Elaboration de processus de développements logiciels spécifiques et orientés modèles : application aux systèmes à évenements discrets

Thomas Collonvillé

Université de Haute-Alsace

École Doctorale Jean-Henri Lambert - ED 494

THESE DE DOCTORAT

Présentée pour l'obtention du titre de

Docteur de l'Université de Haute-Alsace

(arrêté ministériel du 30 mars 1992)

Spécialité Génie Informatique

par

Thomas COLLONVILLÉ

Elaboration de processus de développements logiciels spécifiques et orientés modèles – Application aux systèmes à événements discrets

---

Soutenue publiquement le 08/10/2010
devant la commission d'examen composée de

Mme Isabelle BORNE	Professeur à l'Université de Bretagne Sud	**Rapporteur**
M. Jean-Pierre BOUREY	Professeur à l'Ecole Centrale de Lille	**Rapporteur**
M. Hervé PANETTO	Professeur à l'Université Henri Poincaré – Nancy 1	**Président**
M. Jean-Marc PERRONNE	Professeur à l'Université de Haute Alsace	**Co-encadrant**
M. Bernard THIRION	Professeur à l'Université de Haute Alsace	**Directeur de thèse**
M. Laurent THIRY	Maître de Conférences à l'Université de Haute Alsace	**Co-encadrant**

Laboratoire Modélisation Intelligence Processus Systèmes (MIPS)-EA2332

Remerciements

Je voudrais remercier les membres du jury pour m'avoir fait l'honneur de participer à ma soutenance de thèse et pour l'intérêt qu'ils ont porté à mes travaux. Je remercie également Mme Isabelle Borne et M. Jean-Pierre Bourey pour avoir bien voulu accepter la charge de rapporteur ainsi que M. Hervé Panetto pour avoir accepté d'examiner ce travail de thèse.

Je tiens à témoigner ma profonde gratitude envers mon directeur et mes codirecteurs de thèse M. Bernard Thirion, M. Laurent Thiry et M. Jean-Marc Perronne, pour m'avoir fait confiance durant les années passées au sein du laboratoire MIPS ; ils ont su m'encadrer, me conseiller et me guider tout en me laissant bénéficier d'une grande liberté. Un grand merci pour leurs conseils pertinents et éclairés qui m'ont permis de mieux structurer mes idées et j'espère aussi, de mieux les retranscrire.

Je ne voudrais pas oublier les nombreuses personnes qui, dans l'ombre des doctorants, contribuent à l'avancement de leurs travaux et sans qui de nombreux résultats n'auraient jamais pu voir le jour. Merci aussi aux étudiants avec lesquels j'ai eu l'occasion de travailler et qui m'ont permis de découvrir le monde de l'enseignement.

Je souhaiterais aussi remercier tous ceux qui ont contribué à rendre ces trois années intéressantes tant sur le plan scientifique que sur le plan humain, Cyrille Petitjean, Alban Rasse, Charles-Georges Guillemot et les autres membres du laboratoire.

Enfin, je remercie, pour sa patience, Mlle Christine Mallet qui a su me soutenir durant ces années.

Merci à tous.
Thomas Collonvillé
Mulhouse, le 24 juin 2010

Résumé

Le développement de systèmes logiciels implique généralement différents langages pour modéliser l'organisation des composants d'une application, leur comportement, les propriétés désirées, etc. S'il existe des modèles de processus décrivant les différentes activités pour passer d'une spécification à une réalisation (par exemple, RUP), il n'existe cependant pas de processus général, dirigé par les modèles et expliquant comment relier de façon rationnelle les langages et les activités. Par ailleurs, l'Ingénierie Dirigée par les Modèles (IDM) propose des concepts et des outils pour spécifier et combiner différents langages. Pour cela, l'IDM introduit les concepts de méta-modèles pour spécifier des langages (de modélisation), et de transformations de modèles pour mettre en relation les métamodèles. Un exemple de métamodèle est donné par le standard SPEM (Software Process Engineering Metamodel) proposé par l'Object Management Group, et sert de langage de modélisation de processus de développement logiciel.

Dans ce contexte, la thèse propose de tirer profit des éléments précédents pour élaborer des processus de développement, spécifiques et orientés modèles, adaptés au développement de familles d'applications relevant d'un même domaine. Plus précisément, la thèse propose un schéma conceptuel, dérivé du schéma conceptuel de SPEM, dans lequel des activités d'un processus peuvent exploiter des méta-modèles et peuvent être des transformations manipulant des modèles spécifiques. Partant de ce schéma, les constituants essentiels d'un processus spécifique sont révélés et mis en relation de manière statique et dynamique. L'identification de ces constituants et de leurs relations conduit à proposer un guide méthodologique permettant d'aborder l'ingénierie de ces processus spécifiques afin qu'ils soient dirigés par les modèles et outillés. Les intérêts de l'approche proposée résident dans une meilleure capitalisation des connaissances et une réduction des efforts de développement. En effet, les concepts spécifiques utilisés par une famille d'applications relevant d'un même domaine, sont exprimés au sein de langages spécifiques/méta-modèles qui sont outillés à l'aide de l'outillage mis à disposition par la communauté IDM. Après avoir présenté ces travaux concernant l'ingénierie de processus de développement spécifiques, la thèse propose d'élaborer de tels processus spécifiques pour des applications logicielles relevant du domaine des Systèmes à Evénements Discrets (SED) et intégrant des activités de vérification de modèles ou de synthèse de superviseurs. Ceci permet de montrer les intérêts et les limites de la proposition en reliant les langages nécessaires pour passer d'une spécification à une réalisation validée, en profitant à la fois des théories spécifiques et des outils spécifiques de ce domaine. Afin de valider l'ensemble de la proposition deux applications intégrant concurrence et distribution sont présentées.

Acronymes et abréviations

AADL	Architecture Analysis and Design Language
ALM	Application Lifecycle Management
API	Application Programming Interface
ATL	Atlas Transformation Language
C2M	Transformation de représentation syntaxique vers modèle abstrait
CIM	Computer Independent Model
CSP	Communicating Sequential Processes
CTL	Computation Tree Logic
DestKit	Discret Event System Tool Kit
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
EPF	Eclipse Process Framework
EPM	Entreprise Project Management
FCO	First Class Object
FSP	Finite State Processes
FSM	Finite State Machine
GME	Graphical Modeling Environment
GMF	Graphical Modeling Framework
GOPRR	Graph Object Property Relation Role
IDE	Integrated Development Environment
IDM	Ingénierie Dirigée par les Modèles
J2ME	Java 2 Micro Edition
KM3	Kernel Meta-Meta-Model
LTL	Linear Temporal Logic
LTS	Labelled Transition System
LTSA	Labelled Transition System Analyser
M2M	Transformation de modèle abstrait vers modèle abstrait
M2C	Transformation de modèle abstrait vers représentation syntaxique
MFC	Microsoft Foundation Class
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MGA	MultiGraph Architecture
MIC	Model Integrated Computing
MIPS	Modélisation Intelligence Processus Systèmes
MIPS	Model-Integrated Program Synthesis

MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PDA	Personal Digital Assistant
PDM	Platform Description Model
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
RAD	Rapid Application Development
RUP	Rational Unified Process
SED	Systèmes à Événements Discrets
SPEM	Software & Systems Process Engineering Metamodel
SQL	Structured Query Language
SysML	Systems Modeling Language
UML	Unified Modeling Language
UP	Unified Process
USPM	Unified Software Process Metamodel
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XP	eXtreme Programming

Table des matières

Chapitre 1

Introduction

1.1 Contexte et Problématique

L'informatique tend à une omniprésence des systèmes d'information et de communication [MS03a, EHS06]. Le développement de ces systèmes à forte composante logicielle doit répondre à de nombreuses contraintes comme par exemple la possibilité de réaliser une multitude de tâches de façon concurrente ou encore de proposer un nombre croissant de fonctionnalités ou services.

Ces systèmes à forte composante logicielle tendent à se décliner dans des contextes de production et d'utilisation très variés. Ils sont intégrés dans les systèmes de gestion avec par exemple les banques en ligne ou les sites d'e-commerce ou au sein de systèmes embarqués (automobile, avionique, ferroviaire, etc.).

La diversité des systèmes logiciels implique alors la prise en compte de nombreux types de propriétés. Selon Ricardo Sanz [SA03], les systèmes logiciels peuvent s'organiser en différentes familles selon qu'ils sont :

Hétérogènes – la réalisation des systèmes logiciels demande l'utilisation de langages ou de plateformes différentes mais complémentaires.

Réactifs – les systèmes doivent répondre instantanément aux stimuli externes auxquels ils sont soumis.

Concurrents – les systèmes sont composés généralement de plusieurs éléments individuels communicants et s'exécutant en parallèle.

Critiques – les systèmes doivent, dans certains cas où la vie des personnes peut être menacée, être capables de fonctionner malgré la survenue d'événements imprévisibles menant à d'éventuelles erreurs.

Embarqués – les plateformes utilisées ont des ressources limitées.

Distribués – le déploiement des applications est réalisé sur des supports et en des lieux différents.

Temps-réels – les contraintes de performance en termes de temps d'exécution sont primordiales.

Dans le cadre plus spécifique des systèmes logiciels embarqués, Thomas Henzinger et Joseph Sifakis [HS06] constatent que "ces systèmes sont désormais devenus des systèmes de grande envergure et sont de plus en plus sensibles à la moindre erreur". Le processus de développement pour ces systèmes est donc devenu un élément important à considérer et à améliorer.

De plus, sous la pression du contexte économique, les développements de ces systèmes logiciels doivent être de plus en plus rapides et efficaces, [Boe81]. Pourtant, les facteurs d'échecs dans l'aboutissement des projets de génie logiciel sont multiples [Kru08]. Une analyse sur l'aboutissement des projets de développement logiciels [EEK08, Sta95, Ext01] pose comme constat que seulement 25% des projets aboutissent dans les temps et les coûts imposés.

Qualitativement, le développement des systèmes logiciels est amené à suivre deux types d'objectifs [Som06]. Le premier type d'objectif, appelé *best effort* (meilleur effort ou production au mieux), a pour contrainte de développement la production et la livraison d'un produit opérationnel en un temps et un coût de production minimal. A l'inverse, le second type d'objectif, appelé *hard QoS* ou service garanti, va mettre la priorité sur les contraintes de fonctionnement où les erreurs et les défaillances sont proscrites. Bien qu'antagonistes, ces deux types d'objectifs sont souvent associés, conduisant ainsi à de multiples contextes de développement logiciels.

Pour réduire les efforts de développement et avoir une meilleure maîtrise du processus de développement, une approche possible consiste à améliorer l'information contenue dans les modèles utilisés par les processus de développement. Ces modèles relèvent d'un ou plusieurs des domaines suivants :

1. Le domaine *métier* ou domaine d'application du logiciel à concevoir.
2. Le domaine de l'*ingénierie logicielle* qui supporte à la fois la conception logicielle et les solutions techniques pour sa mise en œuvre.
3. Le domaine de la *gestion de projet* qui couvre l'élaboration et la mise en œuvre d'un processus de développement.

L'approfondissement des connaissances dans le domaine métier est l'élément principal servant à améliorer les modèles métier et leur prise en compte dans la conception logicielle.

Pour l'ingénierie logicielle, différents modèles de processus peuvent être employés avec, par exemple, le cycle en V ou les méthodes de développement agiles comme Unified Process (UP), [Som06]. Classiquement, un cycle de développement va débuter par la définition d'un cahier des charges suivi par une conception/réalisation intégrant les aspects métiers et les solutions qui en permettent la mise en œuvre sous forme logicielle, pour finir avec des tests montrant que le système réalisé est conforme au cahier des charges.

Ces cycles de développement fournissent un environnement plus rationnel à la conception, ils ne proposent cependant pas de moyens pour gérer la complexité liée à la manipulation des modèles ou pour prendre en compte et/ou intégrer les langages de modélisation utilisés.

Pour la réalisation de systèmes complexes, la modélisation devient primordiale. Cette

modélisation repose généralement sur différents (types de) modèles et pouvoir les manipuler, les échanger, les intégrer ou les transformer plus simplement permettrait de réduire les efforts et les coûts de développement. Pour améliorer cette situation, l'Ingénierie Dirigée par les Modèles [FEB06] propose à travers les concepts de *méta-modèle* et de *transformation* de modèles les moyens d'opérationnaliser la création et la manipulation des modèles. Un *méta-modèle* est une description plus ou moins abstraite d'un langage de modélisation utilisé pour décrire un système. Sous la forme de règles définies au niveau des méta-modèles, les transformations de modèles permettent de mettre en relation des modèles. En particulier, des transformations de modèles peuvent servir à projeter un modèle abstrait vers une représentation textuelle, avec pour conséquence la possibilité de bénéficier des outils utilisés dans un ou plusieurs domaines. Ainsi, l'Ingénierie Dirigée par les Modèles apparaît comme une solution pour spécifier, gérer et utiliser des modèles issus de domaines variés. De plus, elle permet la capitalisation des connaissances en fournissant la possibilité de réutiliser les modèles dans des développements différents.

Afin de pouvoir tirer profit des avancés scientifiques et technologiques relevant du domaine de l'Ingénierie Dirigée par les Modèles, il est nécessaire de mettre en place des processus de développement spécifiques orientés modèles. Ce qui pose la question : Quels processus, avec quels modèles ?

1.2 Aperçu de la contribution

Pour tenter de répondre à la question précédente, le travail présenté propose un modèle générique de processus, devant faciliter la manipulation et l'échange de modèles entre les activités considérées par un processus spécifique (à une famille d'applications). Comme illustration et pour bien comprendre les avantages de la proposition, un processus dédié aux Systèmes à Evénements Discrets est aussi présenté. Celui-ci repose sur une famille de méta-modèles spécifiques à ce domaine complétée par des transformations de modèles et des projections vers des outils spécifiques du domaine.

Plus précisément, les éléments proposés dans la suite s'appuient sur le schéma conceptuel du méta-modèle de processus SPEM [OMG08a] dans lequel les produits échangés entre les différentes activités sont des modèles. Le modèle de processus générique intégre dès lors les notions de mèta-modèle, de transformation et d'outil (spécifique). En rationalisant les concepts liés aux processus de développement, ce méta-modèle permet alors de définir et de configurer des modèles de cycle de vie spécifiques à des applications particulières.

La prise en compte des concepts IDM à chaque étape d'un processus conduit alors à des familles de méta-modèles reliées par des transformations et l'utilisation de ces dernières permet une réduction des efforts de développement. Pour résumer, l'approche proposée dans ce manuscrit cherche à obtenir une meilleure intégration des modèles spécifiques liés à un processus de développement spécifique.

En s'appuyant sur le concept de transformation de modèles, il est alors possible d'améliorer l'automatisation du processus allant de la spécification à la réalisation. Plus précisé-

ment, trois types de transformations de modèles sont considérés :

1. Des transformations permettant le passage d'une activité à une autre pour permettre le raffinement des modèles au fil du processus de développement.
2. Des transformations permettant au sein d'une activité de construire et de combiner des modèles en utilisant des langages de modélisation différents.
3. Des transformations vers des syntaxes concrètes permettant de fournir des passerelles vers les différents outils spécifiques du domaine considéré.

Pour une mise en œuvre de cette approche dans un domaine particulier, il convient notamment d'identifier les activités et outils spécifiques à ce domaine. Ceci conduit à identifier et/ou définir des langages de modélisation spécifiques pour lesquels des méta-modèles sont définis afin de fournir un support à la représentation et l'utilisation des concepts de domaine.

Pour effectuer cette opération, un nouvel acteur est introduit : *le méta-modeleur* ou expert de l'IDM. Aidé des experts du domaine, son rôle va être de concevoir les méta-modèles nécessaires au processus. Selon les besoins exprimés pour ce processus, le méta-modeleur aura également pour rôle de définir des transformations nécessaires à une meilleure automatisation ou une meilleure identification des activités du processus. Il aura également pour tâche de définir les transformations (ou projections) vers les outils supportant les activités du processus.

Dans une deuxième étape, les méta-modèles vont être organisés selon leur utilisation au sein des différentes activités. Différents modèles de processus spécifiques peuvent alors être définis. Ces modèles de processus sont appliqués pour la réalisation des applications.

1.3 Organisation du document

La suite du manuscrit comporte de sept chapitres :

Le chapitre 2 présente dans un premier temps les systèmes logiciels et les langages couramment utilisés pour les modéliser. L'accent est ensuite mis sur les processus de développement existants.

Le chapitre 3 présente un état de l'art des principaux éléments utilisés par l'Ingénierie Dirigée par les Modèles.

Le chapitre 4 présente le méta-modèle générique de processus orienté modèles. Plus précisément, ce chapitre présente les principes de la contribution dans le rapprochement des concepts de l'IDM et ceux de l'ingénierie des processus.

Le chapitre 5 propose une instance du méta-processus avec un modèle de processus dédié aux Systèmes à Événements Discrets (SED).

Le chapitre 6 présente deux exemples d'utilisation du modèle de processus défini au chapitre 5. Ces deux exemples proposent d'intégrer des phases de synthèse de superviseur et de vérification de modèle dans le processus de développement.

Le chapitre 7 conclut en rappelant les éléments importants de ce manuscrit et en donnant les perspectives considérées.

État de l’art

L’accumulation des connaissances n’est pas la connaissance.
(René Barjavel)

Le commencement de toutes les sciences, c’est l’étonnement de ce que les choses sont ce qu’elles sont.

(Aristote)

Chapitre 2

Les systèmes logiciels et leur développement

2.1 Introduction

L'élaboration est une tâche complexe qui demande la prise en compte de nombreux aspects et concepts, et nécessite la maîtrise des différents domaines métiers impliqués dans le développement. Des compétences relevant du domaine du génie logiciel sont employées pour rationaliser le développement par la mise en place de techniques outillées et l'optimisation du processus de développement.

Ces deux domaines (domaine métier et domaine du génie logiciel) forment alors les deux aspects fondamentaux du développement logiciel qu'il convient de maîtriser et qui demandent un haut niveau de compréhension et d'expertise.

Ces deux aspects sont essentiels au développement surtout si le développement à mettre en place doit être réutilisable et adaptable pour une famille particulière de logiciels. Il est donc important de prêter une attention toute particulière à la fois :

- aux différents domaines métiers mis en jeux lors d'une conception,
- à la façon dont le logiciel va être produit.

C'est pourquoi, avant d'aborder la proposition défendue dans ce manuscrit, qui consiste en la définition de modèles de processus de développement flexibles, il est nécessaire de faire le point sur les notions générales qui traitent de la nature des systèmes logiciels et de leur développement. Cette partie de l'état de l'art tente de répondre aux questions suivantes :

1. Quelles sont les méthodes et les langages de modélisation permettant de concevoir et de représenter les systèmes logiciels ?

2. Quelles sont les démarches rationnelles de développement ?

3. Quelles sont les difficultés rendant les systèmes complexes à concevoir ?

2.2 Le génie logiciel

Le génie logiciel est la discipline informatique rassemblant l'ensemble des techniques et méthodes permettant la mise en œuvre d'un produit logiciel [Som06].

En suivant cette définition, la préoccupation du génie logiciel est de rationaliser la production du logiciel afin d'optimiser les contraintes de coût, de temps et de qualité. Pour cela, un intérêt particulier est porté à la façon de modéliser les systèmes logiciels et leurs architectures. Différentes approches ont été proposées telles que l'approche orientée objet [JCJ92, BMN07], l'utilisation de frameworks [BMM95] ou de langages de description d'architectures [Dis04], etc.

Cependant, les contraintes de qualité, de temps et de coûts restent problématiques et amènent à considérer deux types d'attitudes [Som06]. Le premier type d'attitude, appelé généralement *best effort* (meilleur effort ou production au mieux), met l'accent sur la production et la livraison d'un produit en un temps et un coût de production minimal tout en essayant de garantir une qualité correcte. A l'inverse, la seconde attitude va mettre la priorité sur les contraintes de fonctionnement où les erreurs et les défaillances sont proscrites tout en essayant de garantir un temps et un coût de développement correct. Bien qu'antagonistes, ces deux types d'attitudes sont souvent associés. Ils amènent alors à des contextes de développement logiciel multiples et variés. Cette situation a pour conséquence de rendre la mise en œuvre des processus de développement plus complexe. Pourtant, cette multitude de contextes de développement doit être maîtrisée.

La rationalisation de la production du logiciel n'est possible que si elle intègre les multiples contraintes du développement. Le génie logiciel va alors chercher à contrôler ces contraintes. Pour cela, il va s'intéresser à la manière dont est organisée et gérée la production du logiciel, sous la forme de cycles de vie.

Afin de faire cohabiter des contraintes de coût et de temps avec le besoin de rationalisation du processus de développement, des ALM (Application Lifecycle Management) ont été développés. Les ALM [GPM08] sont des outils de pilotage du développement logiciel et de ce fait proposent des méthodes outillées permettant une meilleure gestion des différents processus qui interviennent lors du cycle de développement.

2.3 Les systèmes logiciels

Dans cette partie, nous proposons dans un premier temps d'aborder le concept de système logiciel et les propriétés fondamentales qui le caractérise. Ensuite, les différents types de systèmes logiciels existants sont présentés ainsi que les paradigmes de modélisation permettant de les concevoir. Pour finir, cette partie de l'état de l'art s'attardera sur les démarches de développement possibles pour aborder la conception des systèmes logiciels.

2.3.1 Complexité des systèmes logiciels

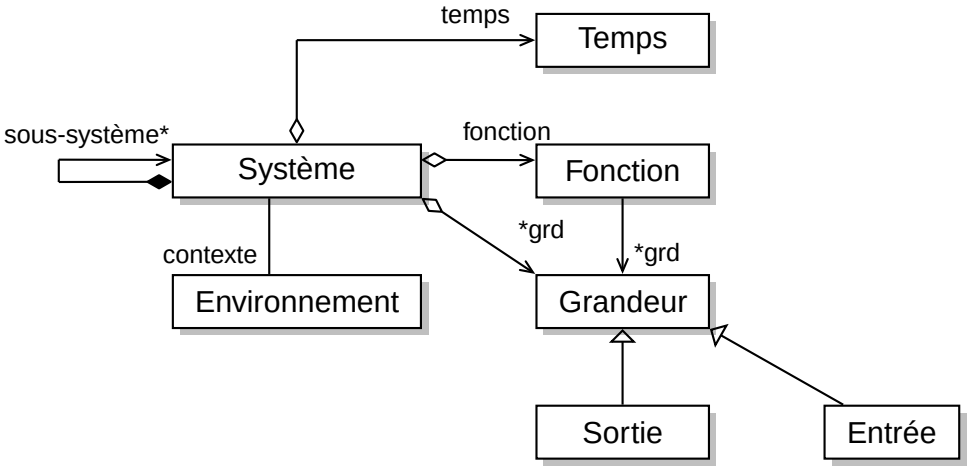
Les systèmes logiciels sont aujourd'hui omniprésents, [MS03a, EHS06]. Ils se situent au cœur des entreprises, des métiers et des produits. Ces systèmes assument de plus en plus de responsabilités les rendant alors critiques face aux exigences de fiabilité et de sécurité qu'ils doivent assurer. De plus, ces systèmes doivent intégrer de plus en plus de fonctionnalités.

T.A.Henzinger et J.Sifakis dans [HS06] posent le constat que le développement des systèmes logiciels actuels nécessitent la prise en compte de nombreuses disciplines dont, entre autre, l'automatique et l'informatique. Deux points de vue complémentaires peuvent alors être énoncés sur la nature des systèmes logiciels :

- D'un point de vue automatique, les systèmes logiciels représentent le moyen de contrôler ou d'automatiser un procédé (mécanique, biologique, électronique, etc.) afin de réaliser une tâche qui est trop pénible ou simplement impossible à effectuer par un être humain.
- D'un point de vue informatique, les systèmes logiciels représentent le moyen d'effectuer le traitement d'une information. Ainsi, l'informatique offre la possibilité de construire des outils pour les besoins spécifiques d'autres domaines en apportant ses propres concepts et technologies.

Illustré par la figure 2.1, le concept général de système est caractérisé par la définition suivante :

> *Un système est généralement décrit comme un ensemble d'éléments en interaction organisés en un tout homogène au sein d'un environnement avec lequel il interagit [Moi90, Ros75]. Il évolue dans le temps [SR67], il transforme des grandeurs d'entrées en grandeurs de sorties [Lar93] et il réalise des fonctions afin d'atteindre un objectif.*



F . 2.1 – Vue conceptuelle de la notion de système

Ainsi un système se compose de divers éléments tels que d'autres sous-systèmes et assure une fonction permettant d'atteindre un objectif en prenant en paramètre des grandeurs d'entrées/sorties. Il évolue dans un contexte défini par un environnement en fonction de références temporelles.

Partant de cette définition, et en l'utilisant pour les systèmes logiciels, la complexité inhérente à la conception logicielle provient de l'ensemble des éléments intervenant au sein du système. Cependant, ce n'est pas la multitude des éléments formant le système qui va être la source de la complexité mais la communication et les interconnexions qui existent entre ces éléments. En effet, la multitude des éléments constituant le système le rend forcément compliqué à étudier. Cependant, par définition, un problème compliqué n'est généralement qu'un assemblage de concepts nécessitant du temps et de la rigueur pour être appréhendé convenablement. Par contre, la complexité apparaît dès qu'il existe de multiples relations entre ces concepts, ici issues du couplage et de la communication entre les éléments du système. Cette communication peut alors induire des comportements difficilement prévisibles.

Cet état de fait rend de tels systèmes difficiles à comprendre et donc à concevoir. En effet, chaque élément est soumis à un ensemble de propriétés dites locales qui lui sont propres. Mais, les interconnexions entre les éléments vont engendrer des propriétés globales qui vont être plus difficiles à identifier et dont les influences vont également être difficilement prévisibles. Cette particularité caractérise généralement les systèmes complexes : ces systèmes comportent des propriétés locales non généralisables à l'ensemble du système.

2.3.2 Nature et caractéristiques du logiciel

Les systèmes logiciels, comme tout système, sont constitués de sous systèmes en interactions et évoluent dans un environnement. A cette caractérisation, se greffe un certain nombre de concepts clefs tels que : *l'architecture, l'état, le comportement, la réactivité, la concurrence, la communication, l'émergence ou encore la complexité* [Per07].

L'architecture décrit l'agencement des différents constituants d'un système. Effectué de manière hiérarchique et/ou modulaire, cet agencement se structure en termes de composants et d'interactions.

L'état se modélise généralement par un vecteur de variables prenant des valeurs caractéristiques pour une configuration donnée du système.

Le comportement représente les diverses évolutions possibles (ou trajectoires) suivies par le système au sein de son espace des états.

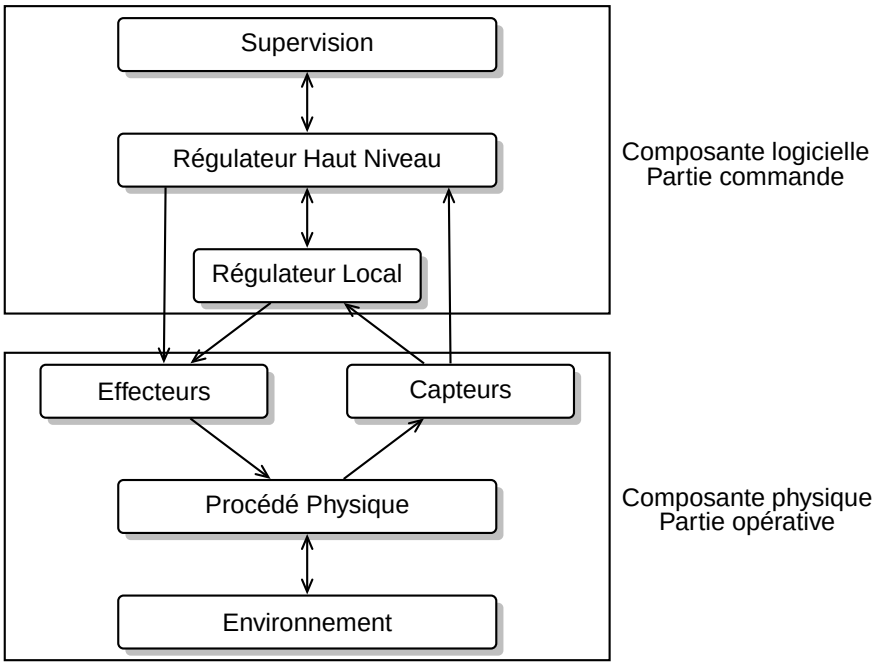
La réactivité dénote la capacité d'un système à réagir et à s'adapter aux événements, le plus souvent asynchrones, de son environnement. Les occurrences successives de ces événements (aussi appelées traces), définissent l'évolution de son comportement sous la forme d'une succession d'états.

La concurrence est une caractéristique intrinsèque à tout système complexe. Elle résulte du fait que ces systèmes sont composés d'un ensemble d'entités autonomes qui collaborent ou se partagent des ressources critiques (compétition).

La communication existe dès lors qu'il y a décomposition d'un système en plusieurs entités (objets, agents, composants, etc.). Il est nécessaire de coordonner ces entités par la communication pour que l'ensemble réalise le comportement global souhaité.

L'émergence consiste en l'apparition, à un niveau d'organisation, de propriétés n'existant pas de manière explicite à un niveau d'organisation inférieur. Elle peut résulter par exemple de l'assemblage d'entités concurrentes en interaction. Cette caractéristique se traduit généralement par l'adage : *le tout est plus que la somme des parties*.

La complexité d'un système provient de la multiplicité des éléments qui le composent, de la diversité de leurs relations et de l'imprévisibilité de son comportement (phénomène d'émergence décrit précédemment).



F . 2.2 – Chaîne d'interactions entre la partie commande et la partie opérative d'un système logiciel de type contrôle commande

Ainsi, par la multiplicité des concepts qui les forment, les systèmes logiciels sont par nature difficiles à modéliser. Ainsi, pour chaque domaine spécifique nécessitant un développement logiciel, il peut être nécessaire de définir une architecture typique afin de guider et faciliter leur modélisation. Par exemple, dans le cas particulier des systèmes logiciels de commande, une architecture typique prenant en compte les éléments et l'environnement de ce type de logiciel, peut être définie (figure 2.2). Une telle architecture [Zay03], propose une décomposition de la chaîne d'interactions entre la partie logicielle, de commande, et la partie opérative.

La partie opérative correspond au cœur du système. Elle est constituée de l'ensemble des entités physiques qui rassemblent les fonctionnalités pouvant être pilotées.

La partie commande assure le pilotage (commande, supervision, coordination, configuration) de la partie opérative en lui appliquant une loi de commande.

De façon à permettre la collaboration de ces deux parties, cette architecture va s'articuler autour de deux types de flots : les flots de données et les flots de contrôles.

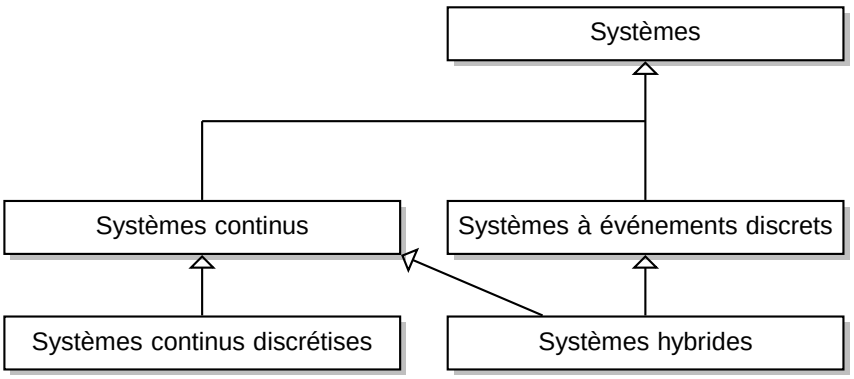
Les flots de données assurent l'acquisition, la manipulation, la modélisation de l'environnement et la transmission des informations entre chaque niveau.

Les flots de contrôles assurent l'exécution (démarrage, initialisation, préemption, etc.) des processus ou des tâches qui composent le système.

Une communication bidirectionnelle entre la partie opérative et la partie commande va permettre la formation d'une boucle d'asservissement. Ainsi, l'environnement entre en interaction avec le procédé physique piloté, les effets de cette interaction vont être répercutés par des capteurs fournissant à la partie logicielle des informations conditionnées. La partie logicielle va alors traiter les informations reçues afin d'élaborer une commande adaptée, en réponse aux stimuli ou perturbations dûes à l'environnement. La commande va redescendre la chaîne d'interactions par le biais des effecteurs jusqu'à la partie opérative.

2.3.3 Modélisation du logiciel

a. Systèmes à Événements Discrets



F . 2.3 – Nature des systèmes

Dans le domaine de l'automatique, les systèmes sont généralement classés en trois familles (figure 2.3) : les systèmes continus, les systèmes à événements discrets et les systèmes hybrides [Zay03]. Cette thèse se limite au développement des logiciels pour les Systèmes à Événements Discrets (SED).

Les SED sont des systèmes dont l'état est défini sur un ensemble fini de valeurs. L'évolution de ces systèmes est généralement due à l'occurrence asynchrone d'événements. Ces événements peuvent être qualifiés de commandables (autorisation/interdiction de leurs occurrences) ou non ou encore observables ou non (annexe B). De nombreux domaines d'application utilisent les systèmes à événements discrets : les chaînes de production, les réseaux de télécommunications, la robotique, les systèmes de transports, etc.

La partie commande des SED peut être synthétisée manuellement par une activité de modélisation et vérifiée moyennant des activités de Model-Checking [MK06] (annexe A).

Une deuxième approche procède par génération automatique des modèles de la commande. Cette approche se fonde sur la théorie de la commande par supervision, initiée par Ramadge et Wonham en 1989 [RW89] (annexe B).

Au delà de la prise en compte et de la modélisation des aspects purement événementiels des SED, de nombreuses approches et/ou langages de modélisation permettent une modélisation plus complète de la partie commande. Issus du domaine du génie logiciel, il est possible de trouver par exemple, l'approche orientée objet [BMN07, BMM95], UML [OMG10a, OMG10b], AADL [Dis04, BFS05] ou encore SysML [OMG10d].

b. Approche Objet

L'approche Objet est généralement reconnue comme offrant des moyens d'appréhender et de maîtriser la complexité des systèmes logiciels [BMN07].

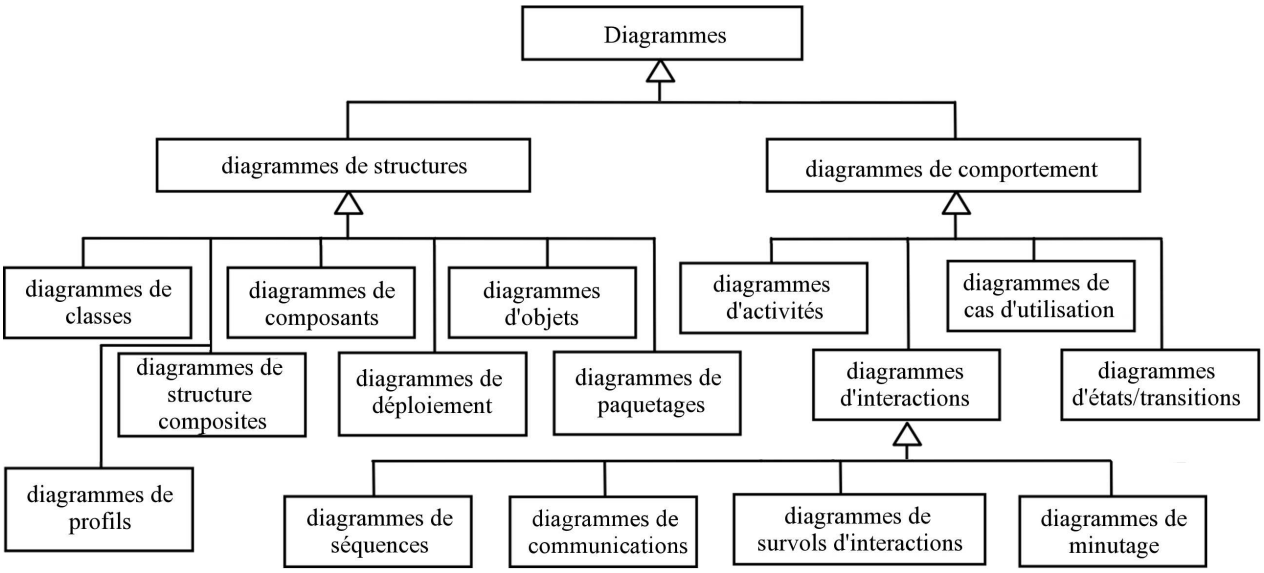
Elle permet la construction d'un logiciel à travers un processus de décomposition et de composition reposant sur les concepts *d'abstraction, d'encapsulation,* de *hiérarchie* et de *modularité*. Les architectures à objets sont composées d'objets (entités ayant une identité, un état, un comportement et une durée de vie), instances de classes, qui sont en interaction statique et dynamique, [JCJ92, BMM95, BMN07].

La réalisation des comportements est issue de la collaboration des objets. L'organisation ou la structure des classes permet d'organiser la complexité des genres à l'aide de relations de généralisation et de composition tout en factorisant les propriétés communes des instances. L'organisation des instances définit des configurations d'objets nécessaires à l'élaboration d'objets ou sous-systèmes de plus haut niveau.

Avec l'approche objet, les classes et objets représentent des abstractions directes de la réalité. A travers une démarche ascendante et une fois le problème cerné, le concepteur peut organiser ses connaissances afin de synthétiser plus facilement des systèmes aux comportements complexes. La conception d'un système logiciel apparaît donc comme une démarche d'intégration et d'organisation d'entités élémentaires.

c. UML

UML (Unified Modeling Language) est un langage de modélisation graphique destiné à visualiser, analyser, spécifier, construire des logiciels orientés objets [OMG10a, OMG10b]. UML est aujourd'hui considéré comme un standard autant dans le milieu industriel qu'académique. Il propose un ensemble de diagrammes afin de couvrir l'ensemble des besoins de modélisation potentiellement nécessaires à la conception des logiciels, ce qui le rend relativement complet et générique. Ainsi, au travers des 14 types de diagrammes (figure 2.4), UML permet de modéliser les aspects statiques et dynamiques des systèmes complexes et de couvrir la plupart des phases du développement logiciel (analyse, conception, implantation, déploiement, etc.).



F . 2.4 – Les différents types de diagrammes UML

Dans le cadre de cette thèse, afin d'illustrer la proposition, nous nous sommes appuyés sur :

Les diagrammes des classes : permettent de visualiser l'agencement des concepts d'un système en représentant les classes, leurs rôles, leurs caractéristiques, les services qu'elles proposent, leurs relations (association, composition, etc.) et enfin leur organisation au sein de l'architecture du système logiciel. Ils permettant aussi de représenter des méta-modèles.

Les diagrammes de communication : décrivent des configurations types en termes d'objets collaborant. Pour cela, ils permettent de représenter d'une part la configuration d'un ensemble d'objets et d'autre part les relations dynamiques entre ces objets.

Les diagrammes d'états transitions : capturent le comportement des objets sous la forme d'un graphe d'états reliés par des transitions. Le franchissement des transitions se réalise à la suite de la réception d'un signal (appel de méthode, exception, etc.).

Les diagrammes de cas d'utilisations : permettent pour leur part d'identifier les fonctionnalités d'un système et les conditions nécessaires à leur bon fonctionnement. Ils font apparaître les éléments fonctionnels, les acteurs et les objets en interaction.

UML s'est rapidement imposé pour la conception des systèmes logiciels, pourtant ce langage comporte un certain nombre de lacunes. En effet, il est souvent reproché aux diagrammes UML de posséder une sémantique ambiguë et/ou incomplète. Dans l'évolution qu'a connu UML, cet aspect du langage est devenu un concept : *le point de variation*. Ainsi, ce coté semi-formel permet au concepteur d'utiliser un langage offrant plus de flexibilité afin d'être étendu vers des contextes de modélisation plus spécifiques.

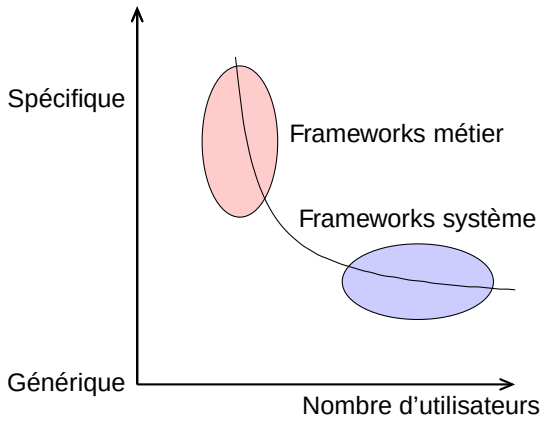
UML peut être étendu par de nouveaux concepts (à l'aide de stéréotypes ou de contraintes OCL). Une extension de la notation à un domaine spécifique est appelée aussi *profil*. Par exemple le profil *TURTLE* [AD05] et le profil *ACCORD/UML* [GMT04] proposent des extensions des diagrammes UML dédiés à la modélisation des systèmes temps-réels.

d. Frameworks

Face à l'industrialisation des processus de développement des systèmes logiciels, une solution a été apportée avec l'utilisation des frameworks. D'une manière générale un framework est défini comme *une architecture réutilisable constituée d'un ensemble de classes interconnectées proposant des éléments de solutions génériques pour une famille de problèmes spécifiques* [BMM95]. Les frameworks permettent d'augmenter la réutilisation des modèles d'architectures et des entités logicielles déjà conçus.

Les frameworks se caractérisent par deux propriétés essentielles. Ils sont capables de capturer et de concrétiser les concepts essentiels d'un domaine métier particulier, amenant ainsi une meilleure maîtrise de la complexité inhérente à ce domaine. Ils fournissent une logique d'exécution adéquate, permettant aux différents concepteurs qui utilisent le framework de ne se préoccuper que des aspects particuliers du problème à traiter sans se soucier des détails des mécanismes d'exécution.

Les frameworks se classent généralement en deux familles : les frameworks verticaux (*frameworks métiers*) et les frameworks horizontaux (*frameworks systèmes* ou *techniques*). La différence majeure entre ces deux types de framework est leur généricité et donc le nombre d'utilisateurs potentiels, ce qui a des conséquences en matière de rentabilité/coût de développement d'un framework (figure 2.5).



F . 2.5 – Comparaison des frameworks métiers et systèmes

Les *frameworks métiers* se caractérisent essentiellement par une capture précise des concepts d'un domaine. Ce sont des outils qui maîtrisent alors le domaine modélisé à un haut niveau d'expertise. Cependant, cette expertise en font des outils peu souples et peu réexploitables pour d'autres domaines d'application. Par exemple, des frameworks associés à des domaines tels que la logique floue [PPT02], la simulation ou la conception de systèmes concurrents hétérogènes (Ptolemy [EJL03]) sont à classer parmi les frameworks métiers.

A l'inverse, les *frameworks systèmes* se caractérisent par une forte généricité. Les concepts mis en œuvre fournissent en général des moyens techniques pour résoudre des problèmes variés. Par contre, ils ne sont d'aucune aide pour modéliser un domaine métier particulier. Par exemple, le framework de développement d'application fenêtré Windows (Microsoft Foundation Classe MFC) est un framework système.

e. Architectures logicielles

L'architecture est reconnue comme un moyen de guider le concepteur lors de la conception d'un système logiciel. Elle permet de raisonner à un haut niveau d'abstraction et d'organiser les composants du système tout en définissant les interactions qui relient ces composants. Correctement décrite, l'architecture favorise la compréhension, l'analyse, l'évolution, la fiabilité et la réutilisation des systèmes logiciels [BCK03].

Différents langages de description d'architecture ont été définis tels que Acme [GMW97], Rapid [Ken96], Wright [All97], Darwin [MNE95] ou AADL (Architecture Analysis and Design Language) [Dis04, BFS05]. Par exemple, les concepts sur lesquels se base le langage Wright sont ceux qui forment les bases de tous les ADL : *des composants, des connecteurs, des rôles et des ports*. Pour décrire les aspects dynamiques Wright utilise CSP [Hoa04]. Plus spécifiquement, le langage Darwin a été conçu afin de permettre la description des systèmes logiciels distribués. Il se base sur les concepts de composants et d'interfaces pour modéliser la structure et utilise FSP [MK06] pour modéliser le comportement.

Le langage AADL, initialement défini pour les systèmes avioniques puis standardisé par la SAE (Society of Automotive Engineers), permet de définir l'architecture de systèmes logiciels temps-réels embarqués. La construction de modèles AADL est décomposée en deux phases : la première phase consiste en la description des interfaces des composants qui forment l'architecture du système logiciel tandis que la seconde phase s'intéresse à l'implantation de ces composants. Cette démarche amène alors à séparer les aspects architecturaux du système des aspects opérationnels des différents composants permettant ainsi une démarche plus rationnelle.

On peut décrire succinctement les composants AADL selon différents types :

Les composants matériels : les mémoires (*memory*), les périphériques (*device*), les processeurs (*processor*) et les bus (*bus*).

Les composants logiciels : les données (*data*), les sous-programmes (*subprogram*), et les threads (*thread*).

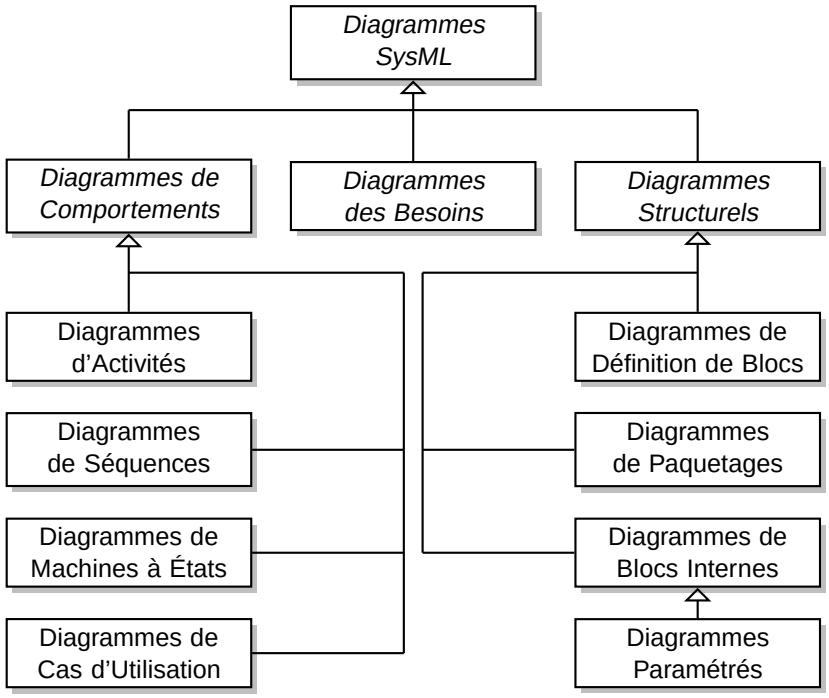
Les composants composites : les groupes de threads (*thread group*), les processus (*process*) et les systèmes (*system*).

Après un raffinement possible des composants qui peut être fait par des relations d'héritage, les différentes implantations des composants peuvent être interconnectées par des connections via des ports de communications qui gèrent les flots de contrôles ou les flots de données.

f. SysML

Le standard SysML, [OMG10d] est un langage de modélisation dédié au domaine de l'ingénierie des systèmes utilisé notamment pour la spécification, la conception et la vérification. Il s'agit d'un profil UML ce qui permet l'utilisation de certains diagrammes UML.

En effet, il regroupe un sous ensemble de ces diagrammes complétés par de nouveaux diagrammes plus spécifiques (figure 2.6).



F . 2.6 – Les différents types de diagrammes du profil SysML

Comme pour UML, il est possible d'utiliser des diagrammes de comportements comme *les diagrammes d'activités*, *les diagrammes de machines à états*, ou encore les *diagrammes de cas d'utilisations*. Ces différents diagrammes correspondent globalement à ceux d'UML.

D'un point de vue structurel, SysML fournit par contre des diagrammes différents d'UML. Les formalismes disponibles sont *les diagrammes de définition de blocs*, *les diagrammes internes de blocs* et *les diagrammes de paquetages*. Ces trois types de diagrammes peuvent sembler dans un premier temps très différents des diagrammes structurels classiques d'UML, pourtant, malgré leur orientation ingénierie des systèmes, certaines analogies peuvent être faites. Ainsi, *les diagrammes de paquetages* sont des diagrammes permettant de représenter et d'organiser les composants à mettre en place au sein du système. Ce type de diagramme SysML est à rapprocher du *diagramme de paquetages* d'UML.

Les *diagrammes de définition des besoins* sont intéressants pour les clients des équipes de développement car ils offrent des concepts simples et abordables, même pour des utilisateurs non initiés. Ils permettent de hiérarchiser les besoins du système, leurs relations et leurs raffinements.

Les *diagrammes de définition de blocs* permettent de mettre en place les objets nécessaires à la conception du système et de représenter leurs éventuelles relations (association, composition ou généralisation). Ces diagrammes peuvent être mis en relation avec *les diagrammes de classes et d'objets* d'UML dont ils tirent les principaux concepts.

Par contre, les *diagrammes internes de blocs* sont des diagrammes différents d'UML. En effet, ils permettent d'aller plus loin dans la définition des relations existantes entre les objets du système en fournissant les moyens de représenter *des diagrammes de flots de contrôle et de flots de données*. Dans ce type de diagramme apparaît ainsi la notion de port de communication qui est déclinée en deux variantes : des ports destinés à recevoir ou envoyer des objets et des ports pour recevoir ou envoyer des flots. Ce type de diagramme, en cohérence avec la théorie des systèmes et de l'automatique permet de modéliser des *schémas blocs*.

2.4 Les approches Modèles

2.4.1 Vers une modélisation systématique

La généralisation de l'utilisation de langages de modélisation comme UML, ADL, SysML, induit une évolution des métiers de l'informatique vers une utilisation plus systématique des modèles dans toutes les phases du développement logiciel. Ce changement intervient pour faire face au problème que pose la complexité inhérente aux développements logiciels actuels.

Un modèle est une abstraction contenant un ensemble restreint d'informations représentant un système selon un point de vue [Fav04]. Il est construit dans un but précis et les informations qu'il contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui en sera faite. La notion de modèle est fondamentale, elle se situe au cœur de toute démarche scientifique et ce, à tel point, que par abus de langage, le système et le modèle le représentant sont souvent confondus.

Par la découverte, la formalisation et l'organisation de concepts spécifiques d'un domaine ou d'un point de vue, les modèles permettent de faire progresser la connaissance que l'on peut avoir d'un système. Les modèles sont d'une part descriptifs ; ils permettent des activités comme l'analyse, la vérification, la validation, et d'autre part, normatifs dans la mesure où ils possèdent suffisamment d'informations autorisant la synthèse manuelle, semi-automatique ou automatique de représentations particulières comme l'implantation, [Per07]. Par exemple, dans le cadre de la commande des systèmes, les modèles mathématiques sont largement utilisés. Ainsi, le passage d'un modèle théorique (d'une commande) à une réalisation logicielle peut être facilité si l'ingénierie logicielle est aussi une ingénierie dirigée par les modèles.

Adopter une ingénierie dirigée par les modèles permet de définir un cadre cohérent pour tous les aspects du développement logiciel, que cela concerne les aspects *métiers* du logiciel ou les aspects *techniques* de la conception logicielle (utilisation de frameworks, par exemple). Elle a également pour intérêt de fournir un cadre plus abstrait qui libère les concepteurs des préoccupations techniques ou technologiques. De manière générale, les modèles améliorent les conceptions car ils masquent les détails non pertinents et soulignent les aspects importants. De plus, ils facilitent la compréhension des problèmes et de leurs solutions tout en fournissant un support de communication, [Per07].

Les modèles logiciels possèdent également un avantage indéniable, ils peuvent être transformés en d'autres modèles (voir chapitre 3) ; ce qui évite l'introduction de certaines erreurs [MS03b].

Les approches modèles sont aujourd'hui nombreuses ; il est possible de citer par exemple les approches Matlab-Simulink [Mat05], MIC-GME [KSL03], Ptolemy [EJL03], MDA (Model Driven Architecture) [Béz04] et l'IDM (Ingénierie Dirigée par les Modèles) [FEB06].

2.4.2 Matlab-Simulink

Conçu par Cleve Moler dans les années 1970, Matlab [Mat05] est un outil issu initialement du monde académique dont le développement et la commercialisation sont aujourd'hui effectués par la société The MathWorks. Grâce à des modèles formels/mathématiques, écrits dans un langage dédié, cet outil permet d'effectuer du calcul scientifique et d'assister certaines phases du développement. Ainsi, Matlab est rapidement devenu un standard dans le monde industriel en particulier pour la réalisation de modèles pour les systèmes dynamiques ou le traitement de signal.

Depuis sa création, l'environnement Matlab s'est enrichi de nombreuses extensions appelées ToolBox parmi lesquelles les principales sont :

Control System Toolbox : une boîte à outils pour la définition et la conception de systèmes de contrôle et de commande.

Real-Time Workshop : une boîte à outils pour la définition et la conception de systèmes temps-réels.

Simulink : un outil de modélisation graphique pour les systèmes dynamiques.

En particulier, Simulink est un outil fournissant un langage de modélisation graphique pour certaines boîtes à outils Matlab. Ainsi, à l'aide de ce langage, il est possible de modéliser des systèmes dynamiques, de les simuler ou les implanter.

Simulink permet la modélisation de systèmes continus avec des schémas blocs, la modélisation des systèmes à évènements discrets à l'aide de diagrammes d'états (StateFlow) ou encore par composition de ces deux formalismes de modéliser des systèmes hybrides.

2.4.3 MIC-GME

Model-Integrated Computing (MIC) [KSL03] s'intéresse au développement de systèmes logiciels en fournissant un environnement de modélisation pour des domaines spécifiques. Afin de faciliter l'ingénierie des systèmes logiciels, MIC permet de concevoir et faire évoluer des modèles basés sur de multiples domaines spécifiques en se fondant sur les concepts, les relations et les principes de compositions de modèles issus des différents domaines. Cela permet alors d'analyser les modèles, ainsi que d'effectuer de la génération de code de façon automatique.

L'objectif de MIC est de fournir :
- une intégration à l'aide de modèles des concepts et des informations nécessaires au développement du système logiciel. Les modèles représentent alors la vision que les concepteurs se font du système en incluant les aspects architecturaux, les modèles d'exécution ainsi que l'environnement dans lequel le système va évoluer ;
- des outils pour analyser les différentes caractéristiques du système (comme la sûreté, l'atteignabilité des états, etc.) ;
- un interpréteur de modèles pour effectuer la génération automatique de code et obtenir un logiciel conforme aux spécifications (cet interpréteur a pour but de réduire l'écart qui existe souvent entre les modèles et leur représentation sous forme de code) ;
- une interface basée sur UML permettant de faire évoluer les environnements de développement des différents domaines.

Pour mettre en œuvre cette démarche, MIC donne une définition de la notion de langage de modélisation [KSL03] : un langage de modélisation pour un domaine spécifique est spécifié en donnant : *Une syntaxe abstraite (S_a), une syntaxe concrète (S_c), un domaine sémantique (Sem), et deux relations liant pour la première les concepts de la syntaxe abstraite à ceux de la syntaxe concrète ($\mathcal{R}_c$), et les concepts de la syntaxe abstraite aux éléments du domaine sémantique ($\mathcal{R}_s$) :*

$$Lm =< S_c, S_a, Sem, \mathcal{R}_c, \mathcal{R}_s >$$

La syntaxe concrète S_c permet d'exprimer les modèles textuellement, graphiquement ou les deux. La syntaxe abstraite S_a définit pour sa part les concepts, les relations, les contraintes d'intégrité du langage. Enfin le domaine sémantique Sem décrit dans un formalisme mathématique les éléments du langage.

L'approche MIC préconise l'utilisation de méta-outils (outils génériques configurables à l'aide de méta-modèles) tels que GME [LMB01]. En effet, les méta-outils permettent la conception de modèles de langages de modélisation, appelés méta-modèles. Ces méta-modèles vont alors servir de support pour la conception de modèles.

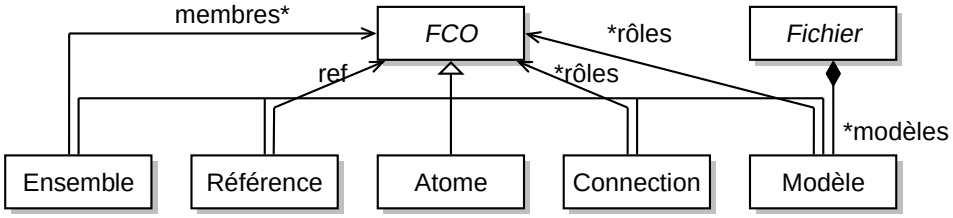
GME va s'appuyer sur trois concepts clefs :

Une architecture multi-graphe (MGA-MultiGraph Architecture) : il s'agit d'un ensemble d'outils permettant la conception d'environnements de modélisation pour des domaines spécifiques. Une MGA permet alors de concevoir un environnement de développement pour l'implantation de modèles (Model-Integrated Program Synthesis (MIPS)).

La génération de programmes à partir des modèles (MIPS) : il s'agit d'un environnement structuré en trois composants :
- un éditeur de modèles pour les domaines considérés,
- les modèles eux-mêmes,
- des interprètes pour la manipulation et la transformation des modèles.

Un paradigme de modélisation : il s'agit de la famille de modèles qui peuvent être créés à partir d'un environnement MIPS. Ce paradigme est concrétisé par un méta-modèle

qui s'intègre dans GME afin qu'il permette à son tour l'édition de modèles.



F . 2.7 – Méta-Modèle FCO (First Class Object)

GME s'appuie sur un méta-méta-modèle. Ce méta-méta-modèle (figure 2.7), nommé FCO (First Class Object), s'articule autour des notions de Modèle (*Models*), d'Atome (*Atoms*), de Référence (*References*), de Connection (*Connections*), d'Ensemble (*Sets*) et de Fichier (*Folders*). Les Fichiers sont des conteneurs pour organiser les modèles. Les Atomes sont les éléments élémentaires de modélisation. Ils ne contiennent aucune entité. Les Modèles sont des objets composites permettant de hiérarchiser les concepts. Des Connections sont utilisées pour mettre en place les relations qui doivent être définies entre les entités des différents modèles existant au sein d'un même conteneur. Les Références sont similaires au concept de pointeur en programmation. Elles sont utilisées pour définir des associations entre des entités du modèle. Enfin, la notion d'Ensemble est un moyen pour créer des groupes d'entités.

2.4.4 Ptolemy II

Ptolemy [EJL03] est un framework et un environnement assurant l'intégration de domaines hétérogènes nécessaires à la réalisation de systèmes embarqués. En effet, ceux-ci requièrent la prise en compte de concepts variés, comme la concurrence, la réactivité ou les aspects temps-réels. Il se charge également de la problématique de la commande des systèmes par l'identification des langages de modélisation facilitant leur élaboration d'un point de vue tout aussi bien matériel que logiciel. Face au problème d'hétérogénéité, Ptolemy propose comme solution d'utiliser un principe de composition de modèles où chaque modèle représente une partie du système et permet des activités de Model-Checking (Annexe A).

Le paradigme de modélisation défendu par Ptolemy s'articule autour d'une architecture orientée flots de données et flots de contrôle. L'entité de base est l'*acteur* qui est un concept équivalent à celui de composant et proche du concept d'objet. Les acteurs sont des entités communicantes via des interfaces nommées *Ports*. Ils peuvent évoluer de manière asynchrone selon leur propre fil d'exécution, ou peuvent évoluer ensemble, de façon synchronisée.

L'utilisation des acteurs permet de souligner l'aspect causal et communicant des systèmes et de voir l'ensemble des états possibles comme une structure d'objets interconnectés. Ce paradigme permet alors de dissocier la transmission des données et la logique de contrôle.

Les modèles sont construits par composition d'acteurs et, selon leur combinaison, par la mise en relation de leurs ports qui peuvent être d'entrée, de sortie, internes ou externes (pour les acteurs composites). L'exécution se déroule en plusieurs phases :

Pré-initialisation : Cette phase gère les informations structurelles telles que la construction des acteurs.

Initialisation : Cette phase initialise les variables et l'état des acteurs.

Itération : Cette phase, récurrente, détermine comment un acteur s'exécute. Elle se divise en trois étapes :
- *prefire* : test des pré-conditions à l'exécution d'une étape de calcul,
- *fire* : exécution d'une étape de calcul associée au comportement de l'acteur,
- *postfire* : mise à jour des variables produites, une fois que tous les acteurs ont réalisé leur étape *fire*.

Ptolemy permet de définir des outils de modélisation pour des domaines variés. L'implantation d'un langage de modélisation pour un domaine particulier est alors réalisé à l'aide de deux classes *Director* et *Receiver*. Le *Director* a pour charge de définir le modèle d'exécution que devront suivre les acteurs. Il a également pour charge de créer les différents *Receiver*. Ces derniers permettent de définir les mécanismes de communication entre les acteurs. Les *Receiver* prennent place au sein des ports d'entrées sorties pour établir des canaux de communication. Par exemple, un *Receiver* peut être défini sous la forme de boîtes aux lettres, etc.

Plusieurs paradigmes de modélisation sont prédéfinis et ont été implantés. Ainsi, Ptolemy offre des possibilités de modélisation à l'aide de Processus Séquentiels Communicants, d'Équations Différentielles, de Systèmes à Évènements Discrets, de Réseaux de Processus ou encore de Flots de Données Synchrones.

Le polymorphisme des acteurs permet de réaliser des modèles réutilisables dans des contextes de modélisation différents. Cela permet une approche de conception intégrant des domaines différents et hétérogènes. De plus, Ptolemy introduit le concept d'hétérogénéité hiérarchique permettant ainsi la conception de domaines hétérogènes dans lesquels différents paradigmes de modélisation peuvent être combinés.

Enfin, au delà du paradigme de modélisation orienté acteurs, Ptolemy fournit une plateforme outillée de modélisation et selon les types de modèle, il est possible d'effectuer de la vérification de modèle. Enfin, en plus de la génération de code pour l'implantation des modèles sur une plateforme d'exécution, il est possible de simuler les modèles afin d'en valider préalablement le comportement.

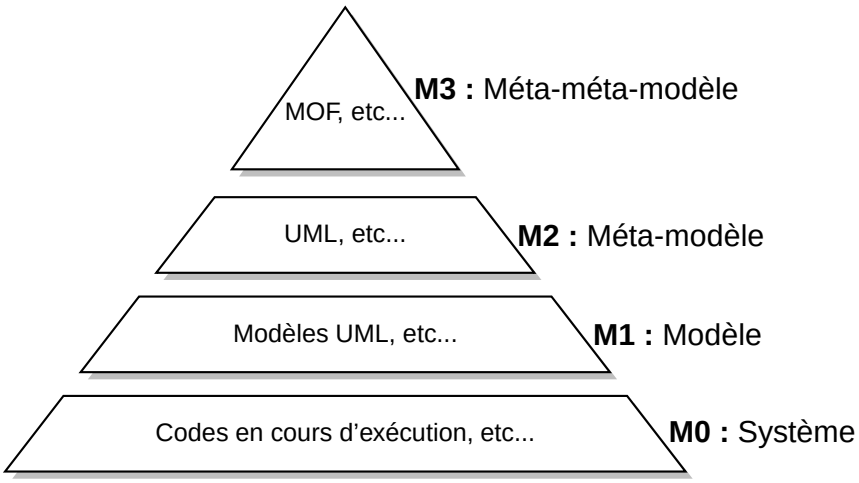
2.4.5 MDA-IDM

Les approches modèles proposées par le Model Driven Architecture (MDA) [Béz04] et par l'Ingénierie Dirigée par les Modèles (IDM) [FEB06] sont des approches issues du domaine du génie logiciel. Leur objectif est de rationaliser et de simplifier les démarches de conception logicielle. Pour cela, elles s'appuient sur les concepts de modèles, de langages

de modélisation et de transformations de modèles.

Historiquement, le MDA avait pour objectif de réduire l'écart existant entre le logiciel (et ses modèles abstraits) et la plateforme sur laquelle il doit s'exécuter. L'idée principale du MDA était de rationaliser et de capitaliser les bonnes pratiques du développement logiciel. Le MDA considère alors l'architecture logicielle selon deux points de vues : celui spécifique à la plateforme (PSM pour Platform Specific Model) et celui indépendant à plateforme (PIM pour Platform Independent Model). Construite autour d'un cycle en Y (détaillé dans le chapitre 2.5.2.c.), l'objectif de cette démarche est, en partant d'un modèle réalisé dans un cadre générique, de pouvoir ensuite le projeter dans des contextes variés, selon des langages et des plateformes d'exécutions spécifiques. Cette démarche a apporté l'une des premières notions importantes que l'on retrouvera dans l'IDM : la réutilisation de modèles.

La seconde notion importante introduite par le MDA est celle de modèle de langage de modélisation, aussi appelé *méta-modèle*. A partir de cette notion, le MDA a proposé la pyramide à 4 niveaux (figure 2.8) définissant les niveaux d'abstraction existant entre les différents types de modèles possibles.



F . 2.8 – Pyramide de modélisation de l'OMG

Une action spécifique menée par le CNRS et portant le nom de Action Spécifique MDA a donné lieu à la publication de l'ouvrage [FEB06] et a posé les bases de l'IDM qui généralisent le MDA. L'IDM ou Ingénierie Dirigée par les Modèles met les modèles au cœur du développement logiciel. Tout comme le MDA, elle est fondée sur les notions de modèle, de méta-modèle, de langage de modélisation et de transformation de modèles, et permet d'envisager une mise en œuvre de systèmes logiciels plus rationnelle tout en permettant l'intégration de domaines hétérogènes.

Dans ce cadre, la rationalisation du discours conduit aux définitions suivantes :

- *un système est l'entité étudiée dans le cadre d'un processus de modélisation ;*
- *un modèle est une abstraction d'un système, réalisé dans une intention et un contexte particulier ;*

- *un langage de modélisation, décrit par une syntaxe, est un ensemble de modèles conformes à cette syntaxe ;*
- *un méta-modèle spécifie un langage de modélisation en donnant sa syntaxe. Il concrétise les concepts du langage de modélisation et définit les règles de construction des modèles.*

Enfin, les transformations de modèles permettent l'analyse, la modification ou la synthèse d'un modèle suivant des règles définies au niveau des méta-modèles. Plusieurs types de transformations sont envisageables : les transformations *exogènes* et les transformations *endogènes*. Les transformations exogènes servent à traduire un modèle défini dans un langage de modélisation vers un modèle dans un autre langage de modélisation ; un exemple typique est la génération de code. Les transformations endogènes ont pour vocation de transformer un modèle vers un autre modèle au sein d'un même langage de modélisation ; un exemple typique est l'optimisation d'un modèle (refactorisation, etc.)

2.5 Les processus de développement

2.5.1 Définitions et Concepts

Différentes approches ont été proposées pour gérer les processus de développement (paragraphe 2.5.2) associés à la production de logiciels, [Som06, Mes07]. Ces approches, appelées cycles de vie, permettent de rationaliser les activités qui interviennent tout au long du développement et de mieux gérer les acteurs qui y participent. Partant du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de conception, un des objectifs de la mise en place des cycles de vie est de détecter ces erreurs au plus tôt et ainsi de mieux maîtriser la qualité du logiciel, les délais de sa réalisation et les coûts occasionnés ; dans la mesure où ils prévoient des phases de vérification/validation "tôt" dans le cycle.

Les activités typiques d'un cycle de vie sont *la définition des besoins, la conception, le développement, l'implantation et enfin les tests* avant la livraison du produit au client. Une première façon d'organiser ces activités conduit aux cycles de vie dits prédictifs dont des illustrations sont le cycle en *V* [MR84] ou le cycle en *Cascade* [Roy70]. Ces cycles sont dits *prédictifs* car ils demandent, dès l'identification et la mise en place du cycle, de définir toutes les échéances qui en jalonneront l'évolution.

Ces cycles ont fait indubitablement la preuve de leur efficacité dans des développement de produits bien définis grâce à la rigueur qu'ils imposent. Cependant, aujourd'hui, ils ont montré des limites dans certains secteurs (développement de produits innovants par exemple) où une maîtrise totale de la planification n'est pas possible. Dans certains domaines d'application, ces cycles ne se sont pas révélés comme étant réalistes face au déroulement réel des processus de développement [Som06]. En effet, certains processus sont en réalité régulièrement remis en cause dans leur planification obligeant la répétition de tâches ou le retour à certaines activités considérées comme terminées. Ce décalage entre une planification rigoureuse et les besoins réels peut alors amener les acteurs à refuser les

changements non prévus et conduire à l'échec du projet de développement.

De plus, ces cycles prédictifs souffrent souvent d'un manque de visibilité sur l'avancement du travail effectué : cela est qualifié d'*effet tunnel* [Mes07]. En effet, un projet de développement se planifiant sur plusieurs mois, voire plusieurs années, peut être comparé à une boîte noire. La phase de définition des besoins peut à elle seule prendre plusieurs mois. La phase de conception/développement qui en suivra peut à son tour voir son temps de réalisation compté en mois ou en années. A cause de l'effet tunnel, il va être difficile de quantifier la véritable durée que demandera une activité ainsi que les retards qui en découleront.

Face à ces problèmes, des approches dites *agiles* ont vu le jour. Ces approches telles que Unified Process (UP), Rational Unified Process (RUP) ou eXtreme Programming (XP) décrits dans [AJ02] et [TS04] apportent alors un nouveau point de vue sur la gestion du processus de développement et sa planification.

Les approches agiles s'articulent autour de deux points clefs :

1. Elles introduisent le principe de l'itération au sein du processus de développement permettant ainsi d'éviter l'effet tunnel, car à chaque itération un état de l'avancement peut être fait.
2. Elles replacent les acteurs du développement au centre des préoccupations et favorisent les interactions au sein de l'équipe ainsi qu'avec le client.

Contrairement aux phases des cycles de vie classiques, ces itérations, aussi appelées *Sprint*, se concrétisent par une découpe du processus de développement en étapes de quelques semaines tout au plus. A chaque itération, un mini processus de développement est mis en place partant de la définition des besoins jusqu'à livraison du produit validé finalement par le client. L'intérêt de cette approche est multiple :

- Elle permet de définir une planification réaliste de la réalisation de certains logiciels en évitant ainsi des dépassements trop importants du planning.
- Elle permet de délivrer un sous ensemble opérationnel du produit final permettant au client de visualiser concrètement l'avancement du développement global (et donc de faire disparaître l'effet tunnel).
- Elle fournit au client la possibilité de changer éventuellement les besoins et les choix de fonctionnalités. Cette démarche permet d'impliquer de façon plus importante le client au sein du développement et ainsi de le rendre sensible aux conséquences liées aux changements qu'il souhaite entreprendre.
- Elle permet de détecter plus tôt les problèmes et les incohérences liées à la nature du système ou aux ambiguïtés présentes dans la spécification.
- Elle permet également une meilleure communication au sein de l'équipe de développement et aussi entre l'équipe et le client, qui idéalement doit s'intégrer à celle-ci.
- Enfin, elle permet d'avoir une meilleure maîtrise des coûts engagés puisque ceux-ci ne sont définis que sur la base de chaque itération.

Quel que soit le type de cycle de vie, l'équipe joue un rôle primordial. Elle se situe au centre de la réalisation et est dirigée par un *Chef de projet* (membre pivot du développement). Le chef de projet a pour rôle d'être le responsable principal du développement.

Il s'entoure d'une équipe définie en fonction des compétences nécessaires à la réalisation du processus de développement. Les acteurs récurrents sont par exemple : *l'analyste, le concepteur, le développeur, le testeur*, etc.

Dans les approches classiques (ou prédictives), tous les acteurs travaillent généralement de manière autonome sur des tâches spécifiques à leur domaine de compétences. Cela requiert une connaissance précise des tâches qu'ils ont à effectuer mais implique également que les interactions entre les membres doivent être clairement définies pour maintenir une cohésion et une évolution cohérente du développement. Cette vision idéale qui compartimente les rôles des différents acteurs peut avoir des conséquences néfastes. En effet, chaque acteur, spécialiste de son domaine, utilise typiquement des langages spécifiques ne faisant pas partie du champ de compétences des autres membres de l'équipe. Ainsi, la séparation des rôles rend difficile la communication entre acteurs et peut conduire à des situations d'échecs, à la remise en cause des compétences des autres acteurs ou au refus d'assumer ses propres responsabilités.

A l'inverse, dans les approches agiles, le focus est mis sur la capacité de communication et d'adaptation des acteurs. Les acteurs possédant de multiples compétences sont favorisés afin d'améliorer le travail collaboratif. Ils vont alors avoir plusieurs rôles à jouer au sein du processus de développement. Par exemple, un développeur peut être amené à devoir interagir avec le client sur la définition des besoins ou, avec le testeur pour la définition des tests de validation. L'intérêt est de favoriser l'implication de tous les membres de l'équipe dans les différentes tâches et ainsi de partager les responsabilités. Finalement, dans les approches agiles, le projet est mené par la répartition des fonctionnalités à réaliser et non comme dans les approches prédictives par la répartition des tâches à effectuer.

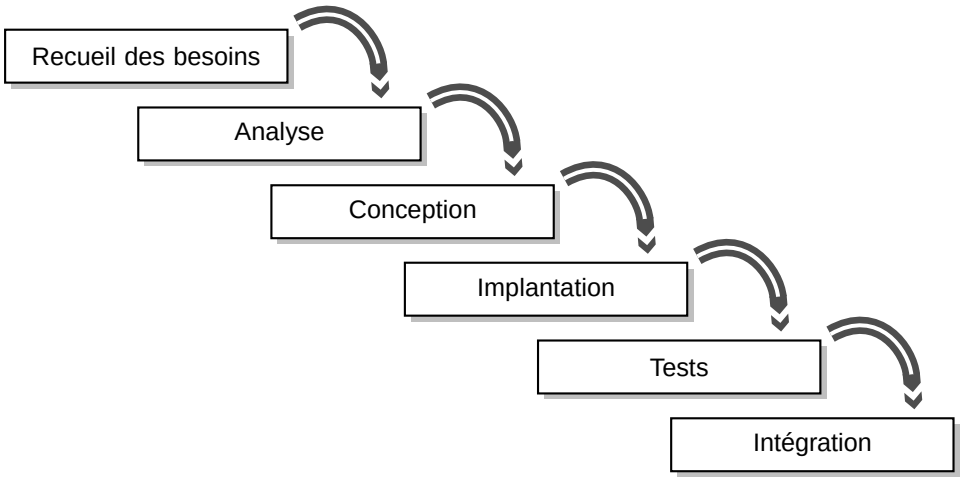
2.5.2 Cycles de vie du logiciel

Dans ce paragraphe, nous proposons de faire un état des lieux plus détaillé des cycles de vie les plus importants et les plus couramment utilisés. Après une présentation des cycles classiques comme ceux en Cascade ou en V, seront présentées des méthodes plus récentes basées sur l'agilité.

a. Cycle en Cascade

Le cycle en cascade [Roy70] est typiquement un cycle de développement prédictif. Provenant du bâtiment, il part du principe que la construction nécessite, en général, un enchaînement logique ; la couverture d'une maison ne peut pas être effectuée sans avoir préalablement fait les fondations. Il définit une démarche de développement séquentiel (figure 2.9) où chaque phase conduit à la production d'un ou plusieurs livrable(s) qui doivent être validés avant d'être utilisés lors de la phase suivante.

Le modèle en cascade nécessite la définition d'un planning détaillé qui énonce toutes les étapes et réalisations attendues. Différentes activités d'analyse, de conception, d'implantation, de tests et d'intégration sont effectuées afin de converger vers l'obtention du système

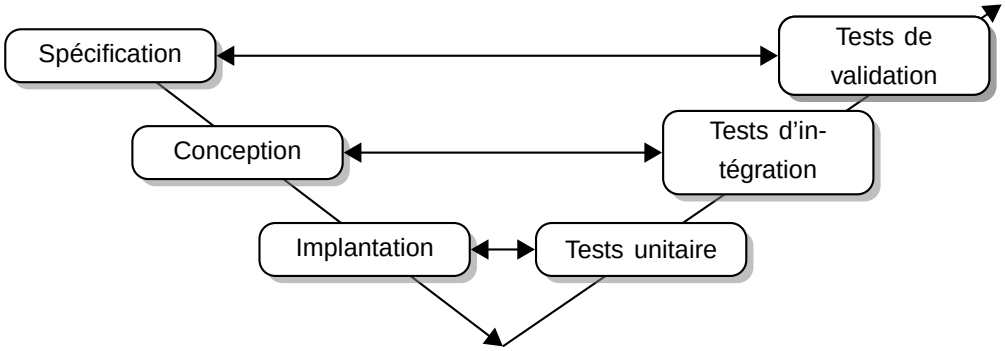


F . 2.9 – Modèle du cycle en cascade

logiciel final. Initialement, le modèle en cascade est un cycle de développement purement séquentiel, cependant, diverses possibilités d'itération ou de retour vers les phases amont ont ensuite été intégrées au modèle. Ces itérations permettent de vérifier les produits obtenus au fil du développement et ainsi fournir plus de souplesse à la conception.

b. Cycle en V

Le cycle en V [MR84] est l'un des cycles les plus connus et utilisés (figure 2.10). C'est un cycle de type prédictif qui a été défini pour remédier aux lacunes du cycle en cascade qui manque de réactivité face aux erreurs découvertes lors de la conception, du développement ou encore de l'analyse. La structure en V du cycle a l'avantage de mettre en vis à vis les activités de développement et de tests permettant de mieux préciser les documents à partager entre ces phases, notamment les rapports de tests et les modifications qu'il est nécessaire d'apporter pour corriger les erreurs. Ainsi, lors de la phase montante du cycle, toutes les réalisations doivent être testées et validées.

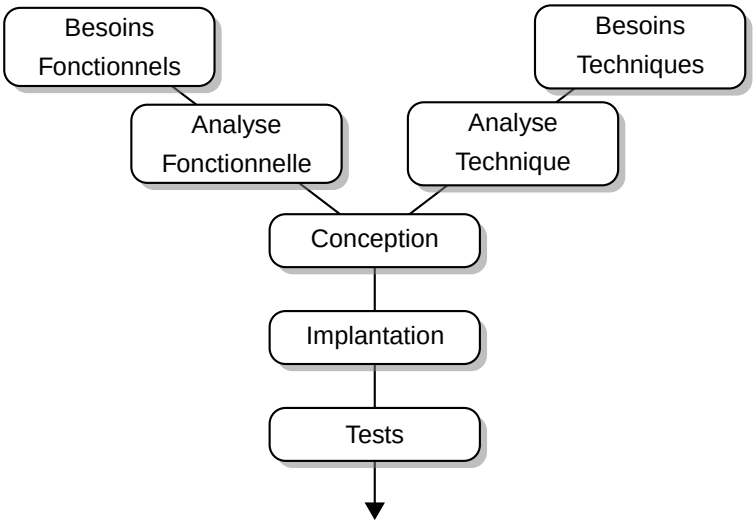


F . 2.10 – Cycle en V

Depuis les années 80, le cycle en V est considéré comme un standard du développement logiciel et de la gestion de projet dans les industries. A la suite du cycle en V sont apparues diverses variantes telles que, par exemple, le cycle en W qui propose d'effectuer deux cycles en V successivement, le premier servant à la conception d'un prototype de l'application, le second à construire l'application finale.

c. Cycle en Y

Dans son principe, le cycle de développement en Y, [RV02], est proche du cycle en cascade dont il reprend l'aspect descendant. Son intérêt est de séparer les préoccupations concernant les aspects fonctionnels liés au domaine métier et les aspects techniques liés aux solutions technologiques à employer pour la mise en œuvre.

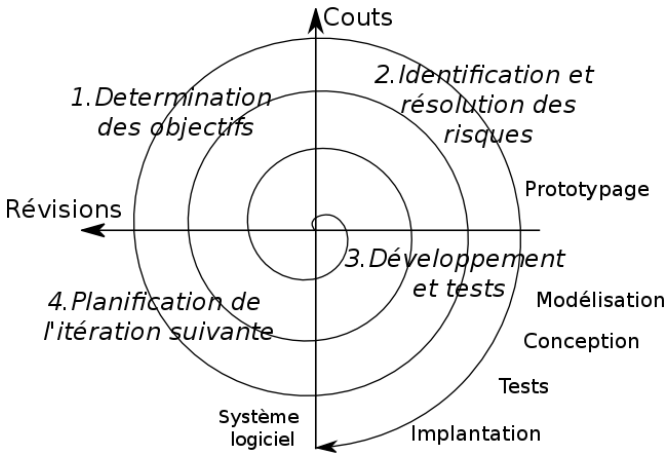


F . 2.11 – Cycle en Y

Avec le MDA, un cycle en Y orienté modèle a été proposé afin de séparer les spécifications fonctionnelles (PIM) et les spécifications techniques (PSM). Cette approche a pour avantage de fournir un modèle plus rationnel de la gestion des modèles et de l'utilisation des transformations de modèles employées lors de la phase de conception pour faire coïncider les modèles techniques et fonctionnels (employés également lors de la phase d'implantation sous la forme de génération de code).

d. Cycle en Spirale

Défini par Barry Boehm, le cycle en spirale [Boe88] est une approche itérative du cycle de développement en V. Ainsi, il en reprend l'essentiel des concepts en s'articulant autour de quatre phases importantes (figure 2.12) : la *détermination des objectifs*, la *détermination des risques*, le *développement* et les *tests* et enfin la *planification de l'itération suivante*.



F . 2.12 – Modèle du cycle en spirale

Par contre, à l'inverse du cycle en V, le modèle en spirale met un focus plus important sur l'analyse et la résolution des risques. Ceci est nécessaire car au fur et à mesure des itérations, la réalisation devient de plus en plus conséquente. Il est donc important d'évaluer correctement le risque à chaque itération sachant que toute erreur sera d'autant plus difficile à corriger que le développement sera avancé.

e. Cycle avec Prototypage Rapide (RAD)

Proposé dés la fin des années 1980, le cycle de développement avec Prototypage Rapide [Mar91] (Rapid Application Development aussi appelé semi-itératif) se situe entre une approche prédictive et une méthode agile dans le sens où il s'agit de l'une des premières approches à introduire la notion d'itération au sein du processus de développement. La méthode RAD se structure ainsi autour de cinq phases :

1. *L'initialisation* pendant laquelle le périmètre du projet et les ressources nécessaires à celui-ci sont définis. Le travail à effectuer est alors organisé par thèmes.
2. *Le cadrage* qui va définir les besoins et les fonctionnalités du produit. Cette phase est effectuée lors de réunions se basant sur des techniques d'animation spécifiques.
3. *La conception* des modèles nécessaires à l'élaboration du système logiciel proprement dit. Le client est fortement impliqué dans cette phase puisqu'il a pour rôle de valider les modèles proposés.
4. *La construction* du système logiciel par l'équipe de développement par une approche itérative selon les fonctionnalités demandées. Le client est toujours présent afin de valider les différentes réalisations.
5. *La finalisation* est une phase de validation globale du livrable final par le client.

Ainsi, les deux premières phases se placent dans une approche descendante comme celle en V ou en Cascade. Par contre, à partir de la troisième phase, le cycle de développement devient itératif en permettant d'alterner conception et validation. De plus, avec l'intégration du client dans le développement, cette démarche fournit plus de réactivité.

La méthode RAD est pionnière dans l'utilisation de l'itération et d'une intégration plus importante du client dans le processus de développement. A ce titre, elle peut être considérée comme initiatrice des méthodes agiles qui sont fondamentalement tournées vers ces concepts. Les paragraphes suivants présentent les démarches et cycles proposant une approche agile du développement.

f. Scrum

Développée en 1993, Scrum [SB01] est une approche agile qui met en avant l'intérêt des petites équipes de développement. Dans ces petites équipes, sont avant tout recherchées les compétences multidisciplinaires et la capacité d'intégration sociale des acteurs. Dans Scrum, les itérations appelées *Sprint* sont planifiées sur quatre semaines. Ces itérations sont axées sur les besoins définis par le client qui constituent alors le référentiel de travail.

Les besoins sont hiérarchisés par degré d'importance et développés selon les priorités définies par le client. Chaque jour constitue une itération pendant laquelle une réunion est organisée pour établir l'état d'avancement du projet et de vérifier que les fonctionnalités et les délais sont bien respectés.

A la fin de chaque Sprint, une réunion établissant le bilan des réalisations est menée afin d'établir l'efficacité de l'équipe et les éventuelles améliorations qui doivent être apportées. La finalité d'un Sprint est de présenter un démonstrateur au client afin que celui-ci puisse valider les réalisations.

Scrum définit un certain nombre d'acteurs qui vont intervenir lors de la mise en œuvre du processus :

1. Le *ScrumMaster* a pour rôle d'aider les acteurs du développement à communiquer au sein de l'équipe ainsi qu'avec le client. Il doit s'assurer que la philosophie et les pratiques de Scrum sont correctement suivies. Par contre son rôle n'est pas à confondre avec celui du chef de projet dont il est plutôt un conseillé.
2. Le *Client* est lui aussi acteur du développement, il a pour rôle de définir les besoins. Il doit également définir les priorités dans les fonctionnalités à réaliser. Ainsi, il participe activement à l'élaboration du produit en suivant les étapes de sa réalisation afin de pouvoir en valider la finalité.
3. Enfin, l'*Équipe* qui est constituée de l'ensemble des corps de métiers nécessaires à l'élaboration du produit. Ces métiers sont classiquement ceux rencontrés lors des développements logiciels classiques (Développeur, Analyste, Testeur, etc.)

Pour finir, la méthode Scrum s'appuie sur la notion de *Visibilité* pour qualifier et quantifier les résultats de l'équipe. Des critères de validation doivent exister afin de définir si une fonctionnalité a été complètement réalisée ou non. Des critères d'*Inspection* doivent être définis afin de déterminer l'existence d'écarts entre la réalisation concrète et l'objectif final. Enfin, la notion d'*Adaptation* permet, lors d'écarts trop importants détectés pendant des *Inspections*, de modifier la gestion interne de l'équipe afin d'éviter que ces écarts ne s'amplifient.

g. Unified Process UP

Tout comme RAD, UP [JBR99] est un cycle de développement qui se positionne entre les approches prédictives et les approches agiles. Ce cycle de vie est né de l'approche orientée objet issue de la collaboration de Ivar Jacobson, Grady Booch et James Rumbaugh.

L'approche UP tente de rationaliser les processus de développement logiciel en fournissant un guide de bonne pratique fondé sur six points :

1. Avoir une démarche incrémentale et itérative pilotée par les risques et les cas d'utilisation.
2. Avoir une gestion rigoureuse des exigences.
3. Centrer le développement sur l'architecture.
4. Faire une modélisation graphique des exigences.
5. Contrôler en permanence la qualité.
6. Contrôler les changements éventuels à effectuer.

Ces six bonnes pratiques prônées par UP sont en l'occurrence ce qui permet de rattacher ce cycle de développement aux méthodes agiles. En effet, contrairement aux démarches de développement classiques, UP oriente naturellement le développement pour que celui-ci soit à même de répondre aux changements.

UP définit une démarche de développement intégrant les neuf disciplines majeures nécessaires au développement : *la modélisation métier*, *la définition des exigences*, *l'analyse et la conception*, *l'implantation*, *les tests*, *le déploiement*, *la gestion des changements*, *la gestion du projet* et enfin *la prise en compte de l'environnement*. Basé sur ces disciplines qui seront employées à chaque itération, le processus de développement va alors pouvoir suivre les quatre phases du processus qui consistent en :

1. Le *début* dont l'objectif est d'unifier les points de vue de chacune des disciplines sur l'ensemble du projet.
2. L'*élaboration* qui a pour objectif de définir et de valider l'ensemble des modèles permettant la conception du système logiciel.
3. La *construction* qui doit fournir une version documentée et fonctionnelle du logiciel.
4. La *transition* dont le but est de finaliser le système.

Avec Unified Software Process Metamodel (USPM) [RK03], les démarches de développement se sont orientées vers la modélisation des processus en suivant le credo issu de la publication de Lee Osterweil [Ost87] : *Software processes are software too*. Traduit littéralement que les processus de développement de logiciel sont aussi une forme de logiciel. Cette publication montre la nécessité de maîtriser le processus de développement en modélisant sa structure et son évolution.

2.5.3 Modélisation des processus et ALM

Selon qu'ils soient prédictifs ou agiles, les cycles de vie proposent des structures ou des méthodes différentes qu'il est nécessaire de modéliser. Certains cycles, de par leur nature prédictive, vont plutôt se focaliser sur les tâches et activités à réaliser. D'autres, issus des démarches agiles, vont plutôt se focaliser à chaque itération sur la réalisation des différentes fonctionnalités afin de constituer progressivement le système dans son ensemble. Cependant leur objectif reste le même : fournir une démarche rationnelle.

Les processus de développement sont fondés généralement sur les flots de travaux. Pour modéliser et intégrer ou automatiser les flots de travaux, différentes approches nommées ALM (Application Lifecycle Management) existent. Les ALM sont des méthodes outillées permettant une meilleure gestion des différents processus intervenants durant le développement global d'un produit.Ils intègrent généralement les langages de modélisation, les activités à réaliser et les différentes ressources (acteurs) du développement. Parmi les ALM, il est possible de citer : Visual Studio Team System[1] de Microsoft, Jazz[2] de IBM [GPM08], ALF[3] sur la plateforme Eclipse ou encore OpenALM[4] de Borland.

Les ALM proposent une approche globale du processus de développement. Ils vont prendre en compte chacune des activités à réaliser et proposer un modèle de processus adapté aux spécificités d'une réalisation. Ainsi, les ALM vont fournir un contexte pour effectuer des activités de :

- Gestion du processus de développement, de l'équipe, des financements, etc.
- Gestion des flots de travaux qui intègrent les différentes étapes du développement.
- Analyse des besoins définis par le client.
- Modélisation des éléments du système par l'utilisation de langages adaptés.
- Conception des éléments précédemment modélisés.
- Gestion de l'implantation sur les différentes plateformes.
- Test du logiciel et validation des spécifications.
- Gestion de la réalisation.
- Gestion du changement afin de maîtriser les changements éventuels des besoins du client, ou même de la composition de l'équipe de développement.
- Gestion du suivi et de la documentation afin de continuer à faire évoluer le logiciel après sa distribution selon d'éventuels nouveaux besoins.

Par la mise en place de ces activités, les ALM ont pour objectif d'accroître la productivité en facilitant la reproductibilité des processus de développement. En effet, ils vont inclure et capitaliser les bonnes pratiques identifiées lors des développements précédents. Ils permettent également un accroissement de la qualité en fournissant un moyen efficace d'identification des erreurs et de leurs résolutions. Ensuite, avec une maîtrise plus importante des flots de travaux, les ALM apportent de la flexibilité ainsi qu'une meilleure gestion des coûts et des temps de production.

[1] http ://msdn.microsoft.com/fr-fr/teamsystem/default.aspx
[2] http ://www-01.ibm.com/software/rational/jazz/
[3] http ://www.eclipse.org/alf/
[4] http ://www.borland.com/us/solutions/index.html

2.6 Conclusion

Dans cette première partie, le problème du développement des systèmes logiciels a été présenté. La problématique principale qui a été posée est la prise en compte des nombreuses sources de complexité. Les systèmes logiciels étant omniprésents dans notre environnement, ils sont de nature variée et ils comportent de nombreuses propriétés à prendre en compte telles que la fiabilité, la coût de production ou la longévité. Pour repondre à cette difficulté, de nombreuses démarches ont été proposées dans la littérature relevant du domaine du génie logiciel.

Pour concevoir un logiciel, il est nécessaire de s'appuyer sur des langages de modélisation et des méthodes. En particulier, des langages de modélisation standards comme UML [OMG10a] ou dédiés à la spécification d'architecture comme AADL [Dis04] permettent aujourd'hui des développements plus rationnels. Utilisés seuls, ces langages de modélisation ne permettent cependant pas de gérer l'ensemble de la complexité du développement, ce qui nécessite la proposition d'approches facilitant l'intégration de ces langages au sein d'un même contexte de développement.

Des approches permettant de composer les concepts de différents langages de modélisation se retrouvent dans Ptolemy, MIC ou MDA-IDM. Ces approches sont indéniablement une avancée dans la gestion des connaissances multiples qui doivent cohabiter pour la mise en œuvre de logiciels. En proposant une architecture orientée acteurs, Ptolemy fournit un ensemble de domaines avec lequel il va être possible de concevoir des modèles variés et hétérogènes. Par ailleurs, en proposant une architecture orientée modèles (avec le métamodèle FCO), MIC fournit ici un paradigme de modélisation permettant de concevoir des langages de modélisation spécifiques. Enfin, avec une approche orientée modèles plus générique, MDA et IDM fournissent des concepts généraux pour la définition des langages de modélisation nécessaires au développement de logiciels variés. Ces nouvelles approches doivent être combinées et intégrées au sein des processus de développement logiciel pour en tirer pleinement profit.

L'état de l'art fait le point sur les modèles de cycle de développement de référence ainsi que ceux plus récents qui offrent de nouveaux concepts. Les cycles de vie tels que ceux en Cascade, en V, ou en Y ont été décrits. Ces cycles, dits prédictifs, ont permis par le passé de poser les bases d'un développement logiciel plus rationnel intégrant les aspects fondamentaux de la bonne conduite d'un projet (Analyse des besoins, Conception, Implantation, Test). Cependant, ces cycles, par leur rigidité, ne sont pas toujours adaptés aux besoins de certains projets de développement actuels. Ainsi, des cycles et méthodes tels que le RAD, UP ou encore Scrum sont utilisés afin de fournir plus de souplesse tout en rétablissant le rôle central que doivent avoir les différents acteurs du développement.

Afin d'aller plus loin, les ALM fournissent à leur tour des solutions technologiques permettant une maîtrise globale du développement. Pour cela, ils gèrent, en plus des activités classiques de développement, d'autres activités comme, par exemple, la gestion du suivi du produit logiciel après sa distribution qui sont connexes au développement mais restent indispensables.

Cependant dans cette approche, le processus de développement est résolument orienté vers la gestion du projet de conception. Elle n'établit pas de liens suffisant ni vers les langages de modélisation nécessaires au développement ni vers les possibilités d'intégration de ces langages telles que l'on peut les trouver dans des approches comme Ptolemy, MIC, MDA-IDM.

Chapitre 3

L'Ingénierie Dirigée par les Modèles

3.1 Généralités

L'Ingénierie Dirigée par les Modèles (IDM) est une branche du génie logiciel dont l'objectif est d'opérationnaliser et de capitaliser le concept de *modèle*. Pour cela, l'IDM se base sur divers concepts comme les méta-modèles, les langages de modélisation et les transformations de modèles [Fav04]. Historiquement, l'IDM peut être vue comme une résultante de l'initiative MDA (Model Driven Architecture) [BBF04] dont l'objectif était de réduire l'écart existant entre un logiciel dédié à un métier particulier et une plateforme technologique spécifique sur laquelle le logiciel doit s'exécuter [Béz04].

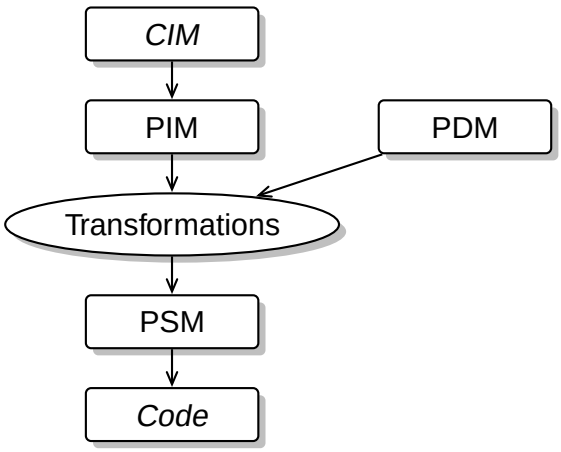


Figure 3.1 – Principe du processus MDA

Pour cela, l'initiative MDA considère le processus de développement logicielle selon divers modèles (figure 3.1) :

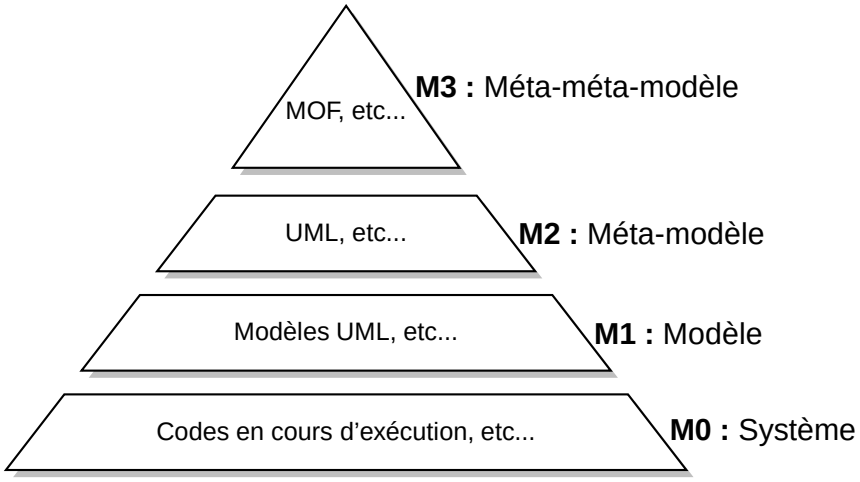
- un modèle indépendant des aspects logiciels appelée CIM (Computer Independant Model) qui contient les informations relatives au domaine métier,
- un modèle indépendant de la plate-forme appelée PIM (Platform Independant Model) qui contient les informations relatives au domaine métier de l'application,

– un modèle de description de la plate-forme appelée PDM (Platform Description Model) qui contient les informations relatives à l'intégration du PIM au sein de la plate-forme,
– un modèle spécifique de la plateforme ou PSM (Platform Specific Model) qui contient tous les détails techniques liés à la réalisation.

Plus précisément, la démarche préconisée par l'initiative MDA consiste, à partir d'un modèle dédié à un métier et selon un ou plusieurs contextes spécifiques de déploiement, à obtenir par raffinement un modèle "adapté" à une plate-forme particulière. Cette démarche s'appuie alors sur un cycle en Y (comme illustré sur la figure 3.1).

3.2 Pyramide de méta-modélisation

MDA s'appuie sur une architecture pyramidale à quatre niveaux, figure 3.2 permettant d'organiser les modèles selon différents niveaux d'abstraction.



F . 3.2 – Pyramide de modélisation de l'OMG

Plus précisément, les niveaux considérés sont décrit de la façon suivante :

Niveau M3 : Dans ce niveau situé en haut de la pyramide résident des entités, nommées *méta-méta-modèles*, utilisées pour décrire des *langages de modélisation*. La propriété principale d'un *méta-méta-modèle* est la réflexivité, c'est à dire la capacité à s'auto-décrire. Dans l'approche MDA, ce méta-méta-modèle est le Meta Object Facility (MOF), [OMG06].

Niveau M2 : Ce niveau contient l'ensemble des méta-modèles conçus à partir d'un méta-méta-modèle du niveau M3. A ce niveau se trouve en particulier le méta-modèle et la spécification pour le langage de modélisation UML, [OMG10a], [OMG10b].

Niveau M1 : Ce niveau contient des modèles issus de l'activité de modélisation avec par exemple, un modèle UML modélisant un système particulier.

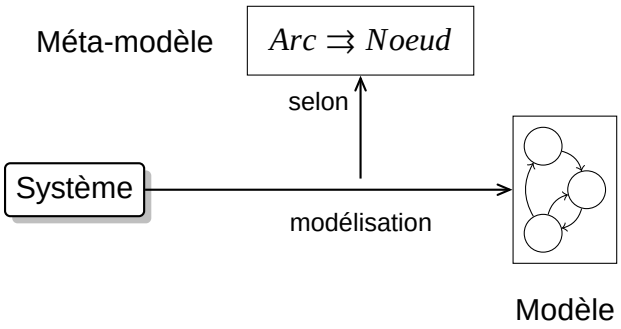
Niveau M0 : Ce niveau situé en bas de la pyramide représente l'ensemble des systèmes à modéliser.

3.3 Définitions et Concepts

L'Ingénierie Dirigée par les Modèles offre un cadre permettant de relier, de façon rationnelle, les concepts issus de théories définies dans un domaine (par la définition d'un méta-modèle) ou de domaines différents (par des transformations de modèles), [FEB06].

3.3.1 Modèles et objets étudiés

L'acte de modélisation est indispensable à la compréhension ou à la conception de systèmes. Un exemple de modélisation est donnée par la figure 3.3.



F . 3.3 – Activité de modélisation

Plus précisément, l'IDM introduit la notion d'objet d'étude (ou système sous étude). Trois types d'objets sont distingués via les entités physiques, les entités numériques et les entités abstraites :

Une entité physique représente un système physique concret et observable sur lequel l'homme peut agir matériellement.

Une entité numérique est un système appartenant à la classe des objets informatiques.

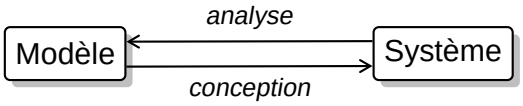
Une entité abstraite est une entité purement conceptuelle comme un objet mathématique (fonction, ensemble, etc.).

A partir de là, un objet d'étude peut avoir deux rôles. Il peut être une entité existante étudiée afin d'en comprendre le fonctionnement. Il peut également être le résultat d'un processus de réalisation où la phase de la modélisation précède celle de sa mise en œuvre. Il existe donc deux relations possibles entre un modèle et un objet d'étude, figure 3.4.

> *Un modèle peut être défini alors comme une abstraction réalisée dans une intention et un contexte particulier*.

L'intérêt des modèles est qu'ils permettent de masquer des détails non pertinents. Ils facilitent ainsi la compréhension des problèmes et l'élaboration de leurs solutions.

De façon générale, un système (ou processus) se compose d'entités en relation. Un modèle est donc une représentation particulière (établie à l'aide d'un formalisme adapté) de



F . 3.4 – Relation entre la notion de modèle et celle d'objet étudié

certaines de ces entités et/ou relations identifiées selon leurs caractéristiques. Par exemple, si le système est une chaîne de production, alors un formalisme adapté pourrait être celui des Réseaux de Petri. Dans ce contexte, un point de stockage se représente par une Place (un cercle), une machine transformant des pièces entre deux places se représente par une Transition (une barre et des arcs montrant les places entrantes et sortantes) et les produits véhiculés dans le processus correspondent à des Jetons (des cercles pleins noirs). Les concepts du formalisme que sont Place, Transition et Jeton définissent alors un langage de modélisation pouvant être spécifié par un méta-modèle.

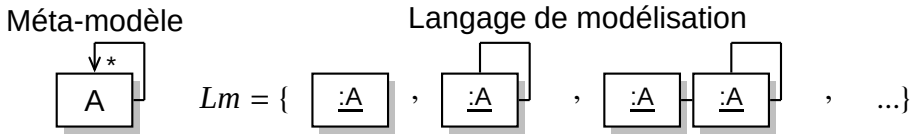
3.3.2 Langages de modélisation et méta-modèles

Un langage L se définit formellement comme un ensemble de termes. L est généralement généré par une grammaire, c'est à dire un ensemble de règles pour produire les termes du langage. La figure 3.5 propose un exemple de grammaire et de langage associé.

Grammaire	Langages
$< L >:= \varepsilon \mid < A > L$	$L = \{\varepsilon, a, b, aa, ab, ba, baba, ...\}$
$< A >:= a \mid b$	$L_a = \{\varepsilon, a, aa, aaa, aaaa, ...\} \subseteq L$

F . 3.5 – Exemple de langage

Un langage de modélisation L_m est un langage dans lequel les termes sont des modèles. Ce langage peut être spécifié par un méta-modèle. Dans ce cas, ce méta-modèle joue le même rôle qu'une grammaire en décrivant l'ensemble des modèles pouvant être produits. La figure 3.6 présente un exemple de méta-modèle pour un hypothétique langage de modélisation décrivant des structures d'entités A en interaction.



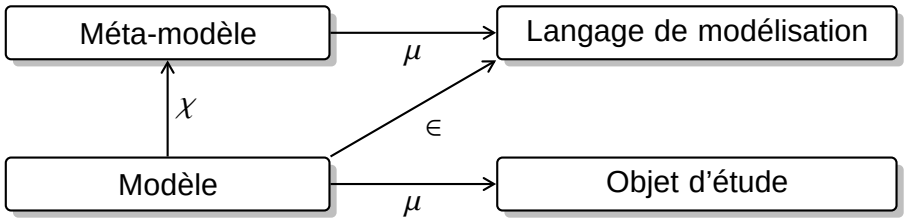
F . 3.6 – Exemple de langage de modélisation

Il est à noter que le méta-modèle est souvent associé à la syntaxe abstraite du langage de modélisation, c'est-à-dire un ensemble des modèles représentés sous la forme de graphes abstraits tels que présentés par la figure 3.6 (bien qu'ici représentés à l'aide d'un

diagramme UML qui sert de syntaxe concrète). La syntaxe concrète est généralement définie et associée à la syntaxe abstraite de façon séparée. Dans l'exemple de la chaîne de production, les concepts de Jeton, de Transition et de Place font partie de la syntaxe abstraite des réseaux de Petri. Leur représentation respective sous la forme de cercle plein, de barre (avec des arcs entrants et/ou sortants) et de cercle constitue alors la syntaxe concrète des réseaux de Petri.

Les méta-modèles sont souvent complétés par des propriétés exprimées, par exemple, en OCL, [OMG10c] sous la forme de prédicat. Une propriété peut alors être vue comme une fonction $p : M \rightarrow Bool$ dans lequel M représente un modèle. Par exemple, la propriété p, *"Tout A est associé à un et un seul A"* et le sous-ensemble de L_m vérifiant p correspond alors à l'ensemble des modèles dits *valides* pour le méta-modèle spécifiant L_m.

La figure 3.7 décrit les relations existant entre les concepts présentés. Ainsi, un méta-modèle décrit ou modélise (μ) un langage de modélisation. Un modèle est alors un élément du langage ($\in$) et est dit conforme (χ) au méta-modèle.



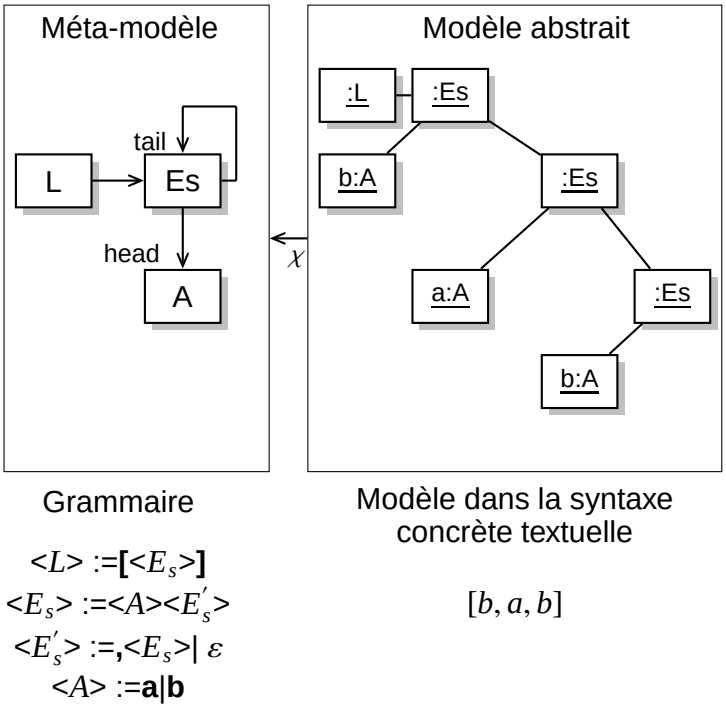
F . 3.7 – Architecture de méta-modélisation

De façon plus générale, une analogie peut être faite avec le Grammarware qui étudie les programmes informatiques et les grammaires des langages de programmation [Béz04]. En effet, comme illustrée par la figure 3.8 représentant un modèle de liste, la notion de grammaire est proche de celle de méta-modèle.

Ce parallèle conduit à considérer que la notion de langage de programmation décrit par une grammaire a le même rôle que le langage de modélisation décrit par un méta-modèle. Dans les deux cas, le langage représente l'ensemble des programmes/modèles conformes à une grammaire/méta-modèle. Enfin ces deux entités que sont les programmes et les modèles peuvent être, elles aussi, mises en parallèle car leur objectif est le même : représenter l'objet étudié. Cela dit, là où le grammarware utilise essentiellement des modèles arborescents (avec les arbres syntaxiques, par exemple), le modelware considère des graphes.

Pour résumer, il est admis que :
- le méta-modèle spécifie la syntaxe abstraite d'un langage de modélisation,
- l'adjonction d'une syntaxe concrète à ce méta-modèle facilite la définition de modèles particuliers (représentable sous forme textuelle ou graphique).

La figure 3.8 donne un exemple de liaisons possibles entre le concept de méta-modèle, de modèles abstraits et concrets et de grammaire.

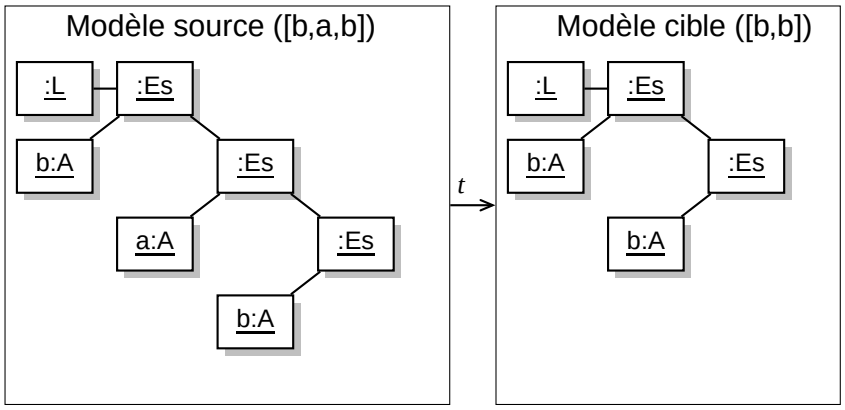


F . 3.8 – Parallèle Grammarware-Modelware

3.4 Transformations de modèles

> *Une transformation de modèles se définit comme étant la modification ou la synthèse d'un modèle en suivant des règles définies au niveau des concepts d'un ou plusieurs méta-modèles, [Fav04].*

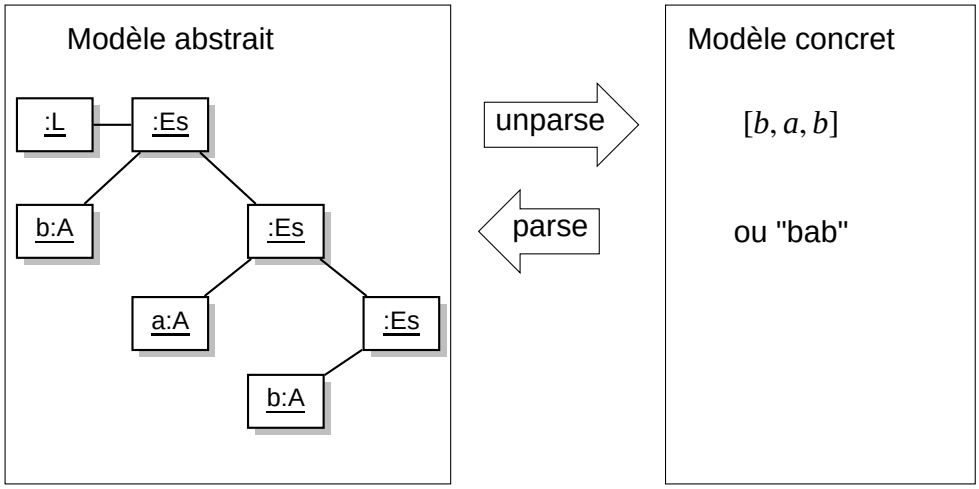
La figure 3.9 propose un exemple de transformation *t* pour le modèle de liste *L* présenté plus haut (figure 3.8). La transformation (ici endogène) consiste simplement à sélectionner les éléments de type *b*.



F . 3.9 – Exemple de transformation

Puisque les transformations de modèles sont définies à partir des méta-modèles plusieurs types de transformations sont envisageables avec plus précisément des transforma-

tions de modèles s'appuyant sur la syntaxe concrète et des transformations de modèles s'appuyant sur la syntaxe abstraite. Entre autre, la figure 3.9 présente une transformation s'appuyant exclusivement sur la syntaxe abstraite. A l'inverse, La figure 3.10 (et 3.8) donne un exemple de liaison possible entre les éléments abstraits manipulés par l'IDM (partie gauche) et une représentation concrète possible sous la forme d'expressions manipulées par d'éventuels outils spécifiques (partie droite). La transformation *t* est donc ici une paire de transformations (*parse* et *unparse*) pour passer d'un modèle à une expression de ce modèle, ou inversement.



F . 3.10 – Liaison entre représentation abstraite et concrète

Si la transformation possède ou non le même méta-modèle en entrée et en sortie, il est possible de distinguer deux types de transformations :

Les transformations endogènes : elles ont pour domaine et codomaine le même langage de modélisation (donc le même méta-modèle). Ces transformations sont utilisées pour décrire les opérations de manipulations/modifications de modèles (figure 3.9).

Les transformations exogènes : elles permettent, pour leur part, de changer de langage de modélisation. Ce changement permet ainsi de changer de contexte de modélisation.

Le problème essentiel que posent les transformations de modèles est la préservation de la sémantique et des propriétés associées à un modèle source. En effet, le changement de langage de modélisation impose de s'assurer que les transformations préservent les propriétés jugées fondamentales contenues dans les modèles.

De façon à proposer des transformations de modèles préservant la sémantique, une façon plus formelle d'utiliser l'IDM doit être envisagée. Ainsi, des approches basées sur les langages fonctionnels telles que [TT09], [TCT08] et [TTC08] proposent d'utiliser l'aspect formel et mathématique de ces langages afin de parvenir à mieux garantir la préservation de la sémantique lors des transformations.

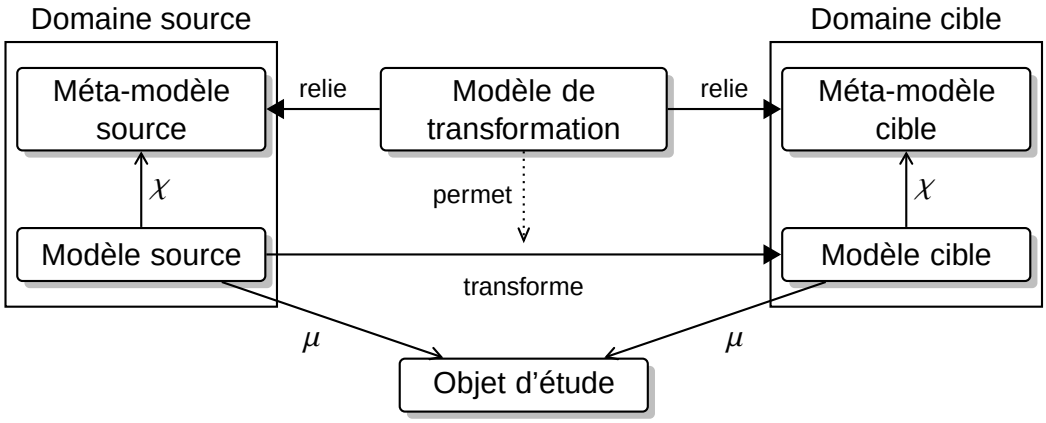
Les transformations de modèles peuvent être organisées en deux grandes familles :

Les transformations horizontales : elles ne changent pas le niveau d'abstraction des modèles manipulés (par exemple, refactorisation, optimisation). Ces transformations

sont souvent situées au sein d'une même activité et ont pour intérêt de permettre l'intégration de divers langages de modélisation.

Les transformations verticales : elles changent le niveau d'abstraction (par exemple, génération de code source ou rétro-ingénierie). Les transformations verticales sont des transformations permettant souvent de changer d'activité dans le processus de développement.

Pour ces deux types de transformations, il est possible de distinguer des transformations de modèles abstrait à modèles abstrait(M2M) et des transformations de modèles abstraits vers des syntaxes concrètes (M2C). Elles vont permettre de bénéficier des outils spécifiques à des domaines.



F . 3.11 – Modèle de transformation de modèles à modèles

Un modèle de transformation générique est donné sur la figure 3.11. Celui-ci consiste en *un méta-modèle source* et un *méta-modèle cible* (qui peut éventuellement être le même que le méta-modèle source). Ces méta-modèles vont être reliés par un ensemble de liens établissant *les règles de transformation*. Celles-ci sont établies à l'aide de langages particulier comme QVT [OMG08b] ou ATL [JK05]. Ainsi, le modèle de transformation, une fois appliqué à un *modèle source*, fournit *un modèle cible* conforme au méta-modèle associé.

Basées sur le même principe, les transformations de modèles vers des syntaxes concrètes sont des transformations permettant de relier la syntaxe abstraite d'un langage de modélisation avec une syntaxe concrète particulière (textuelle ou graphique) permettant de tirer profit d'outils et/ou de plates-formes d'exécution spécifiques.

Dans le cadre des transformations vers des syntaxes concrètes graphiques, il est possible d'utiliser Graphical Modeling Framework [GMF09]. De même, pour des transformations vers des syntaxes concrètes textuelles, des outils tels que Sintaks [MFF06] ou Xpand [EFH07] peuvent être utilisés.

3.4.1 Tissage et composition de modèles

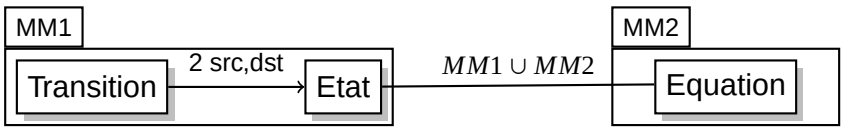
Les transformations de modèles ont certaines limites. En effet, dans le cas où la sémantique des langages de modélisation est différente, il n'est pas possible de définir une

transformation de modèles. Pourtant il est souvent nécessaire de manipuler les concepts des différents langages de modélisation au sein d'un même modèle.

Pour pallier aux limites des transformations de modèles, il est possible d'introduire le concept de composition de modèles. Deux types de composition peuvent être définis :

Composition hiérarchique : Ce type de composition considère l'inclusion des concepts utilisés par deux ou plusieurs langages de modélisation. Un exemple est donné par les automates à états finis dit hybrides qui définissent des équations récurrentes ou différentielles au sein de chacun des états des modèles réalisés (figure 3.12).

Composition modulaire : Ce type de composition considère par contre une mise en parallèle des concepts des différents langages de modélisation. Ainsi, les concepts mis en relation peuvent être multiples. Un exemple de ce type de composition est donné dans la partie illustration de ce manuscrit (Chapitre 5.4) entre le méta-modèle de l'algèbre de processus FSP et le méta-modèle de la logique temporelle LTL.



F . 3.12 – Exemple de composition avec des automates hybrides

Le principal problème rencontré lors de l'utilisation de la composition est que des propriétés mises en évidence au sein de modèles séparés peuvent ne plus être satisfaites une fois les modèles composés.

3.5 Planète IDM

3.5.1 Contributions

L'IDM a apporté de nombreuses contributions au domaine du génie logiciel. La première contribution fondamentale a été la définition d'un standard pour décrire des langages de modélisation (et des méta-modèles) avec le MOF [OMG06]. Le MOF se situe, pour rappel, au sommet de la pyramide de modélisation et propose les concepts de *Class* (associé à des *Property* et *Operation*), de *Type* et de *Package*, pour spécifier des méta-modèles pour des langages de modélisation. Le MOF étant composé d'un très grand nombre de concepts, différentes variantes sont apparues avec en particulier EMOF (Essential MOF) ou Ecore utilisés par des méta-outils, [OMG06, FEB06].

De nombreux outils ont été développés sur les concepts de l'IDM. Parmi eux, il est possible de citer MetaEdit [Met05], Dome [Hon09], GME (Generic Modeling Environment) [KSL03] ou plus récemment EMF (Eclipse Modeling Framework) [BSM03], sur lequel ont été déployés Kermeta [MFJ05], ATL [JK05], Sintaks [MFF06] ou encore Xpand [EFH07].

Il est aussi possible de citer les environnements OpenEmbeDD[1], TopCaseD [Top05] et

[1] http ://openembedd.inria.fr

OpenArchitectureWare[2] qui sont basés sur l'IDE Eclipse et qui intègrent les outils décrits plus haut. Ces derniers ont été utilisés dans la suite pour mettre en œuvre la proposition (chapitre 5 et 6).

Des travaux ont également entrepris de définir des référentiels de méta-modèles utilisables dans les développements logiciels. Par exemple, la bibliothèque AtlanticZoo est basée sur le langage KM3 (Kernel Meta-Meta-Model), [JB06]. Au sein de cette bibliothèque de méta-modèles, existent notamment :

- des modèles de langages de modélisation standards tels que UML, SysML ou AADL,
- des modèles de langages de programmation tels que C++ ou Java,
- des modèles de langages spécifiques à des domaines tels que LaTeX, SQL ou encore celui des Réseaux de Petri.

Il existe d'autres référentiels de méta-modèles pouvant être cités. En particulier, [BSM03] présente une famille de méta-modèles exprimés cette fois ci avec le langage Ecore. Ce référentiel est compatible avec le précédent car il existe un modèle de transformation écrit en ATL entre KM3 et Ecore. De façon analogue, des référentiels de transformations sont disponibles et permettent de relier/transformer deux modèles ayant des méta-modèles différents mais appartenant à une même famille. ATL fournit une telle bibliothèque[3].

3.5.2 Utilisations de l'IDM dans les processus de développement

Les parties précédentes montrent que l'IDM, qui repose sur deux concepts fondamentaux (avec les méta-modèles et les transformations), a permis de capitaliser ou d'opérationnaliser un ensemble de connaissances liées à plusieurs domaines différents. Ainsi, des familles de référentiels ont été définies et permettent alors la prise en compte de plusieurs langages (de modélisation) au sein d'un même outil (EMF, par exemple).

Les langages utilisés peuvent être soit graphiques (avec des diagrammes UML, par exemple) soit textuels. Les modèles graphiques facilitent la compréhension d'un système, à analyser ou à concevoir et les modèles textuels permettent l'interaction avec des outils spécifiques à un domaine. Dans la plupart des cas, des liaisons (ou modèles de transformations) ont déjà été établies entre ces différentes représentations, et un modèle graphique (un diagramme des classes, par exemple) trouve souvent une interprétation dans un ou plusieurs langages de programmation (Java ou C++ par exemple) ou inversement.

Un problème demeure, l'intégration de méta-modèles ou de transformations requiert des compétences particulières (différentes mais complémentaires de celles utilisées par un domaine particulier). L'IDM est donc un domaine en soi, permettant de modéliser l'expertise d'autres domaines.

La connaissance de méta-modèles peut fournir un support "pédagogique" pour mieux comprendre les concepts liés à un ou plusieurs domaines. Cela permet aussi d'aborder les différents langages de modélisation associés aux outils de ces domaines.

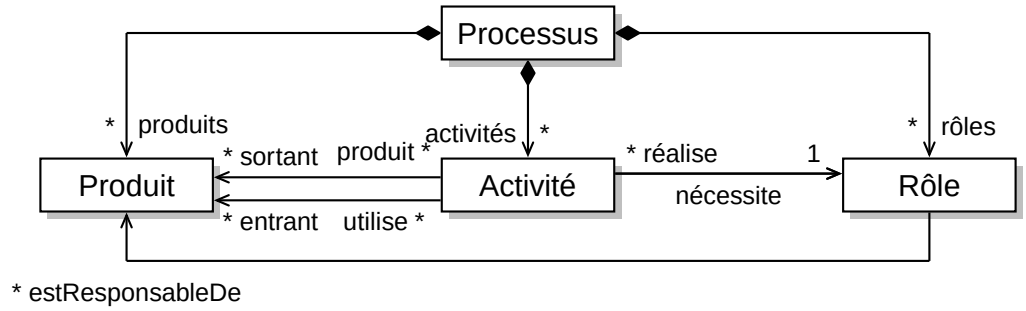
[2]http ://www.openarchitectureware.org/

[3]http ://www.eclipse.org/m2m/atl/atlTransformations/

Une liaison peut être établie entre les concepts de l'IDM et les processus de développement. Les méta-modèles ou les langages de modélisation sont souvent associés à une activité particulière qui s'inscrit dans un processus plus global. Les transformations, permettant de passer d'un modèle à un autre, peuvent être ainsi vues comme un changement d'activité dans un processus. Ce constat est à l'origine de l'idée défendue dans le chapitre 4 et requiert la modélisation du processus de développement.

En partant du constat que les différents types de méta-modèles de processus existant (comme SPEM [OMG08a], EPM [HK89], IBIS [KR70], NATURE [RSM95], etc) ne permettent pas individuellement de couvrir selon le contexte l'ensemble des besoins de modélisation des processus de développement, [HFR09] propose une méthode pour construire un méta-modèle unifié, multi-points de vue et adapté, intégrant les concepts clefs essentiels au contexte considéré.

Pour terminer cette présentation des concepts IDM, la figure 3.13 présente un modèle conceptuel simplifié de processus de développement. Ce modèle simplifié s'appuie sur le méta-modèle du standard SPEM (Software & Systems Process Engineering Metamodel) utilisé pour décrire des flots de travaux particuliers à chaque entreprise.



F . 3.13 – Modèle conceptuel de SPEM

Les flots de travaux se définissent par une succession de tâches à effectuer fournissant divers produits, utilisés à leur tour lors des tâches suivantes. Chacune de ces tâches fait intervenir différents acteurs aux rôles et aux responsabilités clairement identifiés. Le modèle s'articule autour de la notion pivot de *Processus* qui agrège des produits, des activités et des rôles.

Plus précisément, il s'agit d'effectuer un certain nombre d'*Activités* visant un objectif et réalisées par des acteurs. Par exemple, Spécifier, Réaliser, Tester sont des activités standards à la plupart des processus décrits dans le chapitre 2 ; seul le séquencement de ces activités change ainsi que les concepts/langages utilisés au sein de chaque activité.

Les acteurs tiennent certains *Rôles* afin de concevoir les différentes parties du système logiciel sous la forme de *Produits* échangés (c'est à dire entrants ou sortant d'une activité). Par exemple, la Spécification consiste à transformer un cahier des charges généralement décrit en langage naturel (entrant) en un modèle (UML) permettant d'aboutir à la réalisation (sortant).

Dans ce travail, le choix de SPEM se justifie car ce méta-modèle semble le plus appro-

prié pour appuyer la contribution (chapitre 4) où un rapprochement entre les concepts de l'IDM (*Modèle* et *Transformation*) et ceux des processus est établi.

3.5.3 Outils et environnements

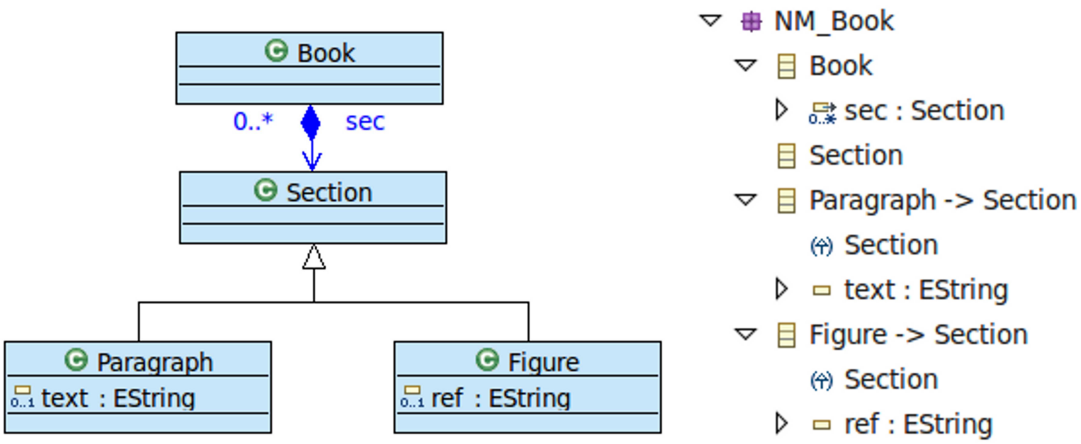
Les exemples développés dans les chapitres 5 et 6 ont profités du méta-outil TopCaseD [Top05] qui intègre déjà un certain nombre de méta-modèles. L'outil s'appuie sur l'environnement Eclipse complété du support de modélisation EMF cité précédemment, [BSM03].

Le choix de TopCaseD (par rapport aux autres outils IDM cités plus haut, c'est à dire GME, MetaEdit, Dome, etc.) est justifié par le fait que le domaine d'application considéré (chapitres 5 et 6) est celui des Systèmes à Événements Discrets (SED), et que TopCaseD se veut dédié à ce type de systèmes.

De façon générale, TopCaseD (et la plupart des autres méta-outils) repose sur une mise en œuvre d'un langage de méta-modélisation (ici, Ecore). Il est alors possible de spécifier un, ou plusieurs, méta-modèles à l'aide de menus contextuels ou d'un langage spécifique textuel ou graphique. En particulier, TopCaseD permet de décrire les composants d'un méta-modèle à l'aide d'un langage proche de celui utilisé par le diagramme des classes UML (entre autre, Ecore Diagram).

Ainsi, un méta-modèle se compose d'un ensemble de concepts représentés par des *EClass*, caractérisées par des *EAttributes* et des *EOperations*, reliées par des *Associations*, des *Aggregations* et des *Generalizations*. La persistance se réalise en XMI ; déclinaison XML[4] pour la description de modèles.

La figure 3.14 présente ces deux représentations possibles pour un méta-modèle de livres.

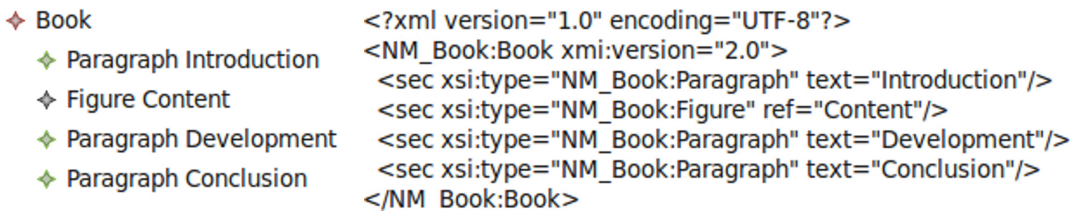


F . 3.14 – Exemple de méta-modèle EMF
(books.ecoredi à gauche et books.ecore à droite)

[4]eXtensible Markup Language

A partir d'un méta-modèle, TopCaseD (et EMF) propose de générer les éléments nécessaires pour définir un plugin pouvant être intégré à l'outil et utilisable alors pour réaliser des modèles. La démarche consiste à créer un "générateur de modèles" en précisant le méta-modèle Ecore, puis à exécuter ce générateur/plugin. Il est alors possible de créer un modèle de livre.

La figure 3.15 présente un exemple réalisé avec TopCaseD avec la vue proposée par l'outil (à gauche) et la sérialisation XML du modèle (à droite).



F . 3.15 – Exemple de modèle

TopCaseD (et EMF) offre la possibilité d'associer une syntaxe concrète particulière à un type de modèle. Ainsi, l'extension GMF (Graphical Modeling Framework) permet d'ajouter une syntaxe graphique.

L'outil Sintaks, [MFF06], permet l'ajout d'une syntaxe textuelle à un type de modèle. Ainsi, il est possible de définir des transformations vers d'autres méta-modèles. Par exemple, il est possible d'envisager une projection du modèle de livre vers des formats supportés par des outils d'édition (LaTeX, par exemple).

L'environnement de développement Eclipse comporte enfin un autre avantage, il est possible de le paramétrer avec de nombreux outils. Les environnements OpenEmbeDD et OpenArchitectureWare sont d'autres exemples classiques d'adaptation de la plateforme. Ainsi, ces environnements vont intégrer d'autres outils comme ATL [JK05], Xpand [EFH07] ou Sintaks [MFF06] utilisés aussi dans la mise en œuvre de la proposition (chapitre 6).

Dans le cadre des processus de développement, le méta-modèle SPEM a également été l'objet de divers projets et implantations dans des méta-outils avec en particulier l'outil EPF[5] (Eclipse Process Framework). Objecteering fournit une autre implantation du standard SPEM avec l'outil SPEM Modeler[6] permettant de décrire graphiquement les processus, de les organiser et les publier pour leur exploitation par les équipes en charge des projets.

Dans le même esprit, la plateforme Microsoft Solution Framework for Agile Software Development[7] propose un environnement pour des processus de développement, ceci en s'appuyant sur des concepts analogues à ceux de SPEM.

[5]http ://www.eclipse.org/epf/
[6]http ://www.objecteering.fr/free_addons_spem.php
[7]http ://msdn.microsoft.com/library/dd380647(VS.100).aspx

3.6 Conclusion

Cette partie montre que l’IDM offre un cadre pour l’opérationnalisation des modèles ou l’intégration de langages de modélisation. Cela dit, comme illustrée dans les sections précédentes, l’utilisation d’une démarche IDM dans un domaine particulier requiert la connaissance d’un certain nombre de technologies.

Malgré cet inconvénient, l’IDM fournit un cadre élégant pour modéliser et intégrer les différentes expertises impliquées dans un processus de développement.

En particulier, les langages utilisés par une activité ou un ensemble d’activités peuvent être exprimés à l’aide de (méta-)modèle, c’est à dire des modèles qui ne décrivent plus un système mais des concepts de modélisation. Ces méta-modèles peuvent être alors associés à des syntaxes concrètes graphiques et/ou textuelles.

Les méta-outils offert par l’IDM peuvent alors être :

1. Configurés avec ces méta-modèles.
2. Utilisés comme outils de modélisation pour des domaines spécifiques.

De même, des transformations peuvent être spécifiées pour tisser des liens entre différents méta-modèles, et passer ainsi d’un langage de modélisation à un autre.

Cependant, de nombreux travaux doivent encore être menés pour montrer comment capitaliser les compétences IDM pour améliorer les processus de développement, en permettant notamment de réduire les coûts (en facilitant par exemple le passage d’une activité à une autre) tout en permettant d’avoir des produits de qualité.

Contributions

Il n'y a pas de problèmes ; il n'y a que des solutions. L'esprit de l'homme invente ensuite le problème.
(André Gide)

Longue est la route par le précepte, courte et facile par l'exemple.
(Sénèque)

Chapitre 4

Vers une ingénierie de processus logiciels spécifiques dirigés par les modèles

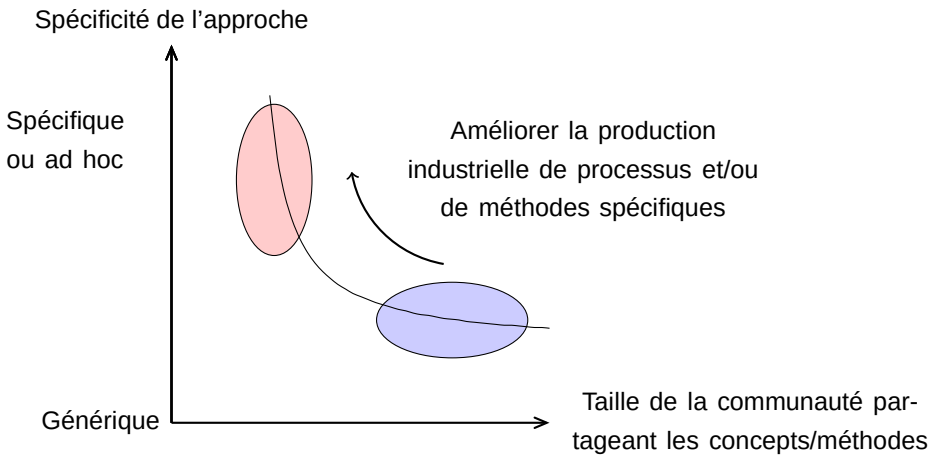
4.1 Introduction

L´ ´ , prédictifs ou agiles, présentés au chapitre 2, illustrent la difficulté de définir un processus de développement universel. En réalité, chaque projet est unique et peut être conditionné par l'usage de méthodes propres à une entreprise ou par des outils spécifiques à un domaine. Cependant, cette diversité n'empêche pas l'existence de normes génériques telles que ISO/IEC 12207 couvrant l'ensemble des processus mis en oeuvre par les différentes parties prenantes d'un produit logiciel. Ces normes décrivent les concepts partagés par une large communauté et peuvent servir de cadre de référence aux développements effectués par une entreprise. Ces cadres normatifs sont généralement ajustables selon les règles de fonctionnement propres à une entreprise, la nature ou les caractéristiques du logiciel à réaliser, les stratégies de développement adoptées, etc.

Face à la diversité de ces approches et à la nécessité de pouvoir ajuster les méthodes d'ingénierie en fonction de la complexité des produits logiciels à développer, le domaine de l'ingénierie des méthodes [Rol05] est une discipline qui tend à rationaliser la conception de méthodes d'ingénierie spécifiques (figure 4.1).

Comme pour tout processus de développement, l'objectif principal d'un processus spécifique est la maîtrise de la qualité du produit développé ainsi que la conformité aux exigences. Pour cela le processus doit fournir le moyen de rationaliser le développement tout en fournissant aux équipes :

- suffisamment de *réactivité* face aux changements éventuels des besoins, face aux erreurs ou encore face aux diverses contraintes de temps,
- suffisamment de *flexibilité* dans la démarche méthodologique en facilitant par exemple le passage d'un langage de modélisation à un autre.



F . 4.1 – Objectif de l'ingénierie des méthodes

Au delà de ces exigences générales, un processus spécifique doit être en meilleure adéquation avec l'équipe ou l'entreprise et la famille de produits logiciels qu'elle développe. Ce processus spécifique doit mieux intégrer les théories, les technologies, les langages de modélisation, les outils et plateformes, etc. d'un domaine spécifique. Il peut alors notamment comporter des activités spécifiques générant des produits spécifiques au domaine. Cette situation est classique pour les logiciels en forte interaction avec le monde physique comme les logiciels temps réel et/ou dédiés aux systèmes de contrôle-commande. Parmi les activités spécifiques à ce domaine, on peut citer : la modélisation et la simulation comportementale de systèmes, l'optimisation de modèles, la synthèse de contrôleurs ou superviseurs, la preuve formelle, l'analyse et la validation de l'ordonnancement temporel, l'émulation, la génération de code embarqué, l'usage de processeurs et algorithmes spécifiques, etc. Il s'agit d'un domaine d'ingénierie où le support théorique, ainsi que l'expertise et la réutilisation du savoir-faire technique sont essentiels à l'aboutissement des projets.

Plus précisément les travaux de la thèse se limitent à l'élaboration de processus de développement logiciels spécifiques dédiés aux systèmes à événements discrets (SED). Ce choix permet de placer naturellement l'approche proposée dans un cadre plus restreint dans lequel l'usage de modèles est courant et pour lequel il sera plus aisé de montrer les difficultés, ainsi que les avantages et limites de la proposition. Malgré cette restriction au domaine des SED, l'approche proposée permet d'illustrer l'introduction d'activités spécifiques comme la synthèse de superviseurs ou la vérification de modèles dans un processus de développement, l'utilisation de langages de modélisation spécifiques pour certaines activités et l'intégration d'outils et plateformes d'exécution spécifiques.

L'objectif de l'ingénierie dirigée par les modèles (IDM), vue au chapitre 3, est de placer les modèles au coeur du développement logiciel. Cette approche par la modélisation systématique, est une réponse à la montée en complexité des systèmes logiciels qui vise également à améliorer la productivité des équipes grâce à l'aspect génératif apporté par les transformations de modèles. Ce chapitre cherche à combiner les acquis de l'IDM et de la modélisation des processus logiciels en vue de l'élaboration de modèles de processus spécifiques et centrés modèles, offrant une meilleure intégration des langages de modélisation,

de l'outillage et des plateformes d'exécution. Ces processus devront donner une part plus large aux activités de manipulation des modèles, et notamment aux transformations de modèles servant de liant entre les différentes phases de réalisation du logiciel : spécification, conception, vérification, implantation, etc.

Le paragraphe suivant précise les principes généraux sur lesquels s'appuie la proposition et notamment l'existence d'un double cycle de vie. Ensuite, le paragraphe 4.3 précise quels sont les constituants essentiels d'un processus spécifique dirigé par les modèles et comment modéliser ce processus. Puis, dans le paragraphe 4.4, une démarche méthodologique, pouvant être qualifiée de méta-processus et servant de guide à la conception de processus spécifiques, est proposée.

Ce chapitre sera complété, au chapitre 5, par la mise en oeuvre des concepts proposés pour l'élaboration de processus spécifiques dédiés aux SED et, au chapitre 6, par la mise en oeuvre de ces processus pour traiter des exemples d'applications intégrant des langages, des outils et des plateformes d'exécution spécifiques.

4.2 Principes généraux

4.2.1 Contexte d'un développement logiciel

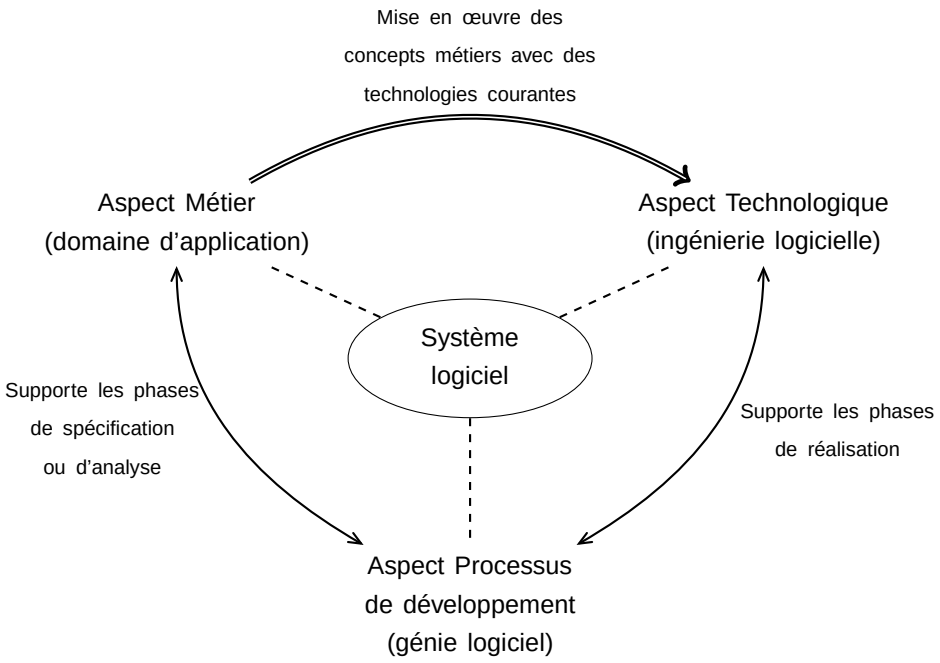
Dans un premier temps il est important de mieux situer le contexte général de tout développement logiciel. Un système logiciel est le résultat d'un processus d'ingénierie mettant en oeuvre les technologies courantes pour satisfaire les exigences spécifiques d'un client. Trois aspects essentiels (figure 4.2), correspondant globalement à trois métiers différents, sont à considérer :

- le *métier*, qui requiert des compétences liées au domaine d'application du logiciel,
- l'*ingénierie logicielle*, qui requiert des compétences en conception de logiciels,
- le *gestion du processus de développement*, qui requiert des compétences en génie logiciel.

Le *métier* ou *domaine d'application* impose la prise en compte de connaissances métiers ou de connaissances théoriques relevant d'un domaine d'application particulier. Il détermine des classes d'applications et peut avoir un impact important sur l'aspect spécificité d'un processus de développement (intégration d'activités ou d'outillage spécifiques, conception ou utilisation d'un framework métier, etc.).

L'*ingénierie logicielle* regroupe l'ensemble des techniques, technologies, solutions logicielles ou encore méthodes de développement à utiliser pour concevoir le logiciel. Elle a un impact sur la spécificité du processus de développement (choix d'une approche orientée objets, ou à base de composants ou d'agents, conformité à un style d'architecture, intégration de frameworks techniques, etc.).

La *gestion du processus* impose un cycle de développement ou une démarche méthodologique qui doivent être suivis par l'équipe de développement.



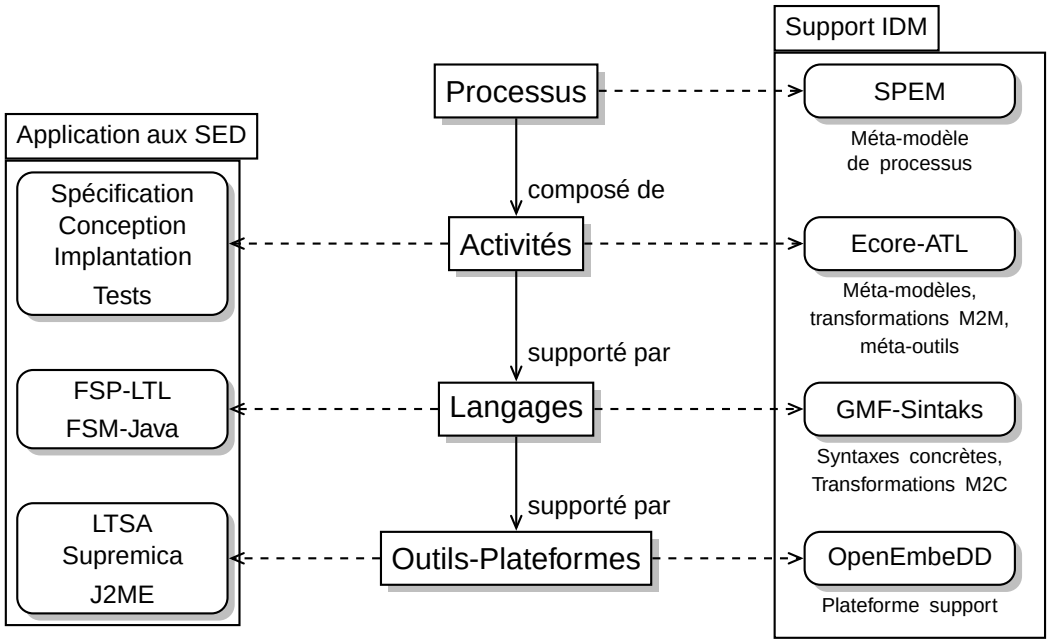
F . 4.2 – Contexte général d'un développement logiciel

Ces trois aspects sont en interaction et visent à répondre aux questions suivantes concernant le système en développement : *pourquoi* et *quoi* pour l'aspect métier, *avec quoi* pour l'aspect ingénierie logicielle et enfin *comment faire* pour l'aspect processus. Un processus spécifique à une classe d'applications relevant d'un domaine métier particulier devra alors intégrer ces différents aspects. Le travail effectué dans la thèse se focalise plus spécialement sur la façon d'intégrer les différents langages de modélisation et les modèles produits tout au long du cycle de vie défini par le processus de développement. Pour progresser dans cette direction l'approche devra mieux comprendre le contexte dans lequel sont produits ou utilisés les différents modèles, et mettre en relation ces modèles notamment par une intégration des outils qui les supportent.

4.2.2 Couplage entre processus et modèles

Afin de pouvoir tirer profit d'une ingénierie dirigée par les modèles et permettre une approche plus outillée du développement il est nécessaire de coupler de manière plus intime le processus de développement et les modèles produits ainsi que les langages de modélisation nécessaires. En partant d'une décomposition classique d'un processus en activités, produits et/ou ressources et en considérant que les modèles sont des produits, une première approche consiste à classifier les différentes entités intervenant dans un développement selon quatre catégories (figure 4.3). Les processus se décomposent en activités. Les activités représentent les différentes tâches à réaliser pour assurer un développement cohérent du logiciel. Ces activités sont supportées par des langages (de modélisation) courants ou spécifiques et fournissant à l'équipe les concepts de modélisation nécessaires au développement. Les différents langages sont intégrés à l'aide d'activités de transformation de

modèles. Enfin, les langages sont supportés par des outils, plateformes cibles, frameworks, qui vont donner les moyens techniques et technologiques pour réaliser le logiciel.



F . 4.3 – Schéma conceptuel de l'approche proposée

La partie gauche de la figure affine cette décomposition dans le cas d'étude des systèmes à événements discrets. Les activités de spécification ou de conception peuvent exploiter des langages spécifiques (cf. chapitres 5 et 6) comme des algèbres de processus (FSP), des logiques temporelles (LTL) ou simplement des automates (FSM). Ces langages sont outillés à l'aide d'outils spécifiques (LTSA, Supremica) ou concrétisés sous la forme de frameworks. Des traductions entre modèles abstraits et syntaxes concrètes sont généralement nécessaires pour pouvoir utiliser l'outillage disponible.

Afin de coupler les activités, les langages et les outils il est possible de s'appuyer sur le domaine de l'Ingénierie Dirigée par les Modèles avec ses propres concepts, ses contributions et ses outils (partie droite de la figure). Par exemple le méta-modèle SPEM peut servir à formaliser de manière précise le processus de développement. Les langages utilisés par le processus peuvent être modélisés en donnant leur méta-modèle en MOF, Ecore, etc. Ces méta-modèles sont couplés à l'aide de transformations définies par exemple avec ATL. Des transformations de modèles vers des syntaxes concrètes pour permettre l'import ou l'export de modèles par les outils disponibles peuvent être nécessaires. Ces transformations sont réalisées à l'aide d'outils spécifiques tels que Xpand, Sintaks. Afin de rationaliser l'ensemble, des plateformes support et orientées IDM, telles que OpenEmbeDD ou OpenArchitectureWare, peuvent être utilisées. Elles proposent alors au sein d'un même environnement les différents aspects du développement avec en particulier sa gestion ou sa réalisation.

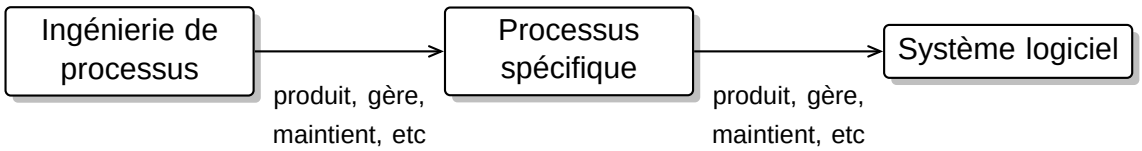
4.2.3 Comment modéliser ce couplage ?

Un processus de développement logiciel est un système en soi dont la nature et la complexité s'apparentent à celles d'un système logiciel [Ost87, Est06]. La conception d'un processus, sa modélisation, son exploitation ou sa maintenance peuvent donc bénéficier de cette proche parenté. Les notations habituelles propres au développement logiciel peuvent donc servir à décrire la structure ou le comportement du processus (diagrammes de classes, d'activités, d'interaction, etc.). Cependant, si l'état des pratiques courant pour la modélisation du logiciel est d'exploiter plutôt un paradigme objet, les processus sont plutôt abordés selon une modélisation orienté activités. Le domaine des processus bénéficie alors de son propre vocabulaire et de ses propres concepts : *activités, rôles, produits, ressources, etc*. Ces concepts peuvent être organisés et formalisés au sein d'un méta-modèle tel que SPEM [OMG02, OMG08a]. Il faut noter que SPEM 2.0 est à la fois un méta-modèle et un profil UML ce qui permet d'exploiter des concepts du méta-modèle d'UML comme par exemple les activités.

Malgré l'existence de méta-modèles dédiés à l'ingénierie de processus [Hug06], il n'existe pas, à notre connaissance, et à ce jour, de méta-modèles ou de langages de modélisation dédiés à l'ingénierie de processus dirigés par les modèles, c'est à dire intégrant les concepts propres à l'IDM tels que : *modèles, méta-modèles, langages de modélisation, transformation de modèles, etc*.

Face à cette problématique, plutôt que d'adopter un méta-modèle comme SPEM 2.0, riche mais ne convenant pas parfaitement, ou de définir un méta-modèle plus adéquat, ce qui n'est pas l'objectif de cette thèse, une approche ascendante sera utilisée. Cette approche cherchera à coupler les modèles du produit réalisé et le processus qui le réalise en cherchant les points d'ancrage entre ces deux domaines.

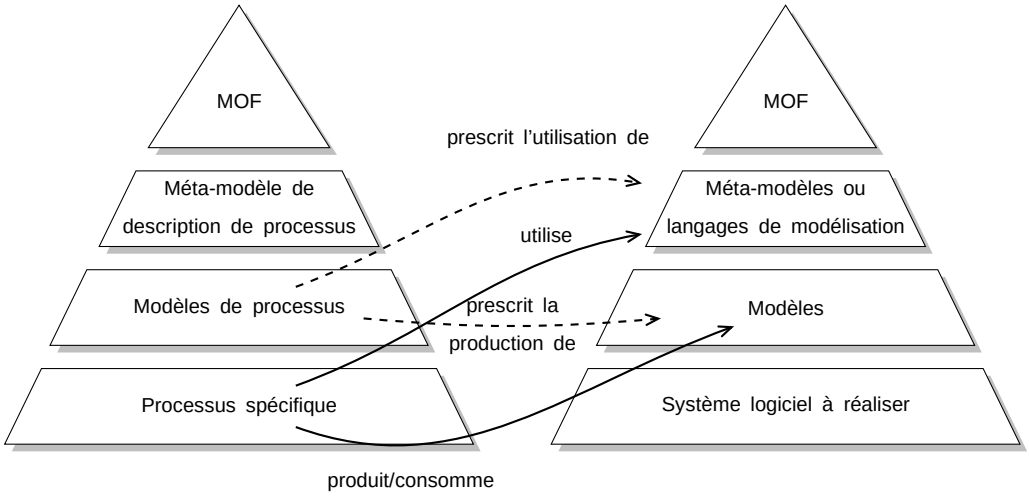
Une première étape consiste à clarifier le couplage entre le produit réalisé, le processus spécifique qui le réalise et la conception d'un processus spécifique. Une vue simplifiée de chacune de ces entités est donnée à la figure 4.4.



F . 4.4 – Vue simplifiée : Ingénierie de processus, processus, système

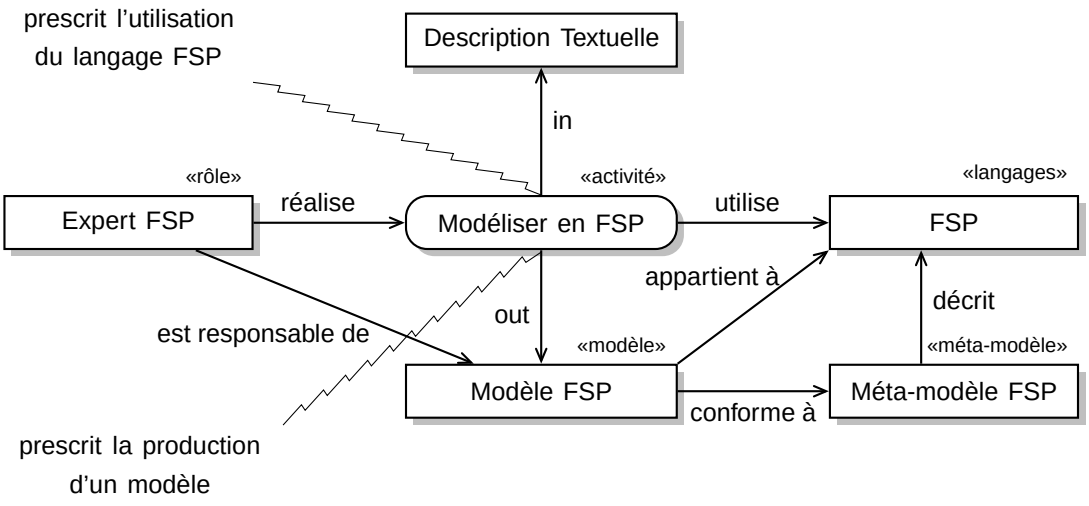
Un processus spécifique produit, gère ou encore maintient un système logiciel. Le processus spécifique est lui-même conçu, produit ou maintenu par un processus qui pourrait être qualifié de "méta-processus", mais que nous préférons qualifier d'*ingénierie de processus spécifique*.

Comme le processus spécifique à construire doit permettre un développement logiciel dirigé par les modèles, la figure 4.4 peut-être affinée en l'injectant dans la pyramide de méta-modélisation de l'OMG. Même si la pyramide de méta-modélisation à 4 niveaux est unique, il peut cependant être intéressant de la séparer en deux pyramides pour mieux faire



F . 4.5 – Schéma conceptuel du couplage entre processus et produit

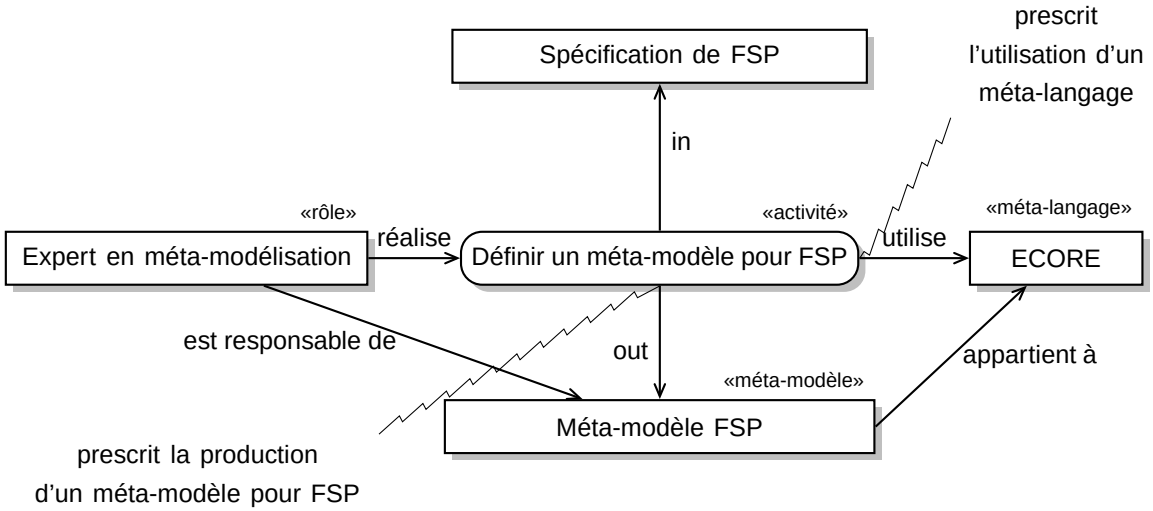
apparaître le couplage entre le processus et le produit réalisé ainsi que la localisation des modèles et méta-modèles de ces deux entités. La figure 4.5 précise ce couplage. Le processus (en exécution) utilise des langages de modélisation (spécifiques ou non) ou leurs méta-modèles pour produire, consommer, transformer des modèles du système à réaliser. Le modèle du processus quant à lui prescrit l'utilisation de ces langages et prescrit la production, la consommation de modèles, etc. La figure 4.6 illustre comment cette prescription pourrait être modélisée dans un modèle de processus spécifique.



F . 4.6 – Extrait d'un modèle de processus spécifique illustrant le couplage entre les concepts d'activité, de rôle, de modèle, de langage et de méta-modèle

La même approche de modélisation pourrait être utilisée pour modéliser l'ingénierie d'un processus spécifique, mais ceci conduirait à une certaine lourdeur que nous souhaitons éviter pour l'instant face à la complexité de la problématique. A titre d'exemple la figure 4.7 illustre l'activité de production d'un méta-modèle spécifique qui peut être considéré comme une ressource qui sera utilisée par le processus spécifique. Ainsi, ce qui a un statut

de produit à élaborer dans le processus d'ingénierie d'un nouveau processus, devient une ressource quand il est exploité dans le processus spécifique.



F . 4.7 – Modélisation d'une activité de production de méta-modèle, servant de ressource dans un processus spécifique

4.2.4 Double cycle de vie

La discussion précédente rend manifeste l'existence d'un double cycle de vie. Le premier cycle de vie a pour objet de définir, mettre en place, outiller, exploiter, maintenir et/ou faire évoluer un produit qui est un processus de développement logiciel spécifique. Ce processus spécifique est constitué de plateformes, d'outils, de logiciels, de méta-modèles ou de langages de modélisation, et de modèles ou modes opératoires correspondant au processus spécifique. Le deuxième cycle de vie est le cycle plus habituel (en V, en cascade, en spirale, agile, etc.) correspondant au développement d'un système logiciel. Ces deux cycles de vie sont en interaction et, comme il a été décrit précédemment les produits de l'un peuvent servir de ressources ou d'outils à l'autre. Si la modélisation d'un processus de développement logiciel est bien connu il n'en va pas de même concernant la modélisation d'un processus ayant pour finalité la production d'un processus de développement logiciel dirigé par les modèles. Ce processus étant lui-même un assemblage complexe d'outils, de personnes, de logiciels et/ou de modèles, une ingénierie de type système est probablement plus pertinente pour aborder cette problématique. Les étapes principales en seraient alors : la définition, la réalisation des constituants, l'intégration, le transfert en exploitation, l'exploitation et le maintien en condition opérationnelle, l'abandon ou le retrait [Roc07].

L'existence de ce double cycle de vie conduit à deux périodes ou phases bien distinctes. Grossièrement, une première période consiste à élaborer un processus de développement spécifique, et une deuxième période consiste à exploiter ce processus spécifique et son outillage pour réaliser des applications particulières. Une telle démarche ne peut-être envisagée que si la première période conduit réellement à un ensemble de processus spécifiques

pouvant être sélectionnés ou configurés pour la réalisation de toute une famille d'applications apparentées et relevant d'un même domaine. Le coût supplémentaire engendré par l'élaboration de processus spécifiques orientés modèles, doit être compensé en allégeant les développements futurs. En effet, l'effort de définition de méta-modèles et de transformations de modèles lors de la première phase doit être compensé par l'utilisation et la réutilisation de ces transformations pour passer plus facilement, et de façon automatique, d'une activité à une autre.

Pour cela, la première phase a pour objectif de définir le contexte et la classe d'applications dans laquelle vont s'inscrire les futurs développements. Selon ce contexte et la classe d'applications envisagée, un recensement des domaines théoriques relevant du domaine métier est effectué. Ce domaine métier va également induire des choix technologiques (styles d'architectures, plateformes support, frameworks, etc.) à exploiter ou à concevoir pour fournir aux concepteurs les moyens d'effectuer le développement. Enfin, cette première phase doit définir ou préconiser un processus de développement adéquat pour ce type d'applications. La seconde phase consiste, selon une ou plusieurs problématiques et dans le contexte défini lors de la première phase, à mettre en application le processus et son outillage tels que définis lors de la première phase.

4.3 Processus spécifiques dirigés par les modèles

Un *processus logiciel* est généralement défini comme un ensemble d'activités techniques et de gestion, exécutées de manière coordonnée pour développer et maintenir un produit logiciel. Dans une vision idéalisée, un *processus logiciel dirigé par les modèles* peut être vu comme un ensemble coordonné de transformations de modèles, prenant des modèles en entrée et produisant des modèles en sorties, jusqu'à l'obtention de code exécutable. De manière plus réaliste, un processus dirigé par les modèles doit combiner et importer ces deux visions du développement.

Ainsi, de manière non exhaustive, un processus de développement logiciel, spécifique et dirigé par les modèles :

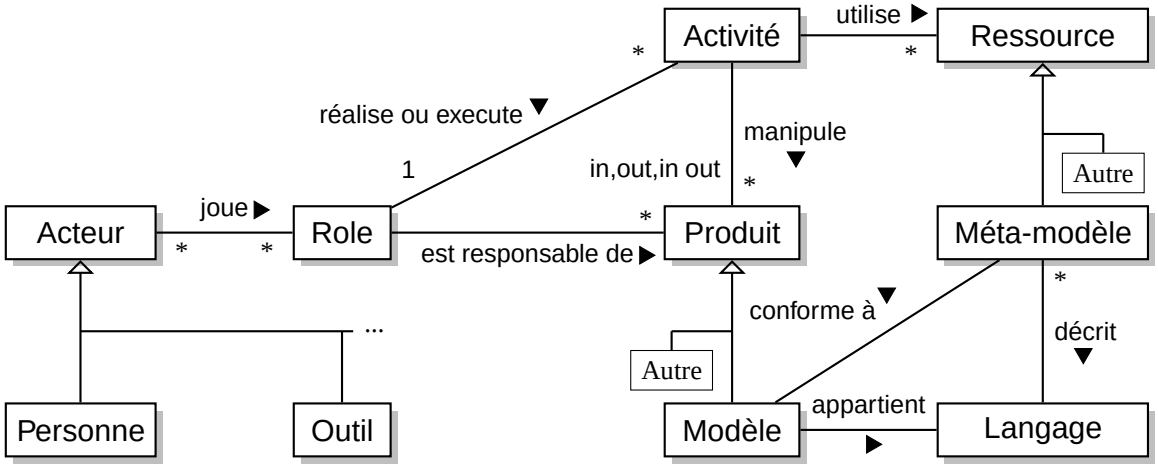
- utilise des langages de modélisation génériques (ex : UML) et spécifiques (ex : LTL),
- produit et consomme des modèles définis avec ces langages,
- exploite des outils d'usage général (ex : Eclipse) ou spécifique (ex : LTSA),
- exploite des plateformes, des frameworks et/ou des composants spécifiques ou d'usage général,
- s'appuie sur des transformations de modèles réalisées par les outils,
- est composé d'activités classiques ou spécifiques qui doivent être coordonnées,
- génère et consomme des produits dont certains sont des modèles,
- confie des rôles, classiques ou spécifiques aux acteurs.

Ces différents points vont être détaillés dans la suite de ce paragraphe.

4.3.1 Modéliser le processus

Afin de mieux formaliser les différents points précédents, il est nécessaire de passer à une représentation plus formalisée du processus. Ceci impose l'utilisation d'un *langage de description de processus* intégrant les concepts de l'IDM induisant aussi des problèmes de syntaxe abstraite (méta-modèle), de syntaxe concrète (textuelle ou graphique) exploitable facilement et de sémantique de ce langage. Ce point a été abordé au paragraphe 4.2.3 en précisant qu'au stade actuel de ce travail, nous préférons nous appuyer sur un schéma conceptuel qui sera une variante du schéma conceptuel de SPEM plutôt que d'utiliser la définition du méta-modèle SPEM et ses stéréotypes UML pour sa syntaxe concrète, ce qui nuirait à ce travail de clarification sans en garantir plus de pérennité. En effet, aux dires mêmes de l'OMG, SPEM 2.0 a été défini comme successeur de SPEM 1.0 vue la faible adoption de ce standard, notamment à cause de son ambiguïté sémantique [OMG08a]. Ce point sera rediscuté en conclusion de chapitre.

La figure 4.8 donne un schéma conceptuel des concepts qui serviront à étayer la suite de la discussion. Ce schéma combine les schémas des figures 3.7 et 3.13.



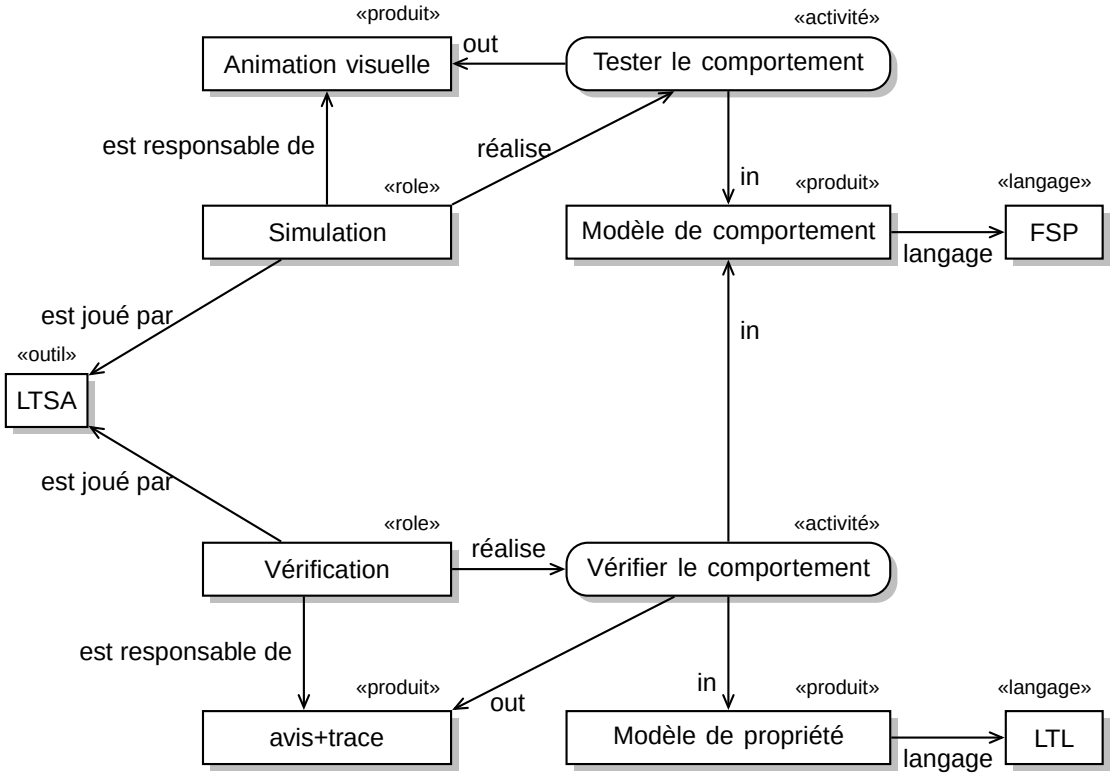
F . 4.8 – Schéma conceptuel des concepts nécessaires à la description structurelle d'un processus spécifique dirigé par les modèles

A partir de ce schéma conceptuel, il est possible de dire qu'une activité nécessite un rôle pour sa réalisation ou encore qu'un acteur joue des rôles. Une seule relation est représentée entre les activités et les produits. Cette relation peut être éclatée en trois relations selon que le produit est consommé (*mode in*), généré (*mode out*) ou transformé (*mode in out*). Certaines relations sont omises, par exemple les méta-modèles sont considérés comme des ressources, mais ce sont aussi des modèles (ils devraient en hériter), de même certains produits peuvent devenir des ressources pour des activités ultérieures d'un processus, etc. La multiplicité 1 qui dit qu'une activité est réalisée par un rôle unique est un choix dérivé du schéma conceptuel de SPEM. Enfin le concept de Processus qui peut être défini comme un ensemble d'activités, de rôles et de produits n'est pas représenté sur la figure 4.8. Ces quelques remarques illustrent la difficulté de produire un méta-modèle précis et largement adopté par une communauté.

Les définitions suivantes précisent les concepts :

- Un *processus* de développement logiciel est une collaboration entre des *rôles* qui sont des entités abstraites exécutant des *activités* sur les produits qui sont des entités concrètes.
- Un *rôle* est un concept abstrait qui décrit un ensemble de responsabilités et de compétences pour réaliser des activités manipulant des entités concrètes ou produits.
- Comme au théâtre, les rôles sont distribués à des *acteurs*. Un rôle est par conséquent un comportement spécifique d'un acteur se déroulant dans un contexte particulier d'un processus. La séparation entre les concepts de rôle (au niveau logique) et d'acteur (au niveau physique) permet de redistribuer les rôles si nécessaire.
- Une *activité* est un ensemble de travaux, de tâches, d'étapes exécutées physiquement par des humains, des machines, etc.
- Un *acteur* est une personne physique, une entité organisationnelle, une machine, etc. qui prend part aux activités d'un processus en jouant certains rôles.
- Contrairement à un produit, servant d'entrée à une activité et qui peut être consommée ou transformée, une *ressource* reste typiquement disponible après l'activité et sert de support à l'activité.

Afin de clarifier ces concepts, la figure 4.9 donne un exemple d'utilisation de ce schéma conceptuel.



F . 4.9 – Exemple d'utilisation du schéma conceptuel proposé

4.3.2 Activités, modèles et langages

Au paragraphe 4.2 décrivant les principes généraux, la figure 4.3 propose un cadre général d'analyse avec un empilement en quatre niveaux. A savoir, que les processus se décomposent en activités qui sont supportées par des langages (de modélisation) qui sont eux-mêmes supportés par des outils. Dans ce paragraphe, la partie haute de cet empilement est approfondie, à savoir : *quelles sont les relations entre activités, modèles et langages ?* Les paragraphes suivants détailleront les relations entre les langages et les outils ou plateformes qui les opérationnalisent.

Dans un *processus dirigé par modèles*, les modèles sont mis au coeur du processus de développement en les associant de manière plus intime aux différentes activités du processus. Ces activités peuvent être classiques ou spécifiques à un domaine d'application et sont supportées par des langages eux-mêmes classiques ou spécifiques. Ces langages sont définis par leurs méta-modèles qui peuvent préexister dans le cas des langages classiques (ex : UML) et qui doivent être implantés pour des langages plus spécifiques.

Pour un *processus spécifique* dédié à un domaine d'application, différents langages de modélisation (et leurs méta-modèles), établis après l'identification des concepts clefs du domaine, peuvent cohabiter. Les modèles manipulés par les activités se déclinent typiquement d'un niveau abstrait (ex : une spécification du problème) jusqu'à un niveau plus concret (ex : la solution en terme de code cible). L'intégration entre ces différents niveaux d'abstraction va s'appuyer sur des activités de transformations de modèles. En s'appuyant sur ces différents langages et activités de transformation, et confronté à un problème relevant du domaine, des modèles pourront être construits et intégrés de manière plus rationnelle.

Généralement, les processus de développement sont des processus, potentiellement itératifs, dans lesquels reviennent certaines activités récurrentes à tout type de développement et permettant de passer d'un cahier des charges abstrait à une réalisation concrète. Les grandes étapes classiques sont typiquement :

1. La *spécification* définissant les exigences du produit à réaliser.
2. La *conception* établissant les modèles des entités logicielles à réaliser, obtenus en fonction des exigences décrites par la spécification.
3. L'*implantation* constituant le codage des modèles pour une plateforme dédiée et fournissant le produit logiciel voulu.
4. Les *tests* permettant de valider la réalisation ou de solliciter la réalisation d'itérations sur les différentes phases du développement.

Ces différentes étapes peuvent s'intégrer dans une structure descendante similaire à une approche en cascade (figure 2.9 décrite dans l'état de l'art).

Le langage UML est largement utilisé lors des étapes de spécification ou de conception. Employé comme langage générique de modélisation, il offre un large choix de diagrammes pour exprimer l'ensemble des aspects des logiciels à réaliser. Ainsi, grâce à sa souplesse et à ses extensions, il permet de définir les exigences, l'architecture, les structures et les comportements des logiciels.

Cette division, à un niveau d'abstraction élevé, en quatre étapes génériques doit être complétée ou affinée dès qu'elle est confrontée à des domaines d'applications spécifiques apportant leurs propres théories et outillages qui doivent être intégrés dans le processus de développement. En réalité, la structuration du processus de développement et le choix des activités à réaliser sont intimement liés au type et au contexte des problèmes à résoudre. Ainsi, pour le domaine des SED, des théories, activités, formalismes ou langages et outils spécifiques sont à considérer. En effet, même si UML peut être d'une grande utilité au cours de certaines parties du développement, des diagrammes tels que les machines à états-transitions d'UML peuvent ne pas convenir et il faudra se replier sur des langages mieux formalisés (automates, systèmes de transitions étiquetés, algèbres de processus, etc.). Ce choix s'impose par la nécessité de vérifier certains modèles de manière algorithmique par des techniques de Model-Checking ou encore par la possibilité d'obtenir certains modèles par synthèse (cf. Annexes) en exploitant le domaine théorique de la commande supervision. Ces activités sont alors supportées par des langages de modélisation formels (ex : FSP, LTL, etc.) ayant une sémantique précise.

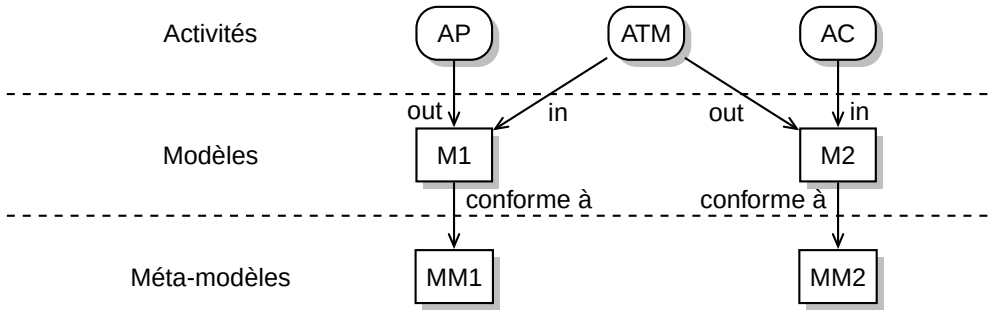
Un système logiciel est en général hébergé par un système d'exploitation qui peut être classique ou spécifique et n'est pas nécessairement le même système d'exploitation que celui servant au développement. Les systèmes d'exploitation hébergent des environnements de programmation offrant notamment des langages de programmation classiques. Dans une démarche de type IDM du code peut-être généré directement pour ces langages cibles. Cette approche a cependant l'inconvénient de généralement perdre les abstractions présentes dans les modèles ce qui pose un problème de traçabilité. A titre d'exemple une application de traitement de signal, modélisée avec un paradigme (une architecture) de type tubes et filtres peut voir ces abstractions dispersées dans le code lors de la phase de génération de code cible. Une deuxième approche consiste à construire des frameworks métiers ou techniques contenant à la fois les abstractions d'un domaine et la logique d'exécution spécifique à ce domaine. Cette approche peut faciliter la génération de code tout en préservant les abstractions du domaine. Dans cette optique, un framework métier peut alors être considéré comme l'implantation d'un langage de domaine, et son modèle comme un méta-modèle pour ce langage. Ces approches seront illustrées par les applications du chapitre 6.

4.3.3 Transformations de modèles

Au delà de l'avantage premier d'une approche orientée modèles qui est d'aborder la complexité des systèmes par la modélisation, un des intérêts majeurs de l'Ingénierie Dirigée par les Modèles est de permettre des transformations de modèles, soit au sein d'un même langage de modélisation soit entre des langages de modélisation différents.

Au sein d'un processus de développement une transformation de modèle peut être assimilée à une activité. La figure 4.10 illustre le cas où une activité AP produit un modèle M1, ce modèle est transformé par une activité de transformation de modèle ATM en un modèle M2 consommé par une activité AC. Certaines de ces activités de transformation peuvent être réalisées de façon manuelle, comme par exemple la traduction d'un diagramme de communication ou de séquence en code source. Cependant le fait de modéliser ces acti-

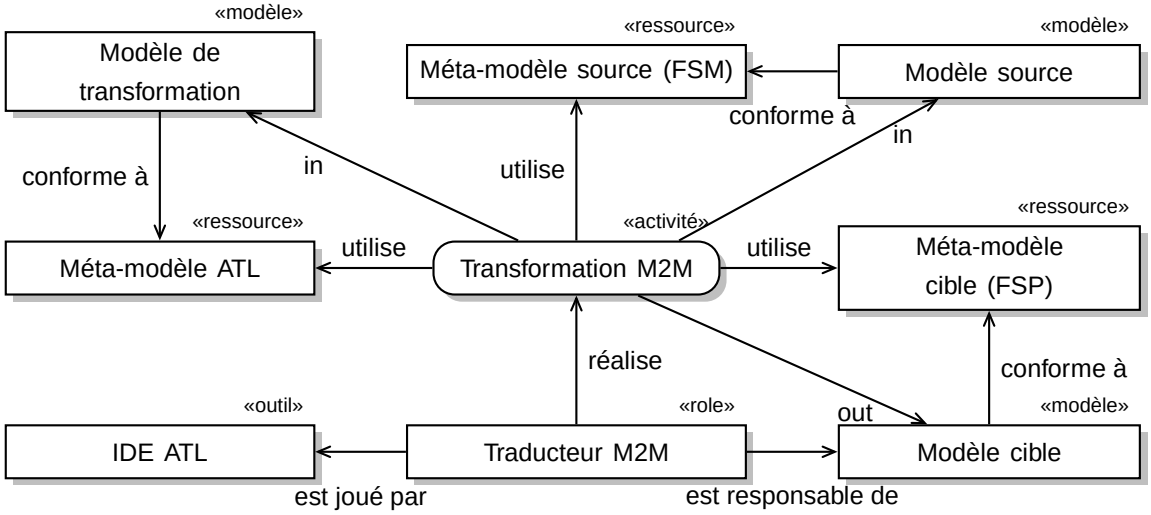
vités comme des activités de transformations de modèles permet à plus ou moins longue échéance d'envisager leur automatisation (ou semi-automatisation) par une traduction algorithmique.



F . 4.10 – Une transformation de modèle est une activité

Ainsi, une transformation de modèle est une activité qui prend un ou plusieurs modèles, conformes à leur méta-modèles, en entrée, qui produit un ou plusieurs modèles, conformes à leurs méta-modèles, en sortie, et ceci en utilisant un modèle de transformation, lui-même conforme à un méta-modèle de langage de transformation. Ce concept de transformation vu comme une activité (une sous-classe) n'est pas explicitement représenté sur le schéma conceptuel de la figure 4.8 mais peut être aisément ajouté à partir de la description précédente.

L'intérêt de placer cette transformation au sein d'un modèle du processus est de mieux visualiser les artefacts nécessaires ou produits par la transformation ainsi que leur dépendances par rapport à d'autres entités du processus. Ce sera l'objectif de l'ingénierie d'un processus spécifique de fournir tout ou une partie de ces artefacts. Afin de clarifier ce point, la figure 4.11 illustre une transformation de type M2M telle que, par exemple, une transformation d'automates (FSM) vers une algèbre de processus (FSP) réalisée avec l'outillage ATL. Cette transformation sera décrite plus en détail au chapitre 5, figure 5.8.



F . 4.11 – Exemple de transformation de modèle vue comme une activité

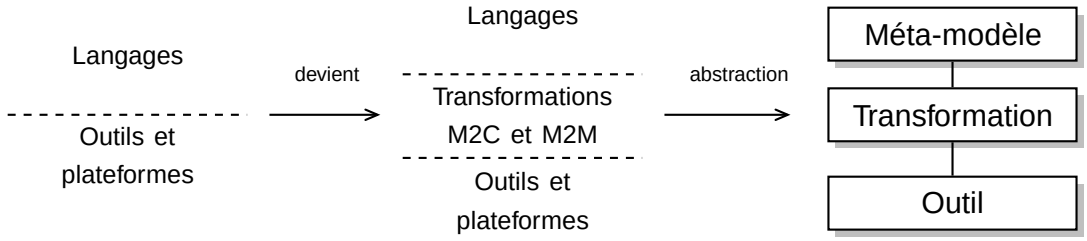
L'analyse de cette figure révèle l'existence de certains produits tels que le méta-modèle source (FSM) et le méta-modèle cible (FSP), le modèle de transformation (FSM2FSP) qui sont à réaliser par un expert en méta-modélisation, lors de l'ingénierie du processus. De même certaines entités (algorithme de transformation, outil, IDE, etc.) peuvent être mises à disposition par la communauté IDM ou par des outilleurs de l'IDM. Ainsi une transformation de modèle peut être vue selon trois points de vue différents : 1) c'est une activité d'un processus spécifique, 2) elle est représentée par un modèle de transformation qui est produit par le processus d'ingénierie du processus spécifique, 3) c'est un algorithme générique, dans un outil générique, paramétré par un modèle de transformation et mis à disposition par un outilleur.

De manière très générale il existe trois types (sous-classes) de transformations :

1. *Le premier type* de transformation se situe entre les grandes phases du processus de développement. Ces transformations vont permettre le passage d'une phase à une autre et ainsi permettre le raffinement des modèles au fil de l'avancement du processus de développement. Par exemple, à partir de modèles de diagrammes à états, conformes à la notation UML et définis lors d'une phase d'analyse, il est possible d'obtenir par transformation des automates à états finis utilisables lors de la phase de conception.
2. *Le second type* de transformation va permettre au sein d'une même phase de construire et combiner des modèles issus de langages de modélisation différents. Ces transformations de modèles vont ici permettre de bénéficier des différents langages de modélisation nécessaires à la réalisation des activités. Un exemple est la combinaison d'un modèle de comportement (ex : FSP) avec un modèle de propriété (ex : LTL) en vue d'une vérification de comportement et production d'un modèle de trace.
3. *Le troisième type* de transformation va fournir les passerelles vers les outils ou des plateformes spécifiques offrant leur propres langages. De manière très générale il s'agit de fonctions d'import/export vers l'existant. Ce type de transformation peut par exemple lors d'une activité d'implantation fournir le moyen de réaliser une génération de code vers un framework spécifique ou une plateforme susceptible d'héberger le logiciel. Ces transformations sont détaillées dans le paragraphe suivant.

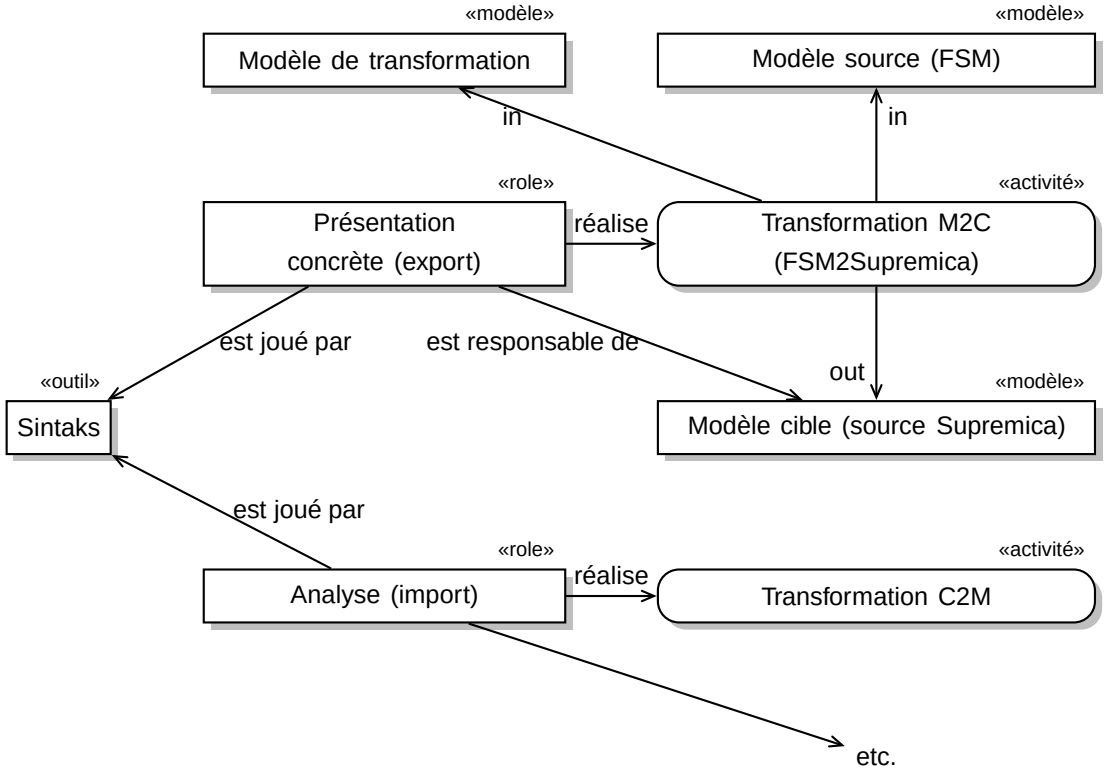
4.3.4 Coupler à l'existant, transformations syntaxiques

La figure 4.3 du paragraphe 4.2 proposait comme cadre général d'analyse un empilement en quatre niveaux. A savoir, qu'un processus spécifique se décompose en activités utilisant des langages classiques ou spécifiques qui doivent être supportés par des outils. Dans ce paragraphe, la partie basse de cet empilement est approfondie, à savoir : *quelles sont les relations entre les langages et les outils ou plateformes ?*



F . 4.12 – Intégration des transformations de modèles depuis/vers les syntaxes concrètes des outils

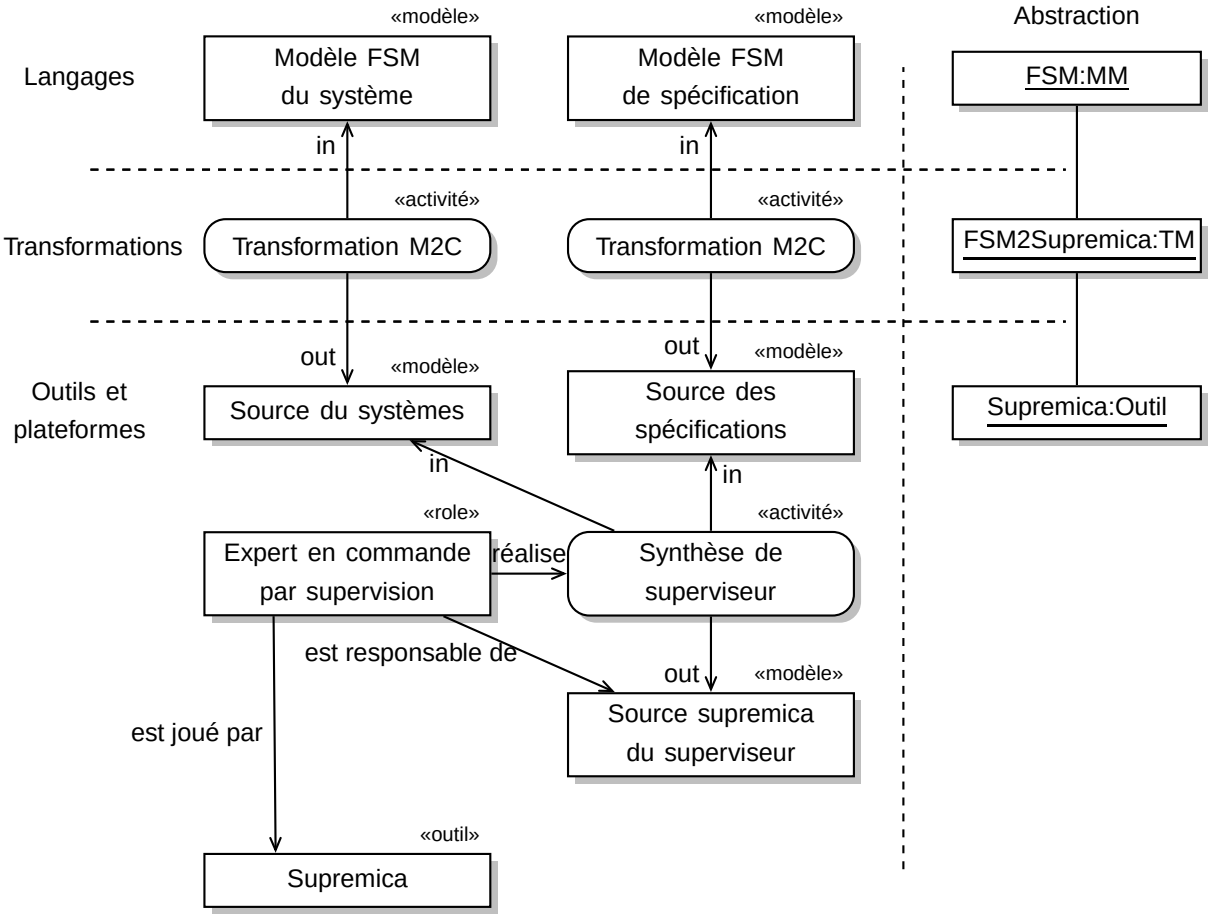
Bien évidemment un outil existant, ou construit spécifiquement, peut supporter directement un langage de modélisation prévu au sein du processus. Cependant, l'utilisation d'outils existants ou plateformes spécifiques peut nécessiter des transformations de modèles abstraits vers la syntaxe concrète de ces outils (transformations M2C) ou inversement (transformations C2M). Ainsi à l'interface entre les langages et les outils ou plateformes vivent des transformations vers des (ou de) syntaxes concrètes de ces outils (figure 4.12). La partie droite de la figure décrit comment cet empilement sera abstrait au chapitre 6 (figure 6.3 et 6.19).



F . 4.13 – Transformations syntaxiques vers des outils ou plateformes

Au sein du processus de développement ces transformations nécessitent des rôles de présentation (ou export) de modèles vers une syntaxe concrète et d'analyse (ou import) de modèles à partir d'une syntaxe concrète. Ces rôles peuvent être joués par des outils établissant des ponts entre des syntaxes concrètes et des syntaxes abstraites. C'est le cas de l'outil Sintaks que nous avons utilisé [MFF06]. La figure 4.13 modélise l'utilisation de cet outil au sein d'un procédé de développement. Cette figure affine la couche transformation M2C et C2M de la figure 4.12. Des exemples de modèles de transformation seront donnés au chapitre 5.

Sintaks est un outil générique, pour l'IDM, mais dans la couche appelée outil et plateformes vivent plutôt des outils spécifiques liés au domaine d'application que couvre le processus. Cette couche doit donc également être affinée afin de cerner les manipulations de modèles effectués par ces outils. A titre d'illustration le problème de la synthèse de superviseur à l'aide de l'outil Supremica est traité. L'objectif d'une synthèse de superviseur est de synthétiser un modèle de superviseur à partir du modèle d'un système non supervisé et d'une spécification du système supervisé. La combinaison du système et du superviseur doit satisfaire la spécification (voir annexe A et chapitre 6). La figure 4.14 décrit l'utilisation de cette synthèse au sein d'un processus spécifique. A noter que la couche transformation M2C n'est pas détaillée, en fait son contenu détaillé est donné par la figure 4.13.

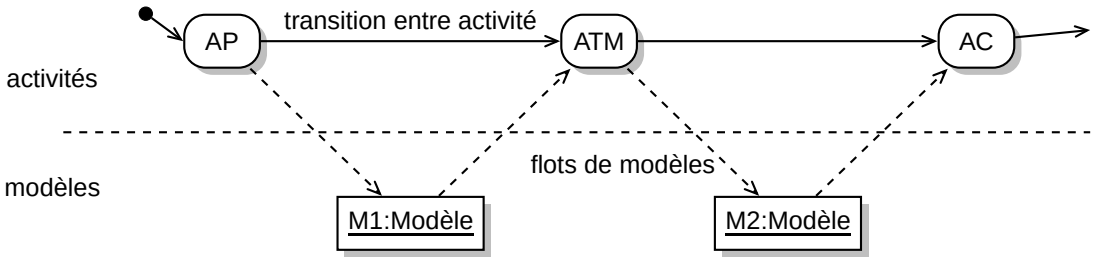


F . 4.14 – Détail de l'utilisation d'un outil spécifique

Le même genre d'approche peut être utilisé pour intégrer les frameworks ou plateformes d'exécution, mise à part que la principale activité de ces entités est de supporter l'exécution pendant la phase d'exploitation du logiciel et non pendant son développement. Pour résumer : les outils supportent le développement en hébergeant des manipulations de modèles spécifiques et il faut prévoir des transformations permettant l'import/export vers ces outils alors que les frameworks ou plateformes spécifiques supportent le système cible et il faut prévoir des transformations de ciblage, de génération de code, d'intégration de frameworks, etc.

4.3.5 Dynamique du processus

Les descriptions précédentes sont des descriptions structurelles du processus, elle ne décrivent pas son déroulement ou sa dynamique. Même si ces diagrammes de processus contiennent des activités ce ne sont pas des diagrammes d'activités (au sens UML du terme). Ce sont des graphes de dépendances qui imposent une certaine logique sur l'enchaînement des activités. Par exemple, comme un modèle dépend de son méta-modèle le méta-modèle doit exister avant l'élaboration des modèles qui y sont conformes. Ce méta-modèle doit donc préexister (un méta-modèle classique) ou doit être conçu explicitement au cours de la phase d'ingénierie du processus spécifique. De même une activité, ne peut avoir lieu que si ses modèles en entrée, ses ressources et ses acteurs sont disponibles.



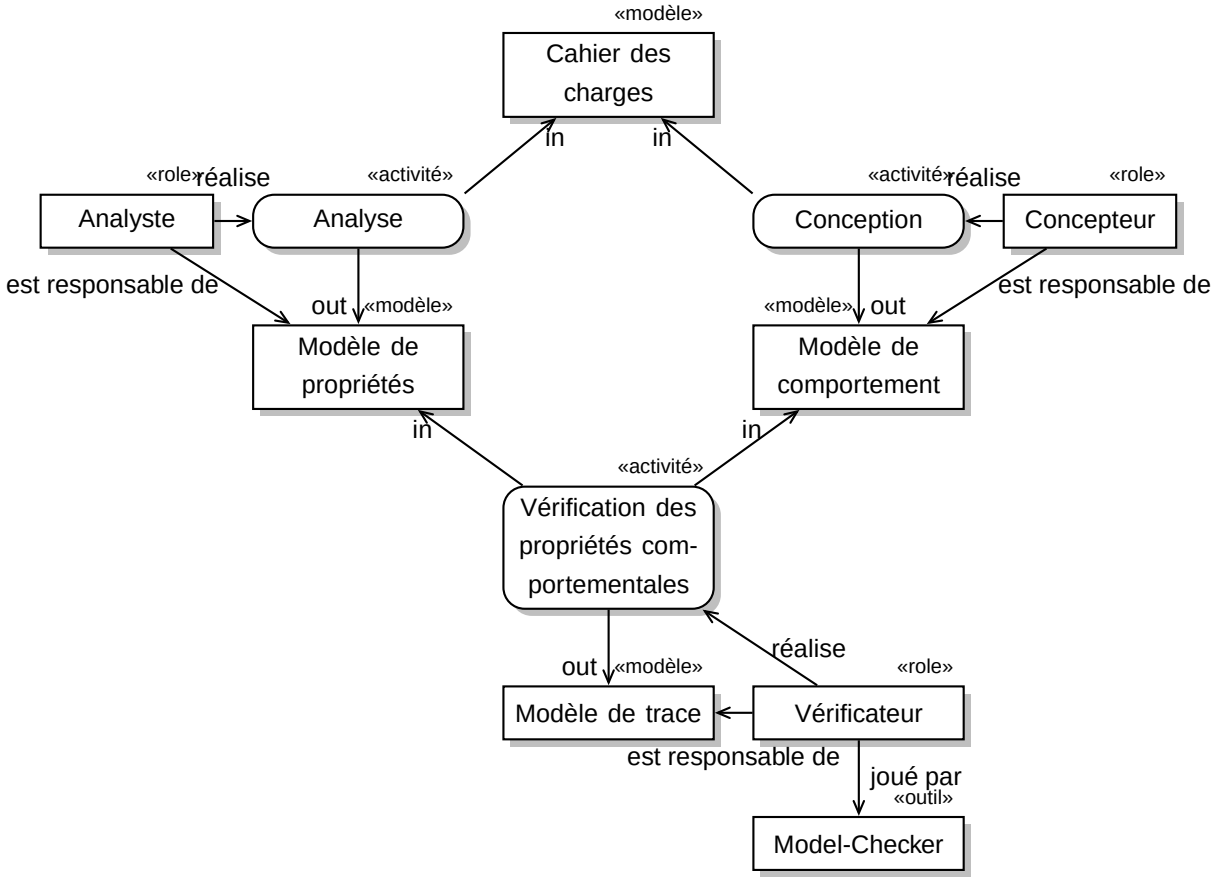
F . 4.15 – La figure 4.10 vue comme un diagramme d'activité

Afin d'aller au delà de ce simple graphe de dépendances, des diagrammes d'activités UML peuvent être utilisés. Ainsi la figure 4.10 peut être redéfinie en utilisant un diagramme d'activité (figure 4.15) pour montrer plus précisément l'enchaînement des activités et les flots de modèles circulant entre les activités. Ces diagrammes peuvent être particulièrement nécessaires en cas d'itération de certaines activités ce qui n'apparaît pas sur les diagrammes de structure du processus. A titre d'illustration prenons le cas d'une activité de vérification de modèles où la conception de modèles doit être affinée lors d'itérations jusqu'à obtention d'un modèle ne violant pas les propriétés spécifiées. La figure 4.16 donne tout d'abord une vue structurelle des différentes activités en précisant notamment les rôles et les responsabilités.

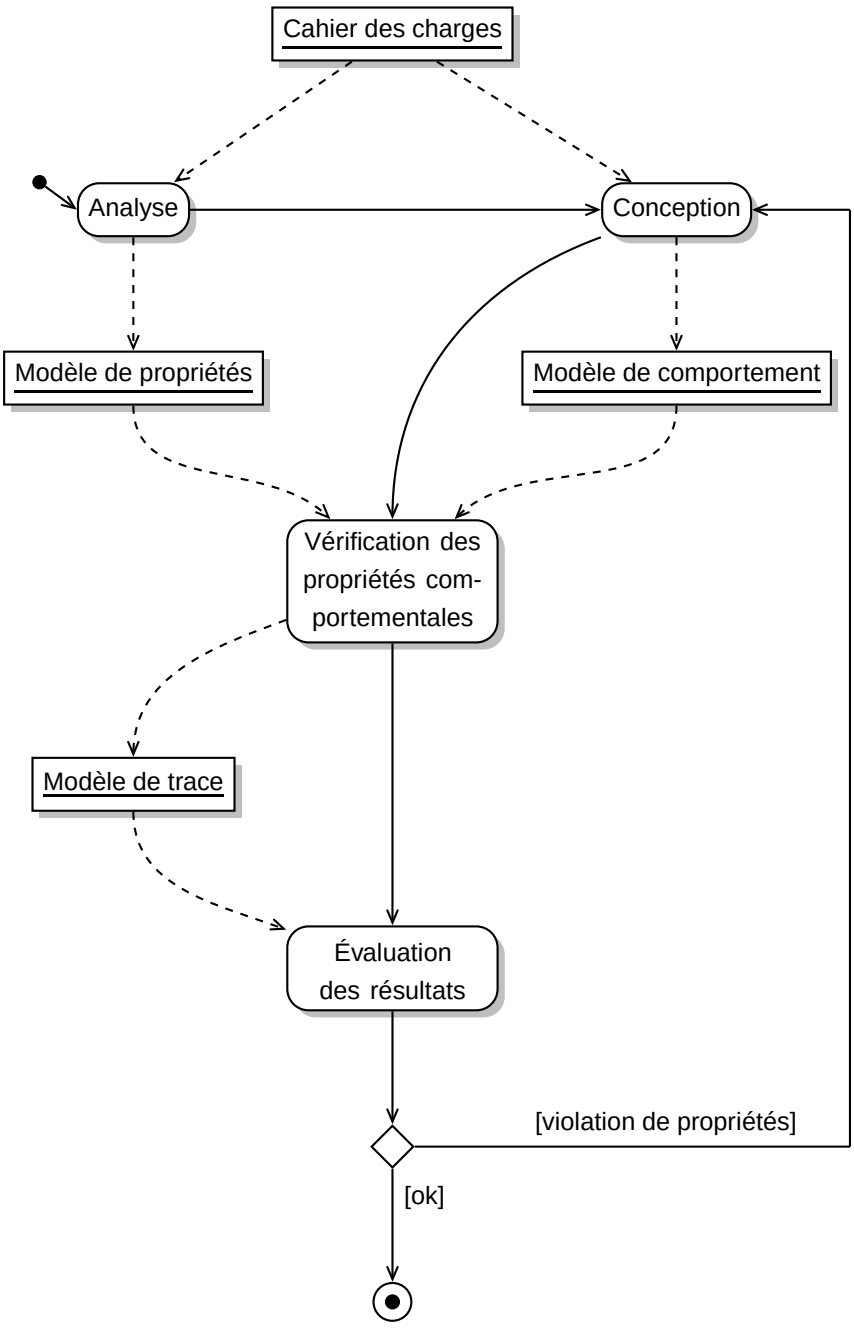
Cette description fait apparaître les rôles d'analyste et de concepteur joués par des acteurs humains et le rôle de vérificateur joué par une machine. Elle pourrait également préciser les différents méta-modèles utilisés (automates, logique temporelle, etc.). A partir

de ce diagramme structurel il est possible de préciser une organisation temporelle des activités et un flot de circulation des différents modèles. La figure 4.17 donne une organisation dynamique de ces différentes activités.

Ces diagrammes d'activités, décrivant des flots de travaux, peuvent être utilisés soit pour modéliser une dynamique globale du processus (un modèle en cascade par exemple), soit une dynamique plus locale au sein d'une même activité pour la découper en étapes plus fines. Ces modèles de dynamique peuvent servir soit de guide méthodologique précisant la manière dont les activités vont se succéder au sein du processus, soit être utilisés pour mieux automatiser le processus de développement en enchaînant des activités de transformation de modèles. Ils peuvent donc servir à opérationnaliser le processus et les flots de travaux au sein d'environnements spécialisés comme les ALM ou les environnements d'ingénierie de processus comme EPF [EPF].



F . 4.16 – Description structurelle d'un processus spécifique intégrant une activité de vérification de modèles



F . 4.17 – Organisation dynamique des activités

4.4 Ingénierie de processus spécifiques

L'ingénierie de processus de développement logiciels consiste, de maniére générale, à concevoir, déployer ou encore maintenir des processus de développement logiciel. Cette ingénierie doit alors définir (et selon le cas concevoir et réaliser) ou prescrire les rôles, activités, acteurs, outils, produits, ressources, etc. de ces processus et les intégrer en un tout cohérent. Lorsque ces processus de développement sont spécifiques à une classe d'applications, cette ingénierie devra également identifier et intégrer des activités, des rôles, des outils, etc. spécifiques à ce domaine d'application. Enfin, lorsque ces processus de développement doivent être orientés modèles, cette ingénierie devra aussi prescrire l'utilisation de langages ou méta-modèles éventuellement spécifiques, et fournir de l'outillage pour la production et la transformation des modèles.

Une telle ingénierie repose sur un processus que l'ont pourrait qualifier de *méta-processus*, dans le sens *processus «qui parle de» processus*. De plus certains de ses constituants sont à un niveau méta par rapport aux constituants du processus de développement. Par exemple, là où les activités du processus de développement produisent des modèles, décrits à l'aide de langages de modélisation et, conformes à des méta-modèles, certaines activités du méta-processus produisent des *méta-modèles*, décrits avec des *langages de méta-modélisation* et, conformes à des *méta-méta-modèles*. De même, ce méta-processus peut contenir des rôles du genre *expert en méta-modélisation* et utiliser des *méta-outils*. Cependant le terme *méta-processus* n'ayant pas de définition largement reconnue nous préférons plutôt parler d'ingénierie de processus.

L'ingénierie de processus spécifiques peut s'avérer complexe si tous les aspects du processus cible doivent être décrits, ce qui est hors de portée de la thèse. La suite du travail se focalise plus particulièrement sur l'aspect «orientés modèles» des processus cibles. C'est à dire que le focus sera mis sur les méta-modèles, les transformations de modèles, les rôles et les activités spécifiques à ces produits, etc.

4.4.1 Principes généraux

L'objectif envisagé par l'ingénierie décrite dans cette partie est de concevoir et mettre en place des processus orientés modèles, dans des environnements outillés, pour le développement d'applications spécifiques à des classes d'applications. L'intérêt escompté est une meilleure adéquation entre un projet logiciel et son processus de développement.

Deux résultats essentiels sont attendus d'un telle ingénierie. Le premier résultat attendu est un *ensemble de constituants* (produits, ressources, outils, etc.) exploités lors du déroulement du processus après son transfert en phase d'exploitation. L'existence de constituants spécifiques, tels que les modèles de transformations, a déjà été évoqué au paragraphe 4.3. Le deuxième résultat attendu est la description (*modèles statiques et dynamiques*) d'un processus, ou d'une famille de processus spécifiques décrivant les intervenants, ce qu'ils font, comment il le font et avec quoi, etc. Plusieurs exemples de tels modèles ont été donnés au paragraphe 4.3.

La démarche globale à adopter pour obtenir de manière conjointe et rationnelle ces deux types de résultats fortement couplés mériterait plus de réflexion et plus d'expérience que le cadre restreint de cette thèse. Afin de progresser, une séparation en deux grandes étapes a été effectuée : 1) définition et réalisation des constituants spécifiques, 2) élaboration de modèles de processus spécifiques intégrant les constituants.

Les principaux constituants spécifiques à obtenir et auxquels on s'intéressera plus particulièrement dans cette thèse sont des langages de modélisation, définis par leurs méta-modèles, des transformations de modèles M2M et M2C, des outils, et des plates-formes et/ou frameworks facilitant l'implantation du logiciel. Afin d'obtenir ces différents constituants, une démarche méthodologique constituée d'un ensemble d'activités effectuées sous la responsabilité de rôles/acteurs spécifiques est nécessaire. Ces activités, rôles et produits sont décrits dans les paragraphes 4.4.2 à 4.4.4.

La modélisation d'un processus spécifique suppose l'utilisation d'un *langage de modélisation de processus*. SPEM [OMG08a], signifiant «Software Process Engineering Metamodel», qui peut être traduit par «Méta-modèle pour l'Ingénierie de Processus Logiciels» a été défini dans ce but. Cependant, SPEM n'intègre pas directement les concepts de l'IDM, c'est pourquoi un schéma conceptuel simplifié (figure 4.8) a été préféré pour représenter les différents extraits de modèles de processus proposés dans cette thèse. Ces modèles pourront toujours être rendus conformes à un méta-modèle plus précis quand un langage de modélisation de processus orientés modèles, largement adopté, existera. L'avantage premier d'une conformité à un méta-modèle précis est alors la possibilité de permettre une meilleure gestion, voire une exécution du processus au sein d'un environnement dédié à cet usage. Afin d'obtenir les différents modèles d'un processus spécifique, une démarche méthodologique ou un guide est également nécessaire ; elle sera décrite au paragraphe 4.4.5.

Avant de détailler les différents points cités rappelons que, de manière générale et non exhaustive, un *processus d'ingénierie de processus logiciels spécifiques et dirigés par les modèles* :

- prescrit et/ou produit des langages de modélisation et/ou méta-modèles,
- utilise des méta-langages ou langages de méta-modélisation (ex : Ecore),
- produit des transformations de modèles ou des modèles de ces transformations,
- produit des transformations d'import/export vers les outils d'un domaine spécifique,
- produit des transformations de ciblage vers des plateformes ou frameworks spécifiques,
- exploite des méta-outils et/ou des outils de la communauté IDM (ex : ATL-IDE),
- utilise un ou des langages de modélisation de processus (ex : SPEM),
- produit des modèles de processus définis avec ces langages,
- prescrit des activités, rôles, produits classiques ou spécifiques.

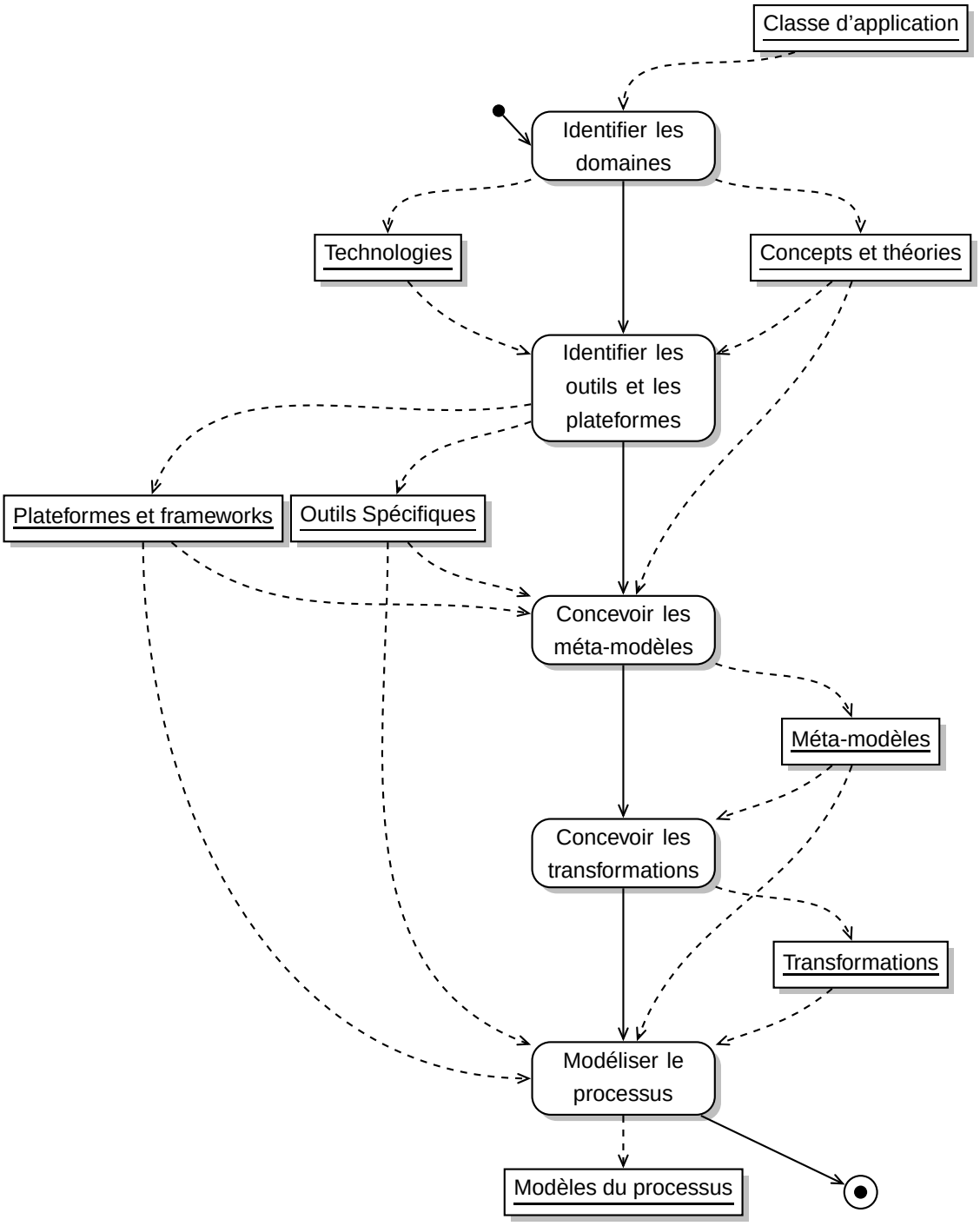
4.4.2 Activités

L'ingénierie de processus spécifiques est supportée par son propre processus. Selon une approche orientée activités ce processus est constitué d'éléments primaires qui sont des activités, des rôles, et des produits ainsi que d'éléments secondaires tels que des ressources

et des acteurs [MHL07]. Ces différentes entités sont décrites dans ce paragraphe et les deux suivants.

La figure 4.18 présente, sous la forme d'un diagramme d'activités simplifié, la démarche générale envisagée. Cette démarche se concentre sur les deux aspects essentiels du processus cible, à savoir que : 1) il est spécifique à une classe d'applications intégrant des domaines spécifiques ayant leurs propres concepts, théories, outils, etc., 2) il est dirigé par les modèles et doit être outillé. Cinq grandes activités sont envisagées :

1. Partant d'une définition de la classe d'applications envisagée, une première activité consiste à *identifier et recenser les domaines* nécessaires à la conception du logiciel. Elle permet de recenser les différentes théories, les concepts qui les accompagnent, les compétences ou expertises requises, et les éventuelles technologies liées à ces domaines et indispensables à la réalisation des applications. En cela, cette étape cerne les limites théoriques et techniques de la classe d'applications.
2. Une fois les domaines identifiés, il faut pouvoir réaliser et manipuler des modèles relevant de ces différents domaines. Pour cela, l'activité suivante consiste à *identifier et recenser les outils spécifiques, les frameworks cibles, les plateformes* d'exécution envisagées, etc. Cette approche traitant de manière conjointe les outils des domaines métier et les frameworks ou plateformes cibles consiste à traiter le métier de l'ingénierie logicielle comme les autres métiers ou domaines. Les frameworks sont généralement une façon de réduire la distance entre un domaine métier et une plateforme d'exécution générique.
3. Partant des outils, plateformes, frameworks, concepts et théories des domaines qui ont été recensés, la troisième activité, va consister à *recenser les méta-modèles nécessaires au développement et pour ceux qui ne sont pas disponibles, les concevoir et les réaliser*. Ces méta-modèles sont conçus en fonction des besoins théoriques ou technologiques des applications et des langages de modélisation utilisés par les outils. Cette activité nécessite une double expertise ou une collaboration entre des experts des domaines métiers et des experts en méta-modélisation. Elle est supportée par l'usage d'outils mis à disposition par la communauté IDM (ATL-IDE, EMF, Sintaks, etc.)
4. L'activité suivante a pour objectif de faciliter la mise en relation des différents modèles réalisés au cours du développement, ceci en fournissant les moyens pour faire progresser le processus de développement en bénéficiant notamment des atouts offerts par les différents outils des domaines. Cette activité consiste donc à *concevoir les transformations de modèles adéquates* entre les différents langages/méta-modèles et les transformations permettant l'import ou l'export de modèles par les outils. Les compétences nécessaires à la réalisation de cette activité sont multiples et peuvent nécessiter la collaboration d'experts IDM et d'experts des domaines correspondant aux méta-modèles sources et méta-modèles cibles. La concrétisation de ces transformations pourra également bénéficier de l'outillage mis à disposition par la communauté IDM. C'est aussi lors de cette activité que sont définies les transformations de ciblage vers des plateformes cibles ou des frameworks spécifiques abstrayant ces plateformes.
5. Finalement, l'ensemble des constituants recensés, conçus et réalisés doivent être intégrés en un processus cohérent. Pour cela, la dernière activité est la *modélisation*



F . 4.18 – Diagramme d'activité de la démarche générale envisagée

du processus cible en définissant notamment un cycle de développement adéquat et intégrant éventuellement des aspects propres à la culture de l'entreprise.

Ces cinq activités aboutissent à l'obtention d'un modèle de processus et un ensemble de constituants essentiels à un processus dirigé par les modèles. Bien évidemment un tel processus doit être assemblé et transféré en exploitation, être exploité, et être maintenu. Ces points ne sont pas abordés par la thèse.

4.4.3 Rôles et acteurs

Rappelons que les rôles sont les entités abstraites qui réalisent les activités et sont responsables des produits. Ces rôles sont assurés par des entités réelles (physiques) qui sont les acteurs : des personnes, des machines, des équipes, etc.

A ce stade il est nécessaire de distinguer les rôles et acteurs intervenant lors de l'exploitation du processus spécifique (développement du logiciel) de ceux intervenants lors de l'ingénierie de ce processus spécifique.

Comme il a été décrit dans l'état de l'art, les acteurs récurrents intervenants lors de l'exploitation du processus sont *le chef de projet, le client*, et les différents intervenants de mise en œuvre du logiciel tels que *l'analyste, l'architecte logiciel, le développeur, le testeur*, etc. Ces acteurs sont multiples, ont des compétences, des connaissances, une expérience, etc. variées. En fonction des rôles qui leur sont attribués ils interviendront, en tant qu'experts, plutôt au niveau des domaines métier ou plutôt en ingénierie logicielle ou encore dans la gestion du cycle de développement. Ces trois aspects, *domaines métier, ingénierie logicielle, gestion du processus*, illustrés par la figure 4.2 ont été discutés au début de ce chapitre.

Lors de la phase d'ingénierie du processus, l'objectif de la modélisation précise du processus est de définir (ou prescrire) ces rôles, les compétences requises pour ceux-ci, et de les associer aux activités qu'ils réalisent et aux produits dont ils sont responsables. Ces rôles doivent être attribués à des humains, des machines (ou outils), des équipes, etc. Dans le paragraphe 4.3 plusieurs exemples ont été donnés notamment les rôles de vérification et de simulation attribués à l'outil LTSA (figure 4.9) de synthèse de modèle attribué à l'outil Supremica (figure 4.14) ou encore les rôles de présentation (export) ou d'analyse (import) de modèles attribués à des outils comme Sintaks (figure 4.13). Un cas particulier est celui de l'utilisation de méta-outils tels que MIC-GME vu au paragraphe 2.4.3. Ces outils sont alors configurés lors de la phase d'ingénierie du processus à l'aide de méta-modèles, puis des rôles variés (génération de code, analyse de modèles, etc.) leurs sont attribués en phase d'exploitation.

Certains des acteurs et rôles intervenant en phase d'exploitation du processus interviennent également en phase d'ingénierie du processus. En effet, une expertise pour les trois aspects, *domaines métier, ingénierie logicielle, gestion du processus/méthodes*, doit être présente. La spécification et la réalisation de méta-modèles spécifiques ou de transformations de modèles nécessite également l'intervention d'un *acteur IDM* ayant des compétences relevant de ce domaine. Cet acteur IDM, jouant un rôle d'*expert en méta-modélisa-*

tion va interagir avec les experts des domaines afin de choisir les outils nécessaires aux développements et envisager leur intégration au sein du processus. De même, il va interagir avec des experts de l'ingénierie logicielle afin de constituer les méta-modèles associés aux architectures cibles, aux frameworks, aux plateformes, etc. En tant qu'*expert en transformation de modèles* il aura pour responsabilité de produire un ensemble de modèles de transformation nécessaires à la mise en relation des différents langages. Ces transformations vont par la suite faciliter la construction de chemins que devra suivre le processus de développement.

4.4.4 Produits

En dehors des produits propres et internes au processus d'ingénierie tels que le recensement des domaines, des concepts et théories, etc., les produits issus du processus d'ingénierie et utilisés par les processus conçus sont des méta-modèles, des transformations, des outils ou méta-outils configurés, des frameworks, et des modèles du processus spécifique.

Les méta-modèles sont des produits du processus d'ingénierie et sont considérés comme des ressources dans les processus conçus. Ces méta-modèles doivent être conçus à l'aide de langages de méta-modélisation et doivent donc être conformes à un méta-méta-modèle (MOF, EMOF, KM3, etc.). Ceci pose le problème du choix de ce méta-méta-modèle et d'un référentiel éventuel de méta-modèles. Ce point a été discuté dans le paragraphe 3.5.1 de l'état de l'art. Dans notre laboratoire des travaux ont été effectués sur la façon de définir ces méta-modèles de manière fonctionnelle en utilisant comme langage de méta-modélisation le langage fonctionnel de haut niveau Haskell [TT09, TCT08, TT08]. Les avantages d'une telle approche est de pouvoir bénéficier d'un langage puissant ayant une sémantique parfaitement définie. L'inconvénient majeur de cette approche est la distance existant entre une telle approche et les pratiques courantes de la communauté IDM.

Les *transformations de modèles* peuvent être produites sous la forme de modèles de transformations. Ces modèles peuvent être décrits à l'aide de langages de transformation et sont alors conformes aux méta-modèles décrivant ces langages (ATL, Sintaks, Xpand, etc). Ces transformations peuvent être également réalisées sous la forme de programmes spécifiques à l'aide d'environnements de programmation spécifiques tels que Kermeta, ou plus simplement à l'aide de langages plus ou moins classiques. Dans notre laboratoire des travaux ont été effectués sur la façon de décrire ces transformations de manière fonctionnelle en utilisant le langage fonctionnel Haskell [TTH08]. L'avantage d'une telle approche est d'apporter à la fois la sémantique précise de ce langage et d'offrir la puissance d'abstraction d'un langage fonctionnel moderne.

La *réalisation d'outils* spécifiques à l'aide de méta-outils tels que MIC-GME ou MetaEdit nécessite la réalisation de méta-modèles conformes aux méta-méta-modèles propres à ces méta-outils, par exemple FCO (First Class Object) pour MIC-GME ou GOPRR (Graph, Object, Property, Relation, Role) pour MetaEdit. Ces méta-méta-modèles spécifiques peuvent alors poser des problèmes d'intégration à cause de leur distance conceptuelle par rapport aux standards définis par la communauté IDM.

La *création de frameworks* visant à diminuer la distance entre des modèles métiers et des abstractions logicielles de plus bas niveau relève d'une ingénierie logicielle relativement classique, elle est typiquement orientée objets et peut faire un large usage de patterns de conception pour offrir suffisamment de flexibilité à ces frameworks. L'ingénierie d'un processus spécifique peut également prescrire l'utilisation de frameworks existants. Ces frameworks peuvent même couvrir différents domaines comme c'est le cas du framework Ptolemy II vu au paragraphe 2.4.4 de l'état de l'art. Il est alors nécessaire de fournir des modèles (méta-modèles) adéquats de ces domaines, et les intégrer au reste du développement notamment par des transformations de modèles de type génération de code dont la logique doit être conforme à la logique du framework.

Enfin le dernier produit important d'une ingénierie de processus est une *collection de modèles*, décrivant les aspects structurels et les aspects dynamiques des processus logiciels envisagés. Ces modèles pourront être informels ou plus formels en les rendant conformes à un méta-modèle de modélisation de processus tel que SPEM. L'avantage d'une modélisation plus formelle du processus est de permettre une opérationnalisation de ce processus à l'aide d'un environnement dédié à cet usage. L'obtention de ces modèles est détaillé dans le paragraphe suivant.

4.4.5 Modélisation d'un processus spécifique

La modélisation d'un processus spécifique requiert l'intervention d'un expert en ingénierie de processus et/ou ingénierie des méthodes. Cet expert devra collaborer avec les autres experts identifiés précédemment afin d'obtenir un processus spécifique à la classe d'applications envisagée et intégrant les différents artefacts nécessaires à une approche dirigée par les modèles. Bien évidemment un tel modèle de processus pourra s'appuyer largement sur les structures habituelles utilisées pour décrire les cycles de vie : modèle en cascade, modèle en V, modèle en spirale, etc.

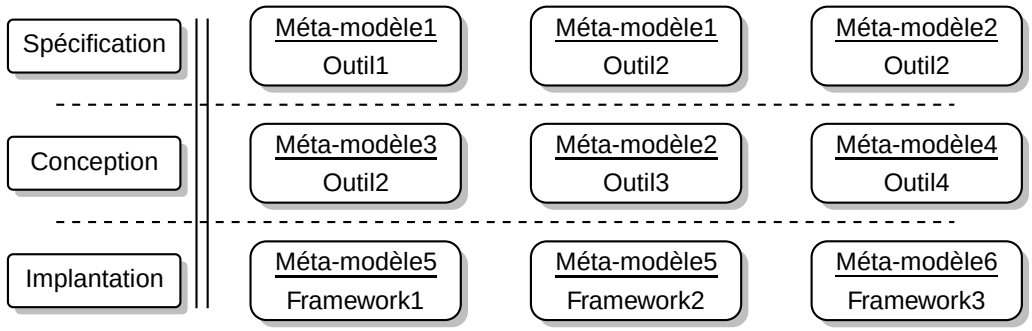
Nous proposons également que ce modèle soit construit en accord avec les principes généraux énoncés dans ce chapitre, en utilisant comme cadre d'analyse l'empilement à quatre niveaux proposé à la figure 4.3. Pour rappel : les processus se décomposent en activités, ces activités sont supportées par des langages (de modélisation) et manipulent des modèles, les langages sont supportés par des outils spécifiques, des plateformes, des frameworks, etc. en utilisant si nécessaire des transformations d'import/export (M2C).

Partant de cette proposition il est possible d'ébaucher une démarche fondée sur quatre grandes étapes pour l'obtention des modèles de processus :

1. Identifier les différentes activités du processus.
2. Recenser, pour chaque activité, les langages/méta-modèles utilisés.
3. Relier, selon les besoins du processus, les langages par des transformations.
4. Intégrer les outils spécifiques, plateformes, frameworks, etc.

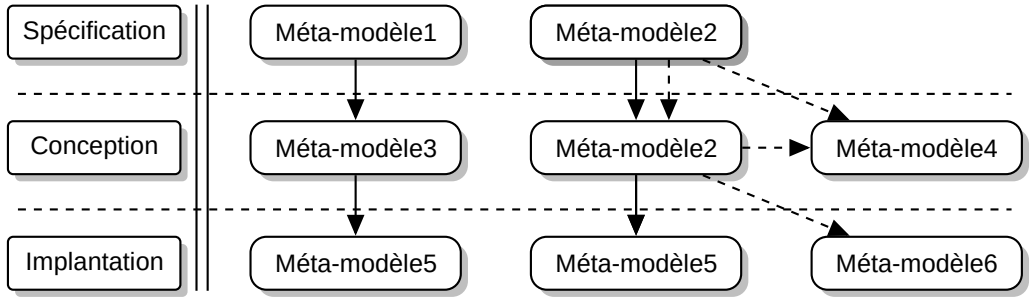
Ces modèles sont constitués d'activités, de produits, de rôles, d'acteurs, de ressources, d'outils, etc. Pour rappel, de nombreux extraits de modèles ont été donnés dans la partie 4.3 avec :

- un modèle prescrivant l'utilisation du langage FSP (figure 4.6),
- un modèle décrivant des activités de modélisation et de test de comportement (figure 4.9),
- un modèle décrivant une activité de transformation FSM2FSP (figure 4.11),
- un modèle d'une activité de transformation vers des syntaxes concrètes (figure 4.13),
- un modèle décrivant l'activité de synthèse d'un modèle de superviseur (figure 4.14),
- un modèle décrivant une activité de vérification de modèle (figure 4.16).



F . 4.19 – Répartition des méta-modèles, outils, frameworks, etc selon les phases d'un cycle de vie

Afin de faciliter l'obtention de ces modèles un tableau classifiant l'usage et le couplage des méta-modèles, frameworks, et outils en fonction des différentes phases du cycle de vie peut-être effectué (figure 4.19).



F . 4.20 – Mise en place de relations entre les différents méta-modèles. Plusieurs chemins sont possibles lors d'un développement

Pour converger vers un processus offrant une meilleure intégration de ces différentes entités, des relations sont envisagées entre les méta-modèles (figure 4.20). En s'appuyant sur des modèles de transformation pour concrétiser ces relations, il sera alors possible de fournir des chemins permettant de «naviguer» entre les différents langages de modélisation (et domaines) nécessaires à la production d'un système logiciel. Des exemples plus concrets des figures 4.19 et 4.20 seront donnés au chapitre 5 (figure 5.17).

L'existence de plusieurs chemins de développement permet de raisonner en terme de processus destiné à une famille d'applications plutôt qu'à un processus dédié à une application unique.

4.5 Conclusion

Ce chapitre décrit nos travaux concernant l'élaboration de processus logiciels spécifiques et dirigés par les modèles. Ces processus donnent une place prépondérante aux concepts de modèles, de méta-modèles et de transformations de modèles de type M2M ou M2C. Afin d'aborder cette problématique et en l'état actuel des connaissances nous avons adopté une approche ascendante. Cette approche, partant d'exemples types, propose une description de ce que seraient de tels processus, pour en donner une meilleure définition en articulant les différents constituants qui les composent. Afin d'y parvenir nous avons proposé un schéma conceptuel associant les concepts du domaine des processus (activités, rôles, produits, acteurs, ressources) à ceux de l'ingénierie dirigée par les modèles. Nous avons adopté cette approche, qui se limite à l'utilisation d'un schéma conceptuel, plutôt que d'utiliser un vrai méta-modèle de processus, afin de simplifier notre approche et éviter la lourdeur d'un méta-modèle qui, de plus, risque d'être immature comme semble l'attester le passage obligé de SPEM 1.0 à SPEM 2.0 ([OMG08a], page 8). Partant d'une meilleure connaissance de ces processus logiciels dirigés par les modèles une ingénierie de ces processus est présentée. Cette ingénierie a pour objectif de concevoir et mettre en place les différents constituants du processus, notamment ses méta-modèles, transformations, outils et plateformes ainsi que d'en donner un modèle statique et dynamique. Pour définir ce modèle du processus, l'intérêt d'utiliser, voire de définir un vrai méta-modèle de processus, serait de permettre l'hébergement et la manipulation des modèles du processus en vue du pilotage du processus par un environnement dédié à cet usage. Ceci est l'un des objectifs du projet EPF (Eclipse Process Framework) [EPF] qui vise à supporter SPEM 2.0 comme méta-modèle. Un autre projet récent concernant le pilotage de processus dirigés par les modèles est le projet TopProcess [GCC09].

La problématique de définir un langage de modélisation adapté à la description de processus dirigés par les modèles est devenu un sujet d'actualité dans la communauté IDM comme le montrent les travaux autour de SPEM4MDE [DLC09] et est l'une des perspectives de cette thèse.

Chapitre 5

Proposition d'un processus outillé pour le développement logiciel des SED

5.1 Introduction

L'objectif est de mettre en œuvre les propositions du chapitre 4 pour proposer un modèle de processus de développement pour les SED, intégrant des langages, des méta-modèles, des transformations, des outils et des plateformes spécifiques à ce domaine d'application. Ce processus doit permettre un développement classique de certaines parties logicielles, enrichies par les aspects théoriques propres aux SED. Il doit permettre également un ciblage sur des plateformes et modèles d'exécution spécifiques qui ne sont pas nécessairement pris en compte par des environnements de développement du type de Matlab/Simulink [Mat05] ou des outils tels que Supremica [AFF03].

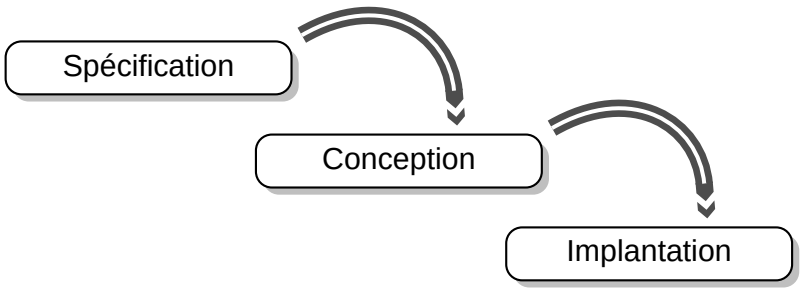
L'approche présentée au chapitre 4 s'appuie sur deux périodes ou phases bien distinctes. La première période consiste à élaborer un processus de développement spécifique et est l'objet de ce chapitre. La deuxième période consiste à exploiter ce processus spécifique et ses outils pour le développement d'une famille d'applications relevant de ce domaine, ce sera l'objet du chapitre 6.

Pour l'élaboration d'un processus spécifique, le chapitre 4 propose une démarche qui consiste à 1) identifier et recenser les domaines, 2) identifier les outils, frameworks, plateformes, etc. 3) concevoir et réaliser les méta-modèles spécifiques, 4) concevoir et réaliser les transformations pour relier les différents méta-modèles et 5) modéliser le processus spécifique.

La partie 5.2, aborde les points (1) et (2) en décrivant le domaine des SED. L'élaboration des méta-modèles sera détaillé dans la partie 5.3, et les transformations dans la partie 5.4. La partie 5.5 propose une synthèse en vue de l'élaboration de processus spécifiques.

5.2 Domaine des Systèmes à Événements Discrets

L’étude se limite au cas des logiciels de contrôle-commande de Systèmes purement discrets ou encore dits à Événements Discrets. Pour ce domaine et de façon à garder les illustrations claires, un modèle classique et simplifié de processus de développement en cascade est envisagé. Présenté à la figure 5.1, ce processus simplifié consiste à réaliser successivement les activités de *Spécification*, de *Conception* et d’*Implantation*.



F . 5.1 – Modèle simplifié du processus de développement

Dans ce domaine, pour garantir la réalisation d’une loi de commande, deux approches typiques sont possibles :

1. Le recours à la commande par supervision initiée par P.J. Ramadge et W.M. Wonham [RW89] (Annexe B), qui permet la synthèse automatique des modèles de superviseurs (lors d’une activité de conception par exemple) à partir d’un modèle de propriétés et d’un modèle permissif du système.
2. L’utilisation du Model-Checking [CGP00], qui permet sur la base de propriétés spécifiées pour un système supervisé, de vérifier que son comportement est conforme aux attentes du client (avant l’implantation). Cette approche peut être utilisée seule en cas de synthèse manuelle du superviseur ou peut compléter la première approche [CTT07].

Ces deux approches présentent des avantages et des inconvénients pouvant être contrebalancés par une utilisation conjointe au sein d’un même processus de développement offrant de la flexibilité.

Les SED sont souvent mis en œuvre à l’aide d’automates programmables en utilisant par exemple un outil de type Supremica [AFF03] pour la synthèse de superviseur. Cependant, les aspects théoriques apportés par la commande par supervision pourraient bénéficier à des développements logiciels plus classiques.

Afin d’appuyer ce point de vue, nous avons pris le parti d’illustrer notre approche à l’aide de plateformes différentes des plateformes types. Deux exemples seront traités dans le chapitre 6 :

1. Le premier exemple (chapitre 6.1) concerne la mise en œuvre d’une application décentralisée mobile. Il s’appuie sur le modèle du jeu du chat et de la souris utilisé par P.J.Ramadge et W.M.Wonham, [RW89], pour présenter le principe de la commande

par supervision. L'aspect décentralisé et mobile de l'application sera supporté par une implantation sur la plateforme *J2ME* de *Sun/Oracle*.

2. Le deuxième exemple (chapitre 6.2) propose de traiter un problème de producteur consommateur. Cet exemple sera implanté sur une plateforme spécifique multi-agents (framework PI [Ras06]) et combine une approche par synthèse de superviseur et une vérification par Model-Checking.

Étant donné les applications envisagées, les domaines de la commande par supervision et du Model-Checking ne peuvent suffire à la mise en place d'un modèle de processus de développement adéquat. Il est également nécessaire de considérer l'ingénierie logicielle classique, avec ses langages de modélisation et concepts propres (objets, patterns, frameworks, etc.) afin de couvrir l'ensemble des activités du processus de développement. Pour cela, des langages de modélisation tels que UML [OMG10a], SysML [OMG10d] ou encore AADL [Dis04] vus dans l'état de l'art de ce manuscrit peuvent être utilisés. Pour les méta-modèles de ces langages non spécifiques, un outil tel que TopCaseD dans lequel ils sont déjà implantés peut être utilisé.

Le domaine métier des SED apporte ses propres outils et formalismes. Si le choix est large, l'étude se limite à des outils et formalismes manipulant des automates finis [Col05] qui vont permettre de synthétiser/générer des contrôleurs discrets à l'aide d'un outil de type Supremica [AFF03], et des outils mariant algèbres de processus et logiques temporelles, tel que LTSA [MK06] pour les aspects vérification de modèles.

L'objectif final du développement est l'obtention d'un système logiciel implanté sur une plateforme standard ou spécifique. Ainsi, afin de préparer l'implantation du système logiciel, il est nécessaire d'identifier les frameworks et les plateformes susceptibles de les accueillir. Les plateformes d'exécution et les frameworks sont nombreux dans le domaine logiciel. Parmis les plateformes standards, il est possible de citer la plateforme Java de *Sun/Oracle* ou .net de *Microsoft*.

Afin de mieux illustrer l'approche, deux plateformes sont envisagées, l'une standard, l'autre spécifique. Ces deux plateformes exploitent cependant le même langage de programmation, à savoir Java, qui sert de référence pour effectuer la génération de code. Cependant, afin de faciliter l'activité d'implantation, des frameworks plus proches du domaine métier sont mis en œuvre sur ces plateformes.

Ainsi, l'implantation de l'application mobile s'appuie sur J2ME de *Sun* complété du framework DestKit [Col05], un framework permettant l'exécution d'automates. L'implantation de l'exemple du producteur-consommateur s'appuie sur le framework d'exécution de systèmes multi-agents PI [Ras06], fondé sur les LTS et utilisant une machine virtuelle Java classique.

En considérant l'utilisation de ces langages et outils, et une fois les méta-modèles réalisés, un certain nombre de transformations vont être nécessaires avec, entre autre, des transformations vers les outils pour faciliter la conception (avec Supremica ou LTSA par exemple) ou vers les plateformes spécifiques ou frameworks J2ME, PI, DestKit pour faciliter l'implantation.

5.3 Élaboration des méta-modèles spécifiques

Pour concevoir efficacement les méta-modèles pour les différents langages de modélisation identifiés lors de l’activité précédente, il est préférable de s’appuyer sur un outil IDM. Ainsi, pour la définition et l’exploitation des méta-modèles, ce travail s’est appuyé sur l’outil TopCaseD [Top05] qui prédéfinit plusieurs formalismes et méta-modèles comme les diagrammes UML.

Cet outil a été combiné à EMF [BSM03] pour produire des diagrammes Ecore permettant la représentation des méta-modèles spécifiques. Ensuite, grâce à l’outil de génération de plugins Eclipse [GW03], un composant est produit puis intégré à l’environnement de développement. Ce composant se comporte alors comme un outil permettant de concevoir dans, l’IDE Eclipse, des modèles conformes à ces méta-modèles.

Les plugins Eclipse sont intéressants car ils peuvent être complétés par une implantation spécifique permettant de définir une réalisation des méta-modèles sous la forme de frameworks. Il est également possible de rajouter à ces plugins une syntaxe concrète graphique à l’aide du framework GMF (Graphical Modeling Framework) [GMF09] facilitant la modélisation.

5.3.1 Méta-modèle des automates à états finis (FSM)

Les automates à états finis sont nécessaires pour la modélisation des SED en particulier pour permettre la mise en œuvre de la commande par supervision. En effet, cette théorie, décrite dans l’annexe B, se base sur la théorie des langages et des automates.

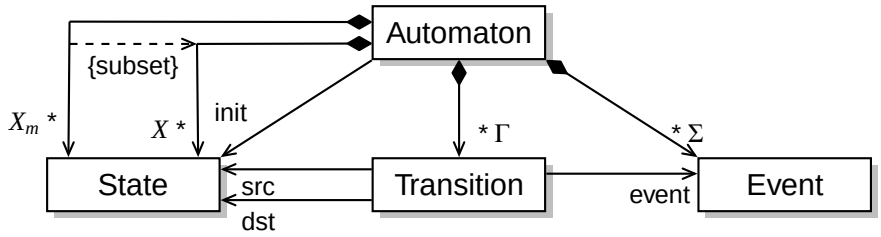
Un méta-modèle pour les automates à états finis aura un double emploi. D’une part, il va servir de pivot entre le processus de développement et l’outil Supremica utilisé pour la synthèse de superviseur. D’autre part, ce méta-modèle va servir de pivot vers le framework Java DestKit [Col05], qui intègre une sémantique opérationnelle pour les automates, facilitant ainsi l’intégration d’automates sur la plateforme cible. Une définition formelle des automates peut être donnée sous la forme d’une structure mathématique :

$$FSM =< X, X_m, x_i, \Sigma, \Gamma >$$

tel que :

- X est l’ensemble des états de l’automate ;
- X_m est l’ensemble des états marqués (ou finaux) de l’automate tel que $X_m \subseteq X$;
- x_i est l’état initial de l’automate ;
- Σ est l’ensemble des événements réalisables par l’automate et est appelé alphabet de l’automate. Selon la commande par supervision, les évènements de l’alphabet peuvent être commandables Σ_c ou non commandables Σ_{uc} et/ou observables Σ_o ou non observables Σ_{uo} tel que $\Sigma = \Sigma_c \cup \Sigma_{uc}$ et $\Sigma = \Sigma_o \cup \Sigma_{uo}$;
- Γ est la relation ternaire définie comme un sous-ensemble de $X \times \Sigma \times X$ et correspondant à l’ensemble des transitions de l’automate.

Cette structure mathématique peut être traduite sous la forme d'un méta-modèle représenté de manière plus classique (figure 5.2). Ce méta-modèle s'articule alors autour des concepts clefs des automates comme ceux d'*états* (State) potentiellement marqués, de *transitions* (Transition) et d'*événements* (Event).

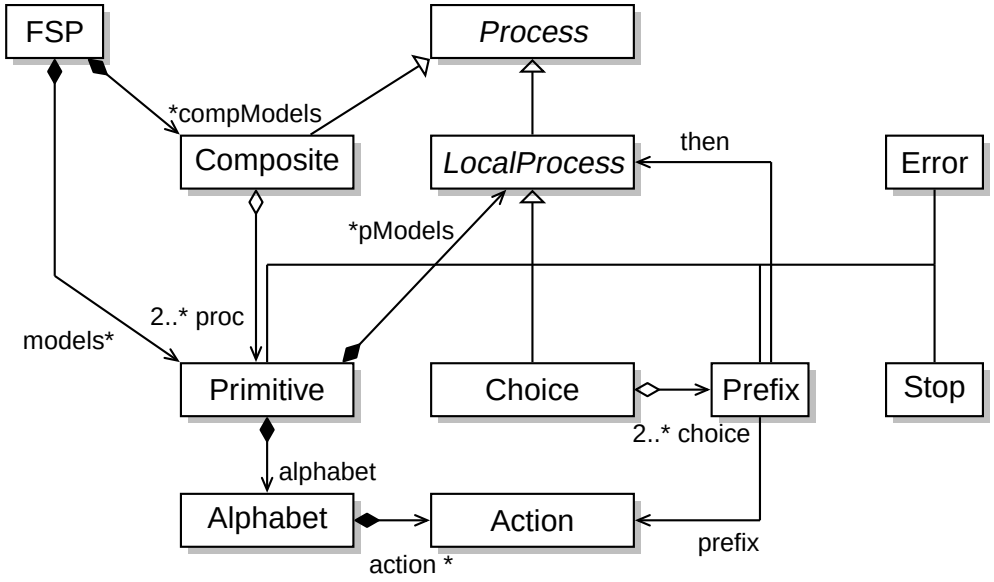


F . 5.2 – Méta-modèles des automates à états finis

5.3.2 Méta-modèle de l'algèbre de processus FSP

Le domaine du Model-Checking décrit en annexe A conduit généralement à utiliser deux langages de modélisation différents. Le premier permet, sous la forme d'automates ou d'algèbres de processus, de décrire le comportement du système. Le deuxième langage permet, sous la forme de logiques temporelles, de spécifier les propriétés de sûreté et de vivacité attendues pour le système.

Plus précisément, pour l'outil LTSA, les deux langages sont :
- FSP, une algèbre de processus utilisée pour modéliser le comportement du système,
- LTL, une logique temporelle permettant de définir et de vérifier les propriétés attendues du système modélisé avec FSP.



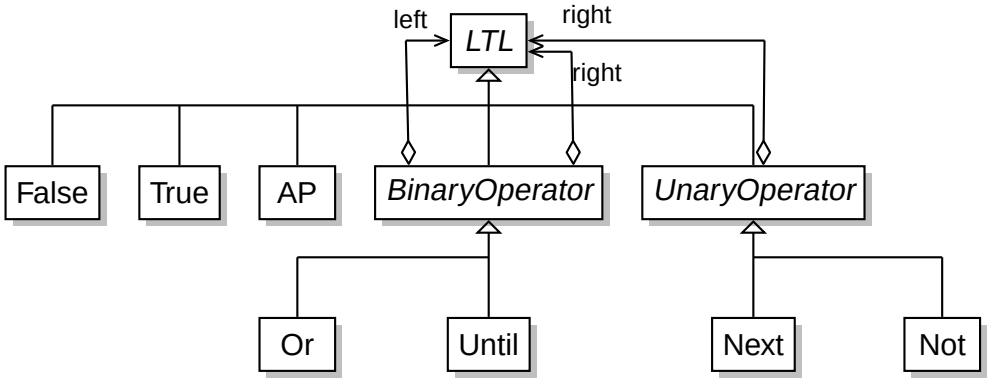
F . 5.3 – Méta-modèle de l'algèbre de processus FSP

Ainsi, le méta-modèle présenté à la figure 5.3[1] va permettre la création de modèles FSP (cohérent avec la grammaire de ce langage et donc avec l’outil LTSA qui l’intègre et vers lequel une transformation est décrite dans les chapitres suivants). Ce méta-modèle se construit autour des concepts de *Process* et de *LocalProcess* qui représentent des entités autonomes pouvant s’exécuter parallèlement. Toutes les entités dérivant du concept de *Process* représentent alors des types de processus particuliers.

Un *Primitive* est un processus permettant la définition d’un modèle constitué de *LocalProcess*. Ces différents processus peuvent être associés à l’aide de processus *Composite* qui servent à mettre en parallèle les comportements des différents processus primitifs. D’autres types de processus vont permettre une modélisation plus fine que celle des processus primitifs. Ces processus sont entre autre les processus *Choice* permettant d’avoir un choix entre plusieurs processus alternatifs, les processus *Prefix* qui sont des processus préfixés par une *Action* permettant leur activation. Ces processus *Prefix* forment la base de la conception des processus primitifs et des processus *Choice*. Enfin, de manière à permettre une modélisation de l’arrêt du système, des processus comme *Stop* ou encore *Error* sont définis.

5.3.3 Méta-modèle de la logique LTL

Le méta-modèle de LTL (figure 5.4) va permettre la définition de formules pour la spécification de propriétés et la vérification de modèles FSP.



F . 5.4 – Méta-modèle des formules de la logique temporelle LTL

Vu, comme une syntaxe abstraite, ce méta-modèle est constitué d’un ensemble minimal de concepts à savoir les opérateurs *Or*, *Next*, *Until* et *Not* à partir desquels il sera possible de reconstituer par combinaison d’autres opérateurs définis dans la logique comme l’opérateur *Futur*, l’opérateur *Globaly* ou l’opérateur *Weak*. *AP* (Atomic Proposition) modélise, comme son nom l’indique une proposition atomique. A noter que *True* et *False* ne sont pas indispensables, en effet en considérant que ϕ, ϕ_1 et ϕ_2 sont toutes trois des formules LTL, nous avons les équivalences logiques suivantes :

– $True \equiv \phi \; Or \; Not(\phi)$ et $False \equiv Not(True)$;
– $\phi_1 \; And \; \phi_2 \equiv Not(Not(\phi_1) \; Or \; Not(\phi_2))$;

[1] L’anglais est employé afin de garder cohérent les méta-modèles avec l’algèbre FSP et la logique LTL.

- ϕ_1 *Implies* $\phi_2 \equiv$ *Not*(ϕ_1) *Or* ϕ_2 ;
- ϕ_1 *Equiv* $\phi_2 \equiv$ (ϕ_1 *Implies* ϕ_2) *And* (ϕ_2 *Implies* ϕ_1) ;
- *Futur*(ϕ) $\equiv$ *True Until* ϕ ;
- *Globaly*(ϕ) $\equiv$ *Not*(*Futur*(*Not*(ϕ))) ;
- ϕ_1 *Weak* $\phi_2 \equiv$ (ϕ_1 *Until* ϕ_2) *Or Globaly*(ϕ_1).

5.3.4 Méta-modèle du framework multi-agents PI

PI [Ras06] est un système multi-agents léger dont les entités collaborent pour accomplir une fonction. PI permet la modélisation et l'exécution de systèmes concurrents à dynamiques hybrides, plus précisément, à évènements asynchrones, et à dynamiques continues à temps discret avec une exécution synchrone des agents. Pour cela, il propose un moteur d'exécution dont le rôle est de gérer : l'évolution des différentes sortes d'agents, l'occurrence des actions, l'évaluation de gardes ou encore la synchronisation d'actions partagées par rendez-vous.

Ce framework possède de multiples facettes offrant :

1. Un méta-modèle d'agents PI permettant :
   - la description des agents en fonction de leur nature (continue, discrète, hybride),
   - la description de comportement des agents,
   - la synchronisation entre agents ;
2. Un moteur d'exécution permettant l'utilisation de modèles dynamiques.

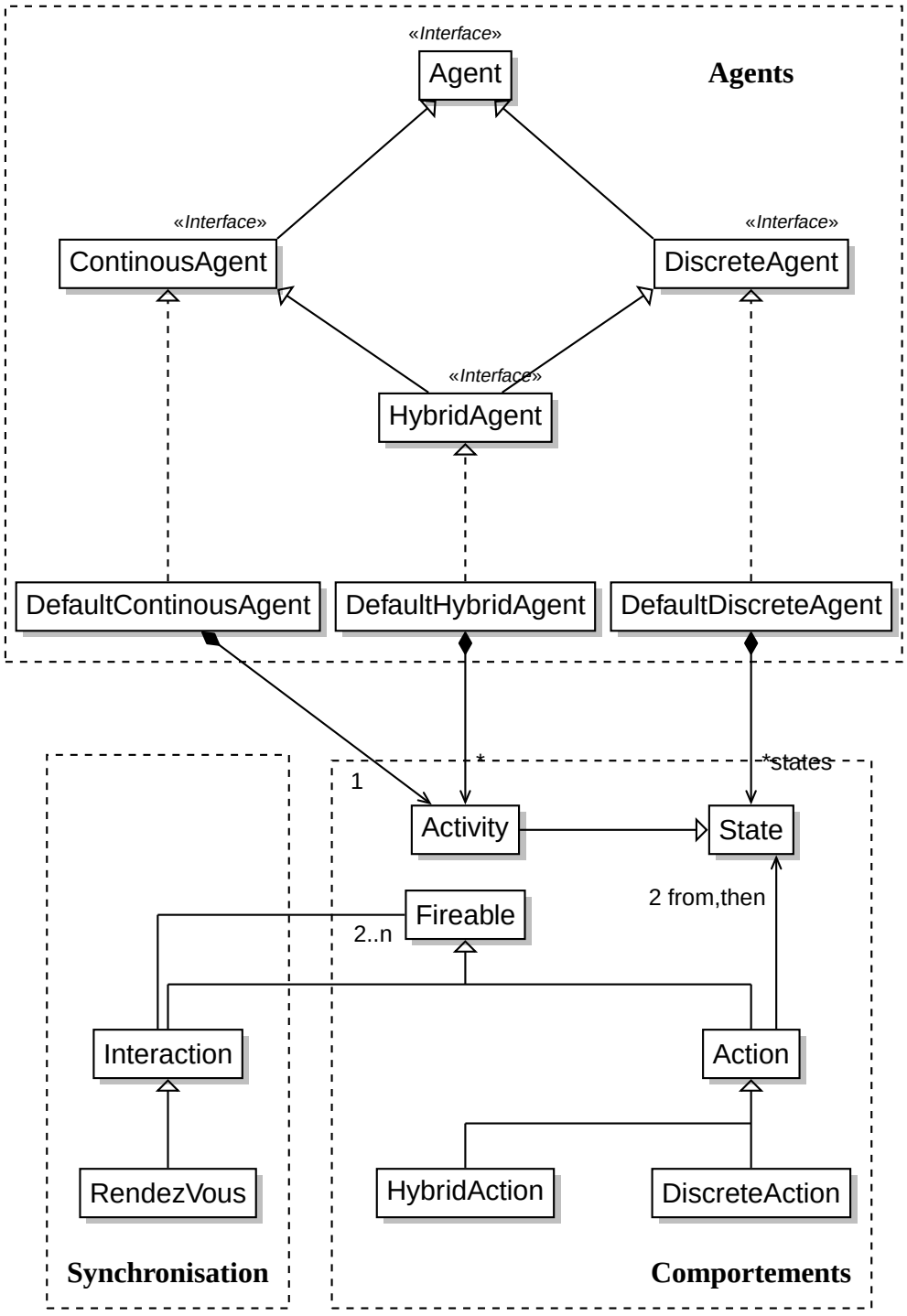
Décrit dans la figure 5.5, le framework peut servir de méta-modèle et fixe un cadre de conception. Il se fonde sur un ensemble de concepts organisés ici en trois groupes principaux : les *agents*, leurs *comportements* (State, Action, Activity, etc.) et leurs *interactions* (RendezVous).

Les agents, dont le type peut être continu à temps discret (*DefaultContinousAgent*), discret (*DefaultDiscreteAgent*) ou hybride (*DefaultHybridAgent*), représentent les entités dynamiques et concurrentes présentes dans le système logiciel modélisé. Le comportement spécifique des agents est modélisé à l'aide des classes : *State*, *Activity*, *Action*, *DiscreteAction* et *HybridAction* permettant de définir des agents dont le comportement est continu, discret ou hybride.

Un modèle d'interaction permet aux agents de communiquer par objets partagés et/ou en se synchronisant afin d'effectuer des actions partagées (*RendezVous*). L'exécution des agents suit alors un modèle d'exécution conforme à leur nature en prenant en compte les différents aspects de leur dynamique.

5.3.5 Méta-modèle pour des applications J2ME

De façon à permettre l'implantation d'un modèle d'application sur une plateforme mobile, il est nécessaire de définir un méta-modèle permettant de servir d'interface entre le logiciel souhaité (et les fonctionnalités réalisées) avec la plateforme J2ME. Ce méta-modèle



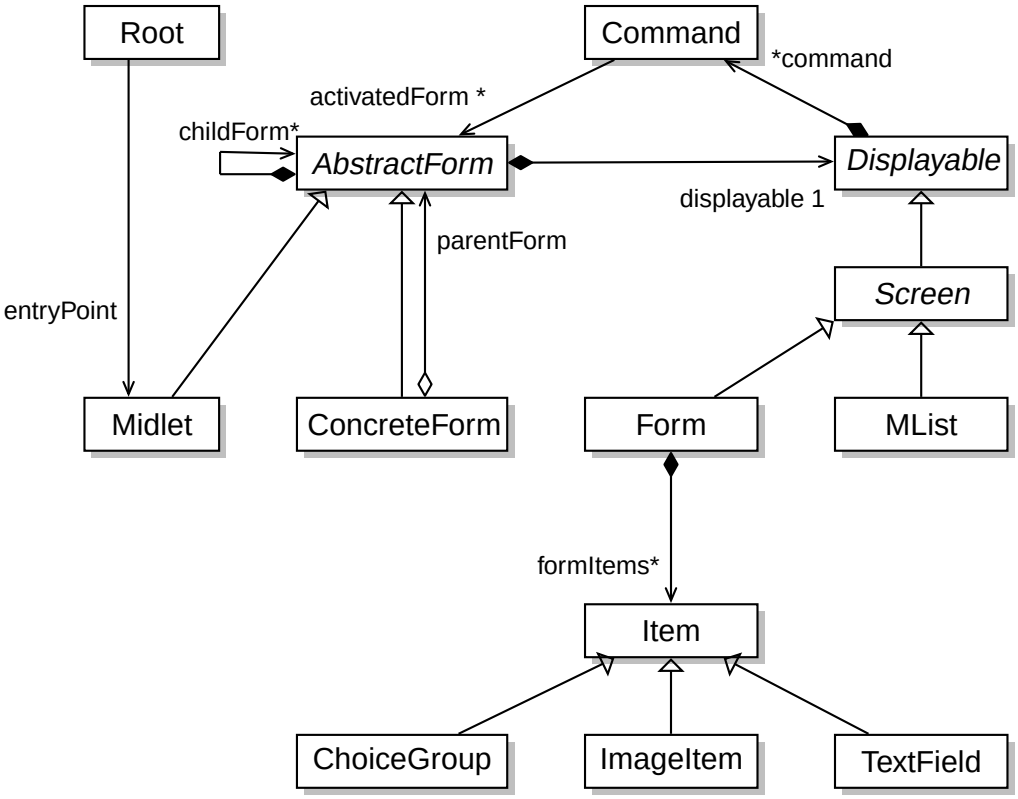
F . 5.5 – Méta-Modèle de la plateforme PI

fournit le support pour la construction d'un modèle d'interface de communication avec l'utilisateur intégrant (dans le sens MVC) des modèles, des vues pour ces modèles et des contrôleurs pour gérer l'évolution de l'application.

Ce méta-modèle (figure 5.6) s'articule autour du concept de *Midlet* qui définit le point d'entrée d'un modèle d'application mobile. Cette *Midlet*, héritant de *AbstractForm* se compose d'un ensemble de *AbstractForm* sous la forme de *ConcreteForm* formant ainsi un arbre (le modèle de l'application). Chaque *AbstractForm* (classe qui généralise les classes *Midlet* et *ConcreteForm*) de l'arbre se compose ensuite d'un *Displayable* offrant les différentes vues possibles de l'application.

Ainsi, les *AbstractForm* constituent le support à la description du comportement de l'application concrètement implanté dans les *ConcreteForm*. Les *Displayable* permettent d'en donner une représentation graphique. Pour cela, différents *Displayable* peuvent être utilisés :

- La classe *MList* permet une représentation simple des choix possibles offerts par une application.
- La classe *Form* permet de fournir des moyens évolués pour représenter les modèles contenus dans les *ConcreteForm*. Cette classe est composée d'éléments nommés *Item* permettant d'utiliser par exemple des formulaires à choix unique (*ChoiceGroup*), des images (*ImageItem*), ou encore des champs texte (*TextField*).



F . 5.6 – Méta-Modèle d'application mobile pour la plateforme J2ME.

Un *Displayable* est donc une abstraction qui sert de pont entre l’application et l’utilisateur. Elle permet de représenter des objets graphiques et se constituera globalement par des *MLists* pour les nœuds de l’arbre de *ConcreteForm* et de *Forms* pour les feuilles de ce même arbre. On notera à ce titre que les *ConcreteForms* situés en tant que feuilles de l’application seront les éléments qui intégreront des modèles métiers propre au contexte de l’application (par exemple des modèles d’automates).

Les *Displayable* intègrent également un certain nombre de *Command* pour réagir aux sollicitations de l’utilisateur. Les *Command* permettent alors le changement d’état de l’application soit en changeant l’*AbstractForm*/*ConcreteForm* courante, soit en modifiant son état.

5.4 Élaboration des transformations de modèles

A ce stade et en suivant la démarche prescrite par le méta-processus, il faut définir les transformations de modèles qui vont permettre d’établir des passerelles entre les langages de modélisation et les outils.

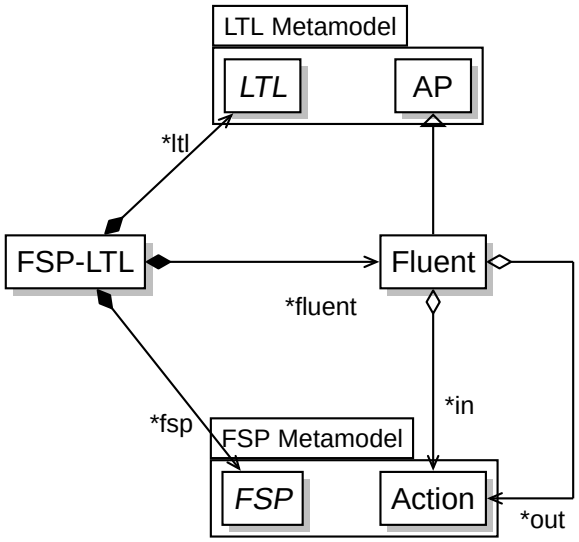
De façon à identifier quelles transformations devront être définies, différents étapes peuvent être suivies : 1) Il faut définir les transformations permettant les différents changements possibles d’activités (par exemple, une transformation entre les automates destinés à la commande par supervision et FSP pour la vérification de modèles), 2) il faut définir les transformations entre les méta-modèles et les formats acceptés/générés par les outils (par exemple entre FSP-LTL et LTSA), et enfin, 3) il faut définir les transformations permettant l’implantation des modèles sur des plateformes d’exécutions potentielles (cf. génération de code par exemple entre FSM et le framework PI).

Ces critères ne sont pas exhaustifs, il est possible selon les besoins du domaine d’avoir d’autres critères pour identifier les transformations à réaliser. Entre autre, une composition de modèles peut être nécessaire pour permettre la définition de modèles comportant des concepts issus de méta-modèles différents. Par exemple, l’outil LTSA, exploite conjointement des modèles LTL et des modèles FSP. Afin de réaliser des modèles cohérents pour cet outil, il est avantageux d’effectuer une composition des deux méta-modèles concernés.

5.4.1 Composition de FSP et de LTL

Afin de concevoir des modèles pour le Model-Checking intégrant à la fois des modèles du système et des modèles de propriétés, il est nécessaire de concevoir un nouveau méta-modèle. Ce méta-modèle basé sur l’association du méta-modèle FSP et du méta-modèle LTL va tisser un lien entre ces derniers. Pour ce faire, il est nécessaire de définir quels éléments (*points pivots*) sont combinés pour que les modèles FSP et LTL puissent interagir.

Dans le cas présent, FSP est une algèbre permettant la modélisation de LTS (Labelled Transition Systems) ne comportant pas de propriétés sur les états. Or les activités de Model-



F . 5.7 – Méta-Modèle de composition des méta-modèles FSP et LTL

Checking se basent souvent sur ces propriétés d'états. Le concept de *Fluent*, [GM03], a été introduit de façon à faciliter l'expression de propriétés sur les états des modèles réalisés. Ainsi, le lien se fait sur la notion d'*Action* dans le méta-modèle FSP et sur la notion de Proposition Atomique (*AP*) dans LTL. Ces deux concepts vont être combinés dans un nouveau concept, celui de *Fluent* héritant du concept de proposition atomique *AP* issu de LTL et intégrant par agrégation le concept d'*Action* issu de FSP. Ce méta-modèle de composition (figure 5.7) va permettre de réaliser des modèles intégrant les concepts de FSP et ceux de LTL, et de faciliter les activités de Model-Checking. Un *Fluent* se construit alors avec une valeur initiale vraie ou fausse, un ensemble d'actions d'entrées et un ensemble d'actions de sorties le définissant. Le *Fluent* changera d'état selon l'occurrence d'une action d'entrée et/ou de sortie.

5.4.2 Transformation de FSM vers FSP

Si on souhaite combiner une approche de type commande par supervision exploitant des automates (FSM) et une approche de type Model-Checking exploitant ici FSP et LTL, des transformations entre ces deux domaines doivent être envisagées. Cette transformation est réalisée en deux étapes, une première étape consiste à définir une transformation entre le méta-modèle FSM et celui de FSP (présentée ci-dessous) ; une seconde étape consiste à générer, par une transformation vers une syntaxe concrète le code FSP correspondant (chapitre 5.4.4). La première étape est réalisée avec le langage de transformation ATL [JK05].

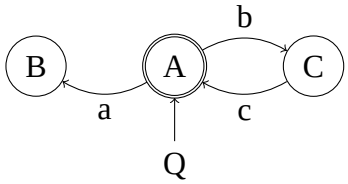
Un modèle partiel de cette transformation est présenté à la figure 5.8. En accord avec les principes énoncés dans le chapitre 3, ce modèle décrit les éléments essentiels nécessaires à la transformation entre les deux méta-modèles. Dans cette transformation, sont définis le méta-modèle d'entrée (*Fsm*) et le méta-modèle de sortie (*Fsp*) à la suite de quoi un certain nombre de fonctions *helper* peuvent être définies afin de fournir des primitives facilitant

```
module Fsm2Fsp;     -- Module Template
create OUT : Fsp from IN : Fsm;

helper context Fsm!StateSet def : getInitialState() :
  Fsm!State = self.states->any( i | i.isInitial = true );
helper context Fsm!State def : isMultipleEvent() :
  Boolean = if (self.possibleEvents.size()>1) then true else false endif;
helper context Fsm!State def : isSingleEvent() :
  Boolean = if (self.possibleEvents.size()=1) then true else false endif;

rule state2PrefixUnique{from
  s:Fsm!State(s.isSingleEvent()) to o:Fsp!Prefix(
    nameProcess <- s.stateName,
    prefix<-s.getOutputTransitions().first().event,
    then<-s.getOutputTransitions().first().dst)
}
rule state2ChoiceMultiple{from
  s:Fsm!State(s.isMultipleEvent()) to o:Fsp!Choice(
    nameProcess <- s.stateName,
    choice <- s.getOutputTransitions())
}
rule transition2PrefixMultiple{from
  s:Fsm!Transition(s.isMultiple()) to o:Fsp!Prefix(
    nameProcess <-s.dst.stateName + s.event.eventName,
    prefix <- s.event,
    then <- s.dst)
}
```

F . 5.8 – Modèle partiel de transformation ATL entre les automates et FSP

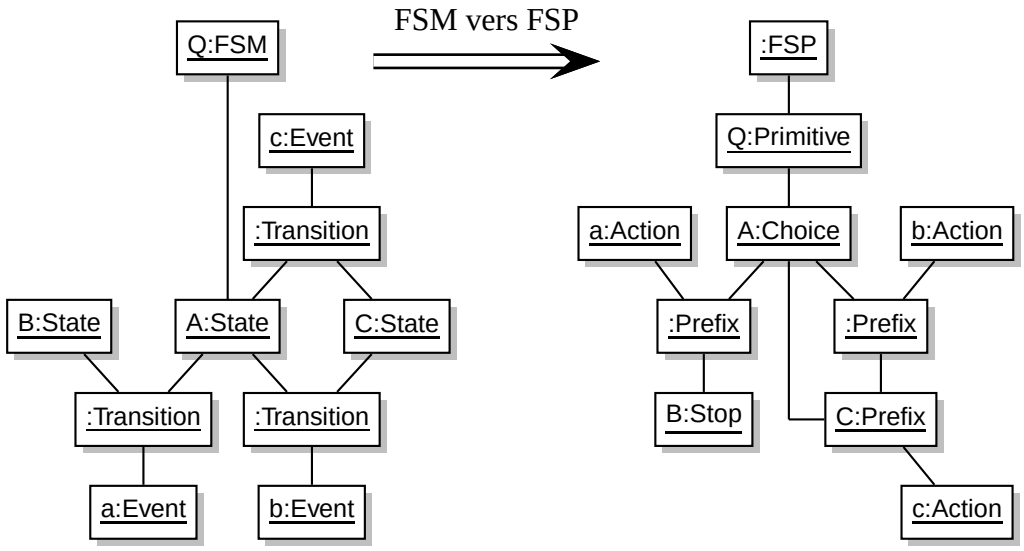
la construction des règles de transformation. Ensuite viennent les règles (*rule*) qui rendent opérationnelle la transformation.

Ici, les *helper* permettent de définir si un état comporte aucune, une ou plusieurs transitions sortantes. Les règles vont alors construire les différents types de *Process* FSP en fonction de la configuration du graphe de l’automate dans un état donné. Ainsi, un état proposant plusieurs transitions de sortie sera transformé en un processus *Choice* faisant un lien avec des processus de type *Prefix* modélisant finalement l’une des transitions sortantes de l’état.



F . 5.9 – Automate Q représenté dans sa syntaxe concrète

En appliquant la transformation ATL, pour le modèle d’automate Q (figure 5.9), cinq processus sont produits. A partir de l’état initial de l’automate, un premier processus *Primitif* est défini (Q pointant sur le processus A). Le processus Q pointe sur un processus *Choice* proposant la possibilité de choisir entre deux processus *Prefix* préfixés respectivement par les actions *a* et *b*. Le processus préfixé par l’action *a* mène à un processus *Stop* tandis que celui préfixé par l’action *b* mène à un autre processus *Prefix* (préfixé par l’action *c*) qui lui-même reboucle sur le processus *Choice* initial.



F . 5.10 – Exemple d'application de la transformation d'un automate Q (FSM) en modèle FSP

5.4.3 Génération de code pour Supremica à partir de FSM

Les transformations syntaxiques de modèles abstraits vers des représentations textuelles offrent la possibilité de construire un pont entre un méta-modèle et un outil spécifique du domaine exploitant une syntaxe concrète et n'offrant pas la possibilité d'importer un modèle abstrait. Dans le cas des automates à états finis utilisés dans la commande par supervision, l'outil identifié est Supremica [AFF03]. Supremica est un outil qui permet de modéliser des automates à états finis donc de définir des systèmes et des spécifications ainsi que de synthétiser des superviseurs. Il permet de vérifier la commandabilité d'un système et sa vivacité. L'outil offre également des possibilités de simulation et propose différents types de génération de code tel que, par exemple, vers le langage Java ou Dot (pour la visualisation des automates).

Pour définir une transformation de syntaxe, il est nécessaire d'identifier préalablement quelle syntaxe sera utilisée. Dans Supremica, la sauvegarde des modèles s'effectue dans des fichiers XML, ce dernier pouvant alors être considéré comme une syntaxe concrète pour la sérialisation des modèles. Nous proposons alors d'utiliser cette structure pour construire la transformation de syntaxe définie ici avec l'outil IDM Sintaks, [MFF06]. Sintaks est un outil qui permet de définir des transformations bidirectionnelles entre des méta-modèles et des syntaxes concrètes textuelles. Il est basé sur un méta-modèle qui permet de définir des modèles de transformation qui vont contenir à la fois la syntaxe abstraite et la syntaxe concrète du langage ciblé. Les éléments de la grammaire sont par ailleurs associés aux concepts du méta-modèle pour lequel la transformation est définie.

La figure 5.11 donne le modèle de transformation entre le méta-modèle des automates à états finis et la syntaxe du format de fichier importable par Supremica. Cette transformation exploite les *Template* Sintaks pour associer un pattern d'écriture (ici de l'XML formaté pour Supremica) aux concepts de base du méta-modèle des automates, à savoir les concepts d'*état*, d'*évènement*, et de *transition*.

```
Template printState(s)      = "<State name=" ++ s.name ++ " />"
Template printEvent(e)      = "<Event label=" ++ e.name ++ " />"

Template printTransition(t) = "<Transition  source=" ++ t.source ++ " dest=" ++ t.dest
                                                    ++ " event="++t.event++" />"

Template printStateSet(s)  = "<States>"++ forAll state in s do (++,printState(state))
                                    ++ "</States>"

Template printAlphabet(e)  = "<Events>"++ forAll event in e do (++,printEvent(event))
                                    ++ "</Events>"

Template printTransFunc(t) = "<Transitions>" ++ forAll trans in t do
                                               (++,printTransition(trans))
                                          ++ "</Transitions>"

Template printFsm(fsm) = "<Automaton>" ++ printAlphabet(fsm.alphabet)
                                    ++ printStateSet(fsm.stateSet)
                                    ++ printTransFunc(fsm.transFunc)
                                    ++ </Automaton>"
```

F . 5.11 – Modèle de transformation syntaxique des automates vers Supremica

5.4.4 Génération de code de FSP-LTL vers LTSA

Afin de pouvoir combiner des approches de type commande par supervision et de type Model-Checking, une transformation M2C entre le méta-modèle FSP-LTL et l’outil LTSA est définie de la même façon que pour la transformation syntaxique entre le méta-modèle des automates et l’outil Supremica (avec Sintaks, figure 5.12).

```
Template printAction(a)        = a.name
Template printPartialPrefix(p) = printAction(p.action) ++ -> p.targetProcessName
Template printPrefix(p)        = p.name ++ "=(" ++ printPartialPrefix(p) ++ ")"
Template printErrorProcess(p)  = p.name ++ "= Error"
Template printChoice(choice)   = p.name ++ "=(" ++ forAll p in choice.list do
                                                    (++,"|",printPartialPrefix(p))
                                        ++ ")"

Template print(p)              = case p of
                                      Prefix       -> printPrefix(p)
                                      Choice       -> printChoice(p)
                                      ErrorProcess -> printErrorProcess(p)

Template printModelProcess(fspModel) = fspModel.name ++ "=" ++ fspModel.firstProcessName ++ ","
                                                    ++ forAll p in fspModel.list do
                                                         (++,",\n",print(p))
                                                    ++ "."

Template printComposedProcess(fspModel) = "||" ++ fspModel.name ++ "=("
                                               ++ forAll p in fspModel.list do
                                                    (++,"||",p.name)
                                               ++ ")."
```

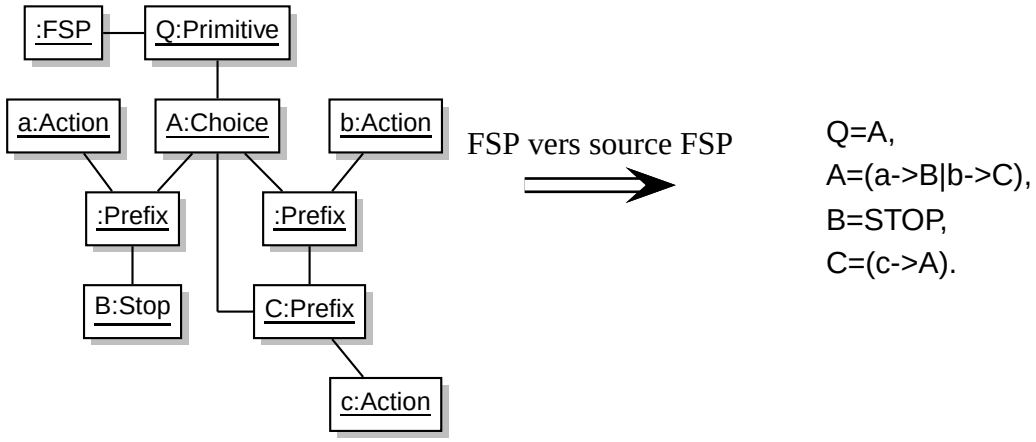
F . 5.12 – Modèle partiel de transformation syntaxique FSP-LTL vers LTSA

LTSA [MK06] est un outil de modélisation et de vérification de systèmes concurrents. Construit sur la base de l’algèbre de processus FSP, il fournit un moyen pratique de tester par simulation ou de vérifier à l’aide de formules LTL le comportement de systèmes. Comme pour Supremica, afin de définir la transformation FSP-LTL vers LTSA, il est né-

cessaire d'identifier quelle syntaxe concrète sera employée pour générer des modèles compatibles avec l'outil.

Le plus simple est de s'appuyer sur la grammaire du langage FSP. Effectué également avec l'outil Sintaks, ce modèle de transformation (présenté partiellement avec la figure 5.12) va définir différents *Template* permettant de transformer chacun des types d'éléments du modèle abstrait en sa représentation concrète FSP équivalente.

Ainsi, en reconsidérant le résultat de l'exemple traité précédemment de la transformation FSM vers FSP (figure 5.10), le modèle obtenu peut être transformé en source FSP afin d'être utilisé dans LTSA. La figure 5.13 donne le résultat d'une traduction obtenue par application des différents *Template*.



F . 5.13 – Application de la transformation d'un modèle FSP en source FSP

5.4.5 Génération de code de FSM vers DestKit

Le framework DestKit [Col05] est une implantation du méta-modèle des FSM en Java afin de faciliter l'utilisation des modèles d'automates à états finis sur une plateforme spécifique. Ici, il est possible de générer directement le source Java à partir des FSM car le framework DestKit intègre déjà ce méta-modèle. Dans ce contexte, l'utilisation de Sintaks est similaire aux deux paragraphes précédents à la différence qu'il ne s'agit pas ici de cibler la syntaxe concrète d'un outil mais celle du langage dans lequel est défini le framework, ici Java.

Cette transformation illustrée par la figure 5.14 possède sensiblement la même structure et est comparable à la transformation vers Supremica. Cela s'explique simplement par le fait que ces deux transformations s'appuient sur le même méta-modèle source, celui des automates. Par contre la différence réside dans les patterns d'écriture qui génèrent ici du code source Java conforme à l'architecture du framework DestKit.

```
Template printState(s)      = "fsm.addState(new State("++ s.name ++"));"

Template printEvent(e)      = "fsm.addEvent(new Event("++ e.label ++"));"

Template printTransition(t) = "fsm.addTransition("++ t.src ++","++ t.event ++","++ t.dst ++");"

Template printStateSet(s)  = forAll state in s do (++,printState(state))

Template printAlphabet(e)  = forAll event in e do (++,printEvent(event))

Template printTransFunc(t) = forAll trans in t do (++,printTransition(trans))

Template printFsm(fsm) = "FiniteStateMachine fsm=new FiniteStateMachine("++fsm.name++");"
                          ++ printAlphabet(fsm.alphabet)
                          ++ printStateSet(fsm.stateSet)
                          ++ printTransFunc(fsm.transFunc)
```

F . 5.14 – Modèle de transformation des automates vers Java pour la plateforme DestKit

5.4.6 Génération de code de FSM vers PI

De façon à permettre une implantation sur la plateforme PI, une transformation de modèles (M2C), définie entre le méta-modèle des automates à états finis et le méta-modèle PI doit être mise en place. Pour cela, une transformation de type génération de code est entreprise en utilisant cette fois ci le langage Xpand (paragraphe 3.5) qui est un langage spécialisé dans la génération de code pour des modèles EMF sous Eclipse [BSM03].

Le choix d’utiliser ici Xpand au lieu de Sintaks se justifie par la nature de la transformation qui devrait, comme dans le cas de la transformation des FSM vers FSP (et la syntaxe de LTSA), se décomposer en une succession de deux transformations : une transformation du méta-modèle FSM vers le méta-modèle PI et une transformation du méta-modèle PI vers du source java conforme à l’implantation d’agents PI. De plus, la transformation recherchée doit permettre l’implantation des modèles, elle ne nécessite donc pas d’être bidirectionnelle comme le permet Sintaks. Ici, Xpand permet donc de s’affranchir d’une certaine complexité inhérente à la définition de transformations avec Sintaks, et permet de traduire directement les modèles d’automates vers du code Java respectant l’architecture de PI.

Avant d’établir cette transformation, deux classes Java sont à réaliser afin de permettre une mise en relation cohérente avec la plateforme PI : la première, *RandWState* héritant de la classe *State* de la plateforme, la seconde, *RandWTransition* héritant pour sa part de la classe abstraite *DiscreteAction*. Ces deux classes vont en fait rendre compte au sein de la plate forme PI des spécificités de la théorie de la commande par supervision [RW89].

Une transformation Xpand se définit en précisant dans un premier temps le méta-modèle source avec le mot clef *IMPORT*. Une règle est introduite par le mot clef *DEFINE* en précisant le type d’objet sur lequel la transformation va s’appliquer. Elle se termine par *ENDDEFINE*. L’appel à d’autres règles déjà définies s’effectue par le mot clef *EXPAND* suivi du nom de la règle cible. Enfin toutes les expressions du langage Xpand doivent être exprimées entre « et » afin de les distinguer du texte brut à placer directement dans le document cible.

```
«IMPORT Fsm»
«DEFINE root FOR Root»
«DEFINE rule_Fsm FOR Fsm»
«FILE this.fsmName+".java"»

public class «this.fsmName» extends DefaultDiscreteAgent{
«EXPAND rule_StateSetListAttribute FOR this.stateSet»
«EXPAND rule_TransitionFunctionListAttribute FOR this.transitionFunction»

public «this.fsmName»() {
«EXPAND rule_StateSetListInstance FOR this.stateSet»
«EXPAND rule_TransitionFunctionListInstance FOR this.transitionFunction»}}
«ENDFILE»«ENDDEFINE»

«DEFINE rule_StateAttibute FOR State»
private RandWState «this.stateName»;
«ENDDEFINE»

«DEFINE rule_TransitionAttibute FOR Transition»
private RandWTransition «this.srcStateName»«this.eventName»«this.dstStateName»;
«ENDDEFINE»

«DEFINE rule_StateInstance FOR State»
this.«this.stateName»=new RandWState("«this.stateName»");
«ENDDEFINE»

«DEFINE rule_TransitionInstance FOR Transition»
this.«this.srcStateName»«this.eventName»«this.dstStateName»
   =new RandWTransition(this,«this.srcStateName»,«this.eventName»,«this.dstStateName»);
«ENDDEFINE»
```

F . 5.15 – Modèle partiel de la transformation FSM vers PI

Pour définir la transformation, chacun des modèles d'automate est transformé en agent héritant de la classe *DefaultDiscreteAgent*. Les états et les transitions deviennent des attributs d'instances de *RandWState* et *RandWTransition*.

5.4.7 Génération de code de l'application J2ME

L'implantation d'une partie applicative gérant l'interface Homme-Machine sur une plateforme mobile est possible en partant d'un modèle d'application J2ME et en le traduisant en code source Java à l'aide de l'outil Xpand.

Le principe utilisé est le même que pour la génération de code pour la plateforme PI vue au paragraphe précédent et est illustré par la figure 5.16. Cette transformation permet de construire automatiquement un ensemble de classes conformes au méta-modèle J2ME (figure 5.6). Partant de l'élèment racine (instance de *Midlet* du méta-modèle J2ME) d'un modèle source, le point d'entrée de l'application est obtenu par dérivation de la classe *Midlet* de l'API J2ME. Puis, l'ensemble des élèments du modèle issus des classes dérivées de *AbstractForm* dans le méta-modèle, vont servir à définir des classes qui réalisent l'interface *CommandListener* de l'API. Ces classes intégrent en données membres des objets *Displayable* pour gérer l'interface graphique et des objets *Command* pour gérer les intéractions utilisateurs.

```
«IMPORT NM_Midlet»
«DEFINE root FOR Midlet»
«FILE this.formTypeName+".java"»

public class «this.formTypeName» extends MIDlet implements CommandListener {
«EXPAND rule_DisplayableCommandListAttribute FOR this.displayable»
«EXPAND rule_ConcreteFormChildAttribute FOREACH this.childForm»
«EXPAND rule_DisplayableAttribute FOR this.displayable»

public «this.formTypeName»() {
«EXPAND rule_DisplayableCommandListAddMethod FOR this.displayable»
«this.displayable.displayableName».setCommandListener(this);
this.show();}

public void commandAction(Command command, Displayable displayable) {
«EXPAND rule_DisplayableCommandListMethodSem FOR this.displayable»;}

public void show() {
System.out.println("Midlet run...");
Display.getDisplay(this).setCurrent(«this.displayable.displayableName»);}
«ENDFILE»
«ENDDEFINE»

DEFINE rule_DisplayableCommandListAttribute FOR Displayable»
«EXPAND rule_CommandAttribute FOREACH this.commandList»
«ENDDEFINE»

«DEFINE rule_CommandAttribute FOR Command»
private final Command «this.commandType»_CMD = new Command("«this.commandName»",
      Command.«this.commandType»,«this.priorityCommand»);
«ENDDEFINE»
```

F . 5.16 – Modèle de transformation Xpand pour la génération de code J2ME

5.5 Synthèse : Processus spécifique aux SED

Les éléments qui précèdent permettent l’élaboration d’un modèle de processus de développement intégrant les différents produits identifiés ou élaborés dans les activités précédentes (langages de modélisation et méta-modèles, transformations et outils). Ce modèle va se baser sur le cycle de vie proposé à la figure 5.1 qui a permis d’identifier les activités qui seront à intégrer, il permettra de :

1. Définir les besoins en utilisant des diagrammes UML (diagrammes de classes ou encore diagrammes de cas d’utilisation).
2. Effectuer la conception en se basant sur UML et les Systèmes à Évènements Discrets pour les domaines de la commande par supervision et du Model-Checking afin de modéliser les entités logicielles et de permettre une synthèse de modèles et/ou la vérification des modèles.
3. Effectuer une implantation en réalisant la génération de code vers les plateformes spécifiques préalablement identifiées.

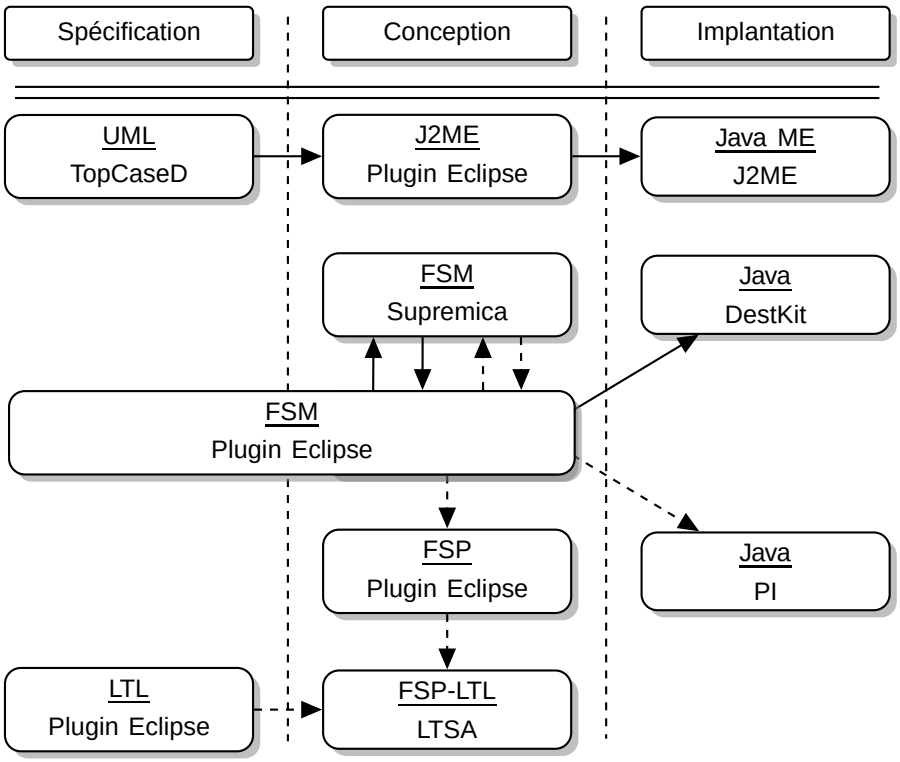
Pour construire ce modèle de processus, il est utile dans un premier temps de classer les langages de modélisation (ou leurs méta-modèles) et les outils selon leur emploi dans les différentes phases du développement.

Lors de la phase de spécification, les concepteurs pourront utiliser différents diagrammes UML pour décrire l’architecture, et s’appuyer sur le méta-modèle des automates à états fi-

nis (FSM) ou celui de la logique temporelle LTL pour spécifier des comportements. De même, lors de la phase de conception, les automates à états finis, l'algèbre de processus FSP et le méta-modèles d'application J2ME seront utilisés. Pour la phase d'implantation, différentes plateformes et frameworks tels que J2ME, DestKit ou PI pourront servir de cible à la génération. Enfin pour certains langages de modélisation, des outils spécifiques ont été identifiés. Entre autres, les outils Supremica et LTSA pourront être utilisés pour effectuer des tâches particulières relevant des domaines de la commande par supervision et/ou de la vérification.

De façon à pouvoir utiliser ces langages au sein d'un processus de développement spécifique, il faut mettre en place les différentes transformations qui ont été définies. Pour cela, il est nécessaire de considérer les besoins imposés par les développements à effectuer. Par exemple, nous avons déjà énoncé l'intérêt d'utiliser la théorie de la commande par supervision combinée à une implantation sur une plateforme J2ME avec DestKit. Enfin pour d'autres types d'applications, en plus de la théorie de la commande par supervision, il pourra être nécessaire d'effectuer une phase de vérification de modèles suivie par une l'implantation à l'aide du framework PI, etc.

La classification des langages et outils ainsi que la mise en place des transformations pour definir un processus de développement sont présentés par la figure 5.17. On notera que l'utilisation des plugins Eclipse permet en intégrant un méta-modèle de fournir un outil de modèlisation mais aussi de fournir un point pivot entre différentes transformations et activités.



F . 5.17 – Utilisation des transformations de modèles pour définir le modèle du processus de développement

Au vue des besoins spécifiques des SED, il est alors possible de définir pour chaque classe d’applications un chemin à suivre par le processus de développement en se basant sur les transformations définies précédemment (figure 5.17). Un premier chemin (illustré par les flèches pleines) peut être défini pour la mise en œuvre d’applications décentralisées mobiles supportées par la plateforme *J2ME* (figure 5.17). Un second chemin propose le cas d’applications composées d’entités autonomes (Agents) où une synthèse de superviseur et une vérification par Model-Checking sont nécessaires (illustré par les flèches en pointillées).

5.6 Conclusion

Ce chapitre a décrit une mise en œuvre des principes définis dans le chapitre 4. En se limitant au contexte du développement de logiciel pour les SED et après avoir identifié les concepts de ce domaine, les outils et les plateformes associés, des activités types et différents langages de modélisation ont été identifiés. Des méta-modèles et des transformations ont été élaborées pour ces langages et les outils qui les manipulent.

Les transformations n’ont pas toutes été définies avec les mêmes outils. ATL [JK05], Sintaks [MFF06] et Xpand [EFH07] ont été utilisés mais d’autres auraient pu être employés (comme Kermeta [MFJ05] par exemple). On peut noter que selon le type de transformation ou selon le contexte dans lequel la transformation doit être définie, le choix de l’outil s’est imposé de lui-même. Pour les transformations de modèles abstraits à modèles abstraits (M2M), le choix s’est porté sur ATL (Sintaks et Xpand n’étant pas adapté à ce type de transformation), qui fournit un langage de transformation simple basé sur des requêtes. Dans ce cas, Kermeta aurait pu être utilisé mais ATL permet l’utilisation directe des modèles et méta-modèles construits avec EMF alors que Kermeta définit son propre langage de modélisation et impose donc des transformations supplémentaires pour l’import ou l’export des modèles EMF : l’utilisation d’ATL est par conséquent plus simple et plus adapté.

Pour les transformations vers des syntaxes concrètes textuelles, ce travail a utilisé Sintaks et Xpand. Sintaks a été utilisé pour interconnecter des outils spécifiques car il permet la définition de transformations bidirectionnelles entre syntaxe concrète et abstraite. Xpand, a été utilisé pour la génération de code source où il s’est avéré bien plus simple à utiliser que Sintaks ; aucun aspect bidirectionnel étant nécessaire. Il permet de plus de projeter un modèle conforme à un méta-modèle vers une syntaxe concrète associée à un autre méta-modèle (comme dans le cas de la transformation des FSM vers PI) permettant ainsi de fournir en une transformation, une transformation équivalente à deux transformations combinant ATL et Sintaks.

L’ensemble des réalisations conduit à un modèle de processus de développement dédié aux SED (figure 5.17). Ce processus présente l’avantage d’intégrer des outils du domaine. Les efforts induits par la définition des méta-modèles et des transformations sont alors compensés par l’inter-opérabilité des outils résultant de cette intégration. Pour montrer les bénéfices de l’approche proposée, le chapitre 6 propose une utilisation du modèle de processus obtenu sur deux exemples de systèmes.

Illustrations

Celui qui donne un bon conseil, construit d'une main, celui qui conseille et donne l'exemple, à deux mains ; mais celui qui donne de bonnes leçons et un mauvais exemple construit d'une main et détruit de l'autre.
(Francis Bacon)

La folie est de toujours se comporter de la même manière et de s'attendre à un résultat différent.

(Albert Einstein)

Chapitre 6

Application du processus

Les méthodes, transformations et outils spécifiques identifiés au chapitre 5 vont être utilisés pour la réalisation de deux applications relevant du domaine des SED.

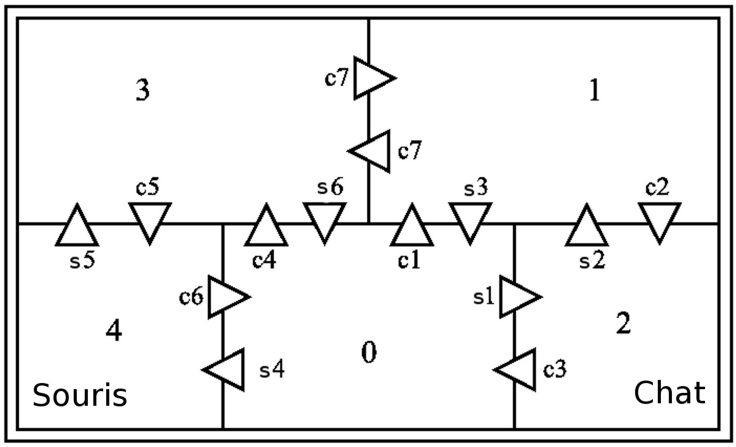
La première application est une illustration de l'utilisation de la commande par supervision sur une plate-forme mobile J2ME. La deuxième application illustre une approche combinant synthèse de superviseur et vérification de modèles pour un ciblage sur une plate-forme spécifique.

6.1 Illustration I : Développement d'une application décentralisée mobile

6.1.1 Problème du chat et de la souris

Le problème du chat et de la souris est un cas d'étude classique d'exclusion mutuelle, utilisé dans le domaine de la commande par supervision initiée par P.J.Ramadge et W.M.Wonham, [RW89]. Dans ce problème, une souris et un chat partagent un ensemble de pièces communicant les unes avec les autres par le biais de portes utilisables dans un seul sens et préfixées de la lettre 'c' et 's', les premières étant exclusivement empruntées par le chat et les secondes par la souris (figure 6.1). Cet exemple a pour intérêt d'être un problème simple mais utilisant suffisamment de concepts de la théorie de la commande par supervision pour en montrer les fondements.

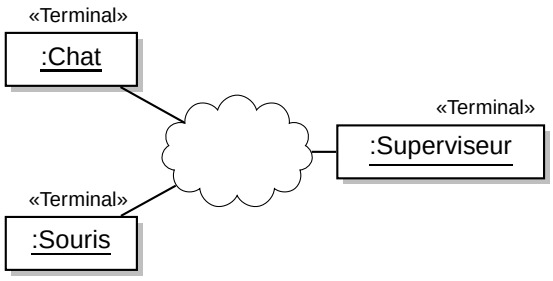
La souris peut emprunter les portes s_1 à s_6 pour se déplacer dans l'ensemble des pièces et le chat peut emprunter les portes c_1 à c_7. Le problème posé consiste à offrir un maximum de liberté de mouvement au chat et à la souris tout en garantissant qu'ils ne se rencontrent jamais dans la même pièce. Pour cela, les portes peuvent être ouvertes ou fermées par un superviseur. Une contrainte supplémentaire est que la porte c_7 ne peut être commandée par le superviseur, il s'agit d'un évènement non-commandable dans la terminologie de la



F . 6.1 – Problème du chat et de la souris

théorie de la supervision, il sera noté $!c_7$. Une solution à ce problème consiste à obtenir le modèle du superviseur, par synthèse, à partir du modèle du système "libre" et du modèle des spécifications.

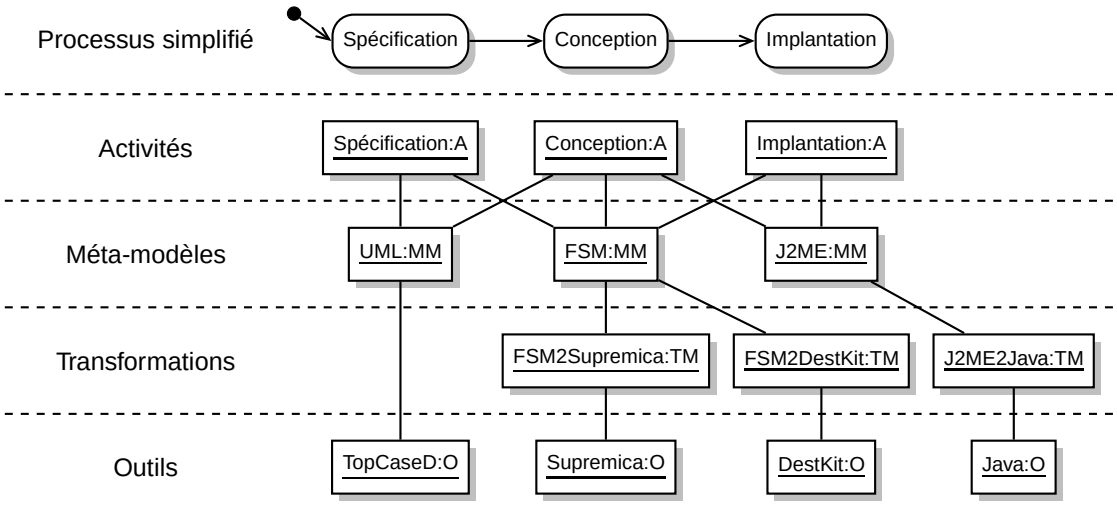
A ce stade, seul le cœur du problème a été décrit, il ne constitue en soit que l'aspect métier de l'application sans la prise en compte d'un éventuel contexte d'implantation ou d'utilisation. De façon à situer ce problème dans un contexte spécifique, nous allons prendre en compte l'ajout de paramètres supplémentaires.



F . 6.2 – Contexte de mise en œuvre de l'application

Le problème sera considéré comme un jeu à deux acteurs jouant respectivement les rôles du chat et de la souris. Ces acteurs résideront sur des terminaux mobiles et feront appel à un serveur jouant le rôle du superviseur afin de déterminer si une porte pourra être franchie (figure 6.2). Ce superviseur garantira le fait que le chat ne puisse occuper la même pièce que la souris au même instant.

Ainsi, ce problème se pose désormais dans le contexte d'une application décentralisée mobile. Ici, à cause du passage à un contexte réel, le problème du chat et de la souris n'est plus juste un problème de supervision. Il sera nécessaire de prendre en compte, les aspects distribués et mobile du système lors du développement.



F . 6.3 – Présentation partielle des éléments constituant le processus de développement (A : Activité, MM : Méta-modèle, TM : Transformation de modèle, O : Outil)

6.1.2 Mise en œuvre du processus

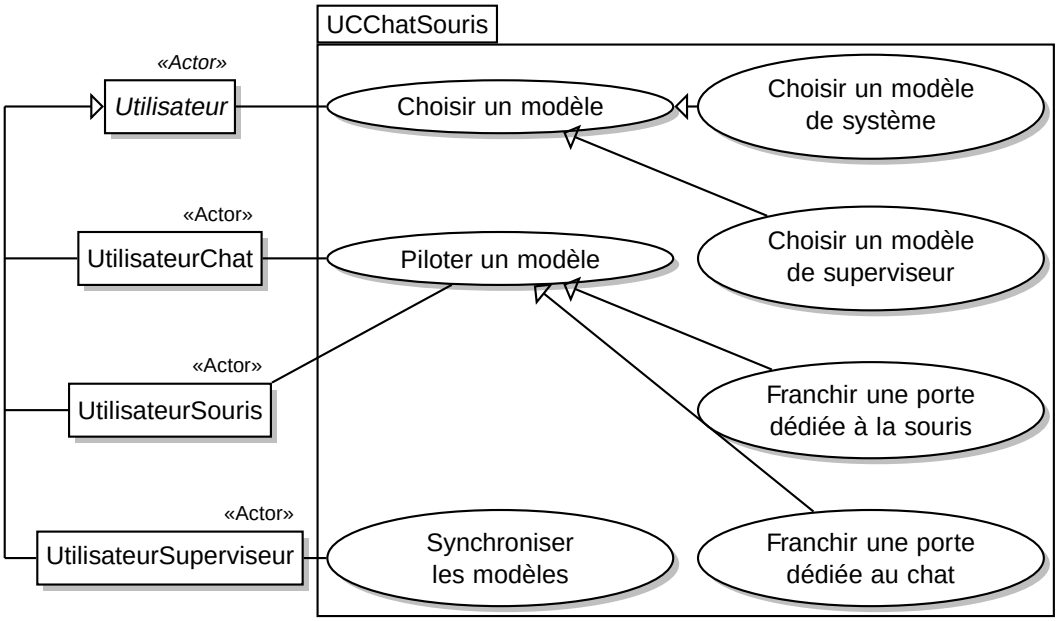
La mise en œuvre d'une telle application requiert la mise en place d'un processus de développement spécifique. Le processus considéré est ici celui qui a été défini dans le chapitre 5 (figure 5.17). Ce processus, décrit partiellement mais de façon plus précise avec la figure 6.3, associe à chaque activité un ensemble de langage de modélisation spécifiques tels que UML, les automates (FSM) ou encore le méta-modèle J2ME, etc. Ces langages sont supportés par différents outils ou plate-formes grâce à des transformations de modèles permettant également l'implantation de l'application finale sur la plateforme spécifique mobile. Les sections suivantes proposent de parcourir ce processus et de présenter les différents modèles réalisés.

a. Spécification

Dans un premier temps, il est nécessaire de traduire le cahier des charges à l'aide de modèles. Ces modèles vont permettre, dans la phase de spécification, de cerner le problème ainsi que de définir le périmètre de la réalisation à mettre en œuvre.

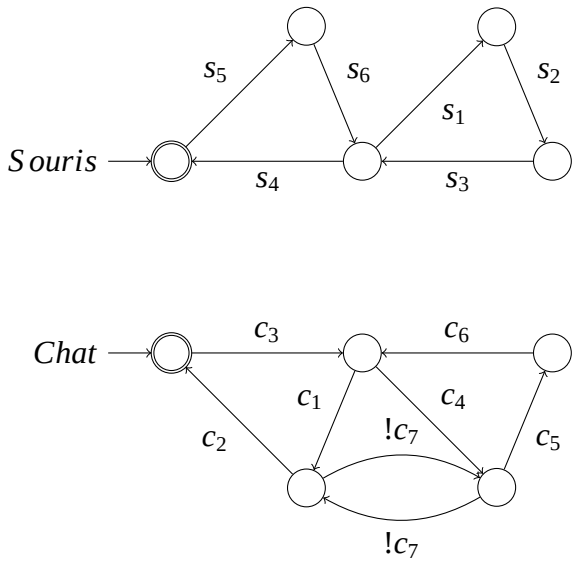
Le diagramme des cas d'utilisation (figure 6.4) présente ici les fonctionnalités qui doivent être mises en place au sein de l'application. Ainsi, chaque utilisateur devra faire le choix d'un modèle à utiliser : celui du chat, celui de la souris ou celui du superviseur. En fonction de ce choix, il aura alors la possibilité d'être soit joueur autonome (s'il choisit le chat ou la souris) en proposant les portes qu'il souhaite franchir soit arbitre en synchronisant l'application et les actions effectuées.

Le cahier des charges permet également de spécifier sous la forme d'automates les comportements du chat et de la souris mais aussi de définir les contraintes d'utilisation des différentes portes.



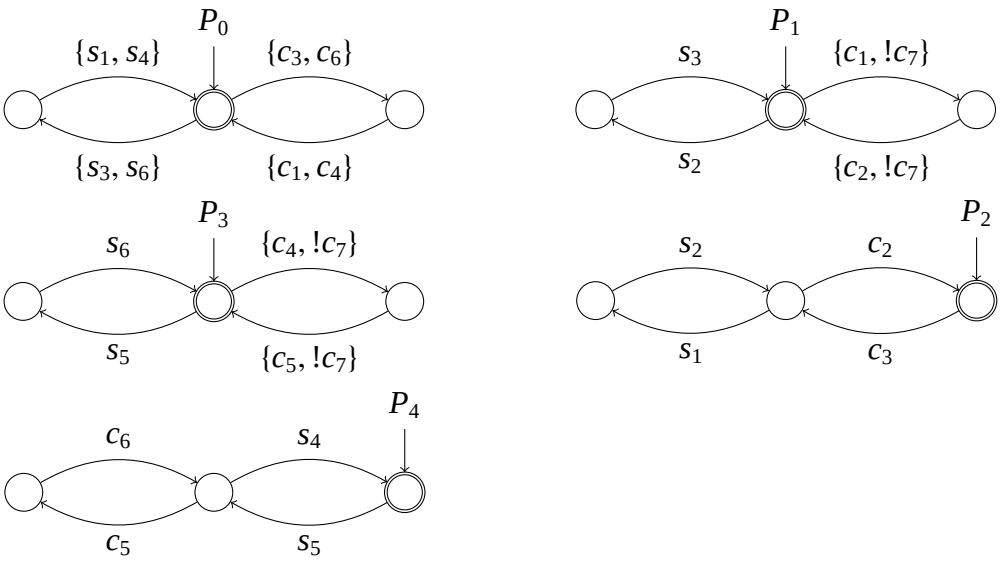
F . 6.4 – Diagramme des cas d'utilisations de l'application

Le chat se trouve initialement dans la pièce 2 et la souris dans la pièce 4. Le plan de circulation est modélisé par les automates de comportement autonome (non contraint par le superviseur) du chat et de la souris (figure 6.5). Afin de garantir l'exclusion mutuelle, la spécification précise les règles d'utilisation des différentes pièces ; des modèles sont donnés par les automates P_0 à P_4 de la figure 6.6.



F . 6.5 – Modèle de comportement autonome du chat et de la souris

Par exemple, l'automate de la pièce P_0 précise que si le chat l'occupe (transition c_3 ou c_6), il interdit les transitions s_3 et s_6 qui permettrait à la souris de l'occuper également. Seule l'occurrence de c_1 ou de c_4 représentant la sortie du chat de cette pièce permet à la souris d'y entrer (avec s_3 ou s_6).



F . 6.6 – Règle d'utilisation des différentes pièces pour éviter une rencontre entre le chat et la souris

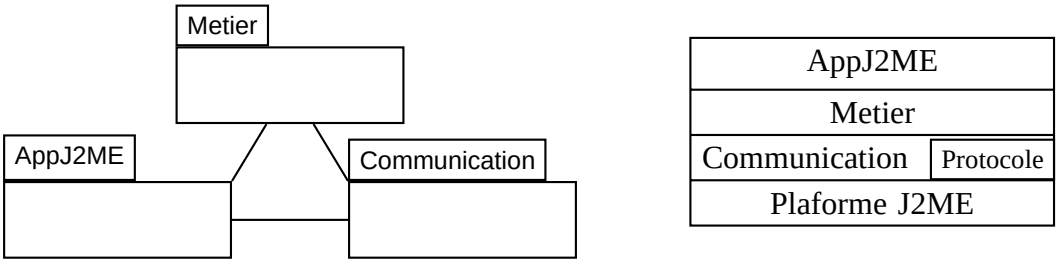
b. Conception générale

La théorie de la supervision ne précise pas la manière dont elle doit être mise en œuvre (Annexe B). Pour être conforme à cette théorie, un système produit spontanément et de façon asynchrone des évènements observés par un superviseur. Celui-ci autorise après chaque évènement l'occurrence des évènements contrôlables conformément aux spécifications. La réaction du superviseur doit être atomique, ce qui signifie que le processus ne peut évoluer tant que le superviseur n'a pas imposé les nouvelles règles d'évolution. Cette contrainte doit être prise en compte par l'architecture de la réalisation.

L'aspect décentralisé de l'application renforce cette contrainte et requiert la définition d'un modèle de communication spécifique, sous la forme d'un protocole, pour garantir l'atomicité entre échanges d'évènements et réactions du superviseur. En effet, une mise en œuvre trop naïve : (1) franchissement d'une porte par le chat ou la souris, (2) notification de l'évènement au superviseur (3) réaction du superviseur par la mise à jour des droits d'évolution auprès des partenaires, ne convient pas. Ce modèle de communication est incomplet car il ne garantit pas qu'entre le franchissement d'une porte par l'un des partenaires et la fin de réaction du superviseur, l'autre partenaire n'ait pas évolué.

Pour rendre atomique la transition, un protocole de communication spécifique doit être conçu, validé et implanté. Il est décrit plus en détail dans la partie "Conception d'un mécanisme de communication".

Après ces considérations générales concernant l'implantation d'une commande par supervision, il est nécessaire de proposer une architecture permettant d'exécuter les modèles abstraits exploités dans le cadre théorique, offrant un mécanisme de communication pour la synchronisation, et permettant l'interaction avec l'utilisateur. Pour cela, trois paquetages (figure 6.7) reposant sur des méta-modèles différents sont envisagés.

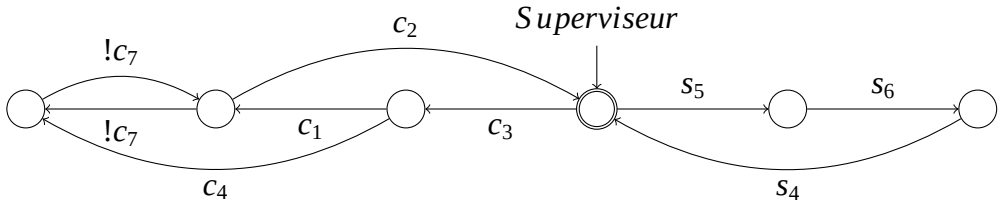


F . 6.7 – Modules nécessaires à l'application.

Le premier paquetage, appelé *Métier*, et qui correspond aux aspects purement métiers prend en charge l'exécution des modèles du chat, de la souris et du superviseur. Il s'appuie sur le méta-modèle des automates à états finis implanté dans le framework java (*DestKit*) dédié à l'exécution des modèles de ce domaine. Le deuxième paquetage nommé *AppJ2ME* réalise l'assemblage des différentes parties de l'application sous la forme d'une Midlet et sera détaillé ultérieurement. Enfin le troisième paquetage, nommé *Communication*, gère les problèmes d'atomicité de la réaction du superviseur par un protocole. Il sera également détaillé ultérieurement. L'ensemble de ces éléments est résumé par le modèle en couche présenté dans la figure 6.7.

c. Conception du modèle du superviseur par synthèse

Cette phase exploite les modèles de comportement autonomes du chat et de la souris (figure 6.5) ainsi que les cinq modèles indépendants spécifiant les règles d'utilisations des pièces (figure 6.6) pour synthétiser le modèle du superviseur. Pour cela, l'outil Supremica est utilisé. Cependant, ces différents modèles définis sur la base du méta-modèle des automates à états finis (défini en Ecore présenté au chapitre 3) doivent être traduits pour être acceptés par l'outil Supremica. A cet effet, la transformation de syntaxe définie avec l'outil Sintaks, et décrite dans le chapitre 5, est utilisée.

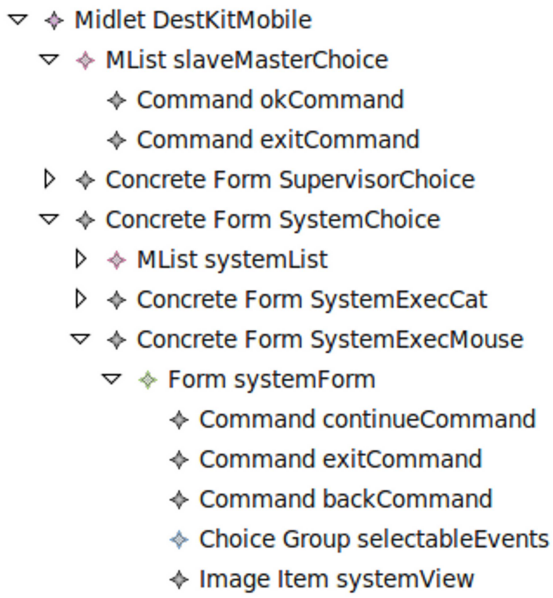


F . 6.8 – Modèle du superviseur pour le chat et la souris

Après transformation des modèles du chat, de la souris et de la spécification, Supremica effectue la synthèse d'un superviseur. Ce superviseur, dont le modèle est donné à la figure 6.8, répond aux spécifications énoncées puisqu'il propose comme solution de choisir initialement lequel du chat ou de la souris pourra se déplacer, tout retour à l'état initial permettant de refaire ce choix. Ce résultat est ensuite à nouveau transformé vers le méta-modèle source afin de le rendre, par la suite, lors de l'implantation, compatible avec le framework *DestKit* permettant son exécution.

d. Conception du paquetage AppJ2ME

Le paquetage *AppJ2ME* permet la mise en place du contexte d'implantation de l'application sous la forme d'une application mobile. Le modèle dans ce paquetage est conforme au méta-modèle d'une application J2ME (figure 5.6). Il vise à fournir à l'utilisateur les moyens de choisir le rôle qu'il va tenir lors de l'exécution de l'application (figure 6.9).



F . 6.9 – Modèle de l'application J2ME

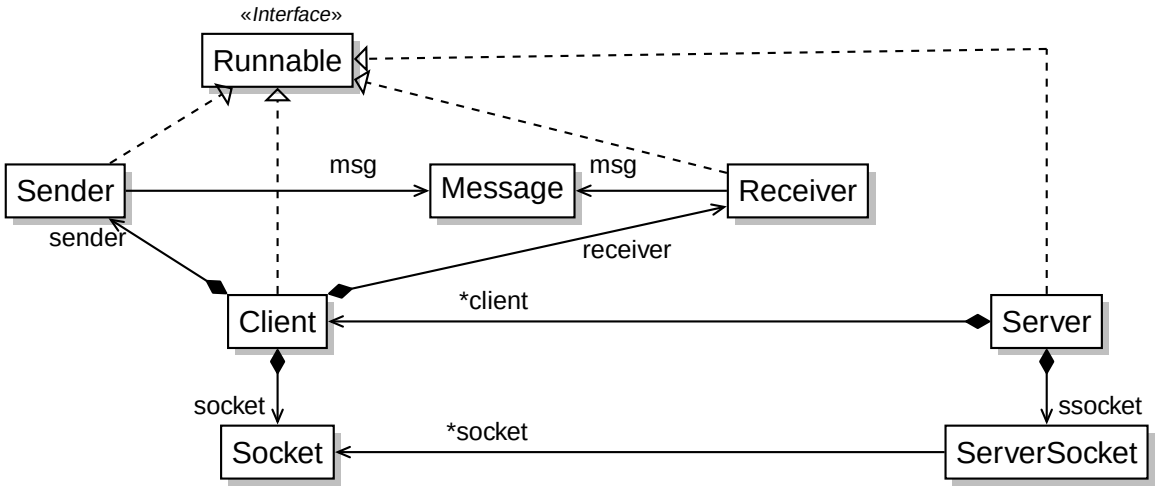
Cette figure représente, sous la forme d'une arborescence, le modèle de l'application et les divers choix possible de son utilisation. La racine de cet arborescence est une classe de type *Midlet*. Un choix est effectué entre différentes *ConcreteForm* (choix pris en charge par un élèment de type *MList*), et est validé par un élèment de type *Command*. L'application opère alors un changement de focus vers un nouvel élèment de type *ConcreteForm* : soit un *SupervisorChoice* (contenant un ensemble de superviseurs potentiels), soit un *SystemChoice* (contenant un ensemble de systèmes potentiels, chat ou souris par exemple).

Un nouveau choix est alors proposé (parmi les éléments de l'ensemble) afin de sélectionner quel modèle métier (le chat ou la souris) sera exécuté au sein de l'application. Une fois le modèle choisi, son évolution est gérée par le même mécanisme : le choix d'un évènement (contenu dans un *ChoiceGroup*) est effectué puis validé par l'utilisation d'un élèment de type *Command* qui lui sera associé.

e. Conception d'un mécanisme de communication

Le package *Communication* assure la communication entre les plateformes mobiles et en particulier l'atomicité d'un cycle de supervision. Ce paquetage est réalisé par un développement classique. Le fonctionnement est de type *Client-Serveur*. Le modèle (figure 6.10) de ce paquetage se fonde sur les spécificités de la plateforme Java en utilisant les

classes *ServerSocket* et *Socket* qui permettent de mettre en place le support de communication de façon à construire un pico-réseau BlueTooth dans lequel le superviseur sera le maître et le chat et la souris seront les esclaves, un *Serveur* est mis en place. Ce serveur va alors se mettre en relation avec les différents clients en créant lui-même ses clients (figure 6.11).



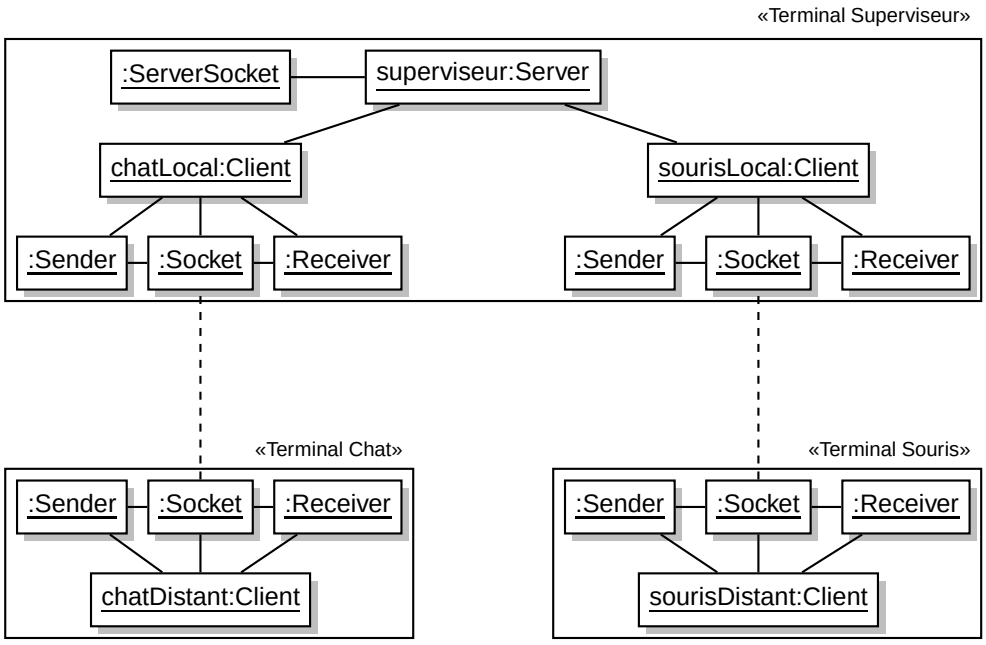
F . 6.10 – Modèle de communication client serveur

Ce modèle de communication sera instancié selon deux configurations résumées par la figure 6.11. Les terminaux mobiles intégrant le chat et la souris utiliseront chacun un objet de type *Client* qui enverra ou recevra des *Messages* gérés par un *Sender* et un *Receiver* au travers de la *Socket*. Le terminal mobile intégrant le superviseur utilisera un objet de type *Server* qui sera en attente de connection avec des clients externes. A chaque requête de connection, le serveur créera un client local (de type *Client*) auquel il donnera une *Socket* créée à cette occasion par son *ServerSocket*. Le client local est alors considéré par le serveur comme le porte parole du client distant.

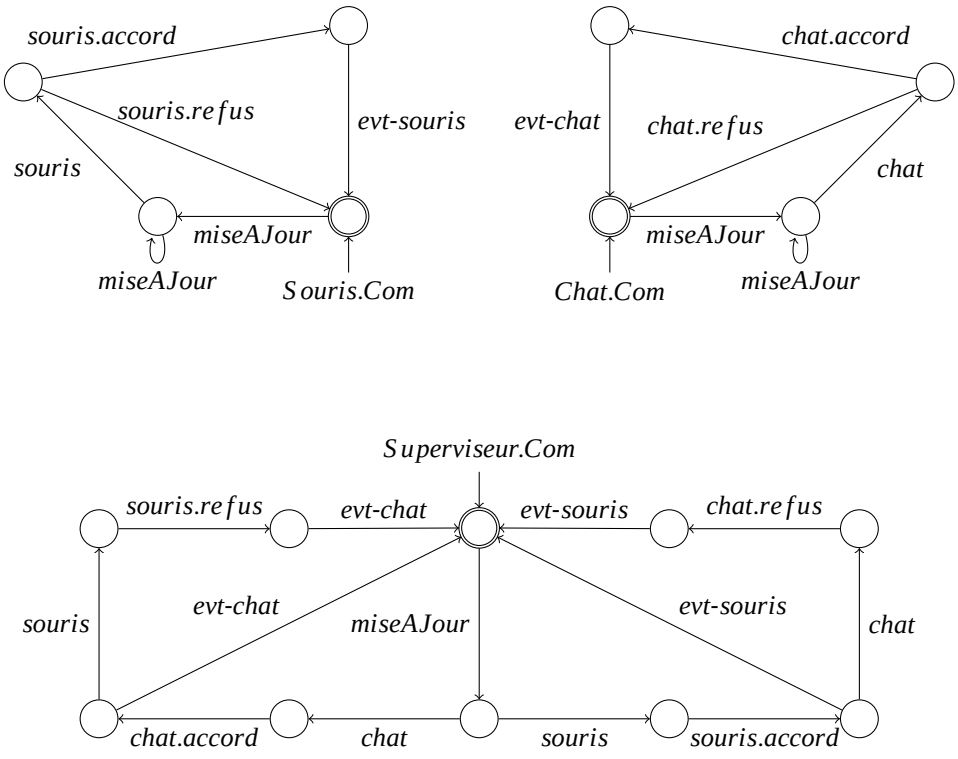
Cette couche de communication doit ensuite être complétée par une couche intégrant un protocole spécifique pour garantir l'atomicité des traitements de supervision. Ce protocole peut être modélisé par des automates communicants servant d'interface entre la couche communication et la couche métier qui contient les modèles abstraits du chat de la souris et du superviseur.

La figure 6.12 présente les modèles de ce protocole pour le chat (*Chat.Com*), la souris (*Souris.Com*) et le superviseur (*Superviseur.Com*). La communication débute par une synchronisation de tous les acteurs par le message *miseAJour*. L'objectif de *miseAJour* est de transmettre au chat et à la souris les évènements autorisés.

Chat.Com et *Souris.Com* peuvent alors émettre un message (respectivement *chat* ou *souris*) afin de demander au superviseur l'accord pour franchir une transition du modèle métier. Cette demande peut être acceptée (*x.accord*) auquel cas un évènement est émis (*evt-souris* ou *evt-chat*) par le modèle métier correspondant, ou refusée (*x.refus*).



F . 6.11 – Utilisation du modèle de communication



F . 6.12 – Modèle du protocole de communication entre le chat, la souris et le superviseur

Du coté superviseur, la réception d'un message de demande d'accord de mouvement de la part d'un premier partenaire induit également, un refus de mouvement temporaire et garantit ainsi l'atomicité du traitement.

f. Implantation

La phase d'implantation consiste en la mise en place de l'application sur la plateforme d'exécution J2ME de Sun/Oracle. La dernière phase du développement consiste alors à effectuer des transformations de génération de code Java à partir des différents modèles. Les modèles d'automates seront alors traduits conformément au framework *DestKit* assurant leur exécution, le modèle de cette transformation a été présenté au chapitre 5.4.5.

```
//Création de l'automate
FiniteStateMachine fsm = new FiniteStateMachine("mouse");

//Création et ajout des états servant de modèles pour les pièces
fsm.addState(new State("M0", State.NOTINITIAL,State.NOTMARKED));
fsm.addState(new State("M1", State.NOTINITIAL,State.NOTMARKED));
fsm.addState(new State("M2", State.NOTINITIAL,State.NOTMARKED));
fsm.addState(new State("M3", State.NOTINITIAL,State.NOTMARKED));
fsm.addState(new State("M4", State.INITIAL, State.MARKED));

//Ajout des transitions qui modélisent le changement de pièce
fsm.addTransition("M0", "s1", "M2");
fsm.addTransition("M2", "s2", "M1");
fsm.addTransition("M1", "s3", "M0");
fsm.addTransition("M0", "s4", "M4");
fsm.addTransition("M4", "s5", "M3");
fsm.addTransition("M3", "s6", "M0");
```

F . 6.13 – Code Java généré du modèle de la souris.

La figure 6.13 présente le résultat de la transformation (FSM vers DestKit) du modèle de la souris réalisée avec l'outil Sintaks. La structure classique d'instanciation d'objets Java se retrouve clairement ici. Ainsi, un automate *souris* est instancié, puis des états comportant les propriétés propres à la théorie de la commande par supervision sont ajoutés. Enfin, des transitions labellisées par les évènements sont établies entre ces états. De la même manière, afin d'implanter la partie J2ME, il est possible d'utiliser la transformation définie en Xpand entre le méta-modèle de la plateforme J2ME et la syntaxe Java correspondante (figure 5.16). Ainsi, une classe *DestKitMobile* héritant de *Midlet* est générée comme point d'entrée de l'application. Elle va réaliser l'interface *CommandListener* de façon à proposer à l'utilisateur le choix entre le rôle de superviseur, de chat ou de souris à l'aide d'objets *Command*, pour intéragir avec l'utilisateur, et de *Displayable* pour les représenter. De la même manière, la transformation va produire un ensemble de classes réalisant l'interface *CommandListener*, reliées entre elles de façon arborescente, et intégrant également des objets *Command* et *Displayable*. La racine de cette arborescence est comme pour le modèle J2ME initial (figure 6.9) la classe *DestKitMobile*.

Après la génération de ces éléments, il est nécessaire de finaliser l'application par une implantation classique de la partie client-serveur en intégrant la réalisation concrète des modèles de communication présentés précédemment (figures 6.10 et 6.11). La réalisation

de l'application étant terminée, une dernière phase consiste à déployer l'application dans un système distribué communicant afin d'en valider les fonctionnalités.

g. Déploiement et fonctionnement

Le système distribué supportant l'installation de l'application sera composé de trois terminaux mobiles de type PDA communicant via BlueTooth.

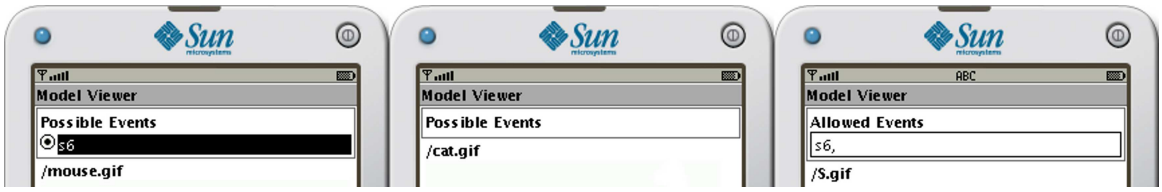
Le fonctionnement de l'application illustrera les différents cas d'utilisations. Entre autre, il montrera en situation la communication entre les différents acteurs autonomes ainsi que leur synchronisation et ceci conformément aux spécifications.

F . 6.14 – Phase initiale de l'application

La même application est installée sur chacun des systèmes. Pour vérifier le bon fonctionnement de l'application globale, trois instances sont exécutées puis configurées selon les rôles des différents acteurs : le chat, la souris et le superviseur (figure 6.14).

F . 6.15 – Phase d'attente de la première action de la part de la souris ou du chat

Le superviseur effectue tout d'abord la synchronisation des différentes entités. Le chat et la souris sont dans leurs états initiaux. Il leur est possible d'effectuer une première action (figure 6.15) qui sera sous le contrôle du processus de supervision.



F . 6.16 – Action de la souris

La figure 6.16 montre le système dans un état où la souris a pris la main en exécutant l'action s_5. Après réaction du superviseur, elle a la possibilité d'effectuer l'action s_6 alors que pendant ce temps, en accord avec les contraintes imposées par le superviseur, le chat ne peut rien faire.

F . 6.17 – Action du chat

La figure 6.17 montre à l'inverse le système dans un état où le chat a pris la main. Après avoir exécuté l'action c_3, il a, en accord avec le processus de supervision, la possibilité d'effectuer les actions c_1 ou c_4 alors que pendant ce temps, la souris ne peut rien faire.

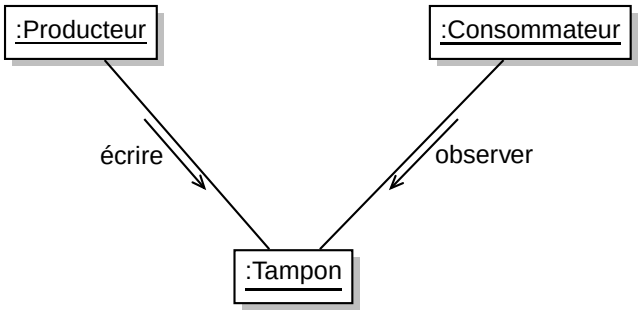
Le processus se poursuit ainsi ; le chat ou la souris génèrent différents évènements en accord avec ceux autorisés par le superviseur pour être ensuite, après chaque action, remis à jour et resynchronisés.

Cette phase de test, bien que simpliste, permet une première validation des deux principaux scénarios. Entre autre que le chat ou la souris lorsqu'ils se déplacent, empêchent jusqu'à leur retour dans leur état initial tous les déplacements du second grâce au contrôle effectué par le superviseur.

6.2 Illustration II : Développement associant synthèse et vérification de modèles

6.2.1 Problème du producteur consommateur

Le deuxième problème à traiter concerne un schéma classique de type producteur consommateur mettant en œuvre une communication par envoi de messages illustré par la figure 6.18. Cet exemple va s'intéresser à la possibilité d'utiliser le domaine du Model-Checking en complément de la commande par supervision pendant l'activité de conception.



F . 6.18 – Exemple du producteur consommateur

Le producteur produit des valeurs cycliquement et les écrit dans un tampon à une place. Le consommateur consomme ces valeurs au fur et à mesure de leur production en observant

le tampon. Des cas d'erreurs sont considérés et le fonctionnement est affiné de la manière suivante :

- Le producteur évalue cycliquement une valeur à fournir au consommateur. Après cette évaluation, deux cas sont à considérer :
  1. La donnée a été correctement évaluée, elle est alors écrite dans le tampon.
  2. La donnée n'a pas pu être évaluée correctement, dans ce cas le producteur émet un signal d'échec et doit être réinitialisé avant de recommencer son évaluation.
- Le consommateur observe les valeurs déposées dans le tampon. Selon la valeur déposée, deux cas sont distingués :
  1. La valeur observée est acceptable auquel cas le consommateur affiche simplement la donnée observée.
  2. La donnée observée est hors limite auquel cas, le consommateur signale ce fait en émettant une alarme qui doit être acquittée avant de recommencer une observation.

On notera que l'émission d'un signal d'échec ou d'alarme ne peut être interdit, ils seront donc considérés comme des événements non commandables. De même, l'écriture ou l'affichage seront considérés comme étant non commandables[1].

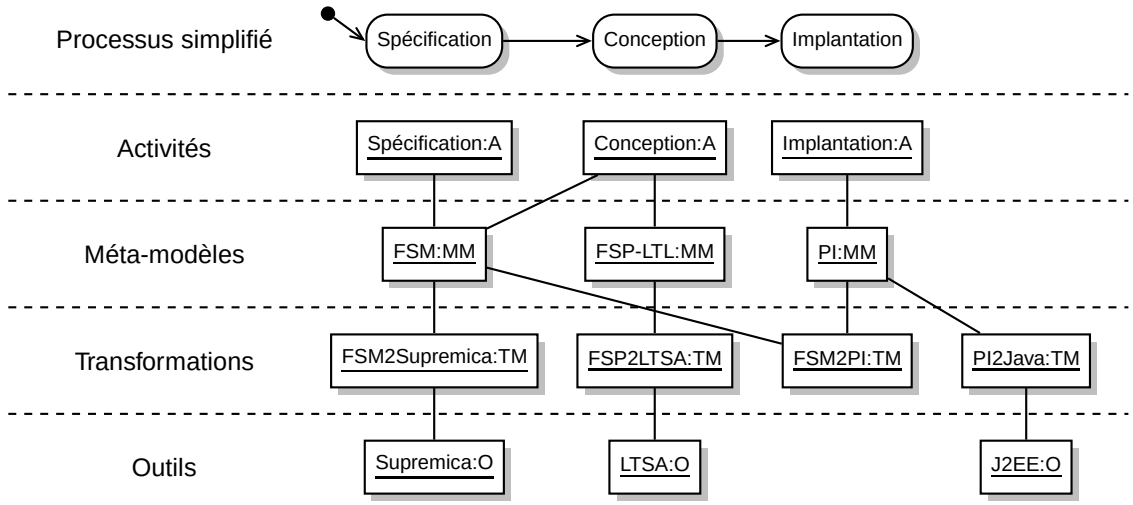
En suivant ce schéma, le cycle normal de fonctionnement souhaité consiste à évaluer puis écrire une donnée et à observer puis à afficher celle-ci. Afin de respecter cette spécification et de ne pas perdre de données, une contrainte doit être définie. Celle-ci imposera les règles d'utilisation du tampon à une place afin d'assurer qu'après une écriture aucune autre ne puisse être faite sans observation.

6.2.2 Mise en œuvre du processus de développement

Ce schéma producteur consommateur sera mis en œuvre sur la plateforme multi-agents PI dont le méta-modèle a été décrit au chapitre 5. La théorie de la commande par supervision sera utilisée pour synthétiser un superviseur assurant un fonctionnement ne violant pas les spécifications imposées, qui cependant, ne sont que des spécifications de sûretés. En effet, la synthèse peut mener à un superviseur interdisant tous les comportements possibles, et donc éviter tous les comportements potentiellement dangereux, mais du coup ne permet plus aucune vivacité au système. Parce que l'algorithme de synthèse ne permet pas de garantir les propriétés de vivacité du système, celles-ci doivent être garanties par une phase complémentaire de vérification de modèles.

Un processus de développement spécifique imposé par ces contraintes conduit alors au modèle de développement de la figure 5.17. Ce processus, décrit partiellement mais de façon plus précise avec la figure 6.19, associe à chaque activité un ensemble de langages de modélisation spécifiques tels que UML, les automates (FSM) pour la théorie de la commande par supervision, l'algèbre de processus FSP pour le domaine du Model-Checking

[1] Rappel : les événements non commandables sont dans la théorie de la commande par supervision préfixé par un " !"

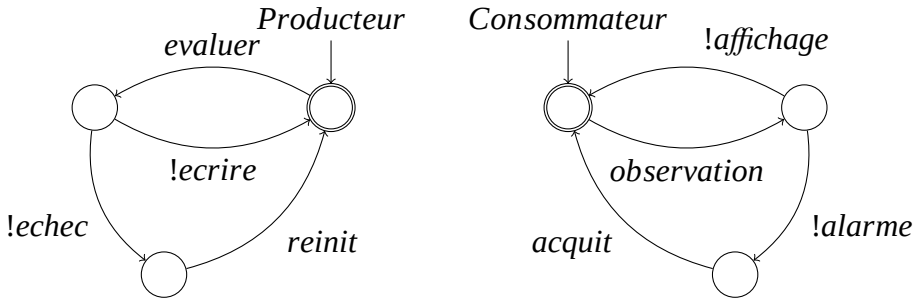


F . 6.19 – Présentation partielle des éléments constituant le processus de développement (A : Activité, MM : Méta-modèle, TM : Transformation de modèle, O : Outil)

ou encore le méta-modèle PI, etc. Ces langages sont supportés par différents outils grâce à des transformations de modèles qui permettent également l'implantation de l'application finale sur la plateforme Java supportant le framework multi-agents PI. Les sections suivantes proposent de parcourir ce processus et de présenter les différents modèles réalisés.

a. Spécification

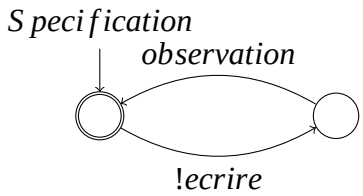
L'essentiel de cette activité de spécification consiste en la traduction de l'énoncé défini précédemment de façon à avoir un modèle plus formel du problème. Ainsi, en utilisant des automates à états finis, il est possible de définir les modèles du producteur, du consommateur (figure 6.20) et de la spécification (figure 6.21).



F . 6.20 – Automates du producteur et du consommateur (le symbole " !" précise la non commandabilité de certains évènements)

Les modèles du producteur et du consommateur décrivent l'ensemble des comportements pouvant être effectués. Par contre, la spécification est définie selon les propriétés devant être absolument respectées pour le bon fonctionnement du système. Elle doit :

– garantir la réalisation successive d'une *écriture* puis d'une *observation* sans interdire le reste des autres comportements possibles,



F . 6.21 – Automate de la spécification

– interdire l'occurrence de deux *écritures* successives sans qu'il y ait eu *observation* entre ces deux actions. Ceci est respecté par défaut car la spécification définit que le système doit réaliser une *écriture* puis une *observation* (sur la base de l'alphabet constitué de ces deux évènements, le modèle va naturellement interdire toute seconde *écriture* après une première).

b. Conception

Lors de la phase de conception, le travail se concentre sur la réalisation du modèle du superviseur produit à partir des modèles du système et du modèle de la spécification.

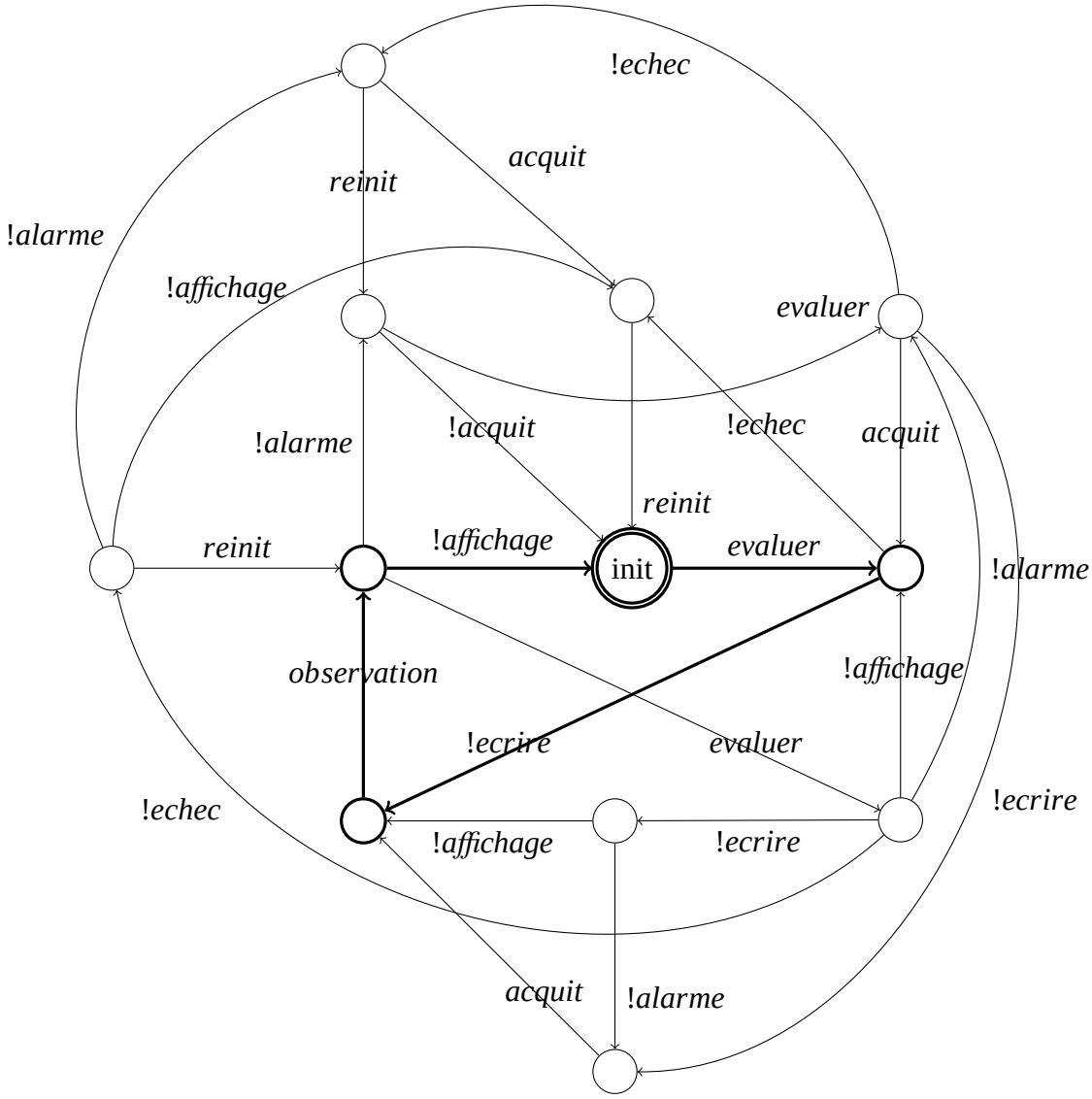
Afin de permettre la synthèse d'un superviseur, les modèles sont transformés pour être compatibles avec l'outil Supremica. L'outil produit automatiquement le modèle de superviseur donné à la figure 6.22. Il est à noter que malgré la simplicité du problème posé, ce modèle est loin d'être évident et serait difficile à construire d'emblée. Après synthèse et sur la base de la transformation de syntaxe, ce modèle est réintégré dans la plateforme de développement initiale (ici TopCaseD/EMF).

La théorie de la supervision permet de générer un modèle de superviseur qui garantit que le système ne réalisera pas de comportements interdits. Cependant des problèmes de contrôle liés à la nature de certains évènements du système peuvent survenir : le superviseur peut éventuellement supprimer des comportements nécessaires à la bonne exécution du système avec une perte éventuelle de propriétés de vivacité. Ainsi, avant de réaliser la phase d'implantation, il est utile de vérifier que le superviseur une fois couplé au système n'interdit pas des comportements essentiels au schéma du producteur-consommateur.

Pour effectuer la vérification du comportement obtenu, il est envisagé d'utiliser l'algèbre de processus FSP complétée de formules LTL pour exprimer les propriétés de vivacité auxquelles on s'intéresse. Pour utiliser ce domaine, il est nécessaire dans un premier temps d'effectuer une transformation de modèle entre le (méta-)modèle d'automates à états fini et celui de FSP afin d'obtenir des modèles FSP du producteur, du consommateur et du superviseur. Ainsi, la transformation de modèles décrite au paragraphe 5.4.2 et basée sur ATL est utilisée.

Aux modèles FSP obtenus à partir de la transformation des automates, il est nécessaire d'adjoindre des formules LTL définissant les propriétés de vivacité. Pour construire ces formules LTL, il faut d'abord définir un marquage à l'aide des *Fluent* qui représenteront les Propositions Atomiques utilisables pour les formules. Le marquage consiste à identifier les

événements entrant et sortant d'un état que l'on souhaite marquer de façon à le discriminer des autres et ainsi pouvoir y associer des propriétés (PA).



F . 6.22 – Modèle du superviseur synthétisé avec Supremica

Un premier marquage nommé *Ecriture* est défini en spécifiant que la réalisation d'une *écriture* active cette propriété et qu'une *observation* la désactive. Un second marquage nommé *Observation* est réalisé en spécifiant qu'une *observation* active cette propriété et qu'*affichage* ou *alerte* clôt ce marquage.

A partir de ces deux marquages qui servent de Proposition Atomique, il est possible de construire les formules LTL permettant de mettre en évidence la vivacité du système :

– Vivacité du producteur :

$$ProducteurVivant = Globaly(\ Futur(\ Ecriture\ ))$$

– Vivacité du consommateur :

$$ConsommateurVivant = Globaly(\ Futur(\ Observation\ ))$$

– Garantir qu'après une production, une consommation aura lieu :

$$ProdPuisCons = Globaly(\ Ecriture\ Implies\ Futur(\ Observation\ ))$$

– Garantir la communication :

$$ComToujoursPossible = (\ ProducteurVivant\ \&\&\ ProdPuisCons\ )$$

Il reste à transformer ces modèles dans la syntaxe concrète utilisée par LTSA afin de vérifier les propriétés. Pour cela, le modèle de transformation de syntaxe (figure 5.12) proposé au chapitre 5.4.4 va être utilisé pour obtenir le modèle donné à la figure 6.23.

Dans ce code FSP représentant les modèles précédents, les modèles du producteur, du consommateur et du superviseur sont synchronisés afin de mettre en œuvre le mécanisme de supervision. La vérification avec l'outil LTSA nous fournit la garantie que le système supervisé ne possède ni blocage (déjà assuré par la génération du superviseur) ni problèmes de vivacité particuliers. Une simulation interactive permet également d'améliorer la compréhension du fonctionnement du système avant son implantation.

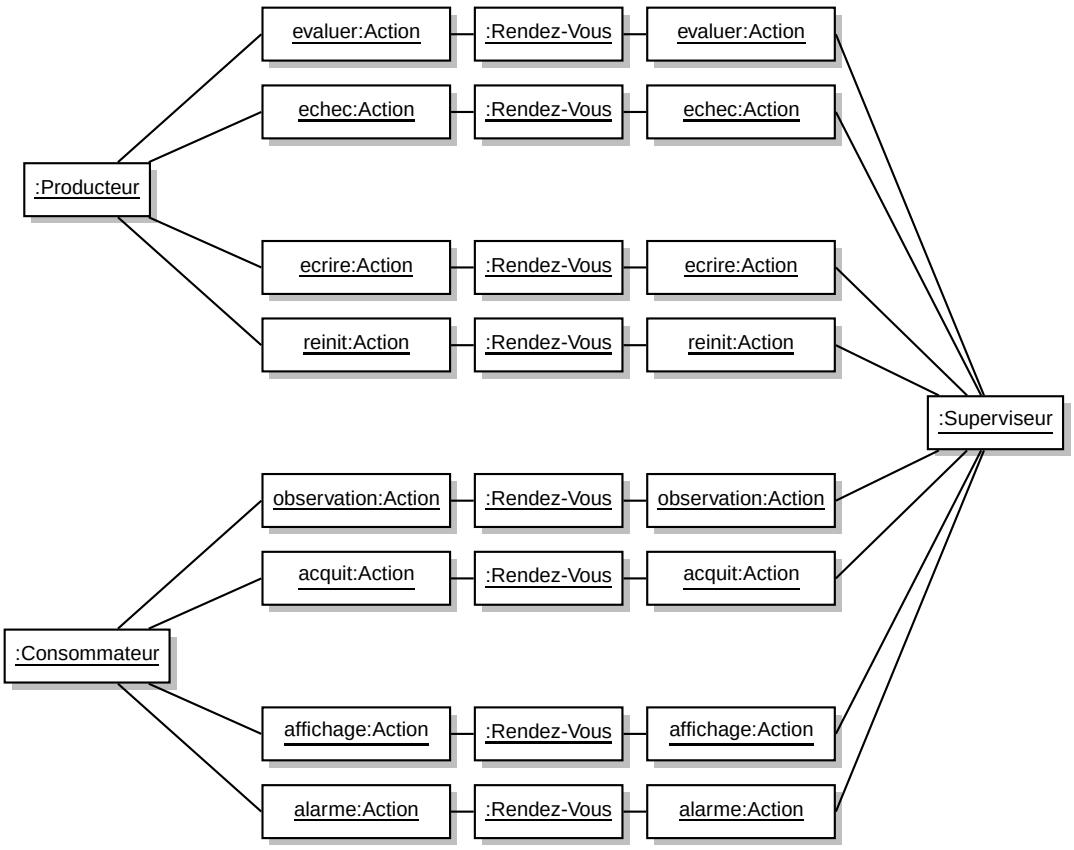
```
Producteur=A,                                                    ====> Figure 6.21
A=(evaluer->B),
B=(ecrire->A|echec->C),
C=(reinit->A)+{evaluer,reinit,ecrire,echec}.

Consommateur=D,                                                  ====> Figure 6.21
D=(observation->E),
E=(affichage->D|alarme->F),
F=(acquit->D)+{observation,acquit,affichage,alarme}.

Superviseur=DAI,                                                 ====> Figure 6.22
DAI=(evaluer->DBI),
DAJ=(observation->EAI),
DBI=(echec->DCI|ecrire->DAJ),
DCI=(reinit->DAI),
EAI=(alarme->FAI|affichage->DAI|evaluer->EBI),
EAJ=(alarme->FAJ|affichage->DAJ),
EBI=(alarme->FBI|affichage->DBI|echec->ECI|ecrire->EAJ),
ECI=(alarme->FCI|affichage->DCI|reinit->EAI),
FAI=(acquit->DAI|evaluer->FBI),
FAJ=(acquit->DAJ),
FBI=(acquit->DBI|echec->FCI|ecrire->FAJ),
FCI=(acquit->DCI|reinit->FAI) + {acquit,alarme,affichage,evaluer,echec,observation,reinit,ecrire}.

||SystemeGlobal=(Superviseur||Producteur||Consommateur).         ====> Synchonisation des modèles

fluent Ecriture=<{ecrire},{observation}> initially 0             ====> Marquages
fluent Observation=<{observation},{affichage,alarme}> initially 0

assert ProducteurVivant=[]<>Ecriture                             ====> Formules LTL
assert ConsommateurVivant=[]<>Observation
assert ProdPuisCons=[](Ecriture-><>Observation)
assert ComToujoursPossible=(ProducteurVivant && ProdPuisCons)
```

F . 6.23 – Code FSP pour la vérification des propriétés de vivacité
([] : opérateur *Globaly*, <> : opérateur *Futur*, -> : opérateur *Implies*)

c. Implantation

Selon le cahier des charges, l'implantation doit être réalisée sur la plateforme PI. Comme cela a déjà été décrit, cette plateforme permet la synchronisation d'agents dont le comportement est décrit à l'aide d'automates.

Pour réaliser cette implantation, la solution envisagée consiste à utiliser l'outil Xpand en transformant directement les modèles d'automates vers un code Java respectant l'architecture de la plateforme PI. Cette solution est décrite au paragraphe 5.4.6 et va permettre la génération des classes *Producteur*, *Consommateur* et *Superviseur* héritant toutes les trois de la classe *DefaultDiscreteAgent* du framework PI. Comme pour l'exemple du chat et de la souris, il est nécessaire de prévoir une façon simple d'implanter le mécanisme de supervision supposé par la théorie. Ici cette réalisation est facilitée par le mécanisme d'actions partagées (ou *RendezVous*) offert par la plate-forme et permettant de synchroniser les actions du producteur, du consommateur et du superviseur. Un modèle objet des éléments ainsi créés est donné par la figure 6.24.



F . 6.24 – Modèle objet du Producteur, du Consommateur et du Superviseur au sein de la plateforme PI

Une fois les classes *Producteur, Consommateur* et *Superviseur* générées, il reste à les intégrer à la plateforme d'exécution et à construire le modèle de l'application conformément à la structure de PI (figure 6.25).

```
Executive exec=new Executive();
Consommateur cons=new Consommateur();
Producteur prod=new Producteur();
Superviseur s=new Superviseur();

new RendezVous(new Fireable[]{cons.getObservation(),s.getObservation()});
new RendezVous(new Fireable[]{cons.getAlarme(),s.getAlarme()});
new RendezVous(new Fireable[]{cons.getAffichage(),s.getAffichage()});
new RendezVous(new Fireable[]{cons.getAcquit(),s.getAcquit()});
new RendezVous(new Fireable[]{prod.getEvaluer(),s.getEvaluer()});
new RendezVous(new Fireable[]{prod.getEchec(),s.getEchec()});
new RendezVous(new Fireable[]{prod.getEcrire(),s.getEcrire()});
new RendezVous(new Fireable[]{prod.getReinit(),s.getReinit()});

exec.plugAgent(cons);
exec.plugAgent(prod);
exec.plugAgent(s);
exec.start();
```

F . 6.25 – Modèle de l'application du producteur consommateur sur la plateforme PI

Ainsi, les seules actions possibles pour le producteur ou le consommateur seront celles effectuées simultanément par le superviseur.

Enfin, le moteur d'exécution est instancié (*Executive*). Dans celui-ci sont insérés les trois agents producteur, consommateur et superviseur. Le moteur d'exécution est lancé par l'appel de sa méthode *start* afin de vérifier le bon fonctionnement de l'application.

6.3 Conclusion

Dans la continuité de l'élaboration de processus de développement spécifiques pour les SED proposé au chapitre 5, ces deux exemples permettent de mieux cerner les avantages et les limites qu'offre l'emploi de l'approche proposée. Cette approche a permis de mettre en place deux modèles de processus de développement différents (figure 5.17) pour deux problèmes particuliers.

Dans le premier exemple, lors de la phase la plus abstraite qui est celle de spécification, des diagrammes de cas d'utilisation ont été utilisés ainsi que des automates à états finis afin de modéliser les différentes parties des modèles. Différents domaines tels que celui de la théorie de la supervision ont été intégrés au développement. Enfin, en cohérence avec l'architecture choisie, un ensemble de transformations de modèles ont permis l'obtention du code Java directement implanté sur la plateforme mobile cible.

Le second exemple illustre une autre utilisation de l'approche proposée pour intégrer deux domaines liés au développement des SED, ceux de la supervision et de la vérification de modèles. A l'aide de transformations de modèles adéquats, nous avons vu qu'il est alors possible de passer d'un domaine à un autre ou de s'appuyer sur des outils spécifiques de domaines (comme Supremica ou LTSA) et d'améliorer les garanties concernant le respect des propriétés énoncées dans le cadre de l'utilisation conjointe de ces deux domaines.

Ainsi, l'apport d'une démarche IDM dans la construction et l'évolution de modèles de cycle de développement spécifiques montre qu'il est possible de rationaliser les démarches de conception. En effet ces deux exemples montrent bien que les concepts fournis par l'IDM permettent de gérer les différents domaines et les langages de modélisation nécessaires à la réalisation que ce soit aux étapes de spécification, de conception ou d'implantation et de faciliter le passage de l'un à l'autre.

Pourtant, il est possible de remarquer que ces exemples ne sont pas exclusivement basés sur l'utilisation de modèles. En effet, il a été nécessaire d'effectuer différentes implantations manuellement (entre autre dans l'exemple du chat et de la souris pour lequel il est nécessaire de finaliser l'application manuellement). Ceci montre en soi les limites de cette approche qui nécessite malgré tout une expertise particulière ainsi qu'une capacité à effectuer des tâches que les concepts de l'IDM ne permettent pas encore d'automatiser.

Conclusion

Ainsi, un manque de connaissances scientifiques peut amener quelqu'un à émettre des conclusions hasardeuses.
(Jean-Pierre Petit)

... un fait courageux ne doit pas conclure un homme vaillant.

(Michel Eyquem de Montaigne)

Chapitre 7

Conclusion

L´ ` est considéré comme une tâche difficile qui nécessite une multitude de concepts divers et variés à intégrer. Plus précisément, la conception de logiciels impose la prise en compte des concepts et aspects technologiques particuliers à l'informatique et de ceux liés aux savoir-faire des domaines d'utilisation de ces mêmes logiciels ; la gestion du processus de développement étant quant à elle à ne pas négliger.

L'état de l'art porte sur les langages généralement utilisés pour décrire les systèmes logiciels et sur les cycles ou processus de développement comme ceux en V, en cascade ou agiles. Ces processus ont pour objectif de rationaliser les démarches de conception et de mieux intégrer les acteurs que sont les experts de domaines. Il est ainsi montré que les approches classiques ne permettent pas de gérer l'hétérogénéité des domaines particuliers à considérer pour un processus de développement spécifique. L'enjeu ici est de concevoir des processus spécifiques aptes à intégrer au mieux les théories, technologies, langages de modélisation, outils et plateformes de domaines spécifiques. Ils comporteront alors des activités spécifiques générant des produits spécifiques aux domaines. Pour ce faire, l'approche proposée cherche à combiner les acquis de l'IDM et de la modélisation des processus logiciels ; dans cet esprit, les concepts de modèles, méta-modèles et transformations de modèles de type M2M ou M2C occupent une place prépondérante. En partant d'exemples types, un schéma conceptuel de ces processus est proposé, il associe les concepts des domaines des processus (activités, rôles, produits, acteurs, ressources) à ceux de l'ingénierie dirigée par les modèles. Enfin, une ingénierie permettant de guider les concepteurs dans la mise en place des différents constituants des processus (méta-modèles, transformations, outils et plates-formes) est proposée.

Comme exemple d'application, un processus dédié au développement de Systèmes à Événements Discrets a été élaboré. Il se conforme au modèle d'ingénierie proposé ; à ce titre, il est préconisé d'identifier et de recenser les domaines (Synthèse de superviseurs, Model-Checking, Génération de code, ...), d'identifier les outils, frameworks, plateformes (Automates à états finis, Algèbre de processus FSP, LTSA, Supremica, ...), de concevoir et réaliser les méta-modèles spécifiques, de concevoir et réaliser les transformations pour relier les différents méta-modèles et enfin de modéliser le processus spécifique. La mise en

œuvre du processus obtenu est illustrée dans le contexte concret de deux développements menant aux réalisations d'une application décentralisée mobile et d'une application fondée sur le problème du producteur-consommateur.

L'approche proposée permet de rationaliser les démarches de développement en proposant grâce à l'IDM l'intégration dans un contexte commun des différents domaines nécessaires à la réalisation logicielle. Ces domaines peuvent provenir de l'ingénierie logicielle classique mais également des domaines métiers. L'utilisation des méta-modèles fournit de la souplesse à la démarche proposée et facilite la réutilisation des entités réalisées au fil des différents développements. De plus, l'IDM est aujourd'hui un domaine outillé proposant un support varié pour la mise en œuvre de cette approche.

Notre approche n'est cependant pas exempte d'inconvénients. En effet, l'utilisation de l'IDM impose l'utilisation des méta-modèles et transformations de modèles. Un inconvénient se situe alors dans la maintenance des modèles, transformations de modèles et outils lorsque les méta-modèles dont ils dépendent évoluent. Une prise en compte rigoureuse des modifications effectuées sur ces entités et de leurs conséquences est primordiale. Une autre limitation dans l'utilisation d'une approche IDM est qu'il reste difficile de définir des transformations syntaxiques pour la génération de code source couvrant la totalité des besoins de l'entité logicielle à produire. Ainsi, il est nécessaire de compléter l'application à l'aide d'approches de développements classiques.

L'approche propose un guide d'élaboration de processus spécifiques, il serait intéressant de poursuivre ces travaux afin de définir un vrai méta-modèle de processus autorisant alors l'hébergement et la manipulation des modèles du processus en vue de leur pilotage par un environnement dédié à cet usage. La problématique consistant à définir un langage de modélisation adapté à la description de processus dirigés par les modèles est d'ailleurs un sujet d'actualité dans la communauté IDM. Enfin, les exemples développés ont été choisis de façon à proposer une démonstration claire de l'approche. En parcourant l'ensemble des étapes d'un processus de développement, ils fournissent un premier moyen de validation qui reste cependant circonscrit à un contexte académique. Il serait intéressant de confronter cette approche à un contexte industriel réel intégrant de multiple types d'acteurs et imposant des contraintes industrielles fortes.

Annexe A

Vérification de Modèles

1 Introduction

Pour les systèmes critiques, où la sûreté de fonctionnement est primordiale, il est nécessaire de garantir que le système réalisé vérifie bien les spécifications définies dans le cahier des charges. Pour cela, en première approche, des techniques de validation basées sur des simulations et des tests sont employées. Elles ont pour but d'exécuter un programme (ou son modèle) afin d'y détecter un ensemble d'erreurs de fonctionnement, et servent à s'assurer que le bon système a été réalisé.

Cependant, bien que ce type de validation soit indispensable pour le développement logiciel, elle n'apporte qu'une réponse partielle au problème de sûreté. En effet, avec ce genre de tests, il est difficile, voir impossible, de parcourir l'ensemble des comportements possibles du système. Il est alors nécessaire de recourir aux techniques de vérification qui consistent à s'assurer que le système a été réalisé correctement.

Les méthodes formelles [Gab06], [JJ04], [LG97] sont des approches reconnues comme pouvant apporter des solutions à cette problématique. Elles reposent sur des spécifications et sur des techniques de vérification formelles.

2 Les langages formels

Un langage est dit formel s'il possède une syntaxe et une sémantique complètement définie reposant sur des fondements mathématiques. C'est sur cette base mathématique que vont s'appuyer les techniques de vérification.

Il existe de nombreux langages formels, parmi lesquels : des langages basés sur la théorie des ensembles (Z, B, etc.), des algèbres de processus (CSP, CCS, FSP, etc.), des logiques temporelles (LTL, CTL, etc.), des formalismes dédiés à la modélisation des protocoles de communication (LOTOS, etc.) ou encore des langages basés sur des automates à états/transitions (réseaux de Petri, automates à états finis, etc.). En particulier, dans ce

travail de thèse, nous nous sommes appuyés sur des formules de logiques temporelles et une algèbre de processus.

3 Les logiques temporelles

Les logiques temporelles sont des extensions des logiques classiques incluant la notion de temps. Elles spécifient un système à l'aide d'un ensemble de propositions qu'elles combinent avec un ensemble d'opérateurs logiques habituels et un ensemble d'opérateurs spécifiques à la gestion du temps. Ces logiques servent à exprimer des propriétés de comportement d'un système.

Différents types de logiques temporelles existent et peuvent être classées en deux grandes familles : les logiques dites à temps linéaires (LTL) [Pnu97] ou les logiques dites arborescentes (CTL) [CES86] selon la façon dont le temps va être considéré. Dans le cas des logiques linéaires, le temps est linéaire dans le passé et dans le futur : à chaque étape il n'existe qu'une seule évolution possible ; elles servent à exprimer des propriétés sur des chemins. Dans le second cas, il peut exister plusieurs évolutions possibles à partir de l'instant courant ; elles servent à exprimer des propriétés sur des arbres d'exécution.

4 Les algèbres de processus

Issus des travaux de C.A.R. Hoare [Hoa04] et R. Milner [Mil89], les algèbres de processus sont considérées comme des langages dont les termes représentent des processus. Elles comprennent un nombre restreint d'opérateurs algébriques qui, par combinaison, permettent de décrire directement et succinctement le parallélisme des systèmes concurrents. La sémantique opérationnelle de ces langages de spécification est généralement décrite en termes de systèmes de transitions [Arn92].

A l'origine, les algèbres de processus sont plutôt dédiées à la modélisation des systèmes asynchrones (CSP [Hoa04], FSP [MK06], CCS [Mil89], etc.) mais elles se sont diversifiées en de nombreuses variantes comme les algèbres de processus synchrones, les algèbres temporisées ou encore les algèbres probabilistes.

5 Les LTS

Les formalismes à états/transitions reposent à l'origine sur la théorie des graphes. Ils comportent des formalismes tels que les Statecharts, les réseaux de Petri ou les nombreuses variantes des automates à états finis. Ces formalismes reposent sur des structures de types systèmes de transitions [Arn92] telles que les structures de Kripke dans lesquelles les états sont étiquetés par un ensemble de propositions (propriétés) ou les systèmes de transitions étiquetées (ou LTS pour Labelled Transition Systems) [Arn92] dans lesquelles ce sont les

transitions qui sont étiquetées. Ces derniers sont couramment utilisés pour spécifier et analyser formellement le fonctionnement des systèmes logiciels réactifs et concurrents, ou pour décrire des sémantiques opérationnelles.

Les LTS sont constitués de l'ensemble des états dans lesquels peuvent se trouver le système et de l'ensemble des transitions pouvant être exécutées par ce dernier. A chaque transition est associée une étiquette qui représente l'action atomique qui provoque le franchissement de cette transition :

Formellement, un LTS est un quadruplet :

$$P = (S, \Sigma, \Delta, q)$$

- S est l'ensemble des états,
- Σ est l'alphabet de P c'est-à-dire l'ensemble des actions pouvant être exécutées,
- Δ est l'ensemble des transitions étiquetée avec les éléments de Σ. Elle associe à toute transition l'état d'origine et l'état cible de la transition,
- q est l'état initial de P.

Le fonctionnement de la plupart des composants d'un système peut être modélisé par un LTS. Des LTS, modélisant différents composants, peuvent être combinés par mise en parallèle de ces comportements grâce à deux opérations fondamentales : le produit libre et le produit synchronisé.

Le produit libre (noté $||$) : il s'agit d'une composition, sans contraintes, des LTS qui modélisent les composants d'un système, permettant ainsi de décrire l'ensemble des exécutions possibles. Dans ce contexte, le nombre d'états du LTS global correspond au produit du nombre d'états de chaque LTS et le nombre de transitions du LTS global est égal à la somme des produits du nombre de transitions avec le nombre d'états de chaque LTS.

Le produit synchronisé (noté $||_s$) : le fait que les composants interagissent nécessite de faire intervenir des contraintes de communication et de synchronisation. Ces contraintes peuvent être représentées simplement et formellement sous la forme de vecteurs de synchronisations (ensemble des actions pouvant se produire simultanément entre les différents composants). Le produit synchronisé introduit par Arnold et Nivat [Arn92] permet de construire le système global à partir du produit libre des LTS et des contraintes de communications. Le LTS résultant, modélisant le comportement global du système, est alors un sous-ensemble du produit libre dans lequel les seules exécutions possibles sont celles conformes aux contraintes de synchronisation.

6 Techniques de vérification formelle

Les techniques de vérification formelle servent à démontrer de manière rigoureuse que le système spécifié est conforme aux attentes du concepteur ; ces attentes étant exprimées sous la forme d'un ensemble de propriétés. Une première approche consiste à utiliser des techniques de preuve pour démontrer que les propriétés sont satisfaites. Une deuxième

approche, le Model-Checking [CGP00], est une approche algorithmique qui calcule la relation de satisfaction entre le comportement modélisé M et les propriétés attendues K tel que ($M \models K$).

Le processus de vérification par Model-Checking s'effectue en trois étapes :

1. *La modélisation du système* : Une première étape consiste à spécifier le modèle comportemental du système considéré à l'aide de spécifications formelles, dans un langage de modélisation accepté par l'outil de Model-Checking choisi.
2. *La spécification des propriétés* : Une deuxième étape consiste à décrire les propriétés devant être vérifiées par le modèle. Celles-ci sont couramment exprimées à l'aide de logiques temporelles. Cinq types de propriétés peuvent être décrits (sûreté, vivacité, équité, absence de blocage et atteignabilité). Cependant, les propriétés de sûreté (quelques chose de mal ne se produira jamais) et de vivacité (quelque chose de bien finira toujours par se produire) sont les plus utilisées pour la vérification des systèmes critiques [SLB99].
3. *L'utilisation d'un algorithme* : L'étape de vérification consiste à utiliser un algorithme de Model-Checking permettant de détecter, à partir d'un modèle et d'un ensemble de propriétés désirées, s'il existe des exécutions du modèle qui violent l'une de ces propriétés. Dans le cas où une violation est détectée, l'algorithme (outil) fournit une trace ayant menée à la violation. Cette trace (aussi appelée contre exemple) peut être utilisée pour aider à dépister l'endroit et la manière dont s'est produite la violation afin de la corriger plus facilement. Ces algorithmes de Model-Checking sont incarnés au sein d'outils spécifiques qui se chargent de cette tâche de manière automatique. Ainsi, l'outil LTSA [MK06] combine l'algèbre de processus FSP et la logique temporelle LTL pour spécifier et vérifier des modèles.

7 Conclusion

Bien que les méthodes formelles aient permis d'améliorer la qualité des logiciels, l'utilisation de formalismes souvent difficiles à appréhender, limite leur utilisation.

En raison de sa facilité d'utilisation, le Model-Checking est devenu plus populaire et est maintenant utilisé avec succès y compris dans l'industrie pour la vérification de systèmes critiques de grande taille.

Malgré ses avantages, le Model-Checking souffre de certaines limites :
– La construction des modèles et des propriétés reste une tâche difficile. Il est impossible de déterminer si les spécifications données couvrent toutes les propriétés que le système doit satisfaire.
– Il ne vérifie pas le système lui-même mais son modèle ; ainsi en cas de mauvaise modélisation du système (ou des propriétés) l'application finale peut tout de même contenir un certain nombre de risques de dysfonctionnements.
– La représentation de toutes les situations possibles conduit à une explosion combinatoire des états. Ce phénomène entraîne un dépassement des capacités de stockage ou de calcul.

Annexe B

Commande par supervision

1 Introduction

L´ des Systèmes à événements Discrets (SED, [CL99]) introduite par P.J.Ramadge et W.M.Wonham [RW89] se base sur la théorie des langages et des automates. Elle définit l'utilisation de trois entités : le système à commander, une spécification du comportement souhaité et un superviseur. Cette théorie offre alors la possibilité de synthétiser de manière automatique, le modèle d'un superviseur à partir d'un modèle du SED et de la spécification.

Deux objectifs sont envisageables :

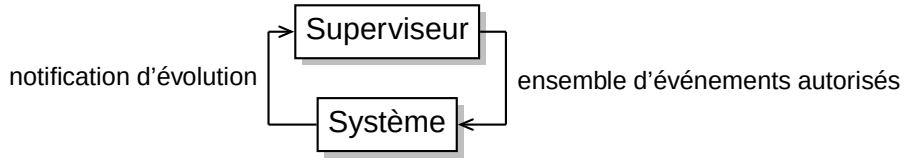
1. Fiabiliser un système susceptible d'avoir un comportement qui peut être gênant ou dangereux.
2. Permettre, pour un système dit ouvert dont les comportements possibles sont multiples, de réaliser différents superviseurs dont les rôles sont de contraindre le système dans un comportement donné pour un contexte donné.

2 Principe de la supervision

La théorie de la supervision s'appuie sur la théorie des langages et des automates pour modéliser les trois entités : le système, la spécification et le superviseur. Ceci permet de bénéficier des opérations liées aux langages et aux automates à états finis puisque les langages utilisés sont des langages réguliers[1]. Par la suite, les modèles de ces entités sont des automates notés G pour le système, K pour la spécification et S pour le superviseur, leurs langages respectifs sont notés $L(G)$, $L(K)$ et $L(S)$.

Le superviseur, une fois réalisé, est composé avec le SED pour former le système supervisé. A chaque changement d'état du système, le superviseur va suivre l'évolution du système et lui notifier l'ensemble des événements qu'il est autorisé à générer (figure B.1).

[1] Langage accepté par un automate à états finis.



F . B.1 – Principe de la commande par supervision

Deux situations, prises en compte par la théorie, peuvent survenir. La première est que certains événements, par nature, ne peuvent pas être interdits. La seconde situation est que certaines transitions du système ne peuvent être vues par le superviseur, l'empêchant de fournir un ensemble d'événements autorisés cohérent. Ces deux situations sont définies dans la théorie de la supervision sous les notions de *commandabilité* et *observabilité* des événements.

Ainsi, l'alphabet Σ des automates est divisé en plusieurs sous-ensembles notés Σ_c pour l'alphabet des événements commandables, Σ_{uc} l'alphabet des événements non commandables, Σ_o l'alphabet des événements observables et Σ_{uo} l'alphabet des événements non observables tel que $\Sigma = \Sigma_c \cup \Sigma_{uc}$ et $\Sigma = \Sigma_o \cup \Sigma_{uo}$ ce qui signifie qu'un événement peut être à la fois non commandable et non observable, commandable et non observable, observable mais non commandable ou enfin commandable et observable. Ces notions de commandabilité et d'observabilité des événements sont très importantes car elles vont conditionner les limites du superviseur et sa réalisation. De nombreux travaux approfondissent ces notions dans [RW89], [GKM91], [KS98], [Flo04].

Pour la réalisation du système supervisé, la théorie de la supervision définit que le modèle du superviseur, donc le langage $L(S)$, peut être concrétisé par une fonction ou une table d'association qui pour une séquence d'événements donnée renvoie une liste d'événements autorisés. Une autre approche utilise l'intersection (ou le produit strict pour les automates) du langage du système et celui du superviseur afin d'obtenir le langage du système supervisé, $L(S/G)$. Ainsi, le modèle du système supervisé est obtenu par :

$$L(S/G) = L(G) \cap L(S)$$

$$Lm(S/G) = Lm(G) \cap Lm(S)$$

où Lm est le langage marqué de L tenant compte du marquage de certains états. La théorie de la supervision définit enfin la notion de superviseur bloquant si l'égalité suivante n'est pas vérifiée :

$$L(S/G) = Lm_{pref}(S/G)$$

où L_{pref} est le langage de L auquel est appliquée la fermeture préfixielle[2].

[2]Augmentation du langage avec tous les préfixes des mots contenus dans le langage y compris ϵ, la chaîne vide.

3 Définition d'une spécification

La définition d'une spécification est l'opération la plus difficile pour la théorie de la supervision car celle-ci doit répondre à différents critères afin d'être utilisable pour la génération du superviseur [OV94]. Le but de la spécification est de définir le comportement du système supervisé (S/G). Ainsi, si K est un automate décrivant une spécification valide, il faut avoir :

$$L(K) = L(S/G)$$

La théorie de la supervision ne fait pas état d'une méthode particulière pour l'obtention d'une spécification, son objectif se limitant aux aspects théoriques liés à la synthèse d'un superviseur. Cependant, différentes méthodes sont envisageables pour l'obtention d'une spécification :

- Une première méthode consiste en la définition d'une spécification par un automate E décrivant le comportement souhaité et prenant en compte tous les événements présents dans le système. La spécification finale est réalisée à l'aide du produit synchronisé, soit $K = E\|_s G$ pour limiter ce comportement au comportement faisable par le système.
- Une deuxième méthode permet de définir le comportement souhaité en ne considérant que certains événements. Cette spécification E doit être augmentée afin de correspondre au système. L'opération de produit synchronisé permet là aussi d'obtenir cette spécification. Cette méthode a pour avantage d'être plus simple mais elle implique une attention particulière aux événements non utilisés pour la définition de E à cause du produit synchronisé qui va augmenter son langage.
- Une troisième approche consiste à définir des automates qui invalident une ou plusieurs séquences (ou traces) données. Ainsi, les automates sont réalisés pour modéliser les séquences dont tous les états sont marqués sauf les derniers représentant la survenue de l'événement qui rend l'automate bloquant.

4 Synthèse d'un superviseur

La spécification étant définie, la synthèse d'un superviseur selon [RW89], [CL99] s'effectue par le produit du système G et de la spécification K suivi de l'opération $Trim$ (permettant de supprimer les états inaccessibles et bloquants) :

$$S = Trim(G \times K)$$

Ceci est la base de la démarche pour la synthèse d'un superviseur mais qui ne prend pas en compte les aspects liés à la commandabilité et à l'observabilité des événements. Elle s'explique par la relation :

$$L(S/G) = L(K) = L(G) \cap L(S)$$

Cette relation signifie que la spécification est l'objectif du système supervisé par l'égalité $L(S/G) = L(K)$. Elle signifie également que le langage du superviseur ne se limite pas nécessairement au langage de la spécification.

Les notions de *commandabilité* et *observabilité* des événements introduisent des particularités dans la relation entre le système et le superviseur lors de la supervision. Les événements non commandables ne peuvent pas être interdits par le superviseur et le système sera toujours susceptible de les réaliser. Les événements non observables ne permettent pas au superviseur de suivre l'évolution du système de manière cohérente. Le superviseur est alors susceptible après la survenue d'un événement non observable de fournir une commande erronée.

Pour prendre en compte les particularités liées aux événements non commandables, la spécification est alors l'élément sur lequel vont intervenir les éventuelles corrections pour que la synthèse donne un superviseur correct. La théorie de la supervision définit alors une spécification comme étant commandable si

$$P(L_{pref}(K), \Sigma_{uc}) \cap L(G) \subseteq L_{pref}(K)$$

où l'opérateur P est l'opérateur de projection d'un automate selon un alphabet permettant de ne considérer que le langage de l'automate réduit à l'alphabet considéré.

Si la spécification n'est pas commandable, alors la théorie de la supervision définit qu'il existe un langage suprême commandable $L(K') \subset L(K)$ qui est commandable. Ce langage est obtenu à l'aide de l'algorithme de Kumar [KS98]. Cet algorithme identifie dans l'automate K les états à partir desquels, dans le système, des événements non commandables sont susceptibles d'être générés.

Lors de la supervision, à chaque changement d'état du système, le superviseur, notifié de l'événement qui vient d'engendrer ce changement d'état, est mis à jour afin de produire un nouvel ensemble d'événements autorisés. Cependant, lors de la survenue d'un événement non observable, le superviseur n'est pas notifié et n'est pas capable de fournir un ensemble d'événements autorisés cohérent. Pour cela, la théorie de la supervision impose que la commande souhaitée pour le système soit la même avant et après un événement non observable sinon la spécification risque d'être violée malgré le processus de supervision.

Si une spécification K n'est pas observable alors, contrairement à la notion de commandabilité, il n'existe pas de langage suprême observable qui puisse être calculé. Dans pareil cas, il est nécessaire de se rattacher à la notion de *normalité*.

La *normalité*, c'est l'invariance d'un langage par rapport à un autre suivant un alphabet donné. La normalité permet de vérifier qu'un langage prend en compte tous les événements contenus dans un alphabet pour un autre langage. Dans le cadre de la théorie de la supervision, la normalité est rapprochée de l'observabilité en définissant l'alphabet intéressant comme étant celui des événements non observables. Ainsi, la spécification K est définie comme normale pour le système G et pour les événements non observables si :

$$L_{pref}(K) = L(G) \cap IP(P(L(K), \Sigma_o), \Sigma_{uo})$$

Où IP est l'opérateur de projection inverse permettant de compléter un automate avec des comportements supplémentaires issus d'un alphabet particulier.

La notion de normalité décrite dans [RW89] et [Lam02] est très pratique car elle permet d'assimiler la notion d'observabilité. En fait, si un langage est normal alors il est observable et il est possible, comme pour la commandabilité, de construire un langage suprême normal (cf. algorithme de Kumar [KS98]) si K n'est pas normal. Il faut tout de même ajouter quelques restrictions, il faut considérer les événements non observables comme étant non commandables, ainsi il y a équivalence entre observabilité et normalité.

5 Outils du domaine

Pour la mise en œuvre de la théorie de la commande par supervision, de nombreux outils tels TCT, UMDES, UKDES, J-DES, BSP, Ver, Valid ont été développés. Tous originaires du monde universitaire ou de la recherche, ces outils permettent de modéliser et de synthétiser des superviseurs à partir de modèles de systèmes et de spécifications définies sous la forme d'automates à états fini déterministes. Ils ne fournissent cependant pas un environnement de développement adapté à une mise en œuvre au sein d'un processus de développement logiciel.

Dans le cadre de ce travail de thèse, l'outil Supremica [AFF03] a été employé car il fournit à l'utilisateur un environnement simple pour la réalisation de modèles de superviseur. Comme ses semblables, il se base sur les automates à états finis déterministes et offre différents types de produits d'automates pour permettre les diverses manipulations de modèles propre au domaine.

Supremica permet également de vérifier la commandabilité et l'observabilité des superviseurs ainsi générés. De plus il offre la possibilité de générer à partir des modèles réalisés divers types de code sources selon les besoins : du Java, du langage dot (utilisable avec l'outil Graphviz), du Ladder, etc. Enfin, Supremica offre également l'intérêt de permettre la simulation des modèles réalisés afin de détecter d'éventuelles erreurs de conception.

6 Conclusion

La théorie de la commande par supervision est une approche intéressante dans la mise en œuvre de contrôleurs pour les systèmes discrets. En effet, elle donne un cadre théorique idéal pour la modélisation et la commande de systèmes à événements discrets par la génération d'un superviseur valide en prenant en compte de nombreux aspects susceptibles de poser des problèmes tels que ceux liés aux événements non commandables ou non observables.

Pour cela, les notions de commandabilité, d'observabilité et de normalité de la spécification permettent de mettre en place des algorithmes spécifiques fournissant alors des superviseurs adaptés aux conditions imposées par le système. Cependant, comme pour le Model-Checking, cette approche souffre de la difficulté à définir des spécifications conformes au cahier des charges.

Bibliographie

[AD05] L. Apvrille and P. De Saqui-Sannes. Turtle : a uml-based environment for the codesign of embedded systems. In *Proceedings of the 8th Sophia-Antipolis MicroElectronics Forum (SAME 2005)*, October 2005.

[AFF03] K. Akesson, M. Fabian, H. Flordal, and A. Vahidi. Supremica- a tool for verification and synthesis of discrete event supervisors. In *The 11th Mediterranean Conference on Control and Automation*, 2003.

[AJ02] S.W. Ambler and R. Jeffries. *Agile Modeling : Effective Practices for Extreme Programming and the Unified Process*. Wiley, 2002.

[All97] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantiques des processus communicants*. Masson, Paris, France, 1992.

[BBF04] J. Bézivin, M. Blay-Fornarino, M. Bouzhegoub, J. Estublier, J-M. Favre, S. Gérard, and J-M. Jézéquel. *Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture)*. CNRS, 2004.

[BCK03] Len Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2 edition, April 2003.

[Béz04] J. Bézivin. In search of a basic principe for model driven engineering. *The European Journal for the Informatics Professional*, 2 :21–24, 2004.

[BFS05] J-P. Bodeveix, M. Filali, and M. Strecker. Towards formalising AADL in proof assistants. *Electron. Notes Theor. Comput. Sci.*, 141(3) :153–169, 2005.

[BMM95] J. Bosch, P. Molin, M. Mattsson, and P. Bengtsson. *Object-Oriented Frameworks – Problems and Experiences*, chapter 4. Wiley and Son, 1995.

[BMN07] G. Booch, R. A. Maksimchuk, J. Newkirk, B. Young, M. Engel, A. Brown, J. Conallen, and K. Houston. *Object-Oriented Analysis and Design With Applications*. Addison-Wesley Educational Publishers In, 3 edition, 2007.

[Boe81] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall advances in computing science & technology series. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Boe88] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21 :61–72, 1988.

[BSM03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework : A Developer's Guide*. The Eclipse Series. Addison Wesley Professional, 2003.

[CES86] E. Clarke, E. Emmerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CGP00] E-M Clarke, O. Grumberg, and D. Peled. *Model-Checking*. The MIT Press, New York, 2000.

[CL99] C. G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Springer, New York, 1999.

[Col05] T. Collonville. Approche orientée modèles et commande par supervision appliquée à la synthèse de systèmes logiciels critiques. Master's thesis, Université de Haute Alsace, Laboratoire MIPS, 2005.

[CTT07] T. Collonville, L. Thiry, and B. Thirion. Idm pour une approche combinant synthèse et vérification de modèles. *Revue en ligne E-Sta Spécial JD-JN-MACS 07*, 4(4), 2007.

[Dis04] P. Dissaux. Using the AADL for mission critical software development. In *ERTS conference*, 2004.

[DLC09] S. Diaw, R. Lbath, and B. Coulette. SPEM4MDE : un métamodèle basé sur SPEM2 pour la spécification des procédés MDE (regular paper). In *MAnifestation des Jeunes Chercheurs STIC (MajecStic), Avignon, 16/11/09-18/11/09*, page (support électronique), http ://lia.univ-avignon.fr/, novembre 2009. Laboratoire Informatique d'Avignon.

[EEK08] K. El-Emam and G-A. Koru. A replicated survey of IT software project failures. *IEEE Software*, 25(5) :84–90, 2008.

[EFH07] S. Efftinge, P. Friese, A. Haase, C. Kadura, B. Kolb, D. Moroff, K. Thoms, and M. Völter. *OpenArchitectureWare User Guide*, 2007.

[EHS06] D.S. Evans, A. Hagiu, and R. Schmalensee. *Invisible Engines : How Software Platforms Drive Innovation and Transform Industries.* MIT Press. 2006.

[EJL03] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1) :127–144, January 2003.

[EPF] EPF. Eclipse process framework. http ://www.eclipse.org/epf/

[Est06] J. Estublier. *Unifying the Software Process Spectrum*, volume 3840 of *Lecture Notes in Computer Science*, chapter Software are Processes Too, pages 25–34. Springer Berlin / Heidelberg, 2006.

[Ext01] *Extreme Chaos*, 2001.

[Fav04] J-M. Favre. Towards a basic theory to model model driven engineering. In *WISME 2004*, 2004.

[FEB06] J-M. Favre, J. Estublier, and M. Blay-Fornarino. *L'Ingénierie Dirigée par les Modèles au-dela du MDA.* Hermes, Paris, 2006.

[Flo04] H. Flordal. *Modular Verification and Synthesis of Discrete Event Systems*. PhD thesis, Chalmers University of Technology, 2004.

[Gab06] H.A. Gabbar. *Modern Formal Methods and Application*. Springer verla edition, 2006.

[GCC09] A. Garcia, B. Combemale, X. Crégut, and J. Vandeur. topPROCESS : vers une ingénierie des procédés dirigée par les modèles. *Revue de l'Electricité et de l'Electronique*, 02, February 2009.

[GKM91] V. K. Garg, R. Kumar, and S. I. Marcus. On controllability and normality of discrete event dynamical systems. *Systems and Control Letters*, 13 :157–168, 1991.

[GM03] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems, 2003.

[GMF09] Gmf (graphical modeling framework) home page. http ://www.eclipse.org/modeling/gmf/, 2009.

[GMT04] S. Gérard, C. Mraidha, F. Terrier, and B. Baudry. A UML-based concept for high concurrency : The real-time object. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0 :64–67, 2004.

[GMW97] D. Garlan, R-T. Monroe, and D. Wile. Acme : An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[GPM08] M. Göthe, C. Pampino, P. Monson, K. Nizami, K. Patel, B. M.Smith, and N. Yuce. *Collaborative Application Lifecycle Management with IBM Rational Products*. An IBM Redbooks publication. RedBooks, 2008.

[GW03] E. Gamma and J. Wiegand. *Contriguting to eclipse*. Addison Wesley, 2003.

[HFR09] C Hug, A. Front, D. Rieu, and B. Henderson-Sellers. A method to build information systems engineering process metamodels. *Journal of Systems and Software*, 82(10) :1730–1742, 2009. SI : YAU.

[HK89] W.-S. Humphrey and M. I. Kellner. Software process modeling : principles of entity process models. In *ICSE '89 : Proceedings of the 11th international conference on Software engineering*, pages 331–342, New York, NY, USA, 1989. ACM.

[Hoa04] C. A. R. Hoare. *Communicating Sequential Processes*. Jim Davies, 2004.

[Hon09] Inc. Honeywell. Dome home page, 2009. http ://www.htc.honeywell.com/dome/

[HS06] T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*. Springer, Springer, August 2006.

[Hug06] C Hug. *Méthode, modèles et outil pour la méta-modélisation des processus d'ingénierie de systèmes d'information*. PhD thesis, Thèse de l'Université Joseph Fourier, Grenoble, Octobre 2006.

[JB06] F. Jouault and J. Bézivin. Km3 : a dsl for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems,*, pages 171–185, Italy, 2006.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Publishing, 1999.

[JCJ92] I. Jacobson, M. Christerson, P. Jonsson, and G. Oevergaard. *Object Oriented Software Engineering : a Use Case Driven Approach*. Addison-Wesley, 1992.

[JJ04] J-M. Jézéquel and J. Julliand. *Approches formelles pour le développement de logiciels*, volume 23 of *Technique et science informatiques RSTI, série TSI*. Lavoisier edition, 2004.

[JK05] F. Jouault and I. Kurtev. Transforming models with ATL. *Lecture Notes in Computer Science*, 3844 :128–138, 2005. Satellite Events at the MoDELS 2005 Conference.

[Ken96] J.J. Kenney. *Executable Formal Models of Distributed Transaction Systems based on Event Processing*. PhD thesis, Stanford University, 1996.

[KR70] W. Kunz and H. Rittel. Issues as elements of information systems. Technical report, 1970.

[Kru08] P. Kruchten. What do software architects really do ? *Journal of Systems and Software*, 81(12) :2413–2416, December 2008.

[KS98] R. Kumar and M. A. Shayman. Formulae relating controllability, observability and co-observability. *Automatica*, 34 :211–215, 1998.

[KSL03] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1) :145–164, Jan. 2003.

[Lam02] M. H. Lamouchi. *Synthèse de Superviseur pour des Systèmes à Evènements Discrets Partiellement Observés*. PhD thesis, Département de Génie Electrique et de Génie Informatique de l'Ecole Polytechnique de l'Université de Montréal, 2002.

[Lar93] P. De Larminat. *Automatique – Commande des systèmes linéaires*. Hermès, Paris, 1993.

[LG97] Luqi and J.A. Guoguen. Formal methods : Promises and problems. *In IEEE Software*, 75(1) :75–85, 1997.

[LMB01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *IEEE International Workshop on Intelligent Signal Processing*, 2001.

[Mar91] J. Martin. *Rapid Application Development*. Macmillan Coll, New York, 1991.

[Mat05] Mathworks. The mathworks home page, 2005. http ://www.mathworks.com.

[Mes07] V. Messager Rota. *Gestion de projet, Vers les methodes agiles*. Eyrolles, 1ére edition, 2007.

[Met05] MetaCase. Domain specific modeling with metaedit+. Technical report, MetaCase, 2005.

[MFF06] P-A. Muller, F. Fleurey, F. Fondement, M. Hassenforder, R. Schneckenburger, S. Gérard, and J-M. Jézéquel. Model-driven analysis and synthesis of concrete syntax. In *MoDELS*, pages 98–110, 2006.

[MFJ05] P-A. Muller, F. Fleurey, and J. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML'2005*, pages 264–278, 2005.

[MHL07] C. Morley, J. Hugues, B. Leblanc, and O. Hugues. *Processus métiers et Systèmes d'Information*. Dunod, Paris, 2007.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MK06] J. Magee and J. Kramer. *Concurrency : state models & java programs*. John Wiley & Sons, Chichester, 2eme edition edition, April 2006.

[MNE95] J. Magee, N.Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. of 5th European Software Engineering Conference (ESEC '95)*, pages 137–153, September 1995.

[Moi90] J.L.Le Moigne. *La modélisation des systèmes complexes*. Dunod, Paris, 1990.

[MR84] J. McDermid and K. Ripken. *Life cycle support in the ADA environment*. University Press, 1984.

[MS03a] D.G. Messerschmitt and C. Szyperski. *Software Ecosystem. Understanding an Indispensable Technology and Industry.* MIT Press. 2003.

[MS03b] L. Motus and B. Selic. Using models in real-time software design. *IEEE Control Systems Magazine*, 23 :43–60, 2003.

[OMG02] OMG. *Software Process Engineering Metamodel Specification*, 2005. Version 1.0, http ://www.omg.org/cgi-bin/doc ?formal/02-11-14 , Novembre 2002.

[OMG06] OMG. *Meta Object Facility (MOF) Core Specification*. Version 2.0, http ://www.omg.org/spec/MOF/2.0/ , Janvier 2006.

[OMG08a] OMG. *Software & Systems Process Engineering Meta-Model Specification*. Version 2.0, http ://www.omg.org/spec/SPEM/2.0/ , Avril 2008.

[OMG08b] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification* Version 1.0, http ://www.omg.org/spec/QVT/1.0/ , Avril 2008.

[OMG10a] OMG. *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure* Version 2.3, http ://www.omg.org/spec/UML/2.3/ , Mai 2010.

[OMG10b] OMG. *OMG Unified Modeling LanguageTM (OMG UML), Superstructure* Version 2.3, http ://www.omg.org/spec/UML/2.3/ , Mai 2010.

[OMG10c] OMG. *Object Constraint Language*. Version 2.2, http ://www.omg.org/spec/OCL/2.2/ ,Février 2010.

[OMG10d] OMG. *OMG SysML Specification*. Version 1.2, http ://www.omg.org/spec/SysML/1.2/ , Juin 2010.

[Ost87] L. Osterweil. Software processes are software too. In *ICSE '87 : Proceedings of the 9th international conference on Software Engineering*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[OV94] A. Overkamp and J. H. Van Schuppen. Control of discrete event systems – research at the interface of control theory and computer science. In *From Universal morphisms to megabytes : a Baayen space Odyssey*, pages 453–467, Stichting Mathematisch Centrum, 1994.

[Per07] J-M. Perronne. *Une contribution Objet pour la conception de systèmes logiciels de commande plus sûrs*. Habilitation à diriger des recherches. Université de Haute-Alasace, 2007.

[Pnu97] A. Pnueli. The temporal logic of programs. In *In Proceedings of 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57, 1997.

[PPT02] J-M. Perronne, C. Petitjean, L. Thiry, and M. Hassenforder. A framework for advanced fuzzy logic inference systems. In *In proceeding of IFAC'2002*, Barcelone, Juillet 2002.

[Ras06] A. Rasse. *Approche Orientée Modèles pour la Spécification, la Vérification, l'Implantation des Systèmes Logiciels Critiques*. PhD thesis, Université de Haute Alsace, Laboratoire MIPS, 2006.

[RK03] P. N. Robillard and P. Kruchten. *Software Processes with the Unified Process for Education (UP/EDU)*, chapter A Process Engineering Metamodel. Addison wesley longman edition, 2003.

[Roc07] S. Rochet. *Formalisation des processus d'ingénierie système : Proposition d'une méthode d'adaptation des processus génériques à différents contextes d'application*. PhD thesis, INSA, Toulouse, Novembre 2007.

[Rol05] C. Rolland. L'ingénierie des méthodes : une visite guidée. *La revue e-TI. La revue electronique des technologies de l'information*, 25 octobre 2005.

[Ros75] J.De Rosnay. *Le macroscope – Vers une vision globale*. Edition du Seuil, Paris, 1975.

[Roy70] W.W. Royce. Managing the development of large software systems. *IEEE Wescon*, pages 1–9, 1970.

[RSM95] C. Rolland, C. Souveyet, and M. Moreno. An approach for defining ways-of-working. *Inf. Syst.*, 20(4) :337–359, 1995.

[RV02] P. Roques and F. Vallée. *UML en action*. Eyrolles, 2eme edition, 2002.

[RW89] P.J. Ramadge and W.M. Wonham. The control of discrete-event systems. *IEEE Transactions on Automatic Control*, 77(1) :81–98, 1989.

[SA03] R. Sanz and K-E. Arzen. Trends in software and control. *IEEE Control Systems Magazine*, 23 :12–15, 2003.

[SB01] K. Schwaber and M. Beedle. *Agile Software Development with SCRUM*. Upper Saddle River, New Jersey, 2001.

[SLB99] P. Schnoebelen, F. Laroussinie, M.l Bidoit, B. Bérard, and A. Petit. *Verification de Logiciels : Techniques et Outils du Model-Checking*. Vuibert, Paris, 1999.

[Som06] I. Sommerville. *Software Engineering 8th Revised edition*. Addison-Wesley Educational Publishers Inc, United Kingdom, 2006.

[SR67] J.L. Shaerer and H.H. Richardson. *Introduction to System Dynamics*. Addison Wesley, NewYork, 1967.

[Sta95] The Standish Group. *Chaos Report 95*, 1995.

[TCT08] L. Thiry, T. Collonville, and B. Thirion. Un framework fonctionnel pour la modélisation et la simulation informatique de systèmes complexes. In *7ème Conf. Int. de Modélisation et Simulation*, MOSIM'08, page 10, Paris, France, 31 Mars-2 Avril 2008.

[Top05] TopCaseDTeam. Topcased specification and architecture. Technical report, 2005.

[TS04] J. Tomayko and G. Scragg. *Human Aspects of Software Engineering*. Charles River Media, Hingham, Masschusetts, 1ére edition, 2004.

[TT08] L. Thiry and B. Thirion. Functional metamodels for the development of control software. In *IFAC WC 2008*, page 6, 6-11 Juillet 2008.

[TT09] L. Thiry and B. Thirion. Functional metamodels for systems and software. *Journal of Systems and Software*, 82(7) :1125–1136, 2009.

[TTC08] L. Thiry, B. Thirion, and T. Collonville. Un framework pour la conception intégrée de commandes logicielles. In *CIFA'08*, Bucarest (Roumanie), 3-5 septembre 2008.

[TTH08] L. Thiry, B. Thirion, and M. Hassenforder. Dsls pour le développement agile de transformations. In *IDM'08*, page 16, Mulhouse, 5-6 Juin 2008.

[Zay03] J. Zaytoon. *Systèmes Dynamiques Hybrides*. Hermes Science. Paris, 2003.

Table des figures