps (séquence), l'espace. Par exemple, l'analyse structurée est une spécification de plus en plus détaillée des fonctions du système.

— **utilisation** : Chaque sous-spécification décrit une partie relativement homogène et autonome du système. Le modèle de description est le même pour toutes les sous-spécifications. Une interface définit les liens entre modules. Le critère de décomposition est souvent complexe : notion de proximité sémantique, de type, etc.
Synonyme : enrichissement.

Relation verticale

La relation verticale exprime le degré de précision entre deux spécifications décrivant la même chose. La notion majeure ici est l'**abstraction** -(*cf.* Section 1.2.5) page 31. Informellement une spécification est moins abstraite qu'une autre si elle contient plus de détails (partitionnement de problèmes, prise en compte de cas d'erreurs, contraintes techniques, sélection d'algorithmes abstraits, représentation de données, implantation d'algorithmes [Abr84 ; Hay92]). L'abstraction apparaît clairement lorsqu'on utilise deux modèles différents pour exprimer la même chose. (par exemple type abstrait de donnée et représentation de données [Hoa72]) ou que les paramètres qu'on prend en compte sont de nature différente (fonctions du système, ordinateur). On parle alors de niveaux d'abstraction (voir page 314). Les modèles sont ordonnés par degré d'abstraction ou mieux par relation d'abstraction - relation (non symétrique) qui permet de passer de l'un à l'autre.

Nous allons tenter d'en donner quelques nuances en nous inspirant de [Gue94 ; Wir93 ; Hay92], mais de manière générale, c'est une notion très relative. On peut voir un objet abstrait comme un objet à n paramètres dont p sont fixés et $n - p$ sont libres.

— l'**implantation** de l'objet est atteinte lorsqu'il n'y a plus de paramètres libres.
Synonymes : réalisation, implémentation.

— le **raffinement** (et non raffinage) revient à augmenter le nombre de paramètres en remplaçant un paramètre par plusieurs paramètres libres ou fixes. Le raffinement dans les spécifications formelles [1] consiste à introduire des structures de données ou de contrôle de plus en plus proches du langage de programmation cible.
Synonymes : reification, enrichissement vertical, héritage de spécialisation.

— la **concrétisation** consiste à fixer certains paramètres. C'est par exemple le cas des spécifications génériques. Par exemple, la spécificication `Liste[T]` definit des listes indépendamment de leur contenu (le paramètre est un type `T`). La spécification `Liste [Entier]` concrétise la spécification `Liste[T]` en fixant son paramètre de type.
Synonymes : instanciation [2] actualisation.

1. Voir le Tome 2 de cet ouvrage.
2. Cette relation n'a pas exactement le même sens dans la programmation à objets, où elle représente la création des objets à partir de leur classe.

Le raffinement et la concrétisation sont des relations transitives.

Combinaison

Les deux axes ci-dessus sont corrélés. Par exemple, on peut raffiner une spécification en ajoutant une relation d'utilisation. On peut concrétiser une spécification en changeant une relation d'inclusion ou d'utilisation. Les mécanismes de structuration des spécifications sont une combinaison des relations décrites ci-dessus.

B.4.3 Outils de spécification

Le dossier des spécifications du logiciel est un ensemble de documents produits par le fournisseur de conseils et de service pour répondre aux besoins exprimés par son client dans le cahier des charges ayant donné lieu au contrat. Il s'agit de couvrir TOUS les besoins exprimés et uniquement ceux-là. Le choix de l'outil se fait en fonction de la méthode de spécification utilisée.

Un article du monde informatique, du 3 juillet 1989 (pages 22-26), donne une étude comparative des outils de spécification du logiciel, en prenant Idefo/SADT comme référence. Les critères de comparaison sont classés selon neuf fonctionnalités significatives :

— l'expression des connaissances,
— les niveaux de formalismes,
— le degré d'interactivité,
— la documentation de la spécification,
— la validation de la spécification,
— le cycle auteur-lecteur,
— les systèmes supportés, l'interfaçage avec d'autres outils, la réutilisabilité, le contrôle de consistance et de complétude sera un bilan global sur l'outil,
— la flexibilité, opposée aux systèmes dédiés,
— le prix.

Des outils, de plus bas niveau, facilitent l'écriture de la spécification et sa compréhension : flux de données entre acteurs, flux de données entre processus, actigrammes, modèle relationnel, modèle Z, modèle entité/association.

B.5 Les stratégies de développement

Le processus de développement dépend plus ou moins des modèles choisis, et vice-versa. Nous donnons ici des éléments de comparaison pour une analyse morphologique des méthodes en termes de stratégie. La morphologie est l'étude des formes (configuration et structure externe). La stratégie est l'ensemble d'actions coordonnées, de manœuvres en vue d'une victoire. La tactique est l'art de combiner tous les moyens militaires au combat, exécution locale, adaptée aux circonstances, des plans de la stratégie.

Niveaux d'abstraction

Pour affiner la description, les modèles sont organisés en niveaux d'abstraction vis-à-vis de l'implantation finale. Trois niveaux d'abstraction sont distingués :

— le niveau conceptuel (fonctionnalités du système)
— le niveau organisationnel (architecture logicielle et interfaces)
— le niveau opérationnel (implantation).

Ces niveaux peuvent être plus ou moins documentés. Par exemple, dans les applications de gestion, les trois niveaux d'abstraction sont :

— la description du système indépendamment de l'organisation par des modèles conceptuels (le `quoi`),
— la description du système vis-à-vis de l'organisation par des modèles logiques (le `ou` et le `quand`),
— la réalisation du système par des modèles physiques (le `comment`).

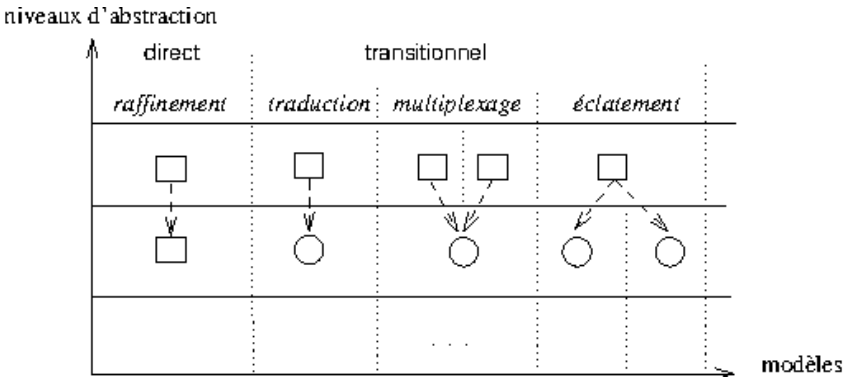


FIGURE B.6 – Analyse morphologique selon les niveaux d'abstraction

Un niveau d'abstraction comprend un ou plusieurs modèles, c'est notamment le cas lorsqu'on considère plusieurs plans de description (structurel, dynamique, fonctionnel). Un même modèle peut aussi être décrit dans plusieurs niveaux d'abstraction. Par exemple, dans les méthodes à objets, les premières descriptions ne contiennent que les noms de propriétés (attributs et opérations). Les types de ces propriétés sont ensuite donnés. Le corps des opérations n'est donné que bien plus tard. Enfin, lorsqu'on passe d'un niveau à un autre, un modèle peut être *concrétisé*, ou *raffiné*, dans un autre modèle ou *éclaté* en plusieurs modèles. Inversement on a *multiplexage* de plusieurs modèles complémentaires en un seul modèle de niveau inférieur.

Croisement entre modèles et processus

Deux familles de cycles pour le processus sont proposées : les processus linéaires et les processus itératifs. Chaque famille comprend les étapes habituelles du développement. Elles sont répétées plusieurs fois si le cycle est itératif. Un niveau d'abstraction est défini en une étape ou en plusieurs étapes. Dans ce dernier cas, il y a indépendance entre les modèles et le processus de développement.

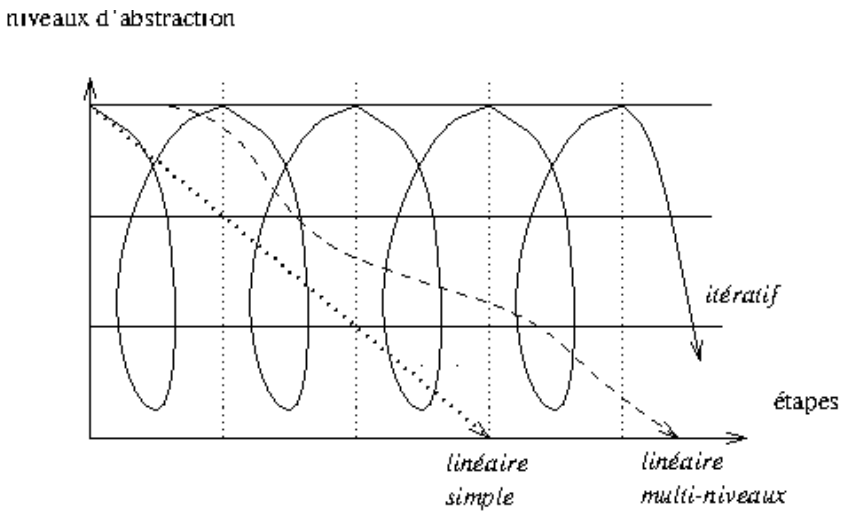


FIGURE B.7 – Croisement modèle/processus

La prise en compte des schémas des figures B.6 et B.7 fournit donc les stratégies possibles de développement. On le voit aisément, le nombre de cas possibles est important. Une représentation graphique synthétique est donc difficile à exprimer. Les choix doivent être faits selon des critères de qualité.

B.6 La qualité

Nous traitons dans cette section de la qualité et des outils des spécifications en général et celle du logiciel en particulier. Nous abordons enfin la qualité du processus de développement.

B.6.1 Qualité de la spécification

Selon [Mey85], les sept défauts d'une spécification sont :

- le **bruit** : éléments inutiles qui empêchent de saisir l'essentiel,
- le **silence** : absence d'information sur une caractéristique du problème,
- la **sur-spécification** : introduction d'éléments de la solution,
- la **contradiction** : une propriété est vraie dans une partie du texte et fausse dans une autre partie du texte,
- l'**ambiguïté** : texte interprétable de plusieurs manières,
- la **référence avant** : élément du texte utilisant une information qui est définie (sans qu'on le précise) plus loin dans le texte,
- le **vœu pieu** : caractéristique présentée de telle façon qu'aucune solution ne sera valide.

ou encore, dualement et selon [Bra88], les qualités attendues d'une spécification sont :

- **au bon niveau** : la spécification correspond uniquement aux objectifs de l'étape en cours.

— **conforme aux besoins réels de l'utilisateur** : il faut s'efforcer de mettre en valeur le besoin, pour des utilisateurs futurs.
— **communicable** : une spécification est faite pour permettre la communication entre divers acteurs d'un projet (client, réalisateur, concepteur, mainteneur...).
— **cohérente** : il est difficile de réaliser un produit dont les spécifications sont contradictoires.
— **structurée et homogène** : une spécification est une référence et, comme pour tout document de ce type, il est nécessaire qu'elle obéisse à un plan logique qui en facilite la lecture et la consultation ponctuelle ainsi que la modification.
— **ayant des règles d'écriture simples** : par exemple, si le document est en langue naturelle, il faut utiliser un style concis, avec des verbes conjugués au présent et à la forme active.
— **complète et précise** : c'est une qualité essentielle, mais difficile à obtenir.

Les méthodes formelles sont un apport fondamental dans l'obtention de telles qualités.

B.6.2 Qualité du logiciel

Le logiciel est un cas particulier de spécification. Des facteurs de qualité du logiciel ont été donnés par B. MEYER dans [Mey97]. Les facteurs externes de qualité sont :

— la **validité** : aptitude d'un produit logiciel à réaliser exactement les tâches définies par sa spécification.
— la **robustesse** : aptitude d'un logiciel à fonctionner même dans des conditions anormales.
— l'**extensibilité** : facilité d'adaptation d'un logiciel aux changements de spécification.
— la **réutilisabilité** : aptitude d'un logiciel à être réutilisé en tout ou en partie pour de nouvelles applications.
— la **compatibilité** : aptitude des logiciels à pouvoir être combinés les uns avec les autres.

D'autres qualités du logiciel sont moins cruciales :

— l'**efficacité** : bonne utilisation des ressources du matériel.
— la **portabilité** : facilité avec laquelle le produit peut être adapté à différents environnements matériels et logiciels.
— la **vérifiabilité** : facilité de préparation des procédures de recette et de certification (test, déverminage).
— l'**intégrité** : aptitude des logiciels à protéger leurs différents composants (programmes, données, documentation) contre des accès et des modifications non autorisés.
— la **facilité d'utilisation** : facilité avec laquelle les utilisateurs d'un logiciel peuvent ap- prendre comment l'utiliser, comment le faire fonctionner, comment préparer les données, mais aussi comment interpréter les résultats et réparer les effets en cas d'erreur.

Des critères internes permettent d'atteindre ces facteurs externes de qualité. Elles sont perceptibles seulement par les informaticiens. Voici les critères proposés par MEYER [Mey97] :

— la **modularité** : c'est la décomposition du logiciel en composants facilement appréhendables et relativement indépendants.
— la **complétude** : c'est le degré d'implantation des spécifications. Un logiciel est complet si toutes ses spécifications externes sont opérationnelles.
— la **cohérence** : c'est la possibilité de faire des retours en arrière dans le cycle de dévelop- pement. En particulier de faire remonter une erreur détectée en maintenance au niveau de l'implantation, de la conception voire de l'analyse.
— la **généralité** : plage d'application potentielle des composants logiciels.

— l'**auto-documentation** et **lisibilité** : possibilité d'extraction de la documentation depuis les composants logiciels.

Le lecteur trouvera dans [Som11] une description détaillée des critères de cohésion (sept niveaux) et de couplage. Il trouvera dans [Mey97] cinq critères pour évaluer la modularité. La FIGURE B.8 donne une idée de la corrélation entre les critères de qualités et les facteurs de qualités. La modularité a incontestablement un poids majeur.

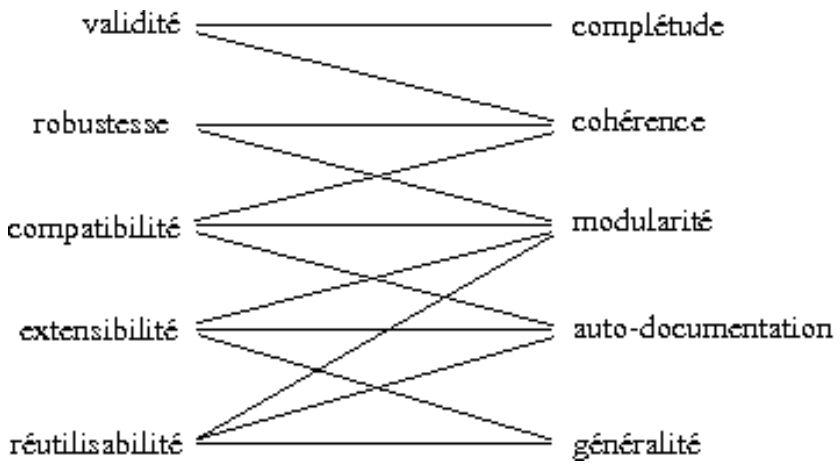


FIGURE B.8 – Lien entre les facteurs et les critères du logiciel

La qualité du logiciel est normalisée sous forme de trois modèles dans le standard ISO/IEC 25010 :2011 intitulé Ingénierie des systèmes et du logiciel — Exigences de qualité et évaluation des systèmes et du logiciel (SQuaRE) — Modèles de qualité du système et du logiciel [3]. *The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value.* La qualité y est définie par un exemble de caractéristiques et sous-caractéristiques jusqu'à atteindre des propriétés.

Il est à noter que des indicateurs

3. `https://www.iso.org/fr/standard/35733.html`

B.6.3 Qualité du processus de développement

Le processus de développement est ce qui permet de faire le lien entre les différents modèles de description. Les principales qualités attendues sont :

— **sûreté** : la démarche doit minimiser les retours arrières et permettre des validations périodiques.
— **terminaison** : le cycle doit permettre d'obtenir les produits en un temps fini, que ce soit à chaque étape ou globalement.
— **rigueur** : l'enchaînement des étapes doit suivre un cheminement logique, correspondant aux habitudes des acteurs du développement.
— **cohérence, complétude** : les étapes doivent être cohérentes entre elles (pas de duplication inutile de tâches) et former un tout (pas d'oublis).
— **souplesse** : la démarche doit s'adapter en fonction de l'application à développer.
— **accessibilité** : c'est la possibilité de comprendre les décisions prises au cours du processus.
— **rentabilité** : c'est la capacité à capitaliser l'expérience dans le processus de développement.

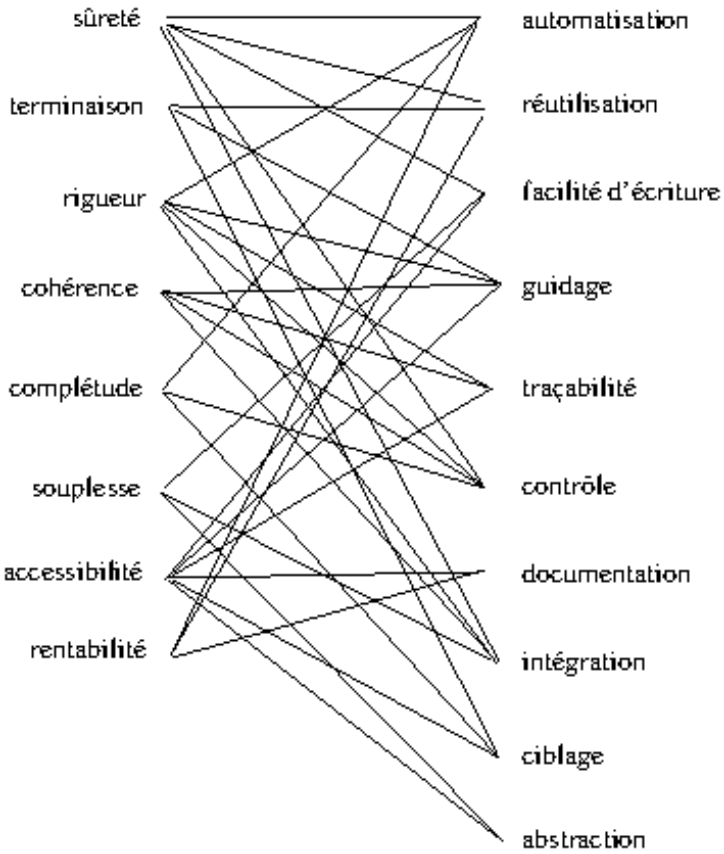


FIGURE B.9 – Lien entre les facteurs et les critères du processus

Voici quelques critères qui permettent d'atteindre ces qualités (il faut noter que ces qualités dépendent aussi des modèles de représentation) :

— **automatisation** : plus on automatise, moins on fait d'erreurs et plus on va vite.

— **réutilisation** : la capitalisation des efforts réduit les coûts en taille, temps. La réutilisation augmente la sûreté.
— **facilité d'écriture** : les modèles choisis doivent avoir des concepts simples et naturels pour le développeur.
— **guidage** : le guidage est la présentation méthodique des règles à appliquer, des opérations à réaliser pour obtenir un "bon" résultat.
— **traçabilité** : c'est un moyen de contrôler la cohérence entre les modèles. Les informations et traitements jugés utiles au départ doivent exister sous une forme ou une autre dans les étapes ultérieures.
— **contrôle** : un contrôle régulier dès les premières étapes.
— **intégration** : cohérence entre les modèles de description d'une même étape ou de deux étapes successives.
— **documentation** : le raisonnement et les décisions doivent apparaître clairement.
— **ciblage** : le domaine d'application de la méthode doit être clairement exprimé pour éviter des blocages dans la modélisation.
— **abstraction** : le raisonnement et la preuve doivent progressivement prendre en compte les concepts de programmation.

La FIGURE B.9 donne une idée de la corrélation entre les critères et les facteurs pour la qualité du processus.

B.6.4 Exigences Non fonctionnelles

Nous donnons ici deux exemples de taxonomie de besoins non-fonctionnels, liés en parties aux qualités décrites ci-avant, celle de [Som11] et celle de [AKS16]. On peut remarquer que ces typologies sont différentes. On notera la catégorie "*conflicting NFR*" dans la FIGURE B.11 qui met en évidence l'idée de conpromis car agir sur une qualité peut dégrader une autre.

B.7 La validation

Valider c'est contrôler que le (produit) résultat correspond à ce qui était attendu. La validation est un contrôle qui fait intervenir largement les "utilisateurs". Vérifier c'est contrôler que le produit respecte le cahier des charges [Som11 ; Pre10]. La vérification est donc plus une "affaire" de spécialistes.

Les études des facteurs de coûts du logiciel de [Boe82] et [Cal90] montrent l'importance de la validation. Plus une erreur est détectée tardivement, plus sa correction est chère. C'est pourquoi l'effort doit être porté sur la spécification plutôt que la conception. De plus la validation doit être réalisée par une équipe pluri-disciplinaire, comprenant des informaticiens certes, mais aussi des utilisateurs et des intervenants extérieurs au projet. Ces participants n'auront pas le même point de vue. Par exemple, les utilisateurs seront à priori détachés de la solution, les intervenants extérieurs permettront d'élargir le champ

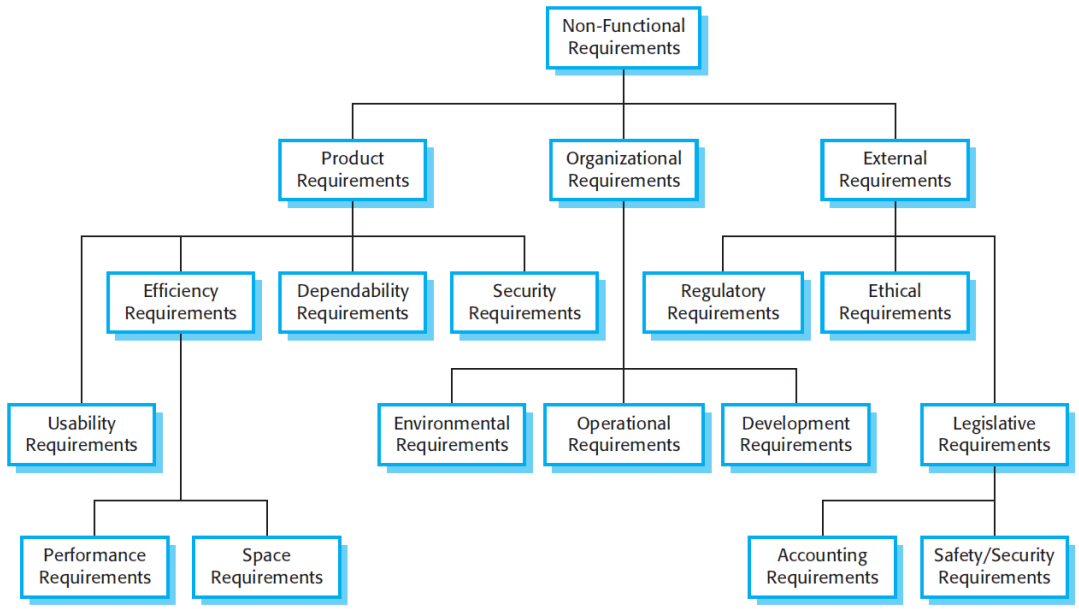


FIGURE B.10 – Taxomonie des exigences Exigences Non fonctionnelles [Som11]

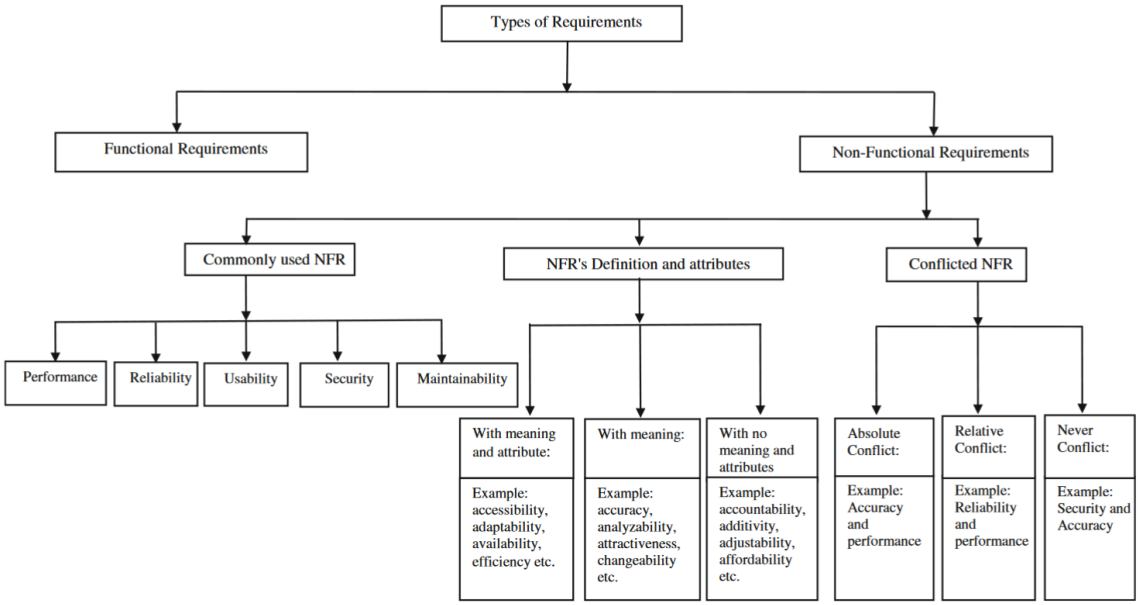


FIGURE B.11 – Taxomonie des exigences Exigences Non fonctionnelles [AKS16]

d'investigation. La confrontation des points de vue devrait aboutir à un compromis raisonnable. Afin de s'assurer de la bonne marche d'un développement, un contrôle est donc obligatoire. Il est généralement fait entre les étapes.

Un point clé de la vérification et plus encore de la validation est la lisibilité des documents. En effet, comment les utilisateurs pourront-ils vérifier l'exactitude d'une spécification, ou pire d'un programme, s'ils ne comprennent pas, ou pas bien, l'information qui leur est communiquée ? La lisibilité dépend du niveau de formalisme du langage utilisé (voir la section B.4.1) et de la pertinence des informations qui accompagnent les spécifications. Si on considère la lisibilité par la concision, la précision et l'exactitude, alors les spécifications formelles, que nous aborderons en détail dans le second volume, sont un apport

considérable. Si on considère la lisibilité par l'abstraction et la rapidité de lecture, alors les spécifications graphiques sont plus riches. Malheureusement, ces dernières souffrent d'un manque de précision qui engendre des ambiguïtés ou des imprécisions.

La vérification (ou le contrôle de la correction) se fait de deux manières : soit par preuve de propriétés de la spécification au moyen d'outils spécialisés (prouveurs), soit par construction automatique (et prouvée) de programmes.

Hormis la lecture des spécifications, à laquelle nous avons fait allusion ci-dessus, le **maquettage** (ou **prototypage**) et les tests sont les deux principaux moyens de validation. Le premier permet à petite échelle de se rendre compte approximativement de l'allure générale du résultat et des problèmes pouvant intervenir. Le second permet de vérifier des hypothèses, des objectifs et des contraintes. Il nécessite une formalisation précise des besoins.

B.7.1 Test

Par définition, le test est la vérification de cohérence entre la spécification et la réalisation, et ce au moyen d'un échantillon (représentatif). La validité du test repose entièrement sur la pertinence du sous-ensemble de données choisi. Deux "écoles" divisent les spécialistes du test :

- les tenants des techniques de génération systématique de jeux d'essais, soit fonctionnels (boîte noire), soit structurels (boîte blanche). Les stratégies de sélection de tests utilisées reposent sur la correction (i.e. les défauts). Les jeux de données sont produits systématiquement à partir d'éléments comme le code (boîte blanche) ou la spécification (boîte noire). Aux stratégies de sélections de données de tests peuvent être associées des *mesures d'efficacité ou de couverture*. Les plus couramment utilisées sont les couvertures de branchement et de modules. Une mesure de couverture est définie par le rapport entre le nombre de jeux de données de test satisfaisant le critère de couverture et le nombre total de jeux de données utilisés. Il s'agit de mesure *dynamique*. *Seules les spécifications formelles ou semi-formelles permettent d'évaluer automatiquement une couverture*. Des mesures de *testabilité* peuvent aussi être associées aux stratégies de sélection. On les définit comme le nombre minimal de jeux de données nécessaires pour satisfaire un critère donné en supposant que ces jeux peuvent être exécutés. Il s'agit de mesures *statiques*.
- les tenants du test statistique et des modèles de fiabilité. Les domaines de données sont échantillonnés selon des lois de répartition *statistiques*. L'échantillon doit refléter la répartition. Des modèles de *fiabilité* permettent d'établir les mesures de fiabilité. Quatre catégories sont distinguées :
  - le temps entre deux pannes,
  - le nombre de pannes observées pendant une durée donnée,
  - le nombre de pannes observées pour un ensemble de défauts introduits dans le programme,

— le nombre de pannes observées pour un jeu de données de test, sélectionné selon un profil d'utilisation donné.

En raison de contraintes de délais et de coûts, il n'est pas toujours possible d'utiliser toutes les stratégies de test, il faut panacher.

B.7.2 Maquettage

Dans l'optique de limiter le nombre d'erreurs dans le logiciel final, on peut soit améliorer la programmation, soit mieux formuler les spécifications, soit construire des prototypes. La programmation structurée et le génie logiciel (outils, environnements, guides, automatisation) ont apporté un premier élément de réponse. Les langages de spécification et les outils qui les accompagnent (éditeurs, bibliothèques, interprètes) doivent permettre d'exprimer clairement les éléments du problème. Enfin le **prototypage** correspond à un besoin de vérifier le comportement réel du produit en cours de développement. Le prototype sert aussi à la communication entre le spécifieur et le réalisateur, il est une approximation de l'interprétation que le réalisateur fait de la spécification. Ces trois aspects sont complémentaires.

Le prototypage consiste en quatre étapes : la *sélection fonctionnelle* (choix des fonctions à réaliser), la *construction* (éventuellement confondue avec la spécification si le langage de cette dernière est exécutable), l'*évaluation* ("feedback" sur le développement), l'*utilisation ultérieure* du prototype, soit est mis de côté, soit fait partie intégrante du produit. Prototyper n'est pas spécifier. Le prototypage doit être préparé pour réussir. La spécifica-tion est indépendante de toute mise en œuvre. La spécification peut servir de contrat entre l'utilisateur et le réalisateur. Enfin et surtout, la spécification est une description cohérente et complète du produit alors que la simulation en est une approximation.

B.8 Industrialisation

L'utilisation de nombreux outils à différents stade du développement et du déploiement du logiciel implique une intégration (continue) de l'ensemble qu'on peut assimiler au fontionnnement d'une usine de production de produits finis comme l'illustre la Figure B.12[4] La Figure B.13[5] montre une telle organisation pour le développement du logiciel (pas son déploiement). Le courant DEVOPS comprend l'intégration à la fois du développement et de la production.

On parle ainsi d'usine logicielle, qu'il ne faut pas confondre avec les lignes de produits (*Software Product Line* SPL) [PM18] qui sont une approche de développement dans

4. source : `http://www.improve-technologies.com/2012/06/29/`

5. source : `https://fr.slideshare.net/DjamelZouaoui/usi-casablanca-2010-industrialisation-et-intgration-continue`

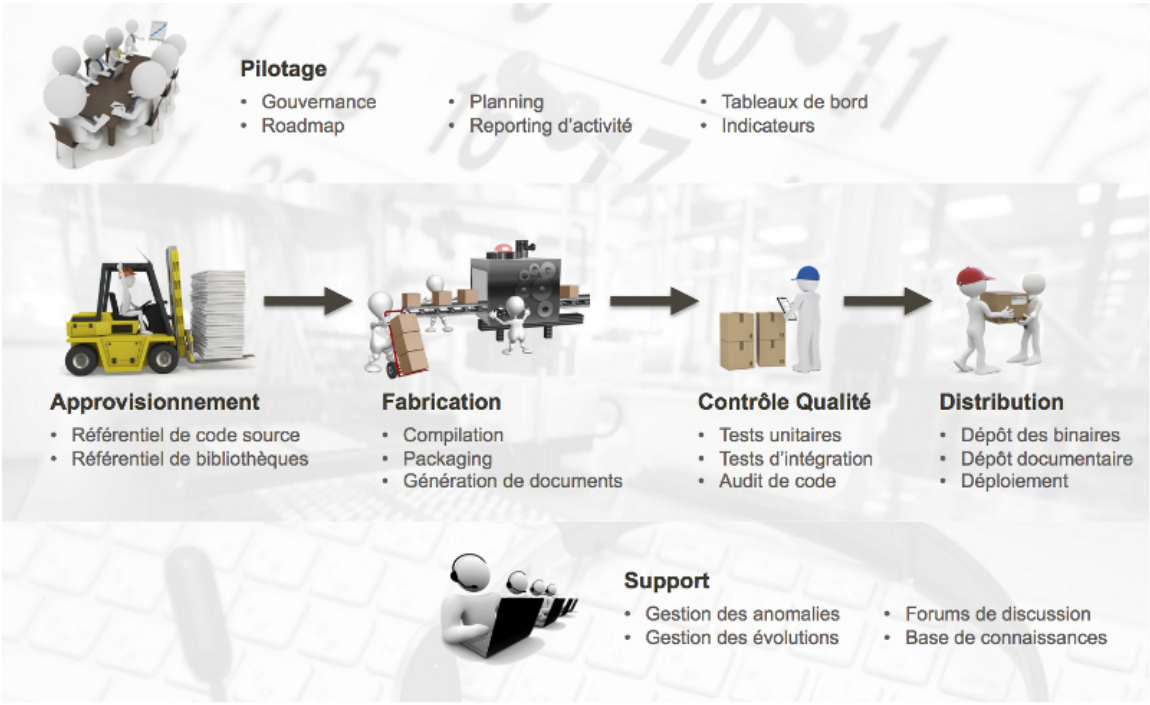


FIGURE B.12 – Production automatisée de logiciel

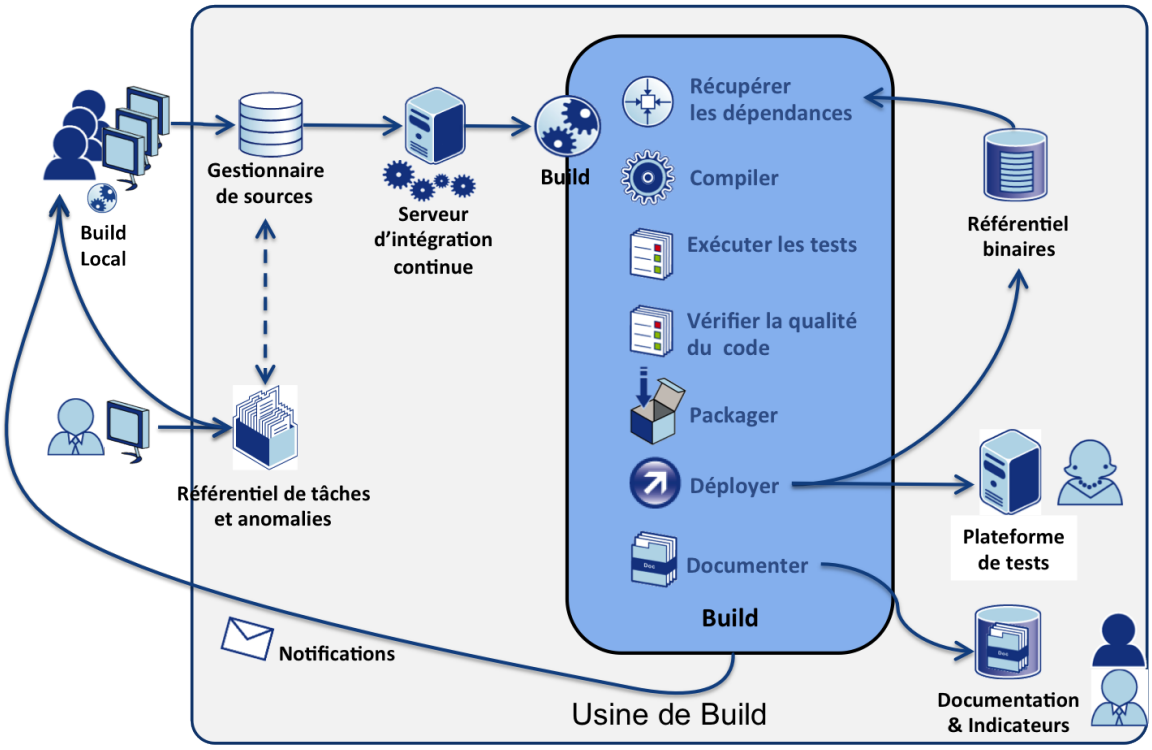


FIGURE B.13 – Usine logicielle

laquelle on configure des applications comme une famille de produits pour un domaine spécifique ayant des propriétés communes (*features*) et des spécificités, on parle de variabilité. Chaque application est alors conçue par assemblage de composants ou services, selon une configuration donnée, comme on le ferait pour choisir les options de sa nouvelle voiture sur le catalogue du constructeur.

Annexe C

Le développement du logiciel avec UML

Cette annexe, reprend en partie le chapitre 1 de [AV13] et le complète. Elle présente le développement du logiciel avec UML, notation large spectre que nous avons déjà introduit dans la section 1.1.3 du chapitre 1. Les bases de la notation UML sont examinés en section C.1. Dans la section C.2, nous présentons les principes, les grandes activités et les principales approches, en insistant notamment sur le RUP et le 2TUP. Puis nous croisons activités et notation dans la section C.2.6 en indiquant quels diagrammes sont utilisés à quelle étape. La section C.3 propose des pointeurs sur des outils autour d'UML.

Le langage de modélisation unifié UML (*Unified Modeling Language*) et le processus unifié UP (*Unified process*) représentent une étape importante dans l'histoire du développement du logiciel. Il s'agit de l'étape où à la fois les éditeurs, les prestataires et les utilisateurs se sont réunis pour décider de se parler dans la même langue. En effet, le langage UML a permis de normaliser la jungle des notations utilisées alors dans les années 1990 dans les méthodes de développement à objets. De son côté le processus unifié est une synthèse de ce qui semble essentiel au développement orienté composants et à grande échelle. UML (*Unified Modeling Language*) s'est imposé comme le langage universel pour modéliser des systèmes logiciels et développer les applications informatiques. UML possède un large éventail de concepts et notations qui lui permet de couvrir la plupart des besoins en modélisation : de l'analyse des besoins à la programmation à objets, et ce pour les trois facettes de modélisation des systèmes (la structure, la dynamique et les fonctions) avec des représentations graphiques adaptées à la communication entre acteurs du développement. Complété par le langage de contraintes OCL (*Object Constraint Language*) et la sémantique des calculs AS (*Action Semantics*), UML devient le langage universel pour l'ingénierie des modèles, l'approche MDA (*Model Driven Approach*) prônée par l'OMG (*Object Management Group*) l'organisme qui normalise UML.

Malheureusement, cette approche idéale atteint rapidement ses limites en pratique. La raison principale est qu'un langage même aussi large-spectre ne couvre pas tous les artefacts métiers ou techniques. D'un point de vue métier, on ne traitera pas de la même manière des applications temps réels, des systèmes d'information d'entreprise, des systèmes scientifique. Par exemple, dans une application temps réel, la gestion du temps, des interruptions et des communications nécessite de nouveaux concepts. Le langage doit alors être étendu ou prendre une sémantique particulière. D'un point de vue technique,

UML ne prend pas en compte tous les artefacts de la programmation. Par exemple, pour un développement Web on utilisera des servlets, des scripts, des pages HTML... Ici aussi le langage doit être étendu ou prendre une sémantique particulière. Une autre critique fondamentale est que la diversité des concepts ne permet pas de définir une sémantique formelle. En résumé, UML n'est ni complet ni cohérent. C'est encore plus vrai avec UML 2 qui étend le nombre de diagrammes de neuf à quatorze, à ce jour, avec des recouvrements de notation qui induisent plus de redondances. Toutefois avec UML 2, la vision de l'OMG est différente, UML devient un standard abstrait, chacun personnalisant ensuite, via des profils, la notation et fixant, si possible, une sémantique précise du langage devenu spécifique et appelé DSL (*Domain Specific Language*). La norme porte alors sur la notation graphique et des règles de formation des diagrammes (*well-formedness*).

C.1 La notation UML

La notation UML inclut un grand nombre de concepts autour de l'objet mais aussi de l'analyse des besoins (acteurs, cas d'utilisation), de la conception du logiciel (composants, modules, processus) ou de l'implantation (nœuds, liaisons, déploiement). La raison est que cette notation est conçue pour décrire des modèles couvrant l'ensemble du cycle de développement. De plus certains concepts sont perçus à des niveaux d'abstraction différents. Par exemple, les opérations des objets sont décrites plus finement à la conception. UML vise à unifier les notations des nombreuses méthodes à objets. Elle est donc un compromis. Heureusement, les auteurs se sont entendus pour définir une notation de base, que chacun peut ensuite étendre par le mécanisme de stéréotypage. Par exemple, une classe abstraite est une classe affinée par le stéréotype `«abstract»`. Ainsi, une limite raisonnable est donnée au nombre de concepts.

Nous désignons par UML 1.X la famille de langage de première génération d'UML. Cette famille est caractérisée par une approche **unificatrice** de la notation, qui fait d'UML un langage universel. Dans cette vision, UML s'inspire de nombreuse références et regroupe un maximum de concept pour correspondre à tous les besoins. Les principales influences sont : (1) Booch : Catégories et sous-systèmes (2) Embley : Classes singletons et objets composites (3) Fusion : Description des opérations, numérotation des messages (4) Gamma, et al. : Frameworks, patterns, et notes (5) Harel : Automates (Statecharts) (6) Jacobson : Cas d'utilisation (use cases) (7) Meyer : Pré- et post-conditions (8) Odell : Classification dynamique, éclairage sur les événements (9) OMT : Associations (10) Shlaer-Mellor : Cycle de vie des objets (11) Wirfs-Brock : Responsabilités (CRC) On peut aussi affirmer que UML 1.X vise à satisfaire les utilisateurs (ceux qui développent avec UML). UML répond en ce sens à la cacophonie des méthodes et notations des années 1990.

Depuis, les principes fondateurs ont évolués, notamment sous l'impulsion des architectures applicatives émergentes (composants, Architectures de logiciels (ADL), services)

mais aussi de l'approche l'*ingénierie des modèles* (IDM) prônée par l'OMG (*Object Management Group*) sous l'appellation *Model Driven Architecture* (MDA). La famille UML 2.X marque cette double rupture, dans laquelle UML devient un langage pivot pour une famille de langages spécifiques, les *Domain Specific Languages* (DSL). De ce fait, la sémantique est devenue moins contraignante pour épouser les sémantiques précises des DSL. UML 2.X n'est plus portée par les utilisateurs mais par les « fournisseurs » d'outils pour les modèles.

Présenter la notation UML est un exercice assez délicat car elle est complexe. Cette complexité trouve ses sources dans le nombre de concepts représentés, la variété d'utilisation de ces concepts et la confusion possible entre la notation et le modèle de la notation (le métammodèle) qui permet de l'exprimer.

> UML 1.X disposait d'un guide de la notation ou d'un manuel de référence de la notation. Dans UML 2.X, la spécification de superstructure donne la vision pour l'utilisateur, mais c'est trop technique pour un manuel utilisateur tandis la spécification d'infrastructure, orientée vers l'architecture, donne bien l'idée d'une référence pour les fournisseurs d'outils.

Compte-tenu des remarques précédentes, nous n'avons pas adopté une présentation rigoureuse et complète de la méthode. Nous évitons de présenter l'ensemble des concepts puis les diagrammes dans lesquels on les retrouve. Nous évitons aussi d'utiliser le métammodèle comme support car il nous éloigne des concepts. Ces deux alternatives entraînent une description structurelle, longue et rébarbative, de la notation. Nous avons choisi de présenter les concepts dans leur contexte d'utilisation habituel.

Avertissement

Cet ouvrage n'est pas conçu comme un manuel de la notation : tous les concepts ne seront pas illustrés individuellement par des exemples. Nous avons choisi une approche plus globale, en commentant la notation sur des exemples plus « large ». Pour un exposé de la notation, consulter [Aud09 ; Fow04 ; PP05 ; Bal06]. Noter que les manuels de référence de l'OMG (infrastructure [Gro11c], superstructure [Gro11b]) sont orientés par l'approche MDA, c'est-à-dire plus pour les manipulateurs de modèles que pour les utilisateurs finaux de la notation, c'est pourquoi nous ne les conseillons pas ici.

Dans cette section, nous traçons les grandes lignes de la notation : les éléments généraux, les concepts fondamentaux, les outils de structuration et de présentation.

C.1.1 Les éléments généraux

Nous présentons dans cette section quelques éléments et mécanismes généraux de la notation qui nous semblent nécessaires à la compréhension de la suite du chapitre.

Terminologie

La notation UML inclut trois sortes de briques : les éléments, les relations et les diagrammes [RJB98]. Les éléments de modélisation représentent les concepts (au sens où nous l'avons entendu jusqu'ici) et des facilités de notation (paquetages, notes, contraintes). Les relations représentent des liens entre éléments de modélisation [1]. Elles sont détaillées dans la section C.1.2. Un paquetage est un élément qui regroupe éléments et relations. La notion de diagramme ne fait pas partie des éléments de modélisation, contrairement au modèle qui est un paquetage. Un diagramme est une vue de l'utilisateur sur un modèle.

Stéréotype

La notion de **stéréotype** est un élément clé de l'extensibilité de la notation UML et de son adaptabilité aux AGL et aux méthodes propriétaires. Chaque concept de la notation peut être "spécialisé" par stéréotype. Le stéréotype est une annotation qui enrichit la description, mais n'intervient pas a priori dans les vérifications des modèle produits. Par exemple, une classe abstraite est un stéréotype de la classe. Les stéréotypes sont représentés par des doubles chevrons « ». Au fil des versions d'UML, certains stéréotypes ont été intégrés directement dans la notation. Par exemple, un nom de classe en italique indique que la classe est abstraite.

Collaboration, interaction

La notion de collaboration a quelque peu perdu de la valeur au fil des évolutions d'UML. Une **collaboration** décrit une structure d'éléments qui collaborent selon des rôles déterminés. Cette terminologie provient de la méthode Fusion, une des sources d'inspiration d'UML. C'est une vision **statique** (structurelle) des liens entre objets. Il est d'usage de dire qu'une collaboration « implante » un cas d'utilisation.

Une **interaction** décrit un échange entre les éléments de la collaboration. C'est la vision **dynamique** (comportementale) des liens entre objets. On dira qu'une interaction se fait au travers d'envois de messages ou de signaux. Dans UML, deux points de vue équivalents représentent les interactions : les diagrammes de séquences qui insistent sur l'enchaînement et la causalité des interactions (vision temporelle) tandis que le diagramme de communications (ou de collaborations) qui met plus en évidence une vision spatiale de la collaboration. Le support de la collaboration est un réseau d'objets interconnectés (objets et liens).

> Dans UML 2.X, il y a clairement une dichotomie mais aussi une dualité entre la vision structurelle et la vision comportementale du système. Est-ce un retour aux sources des méthodes traditionnelles et à la vision Merise de la systémique ?

1. Bien que nous les présentions à part, les relations sont aussi des éléments de modélisation au sens du métammodèle d'UML.

Paquetage

Un **paquetage** est un mécanisme qui permet de grouper des éléments. Il permet de structurer la spécification mais ne correspond pas à une abstraction d'un élément de conception comme le composant. Les paquetages sont surtout utilisés pour regrouper des classes. Un paquetage peut contenir d'autres paquetages. Le paquetage est représenté par un dossier. Des stéréotypes du paquetage le spécialisent en fonction du contexte. On parle de `catégorie` pour les paquetages de la vue logique et de `sous-systèmes` pour les paquetages d'implantation.

Comme pour les autres notations, plusieurs interprétations sont possibles. Dans une première interprétation, un paquetage est un module. Les relations entre paquetages sont alors des variantes de la relation de dépendance et on évite, autant que possible, les dépendances circulaires. Le critère de cohérence est soit une proximité logique (les cassettes et les exemplaires), soit de fonctionalité (objets interfaces, objets métiers)... Une autre interprétation est le découpage "physique" en sous-systèmes.

Système, sous-système

Un système est l'élément qu'on développe et pour lequel on construit des vues. Un sous-système est une partie d'un système. Chaque sous-système d'un système est représenté par un paquetage. La relation principale est l'agrégation mais la spécialisation est aussi autorisée. Le paquetage d'un sous-système comprend trois éléments : l'interface (opération), les éléments de spécification et les éléments de réalisation. Les modélisations faites pour un système peuvent l'être identiquement pour un sous-système. Autrement dit, le discours s'applique quelle que soit l'échelle.

Espace de nommage

Un *espace de nommage* permet de définir un contexte lexical pour les éléments de modélisation. De nombreux concepts sont des expaces de nommages : les paquetages bien sûr, mais aussi les classes, les opérations ou même les états. Un élément sera donc désigné par son nom et l'ensemble des espaces de nommages qui le contiennent hiérarchiquement. On utilise le séparateur `::` pour construire le nom complet, par exemple `pack::class:op(param)`.

C.1.2 Les relations dans le modèle à objets

Les relations permettent de structurer les éléments d'un modèle à objets. Ayant des objets et des classes, nous trouvons naturellement des relations entre instances (objets), des relations entre types (classes, *classifiers*) et des relations entre instances et types.

i) Relations entre objets et types

La relation entre un objet et sa classe est la relation d'**instanciation**. Un objet est instance d'une seule classe qui définit sa structure et son comportement.

ii) Relations entre objets

La base de l'interaction entre objets est l'envoi de message qui suppose un médium, une relation de l'objet client (émetteur) et l'objet serveur (receveur). Le client invoque un service du serveur, c'est une relation de **clientèle**, ou de **délégation**. Par exemple, on peut « demander » à une cassette quel est son film. C'est une relation fondamentale dans un modèle à objets, qui se représente en UML par un **lien** entre objets. Les liens sont abstraits au niveaux des classes par une **association** entre les classes de ces objets.

Un objet (le composé) peut inclure un autre objet (le composant). Cette relation de **structuration** est appelée **agrégation**, relation **tout-partie**. Elle induit une relation de clientèle de l'objet composé vers l'objet composant. L'agrégation est une relation dont les contours ne sont pas toujours bien définis [HSB99] car elle recouvre divers aspects :

— Visibilité. L'objet composant est encapsulé dans l'objet composé. De ce fait il doit suivre les règles de visibilité définies par celui-ci. En particulier, il se pose la question de savoir à quels objets le composant peut être associé : uniquement à des objets composants du même composé ou alors à n'importe quel objet du système ?

— Durée de vie. Quelles sont les règles de synchronisation entre la vie de l'objet composé et celle de ses composants ? Y-a-t'il suppression en cascade ?

— Evolution dynamique. L'objet composant a-t-il un flot de contrôle indépendant (concurrent) de l'objet composé ou son flot est-il assujetti à celui du composé ?

Si l'objet composant est fortement corrélé à l'objet composé (définition à fixer sur les critères ci-dessus) alors on parle de d'agrégation forte ou de **composition**.

Dans une modélisation à objets, les objets sont rarement représentés, on modélise plutôt leurs classes, les relations ci-dessus sont donc abstraites au niveau des classes. Illustrons les différences sur un exemple via la notation UML. Soient les trois relations de la Figure C.1.

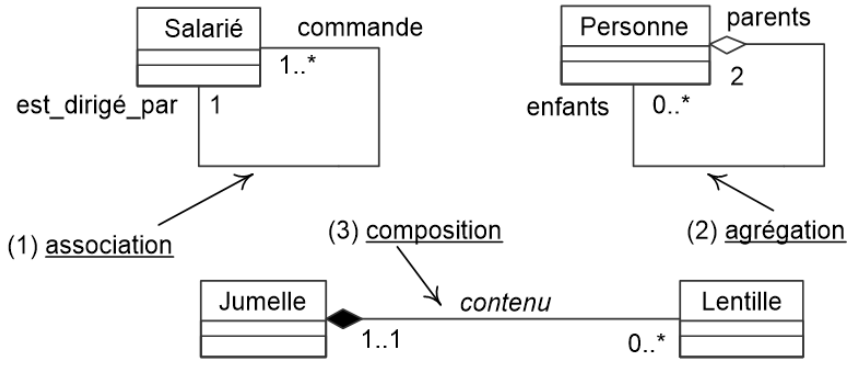


Figure C.1 – Association, agrégation, composition

1. L'association symbolise des liens entre deux instances des classes associées. Un salarié est dirigé par un autre salarié. Le patron est dirigé par lui-même. Dans une association, un objet peut être lié à lui-même.
2. Une personne a deux parents mais ne peut être parent d'elle-même. L'agrégation est pertinente ici car c'est une association non réflexive (un objet n'est pas lié à lui-même) et les objets ont des lignes de vie propres (parents et enfants ont une vie indépendante !).
3. Les lentilles d'une jumelle font partie intégrante de la jumelle. Cette dépendance forte s'exprime en UML par une relation de composition. La composition est exclusive.

iii) Relations entre classes

Il existe deux types de relations entre classe : l'utilisation et l'héritage. Nous traitons aussi de la généricité.

Relation d'utilisation La relation d'**utilisation** exprime le fait que pour réaliser ses fonctionalités un objet utilise les services d'autres objets. Elle correspond à l'importation modulaire : un module utilise les services d'un autre module pour réaliser ses propres services. Cette structuration entre classes peut être affinée. Etudions quelques critères d'affinement, qui font parfois la différence entre deux modèles objets.

— Agrégation ou composition : ce point a été examiné dans le point ii).
— Sens de la relation : pour l'agrégation, la relation est orientée du composé vers le composant. Pour la relation de clientèle, plusieurs cas se présentent. Si toutes les instances d'une classe A sont clientes (au travers d'envois de messages) d'instances d'une classe B alors une relation d'utilisation existe entre la classe A et la classe B. Si inversement les instances de la classe B sont clientes des instances de la classe A, alors la relation d'utilisation est symétrique (ou bidirectionnelle). Par exemple connaissant la cassette, on peut demander quel est le film enregistré et connaissant le film, on peut demander quels sont les exemplaires en boutiques.
— Mémorisation : pour envoyer un message à un objet serveur, le client doit connaître l'objet serveur (ou son identité si le modèle dispose de cette notion). L'objet serveur peut être mémorisé soit au sein de l'objet client soit dans l'environnement (le système à objets). Il peut aussi être passé en paramètre d'une méthode.
— Multiplicité (cardinalité) : une instance d'une classe peut faire appel à une ou plusieurs instances différentes d'une autre classe. Ce critère a une influence sur les critères précédents. Par exemple, un même film peut être enregistré sur plusieurs cassettes.

Nerson distingue cinq types de relation d'utilisation [Ner92], issues des cardinalités du modèle à objets et de la dualité association/agrégation : du client vers le serveur (*utilise, a besoin de, a, consiste en*) et du serveur vers le client (*fournit*). Notons aussi qu'une

distinction est parfois faite dans certaines méthodes entre les attributs typés par un type de base et les attributs typés par une classe.

En programmation à objets, la relation d'utilisation restreinte aux attributs de la structure est appelée *dépendance structurelle* (on suppose ici que le type de l'attribut est connu). La structure de l'objet (ses attributs) sert à mémoriser les valeurs simples (dont le type est un type de base), les objets composés et les objets serveurs. En C++, par exemple, une donnée membre (un attribut) a pour type un type C (type de base), une classe (objet composé) ou un pointeur vers une classe (objet serveur). Si la multiplicité est supérieure à 1, on utilise des collections d'objets ou des collections de pointeurs vers les objets. En analyse et conception à objets, les valeurs simples sont des attributs, les objets composants sont reliés par une relation d'agrégation et les objets serveurs sont reliés par une association.

La remarque précédente montre que l'implantation d'une modélisation à objets implique des choix de représentation dans le langage cible. En particulier, si la relation de clientèle est symétrique ou bidirectionnelle, plusieurs implantations sont possibles, qui nécessitent des pointeurs. Pour les mêmes raisons, définir quel est le receveur d'une méthode quelconque est un problème épineux. Par exemple, `emprunter(cassette, boutique, adhérent)` est-elle une méthode de la classe adhérent, de la classe boutique ou de la classe cassette ou des trois ? Le critère de regroupement n'est pas uniquement la structure mais aussi le comportement.

La relation de **dépendance** est un cas particulier d'utilisation qui exprime que pour définir une classe on fait référence à d'autres classes ou à des interfaces.

Relation d'héritage L'innovation la plus importante du modèle à classes est l'**héritage** (relation de généralisation/spécialisation, sous-classe, sous-typage, raffinement). Voici une définition extraite de [Mey97].

Définition C.1 (héritage) *L'**héritage** est un mécanisme permettant de définir une nouvelle classe (la sous-classe) à partir d'une classe existante (la super-classe) par extension ou restriction.*

L'extension se fait en rajoutant des méthodes dans le comportement ou des attributs dans la structure. Par exemple, les conditions d'emprunts des adhérents peuvent être différentes s'il s'agit d'employés du magasin ou d'autres personnes. La classe employé du magasin est une extension de la classe adhérent à laquelle on ajoute un compte boutique. La restriction consiste à réduire l'espace des valeurs définies par la classe en posant des contraintes (conditions logiques à vérifier) sur ces valeurs. Par exemple, supposons un attribut âge dans la classe adhérent et la règle suivante : les jeunes adhérents peuvent emprunter deux exemplaires de plus que les autres adhérents. La classe jeune adhérent est une restriction de la classe adhérent dont la contrainte porte sur l'âge. La restriction correspond plutôt au sous-typage, qui est souvent défini comme une inclusion ensembliste

des valeurs de deux types. On retrouve donc le double aspect “classe = module” et “classe = type de donnée”.

L'héritage induit le **polymorphisme** : une instance de la sous-classe est aussi une instance de la super-classe. Un jeune adhérent reste un adhérent. Cette instance peut donc utiliser sans les définir les méthodes de la super-classe.

L'héritage est à la fois un mécanisme d'inférence permettant de définir des hiérarchies de généralisation/spécialisation (taxonomie) et un mécanisme de construction incrémentale de classes par réutilisation de code. Ou bien, dit autrement, une sous-classe représente soit un sous-type soit une autre implantation du même type (soit les deux bien sûr). Certains modèles autorisent l'héritage multiple.

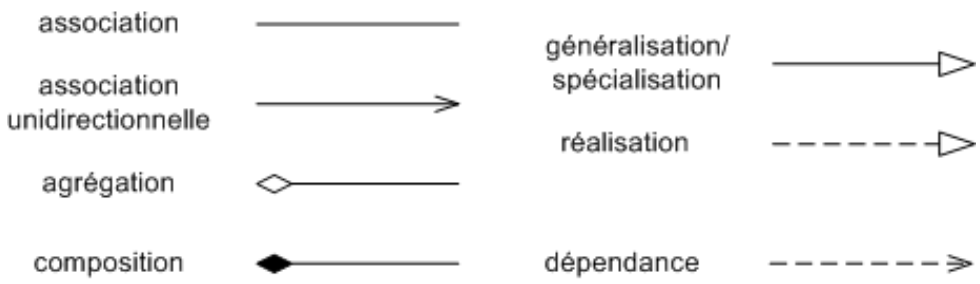
Définition C.2 (généricité) *La **généricité** est un mécanisme permettant de paramétrer des types par d'autres types. L'actualisation (ou instanciation) de paramètres formels de types se fait en fournissant des types définis (des classes ou des types de base). Une classe générique acceptera donc en paramètre d'autres classes (et sous-classes).*

L'exemple classique est celui des collections d'éléments (ensemble d'entiers, de personnes...). Le type des éléments n'influe pas sur les opérations de la collection. Combiné avec l'héritage, la généricité offre un support puissant à la réutilisation.

> L'héritage et la généricité forment les deux cas de vrai polymorphisme (le même code est utilisé quels que soient les types). La coercition (cast, transtypage) et la surcharge (y compris les redéfinitions) sont des cas de faux polymorphisme puisqu'un code différent va être utilisé. Evidemment seul le vrai polymorphisme apporte la réutilisabilité du code.

Notation UML des relations

En UML, les relations s'appliquent aux éléments de modélisation. De ce fait, on peut dire que leur sémantique est surchargée. Toutes ces relations sont *binaires*. Elles peuvent être affinées par stéréotypage.



— La relation d'utilisation est appelée **association**. Par défaut l'association n'est pas orientée, on dit qu'on peut naviguer dans les deux sens de l'association.
— L'association unidirectionnelle est orientée du client vers le serveur seulement.
— Les relations d'**agrégation** et de **composition** sont un cas particulier d'association.

— La relation d'héritage est appelée **généralisation/spécialisation**. L'élément spécialisé a au moins le même comportement que l'élément général.
— La **réalisation** indique qu'un concept réalise un autre concept. Il s'agit d'une variante de la dépendance et de l'héritage, appelée alors **réalisation d'interface**. La notation est ambiguë car la même représentation graphique est employée dans les deux cas.
— La relation de **dépendance** exprime qu'un élément est "sémantiquement" dépendant d'un autre élément. C'est une relation orientée proche de la relation d'utilisation. Elle exprime un lien plus fort que la simple clientèle. Il en existe plusieurs variantes annotées par des stéréotypes : `«use»`, `«merge»`, `«import»`, `«access»`, [Gro11c], p. 160.

Bien que présentées dans le contexte de la conception à objet, il faut noter que ces relations s'appliquent plus largement aux concepts UML, pas seulement aux classes. Par exemple on trouvera des dépendances entre paquetages, de la spécialisation d'états, des associations entre acteurs et cas d'utilisation...

C.1.3 Modèle, vue, diagramme et architecture

Au cours d'un développement, on produit des **modèles** à différents niveaux d'abstraction (le code binaire est le modèle le plus concret !) et pour différents objectifs (selon le destinataire du modèle) (*cf.* Section 1.2.5) page 31. Un **modèle** est une abstraction du système. UML ne gère pas explicitement les modèles car ils sont trop complexes. Du point de vue du métammodèle d'UML, un modèle est simplement un paquetage.

Assez curieusement, la notion de **diagramme**, qui pourtant est la plus communiquée par les utilisateurs d'UML, n'existe pas dans la formalisation d'UML (le métammodèle). Le diagramme est simplement une représentation visuelle, alors d'un point de vue pratique, il s'agit bien d'une ensemble de concepts agencés de manière cohérente. Dans la définition du processus de développement, on travaille avec des modèles et non des diagrammes : un modèle est un ensemble de concepts et non un ensemble de diagrammes. Dès lors on doit s'en remettre aux outils pour définir les combinaisons acceptables de concepts. Les auteurs principaux de la notation proposent une architecture du système organisée en vues.

Chaque **vue** comprend un sous-ensemble des éléments qui appartiennent au modèle et qui forment une perspective du modèle. Les vues sont des projections sur les modèles. Une modélisation de l'architecture en cinq vues a été proposée pour UML, nous discuterons de ce point dans la section C.1.3. Une vue est rattachée à un niveau d'abstraction ou de préoccupation, mais le lien entre les vues reste relativement flou. En effet, UML ne définit pas explicitement de vues cohérentes, mais les outils les implantent. Par exemple, on trouvera la vue des cas d'utilisation, la vue logique, la vue composants et la vue déploiement dans Rational Rose.

Par abus un **diagramme** représente une vue du modèle du système. Par exemple, les diagrammes de classes, de composants ou de déploiement mettent en évidence une structure du système. Les diagrammes états-transitions mettent en évidence l'évolution dynamique des objets. Les diagrammes de séquences, de communication ou d'activité mettent en évidence les aspects fonctionnels et la coordination dans le système. D'autres découpages sont possibles. Ainsi, une architecture de modèle est un découpage en vue des cas d'utilisation, vue logique (ou de conception), vue des processus, vue d'implantation et vue de déploiement.

La variété d'utilisation des concepts est induite par le nombre de combinaisons des concepts au sein d'un modèle. Les **diagrammes** sont des combinaisons cohérentes de concepts. Ces diagrammes décrivent des aspects complémentaires mais non disjoints du système.

Les concepts peuvent apparaître dans différents diagrammes avec parfois des noms différents (par exemple, un paquetage se nomme catégorie dans un diagramme d'analyse et sous-système dans un diagramme de conception). De plus certains concepts sont perçus à des niveaux d'abstraction différents. Par exemple, les opérations des objets sont décrites plus finement par des méthodes à la conception.

La première source de complexité d'UML est donc la combinaison des concepts dans les diagrammes. UML gère en partie ce problème en proposant une "syntaxe", c'est le **métammodèle**. En ce sens, UML est qualifié de langage.

La seconde source de complexité est la combinaison (l'utilisation) des diagrammes dans un modèle. à ce niveau, il n'y a pas de règle précise. Par exemple, un diagramme d'activités, qui décrit un enchaînement d'actions de calcul ou de communication, peut servir à :

— décrire des états d'objets, et plus précisément ce qui se passe dans un état du diagramme états-transitions (*activité-do* en UML 2.X),
— décrire une opération d'une classe (une *méthode*),
— remplacer un scénario pour illustrer un cas d'utilisation,
— décrire un processus métier comme un flot de données [2], souvent en projetant les actions sur plusieurs rôles dans le processus.

Employer un diagramme dans différentes situations induit des règles de modélisation spécifiques ou même des sémantiques spécifiques.

Dans ce qui suit, nous précisions les notions de diagramme et de modèle.

Diagrammes

UML 1.X propose neuf types de combinaisons cohérentes, appelées **diagrammes** représentés dans la Figure C.2.

2. Noter que cette dernière vision est l'évolution la plus importante dans UML 2.

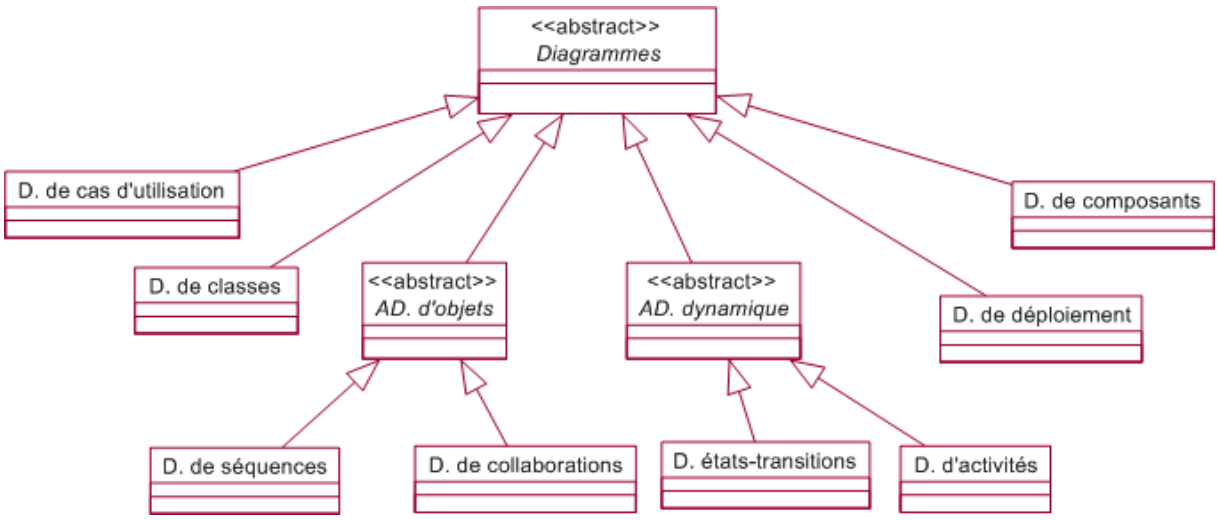


FIGURE C.2 – Notation UML 1 : les diagrammes

— Les diagrammes de *classes* représentent les classes et les relations statiques entre ces classes. Les concepts principaux à ce niveau sont : classe, attribut, opération, visibilité, interface, association, agrégation, héritage, dépendance.
— Les diagrammes d'*objets* décrivent des objets et des liens. Les objets peuvent être actifs et définir leur flot de contrôle. Sur ces liens (réels ou virtuels) circulent des messages (*cf.* Section C.1.1). Les envois de messages sont synchrones ou asynchrones, avec ou sans résultats. Les diagrammes d'objets se retrouvent sous deux formes dans UML :
  — Les diagrammes de *séquences*, qui donnent une vision temporelle des interactions en objets en mettant l'accent sur l'ordonnancement des échanges entre objets ;
  — Les diagramme de *collaboration*, qui donnent une vision spatiale des interactions en mettant l'accent sur les liaisons entre objets.
— Les diagrammes de *cas d'utilisation* (UC - *Use Case*) décrivent les acteurs et l'utilisation du système.
— Les diagrammes *états-transitions* ou diagrammes de *machines à états* modélisent le comportement des objets au cours du temps.
— Les diagrammes d'*activités* décrivent le flot de contrôle interne aux opérations. à grande échelle, ils représentent aussi les échanges entre objets.
— Les diagrammes de *composants* mettent en évidence les composants d'implémentation et leurs relations.
— Les diagrammes de *déploiement* définissent la structure matérielle et la distribution des objets et des composants.

La notation propose des éléments généraux pour enrichir ou structurer les diagrammes : stéréotypes, paquetages, notes, contraintes.

UML 2.0 comprenait 13 diagrammes, comme le montre la FIGURE C.3. Depuis UML

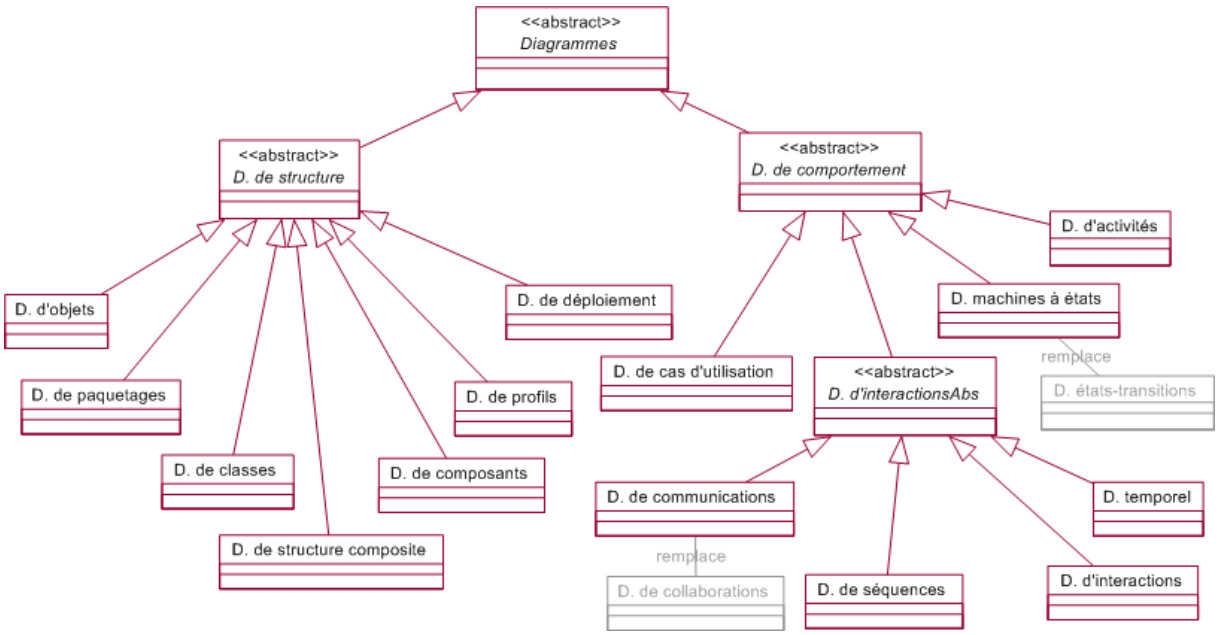


FIGURE C.3 – Notation UML 2 : les diagrammes

2.2, un diagramme des profils a été ajouté[3] à la notation UML 2 qui inclut désormais 14 diagrammes.

Voici les principaux changements par rapport à UML 1.X :

- Les diagrammes d'*Objets* et de *Paquetages* deviennent des diagrammes à part entière.
- Les *composants* sont des spécialisations de *classes*. Cette sémantique, dûe à une factorisation des concepts dans le métammodèle, est pour nous une hérésie tant les concepts de classes et de composants logiciels sont distincts : une classe n'a pas de port, ni d'interface requise explicite. De ce fait les structures composites sont présentes à la fois pour les deux concepts.
- Les diagrammes de *structures composites* placent la relation de composition au premier plan avec une nette orientation composants et architecture de logiciels (ADL). La composition étant transitive, la vue est hiérarchique.
- Le diagramme de *collaboration* devient le diagramme de *communication*.
  NB : par abus, nous utiliserons les deux termes : collaboration et communication. La collaboration devient un élément des structures composites.
- Les diagrammes *d'interaction* (*Interaction Overview Diagrams*) sont un mélange d'activités et de séquences.
- Les diagrammes de *temps* (*timing*) permettent la description d'évolution temporelle de variables, usuelles en automatisme et systèmes temps réels ou embarqués.
- Depuis UML 2.2, une notation a été mise en place pour les *profils*. Cette notion existe depuis UML 1.X mais n'était pas « formalisée ». Un profil est une personnalisation de la notation à un domaine ou une plateforme technique cible. Cette « spécialisation » s'opère via des stéréotypes, des annotations (*tag values*) et des contraintes. Il s'agit

3. `http://www.uml-diagrams.org/profile-diagrams.html`

d'un diagramme de *métammodèle* et non de modèle habituel, il correspond donc à une *sorte de diagramme de classes*.

Par ailleurs, les diagrammes d'activités sont fortement enrichis pour inclure les *Diagrammes de Flots de Données* (DFD) [AV01a].

Pratique

Une bonne pratique est de resteindre le nombre de diagrammes utilisés. Dans la suite du livre, nous ignorerons certains diagrammes ou restreindrons l'usage d'autres diagrammes. Nous en donnons ici les raisons.

1. Le diagramme de profils relève de la métammodélisation et de l'adaptation de la notation aux besoins spécifiques à un type d'application ou de technologie. Nous restons sur l'usage standard d'UML.
2. Le diagramme de paquetages est quelque part inclus dans les diagrammes de classes ou de composants. Ce n'est qu'une question de représentation interne.
3. Le diagramme d'objets est un diagramme de communication sans messages.
4. Un *scénario* sera pour nous un diagramme de séquences simplifié dans le détail des interactions.
5. Les diagrammes de structures composites sont des cas particuliers de diagrammes de classes ou de composants.
6. Les diagrammes d'interaction étant un mélange d'activités et de séquences, leur sémantique est trop ambiguë.
7. Les diagrammes de temps n'ont pas d'intérêt pour nos modèles.

Métamodèle de la notation

La confusion possible entre notation et modèle de la notation est induite en partie par le fait que les règles syntaxique de représentation et de combinaison de concepts de la notation sont définies dans la notation elle-même (métammodèle) d'UML. Un extrait du métammodèle d'UML 1 est illustré par la Figure C.4. De plus, depuis la version 2, le métammodèle, c'est à dire la construction du langage, prime sur l'utilisation de la notation, à tel point qu'un diagramme des profils a été ajouté. Dit autrement, les abstractions de concepts se font sur des proximités syntaxiques et non d'usage, par exemple un composant logiciel est une classe. Le vocabulaire qu'on utilise pour décrire la notation se confond avec le vocabulaire de la notation elle-même. Les manuels de référence [Gro11b ; Gro11c] sont décrits en s'appuyant sur son métammodèle. La structure du métammodèle donne la structure d'explication du langage : on parle d'**élément de modélisation**, de relation, de diagramme. Par exemple, une note (un commentaire) est un élément de modélisation au même titre qu'une classe. On touche donc plus à l'aspect syntaxique de la notation qu'aux concepts.

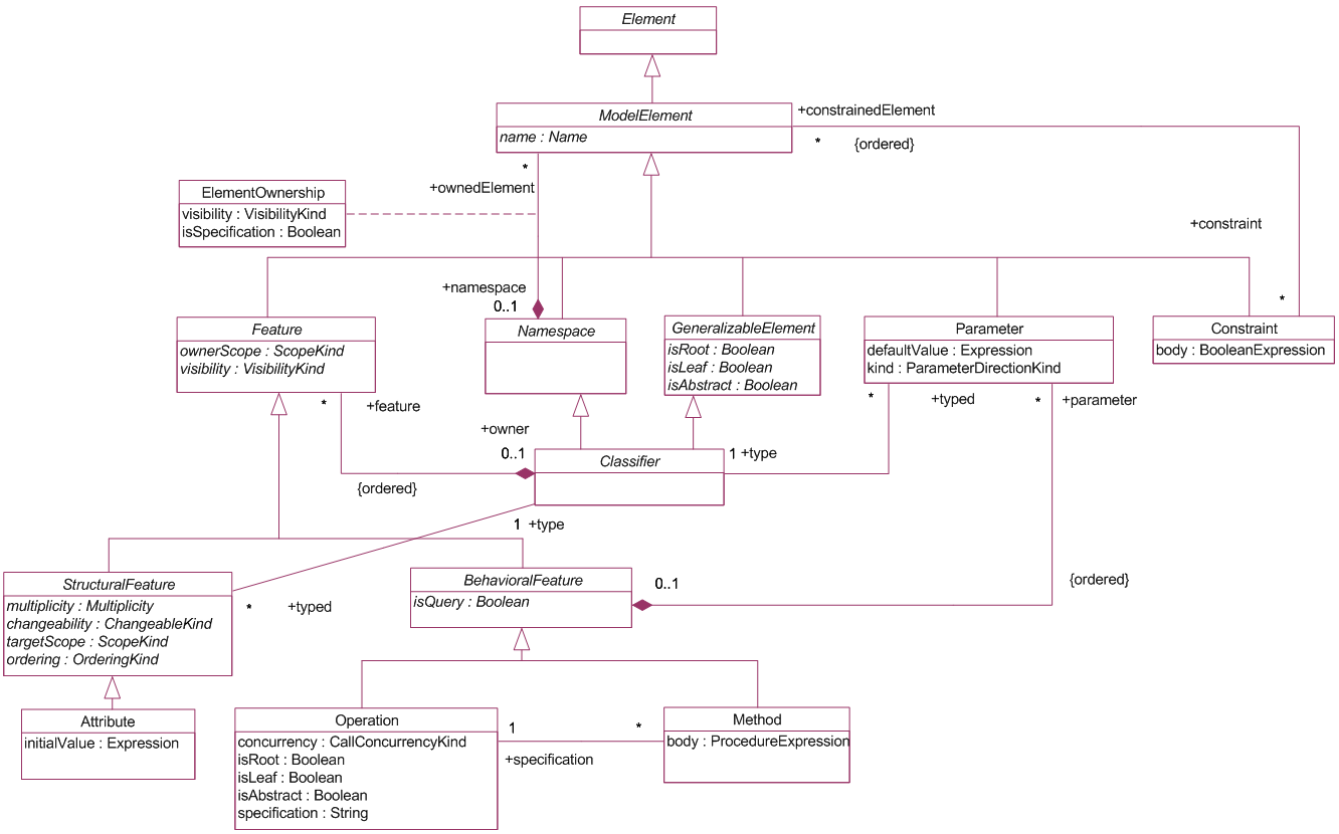


FIGURE C.4 – Notation UML 1.4 : métammodélisation

Un peu de méthodologie et d'abstraction

Dans une vision méthodologique, nous pouvons étudier les relations entre diagrammes et en établir quelques classifications. Cette vision permet

1. de mieux appréhender la pertinence des combinaisons de diagrammes qui forment les modèles produits lors du développement,
2. de mieux cerner les interactions entre diagrammes, notamment pour vérifier la cohérence et la complétude des modèles produits.

La FIGURE C.3 montre que l'OMG établit clairement une dichotomie entre modèles de structure et modèles de la dynamique. L'approche en trois dimensions du système (structure statique, évolution dynamique, calculs fonctionnels), prônée pour le développement à objets dans la méthode OMT [Rum+96] a donc fait long feu. C'est un peu un retour aux sources de Merise, qui séparait données et traitement. Si le diagramme des cas d'utilisation relève clairement de l'aspect fonctionnel, le diagramme d'activité peut couvrir aussi l'aspect dynamique.

Dans [AV01b], nous avions regroupé les diagrammes, qui nous semblaient proches, en quatre familles de modèles : les modèles d'approche (cas d'utilisation et scénarios), les modèles de structure (diagrammes d'objets, de collaboration et de classes), les modèles de la dynamique (diagrammes de séquences, états-transitions et d'activités) et les modèles d'implantation (diagrammes de composants et de déploiement). On peut aussi reclasser ces familles en trois niveaux d'abstraction vis-à-vis du code : les modèles du besoin (ap-

proche), les modèles d'analyse et de conception (structure et dynamique) et les modèles d'implantation et de déploiement. Ces modèles sont ceux produits lors du développement. Dans la FIGURE C.5, les modèles sont classés en niveaux d'abstraction selon leur usage. L'idée est ici de limiter l'emploi des diagrammes à certaines activités du processus de développement.

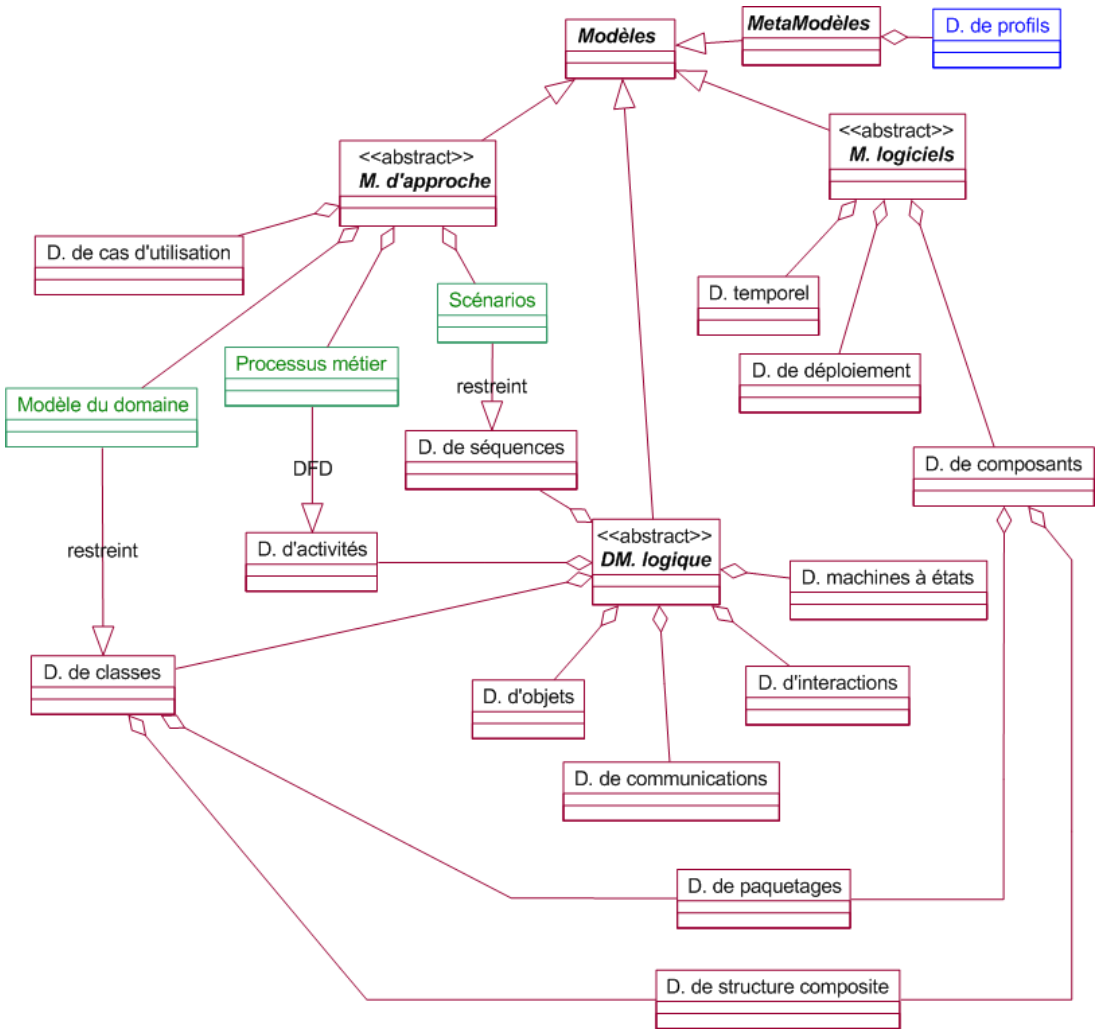


FIGURE C.5 – Notation UML 2 : les diagrammes classés par usage

Les règles de cohérence et de complémentarité des modèles s'écrivent aussi plus facilement aussi lorsqu'on scinde les diagrammes en deux familles : les diagrammes de type (classes, UC, états-transitions, activités, composites...) et les diagrammes d'instances (objets, communications, séquences, déploiement...). Les premiers établissent des règles communes, les seconds représentent des exemples, qui doivent respecter les règles édictées dans les diagrammes de types.

La combinaison cohérente des diagrammes relève non pas de la notation mais du processus et de la méthode de développement. La notion d'architecture y joue un rôle important du point de vue de l'OMG.

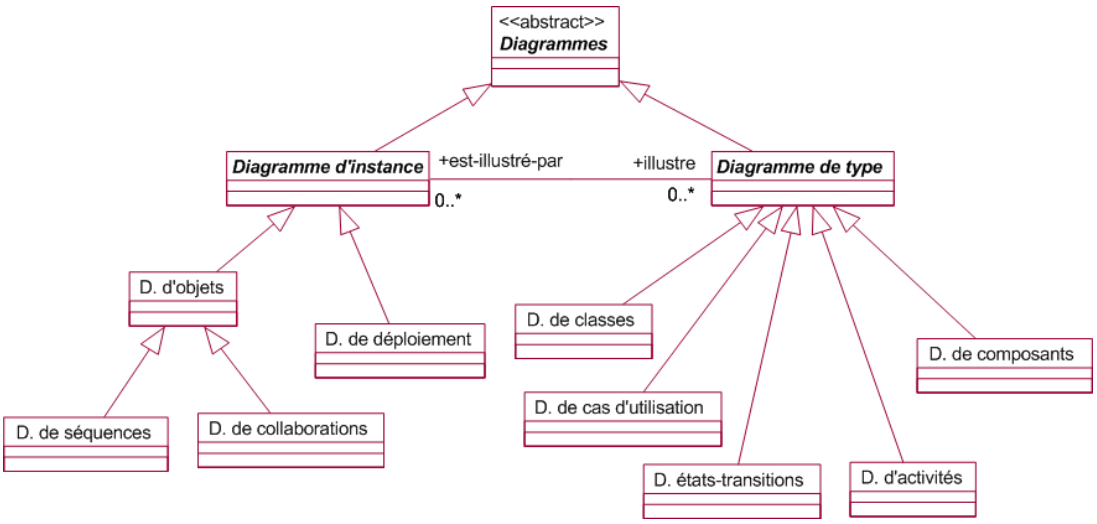


FIGURE C.6 – Notation UML 2 : les diagrammes classés par type/instance

C.1.4 L'architecture logicielle

Le terme **architecture** est comme le terme modèle (chapitre 1 de [AV01a]) un terme générique, employé dans des contextes différents. Dans la littérature sur UML, comme dans le développement du logiciel au sens large, nous avons parfois du mal à discerner le sens précis de ce terme. Ainsi, en modélisation, l'architecture de l'application n'est pas la structure des spécifications. Nous tentons dans cette section de clarifier cette notion pour illustrer son usage dans le contexte d'UML.

L'**architecture d'un système** est relative à la structure ou à l'organisation de ce système. Elle décrit les éléments du système et les liens entre ces éléments. Souvent, par architecture, on entend modèle d'architecture ou architecture de référence, c'est-à-dire une abstraction de la structure, qui s'applique à plusieurs systèmes concrets. Ainsi, une **architecture du logiciel** (ou applicative [GG96]) définit une organisation des éléments du logiciel. L'architecture d'un système informatique décrit les composants du logiciel et du matériel (architectures logique et physique [Ket+98]). Par exemple, « l'architecture client/serveur est un modèle d'architecture applicative où les programmes sont répartis entre les processus clients et serveurs communiquant par des requêtes avec réponses. » [GG96]. Une telle architecture comprendra par exemple les applications clients (sous Windows, OS/2 ou Unix), la base de données et son serveur, et le *middleware*[4]. Cette architecture logicielle sous-entend un réseau « logique », mais une implantation avec trois programmes sur une même machine peut très bien respecter l'architecture client/serveur. Le réseau est lui-même défini par une architecture en couches (les 7 couches OSI). Dans une architecture distribuée, le *middleware* assure la coordination des objets serveurs et des objets clients. Une bonne architecture est garante de la pérennité et de l'évolutivité d'un système informatique.

Dans un système à objets, la structure de base est la coopération entre les objets. L'ar-

4. Ensemble des services logiciels construits au dessus d'un protocole de transport afin de permettre l'échange des requêtes et des réponses associées entre client et serveur de manière transparente [GG96].

chitecture permet en outre d'organiser cette coopération. Plusieurs modèles d'architecture sont proposés par Kettani [Ket+98]. Retenons, en particulier, les architectures en couches qui séparent interface (IHM), application (regroupe parfois interface, contrôle et dispositifs physiques), objets du domaine (métier) et objets techniques (ou d'implantation). Afin de permettre à chacun de définir son architecture, un modèle général a été proposé pour structurer les spécifications en UML. Ainsi, l'approche en 4+1 vues de Kruchten [Kru95] exprime cinq perspectives de l'architecture d'un système informatique. La FIGURE C.7 illustre les différentes perspectives [RJB98].

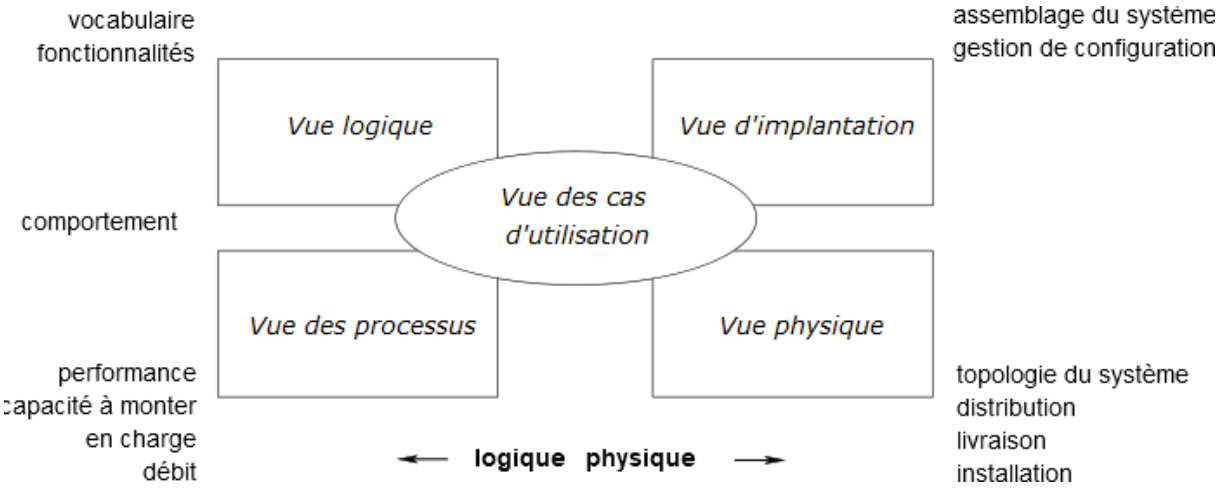


FIGURE C.7 – Notation UML : les vues

Ces vues sont relativement indépendantes. Les cas d'utilisation sont centraux car ils servent tant à l'analyse, la conception qu'au test du logiciel. La vue *logique* (ou vue de conception) représente le domaine du problème (classes, interfaces et communications), c'est la décomposition objet du problème. La vue des *processus* est une projection sur la gestion de la concurrence. La vue d'implantation (ou vue du *développement*) est l'image de la conception logicielle (composants, conception des objets). Enfin, la vue *physique* (ou vue du déploiement) représente la distribution de l'application (topologie matérielle).

L'architecture logicielle est fortement marquée par le type d'application, la technologie employée voire même les traditions de l'entreprise. Le modèle précédent permet d'appréhender la plupart des architectures logicielles : architecture en couche, architecture en peau d'oignon, architecture n-tier, client-serveur, architecture distribuée.

Le TABLE I résume l'emploi des diagrammes selon les vues. La proposition originale de Kruchten [Kru95], notée par un '+' se focalise sur le logiciel. Par extention, d'autres diagrammes sont utilisables notés par un 'x' (utile) ou '-' (accessoire). Nous n'avons retenu que les diagrammes d'intérêt : ceux n'ayant pas une sémantique recouvrante et redondante (interactions, paquetages, composites, interations) ou d'usage de niche (temps, profils) comme précisé à la page 337. La coordination des éléments de modélisation et des diagrammes dans les vues donne une idée de la complexité de l'architecture proposée.

L'architecture logicielle, l'architecture applicative [Pri12] constituent une première couche d'abstraction sur les applications, les programmes et l'architecture technique. L'ur-

	Vue des cas d'utilisation	Vue logique	Vue des d'implantation	Vue des processus	Vue physique
D. de cas d'utilisation	+				
D. d'objets	-	-	x	-	
D. de séquences	-	-	x	messages	
D. de communication	-	-	x	x	
D. de classes		+	-		
D. machines à états		-		x	
D. d'activités	-	-		x	
D. de composants			+	tâches	-
D. de déploiement					+

Tableau I– Croisement des diagrammes et des vues

banisation permet de cartographier et de fédérer les architectures applicatives des systèmes d'information [Lon09 ; UE10 ; RC11]. à plus haut niveau d'abstraction, on trouvera les processus métier [DM04 ; BFGM11], les *Business Process Models* (BPM) et l'architecture métier [DR12] au cœur des préoccupation de la gestion de systèmes d'information [Mor12]. Pour toutes ces visions UML est aussi devenu un langage de référence, notamment avec les diagrammes d'activités pour les processus et les diagrammes de composants et paquetages pour l'urbanisation.

C.1.5 Cohérence et complémentarité entre diagrammes

Dans cette section, nous abordons la notion de description multi-vue cohérente, la correction de modèles, la complétude de modèles. Ce sont des **propriétés de spécification** importantes puisque les modèles sont des entrées d'étapes de développement et codage.

Un modèle UML est un ensemble de modèles hétérogènes, en ce sens que chaque diagramme constitue un modèle à part entière. La sémantique des diagrammes est souvent « assez » cohérente même si UML ne dispose pas de sémantique formelle de sa notation, juste des règles de « bonne formation ».

Dans la littérature et dans la pratique, nous constatons que bon nombre de modèles ne sont pas de bonne qualité, les modèles sont incomplets, les notations pas toujours rigoureuses et correctes, l'interprétation est souvent ambiguë ou liée à un contexte implicite... Ces défauts, nous n'y échappons pas non plus, mais nous essayons de les réduire.

L'objectif ici n'est pas de définir une liste extensive de règles à respecter mais d'initier le lecteur au processus de vérification des propriétés de spécifications, à la fois individuelle et croisée. Il s'agit plus de pratique de vérification par relecture (*pair review*) que de vérification formelle puisque, nous l'avons indiqué précédemment, UML ne dispose pas d'une sémantique formelle et complète de la notation. Les vérifications sont possibles si les diagrammes ont un niveau de détail suffisant.

On considère trois niveaux distincts d'abstraction dans les spécifications :

1. Les modèles d'approche : diagrammes de cas d'utilisation et scénarios.
2. Les modèles logiques : diagrammes de classes, d'objets, de séquences, états-transitions et d'activités en option.

3. Diagrammes de composants et de déploiement.

On peut voir ces niveaux comme des niveaux d'abstractions. Chaque niveau définit un ensemble cohérent de diagrammes. La cohérence dans les cas d'utilisation se résume au fait que les scénarios décrivent des instances de cas d'utilisation. La cohérence du déploiement des composants se résume au fait que les nœuds contiennent des composants. La cohérence entre composants est implicitement liée au code. C'est une vérification très complexe. Seul le niveau intermédiaire peut donner lieu à des vérifications relativement accessibles.

A travers le développement, les niveaux d'abstraction sont connectés par une relation de raffinement. En toute logique il faut donc vérifier la cohérence entre niveaux. La cohérence entre niveau 1 et 2 est difficile du fait du manque de formalisme. On pourra vérifier que les scénarios sont similaires aux diagrammes d'objets. La cohérence entre niveau 2 et 3 est difficile car le modèle de conception comprend des concepts éloignés du modèle logique (bibliothèques, pages web, jsp, frameworks, pilotes de matériel...). En pratique, on utilise plutôt des liens de traçabilité pour mettre en correspondance les éléments de couches différentes. La vérification de cohérence et de complétude inter-niveaux d'abstraction se base alors sur la cohérence de ces liens.

Ce vaste sujet de la vérification de propriétés est détaillé dans [AV13] :

— le chapitre 2pour les bonnes pratiques par diagramme,
— le chapitre 4pour les vérifications d'intégration pour les diagrammes d'approche,
— le chapitre 5pour les bonnes pratiques de structuration du niveau logique,
— et le chapitre 8pour les vérifications unitaires et d'intégration pour les diagrammes du niveau logique et une synthèse sur le sujet.

C.2 Le processus de développement

La **démarche de développement** (aussi appelée le *processus de développement*) définit qui fait quoi, quand et comment.

UML normalise la notation standardisée mais pas le processus, même si des études sur ce sujet sont en cours. La normalisation du processus implique de rationaliser des activités qui varient d'une entreprise à l'autre. Cette ingénierie fait partie du capital de l'entreprise, elle ne souhaite donc pas diffuser son savoir-faire dans les travaux de normalisation.

Il y a actuellement convergence d'idée sur les grandes lignes d'un processus dans la communauté des développeurs :

— Le cycle processus est **itératif** pour les grands projets car les logiciels développés sont souvent très complexes et il est intéressant d'en étudier les grandes parties puis rapidement de valider cette étude par un premier prototype. Les phases itérées sont l'étude du besoin, l'analyse, la conception et le prototypage (réalisation allégée). Une validation du besoin est possible via le prototype : le client affine ainsi sa demande.
— Le processus doit être **incrémental** pour ne pas remettre en question les modèles déjà écrits. Ceci implique une forte modularité des produits et une priorisation des

modules fondamentaux. La cohérence est une propriété implicite du modèle objet, mais le couplage, deuxième composante de la modularité, doit être surveillé.

— Le processus est **basé sur les cas d'utilisation**. Le cas d'utilisation est l'identification parcellaire d'une partie cohérente du fonctionnement du système modélisé. Ceci permet une approche par morceaux d'un problème complexe. Le regroupement des morceaux donne une vue globale du système. Travailler sur des parties facilite la compréhension, la modélisation et la vérification de ces parties. Cela induit aussi assez naturellement un partage simplifié du travail, dans l'hypothèse où l'architecture globale du système aura été bien définie à un moment donné.

— L'**architecture** devient centrale dans la cohérence globale du projet. Plusieurs modèles existent : par couche, distribué, par activités (ex. MVC)...

— La **réutilisation** est un autre élément capital dans le développement et plus encore le développement à objets. Elle sous-entend l'intégration de parties logicielles existantes dans un projet :

  — réutilisation horizontale : délégation de services à d'autres objets existants (ex. bibliothèques d'objets spécialisés),

  — réutilisation verticale : définition de comportement spécifiques de certains objets à partir d'objets existants (héritage),

  — réutilisation complexe combinant les deux, par exemple par instanciation de micro-architectures ou *patterns*,

  — etc.

Si aucun processus normalisé n'existe, les principes communs ont été *formalisés* par un métammodèle appelé *Software Process Engineering Metamodel* (SPEM). Il est défini, comme UML, par une architecture en 4 niveaux M3-M2-M1-M0, comme le montre la Figure C.8, extraite de [DGD05]). On y trouve :

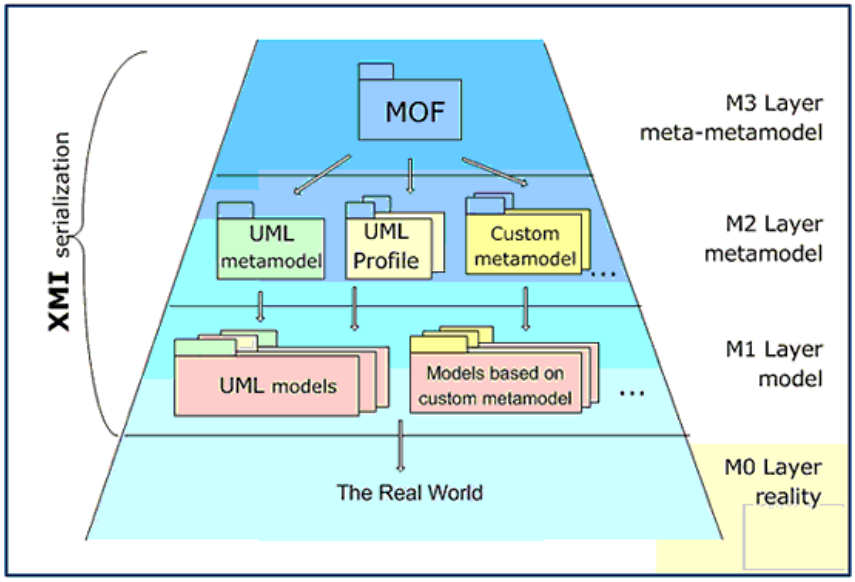


Figure C.8 – Pile des modèles de l'OMG

— *MOF* : le métalangage commun à tous les langages de l'OMG.

— *Process Metamodel* (UPM) : le métammodèle commun aux processus. Les quatre éléments de base pour la modélisation sont :
  — les participants ou rôles (*workers*), le `qui`,
  — les tâches (*activities*), le `comment`,
  — les concepts et productions (*artifacts*), le `quoi`,
  — les activités (*workflows*), le `quand`.
— *Process Model* (RUP, 2TUP, OPEN...) : les modèles généraux des processus,
— *Performing process* : les modèles mis en œuvre de manière opérationnelle sur les projets.

Cela permet ainsi de personnaliser et d'automatiser son processus.

C.2.1 Étapes

Dans la suite, nous reprenons les phases habituelles du processus développement.

1. **Analyse des besoins**. La modélisation du besoin avec les cas d'utilisation : définition des acteurs concernés (acteur primaire qui est à l'origine du déclenchement des cas d'utilisation et acteurs secondaires), définition des cas d'utilisation (des sous-cas liés par `uses` ou `extends`), définition de scénarios illustrant ces cas d'utilisation (cas normaux et exceptions).
2. **Analyse** avec les diagrammes d'*objets*, de *classe*, d'*états/transitions*, d'*activités*, de *communication* et des diagrammes de *séquences*. à partir des cas d'utilisation, des diagrammes de communication ou de séquences (scénarios enrichis) apportent plus de précision quant à la prise en charge des besoins par le système. De nombreux objets sont identifiés qui prennent en charge une partie de la réalisation des besoins. Certains préconisent dès ce niveau la distinction entre objets interfaces (faisant le lien entre le système et son environnement) et objets métiers. Cette distinction convient aux applications de type gestion (une fenêtre de menu prend en charge une communication) mais elle devient plus subtile pour la modélisation de processus réactifs (processus industriel par exemple). Cette partie est relativement complexe dans la mesure où l'intérieur du système commence à être abordé. Des dessins d'interface ou des spécification de mécanismes physiques documentent les modèles UML produits. Des modèles d'objets et de classes/relations partiels sont établis. Les classifications d'objets sont étudiées. Un modèle statique global regroupe et synthétise les différents modèles partiels de classes.
3. **Conception système** (ou préliminaire). Une architecture du système en résulte, qui précise l'organisation logique des logiciels et des matériels. Les diagrammes de classes, de composants et de déploiement sont adéquats à ce modèle.
4. **Conception détaillée**. Après la vision architecturale, on décrit précisément les composants logiciels et les caractéristiques impliqués dans la programmation.
5. **Réalisation** ou prototypage. L'implantation fonde la conception dans les langages de programmation avec les bibliothèques et *framework* techniques supports. La réa-

lisation met en œuvre le déploiement selon la technologie visée dans la conception.

6. **Vérification et validation** (V&V). à chaque transition d'étape, on s'assure de la qualité (des propriétés) des produits (modèles, programmes, documentations).
   — La vérification est la conformité du logiciel avec sa spécification de départ [Gau+96]. Elle inclut tous les moyens utiles à montrer que le programme (ou le modèle) a de bonnes propriétés (correct, complet, fiable, robuste...). Dans un développement de qualité, la vérification se fait dès l'analyse. Les techniques de vérification incluent la **preuve** et le **test**, ce dernier étant le plus employé en pratique.
      — La preuve se fait de deux manières : démonstration de théorèmes dans une logique donnée *theorem proving*, parcours exhaustif des états possibles pour savoir quand une propriété est vraie, *model checking*.
      — Le test cherche à détecter des erreurs à différents niveaux dans le programme (plus rarement dans les modèles). Dans une fonction (une opération, une méthode) ou un module (ou une classe) on parlera de *test unitaire*. Entre modules, on teste les interfaces par le *test d'intégration*. Le *test système* s'applique à l'ensemble.

   La vérification est aussi induite par les revues (*pair programming*) et les inspections du contrôle qualité (assurance qualité).
   — La validation détermine l'adéquation au besoin initial, sa validité [Gau+96]. Le système est valide s'il répond à l'attente des utilisateurs et aux contraintes de l'environnement. L'utilisateur final (ou son représentant à travers la maîtrise d'ouvrage, MOA) joue sa partition ici. La lecture des spécifications, le prototypage et le test sont des techniques pour valider les produits. Le test permet la validation des exigences fonctionnelles (recette) ou non fonctionnelles (test de performance, de montée en charge, de qualité de service...).

   Selon les erreurs rencontrées, on boucle vers la réalisation ou plus haut dans le cycle.
7. **Installation**. L'installation comprend toutes les procédures d'un déploiement à grande échelle.
8. **Maintenance**. La maintenance vise à corriger les erreurs ou améliorer le logiciel, le faire évoluer. La maintenance évolutive comprend des cycles complets.

C.2.2 Approches

Sans méthode unique, chacun adapte la notation à son besoin. Dans la pratique d'UML on constate quatre approches :

— Adaptation de méthodes « classiques » avec restriction ou pas des diagrammes à chaque étape, de l'analyse aux tests d'intégration en suivant un cycle linéaire, en cascade ou en V. Par exemple, certains suivent le processus habituel et général de Merise sous l'angle de la notation UML.
— Utilisation du processus unifié (RUP, 2TUP) élaboré par les concepteurs d'UML

et qui élabore le modèle final par enrichissement progressifs du modèle d'analyse conformément aux principes d'un processus itératif et incrémental, centré sur l'architecture et les cas d'utilisation.

— MDA - *Model Driven Architecture*, approche technique soutenue par l'OMG, qui élabore le modèle final par transformations successives de modèles. Les modèles indépendants des plates-formes (PIM) sont transformés en modèles dépendant des plates-formes (PSM).

— Les méthodes « agiles » (Scrum, *eXtreme Programming* XP, Lean, Puma...), qui travaillent en général sur des cycles courts (*e.g.* les sprints Scrum) avec validation rapide, *on donne la part belle aux programmeurs et aux clients*. Elles s'inspirent de bonne pratique de la programmation à objets (TDD, *pair programming*...), du travail collaboratif et responsable avec souplesse, adaptation et réactivité.

D'autres sociétés proposent d'autres méthodes couplées à leurs outils : OPEN, Objecteering/Softeam, Rhapsody/I-Logix, Catalysis/ICON Computing, Together/Borland...

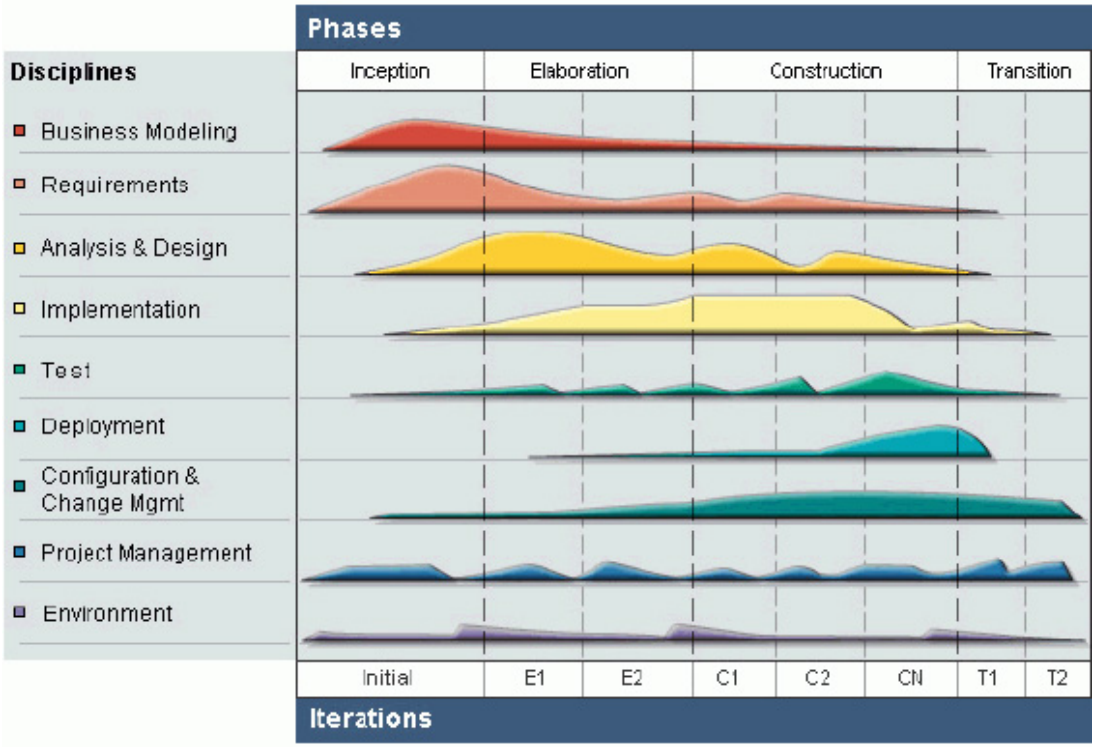
C.2.3 Processus unifié (UP)

Le *processus unifié* n'est pas à proprement parler innovant mais condense les bonnes pratiques actuelles [RJB99]. Il s'agit d'un processus itératif (modèle en spirale) qui reprend les étapes habituelles du développement (des besoins aux tests), les principes du prototypage et du développement incrémental (une structure -une architecture- qu'on complète progressivement), et enfin la *gestion de projet* et le *pilotage* (évaluation, planification, organisation, suivi et approbation). Tous ces points sur la conduite de projets informatiques sont détaillés dans le chapitre 1 de l'ouvrage de Morejon et Rames [MR93]. L'originalité d'UP repose sur l'intégration harmonieuse de ces différentes approches au moyen des cas d'utilisation et sur son adaptation aux techniques récentes du développement (objet, architectures réparties, Web...), la réutilisation et les patrons (*pattern*) ou les architectures à trous (*frameworks*).

Dans le développement d'un système informatique, le processus décrit l'enchaînement des travaux qui conduisent à des résultats. On appelle *produit* le résultat d'un travail, d'une étape dans le développement. Le produit peut être un système informatique mais aussi un ensemble de documents quelconques, des spécifications, des logiciels, des manuels...

Le processus unifié UP [RJB99] se répète à travers une série de cycles formant la vie du système. Chaque cycle conduit à une nouvelle version, c'est-à-dire un produit utilisable par le client. Cette version comprend la description des besoins (cas d'utilisation, besoins non fonctionnels et tests), l'architecture du système et les modèles produits au cours du développement, le logiciel, la documentation... La Figure C.9 montre l'entrelacement des activités de développement et de support dans chaque itération. Elle met aussi en évidence l'effort de développement à chaque étape.

Un cycle se décompose en quatre phases : la préparation, l'élaboration, la construction



source : `http://www.ibm.com/developerworks/webservices/library/ws-soa-term2/`

FIGURE C.9 – Processus unifié

et la transition. Chaque phase peut être perçue comme un projet avec une planification des travaux, des ressources (humaines, techniques, temporelles, financières), des certifications...

La préparation (*inception*) est la phase de lancement du projet, elle établit la faisabilité et le contexte du projet. Durant cette phase, on propose une vision du système (besoins attendus, performances, architectures envisagées...), on définit les risques et les critères d'évaluation, on établit un plan de travail prévisionnel des phases suivants (*Lifecycle objectives*). L'élaboration met au point et valide l'architecture du système, c'est-à-dire qu'à l'issue de cette étape le cadre de développement est bien défini et la faisabilité est assurée. Elle établit l'architecture et la planification contrôlée du projet. Théoriquement, il ne reste qu'à compléter (remplir) l'architecture et tester l'ensemble, c'est l'objectif de la phase de construction. Le résultat est un produit exploitable (*Lifecycle architecture*) qu'il reste à tester en contexte réel. La construction construit un système testable (*Initial Operational Capability*). La phase de transition est la mise en situation réelle du produit (*Product release*), tests complets et finalisation de la version du produit pour le cycle courant. La transition établit le bilan du cycle, capitalise les efforts de développement et prépare les versions ultérieures (les cycles suivants) : éléments à ajouter, à améliorer...

Les phases sont elle-même divisées en itérations et produisent des versions du produit. Chaque itération produit une version du système (un *jalon mineur*) tandis que les phases définissent les grandes étapes du développement (les *jalons majeurs*, qui contrôlent ainsi le nombre d'itérations.

Chaque itération couvre les activités traditionnelles (*workflow*) d'analyse des besoins,

d'analyse, de conception, de réalisation et de test. En fait, les parts respectives de ces activités évoluent selon la phase considérée, les activités en amont sont plus poussées dans les itérations des premières phases, les activités en aval sont plus approfondies dans les itérations des phases ultérieures. Chaque itération donne lieu à un produit résultat. Plus le projet est grand, plus il y a d'itérations dans une phase.

à ces activités correspondent des modèles (cas d'utilisation, analyse, conception, déploiement, implantation et test) décrits avec la notation unifiée, une vue architecturale de ces modèles et des descriptions complémentaires. Nous retrouvons alors la notation UML, qui sert à décrire ces modèles. Nous traçons maintenant les grandes lignes de la corrélation entre le processus et la notation unifiée.

En plus des activités de développement, on trouve des activités support :

— la gestion de configuration & versions,
— la gestion de projet (organisation, risques, planification) et
— l'environnement du projet (support et méthode).

De ce fait UP est une méthode complète pour aborder un projet. Tests et qualité sont pris en compte par le processus.

C.2.4 Processus unifié à deux branches (2TUP)

Le processus *Two Tracks Unified Process* est introduit par P. Roques dans [RV11]. Bien que ce processus n'apporte pas directement de solution sur les étapes cruciales de conception et de réalisation, il nous semble intéressant d'un point de vue pédagogique car il sépare les analyses en deux aspects : la branche fonctionnelle (métier) et la branche technique, comme le montre la Figure C.10 (voir aussi Figure 1.1).

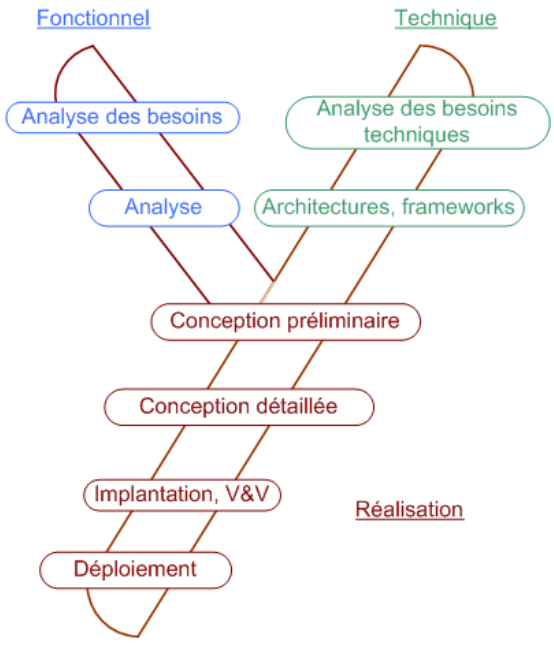


Figure C.10 – Processus unifié à deux branches (2TUP)

Cette dissociation dans les études permet :

1. de rationaliser les ressources par la mise en parallèle du travail des architectes applicatifs (métier) et des architectes techniques (experts),
2. de favoriser la réutilisation à gros grain. Théoriquement on peut interchanger les deux aspects : on place un autre métier sur une même plateforme technique ou bien on change le support technique d'une même application (évolution technique dans la maintenance).

Le processus 2TUP met en valeur les études techniques, qui sont noyées dans les autres cycles de développement. Il met aussi met en évidence le fait que le nœud du problème dans le développement moderne est la conception des applications. La vision 2TUP s'intègre bien avec l'approche MDA dans la mesure où le modèle spécifique (application) est un plongement du modèle indépendant (modèle logique d'analyse) dans une plateforme technique.

C.2.5 Processus MDA

L'approche MDA (*Model Driven Approach*) [KWB03 ; Mel+04 ; Lan05] cible le problème de l'évolution continue tant technique que métier en se focalisant sur les haut niveaux d'abstraction par la création de modèle métier indépendant de l'informatisation (*Computation Independent Model, CIM*) transformés en modèles indépendants des plateformes [Lan05] puis en modèles modèle spécifique à la plate-forme cible (*Platform Specific Model, PSM*).

Le processus réalise ces transformations permettant ainsi d'améliorer — la productivité en réduisant réduire la distance modèle/code comme le montre la Figure C.11, — la portabilité en s'adaptant aux technologies, — l'interopérabilité en établissant des ponts entre domaines, — la maintenance et la documentation par génération automatique.

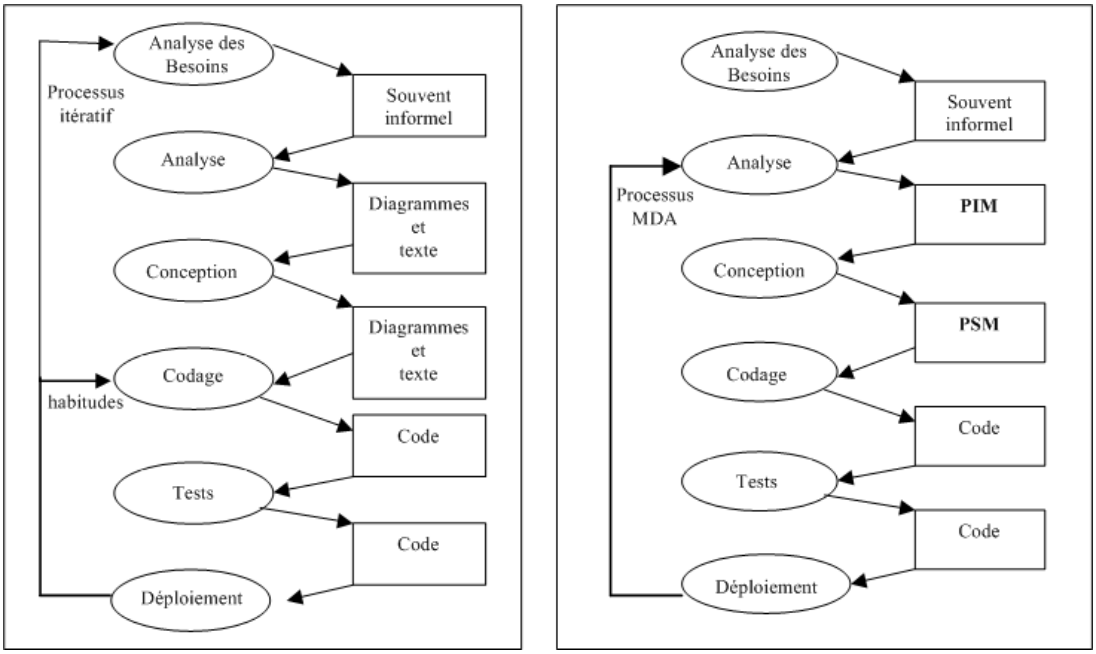


Figure C.11 – Processus de développement classique - processus MDA

Définition (Système) *Le terme **système** correspond à la désignation courante (organisation, personne, système information, programme...).*

Définition (Modèle) *Un **modèle** d'un système est une spéécification de ce système et de son environnemet selon des objectifs donnés (une interprétation).*

Définition (Architecture) *L'**architecture** d'un système est une spécification modulaire du système avec la description des modules et de leurs interactions (connecteurs).*

Définition (Plateforme) *Une **plateforme** est un ensemble de sous-systèmes et de technologies fournissant un ensemble cohérent de fonctionalités décrites par des interfaces.*

On distingue les — platesformes génériques : objet/batch/flots de données — platesformes technologiques : CORBA, J2EE... — platesformes propriétaires : Iona Orbix, Borland Visibroker, IBM Websphere, BEA Weblogic...

Définition (Vue) *Une **vue** (un point de vue) constitue une abstraction selon certains critères de sélection (concepts architecturaux et règles de structuration).*

Le modèle MDA spécifie 3 points de vue : (1) *Computation Independent Viewpoint* : environnement et besoins du système (2) *Platform Independent Viewpoint* : opérations du système, invariant vis-à-vis des plateformes (3) *Platform Specific Viewpoint* : ajoute des éléments spécifiques à une plateforme au PIM. Trois modèles y correspondent :

— *Computation Independent Model* : le modèle selon la vue CIV est sans les détails de la structure du système. Il correspond aussi au modèle du domaine.
— *Platform Independent Viewpoint* : le modèle selon la vue PIV, décrit le système indépendamment de toute réalisation. Par exemple, on utilisera une machine virtuelle comme un ensemble de service invocables.
— *Platform Specific Viewpoint* : le modèle selon la vue PSV, est décrit dans les termes de la plateforme cible *e.g.* CORBA, J2EE.

Définition (Transformation) *Une **transformation de modèle** (un point de vue) convertit un PIM en un PSM selon des règles de transformation. C'est une sorte de patron réplicable.*

La transformation générique est donnée dans la Figure C.12. Ces motifs s'enchaînent dans des processus de transformation.

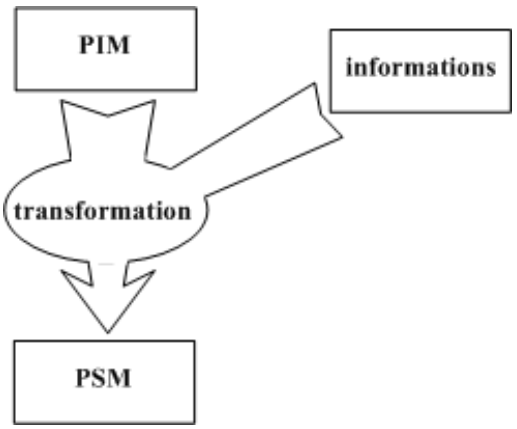


Figure C.12 – Transformation générique - processus MDA

Définition (Processus de Transformation) *Un **processus transformation de modèle** est l'enchaînement d'une suite de transformations pour mettre en œuvre des transformations complexes (le PSM d'une transformation devient le PIM de la suivante). Inversement une transformation complexe est décomposée en transformations plus simples pour mieux maî-*

triser les changements. On peut aussi avoir des transformations en parallèle sur des domaines distincts (mais complémentaires). Le but est de produire des modèles de plus en plus concrets (proches de la plateforme).

Différentes techniques (ou type) de transformations ont été proposées [KWB03], nous en citons quelques-unes : transformations par marquage, via les modèles, transformations via les métammodèles, par pattern, par fusion... Les transformations par métammodèle sont les plus courantes, elles utilisent alors des langages de transformation.

La mise en œuvre implique de disposer de nombreux modèles (langages de modélisation). Les premières applications intégrées se focalisaient souvent sur 3 niveaux d'abstraction (MDA Core) : un modèle UML pour le PIM, plusieurs modèles pour le PSM, chacun relevant d'une technologie particulière (transformations en parallèle + ponts), le code source de l'application. Il n'était pas fait mention de niveaux intermédiaires. Le langage de modélisation peut être un profil UML ou spécifique. Le niveau CIM est ignoré en général car difficilement transformable directement en PIM.

C.2.6 Activités de développement avec UML

Nous donnons un aperçu de l'utilisation de la notation UML dans le processus unifié au travers des activités des itérations. Les activités de support ne sont pas abordées ici.

Modélisation du besoin

La capture des besoins établit les besoins fonctionnels ou non-fonctionnels du système à modéliser et la compréhension de son contexte.

Les besoins fonctionnels sont les services attendus par le demandeur. Les besoins non-fonctionnels regroupent des contraintes (systèmes existants, plate-formes, standards, distribution...) et des critères du résultat attendu (performances, coûts, qualité...). Les besoins sont ensuite consignés sous forme d'un modèle des cas d'utilisation, de glossaires, de descriptions des acteurs, des cas d'utilisation, des interfaces utilisateur et des besoins spéciaux. Essentiellement textuels, les produits de l'activité d'analyse des besoins utilisent les diagrammes de cas d'utilisation, illustrés par des scénarios (diagrammes de séquences réduits au système et aux acteurs externes). Noter que les scénarios sont utiles pour tester les interactions du système et donc fournir des *scénarios de tests*. Pour décrire les traitements de cas d'utilisation, la modélisation de processus se fait via des diagrammes d'activités, éventuellement munis de couloirs pour fixer les rôles. Il s'agit là à notre avis d'une entorse, parfois pratique mais sur laquelle il est difficile d'établir des vérifications formelles. Néanmoins dans l'optique de générer automatiquement les cas de tests, l'étude des graphes de contrôle détermine les différents chemins possibles... et donc la couverture des tests à réaliser.

Le contexte du système est une compréhension de son activité réelle. Elle se traduit sous deux formes : modélisation du domaine (*Domain Model*) et modélisation du métier

(*Business Process Model*).

— Le modèle du domaine est la description technique du domaine d'activité : les termes techniques et le vocabulaire employé, les concepts et leurs relations. Il s'agit essentiellement d'un diagramme de classe qui permet de mieux comprendre de quoi il s'agit.

— Le modèle du métier s'attache à décrire des processus métier. On l'utilise plutôt lorsqu'on travaille sur une organisation, que nous qualifions sommairement d'informatique de gestion. Il s'agit d'une modélisation du système réel, et non du système (informatisé) futur. Ce modèle permet de mieux appréhender ce qui se passe dans le système. Les notations UML utilisées sont essentiellement celles des hauts niveaux d'abstraction, que nous avions classé en modèles de l'utilisateur et modèles de conception dans la section C.1.

Analyse

L'analyse est un raffinement et une structuration des besoins en termes de structure du système (interne au système). Cette structure est une structure logique en terme d'objets (et de classes), qui se situe à un niveau d'abstraction supérieur à l'implantation, aucune hypothèse n'étant faite en ce sens. Le modèle d'analyse peut être perçu comme une répartition (logique) du besoin sur les éléments (essentiellement des objets) du système. Chaque objet ayant des responsabilités vis-à-vis de ces besoins. Il s'agit de *réaliser* les cas d'utilisation. On passe du langage du client à celui du développeur.

Le modèle d'analyse comprend essentiellement des diagrammes de classes et des diagrammes de communication. Les diagrammes de classes décrivent la structure logique du système (une abstraction de ses objets logiques). Les diagrammes de communication décrivent comment les cas d'utilisation sont réalisés par le système. Selon les principes de Jacobson, on sépare nettement les objets de l'interface, ceux du contrôle et de la coordination des activités et ceux de la gestion des informations (entités). L'organisation de ces diagrammes via des paquetages d'analyse et des paquetages de services (et leur dépendances) fournit une première architecture logique du système (*Application-specific layer*, *Application-general layer*).

L'analyse est complétée par la description de besoins spécifiques concernant la persistance, la distribution et la concurrence, la sécurité, la tolérance aux fautes, la gestion de transactions...

Conception

La conception est la détermination du système informatique qui permet de répondre aux besoins mis en évidence dans l'analyse des besoins et structurés dans l'analyse. La conception fait le lien entre l'architecture logique et l'architecture logicielle. Dans une architecture en couche, la conception est grossièrement le lien entre les couches hautes issues de l'analyse et les couches basses provenant de l'environnement d'implantation

(*Middleware layer*, *System-software layer*). C'est à ce niveau que sont mises en évidence les décisions générales d'implantation (architecture logicielle client-serveur, distribué, ou n-tier, environnements de développement et d'interface, communication, persistance, systèmes d'exploitation cibles, applications existantes...). La conception définit un contexte précis et fiable pour l'implantation (*blueprint for the implementation model*).

Lors de la conception sont produits principalement un modèle de conception et un modèle de déploiement. La vue architecturale et des descriptions annexes (contraintes...) complètent le produit. Le modèle de conception décrit la réalisation physique des cas d'utilisation par des diagrammes de classes et d'interaction. Il est structuré en sous-systèmes (paquetages de conception) munis d'interfaces. Les sous-systèmes de service, comme les paquetages de service dans l'analyse, regroupent des classes fournissant un service commun et cohérent. Les classes et leurs relations sont décrites selon le vocabulaire et la sémantique de l'environnement cible (avec des stéréotypes pour préciser la notation). Les méthodes, qui réalisent les opérations, sont décrites plus finement, éventuellement par des diagrammes d'activités. Le comportement des objets actifs est précisé par les diagrammes états-transitions. Les interactions sont définies par des séquences ou des communications. On utilise des diagrammes de composants à ce niveau. Le modèle de déploiement utilise les diagrammes de même nom. On peut affecter des sous-systèmes aux nœuds.

Implantation

L'implantation est la réalisation du modèle de conception et de déploiement à travers les éléments d'implantion (composants, fichiers sources, exécutables...). Il s'agit donc de raffiner le modèle de conception en ajoutant des composants techniques, de réaliser les communications dans un support distribué et persistant puis de procéder au codage et aux tests.

Un modèle d'implantation et un modèle de déploiement sont fournis et enrichis de descriptions telles que la vue architecturale de la conception, le plan d'intégration... Le modèle d'implantation utilise la notation des diagrammes de composants. Il est structuré en sous-systèmes d'implantation munis d'interfaces et reliés par des relations de dépendance.

La traçabilité est assurée par une relation de dépendance entre les éléments du modèle de conception et ceux du modèle d'implantation. Par exemple, une classe de conception est réalisée dans un fichier Java, une table de base de données relationnelle....

Vérification et validation

Revoyons nos définitions de la page 346 sous l'éclairage d'UML. Dans le processus unifié, ces préoccupations sont prises en compte dès le départ, car les cas d'utilisation forment une description du système adéquate pour les revues et inspections des utilisateurs. De plus, le prototypage rapide permet un retour « réel » et rapide des futurs utilisateurs. Les inspections peuvent aussi porter sur les modèles du métiers, les modèles du domaine et les

modèles d'analyse car la notation UML utilisée reste limitée et générale. Le prototypage de modèles UML n'est effectif que pour la sous-classe de langages UML exécutables, qui permettent d'animer les modèles, qui doivent être décrits finement.

Le manque de sémantique complète d'UML fait que la vérification est principalement basée sur le test dans la pratique du développement. La seule vérification formelle proposée par les outils porte sur la bonne formation des diagrammes (contrôle de syntaxe et de type). La vérification de modèles UML nous paraît indispensable, notamment pour vérifier la cohérence entre diagrammes ; elle est détaillée dans le chapitre 6 de [AV13].

Le *test* vise à vérifier le résultat de l'implantation. L'activité de test consiste en une planification et une définition des objectifs de test, la réalisation puis l'analyse des tests, des tests unitaires à la recette et aux tests de performance ou de montée en charge en passant par les test d'intégration et tests systèmes. Il en résulte une activité de correction de l'implantation (et des modèles associés) et un retour sur les tests (sans régression).

Les produits du test sont spécifiques à cette activité : modèle des tests, planification et évaluation des tests, traitement des erreurs. Le modèle de test est un ensemble de cas de test, de procédures de test et de composants de test. Les cas de test correspondent aux cas d'utilisation et à leur réalisation. Les procédures de test décrivent le test d'un ou plusieurs cas de tests. Les composants de test sont des composants logiciels qui automatisent les procédures de test. Le test pour UML fait l'objet d'un ouvrage [Gro04].

L'ensemble de la notation est utilisé au sens où on se réfère aux modèles des activités précédentes pour réaliser les tests. Mais il n'y pas a priori de diagramme UML dédié à cette activité, même si on peut tracer des diagrammes de cas d'utilisation, des diagrammes de séquences ou organiser les programmes en composants.

C.2.7 Croisement niveaux d'abstraction et processus

à travers le processus unifié, on se rend compte que certains principes comme le développement sans couture ne sont pas « purs ». Il nous semble sage en effet, pour éviter des confusions induites par la complexité de la notation, de ne pas utiliser tous les diagrammes à tous les niveaux et de favoriser telle notation pour tel type d'activité. Ainsi, on a une vision plus claire des niveaux d'abstractions, chers à Merise. Dans UML, on distingue aussi trois niveaux : le niveau externe (ou utilisateur, n'existe pas dans Merise), le niveau logique (conceptuel et logique dans Merise) et le niveau physique (plus développé que dans Merise).

Le niveau externe est celui de l'utilisateur (cas d'utilisation, et scénarios provenant essentiellement de la méthode Objectory [Jac+92]). Le niveau logique est celui de la description abstraite du système (il correspond historiquement aux notations issues de l'analyse à objets, et fortement inspirées de la méthode OMT [Rum+96]). Le niveau physique est celui de la description concrète du système (il correspond historiquement aux notations issues de la conception à objets, et fortement inspirées de la méthode de Booch. On retrouve les trois piliers de la notation UML.

En plus de la meilleure visibilité des activités à réaliser en fonction des notations, il est évident que la vérification des produits (des spécifications) est facilitée s'il y a moins de diagrammes à confronter. Il n'en reste pas moins que l'articulation des modèles doit mettre en évidence la continuité du processus et la source des décisions (traçabilité des concepts et des décisions).

C.3 Les outils

Nous proposons ici quelques pointeurs sur les outils et documentation d'UML. La liste est loin d'être exhaustive, d'autant plus qu'elle évolue en permanence. Pour construire les modèles nous avons utilisé les outils Rose 98 de Rational Software Corporation (repris par IBM en 2003) [5] et StarUML, une plateforme libre pour UML/MDA [6]. L'outil Visio 2002 de Microsoft permet de retoucher certains diagrammes ; Visio propose aussi des feuilles de style pour UML 2 et SysML.

De nombreux AGL proposent maintenant la modélisation avec UML. Ces outils permettent l'édition, des vérifications et la génération de code dans différents langages de programmation à objets (principalement les squelettes des classes). Voici quelques adresses d'outils et de comparatifs d'outils sur UML :

- Liens *Developpez* sur les outils UML
  `http://uml.developpez.com/telecharger/index/categorie/449/UML`
- Liens *Wikipedia* sur les outils UML
  `http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools`
- UML Tool Reviews & News
  `http://www.uml-tools.com/`
- Rational Rhapsody (IBM) `http://www-01.ibm.com/software/awdtools/rhapsody/`
- Rational Rose `http://www-142.ibm.com/software/products/fr/fr/enterprise/`
- WinDesign (français) `http://www.win-design.com/fr/WinDesign.htm`
- StarUML (libre) `http://staruml.sourceforge.net/en/`
- BOUml (libre) `http://www.bouml.fr/`
- ARGO/UML (libre) `http://argouml.tigris.org/`
- Modelio (libre, MDA) `http://www.modeliosoft.com/fr.html` et
  `http://modelio-open.sourceforge.net/`
- Papyrus for UML (libre, MDA) `http://www.modeliosoft.com/fr.html`
- Topcased (libre, MDA) `http://www.topcased.org/`
- Objecteering (softeam fr) `http://www.objecteering.com/`
- Obeo Designer + Acceleao (Obeo fr MDA) `http://www.obeodesigner.com/`
- Objecteering `http://www.softeam.fr/`
- Aonix StP/UML `http://www.aonix.com/`
- Outil de dessin multi-formalisme Visio de Visio Corp `http://office.microsoft.com/en-us/visio/`
- Visual Paradigm for UML de Visual Paradigm International `http://www.visual-paradigm.com/product/vpuml/`

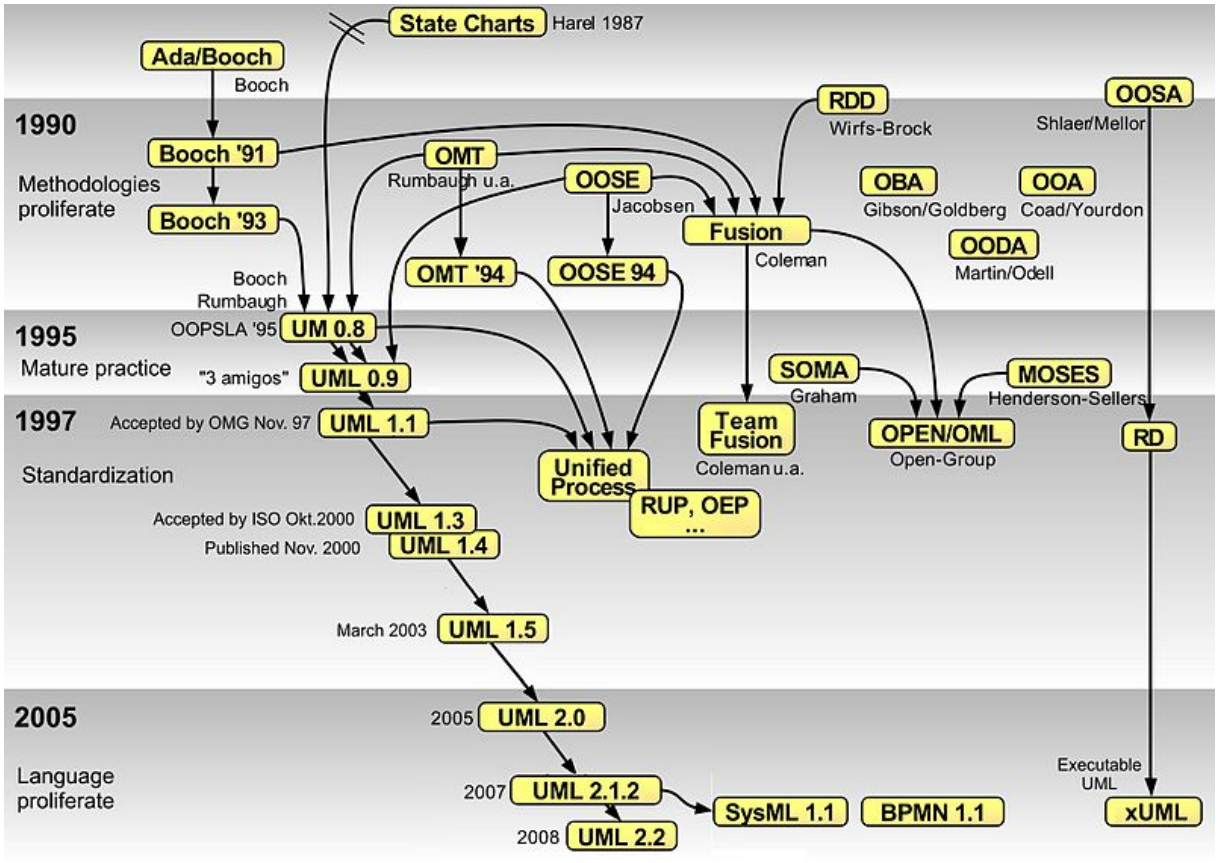
5. `http://www-03.ibm.com/software/products/fr/enterprise/`
6. `http://staruml.sourceforge.net/en/`

C.4 Conclusion

Le développement de logiciels et de systèmes complexes implique d'utiliser des méthodes de développement comprenant une philosophie sous-jacente, des langages et notations, un processus guidant le déroulement à travers des étapes précises et des outils pour automatiser une partie des tâches. La philosophie en vogue est le développement orienté objet avec de la persistance de données via des bases de données relationnelles et une présentation par des clients web, y compris pour des usages mobiles. Au fil du temps, UML s'est imposée comme LA notation standard, acceptée tant par les utilisateurs et développeurs d'applications que par les fournisseurs d'outils de développement. Le processus n'est pas encore unifié, il ne le sera sans doute jamais car il est le savoir-faire même des entreprises de développement. Il varie beaucoup d'une organisation à l'autre même si les principes de bases sont très similaires (les activités de développement, les acteurs, les produits). Actuellement deux approches s'imposent : d'une part les processus lourds tels que le RUP ou le 2TUP (à variante près d'une organisation à l'autre), d'autre part les approches agiles basées sur les « bonnes pratiques » jugées efficaces et productives. L'approche MDA ou MDE, basée sur un outillage par transformation de modèles, pour produire le code final est très active du point de vue de la recherche et l'outillage progresse rapidement. Elle est complètement fondée sur la notation et permet de faire vivre de manière cohérente code et modèles (documentation). Ceci étant, l'offre des outils reste disparate et les solutions intégrées difficiles à appliquer à large spectre. C'est sans doute ce qui nous amène à la création de nombreux clones d'UML, les profils et autres langages dédiés *domain specific languages* (DSL) qui proposent des notations adaptées au besoin (le type d'application ou de plateforme technique). La Figure C.13 montre l'évolution historique des principaux standards autour d'UML à l'OMG. Cela reste encore dans le domaine de l'investigation et de la recherche et développement sur les outils de développpment.

SysML[Wei08], un profil dédié à l'ingénierie système, est devenu un standard à part entière.

Lors de la conférence MODELS 2011, Jon Whittle a rapporté les résultats d'une enquête sur la pratique de l'*ingénierie des modèles* [Whi11]. Il apparaît que si plus de 80% des praticiens utilisent UML et 15% utilisent SysML, 40% d'entre eux utilisent un DSL « maison ». Ces chiffres montrent à la fois qu'UML s'est imposé mais que la sémantique incomplète et surchargée des concepts ne permet pas une interprétation qui convienne à tous.



source : `http://www.start2cloud.com/process-modelling.aspx`

FIGURE C.13 – Vue historique des langages et profils d'UML

TABLE DES FIGURES

INDEX

Nantes Université

Titre : Modélisation rigoureuse au service du logiciel et des systèmes

Mot clés : Génie logiciel, Modèles, Systèmes, Vérification, Méthodes formelles, Systèmes d'information, Compétences

Résumé : Le logiciel présente le paradoxe d'ouvrir un champ immense de possibilités et de rester assez empirique et artisanal dans son développement. Nul doute qu'il soit le seul bien-service dont la maintenance génère des affaires aussi juteuses et les échecs si coûteux. Les travaux abordés dans ce document concernent le génie logiciel et son application. Nous nous inscrivons ici dans une vision modèle avec le cycle vertueux de construction (modélisaton, vérification de propriétés, exploitation) pour produire qualitativement du logiciel de qualité. La vérification de propriétés, telle que la sûreté de fonctionnement ou la sécurité, est fondamentale à l'établissement du contrat de service logiciel. Le logiciel étant un objet vivant dans un contexte mouvant, sa construction doit suivre et s'adapter en permanence. Nous proposons un processus pour cela, fait d'adaptation du contexte technologique et d'évolution du besoin. D'un point de vue applicatif, nous nous intéressons aux systèmes d'information d'entreprise, un écosystème complexe qui entremêlent pléthore de logiciels qu'il faut faire fonctionner ensemble, et aux systèmes de production industriels, qui y ajoutent la complexité des systèmes cyberphysique. Dans de tels cas, on comprend qu'une vision systémique s'impose pour gérer la complexité. Le document se termine sur une réflexion des compétences nécessaire pour diriger des recherches.

Title: Rigorous Modelling for Software and System Engineering

Keywords: Software Engineering, Models, Systems, Verification, Formal Methods, Information Systems, Skills

Abstract: Software presents the paradox of opening up an huge field of potentials while remaining with empirical and craft development. No doubts this is the only good-service with such a lucrative business maintenance and so costly failures. The work discussed in this document deals with software engineering and its application. We embrace a model driven vision with a virtuous construction cycle (modelling, verifying properties, exploitation) to produce qualitatively quality software. The verification of properties, *e.g.* operational safety or security, is fundamental to establish solid software service contract. Software being a living object in a changing context, its construction must continuously follow and adapt. We propose both technology adaptation and requirements evolution. In practice, we are apply to business information systems, a complex ecosystem that weaves a plethora of interacting software, and to industrial production systems, which adds on the cyber physical system complexity. In such cases, a systemic vision is required to manage complexity. The document ends with a reflection on the skills needed to conduct research.