

# Adaptation logicielle pour et par les *i-DSML*

## THÈSE

présentée et soutenue publiquement le 20 novembre 2015

pour l'obtention du

Doctorat de l'Université de Pau et des Pays de l'Adour

par

**Samson Pierre**

### Composition du jury

*Président* : Pr. Xavier Blanc

*Rapporteurs* : Pr. Olivier Barais  
Pr. Lionel Seinturier

*Examineurs* : Dr. Éric Cariou  
Dr. Olivier Le Goaër

*Directeur* : Pr. Franck Barbier

Mis en page avec la classe thesul.

## Remerciements

Au terme de ces trois années de thèse passées au Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour (LIUPPA), c'est avec plaisir que j'écris cette page de remerciements.

Tout d'abord, je souhaite remercier les membres de mon jury pour avoir jugé le résultat de mes travaux suffisamment pertinent pour me permettre de soutenir ma thèse. À cette occasion, j'associe aussi ces remerciements aux personnes présentes lors de la soutenance.

Naturellement, je remercie Franck Barbier, mon directeur, pour m'avoir fait confiance en m'acceptant sous sa direction afin que je puisse travailler sur ma thèse. Il m'a toujours encouragé ce qui m'a conforté dans l'idée que mes travaux aboutiront. De plus, il m'a donné des conseils avisés quant à la rédaction de mon manuscrit.

Ensuite, j'exprime ma gratitude envers Éric Cariou et Olivier Le Goaër, mes deux encadrants, avec qui nous avons effectué un grand nombre de réunions. Ils ont suivi régulièrement mon travail afin de vérifier que je ne partais pas dans une mauvaise direction. D'autre part, ils ont lu et relu mon manuscrit, puis m'ont signalé les passages qui étaient à revoir.

Par ailleurs, je tiens aussi à remercier toutes les autres personnes sympathiques avec qui j'ai eu l'occasion de discuter durant cette thèse. Je pense particulièrement à Carole Harcaut et Régine Dufaur-Dessus, les secrétaires, Christophe Bellocq et Laurent Lacayrelle, les techniciens, Nicolas Belloir, Mauro Gaio, Éric Gouardères, Nabil Hameurlain, Bruno Jobard, Annig Lacayrelle, Wilfrid Lefer et Congduc Pham, les chercheurs, Nour Aboud, Gaëtan Deltombe, Mamour Diop, Ludovic Moncla et Ehsan Muhammad, les doctorants.

En outre, je remercie également les membres de ma famille avec qui j'ai gardé le contact malgré la distance physique qui nous séparait. Pouvoir discuter avec eux par téléphone ou par Internet durant cette période a sans aucun doute contribué à ce que je garde le moral. En plus, ils ont accepté généreusement de participer à la relecture de mon manuscrit de thèse.

Enfin, je souhaite remercier le lecteur de ce document, qui je l'espère appréciera son contenu et trouvera les informations qu'il cherche.



# Sommaire

Table des figures vii

Liste des tableaux ix

Listings xi

## Chapitre 1

### Introduction

- 1.1 Contributions de la thèse . . . . . 2
- 1.2 Plan du document . . . . . 3

## Chapitre 2

### État de l'art

- 2.1 Rappels sur l'ingénierie dirigée par les modèles . . . . . 5
  - 2.1.1 La pile de méta-modélisation selon l'*OMG* . . . . . 5
  - 2.1.2 Syntaxes concrètes . . . . . 7
  - 2.1.3 La manipulation de modèles . . . . . 7
- 2.2 L'exécution de modèles . . . . . 8
  - 2.2.1 Les modèles de processus hors IDM . . . . . 9
  - 2.2.2 L'exécution de modèles en IDM . . . . . 10
- 2.3 L'adaptation logicielle . . . . . 12
  - 2.3.1 Caractérisation de l'adaptation logicielle . . . . . 12
  - 2.3.2 Le contexte . . . . . 13
  - 2.3.3 Les politiques d'adaptation . . . . . 14
  - 2.3.4 Les boucles d'adaptation . . . . . 17
  - 2.3.5 Le *models@run.time* . . . . . 19
- 2.4 L'adaptation de l'exécution de modèles . . . . . 21
  - 2.4.1 Les modèles de processus adaptatifs hors IDM . . . . . 21
  - 2.4.2 L'adaptation de l'exécution de modèles en IDM . . . . . 24
- 2.5 Bilan . . . . . 26

**Chapitre 3****Caractérisation des *i-DSML* adaptables**

3.1	La caractérisation des <i>i-DSML</i> . . . . .	29
3.1.1	La nature exécutable des modèles . . . . .	30
3.1.2	Les moteurs d'exécution . . . . .	30
3.1.3	Les modèles exécutables auto-contenus . . . . .	31
3.1.4	La conception d'un <i>i-DSML</i> . . . . .	31
3.1.5	Les machines à états exécutables . . . . .	32
3.1.6	Un exemple de train . . . . .	35
3.1.6.1	Le contexte du train . . . . .	35
3.1.6.2	Un modèle simple de train . . . . .	35
3.1.6.3	L'exécution de la machine à états du train . . . . .	36
3.2	La caractérisation des <i>i-DSML</i> adaptables . . . . .	37
3.2.1	La conception d'un <i>i-DSML</i> adaptable . . . . .	37
3.2.1.1	Portée des sémantiques d'exécution et d'adaptation . . . . .	37
3.2.2	La partie adaptative pour le méta-modèle de machines à états . . . . .	39
3.2.3	Des machines à états spécialisées . . . . .	39
3.2.4	L'adaptation à l'exécution de la machine à états du train . . . . .	40
3.2.4.1	Les vérifications d'adaptation de la machine à états du train . . . . .	41
3.2.4.2	Les actions d'adaptation de la machine à états du train . . . . .	42
3.2.4.3	Les différentes sémantiques d'exécution de la machine à états du train . . . . .	43
3.3	Conclusion . . . . .	43

**Chapitre 4****Gestion de l'adaptation des *i-DSML* par les familles**

4.1	L'adaptation des <i>i-DSML</i> : un autre exemple . . . . .	46
4.1.1	La définition du méta-modèle <i>PDL</i> . . . . .	46
4.1.2	Un processus de développement logiciel défini en utilisant <i>PDL</i> . . . . .	49
4.1.3	De l'intérêt de la spécialisation pour l'adaptation . . . . .	50
4.2	La gestion de l'adaptation des <i>i-DSML</i> par les familles . . . . .	51
4.2.1	La définition d'une famille . . . . .	51
4.2.2	La spécialisation des familles . . . . .	51
4.2.3	Les politiques d'adaptation domaine et métier . . . . .	53
4.2.4	La définition du méta-modèle <i>FHDL</i> . . . . .	53
4.2.5	La représentation d'une hiérarchie de familles . . . . .	55
4.3	Une hiérarchie de familles pour <i>PDL</i> . . . . .	57
4.3.1	La description de la famille <i>PDL</i> . . . . .	57
4.3.2	La description de la famille <i>AdaptPDL</i> . . . . .	57
4.3.3	La description de la famille <i>SkipAdaptPDL</i> . . . . .	59
4.3.4	La description de la famille <i>DependAdaptPDL</i> . . . . .	61
4.3.5	La description de la famille <i>DependSkipAdaptPDL</i> . . . . .	63

4.3.6	La description de la famille <i>ManagedSkipAdaptPDL</i> . . . . .	65
4.4	Conclusion . . . . .	67

## Chapitre 5

### Orchestration de l'adaptation des *i-DSML*

5.1	Une architecture pour l'adaptation d' <i>i-DSML</i> . . . . .	69
5.2	Le méta-modèle d'orchestration de l'adaptation <i>ASDL</i> . . . . .	72
5.3	La représentation d'une orchestration de l'adaptation . . . . .	75
5.4	Des orchestrations de l'adaptation pour <i>PDLHierarchy</i> . . . . .	78
5.4.1	Une orchestration de l'adaptation pour <i>PDL</i> . . . . .	78
5.4.2	Une orchestration de l'adaptation pour <i>AdaptPDL</i> . . . . .	79
5.4.3	Une orchestration de l'adaptation pour <i>SkipAdaptPDL</i> . . . . .	80
5.4.4	Une orchestration de l'adaptation pour <i>DependAdaptPDL</i> . . . . .	82
5.4.5	Une orchestration de l'adaptation pour <i>DependSkipAdaptPDL</i> . . . . .	83
5.4.6	Une orchestration de l'adaptation pour <i>ManagedSkipAdaptPDL</i> . . . . .	85
5.5	Conclusion . . . . .	87

## Chapitre 6

### Implémentation

6.1	Un exemple d'implémentation d'un moteur d'exécution . . . . .	89
6.2	Le couplage au moteur d'orchestration . . . . .	92
6.3	Le fonctionnement du moteur d'orchestration . . . . .	94
6.3.1	Phase 1 : recherche des politiques d'adaptation . . . . .	96
6.3.2	Phase 2 : exécution d'une politique d'adaptation . . . . .	97
6.3.3	Un exemple du fonctionnement du moteur d'orchestration . . . . .	98
6.4	Conclusion . . . . .	99

## Chapitre 7

### Conclusion et perspectives

7.1	Résumé des contributions . . . . .	101
7.2	Perspectives . . . . .	102
7.2.1	L'outillage . . . . .	102
7.2.1.1	L'implémentation d'un éditeur de politiques d'adaptation . . . . .	102
7.2.1.2	L'automatisation du renommage des opérations à adapter . . . . .	103
7.2.2	L'extension du langage <i>ASDL</i> . . . . .	104
7.2.2.1	L'ajout d'opérateurs logiques à <i>ASDL</i> . . . . .	104
7.2.2.2	Une politique d'adaptation composite . . . . .	104
7.2.3	La méta-adaptation . . . . .	106

## Bibliographie

**Annexe A**

**Annexes**

A.1	La construction du moteur d'exécution d'un <i>i-DSML</i> . . . . .	117
A.1.1	La construction du moteur d'exécution d'un <i>i-DSML</i> avec <i>Kermeta</i> . . . . .	117
A.1.2	La construction du moteur d'exécution d'un <i>i-DSML</i> avec <i>Java/EMF</i> . . . . .	118
A.1.3	La construction du moteur d'exécution d'un <i>i-DSML</i> avec <i>ATL</i> . . . . .	119
A.2	Une étude de quelques moteurs d'exécution . . . . .	120
A.2.1	La construction d'un modèle pour <i>PauWare engine</i> . . . . .	120
A.2.2	La construction d'un modèle pour <i>Apache Commons SCXML</i> . . . . .	123
A.2.3	La construction d'un modèle pour <i>Renew</i> . . . . .	125
A.2.4	La construction d'un modèle pour <i>Tina</i> . . . . .	126
A.3	Le support du langage <i>SCXML</i> pour <i>PauWare engine</i> . . . . .	126

# Table des figures

2.1	La pile de méta-modélisation . . . . .	6
2.2	Les transformations de modèles endogènes . . . . .	8
2.3	Les transformations de modèles exogènes . . . . .	9
2.4	La boucle d'adaptation du cycle d'autogestion . . . . .	17
2.5	La boucle d'adaptation <i>MAPE</i> . . . . .	18
2.6	Les trois boucles d'adaptation de l'approche <i>ARCM</i> . . . . .	18
2.7	La boucle d'adaptation du <i>models@run.time</i> classique . . . . .	19
2.8	Une modification du modèle architectural embarqué à l'exécution . . . . .	20
2.9	Une modification du modèle de fonctionnalités embarqué à l'exécution . . . . .	21
2.10	Le réseau de Petri source . . . . .	22
2.11	Le réseau de Petri cible . . . . .	22
2.12	Le réseau de Petri d'adaptation . . . . .	23
2.13	Une modification du modèle d'un <i>workflow</i> adaptatif avec une « poche de flexibilité » . . . . .	24
2.14	Une modification du modèle d'un <i>workflow</i> adaptatif avec une liste d'ajouts/suppressions . . . . .	24
2.15	La boucle d'adaptation du <i>models@run.time</i> avec un modèle exécuté . . . . .	25
2.16	La boucle d'adaptation du <i>models@run.time</i> pour un modèle exécuté . . . . .	25
2.17	La boucle d'adaptation directe de l'exécution d'un modèle . . . . .	26
3.1	La conception d'un <i>i-DSML</i> . . . . .	32
3.2	Le méta-modèle de machines à états exécutables et adaptables . . . . .	33
3.3	Les signaux pour le train . . . . .	35
3.4	L'exécution de la machine à états d'un train . . . . .	36
3.5	La conception d'un <i>i-DSML</i> adaptable . . . . .	38
3.6	L'adaptation à l'exécution de la machine à états du train . . . . .	40
4.1	La définition du méta-modèle <i>PDL</i> . . . . .	47
4.2	Un modèle conforme au méta-modèle de <i>PDL</i> et son exécution . . . . .	49
4.3	La définition du méta-modèle <i>FHDL</i> . . . . .	54
4.4	Un exemple généraliste d'une hiérarchie de familles . . . . .	55
4.5	Ajout de la famille <i>PDL</i> à la hiérarchie de familles <i>PDLHierarchy</i> . . . . .	58
4.6	Ajout de la famille <i>AdaptPDL</i> à la hiérarchie de familles <i>PDLHierarchy</i> . . . . .	58
4.7	L' <i>i-DSML</i> correspondant à la famille <i>AdaptPDL</i> . . . . .	59
4.8	Un modèle conforme au méta-modèle d' <i>AdaptPDL</i> et son exécution . . . . .	59

TABLE DES FIGURES

---

4.9	Ajout de la famille <i>SkipAdaptPDL</i> à la hiérarchie de familles <i>PDLHierarchy</i> . . . . .	60
4.10	L' <i>i-DSML</i> correspondant à la famille <i>SkipAdaptPDL</i> . . . . .	61
4.11	Un modèle conforme au méta-modèle de <i>SkipAdaptPDL</i> , avant et après adaptation . . . . .	61
4.12	Ajout de la famille <i>DependAdaptPDL</i> à la hiérarchie de familles <i>PDLHierarchy</i> . . . . .	62
4.13	L' <i>i-DSML</i> correspondant à la famille <i>DependAdaptPDL</i> . . . . .	62
4.14	Un modèle conforme au méta-modèle de <i>DependAdaptPDL</i> , avant et après adaptation . . . . .	63
4.15	Ajout de la famille <i>DependSkipAdaptPDL</i> à la hiérarchie de familles <i>PDLHierarchy</i> . . . . .	64
4.16	L' <i>i-DSML</i> correspondant à la famille <i>DependSkipAdaptPDL</i> . . . . .	65
4.17	Un modèle conforme au méta-modèle de <i>DependSkipAdaptPDL</i> , avant et après adaptation . . . . .	65
4.18	Ajout de la famille <i>ManagedSkipAdaptPDL</i> à la hiérarchie de familles <i>PDLHierarchy</i> . . . . .	66
4.19	L' <i>i-DSML</i> correspondant à la famille <i>ManagedSkipAdaptPDL</i> . . . . .	67
4.20	Un modèle conforme au méta-modèle de <i>ManagedSkipAdaptPDL</i> , avant et après adaptation . . . . .	68
5.1	Une architecture pour l'adaptation d' <i>i-DSML</i> . . . . .	71
5.2	Le méta-modèle d'orchestration de l'adaptation <i>ASDL</i> . . . . .	73
5.3	Un exemple généraliste d'une orchestration de l'adaptation . . . . .	76
5.4	Le modèle d'orchestration de l'adaptation pour <i>PDL</i> . . . . .	78
5.5	Le modèle d'orchestration de l'adaptation pour <i>AdaptPDL</i> . . . . .	80
5.6	Le modèle d'orchestration de l'adaptation pour <i>SkipAdaptPDL</i> . . . . .	81
5.7	Le modèle d'orchestration de l'adaptation pour <i>DependAdaptPDL</i> . . . . .	83
5.8	Le modèle d'orchestration de l'adaptation pour <i>DependSkipAdaptPDL</i> . . . . .	84
5.9	Le modèle d'orchestration de l'adaptation pour <i>ManagedSkipAdaptPDL</i> . . . . .	86
6.1	Le diagramme de classes pour la classe du moteur d'orchestration . . . . .	95
6.2	Le diagramme de séquence d'un exemple du fonctionnement du moteur d'orchestration . . . . .	99
7.1	Un éditeur de politiques d'adaptation . . . . .	103
7.2	Une orchestration de l'adaptation contenant une politique d'adaptation composite . . . . .	105
7.3	Une architecture pour l'adaptation de l' <i>i-DSML</i> <i>SkipAdaptPDL</i> avec méta-adaptation . . . . .	107
7.4	Le modèle d'orchestration de l'adaptation pour <i>ASDL</i> . . . . .	108
A.1	Le méta-modèle d'un feu de circulation . . . . .	117
A.2	Le modèle du feu standard . . . . .	118
A.3	Le modèle du feu pour piétons . . . . .	119
A.4	Le modèle du feu standard (machine à états) . . . . .	121
A.5	Le modèle du feu pour piétons (machine à états) . . . . .	122
A.6	Le modèle du feu standard (réseau de Petri) . . . . .	126
A.7	Le modèle du feu pour piétons (réseau de Petri) . . . . .	126

# Liste des tableaux

3.1	Portée des sémantiques d'exécution et d'adaptation . . . . .	39
A.1	Les solutions pour construire le moteur d'exécution d'un <i>i-DSML</i> . . . . .	117
A.2	Des moteurs d'exécution spécifiques à un méta-modèle . . . . .	121



# Listings

2.1	Une politique d'adaptation pour changer le comportement d'un robot de combat . . . . .	14
2.2	Une politique d'adaptation pour changer le rôle d'un nœud (pseudo-code) . . . . .	15
2.3	Une politique d'adaptation pour changer le rôle d'un nœud (format <i>XML</i> ) . . . . .	15
2.4	Une politique d'adaptation pour changer le mode de diffusion des informations . . . . .	15
2.5	Une politique d'adaptation pour changer le mode de connexion réseau . . . . .	16
3.1	Les invariants <i>OCL</i> pour la machine à états . . . . .	33
4.1	Des invariants <i>OCL</i> pour <i>PDL</i> . . . . .	48
5.1	L'implémentation « en dur » d'une politique d'adaptation dans le moteur d'adaptation . .	70
5.2	Des invariants <i>OCL</i> pour <i>ASDL</i> . . . . .	72
6.1	Le code <i>Java/EMF</i> de la méthode <code>next</code> de la classe <code>SequenceImpl</code> . . . . .	89
6.2	Le code <i>Java/EMF</i> de la méthode <code>cLate</code> de la classe <code>ProcessImpl</code> . . . . .	90
6.3	Le code <i>Java/EMF</i> de la méthode <code>aSkip</code> de la classe <code>ProcessImpl</code> . . . . .	90
6.4	Le code <i>Java/EMF</i> de la classe <code>SkipAdaptPDLEngine</code> . . . . .	91
6.5	Le renommage de la méthode à adapter . . . . .	92
6.6	Le code <i>Java/EMF</i> de la classe <code>SkipAdaptPDLEngine</code> après couplage . . . . .	93
6.7	Le code <i>Java/EMF</i> de la classe <code>OrchestrationEngine</code> . . . . .	95
6.8	Le code <i>Java/EMF</i> de la méthode <code>run</code> de la classe <code>AdaptationSemanticsImpl</code> . . . . .	96
6.9	Le code <i>Java/EMF</i> de la méthode <code>run</code> de la classe <code>AdaptationPolicyImpl</code> . . . . .	97
6.10	Le code <i>Java/EMF</i> de la méthode <code>next</code> de la classe <code>SimpleStepImpl</code> . . . . .	97
6.11	Le code <i>Java/EMF</i> de la méthode <code>next</code> de la classe <code>DecisionStepImpl</code> . . . . .	97
6.12	Le code <i>Java/EMF</i> de la méthode <code>dynamicCall</code> de la classe <code>OperationImpl</code> . . . . .	98
A.1	Le comportement écrit avec <i>Kermeta</i> . . . . .	118
A.2	Le comportement écrit avec <i>Java/EMF</i> . . . . .	119
A.3	Le comportement écrit avec <i>ATL</i> . . . . .	120
A.4	Le modèle du feu standard écrit avec <i>Java/PauWare engine</i> . . . . .	122
A.5	Le modèle du feu pour piétons écrit avec <i>Java/PauWare engine</i> . . . . .	122
A.6	Le modèle du feu standard écrit avec <i>SCXML</i> . . . . .	124
A.7	Le modèle du feu pour piétons écrit avec <i>SCXML</i> . . . . .	124
A.8	Le code <i>Java/Apache Commons SCXML</i> qui charge le modèle du feu standard . . . . .	124
A.9	Le code <i>Java/Apache Commons SCXML</i> qui charge le modèle du feu pour piétons . . . .	125



# Chapitre 1

## Introduction

La complexité des systèmes informatiques augmente de façon exponentielle au fil du temps, à tel point que depuis plusieurs décennies, les ingénieurs en logiciel ressentent le besoin de monter en abstraction en représentant leurs systèmes à l'aide de modèles qui permettent chacun de décrire une partie de l'application, selon un point de vue particulier. Ces modèles ont longtemps été considérés comme des outils de communication ou de documentation pour un programme donné, jouant un rôle purement contemplatif. Depuis, la situation a changé et les modèles jouent désormais un rôle productif en étant placés au premier plan des activités de développement et de maintenance des logiciels. À l'intérieur de cette situation, nous trouvons évidemment la génération automatique de code où les modèles sont utilisés en phase de conception mais disparaissent en phase d'exécution. Un courant plus récent propose de les exploiter également en phase d'exécution : c'est l'idée du *models@run.time* (au sens littéral du terme) où le modèle est embarqué et disponible pendant l'exécution du programme. Mais l'autre tendance émergente concerne les modèles exécutables où un modèle produit en phase de conception est réutilisé en tant que tel en phase d'exécution.

De tels modèles directement exécutés sont écrits avec des *interpreted Domain-Specific Modeling Language (i-DSML)* [18] et sont interprétés par un moteur d'exécution. Ils trouvent leur utilité pour de la simulation, de la vérification, du prototypage et cas ultime, permettent de se passer complètement de la phase d'implémentation puisque le modèle est considéré comme une application en soi. Un slogan associé à cette démarche pourrait être : « *What You Model Is What You Get* » (*WYMIWYG*). Elle vise la rapidité (ou l'agilité) et à terme la diminution des coûts de production du programme.

D'autre part, les logiciels peuvent être dotés de capacités adaptatives. Lorsque ces adaptations sont automatiques, elles correspondent au concept de l'informatique autonome, une idée qui a été proposée en 2001 par *IBM* [44, 34, 80] et qui consiste à considérer que les systèmes informatiques sont trop complexes aujourd'hui pour être administrés par des êtres humains. La proposition d'*IBM* est de laisser les systèmes informatiques s'administrer eux-mêmes en parfaite autonomie. Ces systèmes sont constitués d'un ensemble d'éléments autonomes qui gèrent chacun leur comportement interne. Le terme auto-\* (*self-\**) ou auto-gestion (*self-management*) est utilisé pour désigner cette capacité d'auto-adaptation et regroupe les idées d'auto-configuration (*self-configuration*), d'auto-optimisation (*self-optimization*), d'auto-guérison (*self-healing*) et d'auto-protection (*self-protection*). Ces applications adaptatives sont généralement confrontées à un contexte qui est plus ou moins connu. Ce contexte est susceptible de changer au cours de l'exécution du programme adaptatif (un exemple élémentaire concerne le niveau

de batterie d'un smartphone qui diminue ou encore quand aucun point d'accès Wi-Fi n'est trouvé). Pour faire face à ces situations diverses, l'application doit être capable de s'adapter à son contexte et de modifier son comportement dynamiquement (c'est-à-dire durant son exécution) sans interruption de service. Mettre en place une telle adaptation logicielle correspond aux défis de recherche présentés dans [75]. Nous considérons ces adaptations comme étant d'un grand intérêt pour les logiciels puisque, d'un point de vue financier, nous ne pouvons pas envisager de demander aux clients de retourner leur matériel afin d'effectuer une opération de maintenance à chaque fois que l'application rencontre un cas exceptionnel dans son contexte. Ainsi, l'adaptation logicielle peut permettre de réduire drastiquement les coûts de maintenance des programmes.

En résumé, d'un côté nous savons que l'exécution de modèles, grâce aux *i-DSML*, permet de réduire les coûts de production d'un logiciel tandis que de l'autre côté nous savons que l'adaptation logicielle permet de réduire les coûts de maintenance d'une application. Dans cette thèse nous souhaitons prendre le meilleur des deux mondes en fusionnant les deux idées. Le résultat revient *in fine* à directement adapter l'exécution d'un modèle via des *i-DSML* adaptables. Une telle approche répond en partie<sup>1</sup> au cinquième défi de recherche lancé dans [27] qui vise à utiliser des modèles à l'exécution pour les systèmes adaptatifs.

De manière orthogonale à cette fusion, il est à noter que cette thèse s'efforce de définir des adaptations génériques (c'est-à-dire indépendantes du contenu métier du modèle) pouvant ainsi être réutilisées. En effet, la question de la généricité (et donc de la réutilisation) ne s'est pas posée concernant la sémantique d'exécution tant il paraît évident que pour être réellement utile, un moteur donné doit pouvoir exécuter un modèle sans en connaître le contenu métier. Mais étonnamment, cette question n'a pas ressurgi lorsqu'il s'agissait de la sémantique d'adaptation et nous constatons que l'immense majorité des travaux sur l'adaptation logicielle repose en fait sur des adaptations métier, c'est-à-dire qu'elle présuppose le contenu du modèle pour fonctionner. Or, nous sommes convaincus que ce sont les adaptations génériques qui représentent un réel défi scientifique.

## 1.1 Contributions de la thèse

Les travaux de cette thèse consistent à proposer une approche destinée à l'ingénieur logiciel pour qu'il puisse construire des logiciels adaptatifs avec des modèles directement exécutés et adaptés. Cette approche qui sera développée dans les prochains chapitres est scindée en trois contributions scientifiques :

1. la caractérisation des *i-DSML* adaptables
2. l'organisation hiérarchique de la spécialisation des *i-DSML* adaptables
3. l'écriture de politiques d'adaptation à travers un *i-DSML* d'orchestration de l'adaptation

La première contribution consiste à caractériser les *i-DSML* adaptables qui sont des *i-DSML* étendus afin de gérer l'adaptation. En Ingénierie Dirigée par les Modèles (IDM), cette nouvelle façon de procéder se démarque fortement de la technique du *models@run.time* [8] mise au point par les chercheurs et permettant elle aussi à l'ingénieur logiciel de développer des applications adaptatives tout en profitant des avantages offerts par la modélisation. En effet, les principes du *models@run.time* reposent sur un modèle reflétant le système et permettant de raisonner pendant l'exécution sur la nécessité de s'adapter en fonction du contexte. Même si cette technique a fait ses preuves, son inconvénient majeur réside dans le fait qu'il est nécessaire de maintenir un lien causal entre le modèle représentant l'état du système et le

---

1. Nous ne traitons pas dans nos travaux les problèmes relatifs à la sûreté ou à la sécurité.

système lui-même afin que les raisonnements continuent à être valides. La technique proposée dans nos travaux se distingue clairement du *models@run.time* puisqu'ici le système se confond avec le modèle du système, qui est directement exécuté et adapté [12]. Fusionner le système et son modèle en une seule et même entité offre l'avantage de simplifier considérablement l'architecture du logiciel adaptatif car il n'y a plus de lien causal à maintenir entre les deux. De plus, puisque nous adaptons l'exécution d'un modèle (et non pas simplement un modèle statique), de nouvelles possibilités d'adaptation sont envisageables comme par exemple modifier la sémantique d'exécution du modèle en plus de sa structure. Pour concevoir un tel modèle, il est nécessaire de se servir d'un langage de modélisation interprétable particulier offrant des concepts permettant à la fois de gérer l'exécution et l'adaptation du système : nous parlons alors d'*i-DSML* adaptable. Cette caractérisation des *i-DSML* adaptables a fait l'objet d'une publication dans [13].

La deuxième contribution concerne l'organisation hiérarchique de la spécialisation des *i-DSML* adaptables. Une politique d'adaptation (c'est-à-dire une règle indiquant que telles conditions à l'exécution entraînent telles actions d'adaptation) se définit à l'aide de trois types d'opérations : les vérifications d'adaptation, les actions d'adaptation et les opérations d'exécution. Nous avons constaté que spécialiser un *i-DSML* adaptable (en l'étendant ou en le contraignant) peut faciliter la définition de nos politiques d'adaptation et donc de ces trois sortes d'opérations. Nous proposons à l'ingénieur logiciel de s'appuyer sur le concept des familles où chaque famille est associée à un *i-DSML* adaptable et peut être spécialisée pour donner naissance à de nouvelles opérations qui peuvent être combinées entre elles pour construire des politiques d'adaptation. Ces spécialisations nous amènent à construire progressivement une hiérarchie de familles. La notion de famille a fait l'objet d'une publication dans [68].

La troisième contribution traite de l'écriture de politiques d'adaptation à travers un *i-DSML* d'orchestration de l'adaptation. En effet, à chaque famille correspond une ou plusieurs politiques d'adaptation qui pourraient être écrites directement « en dur » dans le code correspondant à l'implémentation du moteur interprétant le modèle mais ceci interdirait par la suite de pouvoir remplacer les politiques d'adaptation par d'autres tout en conservant le même moteur. Pour contourner ce problème, nous proposons d'écrire ces politiques d'adaptation à l'aide d'un modèle interprétable à part : le modèle d'orchestration de l'adaptation qui permet d'ordonnancer les opérations. Cette façon de procéder permet de séparer d'un côté les opérations de différents types et de l'autre côté leur ordonnancement qui est réifié dans un modèle. Ainsi, nous sommes en présence d'un *i-DSML* adapté par un autre *i-DSML*.

## 1.2 Plan du document

Dans le chapitre suivant, nous allons commencer par rappeler ce qu'est l'IDM et expliquer son utilité pour l'ingénieur logiciel. Après cela, nous étudierons l'existant en ce qui concerne l'exécution de modèles. En effet, nous considérons que l'adaptation d'exécution de modèles est un prolongement de l'exécution de modèles ce qui implique que pour étudier cette adaptation il est d'abord nécessaire de s'intéresser à cette exécution. Plus précisément, comme l'exécution de modèles existe depuis longtemps, nous verrons à la fois des approches historiques (réseaux de Petri, *workflows* et machines à états) et une vision IDM plus récente basée sur les *i-DSML*. Aussi, nous profiterons de l'occasion pour comparer les deux voies pour atteindre l'exécution d'un modèle : l'interprétation et la compilation. Ensuite, nous nous intéresserons à la littérature traitant l'adaptation logicielle de façon générale (c'est-à-dire sans se préoccuper de l'exécution de modèles). Nous verrons les notions et concepts bien connus liés à l'adaptation logicielle tels que le contexte, les politiques d'adaptation et la boucle d'adaptation. Nous expliquerons également

la technique du *models@run.time* provenant de l'IDM permettant de mettre en place une adaptation logicielle grâce à un modèle embarqué à l'exécution. Après avoir étudié l'adaptation logicielle classique, nous nous pencherons sur l'adaptation de l'exécution de modèles. Dans cette dernière partie de l'état de l'art, nous verrons à la fois des approches hors IDM (réseaux de Petri adaptatifs, *workflows* adaptatifs et machines à états adaptatives) mais aussi en IDM où la question d'adapter une exécution de modèles n'a pas été beaucoup discutée.

Dans le chapitre 3 nous allons caractériser la notion d'*i-DSML* adaptable. L'idée ici est de présenter d'abord ce qu'est un *i-DSML* (c'est-à-dire un méta-modèle particulier et son moteur d'exécution), puis de montrer comment l'étendre pour qu'il devienne un *i-DSML* adaptable (c'est-à-dire un méta-modèle encore plus particulier et son moteur d'adaptation). Nous illustrerons nos propos avec l'exemple d'une machine à états d'un train circulant sur des voies ferrées et adaptant sa vitesse en fonction d'une signalisation inconnue au départ.

Dans le chapitre 4, nous allons expliquer comment le concept des familles peut favoriser grandement l'écriture de politiques d'adaptation pour le modèle d'un *i-DSML* adaptable. En effet, nous verrons que chaque famille attachée au méta-modèle d'un *i-DSML* adaptable permet d'une part la spécialisation de celui-ci (offrant ainsi de nouvelles possibilités d'adaptation) et d'autre part un accès facile pour l'ingénieur logiciel aux différentes opérations qu'il pourra combiner entre elles pour construire ses politiques d'adaptation. Après avoir présenté le *DSML Family Hierarchy Description Language (FHDL)* permettant la construction d'une hiérarchie de familles, nous appliquerons cette démarche sur l'exemple d'un *i-DSML* adaptable permettant de définir des processus composés d'activités à réaliser dans un temps déterminé.

Dans le chapitre 5, nous allons nous intéresser à l'orchestration de l'adaptation de notre modèle. Nous verrons dans un premier temps que l'architecture de notre moteur va être différente puisqu'il fera usage de deux modèles : celui à exécuter et à adapter bien sûr mais aussi celui présentant cette orchestration, utilisé comme un simple paramètre. Dans un second temps, nous spécifierons le méta-modèle de l'*i-DSML Adaptation Semantics Description Language (ASDL)* permettant de construire un tel modèle d'orchestration de l'adaptation. Nous verrons à la fois sa syntaxe abstraite (contenant les différents concepts) et sa syntaxe concrète (permettant de représenter le modèle sous la forme d'un diagramme). Nous appliquerons cette approche sur le cas d'utilisation de l'*i-DSML* adaptable pour la définition de processus présenté auparavant.

Dans le chapitre 6, nous allons montrer comment implémenter une telle technique au travers des outils mis à notre disposition. Nous partirons d'un modèle adaptable, d'une famille et d'un modèle d'orchestration de l'adaptation déjà existants pour construire progressivement le moteur d'orchestration grâce à *Java/EMF*<sup>2</sup>.

Enfin, nous terminerons cette thèse en donnant notre conclusion et nos perspectives dans le dernier chapitre. Nous mettrons en avant les avantages qu'offre notre solution pour l'ingénieur logiciel souhaitant concevoir des logiciels adaptatifs. C'est dans ces dernières lignes que nous évoquerons les différents défis qu'il reste encore à relever concernant l'adaptation de l'exécution de modèles (par exemple la méta-adaptation).

---

2. *EMF* : *Eclipse Modeling Framework*, <http://www.eclipse.org/modeling/emf/>

# Chapitre 2

## État de l’art

Ce chapitre présente un état de l’art des travaux autour de l’adaptation et des modèles exécutés pour bien comprendre, par la suite, l’approche retenue dans cette thèse. Puisque tout notre travail s’inscrit dans l’IDM, il n’est pas inutile de procéder d’abord à quelques rappels qui fixeront les concepts et le vocabulaire utilisés dans la suite du manuscrit. Dès lors, nous serons en mesure d’étudier comment l’exécution de modèles est traitée, à la fois hors IDM et en IDM. Nous nous intéresserons ensuite à l’adaptation logicielle de façon générale avant de nous concentrer sur une spécificité : l’adaptation d’exécution de modèles. Enfin, nous ferons un bilan des travaux scientifiques que nous avons été amenés à étudier.

### 2.1 Rappels sur l’ingénierie dirigée par les modèles

En Ingénierie Dirigée par les Modèles (IDM), nous cherchons à développer des applications mettant l’accent sur des modèles plutôt que sur du code source : nous modélisons (haut niveau d’abstraction) plus que nous ne programmons (bas niveau d’abstraction). Pour construire de tels modèles, nous utilisons un langage de modélisation de la même façon que pour des programmes nous nous servons d’un langage de programmation. Le langage de modélisation *Unified Modeling Language* (*UML* [64]) est sans doute le plus connu de tous pour concevoir des logiciels. Toutefois, pour répondre à un besoin précis, les ingénieurs en logiciel ressentent souvent le besoin de créer un nouveau langage de modélisation spécifique à leur domaine : un *Domain-Specific Modeling Language* (*DSML*).

Un *DSML* est généralement accompagné de nombreuses autres choses permettant son exploitation : éditeurs, générateurs de code, transformations diverses, . . . Mais leur construction demande beaucoup de temps, d’efforts et de compétences. L’IDM facilite cette tâche laborieuse en apportant un environnement bien outillé basé sur une pile de méta-modélisation.

#### 2.1.1 La pile de méta-modélisation selon l’OMG

Pour définir un langage de modélisation, nous avons besoin d’un autre langage. En IDM, nous parlons alors d’un langage de méta-modélisation afin de le distinguer d’un langage de modélisation classique. Un exemple de langage de méta-modélisation existant est le *Meta Object Facility* (*MOF* [58]) qui est spécifié par l’*Object Management Group* (*OMG*). Il contient tous les concepts nécessaires à la création d’un

langage de modélisation. Nous avons donc trois<sup>3</sup> niveaux différents : celui des modèles ( $M1$ ) conçus avec un langage de modélisation, celui des méta-modèles ( $M2$ ) conçus avec un langage de méta-modélisation et enfin celui des méta-méta-modèles ( $M3$ ). Les méta-méta-modèles se définissent eux-mêmes et sont donc situés au sommet de la pile de méta-modélisation (cf. figure 2.1).

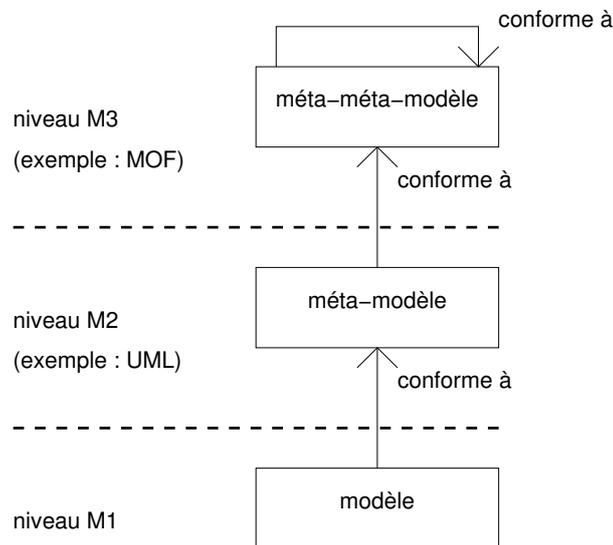


FIGURE 2.1 – La pile de méta-modélisation

Le projet *Eclipse Modeling Framework* (*EMF*), qui est l'environnement de modélisation le plus utilisé aujourd'hui, est aligné avec la pile ci-avant. Ainsi, *EMF* introduit le langage *Ecore* qui est une implémentation du langage *MOF*<sup>4</sup>.

La première remarque est que la hiérarchie ci-dessus induit une notion d'ordre : le méta-méta-modèle ( $M3$ ) doit être conçu avant le méta-modèle ( $M2$ ) qui doit être conçu avant le modèle ( $M1$ ). Pour concevoir un méta-modèle, nousinstancions des méta-méta-éléments composant un méta-méta-modèle et pour concevoir un modèle, nousinstancions des méta-éléments composant un méta-modèle. Ainsi, nous pouvons voir un méta-modèle comme un ensemble d'instances de méta-méta-éléments et un modèle comme un ensemble d'instances de méta-éléments. Dans le jargon de l'IDM, nous disons qu'un méta-modèle est « conforme » à un méta-méta-modèle et qu'un modèle est « conforme » à un méta-modèle.

La seconde remarque est que la hiérarchie est toute relative et que cela introduit parfois de la confusion : en se focalisant uniquement sur le contenu d'un modèle, celui-ci peut parfois être classé indifféremment à chacun des niveaux. Une anecdote illustre cela : *Ecore* a été pensé au départ comme un diagramme de classes *UML* minimaliste, avec des motivations purement métier (cf. les premières présentations de Merks, le responsable du projet *EMF* et le coauteur de [79]). Un des premiers exemples donné était d'ailleurs la modélisation d'un bon de commande (classes `PurchaseOrder` et `Item`), avant d'illustrer la génération automatique de code *Java* permise par l'environnement. *Ecore* était donc un langage de modélisation centré « données » et utilisé exclusivement pour produire des modèles de niveau  $M1$ . Puis, il a subi une « promotion » (selon le terme de circonstance employé par Muller *et al.* dans [53]) pour

3. Il existe un quatrième niveau ( $M0$ ) défini par l'*OMG* que nous ne considérons pas ici pour des raisons de clarté.

4. *Ecore* est une implémentation d'*Essential MOF* (*EMOF*), l'un des deux sous-ensembles de *MOF* et permet de construire des méta-modèles simples. Le *Complete MOF* (*CMOF*) est l'autre sous-ensemble de *MOF* et permet de concevoir des méta-modèles plus sophistiqués.

s'imposer définitivement comme méta-méta-modèle, c'est-à-dire qu'il servira à produire des modèles de niveau *M2*.

### 2.1.2 Syntaxes concrètes

Tandis que les (méta-)méta-modèles décrivent un langage par sa syntaxe abstraite uniquement, il est souvent nécessaire (mais pas toujours) de les doter d'une ou plusieurs syntaxes concrètes, également appelées syntaxes de surface.

Ces syntaxes concrètes peuvent être soit textuelles, soit graphiques (aussi appelée syntaxes visuelles). Nous parlons alors respectivement de modélisation textuelle et de modélisation graphique<sup>5</sup>. Il est possible de proposer plusieurs syntaxes concrètes pour un même langage (comme dans [2]) afin de satisfaire les préférences d'un plus grand nombre d'ingénieurs en logiciel. Avec le langage de modélisation *UML*, une syntaxe graphique est fournie par l'*OMG*. Des éditeurs *UML* comme *Papyrus* respectent cette syntaxe concrète et offrent à l'utilisateur une interface permettant de concevoir leurs modèles avec cette notation.

*Eclipse EMF* facilite la conception de ces éditeurs de syntaxes concrètes avec des outils dédiés à leur construction. Par exemple, *Xtext* permet de générer automatiquement un éditeur de syntaxe concrète textuelle à partir d'un méta-modèle importé. Si au lieu d'une syntaxe textuelle, nous cherchons à proposer une syntaxe graphique, une autre extension pour *Eclipse* appelée *Graphical Modeling Framework (GMF)* sera plus appropriée pour la réalisation d'un tel éditeur. Nous pouvons ainsi écrire des *DSML* avec un visuel dédié comme dans [70] par exemple qui produit un *DSML* pour la spécification de scénarios de tests d'applications de téléphonie mobile.

Au niveau de la méta-modélisation, des syntaxes concrètes sont aussi parfois proposées. Par exemple, avec *EMF*, le langage *Ecore* est pourvu de plusieurs syntaxes concrètes. En effet, pour concevoir notre méta-modèle, selon l'éditeur choisi, nous bénéficions d'une syntaxe textuelle ou graphique. L'éditeur *OCLInEcore* offre une syntaxe textuelle bien adaptée pour visualiser le méta-modèle lorsque celui-ci contient des invariants *Object Constraint Language (OCL* [60]). Le langage *Kernel MetaMetaModel (KM3)* [41] a également opté pour ce type de syntaxe. L'éditeur graphique *Ecore Diagram Editing*, quant à lui, présente le méta-modèle sous la forme d'un diagramme de classes « à la *UML* ».

### 2.1.3 La manipulation de modèles

Après avoir capturé ce que nous désirons dans un modèle, l'autre grande force de l'IDM est d'offrir les moyens de le manipuler, au sens large du terme. Parmi la gamme de manipulations possibles, les plus connues sont certainement les transformations de modèles. Elles sont réalisées à l'aide d'un outil qui permet d'effectuer soit des transformations *Model to Text (M2T)* [59], c'est-à-dire qu'il prend en entrée un modèle et donne en sortie du code (dans l'immense majorité des cas), soit des transformations *Model to Model (M2M)* où nous avons un modèle à la fois en entrée et un (ou plusieurs) en sortie.

Il existe des solutions basées sur des langages généralistes comme *Java/EMF* et *Kermeta*<sup>6</sup> [53]. Ces langages permettent toutes sortes de manipulations dont les transformations *M2M* et *M2T*. Cependant, des outils plus spécialisés, dédiés à une manipulation bien précise, peuvent être plus intéressants à utiliser. Typiquement, *Acceleo* qui se présente comme une extension pour *Eclipse* est l'un de ces outils de transformation *M2T*. Il permet de générer du code à partir de modèles écrits avec *UML* ou tout autre

5. Une idée fausse répandue consiste à considérer qu'un langage de modélisation est forcément graphique.

6. *Kermeta* : *KErnel METAmodeling language*, <http://www.kermeta.org/>

méta-modèle écrit avec *Ecore*. Les transformations *M2M*, quant à elles, sont réalisées par un moteur de transformation auquel nous donnons un ensemble de règles qui sont écrites dans un langage de transformation tel qu'*ATL*<sup>7</sup> ou encore *Query/View/Transformation (QVT)* [61].

Dans le cas des transformations *M2M*, nous pouvons distinguer deux sortes de transformations : endogènes (cf. figure 2.2) et exogènes (cf. figure 2.3). Le modèle source et le modèle cible d'une transformation endogène sont conformes au même méta-modèle tandis que ceux d'une transformation exogène sont issus de deux méta-modèles distincts. Un exemple classique de transformation exogène est celui entre un modèle conforme au méta-modèle d'un diagramme de classes *UML* et un modèle conforme au méta-modèle d'une base de données relationnelle. Pour chaque classe qui est composée d'attributs dans le modèle source, une table qui est constituée de colonnes est créée dans le modèle cible.

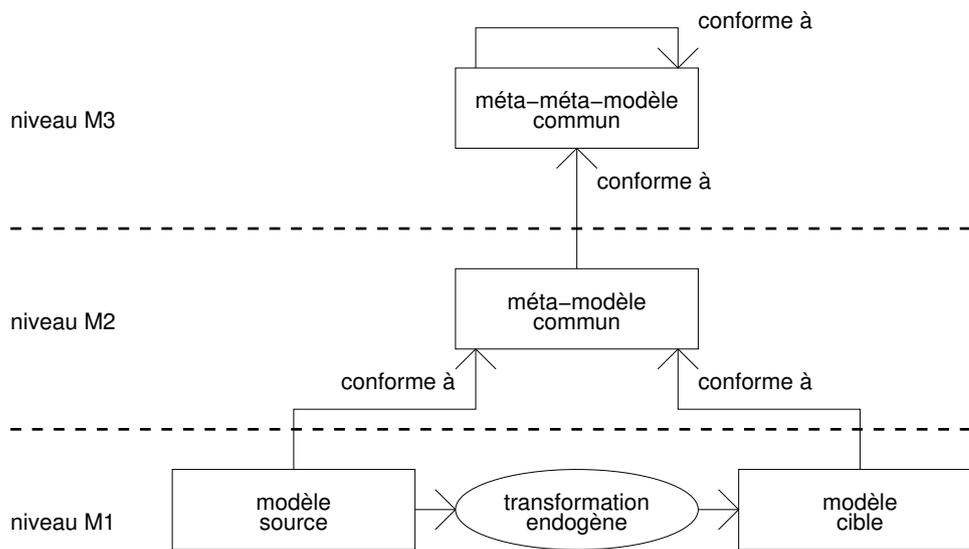


FIGURE 2.2 – Les transformations de modèles endogènes

Il est à noter que l'exécution de modèles est un cas particulier de la manipulation de modèles comme nous le verrons dans la sous-section 2.2.2.

## 2.2 L'exécution de modèles

La section précédente a rappelé ce qu'est l'IDM et a expliqué à quoi elle pouvait servir pour les ingénieurs en logiciel. Après avoir écrit un programme, il est légitime de l'exécuter pour visualiser le résultat de cet effort de programmation. Ainsi, le code écrit dans des langages programmation (*Java*, *C++*, *C#*, ...) et destiné à représenter les éléments du logiciel de manière concrète est naturellement exécutable à l'aide de compilateurs, interpréteurs et autres machines virtuelles. En revanche, il est beaucoup moins intuitif d'exécuter des modèles. En effet, cela peut sembler paradoxal de vouloir exécuter des représentations abstraites écrites dans des langages de modélisation tels qu'*UML*, *State Chart XML (SCXML)* [84] ou encore *Foundational UML (FUML)* [62]). Dans cette section, nous allons nous préoccuper de l'exécution de modèles.

7. *ATL* : *Atlas Transformation Language*, <http://www.eclipse.org/at1/>

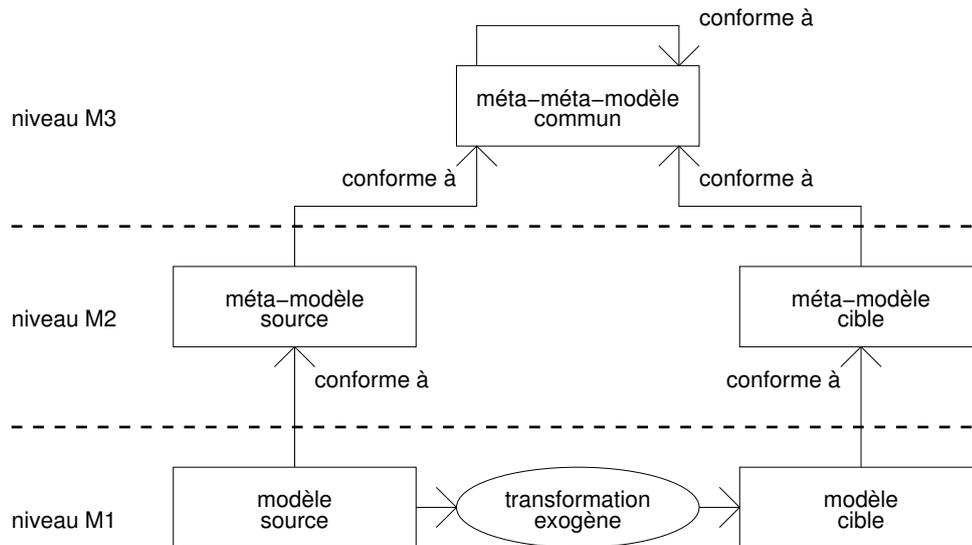


FIGURE 2.3 – Les transformations de modèles exogènes

### 2.2.1 Les modèles de processus hors IDM

Depuis longtemps, nous disposons d'outils dédiés aux simulations et aux tests sur des modèles spécifiques via leur exécution. Nous allons voir dans cette sous-section ces modèles de processus qui sortent du cadre strict de l'IDM.

Les réseaux de Petri, inventés dans les années 1960 par la personne de même nom, reposent sur les concepts de places, de transitions, d'arcs et de jetons. Afin d'indiquer la situation de départ, les places contiennent des jetons (c'est-à-dire le marquage initial). Les arcs, quant à eux, sont positionnés entre les places et les transitions pour préciser si ces places sont situées à l'entrée ou à la sortie des transitions. Le franchissement d'une transition se fait de manière automatique en fonction du nombre de jetons présents dans les places à l'entrée de cette transition. À chaque franchissement, un ou plusieurs jetons sont retirés de chaque place en entrée et ajoutés à chaque place en sortie (c'est-à-dire que le marquage est modifié). La théorie montre que ces processus sont capables de s'exécuter. En pratique, nous disposons d'outils informatiques pour automatiser l'exécution de ces réseaux de Petri. Parmi ces outils, nous avons *Renew*<sup>8</sup> ou encore *Tina*<sup>9</sup> qui proposent à la fois un éditeur afin de construire un réseau de Petri et un moteur pour l'exécuter pas à pas. La création d'un réseau de Petri avec *Renew* et *Tina* sont respectivement présentés en annexes A.2.3 et A.2.4.

Les *workflows* [32], apparus autour des années 1910 avec le diagramme de Gantt [28], sont des modèles pouvant être exécutés. Le principe est de modéliser un processus composé d'activités qui doivent être réalisées de façon séquentielle ou en parallèle. Il existe plusieurs langages de *workflows* tels que *Web Services Business Process Execution Language (WS-BPEL* [54]), *Business Process Model and Notation (BPMN* [56]) ou *Yet Another Workflow Language (YAWL* [1]). Ils offrent tous les concepts permettant de modéliser entièrement un processus exécutable. Pour parvenir à exécuter de tels modèles, des outils informatiques appelés des moteurs de *workflows* sont utilisés. Par exemple, le moteur *Apache ODE*<sup>10</sup>

8. *Renew* : REference NEt Workshop, <http://www.renew.de/>

9. *Tina* : TIme petri Net Analyzer, <http://projects.laas.fr/tina/>

10. *ODE* : Orchestration Director Engine, <http://ode.apache.org/>

permet l'exécution de *workflows* écrits avec le langage *WS-BPEL*.

Les machines à états ont vu le jour dans le milieu des années 1950 avec les automates de Mealy et Moore. En 1987 [37], Harel a proposé un formalisme visuel pour ces machines à états qui sera adopté par de nombreux outils informatiques. Ce formalisme sera plus tard repris par des langages tels qu'*UML* ou *SCXML* et repose sur les concepts d'états, de transitions et d'événements. Une machine à états contient des états qui sont reliés par des transitions. Un état particulier est l'état initial qui correspond à l'état actif au lancement de la machine à états. Chaque transition est associée à un événement et part d'un état source pour arriver sur un état cible. À chaque fois qu'un événement est généré, si celui-ci correspond à une transition dont l'état source est l'état actif, nous passons par celle-ci pour arriver sur son état cible. Pour exécuter de tels processus, nous avons à notre disposition des outils tels qu'*Apache Commons SCXML*<sup>11</sup> ou encore *PauWare engine*<sup>12</sup> fournissant un moteur d'exécution et se présentant sous la forme d'une bibliothèque *Java*. Alors qu'*Apache Commons SCXML* supporte nativement les machines à états écrites avec le langage de modélisation *SCXML*, *PauWare engine*, quant à lui, requiert de passer par *SCXML2PauWare* pour transformer un modèle écrit dans le langage de modélisation *SCXML* vers le langage de programmation *Java/PauWare engine* pour le faire ensuite exécuter par *PauWare engine*. Une particularité de *PauWare engine* est de pouvoir se déployer également sur les mobiles dotés du système d'exploitation *Android* [35]. La création d'une machine à états avec *Apache Commons SCXML* est présentée en annexe A.2.2. Le moteur d'exécution *PauWare engine* et l'outil de transformation *SCXML2PauWare* sont respectivement présentés en annexes A.2.1 et A.3.

### 2.2.2 L'exécution de modèles en IDM

Nous avons vu dans la sous-section précédente des techniques historiques permettant l'exécution de modèles de processus. Ces techniques sortent complètement du cadre de l'IDM puisque les moteurs d'exécution pour ces modèles de nature exécutable sont codés de manière *ad hoc* et ne reposent pas sur cette approche globale permettant de créer des *DSML* à partir d'outils dédiés à cette tâche. Dans cette sous-section, nous allons maintenant nous focaliser sur une approche plus récente grâce à l'IDM pour parvenir à une exécution de modèles facilitée notamment par l'environnement *EMF*.

De la même manière que pour les langages de programmation, les *DSML* peuvent être compilés [38] ou interprétés [39]. Cela signifie que si nous souhaitons exécuter des modèles, nous devons faire un choix entre l'une de ces deux techniques. Cette distinction a été faite par Mernik *et al.* dans [48] lorsqu'ils indiquent que nous devons choisir l'implémentation la plus appropriée pour un *DSML* exécutable :

- un *DSML* compilé : il est traduit vers un langage qui lui est exécutable. Nous parlons souvent de génération de code pour désigner cette approche.
- un *DSML* interprété : il est reconnu et interprété par un moteur d'exécution implémentant une sémantique d'exécution. Avec cette approche, aucune transformation n'est réalisée car le modèle est directement exécutable.

Les *DSML* compilés étant transformés vers d'autres langages, leurs modèles en phase de conception n'existent plus au moment de l'exécution. Par conséquent, un inconvénient majeur des *DSML* compilés est que nous devons attendre l'étape d'implémentation pour nous rendre compte d'éventuels problèmes à corriger dans le modèle. Également, chaque modification dans le modèle requière une nouvelle compilation. Considérons à nouveau le moteur d'exécution *PauWare engine* permettant d'exécuter des machines à

---

11. *Apache Commons SCXML* : <http://commons.apache.org/proper/commons-scxml/>

12. *PauWare engine* : <http://www.pauware.com/>

états. Si nous passons par un éditeur *UML* comme *Papyrus* qui s'inscrit dans une approche IDM, puis que nous utilisons l'outil *UML2PauWare*<sup>13</sup>, nous pouvons transformer un modèle écrit dans le langage de modélisation *UML* vers le langage de programmation *Java/PauWare engine* pour le faire ensuite exécuter par *PauWare engine*. Ainsi, puisqu'il y a une transformation nous pouvons clairement parler de *DSML* compilé dans cette situation.

Les *DSML* interprétés sont aussi appelés des *i-DSML* (avec le « *i* » pour « *interpreted* ») et nous donnons leur caractérisation dans le chapitre 3. L'exécution d'un modèle conforme au méta-modèle d'un *i-DSML* a été longuement étudiée [9, 11, 18, 19, 20, 24, 45, 69] et tous ces travaux ont établi un consensus sur ce qu'est l'exécution d'un modèle. Avec ces langages particuliers, le fait de pouvoir exécuter un modèle avant même son implémentation est un gain de temps et permet de réduire les coûts de production pour au moins deux raisons : (a) il devient possible de détecter et de corriger des problèmes très tôt dans les étapes du cycle du développement logiciel et (b), en définitive, l'étape d'implémentation peut tout simplement être sautée.

Un moteur d'exécution est chargé d'interpréter les instances du méta-modèle d'un *i-DSML*. Ce moteur implémente la sémantique d'exécution du méta-modèle (c'est-à-dire le comportement à l'exécution) et permet donc de donner vie aux modèles. Cette sémantique d'exécution est généralement écrite sous forme opérationnelle à l'aide d'un langage dédié à la manipulation de modèles. Ce langage est aussi appelé un langage d'action et peut être vu comme un langage de programmation orienté modèle. Par exemple, un langage d'action tel que *Kermeta* peut être utilisé pour définir le corps des opérations du méta-modèle d'un *i-DSML*. Le langage *FUML* et sa syntaxe concrète *ALF* [55], quant à eux, ne permettent pas directement de décrire le comportement à l'exécution pour le méta-modèle d'un *i-DSML*. En effet, ces deux langages sont au niveau *M2* dans la pile de méta-modélisation et servent à définir la sémantique d'exécution de modèles (c'est-à-dire au niveau *M1*). Cependant, des chercheurs se sont intéressés à réaliser une « promotion » de *FUML* vers le niveau *M3* afin de pouvoir l'utiliser pour décrire le comportement d'*i-DSML* [47]. Le langage *Java*, lorsqu'il est couplé à la bibliothèque *EMF*, devient un véritable langage d'action offrant la possibilité d'écrire le corps de méthodes correspondant aux opérations du méta-modèle d'un *i-DSML*. Au lieu de décrire cette sémantique d'exécution de façon opérationnelle avec un langage d'action, il est également possible de l'écrire sous forme translationnelle avec un langage de transformation tel qu'*ATL*. Ainsi, cette sémantique d'exécution est ramenée à une série de transformations endogènes où chaque transformation correspond à un pas d'exécution réalisé par le moteur. La description de ce comportement avec *Kermeta*, *Java/EMF* et *ATL* est présentée en annexe A.1. À titre d'exemples, nous avons l'*i-DSML* de machines à états proposé dans le chapitre 3 qui permet de modéliser des machines à états élémentaires et l'*i-DSML Process Description Language (PDL)* du chapitre 4 qui offre la possibilité de modéliser des processus composés d'activités. Le moteur d'exécution de ces *i-DSML* est implémenté avec *Kermeta* pour les machines à états et avec *Java/EMF* pour *PDL*. Un concept important des *i-DSML* est la notion d'état courant du modèle qui permet de déterminer à n'importe quel instant de l'exécution quels sont les éléments actifs dans le modèle. Cette information peut être stockée soit dans le modèle lui-même, en prévoyant cette possibilité au niveau du méta-modèle de l'*i-DSML*, soit à l'extérieur du modèle, dans le code du moteur.

<sup>13</sup>. *UML2PauWare* est un projet en cours de réalisation par des étudiants de master qui utilise *Acceleo* pour générer du code *Java/PauWare engine* à partir de modèles *UML*.

## 2.3 L'adaptation logicielle

La section précédente a présenté l'exécution de modèles avec des techniques historiques (réseaux de Petri, *workflows* et machines à états) mais aussi et surtout avec les *i-DSML* dans le cadre de l'IDM. Dans cette section, nous allons nous intéresser à l'adaptation logicielle de façon générale (c'est-à-dire sans se préoccuper de l'exécution de modèles). Nous donnerons d'abord une caractérisation de l'adaptation logicielle. Ensuite, nous verrons les concepts bien connus liés à l'adaptation logicielle tels que le contexte, les politiques d'adaptation et la boucle d'adaptation. Nous nous pencherons enfin sur la technique du *models@run.time* issue de la communauté IDM et qui permet de réaliser une adaptation logicielle via un modèle embarqué à l'exécution.

### 2.3.1 Caractérisation de l'adaptation logicielle

D'après le dictionnaire *A Dictionary of Sciences* [22], l'adaptation correspond à tout changement dans la structure ou le fonctionnement d'un organisme qui le rend plus adapté à son environnement. Dans le cas de l'adaptation logicielle, l'« organisme » dans cette définition désigne plus précisément le logiciel tandis que l'« environnement » coïncide avec le contexte de l'application. Cette notion de contexte est très importante puisque la décision de s'adapter ou non se base sur le contenu de celui-ci. En effet, un contexte est susceptible de changer au cours de l'exécution du programme et cela implique de vérifier si le logiciel est dans un état adéquat par rapport à ce nouveau contexte.

Dans [10], Capra distingue deux sortes d'adaptations. La première est l'adaptation réactive qui correspond à la capacité d'une application à se modifier et à se reconfigurer en réaction à des changements du contexte. Le second type d'adaptation est l'adaptation proactive qui correspond cette fois à la capacité d'une application à délivrer le même service de différentes façons quand demandé dans différents contextes et à des instants différents.

Dans [5], nous proposons une toute autre caractérisation de l'adaptation logicielle. L'évolution et l'adaptation sont deux termes très proches puisque dans chaque cas une opération de modification est réalisée sur le système. Nous proposons de les différencier comme suit : l'évolution nécessite d'arrêter l'exécution du logiciel pour effectuer la modification avant de la relancer, tandis que l'adaptation se réalise « à chaud » (c'est-à-dire au cours de l'exécution du logiciel). Ainsi, nous pouvons également parler d'adaptation statique dans le premier cas (c'est-à-dire en dehors de l'exécution) et d'adaptation dynamique dans le second (c'est-à-dire pendant l'exécution). Évidemment, l'adaptation dynamique est la plus complexe à mettre en œuvre mais offre l'avantage de ne pas interrompre le service proposé par l'application adaptative. D'autre part, nous distinguons différentes dimensions d'adaptations :

- l'adaptation fonctionnelle<sup>14</sup> ou non-fonctionnelle : l'adaptation peut modifier ou ajouter des fonctionnalités ou en revanche concerner uniquement la qualité de service des fonctionnalités existantes
- l'adaptation prévue à l'avance ou non-anticipée : l'adaptation est « précâblée » dans le système dès le départ ou en revanche nous pouvons gérer des cas non prévus à l'avance
- l'adaptation prédictible ou non-déterministe : l'adaptation amène le système dans un état bien défini et stable ou en revanche laisse place à une certaine incertitude
- l'auto-adaptation ou l'adaptation par un tiers : l'adaptation est réalisée par le système lui-même ou en revanche par une entité extérieure qui agit sur le système

14. La différence que nous faisons entre l'adaptation fonctionnelle et non-fonctionnelle semble coïncider avec celle faite par Capra entre l'adaptation réactive et proactive.

- l'adaptation générique ou métier : l'adaptation est agnostique (s'applique à tout) ou en revanche dédiée à un contenu métier particulier

Dans cette thèse, c'est l'adaptation dynamique qui va nous intéresser et il sera sous-entendu qu'elle est dynamique lorsque ce n'est pas précisé. En outre, le défi majeur est clairement de pouvoir réaliser une auto-adaptation fonctionnelle, non-anticipée, prédictible et générique. En effet, l'auto-adaptation a l'avantage de ne plus avoir besoin de faire intervenir l'être humain pour des opérations de maintenance (dans la même idée que l'informatique autonome). Concernant l'adaptation fonctionnelle, il semble assez évident qu'un logiciel qui ne fait que changer la qualité de service de ses fonctionnalités existantes est plutôt limité et se trouvera rapidement dans une impasse pour trouver une solution d'adaptation acceptable dans certains contextes. Au niveau de la non-anticipation des adaptations, cela revient à bénéficier d'un programme intelligent qui n'a pas besoin de tout savoir sur le contexte au départ et qui va le découvrir au cours de son exécution et prendre une décision d'adaptation appropriée. L'adaptation prédictible, quant à elle, nous assure une certaine stabilité au niveau du raisonnement du système qui sous des conditions identiques, une fois relancé, doit réagir exactement de la même façon. Enfin, la généricité de l'adaptation est un point clé puisque cela permet de réutiliser nos adaptations quel que soit le contenu métier que traite l'application considérée. C'est donc sur ce type d'adaptation logicielle que nous allons nous focaliser dans les prochains chapitres.

### 2.3.2 Le contexte

Comme indiqué en début de section, le contexte est un concept essentiel dans le domaine de l'adaptation logicielle puisque nous nous adaptons en fonction de celui-ci lorsqu'il a changé. Ces modifications dans le contexte surviennent généralement lorsque de nouvelles informations sont obtenues par des capteurs et transmises au système logiciel. Cependant, certaines applications adaptatives peuvent se passer de la présence de ce matériel dès lors qu'elles ont déjà accès aux données du contexte. Nous allons voir ici des exemples concrets de contextes pour des programmes adaptatifs.

Dans [51], l'application adaptative est un système de gestion dynamique de la relation client. Son contexte correspond à un triplet : la position physique de son utilisateur (au bureau, au volant ou en rendez-vous), l'équipement dont il dispose (un ordinateur de l'entreprise, son smartphone ou un téléphone classique) et la bande passante disponible (de 0% à 100%). Un système de domotique joue cette fois le rôle de l'application adaptative dans [15]. Le contexte de ce logiciel est obtenu à l'aide des capteurs de mouvement placés à l'intérieur de l'habitation. Ici, le contexte correspond à la présence ou non d'une personne dans la maison. Dans [33], l'exemple de système adaptatif considéré est celui d'un robot de combat. Le logiciel contrôlant le robot obtient son contexte grâce à un capteur. Le contexte pour le robot permet de connaître son niveau d'énergie ou encore le nombre de robots ennemis restant sur le terrain de combat. Les travaux réalisés dans [44] sont illustrés par un système de mises à jour automatiques. L'application adaptative gère automatiquement l'installation et la configuration des logiciels. Dans cet exemple, le contexte correspond aux mises à jour disponibles pour les programmes. Dans [7], l'application adaptative est un système d'alerte de crue. Des capteurs communiquant au travers des réseaux sans fil transmettent au système adaptatif des informations nécessaires à l'obtention du contexte. Ici, le contexte de ce logiciel permet de savoir si une crue est prévue ou si le courant de l'eau est élevé. Un système de diffusion audio par *GSM* est présenté dans [85]. Le programme adaptatif reçoit la diffusion audio au travers d'un réseau sans fil avec perte. Le contexte est ici le niveau du taux de perte qui peut être

faible ou élevé. Dans [12], le cas d'utilisation considéré est celui d'un train circulant sur des voies ferrées. Le logiciel adaptatif est amené à modifier la vitesse du train en fonction de la signalisation inconnue au départ. Son contexte correspond aux signaux de couleurs différentes qu'il croise. L'exemple d'une application adaptative de contrôle du chauffage est présenté dans [21]. L'idée est de chauffer la maison lorsque l'utilisateur est en train de rentrer chez lui. Ainsi, une fois arrivé à destination, la maison est déjà à bonne température. Ici, le contexte est composé de deux éléments : la position *GPS* de l'utilisateur et la température de la maison.

### 2.3.3 Les politiques d'adaptation

Pour parvenir à adapter un logiciel nous avons d'abord besoin de décrire cette adaptation. Pour spécifier quand et comment l'application doit s'adapter, des règles doivent donc être écrites. Ces règles s'appellent des « politiques d'adaptation » et permettent d'indiquer le comportement adaptatif du programme (c'est-à-dire quelles conditions dans le contexte entraînent quelles actions). Ces politiques d'adaptation peuvent être écrites soit « en dur » à l'intérieur du code du logiciel, soit de façon externalisée dans un langage particulier. Un premier avantage des politiques d'adaptation externalisées est qu'elles peuvent être remplacées par d'autres sans avoir à intervenir sur le code de l'application adaptative, ce qui offre une certaine souplesse à l'utilisation. Un deuxième avantage de cette externalisation est que cela permet de mettre en place une séparation des préoccupations avec d'un côté l'exécution et de l'autre la réalisation de l'adaptation. Nous allons voir ici quelques politiques d'adaptation tirées de la littérature et exprimées avec un langage spécifique : un *Domain-Specific Language (DSL)*.

Dans [33], les politiques d'adaptation sont utilisées pour changer le comportement d'un robot de combat. Elles viennent modifier le modèle architectural embarqué à l'exécution en remplaçant un composant par un autre si une condition particulière est vérifiée dans le contexte. Dans l'exemple proposé, une politique d'adaptation appelée *ReplaceFiring* consiste à remplacer le composant de tir du robot (cf. listing 2.1). Lorsque l'énergie du robot est inférieure à soixante unités, le composant *ReactiveFire* est remplacé par le composant *DistanceFire*. Ici, le mot-clé **Observation** permet de spécifier une condition tandis que **Response** indique une action à réaliser pour l'adaptation.

```
1 (AdaptationPolicy ReplaceFiring
2   (Description "Replaces firing component")
3   (Observation energyReport (energy < 60))
4   (Response RemoveComponent ReactiveFire)
5   (Response AddComponent DistanceFire)
6   DistanceFire_type SequencingConnector ReactiveConnector)
```

Listing 2.1 – Une politique d'adaptation pour changer le comportement d'un robot de combat

Les politiques d'adaptation sont appelées des politiques de reconfiguration dans [6]. Ces politiques appliquent une action lorsque des événements particuliers dans le contexte du système considéré se produisent. Ces actions correspondent à des changements appliqués au modèle architectural du système. Le cas d'utilisation choisi est celui d'un système de découverte dynamique de services destiné aux matériels portatifs (*PDA*, téléphones mobiles ou ordinateurs portables) qui sont considérés comme des nœuds au sein d'un réseau. Trois agents (qui peuvent être vus comme des rôles pour ces nœuds) sont supportés par le système : le *UserAgent (UA)* qui correspond au client qui demande une inscription à un service, le

*ServiceAgent* (*SA*) qui présente la liste des services disponibles et le *DirectoryAgent* (*DA*) qui répond à la demande du *UA* en fonction de la liste du *SA*. Dans l'exemple proposé, une politique présentée indique que si un nœud dont le rôle est *UA* est élu pour tenir le rôle d'un *DA*, ce nœud doit être reconfiguré pour correspondre à ce nouveau rôle. Cette politique est donnée à la fois en pseudo-code et dans le format *XML* (cf. listings 2.2 et 2.3). Au niveau du pseudo-code, les conditions sont placées classiquement entre parenthèses après le mot-clé **if** tandis que les actions à réaliser pour l'adaptation sont situées juste après **then**. Concernant le contenu du fichier *XML*, la balise `<Event>` signale une condition alors que la balise `<Reconfiguration>` indique une action.

```

1 if (Elected-DA) then
2   reconfigure(UA,DA)
3 end

```

Listing 2.2 – Une politique d'adaptation pour changer le rôle d'un nœud (pseudo-code)

```

1 <ReconfigurationRule>
2 <Framework>SDP</Framework>
3 <Event>
4   <Type>Elected-DA</Type>
5   <Value>True</Value>
6 </Event>
7 <Reconfiguration>
8   <FileType>Java</FileType>
9   <Name>Reconfiguration.DA</Name>
10 </Reconfiguration>
11 </ReconfigurationRule>

```

Listing 2.3 – Une politique d'adaptation pour changer le rôle d'un nœud (format *XML*)

Dans [31], les politiques d'adaptation sont nommées des tactiques et sont elles-mêmes encapsulées dans des stratégies. Elles sont spécifiées à l'aide du langage *Stitch* [16] qui permet de décrire quelles conditions impliquent quelles actions. L'exemple considéré est celui d'un système qui diffuse des informations de type multimédia à des clients via des serveurs. Une tactique présentée est que si la bande passante utilisée dans le contexte est trop élevée alors les serveurs diffusent du contenu texte au lieu de multimédia aux clients (cf. listing 2.4). Ici, les blocs qui suivent les mots-clés **condition** et **action** donnent respectivement les conditions et actions de la politique d'adaptation. Le dernier bloc **effect**, quant à lui, n'est là que pour indiquer des post-conditions qui doivent être vérifiées après l'adaptation.

```

1 module newssite.tactics.example;
2 import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
3 import op "newssite.operator.ArchOp" { ArchOp as Sys };
4
5 tactic switchToTextualMode() {
6   condition {
7     exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;
8   }
9   action {

```

```
10 svrs = { select s : T.ServerT | !s.isTextualMode };
11 for (T.ServerT s : svrs) {
12   Sys.setTextualMode(s, true);
13 }
14 }
15 effect {
16   forall c : T.ClientT in M.components | c.experRespTime <= M.MAX_RESPTIME;
17   forall s : T.ServerT in M.components | s.isTextualMode;
18 }
19 }
```

Listing 2.4 – Une politique d’adaptation pour changer le mode de diffusion des informations

Le système *Self-Adaptive FRactal compoNents (SAFRAN)* [23] propose une extension du modèle de composants *Fractal* afin de supporter les composants adaptatifs. Pour cela, un langage dédié inclus dans ce système permet l’écriture de politiques d’adaptation. Une politique d’adaptation est constituée d’un ensemble de règles d’adaptation. Un des exemples proposés est celui d’une application de lecture de courrier électronique. La politique d’adaptation associée au sous-composant **connexion** de l’architecture du logiciel contient deux règles (cf. listing 2.5). La première se déclenche lorsque nous sommes déconnectés du réseau. L’action consiste à activer le mode déconnecté, c’est-à-dire intégrer un nouveau méta-composant **deferred-messages** dans le programme et désactiver la réception de courrier en affectant la valeur 0 au paramètre `pollInterval`. La deuxième règle se déclenche lorsque nous sommes de nouveau connectés au réseau. L’action consiste cette fois à activer le mode connecté, c’est-à-dire stopper le méta-composant, le déconnecter et rétablir la réception de courrier en affectant la valeur  $5 * 60$  au paramètre `pollInterval` (pour une réception de courrier toutes les cinq minutes). Ici, pour chaque règle, le mot-clé **when** permet de spécifier une condition tandis que **do** indique une action à réaliser pour l’adaptation.

```
1 policy disconnected-mode = {
2   rule {
3     when disappears(sys://network/interfaces/eth0)
4     do {
5       if ($target/sibling::deferred-messages) then {
6         set-meta($target, $target/sibling::deferred-messages);
7       }
8       else {
9         meta := new("deferred-messages");
10        foreach p in $target/parent::* {
11          add($p, $meta);
12        }
13        set-meta($target, $meta);
14      }
15      start($target/meta::*);
16      set-value($target/@pollInterval, 0);
17    }
18  }
```

```

19 rule {
20   when appears(sys://network/interfaces/eth0)
21   do {
22     if ($target/meta::*) {
23       stop($target/meta::*);
24       unset-meta($target);
25     }
26     set-value($target/@pollInterval, 5*60);
27   }
28 }
29 }

```

Listing 2.5 – Une politique d’adaptation pour changer le mode de connexion réseau

### 2.3.4 Les boucles d’adaptation

Au cours de l’exécution d’un logiciel adaptatif, celui-ci doit vérifier continuellement s’il est adapté à son contexte. Lorsqu’il s’avère que le système doit subir une adaptation, la modification qui correspond est réalisée conformément aux politiques d’adaptation définies. Ce processus appelé « boucle d’adaptation » est implémenté de plusieurs façons (c’est-à-dire avec plus ou moins d’étapes). Nous allons voir ici ces différentes implémentations de la boucle d’adaptation que nous pouvons retrouver dans la littérature sur l’adaptation logicielle.

Une boucle d’adaptation nommée « cycle d’autogestion » (*self-management cycle*) est présentée dans [72] (cf. figure 2.4). Celle-ci est constituée de trois étapes : surveillance (*monitoring*), vérification et diagnostic (*checking and diagnosis*) et reconfiguration. Le première étape consiste à observer le système. Cette surveillance est réalisée par l’intermédiaire de capteurs et correspond à l’obtention du contexte telle que décrite dans la sous-section 2.3.2. Une fois ces données récupérées, elles sont assemblées et vérifiées afin d’établir un diagnostic. Dans certaines conditions, le diagnostic identifie la nécessité d’une reconfiguration du système.

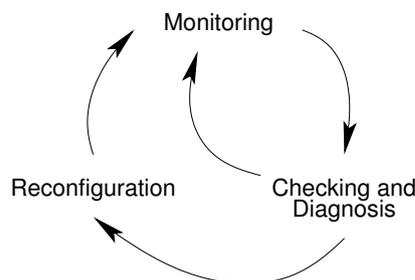


FIGURE 2.4 – La boucle d’adaptation du cycle d’autogestion

Dans [44, 40, 31, 50], une boucle d’adaptation plus complexe appelée « la boucle *MAPE* » est présentée (cf. figure 2.5). Son nom vient du fait qu’elle est composée des étapes : surveiller (*Monitor*), analyser (*Analyze*), planifier (*Plan*) et exécuter (*Execute*). Elle est contenue dans un élément autonome (un concept venant du domaine de l’informatique autonome) et chaque étape est réalisée par le gestionnaire d’autonomie (*autonomic manager*) qui s’appuie sur les connaissances (*knowledge*) concernant le contexte. Une

fois ce contexte obtenu et analysé, l'élément géré peut être adapté lorsque c'est nécessaire en exécutant les plans d'adaptation établis. Une version modifiée de cette boucle d'adaptation, nommée cette fois « la boucle de *feedback* » est présentée dans [83, 82]. La principale différence avec la version originale de cette boucle est que la notion abstraite de *knowledge* est remplacée par un ensemble de modèles.

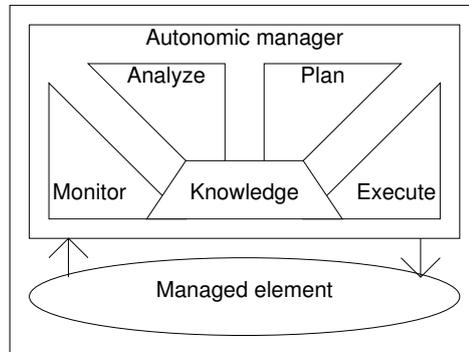


FIGURE 2.5 – La boucle d'adaptation *MAPE*

Les trois boucles de l'approche *Architectural Runtime Configuration Management (ARCM)* sont présentées dans [33] (cf. figure 2.6). Celle de gauche correspond à une boucle d'adaptation classique qui observe le contexte du système puis applique des changements sur celui-ci en s'appuyant sur des politiques d'adaptation (comme celle du listing 2.1). Celle d'en haut permet à l'humain d'intervenir indirectement sur le système au cours de son exécution en modifiant le modèle architectural. Quant à celle de droite, elle sert à synchroniser le modèle et le système pour conserver une cohérence entre les deux. Même si la première boucle est similaire à celle de notre approche, les deux dernières boucles ne correspondent pas à la boucle d'adaptation que nous implémentons dans cette thèse. En effet, la boucle de droite n'a pas de raison d'être dans nos travaux puisque nous considérons le modèle et le système comme une même entité. Quant à celle d'en haut, nous rappelons que nous nous intéressons uniquement aux adaptations réalisées par le système lui-même.

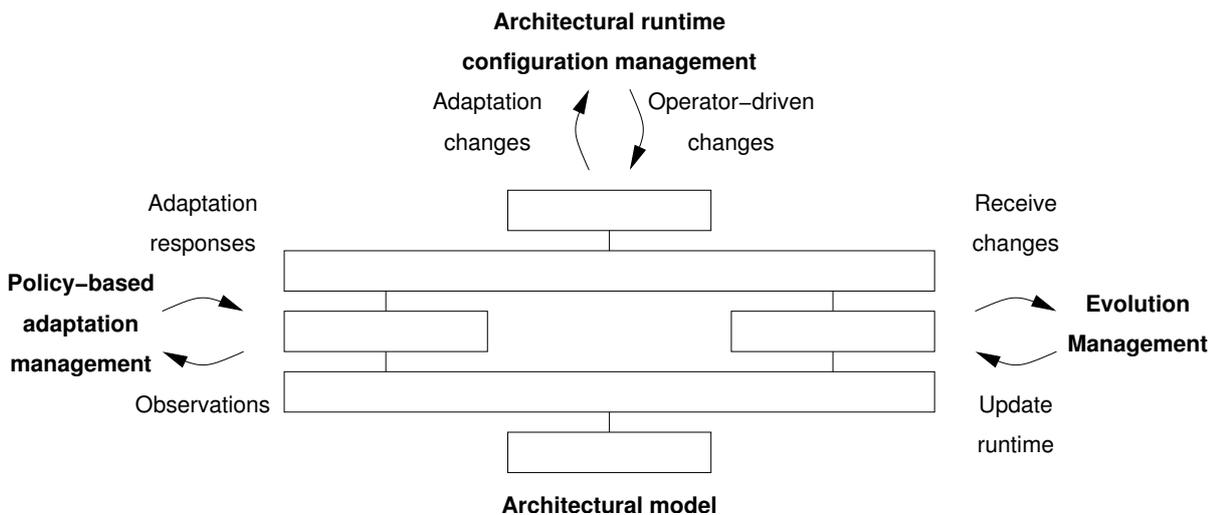


FIGURE 2.6 – Les trois boucles d'adaptation de l'approche *ARCM*

Dans [12], nous montrons la boucle d'adaptation qui correspond au cas classique du *models@run.time*

que nous détaillons dans la sous-section suivante. Comme le montre la figure 2.7, pendant l'exécution nous avons un système adaptatif exécuté et un modèle embarqué qui contient des informations sur l'état du système et qui permet de s'interroger sur la nécessité d'adapter ce système. Si une adaptation est nécessaire alors une modification du système est réalisée. Nous attirons l'attention sur le fait qu'aucun modèle n'est exécuté avec cette approche.

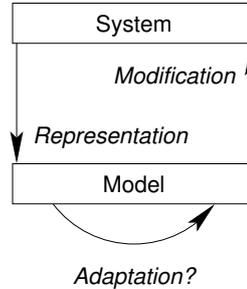


FIGURE 2.7 – La boucle d'adaptation du *models@run.time* classique

### 2.3.5 Le *models@run.time*

La technique du *models@run.time* [8] est une approche issue de l'IDM et permettant d'adapter à l'exécution un système à l'aide d'un modèle. Ce modèle décrit l'état du système à l'exécution et permet de déterminer s'il est temps d'adapter. En d'autres termes, ce modèle peut être interrogé pour savoir si le système est correctement configuré par rapport à son contexte. Si des modifications sont apportées au modèle, elles sont répercutées indirectement sur le système en cours d'exécution. En effet, ce modèle n'est pas exécuté et ne correspond pas au système lui-même mais à une représentation abstraite de celui-ci. La technique du *models@run.time* est parfois utilisée dans le domaine de l'informatique autonome pour gérer ces systèmes adaptatifs. Le modèle embarqué à l'exécution correspond souvent à un modèle architectural [66, 26, 31] ou à un modèle de fonctionnalités [15] qui est modifié dynamiquement. Nous allons voir dans cette sous-section des exemples d'application des principes du *models@run.time* trouvés dans la littérature sur l'adaptation de modèles.

Un *DSML* utilisé pour construire le modèle permettant l'adaptation est présenté dans [25]. Il contient à la fois des informations concernant le contexte et les politiques d'adaptation. L'exemple considéré est celui d'un robot explorateur qui se déplace dans un contexte inconnu et qui doit construire une carte du terrain. Ce robot est équipé de capteurs donnant des informations utiles pour le choix du chemin à suivre mais aussi pour le dessin de la carte. En fonction du contexte, le robot doit dynamiquement s'adapter en choisissant les capteurs et algorithmes appropriés à la situation. Le contexte étant inconnu en phase de conception, le modèle est modifié au cours de l'exécution du système adaptatif pour ajuster les valeurs qu'il contient.

Dans [33], un modèle architectural embarqué à l'exécution représente les différents composants attachés à un robot de combat. Suite à un changement dans le contexte du système indiqué par un capteur, une politique d'adaptation (cf. listing 2.1) est appliquée pour remplacer le composant de tir dans le modèle architectural. La figure 2.8 montre le modèle avant et après modification. Une fois cette modification appliquée, elle doit ensuite être répercutée sur le système exécuté afin de conserver une cohérence entre le modèle et le système comme l'indiquent les boucles d'adaptation de la figure 2.6.

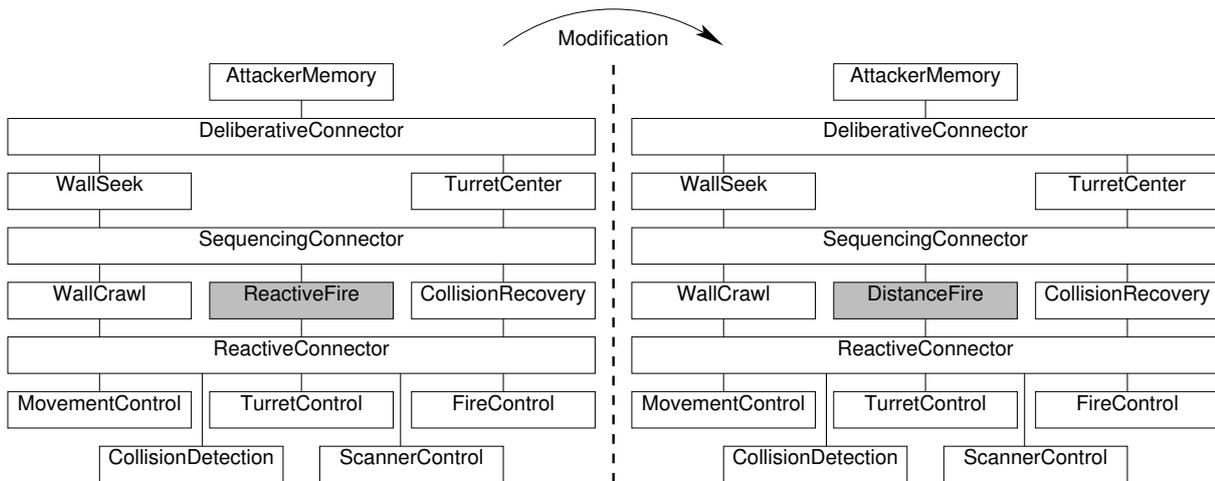


FIGURE 2.8 – Une modification du modèle architectural embarqué à l'exécution

Un modèle de fonctionnalités embarqué à l'exécution et représentant la configuration d'une maison concernant ses besoins en domotique est présenté dans [15]. Lorsque des capteurs signalent dans le contexte du système que l'utilisateur est absent de la maison, une politique d'adaptation est appliquée pour changer la configuration du modèle de fonctionnalités. La figure 2.9 montre le modèle avant et après modification. Nous voyons que comme l'occupant n'est plus dans l'habitation, la fonctionnalité de l'éclairage (*lighting by occupancy*) est désactivée car il n'y a plus de raison d'éclairer les pièces de la maison. Par contre, comme la personne est absente de la demeure, la fonctionnalité de détection d'intrusion (*in-home detection*) est activée par mesure de sécurité. Aussi, pour simuler la présence d'une personne et ainsi dissuader quiconque souhaite s'introduire dans la maison, la fonctionnalité de simulation d'occupation (*occupancy simulation*) est activée.

Dans [31], le cas d'utilisation considéré est celui d'un système qui diffuse des informations de type multimédia à leurs clients. Un ensemble de clients envoie des requêtes à un ensemble de serveurs qui répond. Ici, le contexte correspond à la bande passante utilisée par les connexions entre les clients et les serveurs. Une première idée concernant l'adaptation est d'ajouter des serveurs quand la bande passante consommée est trop importante ou de réduire le nombre de serveurs dans le cas contraire. Une deuxième adaptation possible est de changer le type de contenu en diffusant des informations de type texte au lieu de multimédia lorsque l'utilisation de la bande passante est trop élevée (cf. listing 2.4). Le modèle architectural du système est embarqué à l'exécution et contient des informations concernant l'état des serveurs. Il est régulièrement mis à jour pour maintenir une cohérence avec le système en cours d'exécution.

Le modèle architectural d'un système de logistique permettant de planifier la route empruntée par des véhicules pour amener une cargaison à des entrepôts est présenté dans [66]. Des modifications appliquées à ce modèle architectural embarqué, telles que le remplacement d'un composant ou d'un connecteur par un autre sont réalisées durant l'exécution de l'application adaptative en fonction des changements dans le contexte. Ces modifications sont ensuite répercutées sur le système grâce à la boucle d'adaptation qui applique des politiques d'adaptation.

Dans [51], pour mettre en œuvre un système de gestion dynamique de la relation client, quatre modèles sont embarqués à l'exécution. Un modèle de fonctionnalités représente les variabilités du système, un modèle indique le contexte récupéré à l'aide de capteurs, un modèle spécifie les politiques d'adaptation

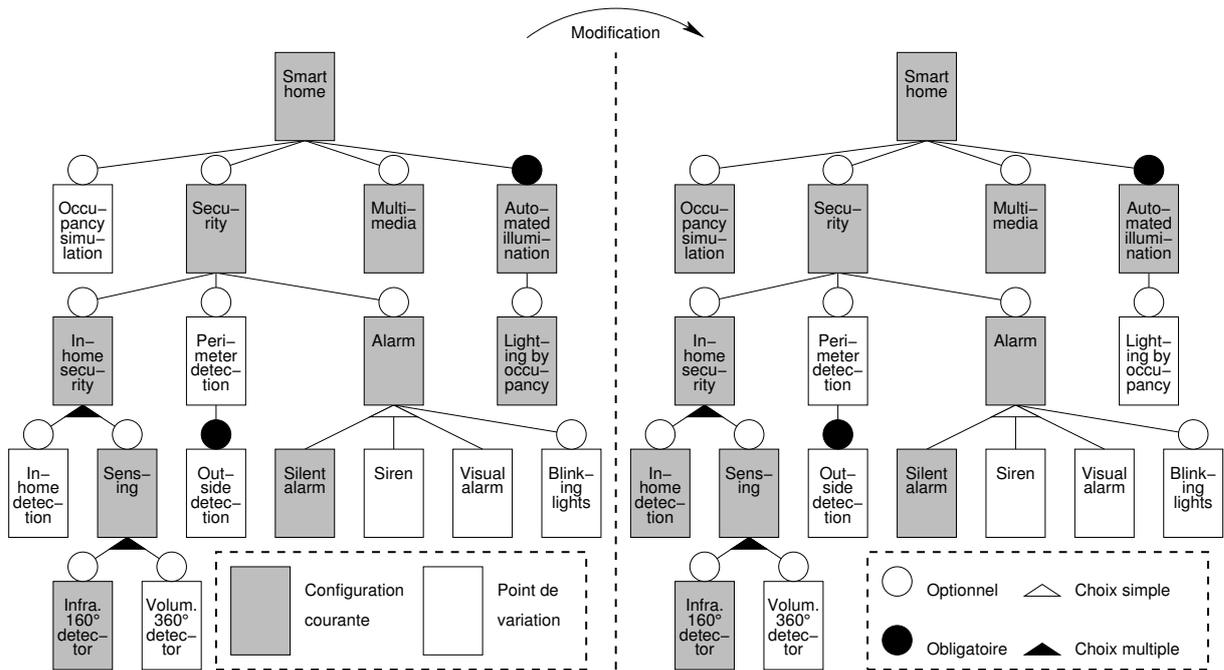


FIGURE 2.9 – Une modification du modèle de fonctionnalités embarqué à l'exécution

et un modèle architectural décrit les composants du système. Le modèle architectural est modifié durant l'exécution du système en ajoutant ou en retirant des composants lorsqu'une adaptation est nécessaire en fonction du contexte.

## 2.4 L'adaptation de l'exécution de modèles

La section précédente a traité de l'adaptation logicielle de façon générale (c'est-à-dire sans se préoccuper de l'exécution de modèles). Dans cette section, nous allons étudier une adaptation plus spécifique : celle de l'exécution de modèles. Nous allons voir à la fois des approches hors IDM telles que les réseaux de Petri adaptatifs, les *workflows* adaptatifs ou encore les machines à états adaptatives mais aussi et surtout des approches qui s'inscrivent totalement dans le domaine de l'IDM où bien trop peu de travaux ont traité cette idée en profondeur.

### 2.4.1 Les modèles de processus adaptatifs hors IDM

Nous avons vu dans la sous-section 2.2.1 qu'il existe depuis bien longtemps des processus (réseaux de Petri, *workflows* et machines à états) reposant sur une exécution de modèles. Dans cette sous-section, nous allons nous intéresser plus spécifiquement à l'adaptation de l'exécution de ces modèles.

Des chercheurs ont souhaité rendre adaptables les réseaux de Petri (c'est-à-dire que le comportement initial spécifié par le réseau de Petri est modifié dynamiquement). Dans [85], les réseaux de Petri sont utilisés pour modéliser le comportement de programmes adaptatifs. Pour cela, un modèle source et un modèle cible sont définis et représentent respectivement le programme avant et après adaptation. Un modèle d'adaptation est ensuite construit pour connecter le modèle source au modèle cible. L'exemple de processus considéré est celui d'un système de diffusion audio par *GSM*. Le réseau de Petri source

correspond à un codage *GSM* destiné à un réseau dont le taux de perte est faible tandis que le réseau de Petri cible correspond à un codage *GSM* approprié à un réseau dont le taux de perte est élevé. Une fois les trois réseaux de Petri créés (cf. figures 2.10, 2.11 et 2.12), l'adaptation est rendue possible. En effet, dès que le taux de perte sur le réseau devient important, en passant par la transition `adapt`, le codage *GSM* pour la diffusion change. Pour parvenir à une adaptation, les réseaux de Petri peuvent également être étendus comme dans [3]. Ainsi, en apportant une notion de temps à ces modèles, il devient possible de mieux contrôler le déclenchement des transitions dans le réseau de Petri. Typiquement, si le temps écoulé dépasse le temps spécifié sur les arcs du réseau de Petri, la transition associée ne peut plus être déclenchée. D'autre part, en ajoutant le concept de « composant dégradable », il est possible de définir un ensemble de réseaux de Petri qui partagent une même interface et passer ainsi de l'un à l'autre lorsqu'une adaptation est nécessaire.

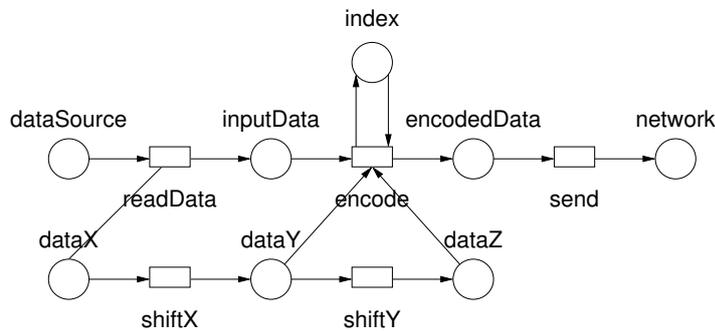


FIGURE 2.10 – Le réseau de Petri source

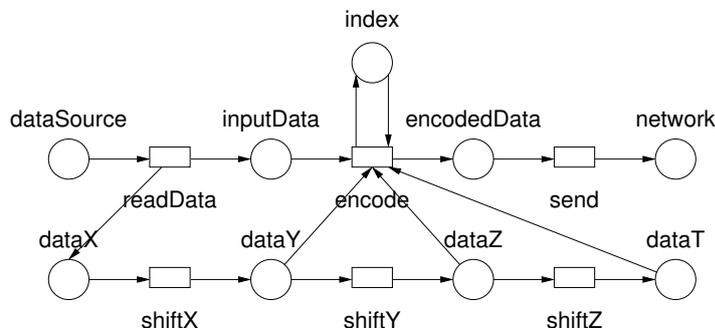


FIGURE 2.11 – Le réseau de Petri cible

Comme pour les réseaux de Petri, les *workflows* peuvent être étendus pour les rendre adaptables. Ainsi, les processus définis grâce à ces *workflows* adaptatifs peuvent être adaptés durant leur exécution [71]. L'orchestration ou la chorégraphie de services Web reposant sur le principe des *workflows* adaptatifs offrent également ces possibilités d'adaptation de processus [43, 52, 46]. Ces adaptations peuvent servir par exemple à remplacer un service par un autre pour des raisons de qualité de service. Dans [73], trois catégories de *workflows* adaptatifs sont mises en évidence : les *workflows* dynamiques, adaptatifs et flexibles. Les *workflows* dynamiques sont ceux qui changent quand le processus évolue pour être amélioré. Les *workflows* adaptatifs, quant à eux, correspondent à ceux qui ont la capacité de réagir à des circonstances exceptionnelles. Enfin, les *workflows* flexibles sont ceux qui peuvent être exécutés alors que le processus est partiellement spécifié. La spécification complète du modèle est donnée plus tard,

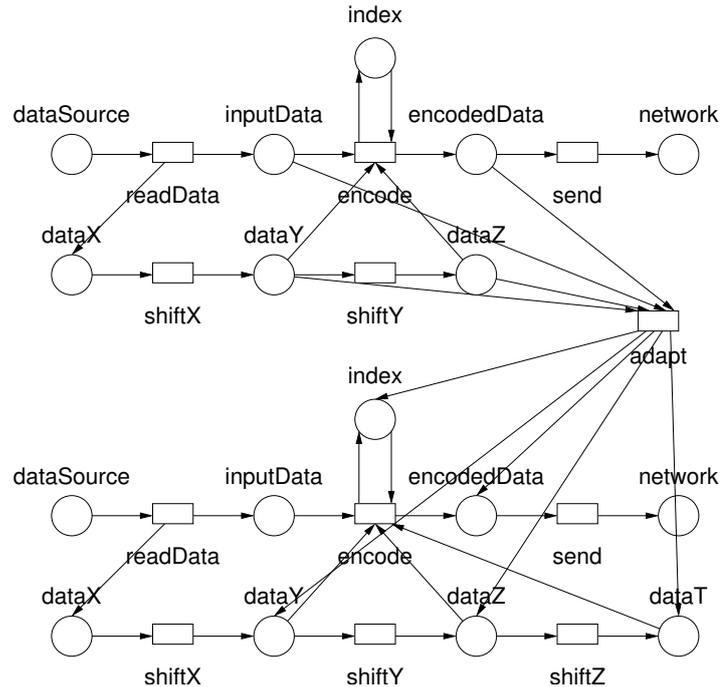


FIGURE 2.12 – Le réseau de Petri d'adaptation

au cours de l'exécution. Cette dernière catégorie de *workflows* adaptatifs est illustrée par un exemple de processus représentant le diagnostic du cancer du sein pour un patient, construit avec un langage de *workflows* générique [74]. Ici, la particularité de l'approche est de se servir d'une « poche de flexibilité » qui peut être vue comme une activité spéciale contenant un ensemble de spécifications possibles pour ce *workflow*. Une adaptation présentée consiste à remplacer la spécification partielle par une spécification complète indiquant que nous devons seulement faire subir les ultrasons au patient (cf. figure 2.13). Une liste d'ajouts et de suppressions d'éléments est utilisée dans [49] pour décrire les étapes d'adaptation du modèle de *workflow* adaptatif. L'exemple considéré est le *workflow* adaptatif d'un processus de mises à jour automatiques pour le système d'exploitation *Windows*, construit à l'aide des diagrammes d'activité du langage *UML*, qui sont tout à fait appropriés à la représentation de processus. Si pendant l'évaluation des prochaines mises à jour, le lecteur *CD-ROM* ne fonctionne plus, deux activités sont ajoutées au modèle de *workflow* pour réinstaller le pilote du lecteur *CD-ROM* et le tester (cf. figure 2.14).

Les machines à états ont également suscité un intérêt particulier pour l'adaptation. Dans [42], Junger propose de modifier ces machines à états adaptatives au cours de leur exécution. Il indique certaines règles que doit respecter un interpréteur *SCXML* pour gérer correctement ces mutations qui interviennent sur la machine à états. Nous montrons dans [5] une machine à états représentant un système de gestion de crises lors d'accidents de voitures. Les pompiers et les policiers décident ensemble de la route à suivre pour positionner leurs véhicules afin de gérer la crise. En temps normal, si l'une des deux routes est rejetée, cela n'a aucune incidence sur l'acceptation ou le rejet de l'autre route. Cependant, nous décidons de changer ce comportement de la machine à états qui doit dorénavant rejeter automatiquement la route proposée par les policiers lorsque celle des pompiers a été rejetée. Le processus d'adaptation que nous mettons en place consiste à retirer et à ajouter une transition au cours de l'exécution de la machine à états en passant par le moteur d'exécution *PauWare engine* et le formalisme *SCXML*.

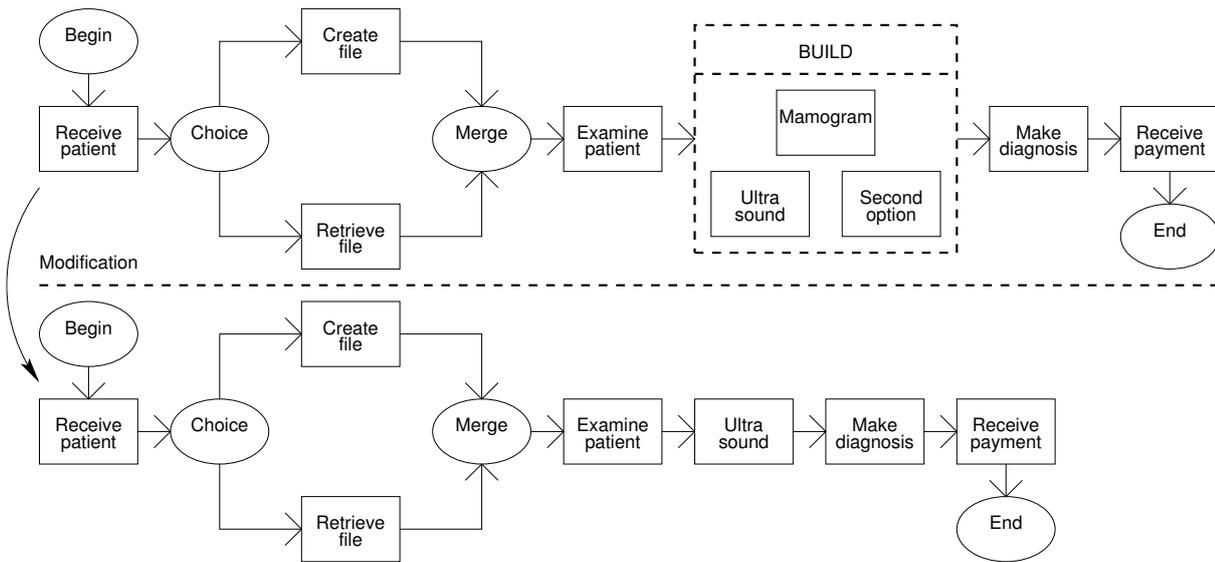


FIGURE 2.13 – Une modification du modèle d'un *workflow* adaptatif avec une « poche de flexibilité »

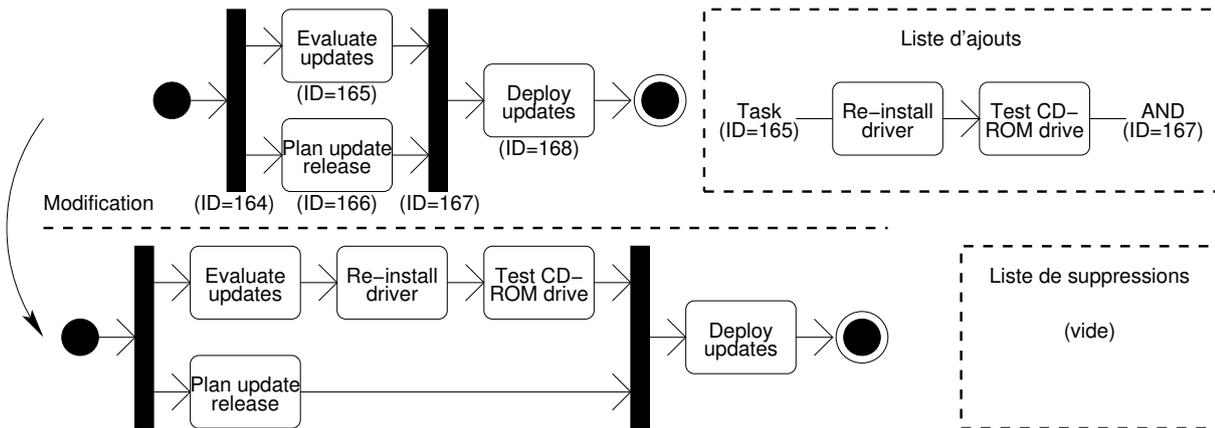


FIGURE 2.14 – Une modification du modèle d'un *workflow* adaptatif avec une liste d'ajouts/suppressions

## 2.4.2 L'adaptation de l'exécution de modèles en IDM

Nous avons vu dans la sous-section 2.2.2 que l'IDM favorise grandement la mise en place d'une exécution de modèles, notamment grâce aux facilités qu'elle offre pour la construction de *DSML* compilés ou interprétés. Dans cette sous-section, nous allons nous intéresser plus spécifiquement à l'adaptation de l'exécution de ces modèles.

Nous avons mis en évidence dans la sous-section 2.3.5 que le *models@run.time* ne réalise pas d'exécution de modèles. Lehmann *et al.* [45] mettent en place une boucle d'adaptation légèrement différente de celle sur la figure 2.7 puisque le modèle embarqué est cette fois un modèle en cours d'exécution (cf. figure 2.15). Nous avons donc à la fois une exécution de modèles et une adaptation de l'exécution de ce modèle. Cependant, avec cette solution, de la même manière qu'avec le *models@run.time* classique, puisque le modèle en cours d'exécution et le système sont deux entités distinctes, ce modèle est toujours lié de manière causale avec le système en cours d'exécution. De plus, les auteurs ne vont pas aussi loin

que nous à propos des éléments qui peuvent être modifiés par l'adaptation : ils ne considèrent pas que les sémantiques d'exécution et d'adaptation peuvent être changées.

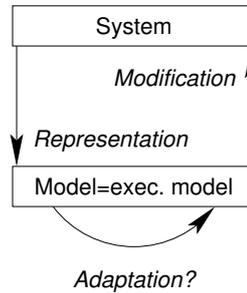


FIGURE 2.15 – La boucle d'adaptation du *models@run.time* avec un modèle exécuté

Dans [12], nous montrons une autre façon d'appliquer les principes du *models@run.time* tout en profitant d'une exécution de modèles (cf. figure 2.16). De cette manière, nous avons deux modèles au cours de l'exécution du système : celui qui sert à se demander s'il est temps d'adapter et qui n'est pas exécuté ainsi que celui qui correspond au système et qui est exécuté. À notre connaissance, le seul exemple qui met en pratique cette technique est la plateforme *Model Of Components for Adaptive Systems (MOCAS)* [4] qui permet l'adaptation d'une machine à états *UML* en cours d'exécution. Cette machine à états est observée par une autre machine à états qui a la charge de gérer l'adaptation de la première. Conceptuellement, la première représente la partie métier de l'application et la seconde est un agent chargé de l'adaptation de la partie métier. Ainsi, nous avons avec *MOCAS* une plateforme permettant à la fois d'exécuter une machine à états et de l'adapter via un modèle l'observant dans l'esprit des principes du *models@run.time* où un système est représenté par un modèle. La limite notable de cette approche est que le système et le modèle ne sont pas bien dissociés. La logique métier et la logique d'adaptation via les deux machines à états restent entremêlées. De plus, la plateforme ne permet pas de remplacer, au cours de l'exécution, les sémantiques d'adaptation ou d'exécution (contrairement à nos travaux, comme nous l'expliquerons dans le chapitre 3).

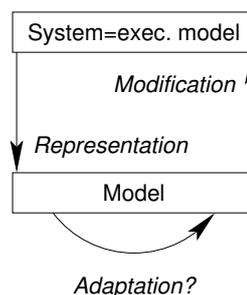


FIGURE 2.16 – La boucle d'adaptation du *models@run.time* pour un modèle exécuté

Toujours dans [12], nous proposons une alternative à la boucle d'adaptation de la figure 2.16 où le système reste un modèle exécuté (cf. figure 2.17). La différence de taille est que nous nous éloignons fortement des principes du *models@run.time* puisqu'il n'y a plus de modèle embarqué dédié à l'adaptation et qui n'est pas exécuté. En effet, ici nous avons un unique modèle qui est à la fois le système et le modèle que nous pouvons questionner pour l'adaptation. Même si cette solution présente l'inconvénient

de compliquer le modèle qui doit désormais contenir à la fois des informations d'exécution et d'adaptation, elle simplifie considérablement la boucle et fait disparaître le problème épineux du lien de causalité. Ainsi, l'adaptation de l'exécution de modèles est directe puisque lorsque nous modifions le modèle, nous modifions *de facto* le système. D'autre part, lorsque nous utilisons deux modèles à l'exécution (celui du système et celui pour l'adaptation), nous trouvons des redondances d'informations entre ces modèles. Utiliser un unique modèle nous évite donc d'introduire plusieurs fois les mêmes informations. Dans cette thèse, nous allons discuter et implémenter ce cas qui nous semble être plus intéressant que les autres approches pour les raisons évoquées précédemment. Le modèle sera conçu à l'aide d'un *i-DSML* adaptable dont la caractérisation est donnée dans le prochain chapitre.

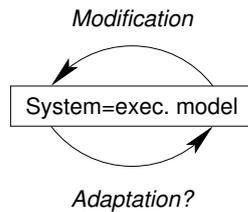


FIGURE 2.17 – La boucle d'adaptation directe de l'exécution d'un modèle

## 2.5 Bilan

Dans cet état de l'art, après avoir effectué quelques rappels sur l'IDM, nous avons présenté l'exécution de modèles, l'adaptation logicielle et finalement l'adaptation de l'exécution de modèles. L'IDM offre un cadre pour l'ingénieur logiciel qui repose sur une pile de méta-modélisation. L'idée sous-jacente à cette pile de méta-modélisation est de contraindre un modèle à une relation de conformité avec son méta-modèle (idem pour un méta-modèle et son méta-méta-modèle). De cette manière, nous pouvons nous assurer de travailler dans un environnement stable et propice à la construction de *DSML*. Le méta-modèle de ces *DSML* qui constitue la syntaxe abstraite du langage est généralement accompagné d'une ou plusieurs syntaxes concrètes (graphiques ou textuelles) qui aident à représenter les instances des méta-éléments en phase de modélisation. L'IDM apporte avec elle de nombreux outils permettant de manipuler nos modèles en appliquant par exemples des transformations *M2M* ou *M2T*.

Concernant l'exécution de modèles, nous avons passé en revue des modèles historiquement voués à être exécutés : les réseaux de Petri, les *workflows* et les machines à états. Des outils hors IDM nous permettent de les exécuter mais ils sont créés de manière *ad hoc* sans profiter des avantages offerts par les techniques modernes de l'IDM avec son cadre dédié à la création de *DSML*. Avec l'IDM, il est aujourd'hui plus aisé de concevoir un *DSML* compilé ou un *i-DSML* pour des réseaux de Petri, des *workflows* ou encore des machines à états car les moteurs correspondants peuvent être en partie générés. Les *i-DSML* ont toutefois un net avantage face aux *DSML* compilés puisqu'ils peuvent être réutilisés en tant que tel en phase d'exécution et qu'ils sont tout à fait appropriés pour la simulation, la vérification et le prototypage. Ils permettent également de se passer complètement de la phase d'implémentation car le modèle est lui-même une application, ce qui apporte une certaine rapidité (ou agilité) pour concevoir un logiciel et a un impact significatif concernant la réduction des coûts de production du programme.

Au niveau de l'adaptation logicielle, nous avons mis en évidence un certain nombre de critères permettant de la caractériser. Elle peut être fonctionnelle ou non-fonctionnelle, prévue à l'avance ou non-

anticipée, prédictible ou non-déterministe, automatique (auto-adaptation) ou non, générique ou métier. Dans cette thèse, nous allons nous focaliser sur l'auto-adaptation (car nous souhaitons laisser le logiciel décider lui-même, dans l'esprit de l'informatique autonome), l'adaptation fonctionnelle (car changer la qualité de service des fonctionnalités existantes reste insuffisant), l'adaptation non-anticipée (car nous souhaitons bénéficier d'un programme intelligent qui sait réagir face à un contexte inconnu), l'adaptation prédictible (car nous souhaitons un système avec un raisonnement stable) et l'adaptation générique (car nous souhaitons pouvoir réutiliser nos adaptations). D'autre part, nous avons vu que l'adaptation logicielle repose toujours sur les concepts de contexte, de politiques d'adaptation et de boucle d'adaptation. Le contexte est soit endogène, lorsque les informations sont récupérées en interne dans le système, soit exogène, lorsque des capteurs sont nécessaires pour collecter des données provenant de l'extérieur des limites du système. Les politiques d'adaptation peuvent être écrites soit « en dur » dans du code au sein du moteur d'exécution du modèle, soit de façon externalisée dans un langage dédié (un *DSL*). Nous avons deux avantages à proposer cette externalisation des politiques d'adaptation : d'une part, les politiques d'adaptation peuvent être remplacées par d'autres sans avoir à intervenir sur le code de l'application adaptative (ce qui offre une certaine souplesse) et d'autre part, cela met en place une séparation des préoccupations exécution/adaptation. Dans nos travaux également, nous allons choisir un langage dédié à la représentation de ces politiques d'adaptation comme nous le verrons dans le chapitre 5. Bien qu'il existe de nombreuses boucles d'adaptation, elles ont toutes en commun une phase durant laquelle le contexte du logiciel adaptatif est vérifié et une autre durant laquelle le système est modifié en conséquence. Dans cette thèse, nous verrons que nous distinguons bien ces deux notions de vérification et d'action pour l'adaptation en considérant deux sortes d'opérations pour l'adaptation.

Concernant l'adaptation de l'exécution de modèles, nous avons vu qu'en dehors de l'IDM, les réseaux de Petri, les *workflows* et les machines à états, une fois augmentés, offrent déjà des capacités adaptatives. Toutefois, ces solutions sont encore une fois implémentées de manière *ad hoc*, ce qui est la source d'éventuelles erreurs et ce qui demande beaucoup de temps et d'efforts. Grâce au cadre offert par l'IDM, nous pouvons mettre en place une adaptation de l'exécution de modèles en réduisant considérablement les efforts et en se reposant sur une pile de méta-modélisation qui nous assure la propriété de conformité de nos modèles. En IDM, très peu de travaux ont envisagé et approfondi l'idée de l'adaptation appliquée à une exécution de modèles. Certaines approches proposent de mettre en place un modèle en cours d'exécution et d'appliquer des adaptations durant cette exécution, mais la boucle d'adaptation utilisée souffre soit d'un problème de complexité à maintenir un lien causal entre le système et le modèle, soit d'un problème de redondances d'informations entre les différents modèles en jeu. Parmi ces boucles, la seule qui n'est pas affectée par ces problèmes correspond à celle d'une adaptation directe de l'exécution d'un modèle (cf. figure 2.17) qui est la voie choisie pour mener ces travaux de thèse. Nous montrerons dans les prochains chapitres comment la mettre en œuvre.



## Chapitre 3

# Caractérisation des *i-DSML* adaptables

Dans le chapitre précédent, nous avons vu différentes approches permettant l'exécution de modèles. L'une d'elles consiste à interpréter des modèles grâce aux *i-DSML* et se démarque des autres puisqu'elle offre de nombreux avantages pour produire des logiciels. D'autre part, concernant l'adaptation de cette exécution de modèles, nous avons constaté que l'implémentation de la boucle d'adaptation la plus satisfaisante correspond au cas de la figure 2.17. En prenant l'exemple d'une machine à états simplifiée décrivant le contrôle de la motorisation d'un train, nous allons proposer dans ce chapitre une caractérisation conceptuelle des *i-DSML* adaptables. Un *i-DSML* adaptable est un *DSML* dont les modèles sont à la fois exécutables et adaptables.

Dans le cas d'utilisation de la machine à états du train, l'adaptation est réalisée par le système lui-même (c'est-à-dire sans intervention extérieure), ce qui signifie que c'est de l'auto-adaptation. D'autre part, nous verrons que cette adaptation permet au train de prendre en compte de nouveaux signaux lumineux qui étaient inconnus au départ, ce qui indique que nous sommes face à une adaptation fonctionnelle mais aussi non-anticipée. La façon dont est implémenté le comportement de notre machine à états du train implique que si nous la relançons dans un contexte identique elle réagira de la même manière, ce qui nous assure que notre adaptation est prédictible. En outre, l'adaptation que nous proposons dans ce chapitre est générique puisque nous verrons qu'elle s'appuie sur des propriétés qui sont indépendantes du contenu métier du modèle. En conclusion, cet exemple correspond à de l'auto-adaptation fonctionnelle, non-anticipée, prédictible et générique.

La section suivante rappelle ce qu'est l'exécution d'un modèle et en donne sa caractérisation conceptuelle. Cette caractérisation est ensuite étendue dans la section 3.2 pour décrire ce qu'un *i-DSML* adaptable contient et sur quoi il repose. Si l'exécution de modèles modifie seulement les éléments dynamiques du modèle, nous montrons que l'adaptation, quant à elle, peut modifier chaque partie du modèle (cela inclut les sémantiques d'exécution et d'adaptation elles-mêmes).

### 3.1 La caractérisation des *i-DSML*

Les modèles exécutables ne sont pas vraiment une nouvelle idée. En effet, comme expliqué dans la sous-section 2.2.2, cette notion a déjà fait l'objet de plusieurs travaux en IDM qui ont abouti aux mêmes conclusions :

- l'exécution d'un modèle n'a de sens que si le modèle a une nature exécutable (ce qui n'est pas le

cas de tout modèle)

- un moteur est responsable de l'exécution du modèle (il se charge de faire évoluer le modèle au cours du temps)
- le modèle embarque toutes les informations nécessaires à sa propre exécution (ce modèle est donc auto-contenu)

Avant de caractériser précisément ce qu'un *i-DSML* contient, nous allons donner des précisions sur ces trois critères.

### 3.1.1 La nature exécutable des modèles

Il existe une classification générale des modèles qui peut nous aider à identifier les modèles qui ont la capacité de s'exécuter et ceux qui ne l'ont pas : la notion de produit et de processus. En effet, les modèles (et leurs méta-modèles associés) peuvent décrire soit des produits, soit des processus (et ceci sans se préoccuper du système étudié). Par essence, seuls les modèles de processus offrent la possibilité de s'exécuter puisque leur contenu contient des concepts en relation directe avec le monde de l'exécution : point de départ, point d'arrivée, temps (passé, présent et futur), évolution du pas d'exécution, ...

Pour illustrer cette classification en reprenant des standards du développement logiciel, nous pouvons citer *SPEM* [63] comme étant un langage de modélisation de processus tandis que *CWM* [57], quant à lui, est un langage de modélisation de produits. Afin de citer un exemple de classification venant tout droit de l'*OMG*, *UML* lui-même donne trois catégories de diagrammes : les diagrammes structurels (classes, paquetages, ...), les diagrammes comportementaux (machines à états, activités, ...) et les diagrammes d'interactions (communications, séquences, ...). Intuitivement, seuls les diagrammes comportementaux et d'interactions peuvent être exécutés. Au-delà de ces exemples spécifiques, à chaque fois que nous concevons un *DSML*, il est important de garder à l'esprit sa potentielle nature exécutable.

### 3.1.2 Les moteurs d'exécution

Un *i-DSML* est plus qu'un simple méta-modèle (c'est-à-dire une syntaxe abstraite et un ensemble de règles forçant le modèle à être bien formé). La définition d'un langage contient aussi une syntaxe concrète et une sémantique d'exécution. La sémantique d'exécution d'un langage se trouve au niveau des règles de transformation dans le cas de *DSML* compilés ou dans le moteur d'exécution dans le cas de *DSML* interprétés. Un moteur d'exécution est dédié à un seul *i-DSML* (par exemple machines à états, *SPEM*, réseau de Petri, ...) et peut exécuter tout modèle conforme au méta-modèle de cet *i-DSML*.

Le rôle d'un moteur d'exécution est de « jouer » ou d'« animer » le modèle, permettant ainsi de faire évoluer son état, pas à pas. Les opérations d'exécution, implémentées par le moteur d'exécution et attachées à des éléments du méta-modèle, gèrent chaque pas d'exécution du modèle. L'état courant du modèle peut être conservé localement à l'intérieur du moteur ou embarqué dans le modèle lui-même. L'approche des *i-DSML* implique d'utiliser des modèles auto-contenus embarquant leur état courant mais la première solution peut s'avérer utile dans certains cas. Typiquement, cela arrive quand nous avons besoin d'exécuter un modèle conforme à un méta-modèle qui n'a pas été conçu de manière à gérer l'état courant du modèle, ce qui est le cas de tous les diagrammes dynamiques d'*UML*. En effet, le standard *UML* ne définit jamais un état courant pour ses diagrammes qui pourraient être exécutables. Dans ce cas, la solution est de stocker l'état courant du modèle à l'intérieur de la mémoire du moteur ou alors d'étendre le méta-modèle pour gérer l'état courant du modèle. En l'occurrence, Cariou *et al.* dans [11]

ont effectué cette extension pour les machines à états *UML*. L'inconvénient étant que le méta-modèle étendu diverge du standard *UML*.

### 3.1.3 Les modèles exécutables auto-contenus

Dans le cas d'un modèle auto-contenu, l'état courant du modèle à exécuter est stocké dans le modèle lui-même et donc chaque pas d'exécution change légèrement ce modèle. À première vue, cette stratégie semble polluer le méta-modèle avec beaucoup de détails qui ne se sont pas pertinents en phase de conception. De plus, cette approche peut sembler aller à l'encontre de l'abstraction offerte par les principes de la modélisation traditionnelle du fait qu'un modèle est l'abstraction d'un système (c'est-à-dire une vue simplifiée de celui-ci). Pourtant, il y a deux principales raisons justifiant l'utilisation de modèles auto-contenus.

D'abord, un modèle auto-contenu offre l'avantage qu'après chaque pas d'exécution, l'état courant du modèle peut être sérialisé dans un fichier de sortie<sup>15</sup>, ce qui fournit une traçabilité complète de l'exécution sous la forme d'une séquence de modèles. Certains travaux, comme [9], considèrent même que le modèle peut embarquer sa trace complète d'exécution en plus de son état courant. En partant de cette séquence de clichés, nous pouvons réaliser des opérations utiles comme des *rollbacks*, des vérifications au cours de l'exécution (telles que la vérification d'exécution en mode boîte noire dans [11]), du débogage, des tests, ...

La deuxième principale raison d'utiliser des modèles auto-contenus est en relation avec la nature exécutable de ces modèles. De tels modèles ont pour objectif de se substituer au code, qui se situe au plus bas niveau, très éloigné des modèles de conception plus abstraits. Par la force des choses, leur exécutabilité exige un plus grand niveau de détails et une complexité accrue.

### 3.1.4 La conception d'un *i-DSML*

Comme tout *DSML*, la conception d'un *i-DSML* se base sur la définition d'un méta-modèle, avec la particularité que les modèles conformes à ce méta-modèle seront exécutables par un moteur d'exécution. Nous pouvons identifier un patron récurrent [20] concernant les éléments constituant un *i-DSML*, ce qui nous amène à identifier trois parties :

1. les méta-éléments qui expriment l'organisation du contenu métier du modèle
2. les méta-éléments qui expriment l'état courant du modèle destiné à être exécuté
3. la sémantique d'exécution qui exprime comment le modèle évolue lorsqu'il est exécuté

La première partie est réalisée grâce à une méta-modélisation traditionnelle où l'ingénieur se concentre à définir la structure statique de son méta-modèle ainsi qu'un ensemble de règles forçant le modèle à être bien formé. Elle est appelée la « partie statique » et est composée d'éléments qui ne changent pas au cours de l'exécution. La deuxième partie introduit des méta-éléments structurels dont l'intention est de stocker l'état du modèle à un instant précis dans le temps. Cette partie peut également être contrainte par des règles. Elle est appelée la « partie dynamique » et est constituée d'éléments destinés à être modifiés durant l'exécution. La dernière partie, non la moindre, permet de définir comment le modèle évolue au cours du temps, modifiant seulement la partie dynamique (la partie statique ne change jamais). Elle est

<sup>15</sup>. C'est pourquoi certains auteurs peuvent considérer un processus d'exécution comme étant simplement une séquence de transformations de modèles endogènes, comme expliqué dans la sous-section 2.2.2.

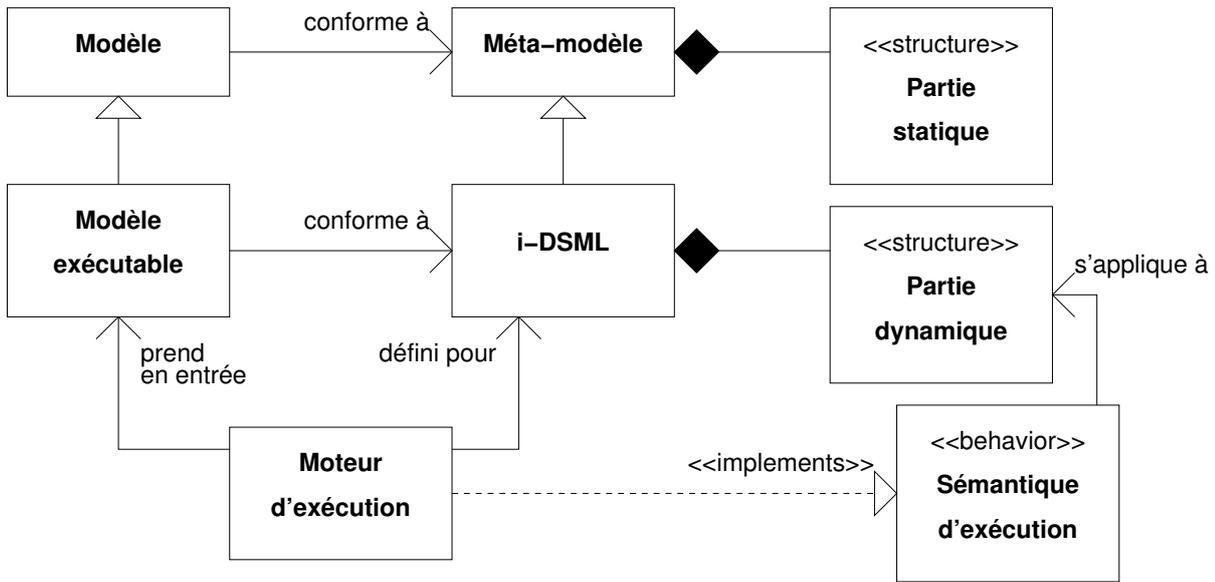


FIGURE 3.1 – La conception d'un *i-DSML*

appelée la « sémantique d'exécution » et celle-ci peut être définie de différentes façons. Une sémantique axiomatique permet de compléter la spécification du méta-modèle avec des règles de bonne évolution définissant les contraintes sous lesquelles le modèle peut évoluer [11]. Une sémantique translationnelle, quant à elle, peut être utilisée pour appliquer des techniques de simulation ou de vérification d'un autre espace technologique, comme dans [19]. Finalement, une sémantique opérationnelle consiste à mettre sous la forme d'opérations le comportement d'exécution à l'aide d'un langage d'actions. Les opérations correspondent alors à des actions implémentées par un moteur d'exécution. Comme illustré dans la figure 3.1, la partie dynamique et la sémantique d'exécution sont spécifiques à l'*i-DSML*.

### 3.1.5 Les machines à états exécutables

Les machines à états *UML* sont typiquement l'un des meilleurs exemples de modèles exécutables, en plus d'être bien connus. Dans ce chapitre, nous décidons donc de nous en servir comme fil rouge, mais pour des raisons de simplification, nous définissons un méta-modèle de machines à états réduit, avec un nombre limité de fonctionnalités : des états composites avec un état historique et des transitions associées à un événement. Il s'agit donc d'un langage qui ressemble aux machines à états *UML* mais qui est taillé sur mesure : c'est donc bien un *DSML*. De plus, les machines à états *UML* telles qu'elles sont définies par l'*OMG* n'incluent pas la partie dynamique nécessaire pour une solution d'exécution 100% modèle. Dans ce qui suit, il ne faudra pas confondre le mot « état » faisant référence aux états contenus dans une machine à états (ou *statechart*) et celui faisant référence à l'état courant de l'exécution dudit modèle.

Notre méta-modèle de machines à états est représenté sur la figure 3.2 (les éléments `AdaptableElement` et `[Integer]Property` sont dédiés à la gestion de l'adaptation et seront introduits dans la section suivante). Sa partie statique permet de définir l'ensemble des états et des transitions d'une machine à états. Elle contient classiquement les éléments suivants :

- un état qui a un nom et qui est contenu dans un état composite
- deux types de pseudo-états qui peuvent être définis : un état initial et un état historique référen-

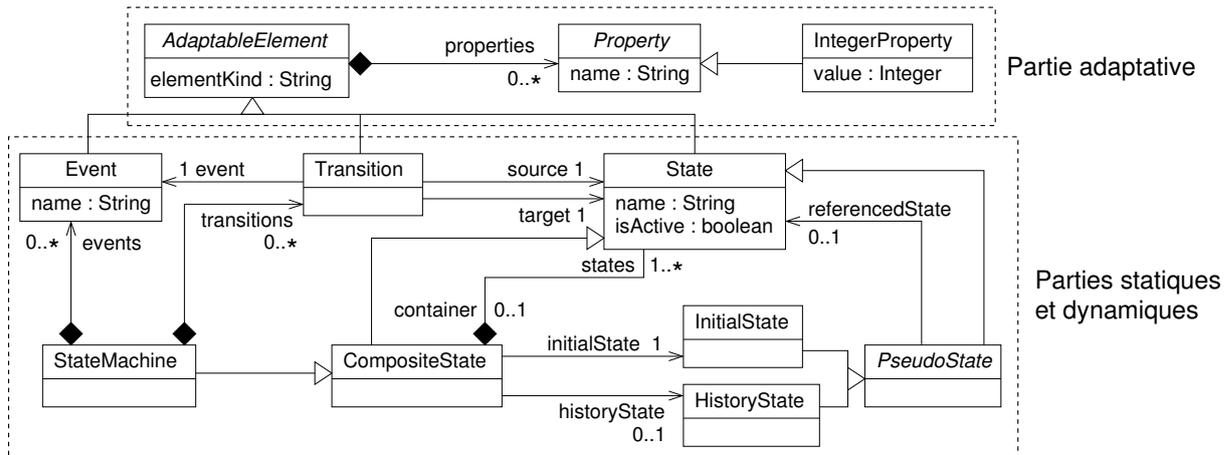


FIGURE 3.2 – Le méta-modèle de machines à états exécutables et adaptables

çant chacun un état de l'état composite auquel ils appartiennent (tout état composite contient obligatoirement un état initial et facultativement un état historique)

- une transition entre un état source et un état cible, associée à un événement représenté par un nom
- une machine à états qui est un type particulier d'état composite (représentant la hiérarchie des états de la machine) possédant un ensemble d'événements et de transitions

La partie dynamique du méta-modèle, qui permet de représenter l'état courant d'une machine à états en cours d'exécution, est composée des deux éléments suivants :

- l'attribut booléen `isActive` d'un état qui précise si l'état est actif ou non
- l'état référencé par un état historique (cf. la référence `referencedState`) qui désigne le dernier état qui est (ou était) actif pour l'état composite dans lequel il est contenu

Les parties statiques et dynamiques sont complétées avec des invariants *OCIL* définissant les règles pour que le modèle soit bien formé afin de spécifier plus précisément le méta-modèle. Pour la partie statique, il est nécessaire par exemple qu'un pseudo-état référence l'un des états de son état composite (cf. invariant aux lignes 9 et 10 du listing 3.1). Pour la partie dynamique, le principal invariant spécifie que soit tous les états sont inactifs (c'est-à-dire le modèle n'est pas en cours d'exécution), soit il y a dans la machine à états un et seulement un état actif et si cet état est un état composite, alors il contient un et seulement un état actif et ainsi de suite (cf. invariant aux lignes 19 et 20 du listing 3.1). Le listing 3.1 donne l'ensemble de ces contraintes.

```

1 context AdaptableElement inv uniquePropertyName:
2   self.properties->forall(p1 : Property, p2 : Property | p1 <> p2 implies p1.name <> p2.
   name)
3 context CompositeState::activeSubTree(): Boolean
4 body: self.states->select(s : State | (not s.oclIsKindOf(PseudoState) and s.isActive)
   )->size() = 1 and self.states->select(s : State | s.oclIsTypeOf(CompositeState))->
   forall(s : State | if s.isActive then s.oclAsType(CompositeState).activeSubTree()
   else s.oclAsType(CompositeState).unactiveSubTree() endif)
5 context CompositeState::unactiveSubTree(): Boolean
    
```

```

6  body: self.states->select(s : State | not s.ocIsKindOf(PseudoState))->forall(s :
    State | not s.isActive) and self.states->select(s : State | s.ocIsTypeOf(
    CompositeState))->forall(s : State | s.ocAsType(CompositeState).unactiveSubTree()
    )
7  context CompositeState inv pseudoStatesCount:
8  self.states->select(s : State | s.ocIsTypeOf(InitialState))->size() = 1 and self.
    states->select(s : State | s.ocIsTypeOf(HistoryState))->size() <= 1;
9  context CompositeState inv pseudoStatesInComposite:
10 self.states->includes(self.initialState.referencedState) and self.states->includes(self.
    initialState) and if self.historyState.ocIsUndefined() then true else self.states->
    includes(self.historyState) and if self.historyState.referencedState.ocIsUndefined
    () then true else self.states->includes(self.historyState.referencedState) endif
    endif
11 context CompositeState inv uniqueCompositeName:
12 self.states->forall(s1 : State | self.states->forall(s2 : State | s1.name = s2.name
    implies s1 = s2))
13 context Event inv uniqueEventName:
14 Event.allInstances()->select(e : Event | e.ocIsTypeOf(Event))->forall(e : Event | e.
    name = self.name implies e = self)
15 context InitialState inv initialStateNeverEmpty:
16 not self.referencedState.ocIsUndefined()
17 context State inv containerForAllStates:
18 not self.ocIsTypeOf(StateMachine) implies not self.container.ocIsUndefined()
19 context StateMachine inv activeStateHierarchyConsistency:
20 if self.isActive then self.activeSubTree() else self.unactiveSubTree() endif
21 context StateMachine inv noContainerForStatemachine:
22 self.container.ocIsUndefined
23 context StateMachine inv singleStateMachine:
24 StateMachine.allInstances()->size() = 1
25 context Transition inv transEventSource:
26 Transition.allInstances()->forall(t : Transition | t.source = self.source and t.event
    = self.event implies self = t)
27 context Transition inv transHistory:
28 not self.source.ocIsTypeOf(HistoryState)
29 context Transition inv transInitialState:
30 not self.source.ocIsTypeOf(InitialState) and not self.target.ocIsTypeOf(InitialState
    ) and not self.source.ocIsTypeOf(StateMachine) and not self.target.ocIsTypeOf(
    StateMachine)

```

Listing 3.1 – Les invariants *OC*L pour la machine à états

### 3.1.6 Un exemple de train

L'exemple utilisé dans ce chapitre est librement inspiré d'un système ferroviaire<sup>16</sup>. Le comportement de la puissance moteur du train est spécifié au travers d'une machine à états. Le train est soit à l'arrêt, soit en marche à une vitesse donnée. Cette vitesse dépend de la couleur des signaux lumineux croisés le long de la voie de chemin de fer. Le contexte du train correspond aux signaux qui contrôlent sa vitesse. Concrètement, les différentes vitesses du train sont spécifiées au travers des états de la machine à états alors que les signaux correspondent aux événements associés aux transitions entre ces états. Avec la même machine à états, nous pouvons spécifier le comportement du système (le train) et ses interactions avec le contexte (les signaux lumineux).

#### 3.1.6.1 Le contexte du train

Le train circule sur les voies ferrées et peut rencontrer des signaux à trois ou quatre couleurs différentes. Il y a deux sortes de voies : celles à vitesse normale (jusqu'à 130 km/h) et celles à haute vitesse (jusqu'à 300 km/h). Le signal à trois couleurs, correspondant au signal (a) de la figure 3.3, est destiné aux voies à vitesse normale tandis que le signal à quatre couleurs, correspondant au signal (b) de la figure 3.3, est prévu pour celles à haute vitesse. La signification de ces couleurs sont les suivantes (seulement une lumière est émise au même instant) : le rouge indique que le train doit s'arrêter immédiatement, l'orange qu'il ne doit pas circuler à plus de 40 km/h, le vert qu'il peut circuler à vitesse normale (mais pas au-delà) et le violet qu'il peut circuler à haute vitesse.

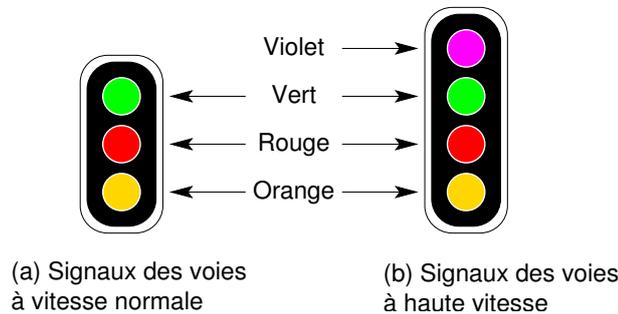


FIGURE 3.3 – Les signaux pour le train

#### 3.1.6.2 Un modèle simple de train

La figure 3.4 représente une machine à états définissant le comportement d'un train destiné à circuler à vitesse normale et illustre trois pas d'exécution. Concrètement, ce train n'est pas capable de circuler à une vitesse supérieure à 130 km/h mais est autorisé à circuler à cette vitesse sur des voies à haute vitesse. Les états définissent la vitesse du train. Pour simplifier, le nom d'un état correspond à la vitesse de croisière du train (en km/h) associée à cet état. Les états 0 et 40 font donc référence respectivement aux vitesses de 0 km/h et 40 km/h et correspondent à l'état d'arrêt et à l'état de vitesse lente. Quand le train circule à vitesse normale, le conducteur de train peut choisir entre deux vitesses : 100 km/h ou 130 km/h. Ces deux états ont été placés dans un état composite qui correspond à l'état de vitesse normale.

<sup>16</sup>. Les machines à états représentant le comportement du train et leurs signaux associés sont une simplification de la réalité (il ne faut donc pas considérer cet exemple comme le cas d'étude d'un système réel).

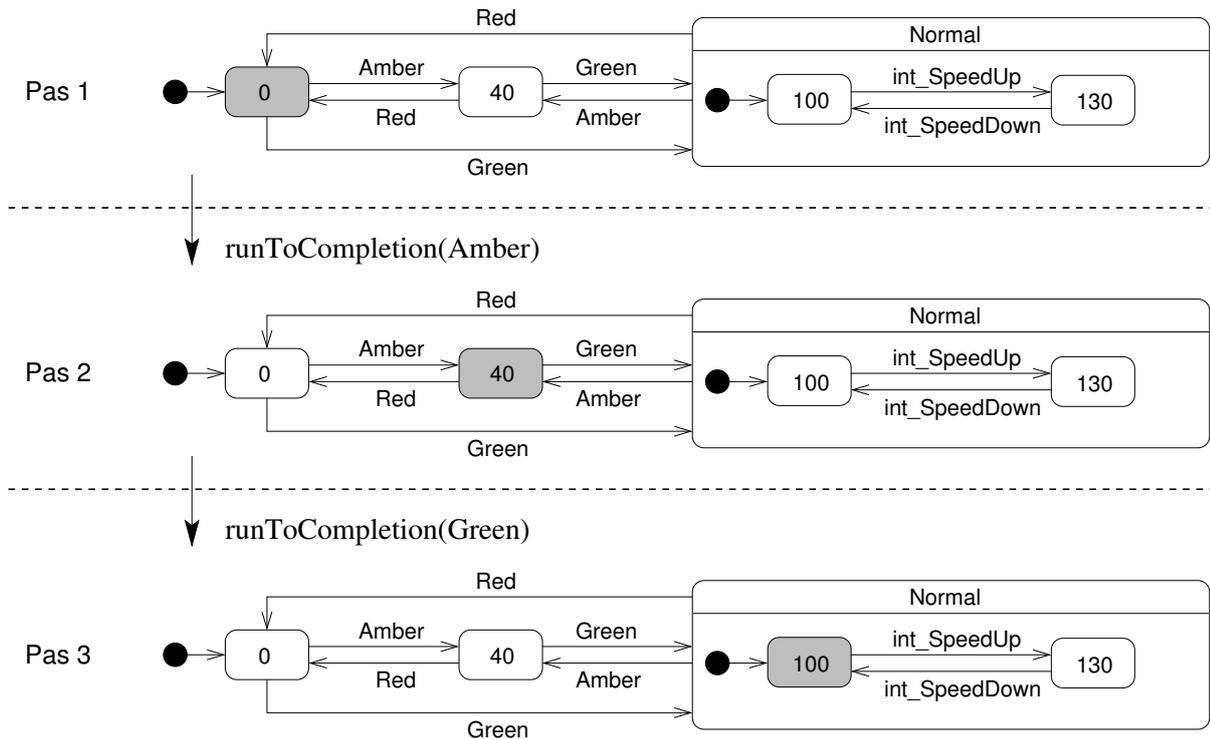


FIGURE 3.4 – L'exécution de la machine à états d'un train

Les transitions entre les états sont associées à la couleur du signal lumineux : **Red** pour rouge, **Green** pour vert et **Amber** pour orange. La couleur violette n'est pas gérée ici puisque le train ne peut pas circuler à plus de 130 km/h (mais il peut toutefois circuler à 100 km/h ou 130 km/h seulement sur des voies à haute vitesse). Il y a deux événements particuliers : `int_SpeedUp` et `int_SpeedDown`. Ces événements sont des actions internes du train (c'est-à-dire elles correspondent à des actions directement effectuées par le conducteur de train sur son tableau de bord). Pour ne pas confondre ces événements internes avec ceux externes venant du contexte, leur nom est préfixé par `int_`.

### 3.1.6.3 L'exécution de la machine à états du train

La figure 3.4 montre trois pas d'exécution de la machine à états réalisés par l'opération `runToCompletion`, qui prend un nom d'événement en tant que paramètre. Cette opération traite l'occurrence d'un événement. Le premier pas consiste à activer l'état initial de la machine à états (un état actif est représenté graphiquement sur fond gris). Ensuite, pour le deuxième pas, l'événement **Amber** provient du contexte, ce qui implique un changement de l'état actif qui est maintenant l'état 40. Finalement, le troisième pas d'exécution correspond au résultat de l'apparition de l'événement **Green** dans le contexte, ce qui nous amène à activer l'état composite **Normal** et par conséquent son état initial (c'est-à-dire l'état 100).

Exécuter une machine à états consiste donc à seulement alterner la valeur de l'attribut booléen `isActive` des états. Pour les états composites, il faut penser également à changer l'état référencé par son éventuel état historique. En résumé, seuls les éléments du modèle appartenant à la partie dynamique sont modifiés durant l'exécution.

## 3.2 La caractérisation des *i-DSML* adaptables

Nous considérons qu'un *i-DSML* adaptable est l'extension logique d'un *i-DSML*. En effet, les modèles adaptables sont des modèles exécutables dotés de capacités adaptatives.

### 3.2.1 La conception d'un *i-DSML* adaptable

La figure 3.5 décrit la conception d'un *i-DSML* adaptable. Comme un *i-DSML* adaptable est une extension d'un *i-DSML*, cette figure étend la figure 3.1. Les éléments ajoutés sont :

1. les méta-éléments qui expriment des propriétés sur le modèle et qui aideront à son adaptation
2. la sémantique d'adaptation qui donne vie à ces propriétés et qui exprime des problèmes d'adaptation et leurs solutions

Nous avons donc identifié deux parties distinctes. La première est appelée la « partie adaptative » et fait références aux éléments structurels ou aux règles pour forcer le modèle à être bien formé, qui sont ajoutés au méta-modèle afin de faciliter les adaptations ultérieures. La deuxième partie est appelée la « sémantique d'adaptation » et correspond à une spécialisation de la sémantique d'exécution. En effet, la sémantique d'exécution décrit un comportement nominal tandis que la sémantique d'adaptation exprime un comportement lors de cas extraordinaires ou situations anormales nécessitant une adaptation. Tout comme la sémantique d'exécution, la sémantique d'adaptation peut être définie de différentes façons. Elle peut prendre la forme d'une sémantique axiomatique afin de compléter la spécification du méta-modèle [14] ou prendre la forme d'une sémantique opérationnelle. Par conséquent, un moteur d'adaptation implémentant une sémantique d'adaptation est une extension d'un moteur d'exécution (c'est-à-dire il traite à la fois la sémantique opérationnelle relative à l'exécution et celle relative à l'adaptation).

Sans entrer dans les détails de la gestion ou du traitement d'une sémantique d'adaptation par un moteur, nous pouvons d'ores et déjà dire qu'elle sera composée d'un ensemble de politiques d'adaptation ; chacune de ces politiques combinant des opérations d'adaptation. Certaines opérations sont dédiées à la vérification de la cohérence du modèle face à son contexte tandis que d'autres sont dédiées à la réalisation concrète d'actions d'adaptation. Les politiques d'adaptation peuvent être exprimées sous la forme `if <check> then <action>` ou n'importe qu'elle autre combinaison plus complexe ou récursive de la sorte. Dans le chapitre 5, nous définirons un *i-DSML* permettant de spécifier et exécuter ces politiques.

Un point important à souligner est que la sémantique d'adaptation s'applique sur tous les constituants de l'*i-DSML* adaptable (c'est-à-dire les parties structurelles statiques, dynamiques et adaptatives mais aussi les parties comportementales d'exécution et d'adaptation sont concernées). Concrètement, à l'exécution, le contenu entier du modèle peut être changé incluant même la sémantique exécutée. Ceci apporte la propriété de réflexivité aux *i-DSML* adaptables puisque cela permet l'adaptation de l'adaptation (c'est-à-dire la méta-adaptation).

#### 3.2.1.1 Portée des sémantiques d'exécution et d'adaptation

Nous avons identifié deux catégories de modifications pendant l'exécution d'un modèle : les *Create/Update/Delete* (*CUD*) et les substitutions. Les *CUD* ciblent les instances de n'importe quelle partie structurelle (statique, dynamique ou adaptative). Les substitutions, quant à elles, ciblent les parties comportementales (sémantique d'exécution et d'adaptation). Nous considérerons que les substitutions

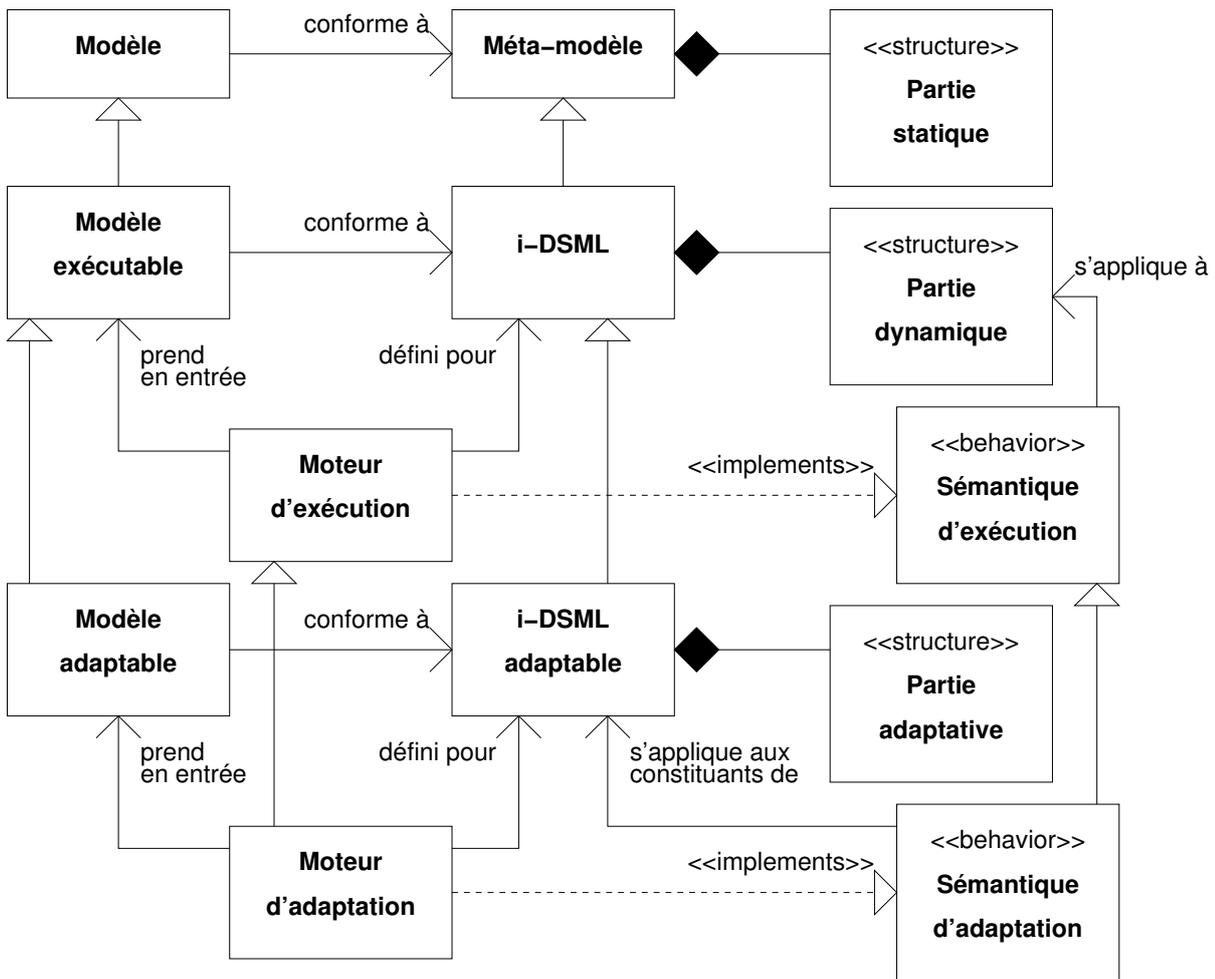


FIGURE 3.5 – La conception d'un *i-DSML* adaptable

s'appliquent seulement aux parties comportementales parce qu'il n'est pas possible de définir une nouvelle sémantique à partir de rien (concrètement cela reviendrait à écrire des lignes de code dans le moteur). Au lieu de cela, il est plus simple d'avoir un ensemble de sémantiques déjà implémentées et de choisir, à l'exécution, laquelle sera utilisée. Les *CUD* et substitutions peuvent être appliquées sur différentes parties afin d'atteindre les objectifs suivants :

- les *CUD* sur la partie dynamique : l'état courant d'exécution est altéré (par exemple forcer à aller en arrière ou en avant, redémarrer ou activer un état donné pour une machine à états)
- les *CUD* sur la partie statique : la structure du modèle lui-même est changée (par exemple modifier le contenu métier modélisé, comme ajouter un état ou changer l'état initial d'un état composite pour une machine à états)
- les *CUD* sur la partie adaptative : un élément relatif à l'adaptation est changé (par exemple la valeur d'une propriété de qualité de service est modifiée pour correspondre à un contexte changeant)
- la substitution sur la sémantique d'exécution : elle passe d'une interprétation de modèle donnée à une autre (par exemple gérer les différentes exécutions, comme celle d'Harel ou celle d'*UML*, qui diffèrent sur la sémantique d'exécution pour les transitions des machines à états)

Partie de l' <i>i-DSML</i> adaptable	Sémantique d'exécution	Sémantique d'adaptation
«Structure»	Partie statique	N/A
	Partie dynamique	<i>Create/Update/Delete</i>
	Partie adaptative	<i>Create/Update/Delete</i>
«Behavior»	Sémantique d'exécution	Substitution
	Sémantique d'adaptation	Substitution

TABLE 3.1 – Portée des sémantiques d'exécution et d'adaptation

- la substitution sur la sémantique d'adaptation : elle passe d'un ensemble d'opérations à un autre à l'intérieur des politiques d'adaptation (par exemple vérifier la cohérence d'un modèle avec son contexte dans un mode exact ou relâché, comme nous le verrons dans notre exemple)

Le tableau 3.1 donne un résumé de ces modifications envisageables lors de l'exécution d'un modèle en considérant d'une part la sémantique d'exécution et d'autre part la sémantique d'adaptation. Nous pouvons remarquer que, dans le cas de la sémantique d'exécution, seule la partie dynamique du modèle est modifiée alors qu'avec la sémantique d'adaptation, toutes les parties structurelles et la sémantique peuvent être impactées. Ainsi, nous pouvons en déduire que l'adaptation a une portée plus large que l'exécution.

### 3.2.2 La partie adaptative pour le méta-modèle de machines à états

Le méta-modèle de machines à états sur la figure 3.2 inclut les éléments dédiés à la gestion de l'adaptation. Le premier élément est l'attribut `elementKind` disponible pour les méta-éléments `Event`, `Transition` et `State` grâce à la spécialisation d'`AdaptableElement`. Cet attribut permet la définition de « types » d'événement, de transition et d'état d'une machine à états. Un type a pour but de préciser qu'un élément joue un rôle particulier. Conceptuellement, un type est équivalent à un stéréotype des profils *UML*. De plus, à travers l'association `properties`, les événements, les transitions et les états peuvent être associés à des propriétés. Une propriété est simplement composée d'un nom et d'une valeur. Pour simplifier, seules les propriétés entières sont considérées ici (mais bien sûr, des propriétés de n'importe quel type pourraient être ajoutées au méta-modèle). Ces propriétés peuvent être liées à des paramètres et valeurs de qualité de service. Conceptuellement, une propriété est équivalente à une valeur marquée des profils *UML*.

Dans ce qui suit, nous allons montrer que les types et les propriétés peuvent être utilisés pour gérer l'adaptation de l'exécution d'une machine à états grâce aux informations supplémentaires qu'ils apportent. Les types peuvent être utilisés pour définir un mode relâché concernant la vérification de la cohérence du modèle face à son contexte. Les propriétés associées aux événements du contexte, quant à elles, permettent la modification de la machine à états exécutée pour gérer des cas d'événements inattendus.

### 3.2.3 Des machines à états spécialisées

Pour définir des actions d'adaptation génériques, comme nous verrons dans le chapitre suivant, il est généralement utile voire nécessaire de spécialiser un *i-DSML* en définissant des sous-types de modèles. Par générique, nous entendons être totalement indépendant du contenu métier du modèle. Concrètement ici, pour nos machines à états, l'adaptation doit pouvoir se faire de manière automatique sur un modèle sans avoir besoin de savoir s'il représente le comportement d'un train, d'un ascenseur ou d'un four à micro-ondes.

Nous ajoutons donc ici une spécialisation complémentaire sur nos machines à états. Il s'agit d'intégrer le fait qu'un événement donné conduit toujours à un même état quelque soit le départ de la transition associée. La modélisation de notre train respectera cette contrainte. Par exemple, indépendamment de l'état source, l'événement `Amber` amène toujours à l'état 40. En effet, un signal d'une certaine couleur oblige le train à toujours rouler à une vitesse donnée, quelle que soit la vitesse courante du train. Par exemple, un feu rouge oblige systématiquement le train à s'arrêter (aucun autre comportement n'est acceptable). Comme nous le verrons par la suite, sans cette spécialisation, il ne sera pas possible de savoir comment automatiquement modifier le modèle pour l'adapter à un événement inconnu. Techniquement, cette spécialisation est définie en rajoutant des invariants *OCL* sur le méta-modèle.

### 3.2.4 L'adaptation à l'exécution de la machine à états du train

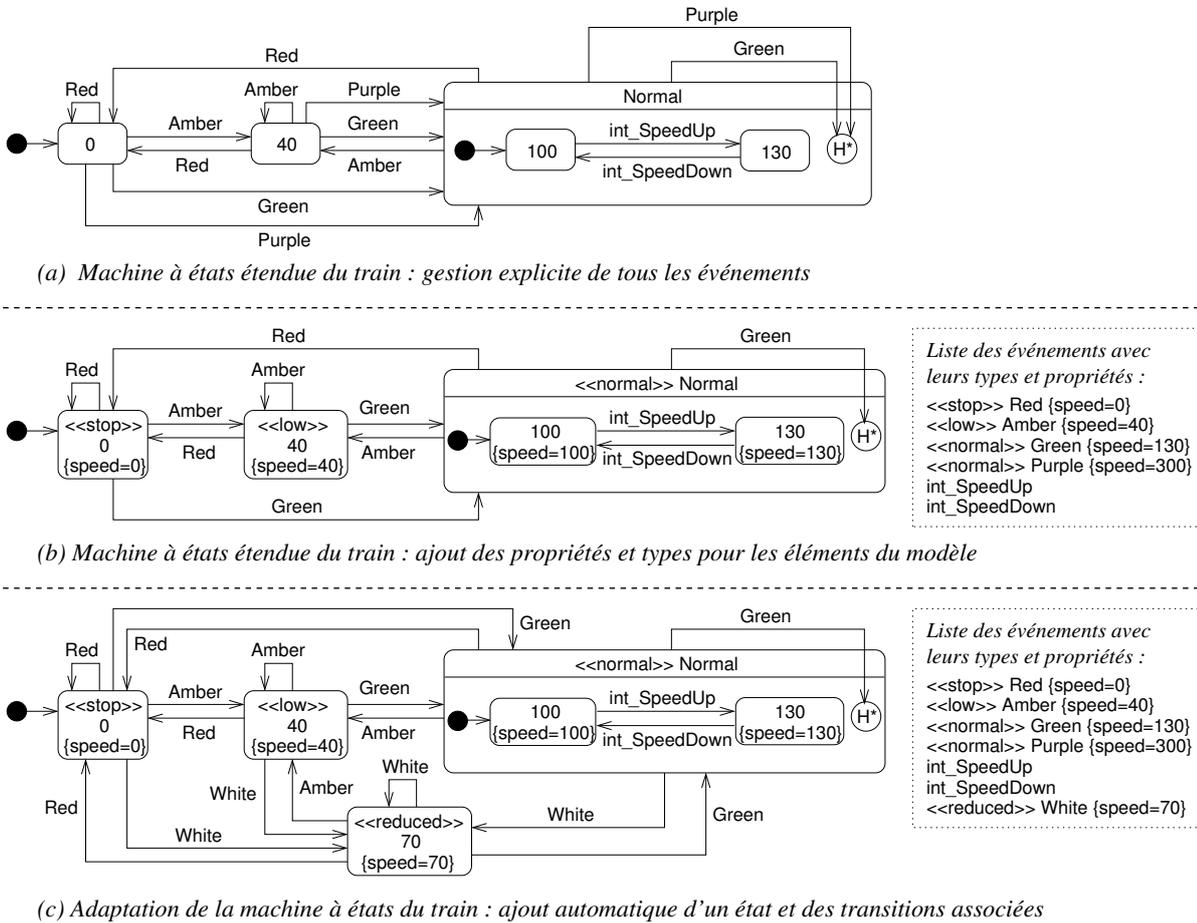


FIGURE 3.6 – L'adaptation à l'exécution de la machine à états du train

La principale vérification d'adaptation consiste à d'abord tester si le comportement du système est cohérent avec son contexte. Ici, concrètement cela revient à vérifier si tout signal est compris et correctement traité par la machine à états. Ensuite, quand le système n'est pas cohérent avec son contexte, la politique d'adaptation va réaliser une action d'adaptation telle que passer dans un mode de vérification relâché ou bien modifier la structure du modèle.

Des contraintes spécifiques de conception peuvent être appliquées pour vérifier la cohérence d'un modèle face à son contexte. Le problème est qu'il est nécessaire d'être capable de distinguer une interaction inattendue – nécessitant de prendre une décision d'adaptation – d'une interaction attendue et déjà gérée. Pour les machines à états, il est nécessaire de pouvoir déterminer si un événement est attendu ou non. Plusieurs solutions sont possibles, comme paramétrer le moteur d'exécution avec la liste des événements attendus et vérifier pour chaque occurrence d'événement si l'événement correspondant est dans cette liste. Une autre solution appliquée ici est auto-contenue dans le modèle. Nous ajoutons explicitement une transition partant de chaque état pour chaque événement attendu afin de créer un automate « complet ». La partie (a) de la figure 3.6 montre cette modification de la machine à états : pour toutes les couleurs d'événement attendues (rouge, orange, vert et violet), il y a une transition partant de chaque état. Quand une couleur d'événement nous amène de nouveau dans le même état, c'est une auto-transition (en cas d'état composite, nous arrivons sur l'état historique). Dès lors, sauf pour les événements internes commençants par `int_`, chaque événement connu (correspondant à une couleur de signal croisé par le train) déclenche nécessairement une transition. Cela façon de faire offre la certitude que s'il n'y a pas de transition associée à un événement, en considérant les états actifs courants, alors cet événement est inattendu et une action d'adaptation doit être réalisée.

En partant sur cette nouvelle définition de la machine à états du train, pour illustrer nos propos, nous allons décrire sept opérations d'adaptation (deux vérifications d'adaptation et cinq actions d'adaptation dont une modification de la sémantique d'adaptation) et deux différentes sémantiques d'exécution pour le méta-modèle de la machine à états. Notons que toutes les opérations d'adaptation sont ici génériques et que par conséquent la sémantique d'adaptation l'est également.

#### 3.2.4.1 Les vérifications d'adaptation de la machine à états du train

L'objectif de la vérification consiste à déterminer si un événement est attendu ou non. Cette vérification peut être effectuée en mode exact ou relâché à l'aide des types d'événements, ce qui donne lieu à deux vérifications d'adaptation distinctes. Considérons l'occurrence d'un événement correspondant au signal violet. Le train n'est pas capable de circuler à haute vitesse, donc quand il croise ce signal violet, le train circulera à vitesse normale. C'est pour cette raison que la machine à états du train dans la partie (a) de la figure 3.6 amène à l'état `Normal` pour chaque occurrence de l'événement `Purple`.

Le vérification exacte consiste à s'assurer qu'il existe une transition associée à un événement dont le nom est identique tandis que la vérification relâchée revient à s'assurer qu'il existe une transition associée à un type d'événement identique et pas forcément à un événement de même nom. La partie (b) de la figure 3.6 montre une variation de la machine à états où les types et les propriétés ont été ajoutés aux états et événements. Il y a trois types d'états (qui sont représentés entre guillemets « ... », comme pour la notation des stéréotypes dans *UML*) : l'état 0 est de type arrêt, l'état 40 est de type lent et l'état composite `Normal` est de type normal. Les événements, en fonction de la cible de la transition associée, sont aussi marqués avec des types : `Red` est de type arrêt, `Amber` est de type lent, `Green` et `Purple` sont de type normal. Nous pouvons remarquer que les transitions associées à la couleur violette ont disparu. En effet, dans le mode relâché, l'événement `Purple` sera traité par défaut comme l'événement `Green` parce qu'ils sont du même type : `normal`. La couleur violette est donc considérée par le train comme une couleur verte même si elles n'ont pas le même rôle ni la même signification.

### 3.2.4.2 Les actions d'adaptation de la machine à états du train

Le mode de vérification peut être changé au cours de l'exécution du modèle : une opération de vérification d'adaptation peut être substituée par une autre (changeant de ce fait la sémantique d'adaptation). Si le modèle dans la partie (b) de la figure 3.6 est exécuté et si le train circule sur des voies à vitesse normale alors le mode de vérification peut être le mode exact car les seuls signaux que le train peut croiser (rouge, orange ou vert) sont directement et exactement gérés pour chaque état de la machine à états. Cependant, si le train circule cette fois sur des voies à haute vitesse, il peut croiser un signal violet qui n'est pas directement géré en mode exact. Une action d'adaptation peut être de passer en mode vérification relâché et de révérifier la validité du signal violet. Dans ce mode, comme expliqué précédemment, ce signal sera considéré comme un signal vert et sera traité.

En plus de substituer une vérification d'adaptation par une autre, d'autres actions d'adaptation peuvent être utilisées dans le cas d'événements inattendus. Considérons le cas d'un changement dans le contexte. Une action élémentaire est d'arrêter l'exécution de la machine à états s'il n'y a pas moyen de décider comment gérer l'événement. Une action d'adaptation plus pertinente est de charger une nouvelle machine à états (une machine à états de référence, telle que définie dans [12]) qui est capable de gérer le nouveau contexte, si une telle machine à états est disponible bien sûr.

Si l'événement inattendu est associé à des propriétés, elles peuvent être exploitées pour déterminer si cet événement peut cibler un état existant de la machine à états. S'il se trouve que la réponse est négative, alors nous ajoutons à la machine à états l'état et les transitions associées à l'événement. La partie (b) de la figure 3.6 définit une propriété de vitesse pour chaque événement et état. Les propriétés sont représentées d'une façon similaire aux valeurs marquées des profils *UML* (par exemple `{speed=40}`). Par exemple, l'événement `Amber` et l'état `40` sont tous les deux associés à une propriété de vitesse de valeur 40 qui signifie 40 km/h. Supposons que la machine à états du train est exécutée et qu'un signal blanc d'un type réduit et d'une propriété de vitesse de 70 km/h est croisé. Dans les deux modes de vérification (exact et relâché), l'événement `White` demeure inattendu. Comme aucun état n'a de propriété de vitesse de 70 km/h, un nouvel état appelé `70` est créé avec le même type et la même propriété que l'événement `White`. Toutes les transitions requises partant de ce nouvel état ou arrivant à ce nouvel état sont également ajoutées, pour continuer à respecter ce qui a été énoncé précédemment : tout état existant doit être la source d'une transition associée à l'événement `White` qui amène à ce nouvel état et pour chaque couleur d'événement (`Red`, `Amber` et `Green`), il doit exister une transition partant de ce nouvel état et qui amène à l'état de cette couleur d'événement. La partie (c) de la figure 3.6 montre le résultat de cette modification en cours d'exécution pour gérer le signal blanc. Un point important à noter concernant la modification du modèle est qu'elle se base sur la comparaison de propriétés sans nécessiter de savoir ce qu'elles sont et ce qu'elles représentent. Ainsi, le moteur d'adaptation est générique et compare simplement deux ensembles de propriétés grâce à leur nom et leur valeur.

Une dernière action d'adaptation pourrait être de forcer l'activation d'un état de type arrêt (l'état `0` pour la machine à états du train) en cas d'événement inattendu. Ceci correspond à une adaptation métier puisqu'elle suppose que le modèle en cours d'exécution contient un tel état. Cette action est différente de celle qui consiste à arrêter l'exécution de la machine à états que nous avons vu précédemment car ici, la machine à états reste exécutée. En gros, dans un cas c'est tout le système informatique qui pilote le moteur qui s'arrête (et par voie de conséquence le moteur aussi), alors que dans l'autre l'ordre est donné au moteur d'atteindre l'état `0`. Dans les deux cas, le résultat est le même : le train s'arrête s'il ne

comprend pas le signal croisé.

### 3.2.4.3 Les différentes sémantiques d'exécution de la machine à états du train

Pour les machines à états, un conflit de transition apparaît quand une transition part d'un état composite et qu'une autre transition associée au même événement part d'un état interne à cet état composite. La façon de choisir laquelle des transitions déclencher, quand un événement survient, est un point qui différencie les sémantiques d'exécution existantes. Selon la sémantique d'*UML*, c'est la transition partant de l'état le plus profond dans la hiérarchie des états qui sera déclenchée alors qu'avec la sémantique d'Harel [37] c'est tout le contraire puisqu'il faut choisir la transition la moins profonde partant de l'état composite.

Au moment de charger le modèle d'une machine à états ou pendant son exécution, lorsque nous changeons l'état courant de la machine à états par un autre, il est nécessaire de savoir pour quelle sémantique d'exécution elle a été conçue. Nous pouvons imaginer une propriété associée à la machine à états qui précise quel type de sémantique d'exécution utiliser pour franchir les transitions. Le moteur d'exécution embarque alors le code correspondant à chaque sémantique opérationnelle. Ainsi, nous nous retrouvons à sélectionner la bonne sémantique d'exécution en fonction de la valeur de cette propriété. En d'autres termes, en modifiant le modèle via une action d'adaptation, la sémantique opérationnelle peut être substituée par une autre si cela s'avère nécessaire.

## 3.3 Conclusion

Dans ce chapitre, nous proposons une caractérisation conceptuelle de l'adaptation directe de l'exécution d'un modèle au travers du concept d'*i-DSML* adaptable. Bien que l'exécution et l'adaptation d'un modèle soient très liés (car une sémantique d'adaptation est une forme spéciale de sémantique d'exécution), il existe deux différences. La première différence concerne, d'un point de vue général, l'intention de la sémantique. En effet, une sémantique d'exécution décrit un comportement nominal alors qu'une sémantique d'adaptation exprime un comportement lors de cas extraordinaires ou situations anormales nécessitant une adaptation. La deuxième différence concerne, d'un point de vue technique, les parties modifiées du modèle. En effet, l'exécution de modèles modifie seulement les éléments dynamiques du modèle alors que la sémantique d'adaptation peut modifier chaque partie du modèle (cela inclut les sémantiques d'exécution et d'adaptation elles-mêmes).

Nous avons constaté dans notre exemple de machines à états que l'*i-DSML* de base n'était pas directement adaptable. Nous avons alors été amenés à contraindre la machine à états, comme expliqué dans la sous-section 3.2.3 et donc à spécialiser notre méta-modèle d'origine. Cette spécialisation apporte avec elle des informations supplémentaires sur lesquelles nous pouvons nous appuyer pour permettre l'adaptation. Ainsi, il devient possible d'écrire des opérations de vérification et d'action qui vont constituer la sémantique d'adaptation de notre *i-DSML* adaptable. De manière générale nous nous sommes rendu compte que ces spécialisations sont souvent indispensables pour définir des politiques d'adaptation, surtout quand elles sont génériques. Dans le chapitre suivant, nous allons définir le concept de famille qui est un cadre permettant d'organiser ces spécialisations.



## Chapitre 4

# Gestion de l'adaptation des *i-DSML* par les familles

Nous avons vu dans le chapitre précédent que les *i-DSML* adaptables sont des méta-modèles auto-contenus qui sont interprétés par un moteur permettant leur exécution. Leur adaptation est rendue possible grâce à la partie structurelle d'adaptation du méta-modèle ainsi que la sémantique d'adaptation associée. Cette sémantique d'adaptation est composée de vérifications et d'actions qui, une fois assemblées, permettent de construire des politiques d'adaptation.

Dans ce chapitre, nous allons montrer comment le concept des familles vient en aide à l'ingénieur logiciel pour la construction de ces politiques d'adaptation. D'abord, la famille est une structure permettant d'organiser les différents éléments d'adaptation. Ensuite, les familles peuvent être spécialisées. Grâce à leur structure, l'ingénieur pourra d'un seul coup d'œil identifier précisément les éléments d'adaptation dont il aura besoin. Définir des vérifications et actions pour l'adaptation n'est pas intuitif quand nous sommes face à un *i-DSML* trop général. En effet, nous avons remarqué que c'est la spécialisation des méta-modèles qui permet véritablement l'émergence de politiques d'adaptation, *a fortiori* lorsqu'elles sont génériques. Grâce à la spécialisation de familles, l'ingénieur va pouvoir définir de nouvelles sous-familles, plus propices à l'adaptation, tout en profitant de la possibilité de réutiliser des politiques d'adaptation existantes.

Le concept des familles vise donc à simplifier le travail de l'ingénieur logiciel pour concevoir ses applications adaptatives. Une famille contient des opérations d'exécution, des vérifications d'adaptation et des actions d'adaptation que nous appelons « attributs ». Il suffit pour l'ingénieur logiciel de choisir la famille associée au méta-modèle pour accéder à tous ses attributs. Ces attributs, une fois sélectionnés et assemblés pour former des politiques d'adaptation, constitueront la sémantique d'adaptation de son application. Ce concept des familles, repose à la fois sur des idées venant du domaine de l'architecture logicielle et de la communauté IDM. Nous avons appliqué ces idées sur le récent concept d'*i-DSML*.

En effet, notre inspiration provient du travail réalisé sur les styles architecturaux, un thème de recherche fleurissant durant la fin des années 1990 [29, 77, 76]. Un style architectural définit une famille d'architectures logicielles similaires (par exemple client/serveur, *pipe&filter*, ...) et fournit des éléments de conception spécifiques et des règles régissant leur disposition. Dans [29], Garlan *et al.* mettent en avant une relation de spécialisation qui peut s'appliquer sur des styles (par exemple *pipeline* est un sous-style

de *pipe&filter*) de telle façon que certains langages de description d'architecture (*ADL* pour l'acronyme anglais) ont supporté cette fonctionnalité expérimentale, comme *Acme* [30] ou encore *ArchWare* [65]. Pendant ce temps, certains auteurs se sont intéressés à comment le concept de style pouvait faciliter l'(auto)-reconfiguration de systèmes à l'exécution [17, 67] en se basant sur des politiques d'adaptation définies au niveau architectural et avec une boucle d'adaptation très proche du cas de la figure 2.7, rétrospectivement parlant.

Parallèlement, Steel *et al.* [78] ont proposé le typage de modèles où un modèle, conforme à un méta-modèle, pouvait adhérer à un ou plusieurs types de la même manière qu'une architecture décrite avec un *ADL* pouvait en plus satisfaire un ou plusieurs styles. Guy *et al.* [36] ont continué ces travaux à travers l'étude de la relation de sous-typage et plus généralement l'influence d'un tel système de types au niveau des modèles [81] mais sans prendre en compte l'aspect adaptatif. Nous allons réutiliser ces travaux pour la définition de familles puisque nous verrons qu'en spécialisant une famille, nous spécialisons également le méta-modèle d'un *i-DSML*, ce qui nous amène à définir des sous-types de modèles.

## 4.1 L'adaptation des *i-DSML* : un autre exemple

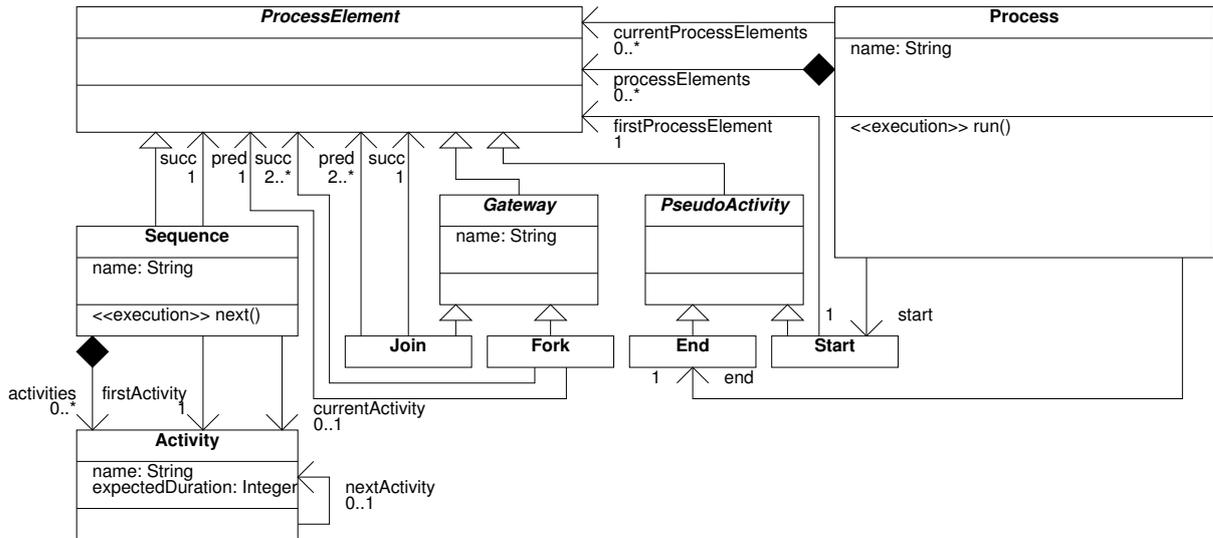
Dans le chapitre précédent, nous avons illustré nos propos à l'aide d'un *i-DSML* pour des machines à états. Afin de montrer que notre approche peut s'appliquer à d'autres exemples, nous allons cette fois considérer l'*i-DSML* nommé *Process Description Language (PDL)*, qui permet de modéliser toute sorte de processus en une liste ordonnée d'activités. Celui-ci a l'avantage d'offrir des cas d'adaptation plus intuitifs que celui du chapitre précédent. Il est librement inspiré de langages de processus connus tels que *WS-BPEL* [54], *SPEM* ou encore *BPMN* [56] qui sont typiquement couplés à des moteurs pour leur exécution. Ici, c'est une version simple qui supporte les activités en parallèle et inclut une notion de temps (dates butoirs).

L'adaptation présentée dans ce chapitre, comme celle du chapitre précédent, est réalisée par le système lui-même et nous sommes donc toujours dans un cas d'auto-adaptation. D'autre part, nous verrons que cette adaptation permet d'altérer le processus, comme par exemple sauter des activités facultatives. En ce sens, comme le processus réalisé ne fera pas forcément la même chose que celui initialement spécifié, nous sommes dans le cas d'une adaptation fonctionnelle. Puisque nous prévoyons à l'avance que le processus peut être en retard, nous travaillons avec une adaptation anticipée. La façon dont est implémenté le comportement de notre processus implique que si nous le relançons dans un contexte identique il se comportera de la même manière, ce qui nous assure que notre adaptation est prédictible. En outre, l'adaptation que nous proposons dans ce chapitre est générique puisque nous verrons qu'elle s'appuie sur des méta-éléments communs à tout processus quel que soit son contenu métier. Par conséquent, cet exemple correspond à de l'auto-adaptation fonctionnelle, anticipée, prédictible et générique.

Le méta-modèle et la sémantique d'exécution de cet *i-DSML* sont décrits, puis un exemple concret de son utilisation est fourni. Ensuite, des questions concernant ses possibles adaptations sont soulevées.

### 4.1.1 La définition du méta-modèle *PDL*

Dans cette sous-section, l'*i-DSML PDL* est décrit en suivant la caractérisation détaillée dans le chapitre précédent. La figure 4.1 définit le méta-modèle de cet *i-DSML*. La partie statique du méta-modèle de notre *i-DSML* définit les éléments qui ont pour objectif de former la structure du processus. Ces éléments sont


 FIGURE 4.1 – La définition du méta-modèle *PDL*

nombreux et abstraits sous le méta-élément **ProcessElement**. Le principal élément de processus concret est une séquence (**Sequence**) contenant un ensemble d'activités (**Activity**). Ces activités à l'intérieur d'une séquence sont ordonnées de telle manière que chaque activité (exceptée la dernière) ait une activité suivante. Chaque activité a une durée attendue qui correspond au temps idéal écoulé pour compléter la tâche. Les séquences d'activités peuvent être parallélisées à l'aide de portes (**Gateway**). Une fourche (**Fork**) a pour objectif de mettre en parallèle plusieurs séquences tandis qu'un raccord (**Join**) est un point de synchronisation pour plusieurs séquences. Chaque processus contient exactement deux pseudo-activités (**PseudoActivity**) : l'une définissant son début (**Start**) et l'autre sa fin (**End**).

La partie dynamique du méta-modèle de notre *i-DSML*, quant à elle, définit l'état courant du modèle durant son exécution. Cet état courant correspond aux deux méta-associations **currentProcessElement** et **currentActivity**. La première indique, pour un processus, quels éléments de processus sont actuellement actifs. La deuxième donne, pour une séquence, son activité courante. La combinaison de ces deux fournit un état global du modèle lors de son exécution qui est modifié après chaque pas d'exécution.

Les parties statiques et dynamiques du méta-modèle sont complétées avec des invariants *OCL* imposant aux modèles d'être bien formés. Puisqu'ils sont très nombreux, nous allons nous contenter de lister uniquement ceux en référence à la méta-classe **Activity** dans un souci de clarté (cf. listing 4.1). Nous souhaitons éviter les boucles que ce soit sur les activités, sur les séquences, sur les fourches et raccords ou encore sur le début du processus. Ainsi, l'invariant à la ligne 11 du listing 4.1 vérifie que l'activité suivante d'une activité ne soit pas la même activité. Nous devons faire pareil pour le successeur d'une séquence qui ne doit pas être la même séquence. Concernant les fourches et raccords, le prédécesseur d'une fourche ne doit pas correspondre à cette fourche, les successeurs d'une fourche ne doivent pas contenir cette fourche, le successeur d'un raccord ne doit pas correspondre à ce raccord et les prédécesseurs de ce raccord ne doivent pas contenir ce raccord. Au niveau du début d'un processus, il faut veiller à ce qu'il ne se référence pas lui-même. D'autre part, nous considérons important d'utiliser des noms uniques pour distinguer sans ambiguïté les activités, les séquences et les portes. Ainsi, l'invariant à la ligne 15 du listing 4.1 vérifie qu'une activité n'ait pas le même nom qu'une autre activité dans le processus. Nous devons faire pareil pour les séquences et les portes. En outre, la durée attendue d'une activité ne doit pas être négative (cf.

invariant à la ligne 13 du listing 4.1). De plus, nous souhaitons qu'une activité dans une séquence ne soit pas à la fois la première activité de cette séquence ainsi qu'une activité suivant une activité de cette séquence (cf. invariant à la ligne 17 du listing 4.1). De même, il doit y avoir exactement une seule dernière activité dans une séquence (cf. invariant à la ligne 19 du listing 4.1). Toutes les activités référencées au sein d'une séquence (c'est-à-dire la première activité ainsi que les activités suivantes) doivent appartenir aux activités de cette séquence (cf. invariant à la ligne 21 du listing 4.1).

```

1 context Activity def: isNotInNext(a: Activity): Boolean =
2   if self.nextActivity.oclIsUndefined() then
3     true
4   else
5     if self.nextActivity = a then
6       false
7     else
8       self.nextActivity.isNotInNext(a)
9   endif
10 endif
11 context Activity inv noActivityLoop:
12   self.isNotInNext(self)
13 context Activity inv noNegativeActivityExpectedDuration:
14   self.expectedDuration > 0
15 context Activity inv noSameActivityName:
16   Activity.allInstances()->forall(a1: Activity, a2: Activity | a1 <> a2 implies a1.name
17     <> a2.name)
17 context Sequence inv noFirstAndNextSequenceActivity:
18   not self.activities->exists(a: Activity | self.firstActivity = a.nextActivity)
19 context Sequence inv noMoreOrLessThanOneLastSequenceActivity:
20   self.activities->select(a: Activity | a.nextActivity.oclIsUndefined())->size() = 1
21 context Sequence inv noReferencedActivityOutOfTheSequence:
22   self.activities->includes(self.firstActivity) and self.activities->forall(a: Activity |
23     not a.nextActivity.oclIsUndefined() implies self.activities->includes(a.
24     nextActivity))

```

Listing 4.1 – Des invariants *OCL* pour *PDL*

La sémantique d'exécution associée au méta-modèle de notre *i-DSML* exprime comment les éléments du modèle évoluent durant l'exécution. Pour rappel, la sémantique d'exécution modifie uniquement les éléments dynamiques du modèle et est implémentée au travers d'opérations d'exécution qui sont attachées aux méta-éléments. Ici, la sémantique d'exécution est embarquée à l'intérieur de l'opération `run()` dans `Process` et de l'opération `next()` dans `Sequence`. Pour des raisons de clarté, ces opérations spéciales sont préfixées par un stéréotype conceptuel «*execution*». L'opération `run()` lance l'exécution du processus. Sa première action consiste à exécuter l'élément de début du processus. Ensuite, il exécute son successeur et ainsi de suite jusqu'à atteindre le dernier élément du processus. Exécuter une séquence consiste à exécuter son opération `next()`. Si la séquence vient juste d'être lancée, sa première activité est exécutée. Sinon, c'est l'activité suivante de l'activité courante qui est exécutée. Une fois qu'une activité est terminée,

l'opération `next()` est rappelée et ainsi de suite jusqu'à atteindre la fin de la séquence. Exécuter une fourche consiste à exécuter toutes ses séquences successeurs. En revanche, exécuter un raccord consiste à attendre que toutes ses séquences prédécesseurs soient terminées avant d'exécuter le successeur.

#### 4.1.2 Un processus de développement logiciel défini en utilisant *PDL*

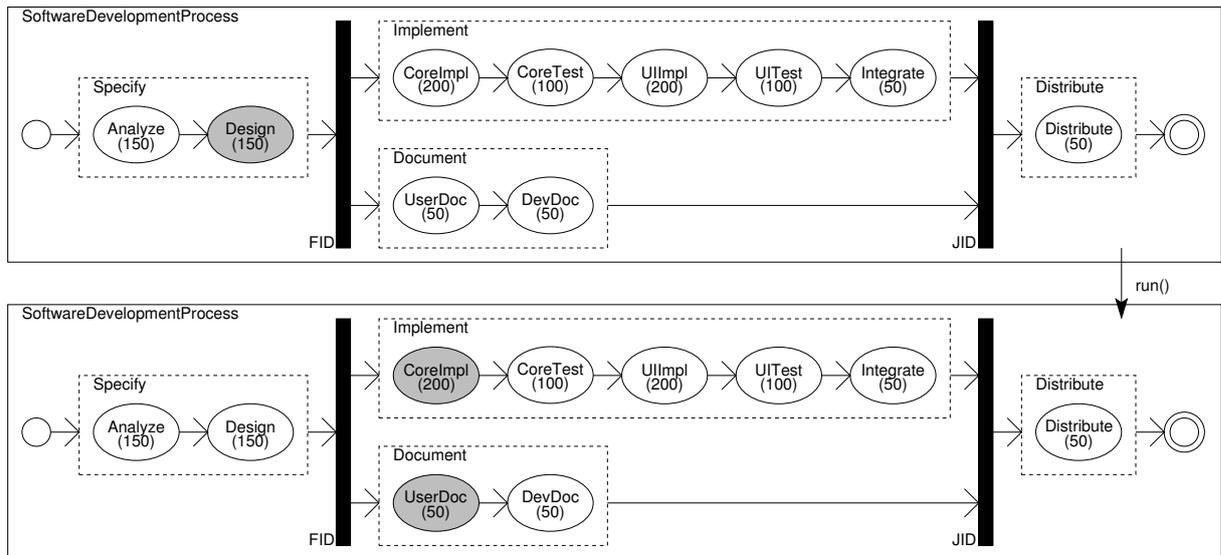


FIGURE 4.2 – Un modèle conforme au méta-modèle de *PDL* et son exécution

Comme exemple familier au domaine du génie logiciel, nous avons choisi de modéliser un processus de développement logiciel typique (cf. figure 4.2, partie haute). Ce processus contient quatre séquences (représentées par des rectangles en pointillés) : spécification (**Specify**), implémentation (**Implement**), documentation (**Document**) et distribution (**Distribute**) du logiciel. La spécification est la première tâche du processus tandis que la distribution est la dernière, une fois que tout le reste a été terminé. Entre les deux, l'implémentation et la documentation sont réalisées en parallèle. Cette parallélisation est obtenue grâce à l'aide de la fourche nommée *FID* (*Fork for Implement and Document*) et du raccord *JID* (*Join for Implement and Document*). La spécification contient deux activités (représentées par des ellipses) : l'analyse suivie par la conception. Le nombre sous le nom de l'activité, ici 150 pour chacune, est la durée attendue de l'activité (sa date butoir). La documentation contient les activités de documentation pour l'utilisateur et de documentation pour le développeur. L'implémentation est la plus longue des séquences. Elle commence avec l'implémentation du cœur du logiciel (**CoreImpl**), qui correspond à la logique métier et à l'implémentation des services. Ensuite, nous trouvons son activité de test associée (**CoreTest**). Après cela, l'interface utilisateur est implémentée (**UIImpl**) et testée (**UITest**). Finalement, toutes les implémentations sont intégrées.

Nous commençons par la première activité de la première séquence (c'est-à-dire l'activité **Analyze**). Ensuite, nous continuons avec l'activité suivante de la séquence (c'est-à-dire l'activité **Design**). Cette situation est représentée en haut de la figure 4.2 où l'activité courante est coloriée en gris. Après cela, nous rencontrons une fourche, ce qui implique l'activation de la première activité de chaque successeur (c'est-à-dire les activités **CoreImpl** et **UserDoc**). Cette situation correspond au bas de la figure 4.2. Une

fois que chaque activité de ces deux séquences a été exécutée, nous sommes capables de traverser le raccord. La dernière activité exécutée est ensuite `Distribute`.

### 4.1.3 De l'intérêt de la spécialisation pour l'adaptation

Toujours en suivant la caractérisation donnée dans le chapitre précédent, un *i-DSML* est étendu de deux manières pour devenir adaptable. Premièrement, le méta-modèle est étendu avec des éléments dédiés à l'adaptation et deuxièmement, une sémantique d'adaptation est associée au méta-modèle et implémentée par un moteur d'exécution qui devient donc *de facto* un moteur d'adaptation. Comme vu dans le chapitre précédent, la sémantique d'adaptation a pour objectif d'agir sur le modèle pour l'adapter. Cette adaptation peut impacter tout le modèle, incluant la modification, la création ou la suppression d'éléments statiques, dynamiques ou d'adaptation. Concrètement, la sémantique d'adaptation est implémentée au travers de deux sortes d'opérations. La première est une opération de requête retournant un booléen et exprimant si une adaptation est nécessaire ou pas. Ces opérations sont appelées des vérifications d'adaptation. La seconde est une opération d'action qui applique une adaptation sur le modèle. Ces opérations sont appelées des actions d'adaptation.

Concernant l'*i-DSML PDL*, à première vue, une situation nécessitant une adaptation apparaît lorsque le processus est en retard. Cependant, pour évaluer ce cas, nous avons besoin de la différence entre le temps attendu et le temps qui s'est réellement écoulé. C'est pour cette raison qu'un élément d'adaptation a été ajouté au méta-modèle : un attribut `elapsedTime` dans le méta-élément `Process`. Cette simple extension du méta-modèle est intuitive car avoir un temps écoulé est le complément logique d'une durée attendue pour une exécution de processus. Il est maintenant possible d'implémenter une vérification d'adaptation qui va déterminer si nous sommes en retard durant l'exécution du processus. Cette opération va tout simplement comparer les durées attendues de toutes les activités déjà terminées au temps qui s'est effectivement écoulé.

Maintenant la question est : « Si nous sommes en retard, que devons-nous faire ? ». La réponse est : « Avec la définition actuelle de l'*i-DSML*, nous ne savons pas ! ». En effet, il n'y a pas d'action d'adaptation intuitive qui peut être réalisée sans ajouter des informations supplémentaires. Nous pouvons imaginer de supprimer des activités dans le reste du processus. Lesquelles ? Un effacement arbitraire n'est clairement pas une bonne idée. Dans l'exemple, les activités de documentation et de test peuvent être facultatives. Nous savons cela parce que nous avons une connaissance métier sur, de façon générale, ce qu'est réellement un processus de développement logiciel. La documentation n'est jamais au cœur de la production d'un logiciel et les tests, dans le pire des cas, peuvent éventuellement être sautés en dernier recours. Le problème est que le moteur d'exécution n'a pas de vision métier et donc, il ne dispose pas de cette connaissance. En effet, le moteur réalise l'adaptation pour n'importe quel modèle, que ce soit un processus de développement logiciel ou une recette de cuisine. Une autre idée pourrait être de paralléliser quelques activités pour réduire le temps d'exécution du processus. Typiquement, d'un point de vue de la connaissance métier, nous savons que le développement du cœur du logiciel peut potentiellement être effectué en parallèle avec le développement de l'interface utilisateur. Cette fois encore, le moteur ne possède pas cette connaissance métier. Que l'action d'adaptation consiste à supprimer des activités ou à en paralléliser, nous voyons bien qu'il manque des informations nécessaires à sa réalisation. Par conséquent, nous devons étendre la partie adaptative du méta-modèle de notre *i-DSML* afin d'ajouter les méta-éléments sur lesquels reposent nos actions d'adaptation.

En résumé, avec la simple définition de l'*i-DSML* (incluant la petite et intuitive extension qui vient d'être présentée), il n'est pas possible de savoir comment adapter un modèle conforme au méta-modèle de *PDL*. Autrement dit, il est impossible de définir une sémantique d'adaptation car le modèle est trop général. Cependant, nous pensons qu'avec des informations additionnelles, une action d'adaptation peut être définie et faire sens. Le méta-modèle a donc besoin d'être spécialisé pour embarquer ces informations complémentaires. Le constat est que, comme expliqué dans le chapitre précédent, plus un méta-modèle est spécialisé et contraint, plus la manipulation automatique de modèle devient envisageable. Aussi, la définition d'une sémantique d'adaptation plus pertinente est rendue possible. Dans la section suivante, nous définissons le concept de famille qui est une spécialisation d'*i-DSML* associée à une sémantique d'adaptation dédiée. Ensuite, dans la section 4.3, nous définissons une adaptation concrète de familles pour l'*i-DSML PDL*.

## 4.2 La gestion de l'adaptation des *i-DSML* par les familles

Si nous considérons le fait qu'à partir d'un *i-DSML* minimal, nous pouvons créer plusieurs extensions, chacune fournissant une base pour des adaptations dédiées, il devient souhaitable d'organiser tous ces éléments. C'est pourquoi nous proposons les familles d'adaptation et la relation de spécialisation entre elles.

### 4.2.1 La définition d'une famille

Chaque méta-modèle d'un *i-DSML* définit des méta-éléments ainsi qu'un ensemble d'opérations implémentables. Ces opérations reposent sur ces méta-éléments et définissent une sémantique d'exécution ou d'adaptation. Comme ces opérations sont attachées à un méta-modèle donné, nous proposons de tout regrouper sous un unique concept : la famille. Voici la définition d'une famille :

**Définition 1.** *Une famille lie un méta-modèle et un ensemble d'opérations associées (des opérations d'exécution, des vérifications d'adaptation et des actions d'adaptation). Elle fournit un guide à l'ingénieur logiciel qui peut assembler les opérations disponibles.*

Ainsi, une famille est vue comme un cadre dans lequel l'ingénieur peut puiser des éléments de solution existants en toute confiance. Cela dit, elle ou il est responsable de les orchestrer correctement. Une famille est identifiée par un nom unique référençant son méta-modèle et contient trois sortes d'éléments :

1. les opérations d'exécution : des opérations qui contrôlent le flot d'exécution du modèle
2. les vérifications d'adaptation : des opérations booléennes exprimant si le modèle est actuellement adapté à son contexte et qu'il respecte des contraintes spécifiques (s'il ne l'est pas, une adaptation doit être réalisée)
3. les actions d'adaptation : des opérations qui modifient le contenu du modèle dans le but de l'adapter ou qui modifient la sémantique d'exécution ou d'adaptation (cf. le tableau 3.1)

Nous appelons « attributs » tous ces éléments à l'intérieur des familles.

### 4.2.2 La spécialisation des familles

La conclusion de la discussion à la sous-section 4.1.3 était que spécialiser un méta-modèle est pertinent pour définir une adaptation. L'*i-DSML PDL* a d'abord été étendu pour définir une vérification d'adapta-

tion, puis la discussion s'est terminée sur la nécessité de l'étendre un peu plus afin d'avoir la possibilité de définir une sémantique d'adaptation concrète. En conséquence, nous proposons de spécialiser des familles et ainsi de construire une hiérarchie de familles pour définir des sémantiques d'adaptation.

La spécialisation des familles est basée sur la spécialisation des méta-modèles. Le typage et le sous-typage de modèles, qui est une relation de spécialisation entre des méta-modèles ou des parties de méta-modèles, ont été définis dans [78, 36, 81]. La spécialisation des méta-modèles que nous utilisons dans notre approche est basée sur cette définition. Un méta-modèle définit une structure (c'est-à-dire un ensemble de méta-éléments associés) et est augmenté avec un ensemble d'invariants, typiquement écrits avec *OCL*, pour spécifier des règles forçant ainsi le modèle, conforme à ce méta-modèle, à être bien formé. Dans la même idée que les profils *UML*, une spécialisation d'un méta-modèle étend strictement deux parties du méta-modèle :

**Définition 2.** *Un méta-modèle  $MM'$  est une spécialisation d'un méta-modèle  $MM$  si  $MM'$  étend la structure et/ou les invariants de  $MM$ .  $MM'$  est construit en ajoutant à  $MM$  des nouveaux méta-éléments, des nouveaux attributs dans les méta-éléments, des nouvelles opérations dans les méta-éléments et/ou des nouvelles méta-associations entre les méta-éléments sans retirer aucun élément existant de  $MM$ .  $MM'$  définit des invariants supplémentaires sans retirer ceux existants sur  $MM$ .*

La spécialisation d'une famille correspond à la définition d'une sous-famille à partir d'une super-famille. Elle est simplement réalisée par, dans un premier temps, la spécialisation de son méta-modèle. Comme cette spécialisation est une extension stricte (c'est-à-dire elle ne retire rien), toutes les assertions que nous avons faites à propos de la super-famille sont aussi appliquées à toutes les sous-familles. Les sous-familles héritent des opérations d'exécution, des vérifications d'adaptation et des actions d'adaptation provenant de la super-famille. Tout ce qui peut être fait (d'un point de vue exécution ou adaptation) avec un modèle de la super-famille peut également être fait avec un modèle de la sous-famille.

Dans un second temps, en plus des nouveaux éléments (structurels ou invariants) dans le méta-modèle, de nouveaux attributs peuvent être définis pour une sous-famille. Ces attributs sont des opérations d'exécution, des vérifications d'adaptation et des actions d'adaptation. En résumé, une spécialisation de famille est définie ainsi :

**Définition 3.** *Une famille  $F'$  est une spécialisation d'une famille  $F$  si le méta-modèle de  $F'$  est une spécialisation du méta-modèle de  $F$ .  $F'$  hérite de tous les attributs (opérations d'exécution, vérifications d'adaptation et actions d'adaptation) de  $F$  et peut en définir de nouveaux.*

L'héritage multiple entre méta-modèles et familles est autorisé. Cependant, les conflits sont supposés être évités via une conception rigoureuse. Puisque les familles peuvent hériter les unes des autres, il est donc possible de définir une hiérarchie de familles. La racine d'une hiérarchie de familles est un *i-DSML* offrant seulement des modèles exécutables sans aucune préoccupation d'adaptation. L'*i-DSML* de la famille à la racine peut être spécialisé pour définir d'autres *i-DSML* (sans adaptation) ou des *i-DSML* adaptables (incluant l'adaptation). Plus une famille est placée en bas de la hiérarchie, plus son méta-modèle est spécialisé et permet de définir des politiques d'adaptation spécifiques. Tout comme en programmation orientée objet où des classes abstraites ne pouvant être instanciées servent de base à des sous-classes, lors de la construction d'une hiérarchie de familles, des familles abstraites peuvent être définies pour servir de base à des sous-familles.

**Définition 4.** *Dans le cas d'une famille racine, celle-ci est abstraite si ses attributs ne permettent pas de définir une sémantique d'exécution. Dans le cas d'une famille spécialisée, celle-ci est abstraite si les*

*modifications du méta-modèle et les nouveaux attributs ne permettent pas de définir une ou plusieurs nouvelles sémantiques par rapport à celles héritées, soit d'exécution dans le cas d'une famille d'exécution, soit d'adaptation autrement.*

### 4.2.3 Les politiques d'adaptation domaine et métier

Une autre notion émerge des idées précédentes même s'il peut être délicat de parvenir à la formaliser. En suivant les concepts de généralisation et de spécialisation, une famille peut représenter un ensemble de modèles plus ou moins liés au métier. Une famille non liée au métier se situe au niveau domaine et exprime une politique d'adaptation qui peut être appliquée sur n'importe quel modèle conforme au méta-modèle de cette famille, indépendamment de son contenu. Elle est dite « domaine », au sens de « générique », parce qu'elle est basée seulement sur la construction du langage de modélisation spécifique à un domaine, avec le domaine correspondant ici au « *D* » de *DSML*, donc au langage de modélisation. Au contraire, une politique d'adaptation métier est basée sur des éléments spécifiques au métier contenus dans le modèle.

Par exemple, la famille qui contraint un processus à avoir au moins une fourche n'est pas liée au métier et est donc située au niveau domaine. En effet, tout processus peut potentiellement satisfaire cette contrainte, quel que soit son contenu métier (par exemple une recette de cuisine, un développement logiciel, ...). Ainsi, les adaptations associées peuvent être réutilisées à travers une large variété de modèles. Au contraire, contraindre un processus à avoir une activité nommée « battre les œufs » brise la neutralité de la famille face au niveau domaine puisqu'il est supposé que le processus appartienne au métier des recettes de cuisine. Dans ce cas, l'adaptation écrite pour une telle famille peut être très précise mais est loin d'être aussi réutilisable.

Techniquement parlant, la frontière entre les niveaux domaine et métier est quelque peu floue. Cependant, nous pouvons dire que si la sémantique d'adaptation, à l'intérieur d'une vérification d'adaptation ou d'une action d'adaptation, est basée sur une valeur littérale (par exemple la chaîne de caractères « *battre les œufs* »), alors l'adaptation sera considérée comme étant au niveau métier. En résumé, nous avons donc trois types de familles (et donc de méta-modèles d'*i-DSML*) : celles utilisées pour de l'exécution pure, celles destinées à de l'adaptation au niveau domaine et celles vouées à de l'adaptation métier.

### 4.2.4 La définition du méta-modèle *FHDL*

Comme indiqué dans la sous-section 4.2.2, nous pouvons représenter une hiérarchie de familles où les familles héritent les unes des autres. Pour cela, nous avons besoin d'un langage spécifique dédié à la représentation de cette hiérarchie. Nous avons nommé ce langage *Family Hierarchy Description Language (FHDL)*, ce qui a pour avantage d'indiquer clairement ce à quoi il est destiné. Il est important de noter que, cette fois, nous avons à faire à un *DSML* (sans le « *i* »), puisque ce langage n'est pas exécutable. En effet, c'est un simple langage de description et donc, contrairement à *PDL* présenté en sous-section 4.1.1, celui-ci n'est pas interprété par un moteur d'exécution. La figure 4.3 définit le méta-modèle de ce *DSML*.

L'élément racine de ce méta-modèle est la hiérarchie de familles. Celle-ci contient un ensemble de familles elles-mêmes composées d'attributs. Les familles pouvant être abstraites ou non, un attribut booléen `isAbstract` dans le méta-élément `Family` précise cette information. Quant au type de chaque famille, il est obtenu grâce à l'attribut `kind` de ce même méta-élément. Nous pouvons remarquer que,

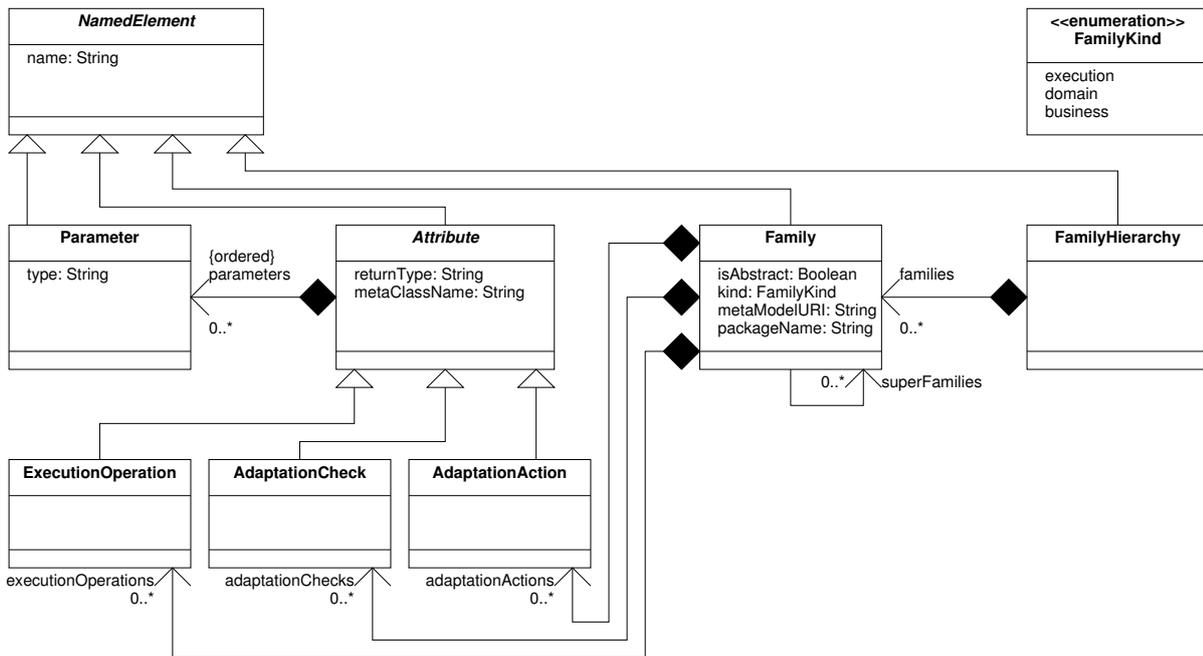


FIGURE 4.3 – La définition du méta-modèle *FHDL*

grâce à l'énumération `FamilyKind`, les seules valeurs possibles pour cet attribut sont : exécution, domaine ou métier. La famille étant associée à un méta-modèle, il est logique de trouver l'*Uniform Resource Identifier (URI)*, permettant d'identifier précisément ce dernier. Celui-ci est indiqué à l'aide de l'attribut `metaModelURI` présent dans le méta-élément `Family`. Un méta-modèle associé à une famille regroupe tous ses méta-éléments au sein d'un même paquetage. Ainsi, pour retrouver ces méta-éléments, il est nécessaire de connaître le nom de ce paquetage. Pour cette raison, un attribut `packageName` est disponible dans la méta-classe `Family`. Afin de conserver l'information concernant l'ensemble des super-familles d'une famille, une référence `superFamilies` est ajoutée à ce même méta-élément.

Concernant les attributs, nous avons vu qu'ils peuvent être de trois sortes (opération d'exécution, vérification d'adaptation et action d'adaptation). Par conséquent, nous avons trois liens d'héritage vers la méta-classe `Attribute` qui est abstraite (d'où la notation en italique de son nom). Un attribut correspond à une opération disponible dans le méta-modèle associé à la famille. Nous retrouvons donc logiquement l'ensemble des informations permettant d'identifier de façon unique une opération dans ce méta-modèle : sa signature. Celle-ci est composée d'un nom, de paramètres ordonnés et d'un type de retour (ce dernier correspondant à l'attribut `returnType` du méta-élément `Attribute`). Cette opération est attachée à un méta-élément dans le méta-modèle. Afin de retrouver facilement ce méta-élément, nous avons besoin d'indiquer le nom de celui-ci. C'est le rôle de l'attribut `metaClassName` présent dans la méta-classe `Attribute`.

Nous pouvons constater que tous les méta-éléments présentés ici doivent porter un nom qui permet de mieux les identifier. Ainsi, il semble cohérent de créer une méta-classe abstraite `NamedElement` qui permet de donner un nom à chacun d'eux. Cette façon de faire peut rappeler le méta-modèle *UML* qui possède, tout comme ici, un méta-élément abstrait de même nom et utilisé pour la même raison.

Le méta-modèle sur la figure 4.3 constitue la syntaxe abstraite de notre *DSML FHDL*. Cette syntaxe représentant la structure de notre langage permet de définir une hiérarchie de familles, mais reste

insuffisante pour pouvoir la visualiser graphiquement. La sous-section suivante vient combler ce manque en indiquant cette fois la syntaxe concrète du *DSML FHDL*. Grâce à cette syntaxe complémentaire, qui donne une notation pour chaque élément structural de ce langage, nous serons en mesure de représenter une hiérarchie de familles.

### 4.2.5 La représentation d'une hiérarchie de familles

La sous-section précédente a présenté la syntaxe abstraite de notre *DSML FHDL*. Dans cette sous-section nous allons décrire sa syntaxe concrète pour permettre de représenter n'importe quelle hiérarchie de familles à l'aide d'une même notation. La figure 4.4 montre la définition d'une hiérarchie de familles générique, basée sur cette syntaxe concrète, permettant d'illustrer les propos de cette sous-section.

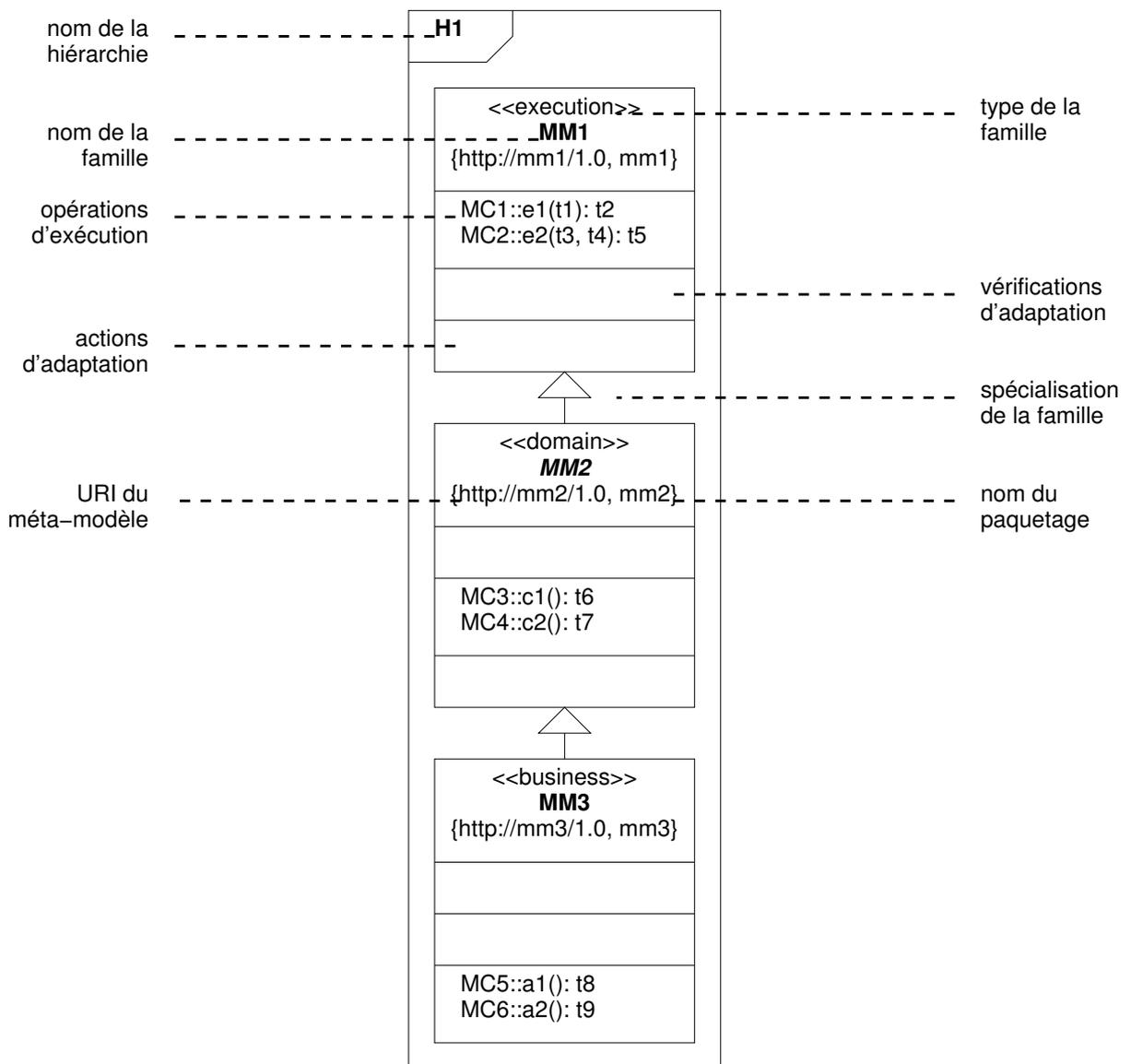


FIGURE 4.4 – Un exemple généraliste d'une hiérarchie de familles

Comme nous pouvons le constater, une hiérarchie de familles est décrite graphiquement à l'aide d'un

ensemble de boîtes reliées par des flèches creuses. Chaque boîte représente une famille de la hiérarchie tandis que les flèches indiquent la super-famille d'une sous-famille. Si nous nous trouvons dans un cas d'héritage multiple, alors plusieurs flèches peuvent partir d'une même boîte.

Ces boîtes sont séparées en quatre compartiments. Le premier contient le nom de la famille écrit en gras et aligné au centre. Si la famille est une famille abstraite alors son nom sera également noté en italique. Ce nom est identique à celui du méta-modèle associé à cette famille. Au-dessus de ce nom, nous trouvons le type de cette famille indiqué entre guillemets et aligné au centre. Nous aurons le choix entre les stéréotypes conceptuels «*execution*», «*domain*» ou «*business*». Au-dessous du nom de la famille, nous indiquons entre accolades et alignés au centre l'*URI* du méta-modèle associé à la famille ainsi que le nom du paquetage. Pour délimiter ces deux informations, une virgule est utilisée en tant que séparateur. Cette notation entre accolades est inspirée du méta-modèle *UML* dans lequel les méta-éléments peuvent être associés à des valeurs marquées représentées de la même façon.

Les trois autres compartiments servent respectivement à présenter les opérations d'exécution, les vérifications d'adaptation et les actions d'adaptation de la famille. Pour chaque attribut, une ligne est réservée à la présentation de sa signature alignée à gauche. Ainsi, nous commençons par indiquer le nom de sa méta-classe, puis son nom. Le séparateur utilisé pour délimiter ces deux informations est le double deux-points (: :). Pour compléter la signature, le type des paramètres de l'attribut est noté entre parenthèses juste après. Ils peuvent éventuellement être précédés d'un nom, ce qui permettra de mieux identifier le rôle du paramètre. Dans ce cas facultatif, il faudra alors utiliser le deux-points comme séparateur entre le type du paramètre et le nom de ce dernier. Si plusieurs types sont à préciser, alors ils seront séparés par une virgule. À la fin de cette signature, le type de retour est présenté en le séparant du reste à l'aide d'un deux-points.

La hiérarchie de familles est elle aussi représentée à l'aide d'une boîte. Celle-ci contient un petit rectangle dans le coin supérieur gauche possédant un angle inférieur droit biseauté. Il est destiné à accueillir le nom de la hiérarchie qui doit être écrit en gras et aligné à gauche. Cette boîte est la plus grande du diagramme car elle contient l'ensemble des familles constituant la hiérarchie.

La hiérarchie sur la figure 4.4 est nommée H1. Elle est composée des trois familles MM1, MM2 et MM3. La première famille (MM1) définit deux opérations d'exécution  $MC1::e1(t1):t2$  et  $MC2::e2(t3, t4):t5$ . Ce premier attribut n'a qu'un seul paramètre ( $t1$ ) tandis que ce second en possède deux ( $t3$  et  $t4$ ). Cette famille n'indique aucune vérification d'adaptation ou action d'adaptation. Elle est donc logiquement dédiée à l'exécution seulement et c'est pour cette raison que son type est «*execution*». Cette famille est particulière puisqu'elle se situe au sommet de la hiérarchie. Nous pouvons la désigner comme étant la racine de cette hiérarchie de familles.

La deuxième famille (MM2) spécifie deux vérifications d'adaptation  $MC3::c1():t6$  et  $MC4::c2():t7$ . Nous pouvons remarquer que ces attributs n'ont aucun paramètre (les parenthèses sont vides). Par définition (cf. sous-section 4.2.1), comme ces deux attributs sont des vérifications d'adaptation, les types de retour  $t6$  et  $t7$  sont forcément des booléens. Cette famille spécialisée est abstraite puisqu'à ce niveau de la hiérarchie, aucune vérification ou action d'adaptation ne permet de définir une politique d'adaptation. Pour indiquer que cette famille est abstraite, son nom apparaît en italique. Contrairement à la famille précédente, celle-ci tend à gérer l'adaptation, ce qui signifie que son type est soit «*domain*», soit «*business*». Dans cet exemple, nous considérons que les vérifications d'adaptation de cette famille permettent de construire des politiques d'adaptation qui ne se basent pas sur un contenu métier du modèle. Ainsi, cette famille est typée par «*domain*».

La dernière famille (MM3) de cette hiérarchie H1 apporte avec elle deux actions d'adaptation MC5- : :a1() :t8 et MC6- : :a2() :t9. Ici aussi, les parenthèses vides signalent que ces attributs ne possèdent pas de paramètres. Cette famille se situe au niveau métier, ce qui est précisé par «**business**».

Dans cette sous-section, nous avons vu la syntaxe concrète de notre *DSML FHDL*. Elle est constituée de boîtes et de flèches et permet de représenter n'importe quelle hiérarchie de familles. Dans la section 4.3, nous allons exploiter cette syntaxe afin de modéliser la hiérarchie de familles correspondant à l'*i-DSML PDL* vu précédemment.

### 4.3 Une hiérarchie de familles pour *PDL*

Afin de mieux comprendre les idées développées dans la section précédente, nous considérons de nouveau l'exemple de l'*i-DSML PDL*, mais cette fois d'un point de vue du concept des familles. Grâce au langage *FHDL* pour lequel la syntaxe abstraite a été présentée dans la sous-section 4.2.4 tandis que la syntaxe concrète a été décrite dans la sous-section 4.2.5, nous allons construire une hiérarchie de familles *PDLHierarchy* au fur et à mesure que nous spécialisons l'*i-DSML PDL*. Toutes les spécialisations pour l'*i-DSML PDL* effectuées dans le reste de cette section étendent la structure du méta-modèle. Cependant, dans d'autres cas, restreindre les modèles possibles uniquement en ajoutant des invariants *OCL* est suffisant pour définir des politiques d'adaptation. Par exemple, dans le chapitre précédent, nous étudions l'adaptation d'une machine à états *UML* simple dans le cas d'événements inattendus. Avec une machine à états générale, aucune décision d'adaptation ne peut être prise. Cependant, imposer qu'une transition associée à chaque événement attendu part de chaque état de la machine à états nous permet de déterminer dorénavant si un événement est attendu ou non (en vérifiant s'il existe, ou pas, une transition associée). De plus, imposer qu'un événement donné cible toujours le même état rend possible de déduire automatiquement comment ajouter un état et ses transitions associées à un événement totalement inconnu. Ces deux restrictions sont définies uniquement au travers de deux invariants *OCL* sans aucune modification du méta-modèle mais ont pourtant un impact fort sur les politiques d'adaptation envisageables.

#### 4.3.1 La description de la famille *PDL*

La racine de la hiérarchie de familles est appelée *PDL*. Elle correspond à l'*i-DSML* présenté dans la sous-section 4.1.1. Comme cet *i-DSML* est seulement dédié à l'exécution, il n'y a pas de vérification d'adaptation ou d'action d'adaptation définie mais seulement des opérations d'exécution (`run()` dans `Process` et `next()` dans `Sequence`). Le modèle conforme au méta-modèle de cet *i-DSML* ne sait donc que s'exécuter de façon classique, tel que présenté dans la sous-section 4.1.2. Nous ajoutons cette première famille à notre hiérarchie de familles dans la figure 4.5.

#### 4.3.2 La description de la famille *AdaptPDL*

En suivant la discussion de la sous-section 4.1.3, une première extension simple du méta-modèle *PDL* consiste à ajouter un attribut entier `elapsedTime` au méta-élément `Process` qui indique le temps réel d'exécution du processus. Ceci nous amène à définir la famille *AdaptPDL* qui étend la famille *PDL*. Nous ajoutons cette deuxième famille à notre hiérarchie de familles dans la figure 4.6.

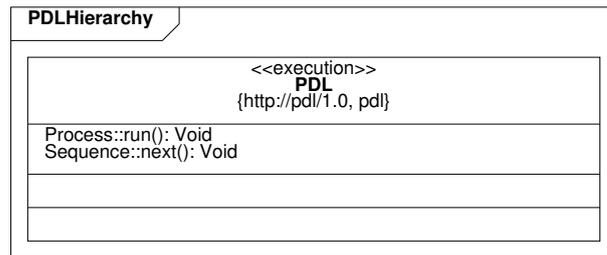


FIGURE 4.5 – Ajout de la famille *PDL* à la hiérarchie de familles *PDLHierarchy*

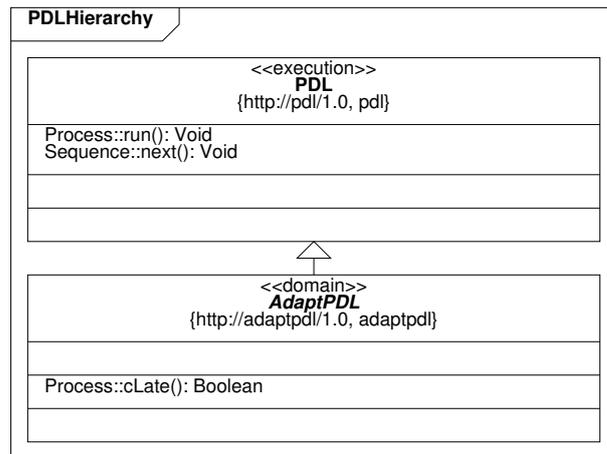


FIGURE 4.6 – Ajout de la famille *AdaptPDL* à la hiérarchie de familles *PDLHierarchy*

Grâce à l'attribut `elapsedTime`, il est dorénavant possible de déterminer si une exécution d'un processus est en retard en comparant la durée attendue avec le temps réellement écoulé depuis le début du processus. Cette vérification est réalisée par l'opération `cLate()` ajoutée au méta-élément `Process` et préfixée par un stéréotype conceptuel `<<check>>` comme nous pouvons le voir dans la figure 4.7. Elle apparaît également dans le troisième compartiment de la boîte de la famille *AdaptPDL* sur la figure 4.6. Cependant, comme expliqué dans la discussion de la sous-section 4.1.3, il n'est pas encore possible de définir d'actions d'adaptation avec ce méta-modèle. Ce sera fait avec les spécialisations suivantes en ajoutant de nouveaux éléments au méta-modèle.

Nous pouvons nous demander pourquoi est-il pertinent de définir une famille pour un *i-DSML* adaptable qui ne définit aucune action d'adaptation concrète. La raison est que l'attribut défini et la vérification d'adaptation associée peuvent être partagés par plusieurs sous-familles. Ces éléments sont donc destinés à être hérités dans toutes les sous-familles pour éviter de les définir plusieurs fois. La famille spécialisée *AdaptPDL* est abstraite puisqu'elle définit une seule vérification d'adaptation qui ne nous permet toujours pas de définir de politiques d'adaptation. Pour cette raison, son nom a été écrit en italique sur la figure 4.6.

Dans le modèle conforme au méta-modèle de cet *i-DSML*, nous avons maintenant l'information concernant le temps écoulé. Celle-ci est indiquée entre parenthèses, à droite du nom du modèle sur la figure 4.8. Comme nous pouvons le constater, même en cas de retard (350 unités de temps écoulées au lieu des 300 attendues), aucune adaptation n'est réalisée. La raison est qu'il nous manque une action d'adaptation, c'est pourquoi nous allons créer une sous-famille dans la sous-section suivante offrant cet attribut

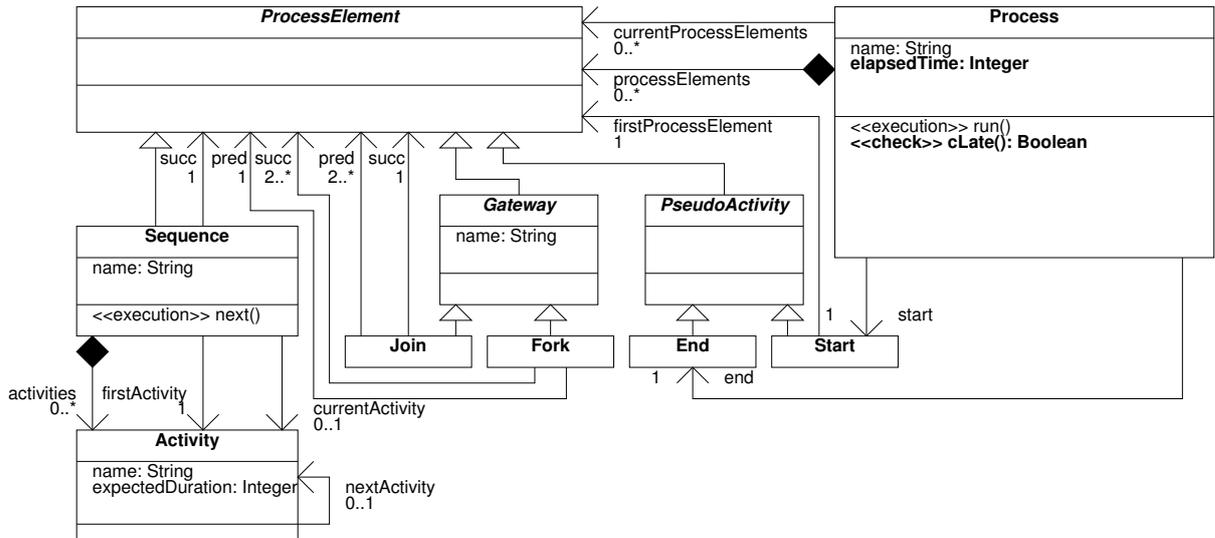


FIGURE 4.7 – L'*i-DSML* correspondant à la famille *AdaptPDL*

nécessaire à l'adaptation.

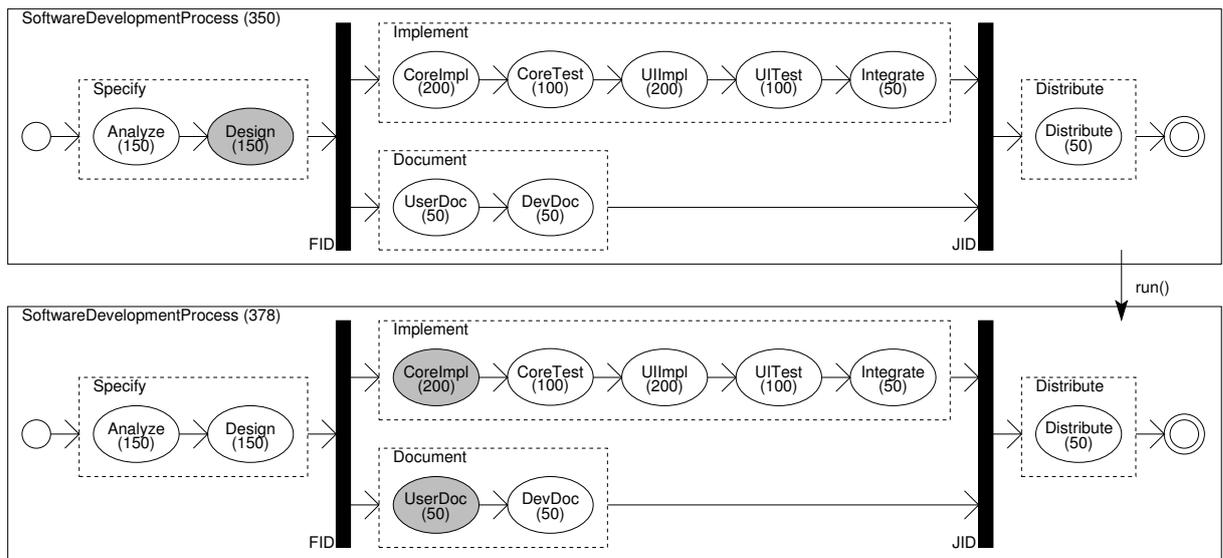


FIGURE 4.8 – Un modèle conforme au méta-modèle d'*AdaptPDL* et son exécution

### 4.3.3 La description de la famille *SkipAdaptPDL*

Comme proposé dans la sous-section 4.1.3, en cas de retard, une action d'adaptation pourrait être de supprimer les activités facultatives dans le reste du processus. Pour parvenir à déterminer les activités qui peuvent être retirées, nous avons besoin de les marquer. Pour cette raison, nous avons ajouté un attribut booléen `skippable` au méta-élément `Activity` comme nous pouvons le constater sur la figure 4.10. L'ingénieur logiciel doit maintenant indiquer quelles sont ces activités que nous pouvons supprimer dans la définition de son processus. Ceci nous amène à définir la famille *SkipAdaptPDL* qui étend la famille

*AdaptPDL*. Nous ajoutons cette troisième famille à notre hiérarchie de familles dans la figure 4.9.

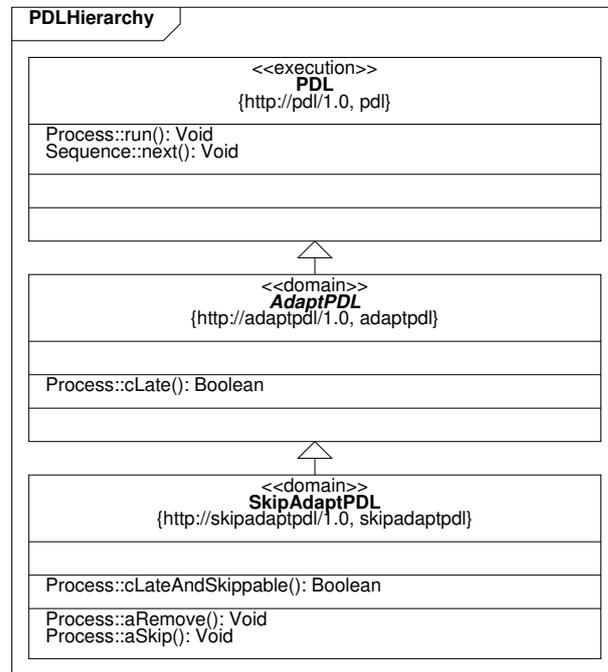


FIGURE 4.9 – Ajout de la famille *SkipAdaptPDL* à la hiérarchie de familles *PDLHierarchy*

Grâce à cet attribut, en plus de savoir si nous sommes en retard avec la vérification d'adaptation `cLate()`, il est dorénavant aussi possible de déterminer si la prochaine activité peut être évitée. Cette vérification est donc double et est réalisée par la vérification d'adaptation `cLateAndSkippable()` ajoutée au méta-élément `Process` comme nous pouvons l'observer sur la figure 4.10. Deux actions d'adaptation ont été définies. La première supprime les activités que nous pouvons sauter (c'est-à-dire elle modifie la partie statique de l'*i-DSML*). Cette action d'adaptation étant radicale, il est possible que l'ingénieur logiciel préfère une solution plus souple. Pour cette raison, au lieu de statiquement retirer des activités facultatives, nous proposons une autre action d'adaptation qui effectue seulement un saut de ces activités (c'est-à-dire elle modifie la partie dynamique de l'*i-DSML*, en mettant à jour la référence de l'activité courante pour qu'elle désigne l'activité suivante). Ces actions d'adaptation sont respectivement réalisées par les opérations `aRemove()` et `aSkip()` ajoutées au méta-élément `Process` comme nous pouvons le voir dans la figure 4.10. Elles sont préfixées par un stéréotype conceptuel «`action`». Comme ces opérations sont des actions d'adaptation, elles apparaissent dans le quatrième compartiment de la boîte de la famille *SkipAdaptPDL* sur la figure 4.9.

Dans le modèle conforme au méta-modèle de cet *i-DSML*, nous savons maintenant si une activité peut être évitée. La notation sur la figure 4.11 permettant de représenter cette nouvelle information consiste à utiliser des contours en pointillés pour les activités facultatives. Dans cet exemple, à l'intérieur de la séquence d'implémentation, `CoreTest` et `UITest` sont marquées comme facultatives. Nous sommes en retard (550 unités de temps écoulées au lieu des 500 attendues) et l'activité courante est `CoreImpl`. Une fois cette activité terminée, l'activité `CoreTest` est sautée et la prochaine activité exécutée est `UIImpl`.

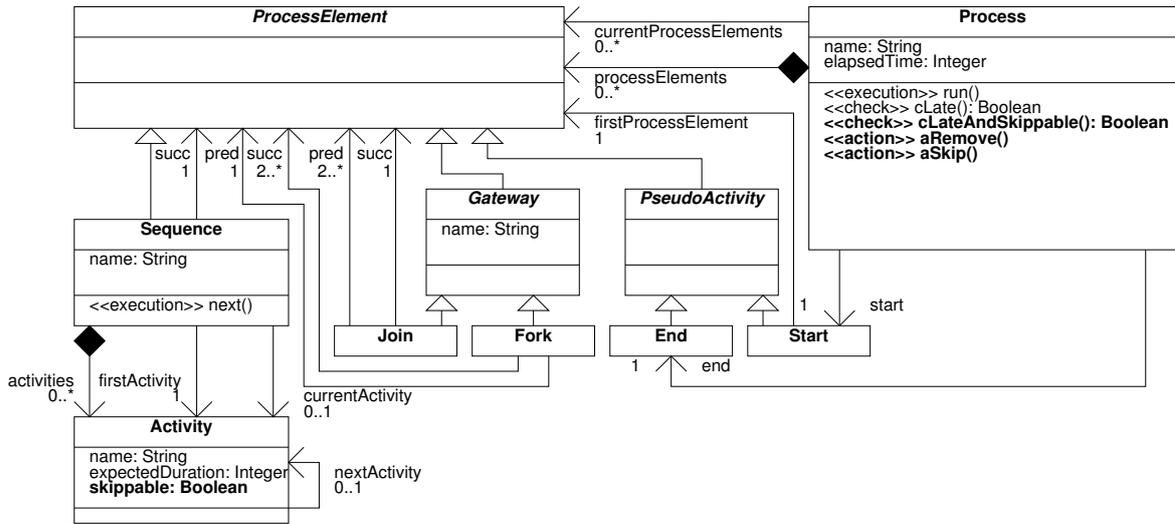


FIGURE 4.10 – L'*i-DSML* correspondant à la famille *SkipAdaptPDL*

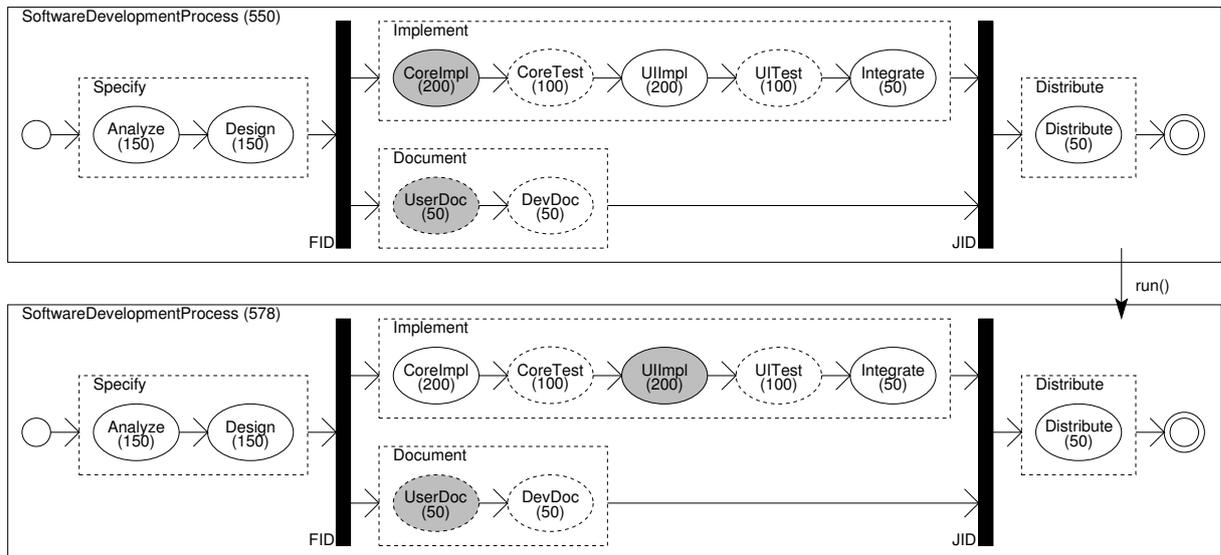


FIGURE 4.11 – Un modèle conforme au méta-modèle de *SkipAdaptPDL*, avant et après adaptation

#### 4.3.4 La description de la famille *DependAdaptPDL*

Dans la sous-section 4.1.3, au lieu de supprimer les activités facultatives nous proposons de paralléliser certaines activités dans le but de réduire le temps du processus. Évidemment, nous ne pouvons pas sélectionner au hasard les activités qui seront parallélisées parce qu'une activité peut dépendre d'une autre. En conséquence, cette adaptation peut être réalisée seulement si nous sommes informés des dépendances entre les activités. Afin de donner la liste de ces dépendances, nous ajoutons une référence `dependencies` au méta-élément `Activity` comme nous pouvons le constater sur la figure 4.13. Ceci nous amène à définir la famille *DependAdaptPDL* qui étend la famille *AdaptPDL*. Nous ajoutons cette quatrième famille à notre hiérarchie de familles dans la figure 4.12.

Grâce à cette référence, en plus de savoir si nous sommes en retard avec la vérification d'adaptation

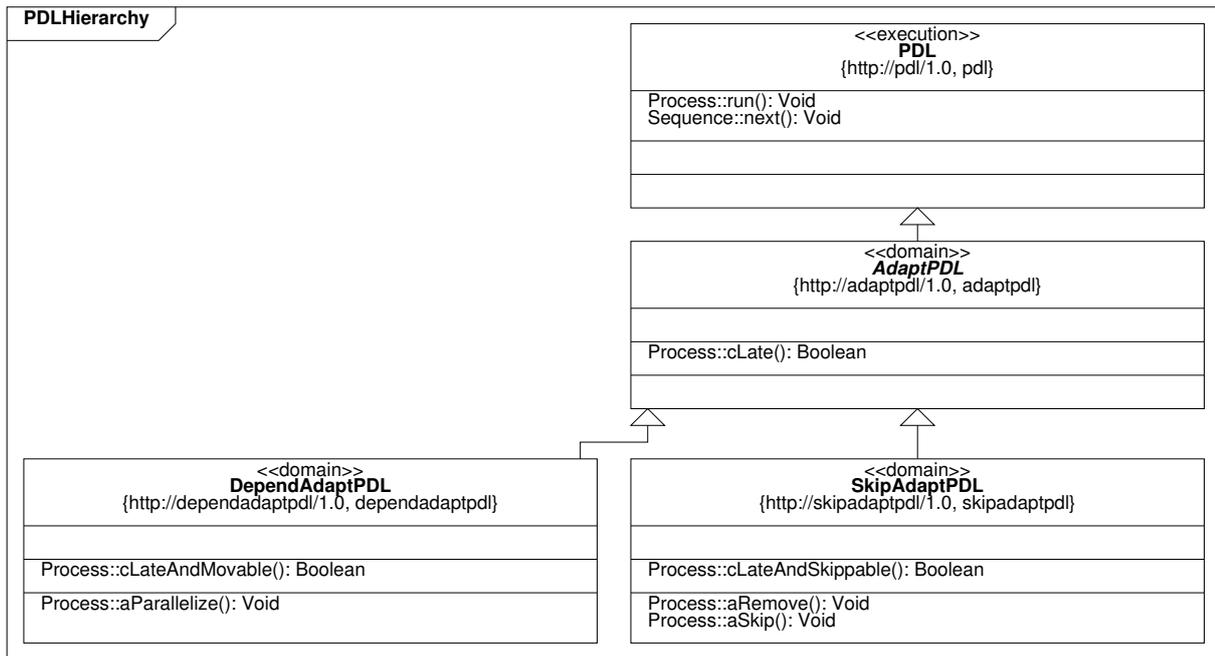


FIGURE 4.12 – Ajout de la famille *DependAdaptPDL* à la hiérarchie de familles *PDLHierarchy*

`cLate()`, il est dorénavant aussi possible de déterminer si la prochaine activité est déplaçable (en tenant compte de ses dépendances). Cette vérification est donc double et est réalisée par la vérification d'adaptation `cLateAndMovable()` ajoutée au méta-élément `Process` comme nous pouvons l'observer sur la figure 4.13. L'action d'adaptation qui correspond est de transformer des séquences uniques en séquences multiples placées en parallèle. Cette action d'adaptation est réalisée par l'opération `aParallelize()` ajoutée au méta-élément `Process` comme nous pouvons le voir dans la figure 4.13.

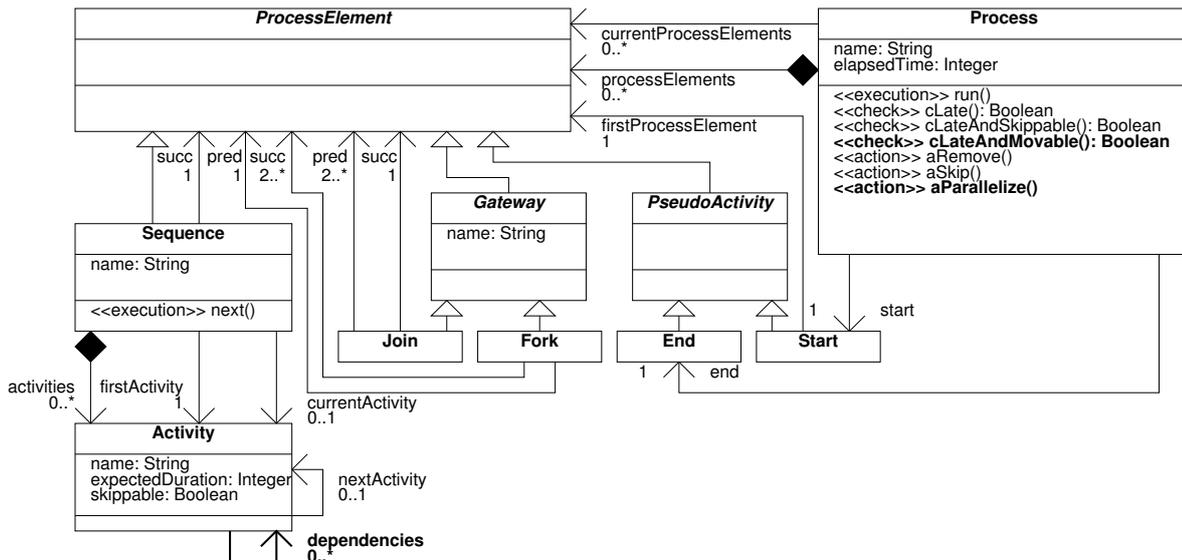


FIGURE 4.13 – L'*i-DSML* correspondant à la famille *DependAdaptPDL*

La figure 4.14 donne un exemple de cette adaptation. Dans la partie haute, il y a un modèle avant son

adaptation. Les flèches en pointillés représentent les dépendances entre activités. Ces dépendances ont été définies par l'ingénieur logiciel. Nous sommes actuellement en retard (350 unités de temps écoulées au lieu des 300 attendues) et quelques activités sont déplaçables (en tenant compte de leurs dépendances). Dans la partie basse, nous voyons ce modèle après adaptation. Par exemple, pour la séquence d'implémentation, comme *CoreTest* dépend de *CoreImpl*, ils doivent être membres de la même séquence. C'est pareil pour *UITest* et *UIImpl*. Cependant, les activités dont le nom est préfixé par « *Core* » et « *UI* » n'ont pas de dépendances entre elles. C'est pour cette raison que deux sous-séquences ont été créées.

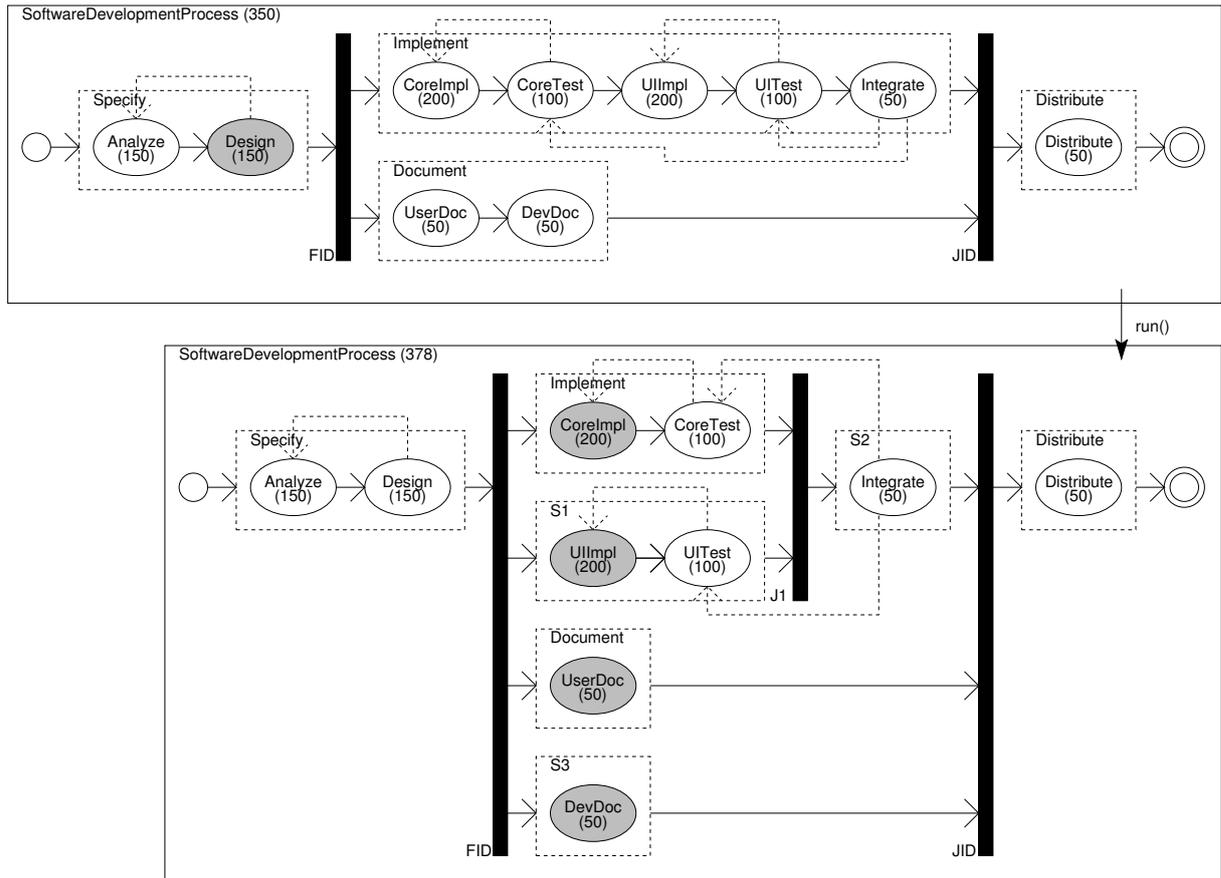


FIGURE 4.14 – Un modèle conforme au méta-modèle de *DependAdaptPDL*, avant et après adaptation

### 4.3.5 La description de la famille *DependSkipAdaptPDL*

Une autre adaptation intéressante pourrait être de reporter une activité (c'est-à-dire déplacer une activité vers la fin du processus) et ainsi l'exécuter seulement si le retard accumulé a été rattrapé depuis. Il est acceptable de reporter une activité si celle-ci est facultative (car, éventuellement, elle ne sera pas réalisée) mais elle ne sera pas déplacée plus loin qu'une activité qui en dépend, sous peine de dénaturer profondément l'objectif du processus. En conséquence, nous avons besoin à la fois de la notion d'activité facultative et de l'idée de dépendances entre ces activités. Pour être en possession de ces informations à l'exécution, nous construisons la famille *DependSkipAdaptPDL* qui hérite de deux super-familles : *SkipAdaptPDL* et *DependAdaptPDL*. Nous ajoutons cette cinquième famille à notre hiérarchie de familles dans

la figure 4.15.

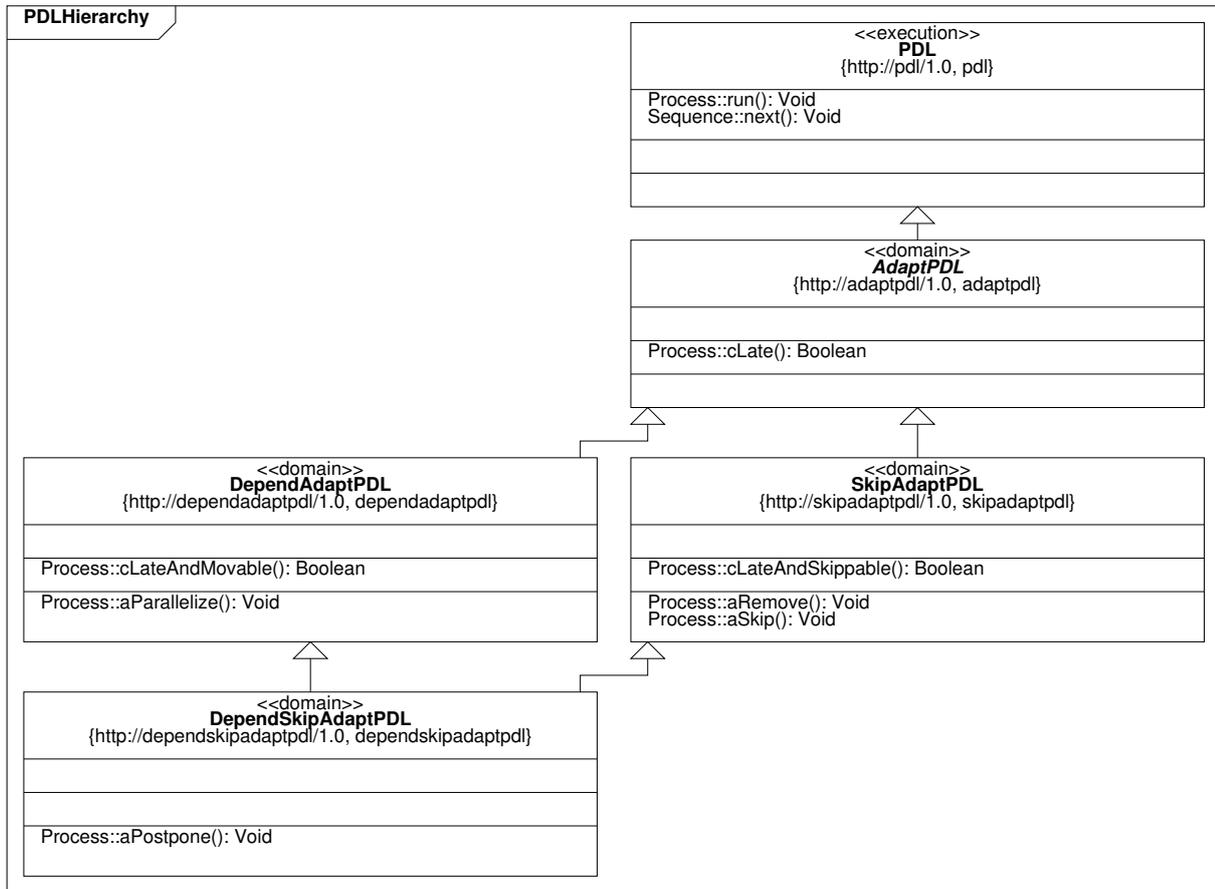


FIGURE 4.15 – Ajout de la famille *DependSkipAdaptPDL* à la hiérarchie de familles *PDLHierarchy*

Cette spécialisation signifie que les éléments disponibles dans l'*i-DSML* sont ceux qui correspondent aux *i-DSML* des deux super-familles. De cette manière, une vérification complexe peut être réalisée en combinant les vérifications d'adaptation `cLateAndSkippable()` et `cLateAndMovable()`. L'action d'adaptation qui correspond est de déplacer aussi loin que possible une activité vers la fin du processus. Cette action d'adaptation est réalisée avec l'opération `aPostpone()` ajoutée au méta-élément `Process` comme nous pouvons le voir dans la figure 4.16. Grâce à l'héritage multiple, nous sommes capables d'écrire une politique d'adaptation qui est basée sur la fusion de deux familles.

Dans le modèle conforme au méta-modèle de cet *i-DSML*, si nous sommes en retard comme sur la figure 4.17 où 550 unités de temps se sont écoulées au lieu des 500 attendues, alors une adaptation est nécessaire. Dans cet exemple, à l'intérieur de la séquence d'implémentation, la prochaine activité `CoreTest` est marquée comme facultative. Une fois l'activité courante `CoreImpl` terminée, l'activité `CoreTest` est reportée à plus tard dans le processus. Comme `Integrate` a pour dépendance `CoreTest`, nous ne pouvons pas la déplacer plus loin.

Jusqu'à maintenant, toutes les familles que nous avons vues étaient exclusivement au niveau domaine. Comme expliqué dans la sous-section 4.2.3, cela signifie que pour procéder à l'adaptation, il n'est pas nécessaire de savoir ce que les instances du méta-élément `Process` représentent du point de vue métier. Cela pourrait être par exemple un processus de développement logiciel ou tout aussi bien, une recette de

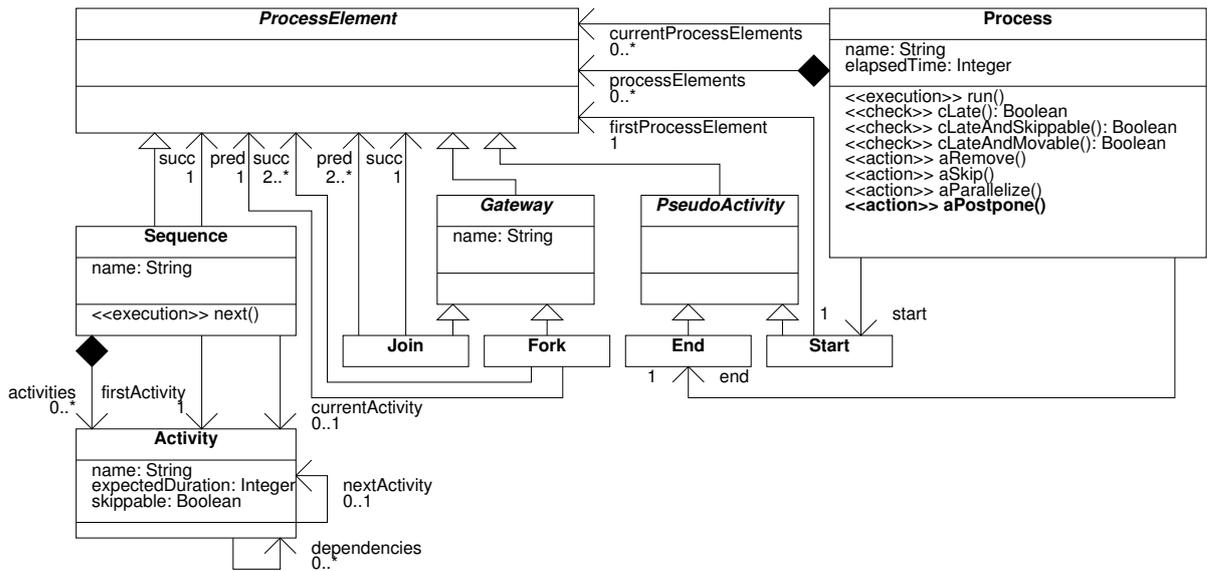


FIGURE 4.16 – L'*i-DSML* correspondant à la famille *DependSkipAdaptPDL*

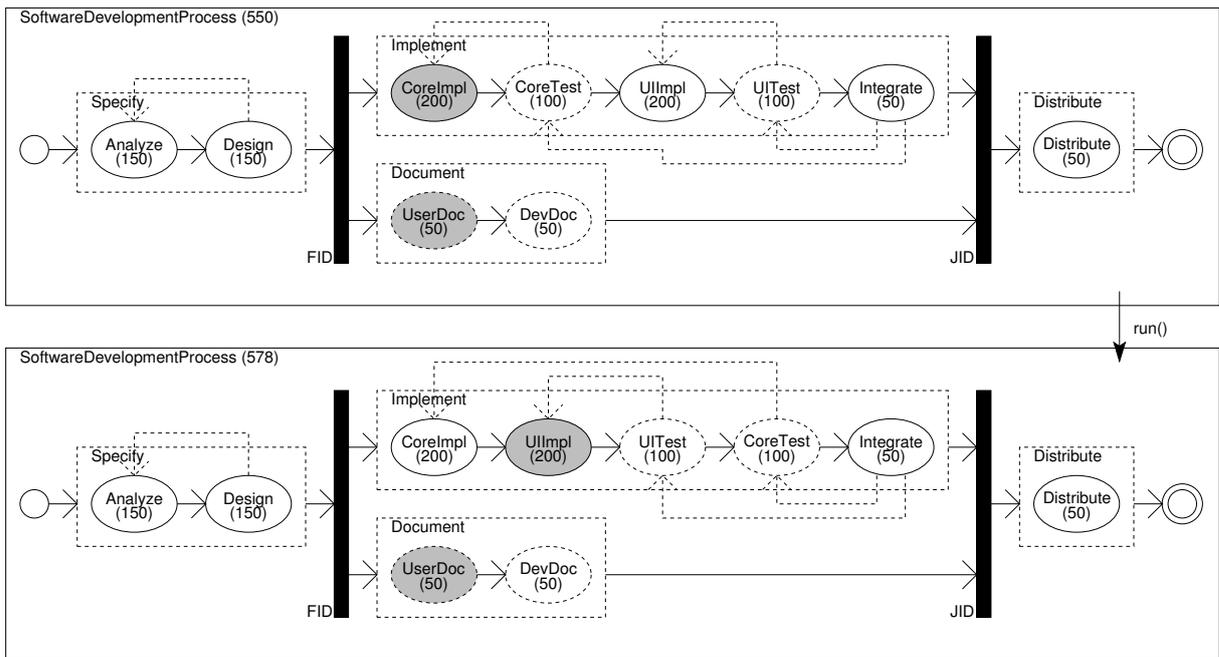


FIGURE 4.17 – Un modèle conforme au méta-modèle de *DependSkipAdaptPDL*, avant et après adaptation

cuisine ou encore un processus de fabrication sur une chaîne de production robotisée. Dans la prochaine sous-section, nous allons donner un exemple concret de famille au niveau métier.

### 4.3.6 La description de la famille *ManagedSkipAdaptPDL*

Durant le processus de développement logiciel, si nous sommes vraiment très en retard, une cause pourrait être l'incompétence du chef de projet. Être très en retard se mesure en calculant le nombre d'activités facultatives sautées, puis en le comparant à une valeur maximale qu'il ne faut pas dépasser.

Ainsi, nous avons besoin de la notion d'activité facultative disponible au sein de la famille *SkipAdaptPDL*. Ceci nous amène à définir la famille *ManagedSkipAdaptPDL* qui étend la famille *SkipAdaptPDL*. Nous ajoutons cette dernière famille à notre hiérarchie de familles dans la figure 4.18.

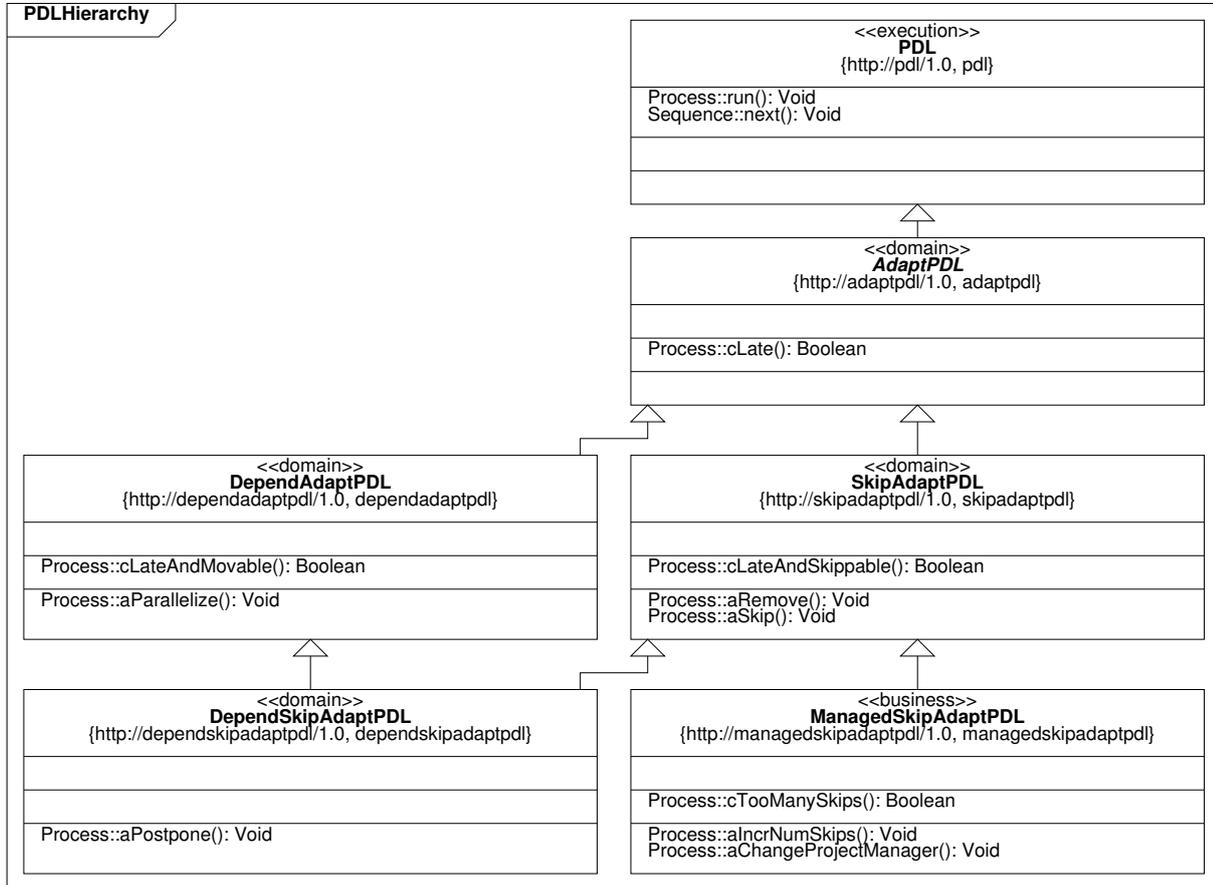


FIGURE 4.18 – Ajout de la famille *ManagedSkipAdaptPDL* à la hiérarchie de familles *PDLHierarchy*

Pour mesurer si nous sommes très en retard, nous avons besoin de connaître le nombre d'activités sautées ainsi que le nombre maximum de sauts autorisés. Afin de conserver ces deux informations au cours de l'exécution du modèle, nous avons ajouté respectivement les attributs entiers `numSkips` et `maxSkips` au méta-élément `Process` comme nous pouvons le constater sur la figure 4.19. Dans cette situation, une adaptation pourrait être de remplacer le chef de projet actuel par un nouveau. Grâce à ces attributs, en plus de savoir si nous sommes en retard et que la prochaine activité peut être évitée avec les vérifications d'adaptation `cLate()` et `cLateAndSkippable()`, il est dorénavant aussi possible de déterminer pour un nombre maximum donné de sauts autorisés (par exemple deux), si le nombre d'activités sautées a dépassé cette valeur. Cette vérification est réalisée par l'opération `cTooManySkips()` ajoutée au méta-élément `Process` comme nous pouvons l'observer sur la figure 4.19. Une première action d'adaptation incrémente la valeur de `numSkips` et est destinée à être appelée à chaque fois qu'une activité facultative est sautée. Une deuxième action d'adaptation crée une activité particulière dans le processus pour changer le chef de projet. Cette activité est ajoutée en parallèle avec les activités existantes du processus et est immédiatement activée. Cette deuxième action d'adaptation est, quant à elle, destinée à être appelée lorsque le nombre d'activités sautées a dépassé la valeur de `maxSkips`. Ces actions d'adaptation sont

respectivement réalisées par les opérations `aIncrNumSkip()` et `aChangeProjectManager()` ajoutées au méta-élément `Process` comme nous pouvons le voir dans la figure 4.19.

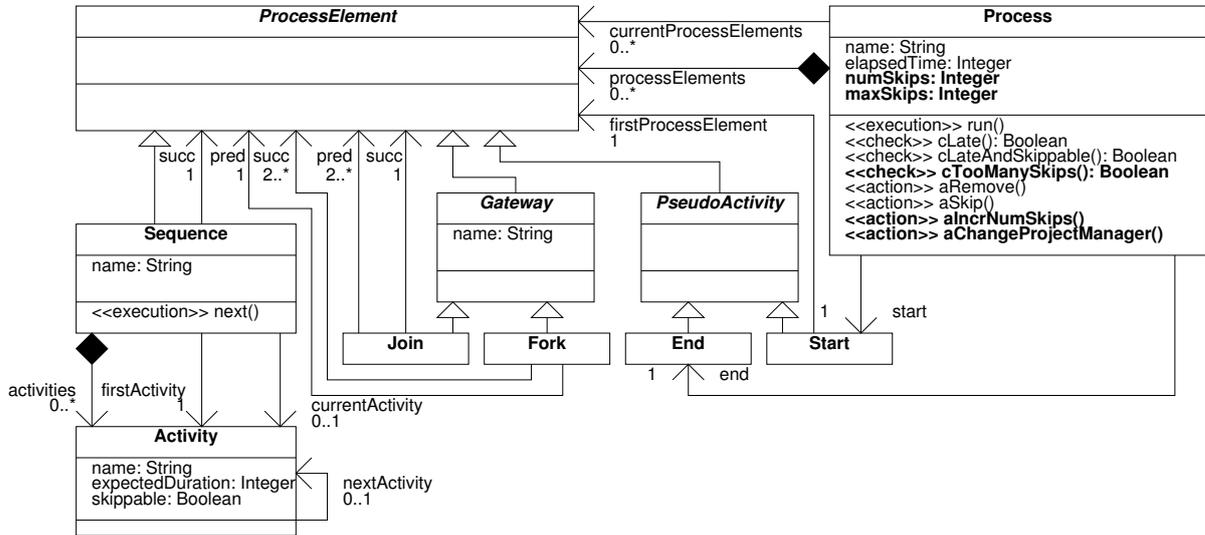


FIGURE 4.19 – L'*i-DSML* correspondant à la famille *ManagedSkipAdaptPDL*

Dans le modèle conforme au méta-modèle de cet *i-DSML*, nous avons maintenant les informations concernant le nombre d'activités sautées et le nombre maximum de sauts autorisés. Celles-ci sont indiquées dans le coin en haut à droite sur la figure 4.20. Comme nous pouvons le constater, nous sommes en retard (850 unités de temps écoulées au lieu des 800 attendues). Dans cet exemple, à l'intérieur de la séquence d'implémentation, la prochaine activité `UITest` est marquée comme facultative. Aussi, nous voyons qu'un saut a déjà été effectué et que la limite autorisée d'activités sautées est de deux. Une fois l'activité courante `UIImpl` terminée, l'activité `CoreTest` est sautée, ce qui nous amène à remplacer le chef de projet actuel par un nouveau. Ainsi, une nouvelle activité `ChangePM` (pour *Change Project Manager*) est créée dans le processus pour changer le chef de projet. Celle-ci est ajoutée en parallèle à l'activité sautée, ce qui implique la création d'une fourche `F1`, d'un raccord `J1` et d'un certain nombre de sous-séquences.

Cette famille est au niveau métier parce que nous sommes informés que l'instance du méta-élément `Process` représentera un processus de projet dans lequel nous trouvons un chef de projet. Une telle activité n'a aucun sens pour beaucoup de processus définis avec l'*i-DSML PDL*, comme par exemple, un processus pour une recette de cuisine où la notion métier de chef de projet est absente.

## 4.4 Conclusion

Dans ce chapitre, nous avons proposé le concept des familles qui facilite la définition d'*i-DSML* adaptables. Le concept de famille a pour objectif de mettre en ordre proprement un certain nombre d'éléments de différentes natures qui sont utiles à la définition d'un *i-DSML* adaptable. Une famille rassemble le méta-modèle donné d'un *i-DSML* avec les opérations dédiées à son exécution et à son adaptation que nous nommons « attributs ». Nous avons montré, par l'exemple, que spécialiser successivement le méta-modèle d'un *i-DSML* ouvre la voie à la définition de nouvelles politiques d'adaptation. Pour cette raison, les familles peuvent hériter les unes des autres, nous permettant ainsi de définir des hiérarchies de familles,

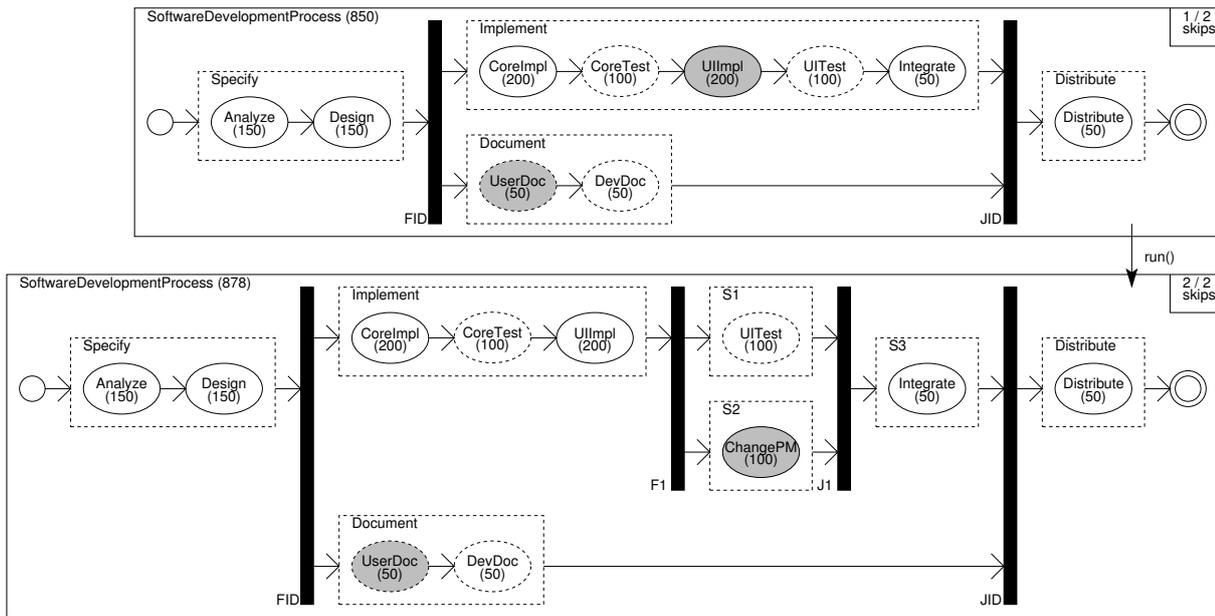


FIGURE 4.20 – Un modèle conforme au méta-modèle de *ManagedSkipAdaptPDL*, avant et après adaptation

en partant de la plus générale pour arriver à la plus spécifique. L'héritage offre conceptuellement les mêmes avantages qu'en programmation orientée objet tels que la réutilisation d'attributs existants ou la factorisation de mêmes attributs au travers d'une super-famille commune. Comme nous étendons une famille que par ajout, tout attribut hérité sera fonctionnel pour une sous-famille. En conséquence, une politique d'adaptation étant une combinaison de ces attributs, nous pouvons soit la réutiliser dans une sous-famille, soit nous en servir de base pour définir de nouvelles politiques. Une famille peut être définie à un niveau domaine ou métier selon qu'elle est basée sur un contenu métier particulier ou non. La spécialisation est le déclencheur pour l'émergence d'adaptations domaine. En effet, il est souvent nécessaire de contraindre le contenu d'un modèle pour être en capacité de le modifier sans même connaître son contenu métier. Sans cela, nous nous contenterons d'adaptations au niveau métier qui *de facto* sont plus difficilement réutilisables. Nous avons appliqué cette approche sur l'exemple concret d'un modèle de processus où plusieurs familles ont été construites grâce à la spécialisation de familles.

Le moteur exécutant et adaptant un modèle doit, pour l'instant, contenir des politiques d'adaptation codées « en dur ». En effet, les opérations d'exécution, les vérifications d'adaptation et les actions d'adaptation sont orchestrées et combinées à travers le code du développeur à l'intérieur du moteur d'exécution. Cependant, il peut être utile de modifier cette orchestration durant l'exécution du modèle. Si une famille offre plusieurs actions d'adaptation, l'une pourrait être plus appropriée que l'autre, selon la situation courante. Pour parvenir à ce résultat, nous proposons dans le chapitre suivant de définir un *i-DSML* dédié à l'orchestration des opérations disponibles pour une famille. Un modèle d'orchestration sera interprété par son propre moteur d'exécution en plus du modèle exécuté. Concrètement, ce modèle va définir une sémantique d'adaptation (c'est-à-dire des combinaisons de vérifications d'adaptation, d'actions d'adaptation et d'opérations d'exécution).

## Chapitre 5

# Orchestration de l'adaptation des *i-DSML*

Nous avons vu dans le chapitre précédent que les *i-DSML* adaptables associés à des familles peuvent aider l'ingénieur logiciel à construire ses applications adaptatives avec une solution basée à 100% sur les modèles. En l'état, les opérations contenues dans ces familles sont orchestrées et combinées librement afin de construire des politiques d'adaptation. Si ces politiques d'adaptation sont codées « en dur » à l'intérieur du moteur d'adaptation, alors il sera techniquement nécessaire d'intervenir sur le code du moteur lui-même pour venir les modifier. Cette façon de procéder va à l'encontre d'une vision IDM puisque cela incite l'ingénieur logiciel à programmer au lieu de modéliser. Ainsi, pour rester en accord avec les idées présentées jusqu'ici, nous allons continuer à donner une place plus importante aux modèles qu'au code, en proposant une approche qui consiste à s'appuyer sur un *i-DSML* d'orchestration.

Dans ce chapitre, nous allons d'abord présenter l'architecture d'une application adaptative « type » au sens des idées proposées dans ce mémoire. Ensuite, nous définirons le méta-modèle de notre *i-DSML* d'orchestration de l'adaptation. Celui-ci constitue la syntaxe abstraite de notre langage et permet d'identifier les différents concepts liés à une orchestration de l'adaptation. Puis nous proposerons une syntaxe concrète pour ce langage afin d'être en mesure de représenter un modèle d'orchestration de l'adaptation sous la forme d'un diagramme avec une notation précise et intuitive. Dans la section 5.4, nous montrerons des exemples d'utilisation de ce langage en proposant une orchestration de l'adaptation pour les *i-DSML* adaptables de la hiérarchie de familles *PDLHierarchy* introduite dans le chapitre précédent.

### 5.1 Une architecture pour l'adaptation d'*i-DSML*

Considérons à nouveau l'exemple de *SkipAdaptPDL* présenté en sous-section 4.3.3. Le code du moteur réalisant l'exécution seule appelle une opération d'exécution `run` qui permet d'exécuter le processus et ainsi de passer à la prochaine activité. Pour mettre en œuvre une adaptation, nous pouvons masquer cette opération d'exécution par redéfinition (*override*). Ainsi, nous avons la possibilité d'ajouter des appels aux vérifications et actions d'adaptation (cf. lignes 3 et 4 du listing 5.1). Ici, l'idée est que tant que nous sommes en retard dans le processus et que la prochaine activité est facultative, nous supprimons cette activité. Cette adaptation est codée « en dur » dans le moteur d'adaptation et par conséquent il serait

plus intéressant de la réifier dans un modèle. Cependant, cela nécessite l'utilisation d'une architecture particulière pour le logiciel adaptatif.

```
1 @Override public void run()
2 {
3     while(this.cLateAndSkippable())
4         this.aRemove();
5     super.run();
6 }
```

Listing 5.1 – L'implémentation « en dur » d'une politique d'adaptation dans le moteur d'adaptation

La figure 5.1 présente une architecture pour l'adaptation d'*i-DSML*. Nous pouvons constater que deux moteurs coexistent : le moteur d'exécution d'une part et le moteur d'orchestration d'autre part. Le moteur d'exécution est autonome, ce qui signifie qu'il est tout à fait capable à lui seul d'exécuter un modèle adaptable. En effet, comme vu précédemment, il embarque la sémantique d'exécution de l'*i-DSML* adaptable nécessaire à l'exécution de ses modèles. Ce moteur prend en entrée un modèle conforme au méta-modèle d'un *i-DSML* adaptable et constitue le point d'entrée de notre logiciel adaptatif. Bien qu'il contienne également les opérations d'adaptation (c'est-à-dire les vérifications et actions d'adaptation), l'orchestration de celles-ci est une tâche réservée au second moteur de l'architecture proposée.

Le moteur d'orchestration, quant à lui, vient compléter la sémantique d'adaptation du moteur d'exécution. En effet, le moteur d'orchestration est capable d'interpréter la sémantique d'adaptation de l'*i-DSML* adaptable qui est séparée en deux parties distinctes. Une première partie de cette sémantique est stockée sous forme opérationnelle (dans le moteur d'exécution) tandis que le reste est stocké sous forme d'une combinaison d'opérations (dans le modèle d'orchestration de l'adaptation). Ce moteur prend en entrée un modèle conforme au méta-modèle d'un *i-DSML* d'orchestration de l'adaptation et est invoqué par le moteur d'exécution lorsqu'une opération d'exécution doit être adaptée. La figure 5.1 illustre cette délégation (1) effectuée par le moteur d'exécution qui confie au moteur d'orchestration le soin d'adapter ladite opération. Le moteur d'orchestration se met alors à chercher (2) dans le modèle d'orchestration chargé en entrée quelles politiques d'adaptation correspondent à cette opération. Dès qu'une ou plusieurs politiques d'adaptation sont trouvées, le moteur d'orchestration les exécute séquentiellement et suit l'ordre dans lequel sont combinées les opérations en invoquant (3) les opérations de rappel (*callback*) correspondantes qui se situent dans le moteur d'exécution. Cette façon de procéder correspond en fait à une inversion de contrôle puisqu'au lieu de laisser le moteur d'exécution gérer lui-même l'adaptation de son modèle adaptable, il passe le relais au moteur d'orchestration qui va s'occuper de cette tâche et intervenir sur le flot d'exécution du programme adaptatif.

Un premier avantage de cette séparation de l'application adaptative en deux moteurs est d'offrir une semi-généricité de celle-ci. En effet, autant le moteur d'exécution est spécifique à l'*i-DSML* adaptable dont le modèle est à adapter, autant le moteur d'orchestration reste inchangé. Ainsi, pour orchestrer l'adaptation d'un autre modèle adaptable, seul le moteur d'exécution doit être remplacé et nous pouvons nous servir de nouveau du même moteur d'orchestration.

Un deuxième avantage de cette séparation est de respecter le principe de la séparation des préoccupations. En effet, une première brique constituée du moteur d'exécution et de son modèle traite tout ce qui relève de l'exécution de l'application adaptative, tandis qu'une seconde brique constituée du moteur d'orchestration et de son modèle gère tout ce qui concerne l'adaptation du logiciel adaptatif. Ainsi, pour

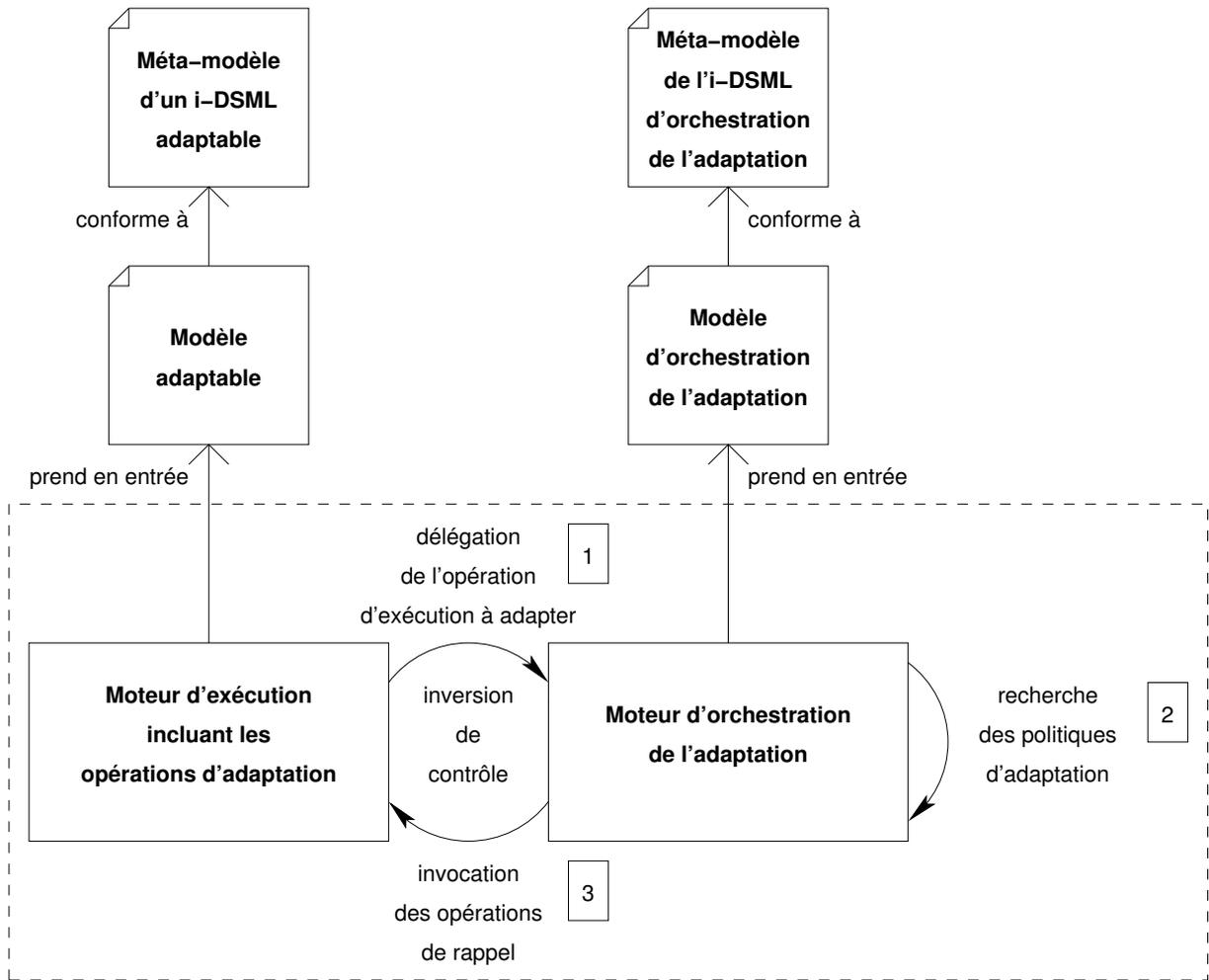


FIGURE 5.1 – Une architecture pour l'adaptation d'*i-DSML*

effectuer une opération de maintenance sur le programme adaptatif, il devient plus simple de déterminer quelle brique nous intéresse grâce à cette séparation en modules.

Même si l'application adaptative est composée de deux moteurs, qui ont chacun un rôle spécifique à jouer et qui prennent chacun un modèle en entrée, d'un point de vue général (c'est-à-dire sans se préoccuper de l'architecture interne du logiciel adaptatif), il est possible de considérer le programme adaptatif comme un seul et unique moteur prenant deux modèles en entrée. Concernant le premier modèle, celui qu'il faut adapter, nous avons déjà vu des exemples dans les chapitres précédents (par exemple un modèle conforme au méta-modèle de *SkipAdaptPDL*). Par contre, pour ce qui est du second modèle (celui d'orchestration de l'adaptation), aucun exemple n'a été montré jusqu'ici. Dans les sections suivantes, nous allons donc expliquer comment construire un tel modèle d'orchestration de l'adaptation à l'aide d'un *i-DSML* d'orchestration de l'adaptation.

## 5.2 Le méta-modèle d'orchestration de l'adaptation *ASDL*

La section précédente a présenté l'architecture de l'application adaptative et a montré que l'un des modèles pris en entrée par celle-ci est le modèle d'orchestration de l'adaptation construit grâce à un *i-DSML* d'orchestration de l'adaptation. Dans cette section, nous allons nous intéresser plus particulièrement au méta-modèle de cet *i-DSML* représenté par la figure 5.2. Celui-ci est nommé *Adaptation Semantics Description Language (ASDL)* car définir une orchestration de l'adaptation revient finalement à décrire la sémantique d'adaptation (c'est-à-dire le comportement adaptatif) de l'application adaptative à l'aide d'un langage dédié. Comme tout méta-modèle d'*i-DSML*, celui-ci est constitué d'une partie structurelle statique et d'une partie structurelle dynamique. Dans la partie statique, nous trouvons d'abord le méta-élément racine de notre méta-modèle : **AdaptationSemantics**. Il représente globalement une orchestration de l'adaptation et il est donc composé d'un certain nombre de politiques d'adaptation. Ces dernières, représentées par la méta-classe **AdaptationPolicy**, sont constituées logiquement d'un début, d'une fin et d'un ensemble de pas d'orchestration à réaliser les uns après les autres. Le début et la fin sont représentés respectivement par les méta-classes **Start** et **End**. La première indique le pas d'orchestration par lequel nous commençons la politique d'adaptation grâce au lien **firstStep** tandis que la deuxième spécifie le dernier pas d'orchestration de la politique d'adaptation à l'aide de la référence **lastStep**. Pour mettre en évidence l'opération adaptée par la politique d'adaptation, un lien **adaptedOperation** part de la méta-classe **AdaptationPolicy**. Concernant les pas d'orchestration, ils sont de deux types et apparaissent chacun comme une spécialisation de la méta-classe abstraite **Step**. Nous avons d'un côté les pas d'orchestration simples (**SimpleStep**) qui ne font qu'indiquer leur prochain pas via la référence **nextStep**, de l'autre les pas d'orchestration qui requièrent une prise de décision (**DecisionStep**) et qui désignent les deux prochains pas possibles via les références **ifFalseStep** et **ifTrueStep**. Chaque pas d'orchestration contient une opération qui peut être de trois sortes comme le montre la spécialisation de la méta-classe abstraite **Operation**. Nous avons les opérations d'exécution contenant la sémantique d'exécution de notre logiciel adaptatif, puis nous trouvons les vérifications et actions d'adaptation contenant la sémantique d'adaptation. Elles sont respectivement représentées par les méta-classes **ExecutionOperation**, **CheckOperation** et **ActionOperation** qui possèdent chacune un lien d'héritage vers la méta-classe abstraite **Operation**.

Nous pouvons remarquer qu'il existe une correspondance entre les types de pas d'orchestration et les sortes d'opérations. Tandis qu'un pas d'orchestration de type simple contient forcément une opération d'exécution ou une action d'adaptation puisque celles-ci n'ont qu'une seule opération suivante, un pas d'orchestration avec décision, quant à lui, contient obligatoirement une vérification d'adaptation puisque cette dernière offre un choix parmi deux opérations. En effet, l'opération suivante varie selon que la vérification d'adaptation répond « vrai » ou « faux ». Le listing 5.2 donne ces deux contraintes sous la forme d'invariants *OCL*.

```

1 context SimpleStep inv simpleStepOperationType:
2   self.operation.oclIsTypeOf(ExecutionOperation) or self.operation.oclIsTypeOf(
      ActionOperation)
3 context DecisionStep inv decisionStepOperationType:
4   self.operation.oclIsTypeOf(CheckOperation)

```

Listing 5.2 – Des invariants *OCL* pour *ASDL*

L'ingénieur logiciel peut placer une opération dans n'importe quelle méta-classe du méta-modèle

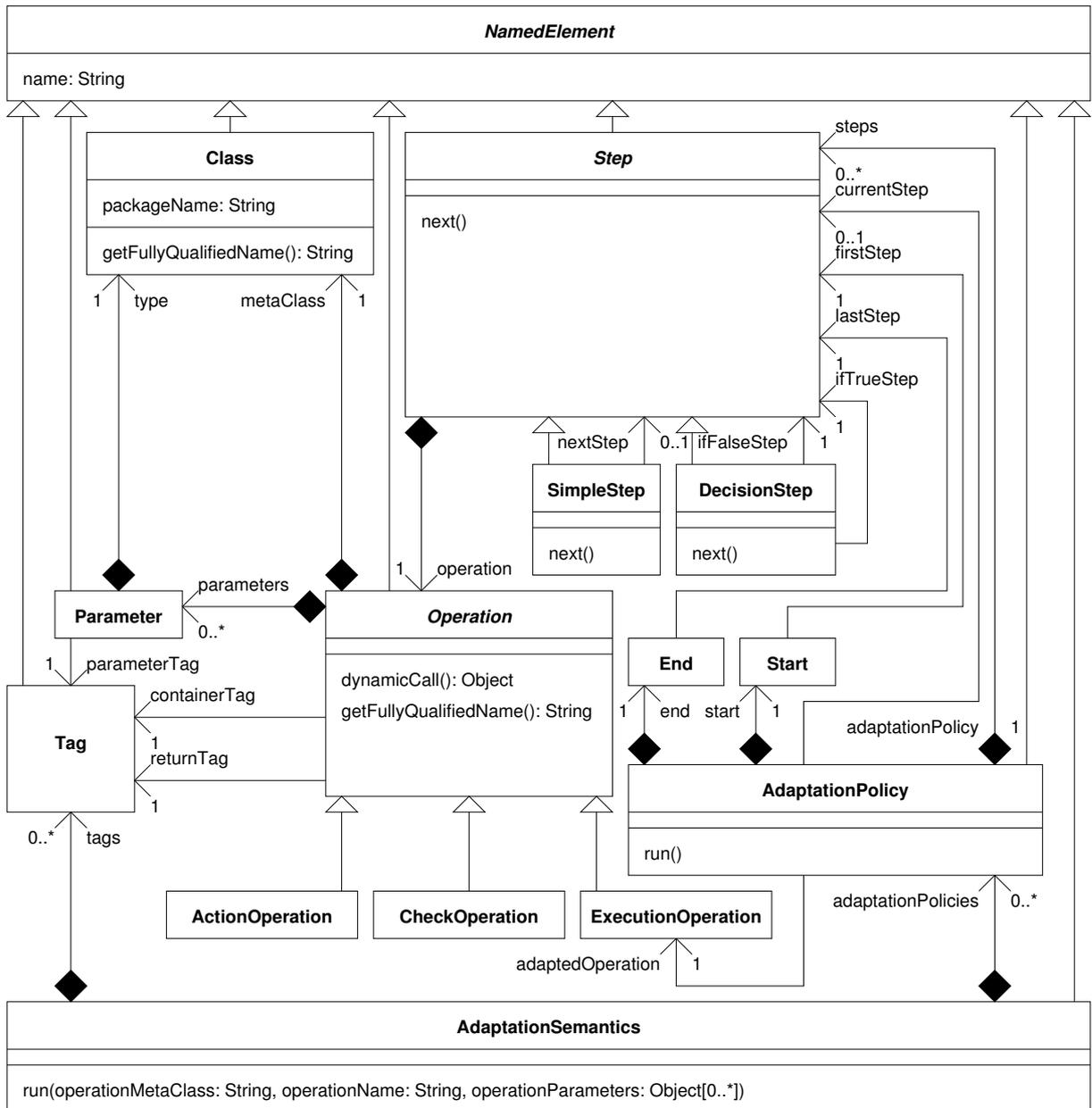


FIGURE 5.2 – Le méta-modèle d’orchestration de l’adaptation *ASDL*

de son *i-DSML* adaptable. En conséquence, pour localiser précisément l’emplacement d’une opération, nous donnons des informations sur la méta-classe dans laquelle elle se situe. La méta-classe abstraite **Operation** possède donc logiquement un lien vers un méta-élément **Class** qui permet d’indiquer le nom de la méta-classe et du paquetage contenant l’opération. Ce méta-élément **Class** possède une opération `getFullyQualifiedName` permettant de retourner le nom complet de la méta-classe. Ce dernier correspond simplement au résultat de la concaténation du nom du paquetage, d’un point et du nom de la méta-classe.

Pour distinguer clairement les opérations les unes des autres, en plus de leur nom, nous avons besoin d’informations concernant leurs paramètres. Ainsi, le nom et les paramètres permettent de posséder la

signature complète des opérations et lorsqu'il s'agira d'invoquer une opération au cours de l'exécution du modèle, il n'y aura aucun risque d'ambiguïté sur celle qu'il faudra choisir. La méta-classe abstraite `Operation` possède donc logiquement un lien vers un méta-élément `Parameter` qui permet d'indiquer pour chaque paramètre son nom et son type. Le type est représenté lui aussi à l'aide du méta-élément `Class` puisqu'une classe peut potentiellement représenter un type (d'où la réutilisation de ce méta-élément au sein du méta-modèle de notre *i-DSML ASDL*).

Une opération nommée contient donc des informations concernant sa méta-classe et ses paramètres. En rassemblant toutes ces données, nous pouvons obtenir un identifiant unique pour chaque opération. C'est le rôle de l'opération `getFullyQualifiedName` appartenant à la méta-classe abstraite `Operation` qui va retourner le résultat de la concaténation du nom complet de la méta-classe, d'une parenthèse ouvrante, d'un ensemble éventuel de types de paramètres séparés par des virgules et d'une parenthèse fermante.

Lors de l'exécution du logiciel adaptatif, les opérations sont contenues par un objet qui est l'instance d'une classe. Ces opérations ont des valeurs qui sont passées en paramètre et qui sont également des objets enregistrés dans la mémoire du système. Ces opérations retournent une valeur qui est aussi un objet stocké à une adresse particulière en mémoire. Tous ces objets constituent les données gérées par l'application adaptative et nous avons besoin d'une référence vers chacun d'eux puisque c'est dorénavant nous qui contrôlons le flot d'exécution du programme. Pour cela, nous utilisons des étiquettes que nous attachons à ces objets afin de facilement les retrouver et les manipuler lors de l'exécution. Ces étiquettes sont représentées par la méta-classe `Tag` et permettent de retrouver les objets contenant les opérations grâce à la référence `containerTag`, les objets correspondant aux valeurs des paramètres de ces opérations via le lien `parameterTag` et les objets relatifs aux valeurs de retour de ces opérations à l'aide de la référence `returnTag`.

Les méta-éléments `AdaptationSemantics`, `AdaptationPolicy`, `Step`, `Operation`, `Class`, `Parameter` et `Tag` présentés précédemment ont tous besoin d'un nom pour identifier leurs instances. Pour cette raison, chacun possède un lien d'héritage vers une méta-classe abstraite `NamedElement` qui possède un attribut `name` de type chaîne de caractères. Comme pour ce qui a été fait avec le méta-modèle *FHDL*, dans la sous-section 4.2.4 du chapitre précédent, cette façon de procéder est reprise du méta-modèle *UML* qui possède un méta-élément tout à fait similaire.

La partie dynamique de notre méta-modèle, quant à elle, est beaucoup moins imposante que sa partie statique. En effet, elle contient uniquement les méta-éléments susceptibles d'évoluer au cours de l'exécution du modèle. Or, nous avons un seul méta-élément qui va changer dans le temps : la référence `currentStep` qui indique le pas d'orchestration courant et donc l'opération courante (c'est-à-dire celle en cours d'exécution). À chaque fois que le pas d'orchestration courant est mis à jour, la partie dynamique de l'*i-DSML ASDL* est modifiée. Les pas d'orchestration sont réalisés par le moteur d'orchestration de l'application adaptative et l'ensemble de ces pas, pour une opération adaptée, constitue la trace d'orchestration pour cette opération.

En plus de ces parties structurelles, le méta-modèle de l'*i-DSML ASDL* est associé à une sémantique d'exécution. Elle décrit le comportement durant l'exécution du modèle. Ici, la sémantique d'exécution est embarquée dans l'opération `run` des méta-classes `AdaptationSemantics` et `AdaptationPolicy`, dans l'opération `next` de la méta-classe abstraite `Step` et de ses sous-méta-classes ainsi que dans l'opération `dynamicCall` de la méta-classe abstraite `Operation`. Le rôle de l'opération `run`, dans la méta-classe `AdaptationSemantics`, est d'orchestrer l'adaptation (c'est-à-dire elle doit choisir les politiques d'adapta-

tion, puis les jouer). Pour sélectionner les bonnes politiques d'adaptation, cette opération analyse la valeur de ses paramètres `operationMetaClass`, `operationName` et `operationParameters` qui contiennent respectivement le nom complet de la méta-classe, le nom de l'opération à adapter et l'ensemble de ses paramètres. Une fois qu'elle a déterminé ces politiques d'adaptation, il ne lui reste plus qu'à appeler l'opération `run` sur celles-ci. Le rôle de l'opération `run`, dans la méta-classe `AdaptationPolicy`, est de réaliser dans le bon ordre les différents pas d'orchestration afin d'appliquer une politique d'adaptation. Les pas d'orchestration sont réalisés les uns à la suite des autres, en commençant par celui de début, jusqu'à ce que le pas d'orchestration courant soit le pas d'orchestration final. Pour obtenir le pas d'orchestration suivant, cette opération `run` appelle l'opération `next` de la méta-classe abstraite `Step` et de ses sous méta-classes qui doit déterminer quel est le pas d'orchestration successeur. Cette opération `next` appelle l'opération `dynamicCall` de la méta-classe abstraite `Operation` afin d'invoquer dynamiquement l'opération associée au pas d'orchestration courant. Lorsque nous avons un pas d'orchestration courant qui requière une décision, celle-ci est prise en fonction de la valeur de retour booléenne de cette opération `dynamicCall`. En revanche, lorsque nous avons un pas d'orchestration courant simple, la valeur de retour n'a aucune incidence puisque le pas d'orchestration successeur est déjà déterminé. Comme le comportement de cette opération `next` est différent selon le type de pas, cette opération est redéfinie dans les sous-méta-classes de la méta-classe abstraite `Step`. Une fois que ce pas d'orchestration successeur a été trouvé, celui-ci prend la place du pas d'orchestration courant et ainsi de suite (c'est-à-dire le méta-élément `currentStep` est modifié).

Nous avons présenté dans cette section le méta-modèle de l'*i-DSML ASDL* (cf. figure 5.2) qui constitue sa syntaxe abstraite. Cette syntaxe représentant la structure de notre langage permet de définir une orchestration de l'adaptation, mais reste insuffisante pour pouvoir la visualiser graphiquement. La section suivante vient combler ce manque en indiquant cette fois la syntaxe concrète de l'*i-DSML ASDL*. Grâce à cette syntaxe complémentaire, qui donne une notation pour chaque élément structurel de ce langage, nous serons en mesure de représenter une orchestration de l'adaptation.

### 5.3 La représentation d'une orchestration de l'adaptation

La section précédente a présenté la syntaxe abstraite de notre *i-DSML ASDL*. Dans cette section nous allons décrire sa syntaxe concrète pour permettre de représenter n'importe quelle orchestration de l'adaptation à l'aide d'une même notation. La figure 5.3 montre la définition d'une orchestration de l'adaptation quelconque, basée sur cette syntaxe concrète, permettant d'illustrer les propos de cette section.

Comme nous pouvons le constater, une orchestration de l'adaptation est décrite graphiquement à l'aide d'une boîte. Celle-ci possède un petit rectangle dans le coin supérieur gauche dont l'angle inférieur droit est biseauté. Il est destiné à accueillir le nom de l'orchestration de l'adaptation qui doit être écrit en gras et aligné à gauche. Cette boîte est la plus grande du diagramme car elle contient l'ensemble des politiques d'adaptation associé à cette orchestration de l'adaptation.

Les autres boîtes, contenues à l'intérieur de la plus grande, indiquent donc les politiques d'adaptation. Tout comme la boîte représentant l'orchestration de l'adaptation, celles-ci possèdent un petit rectangle similaire dans le coin supérieur gauche mais cette fois, il est destiné à présenter trois informations alignées à gauche et séparées sur trois lignes. La première ligne contient le nom de la politique d'adaptation qui est écrit en gras. Les deux lignes suivantes contiennent des informations permettant d'identifier l'opération

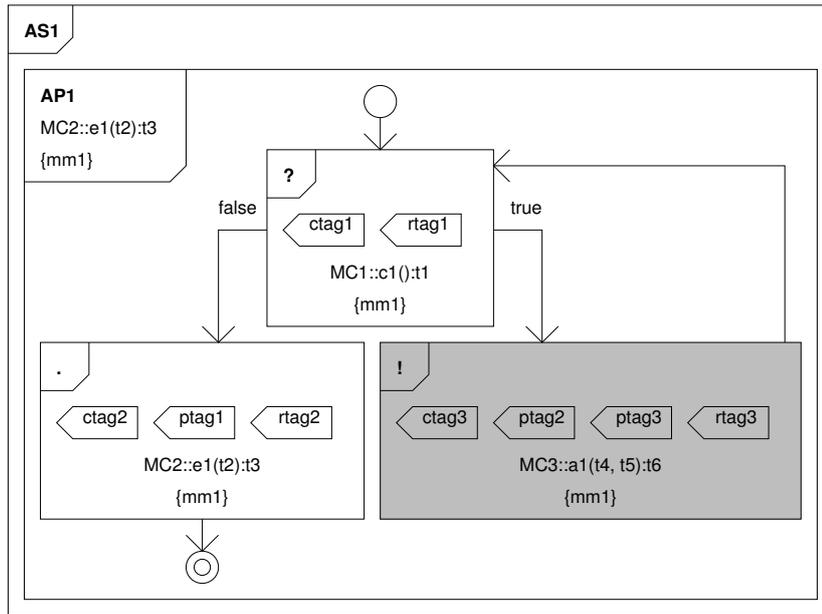


FIGURE 5.3 – Un exemple généraliste d’une orchestration de l’adaptation

d’exécution qui est adaptée par la politique d’adaptation. Ainsi, nous présentons sur la deuxième ligne la signature de l’opération d’exécution à adapter. Nous commençons par indiquer le nom de sa méta-classe, puis son nom. Le séparateur utilisé pour délimiter ces deux informations est le double deux-points (::). Pour compléter la signature, le type des paramètres de l’opération est noté entre parenthèses juste après. Ils peuvent éventuellement être précédés d’un nom, ce qui permettra de mieux identifier le rôle du paramètre. Dans ce cas facultatif, il faudra alors utiliser le deux-points comme séparateur entre le type du paramètre et le nom de ce dernier. Si plusieurs types sont à préciser, alors ils seront séparés par une virgule. Au-dessous de la signature de l’opération, sur la troisième ligne, nous indiquons entre accolades le nom du paquetage contenant la méta-classe de l’opération. Cette notation entre accolades est inspirée du langage *UML* dans lequel les méta-éléments peuvent être associés à des valeurs marquées représentées de la même façon.

À l’intérieur de chaque politique d’adaptation, nous trouvons les éléments graphiques permettant d’indiquer le début et la fin d’une orchestration. Le début est signalé à l’aide d’un cercle tandis que la fin est indiquée par un double cercle (l’un contenant l’autre). Cette notation est directement inspirée de langages tels qu’*UML*, dans lequel les diagrammes d’activités et de machines à états ont un symbole similaire pour identifier les nœuds ou états initiaux et finaux, ou encore *BPMN*, dans lequel des cercles sont utilisés dans les processus pour identifier les événements de début et de fin.

Les opérations, quant à elles, sont représentées par des boîtes contenues au sein d’une politique d’adaptation. Comme pour la boîte de l’orchestration de l’adaptation et celles des politiques d’adaptation, celles-ci possèdent un petit rectangle identique dans le coin supérieur gauche. Toutefois, celui-ci est destiné à un tout autre usage que celui de donner le nom de l’opération puisqu’il permet d’indiquer la catégorie d’opération. Ainsi, lorsque nous avons une opération d’exécution, nous devons y inscrire un point (« . »), lorsqu’il s’agit d’une vérification d’adaptation, nous devons y inscrire un point d’interrogation (« ? ») et lorsqu’il s’agit d’une action d’adaptation, nous devons y inscrire un point d’exclamation (« ! »). L’opération qui est coloriée en gris est celle qui est dynamiquement invoquée et correspond au pas

d'orchestration courant.

Au centre de chaque opération, une première ligne est réservée à la présentation des étiquettes. Chaque étiquette est représentée par un pentagone possédant deux angles droits consécutifs. Elles sont placées horizontalement dans cet ordre : l'étiquette pour l'objet qui contient l'opération, celles pour les objets correspondant aux valeurs des paramètres de l'opération et celle pour l'objet correspondant à la valeur de retour de l'opération. Le nom des étiquettes est aligné au centre des pentagones. Ces étiquettes ont une portée globale sur l'orchestration de l'adaptation, ce qui signifie que si deux étiquettes portent le même nom, elles référencent le même objet. La deuxième et la troisième ligne, au centre d'une opération, sont réservées respectivement à la présentation de la signature de cette opération et au nom du paquetage contenant la méta-classe de l'opération. La notation est identique à celle utilisée pour indiquer l'opération adaptée dans une politique d'adaptation sauf que cette fois, les informations doivent être alignées au centre.

Les flèches, dans les politiques d'adaptation, ont un sens différent en fonction de l'élément graphique d'où elles partent ou arrivent. Si une flèche part d'un début de politique d'adaptation, alors elle indique le pas d'orchestration par lequel nous commençons. Lorsqu'une flèche part d'une opération d'exécution ou d'une action d'adaptation, elle montre le pas d'orchestration suivant. Quand une flèche part d'une vérification d'adaptation, alors elle peut désigner soit le pas d'orchestration suivant dans le cas où la vérification retourne « vrai », soit le pas d'orchestration suivant dans le cas où la vérification retourne « faux ». Pour déterminer si nous sommes dans le premier ou dans le second cas, il suffit de regarder l'indication sur les flèches qui peut être soit « *true* » (pour « vrai »), soit « *false* » (pour « faux »). Enfin, si une flèche arrive sur une fin de politique d'adaptation, alors elle signale que nous sommes sur le dernier pas d'orchestration.

L'orchestration de l'adaptation sur la figure 5.3 est nommée **AS1**. Elle est composée d'une seule politique d'adaptation **AP1**. Cette politique d'adaptation définit les trois opérations **MC1::c1():t1**, **MC2::-e1(t2):t3** (l'opération adaptée) et **MC3::a1(t4, t5):t6**. Cette première opération n'a aucun paramètre, la seconde en possède un (**t2**) et la dernière en a deux (**t4** et **t5**). Au niveau des étiquettes, **ctag1** est associée à l'objet qui contient l'opération **MC1::c1():t1**, **ctag2** à celui qui contient **MC2::-e1(t2):t3** et **ctag3** à celui qui contient **MC3::a1(t4, t5):t6**. Les étiquettes **ptag1**, **ptag2** et **ptag3** sont associées respectivement aux objets correspondant aux valeurs des paramètres **t2**, **t4** et **t5** tandis que les étiquettes **rtag1**, **rtag2** et **rtag3**, quant à elles, sont associées respectivement aux objets correspondant aux valeurs de retour **t1**, **t3** et **t6**. Toutes ces opérations appartiennent à des méta-classes qui sont contenues dans le même paquetage : **mm1**. Cette politique d'adaptation commence par effectuer une vérification d'adaptation (**MC1::c1():t1**). Si l'opération répond « vrai » alors une action d'adaptation (**MC3::a1(t4, t5):t6**) sera effectuée, puis la vérification sera de nouveau réalisée et ainsi de suite. Au contraire, si l'opération répond « faux » alors une opération d'exécution (**MC2::-e1(t2):t3**) sera appelée car aucune adaptation n'est nécessaire dans ce cas. Nous pouvons remarquer que le diagramme indique que le pas d'orchestration courant est celui de l'action d'adaptation **MC3::a1(t4, t5):t6**. Nous pouvons facilement en déduire qu'au prochain pas d'orchestration, la vérification d'adaptation **MC1::c1():t1** sera l'opération du pas d'orchestration courant.

Dans cette section, nous avons vu la syntaxe concrète de notre *i-DSML ASDL*. Elle est constituée essentiellement de boîtes, de pentagones ainsi que de flèches et permet de représenter n'importe quelle orchestration de l'adaptation. Dans la section 5.4, nous allons le démontrer en se servant de cette syntaxe afin de modéliser l'orchestration de l'adaptation correspondant aux *i-DSML* adaptables de la famille

*PDLHierarchy*.

## 5.4 Des orchestrations de l'adaptation pour *PDLHierarchy*

Dans la section 4.3 du chapitre précédent, nous avons construit la hiérarchie de familles *PDLHierarchy* comprenant les familles et *i-DSML* associés. Grâce à la spécialisation de familles, ces *i-DSML* offrent des opérations héritées (sauf pour celui correspondant à la famille racine) mais aussi de nouvelles opérations. Dans cette section, nous allons modéliser l'orchestration de l'adaptation correspondant à chaque *i-DSML* de la hiérarchie de familles *PDLHierarchy*. Pour cela, nous allons utiliser le langage *ASDL* et combiner les différentes opérations entre elles.

### 5.4.1 Une orchestration de l'adaptation pour *PDL*

La première famille (c'est-à-dire la racine) que nous avons définie dans la sous-section 4.3.1 s'appelle *PDL*. Elle vient avec son *i-DSML* adaptable associé qui contient les opérations d'exécution `run` et `next`. Cet *i-DSML* adaptable ne propose aucune vérification ou action d'adaptation. D'ailleurs, il serait plus juste de parler d'*i-DSML* tout court car celui-ci ne contient pas de partie structurelle adaptative. Autant dire qu'à ce stade, l'adaptation est tout simplement impossible à cause d'une spécialisation insuffisante.

Avec l'architecture présentée en section 5.1, nous devons forcément fournir un modèle d'orchestration. Si nous voulons systématiquement utiliser cette architecture, même pour les *i-DSML* ne permettant pas l'adaptation, nous devons définir un modèle d'orchestration constitué de politiques d'adaptation minimales ne réalisant pas d'adaptation. Dans cette section, nous allons modéliser l'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable *PDL*.

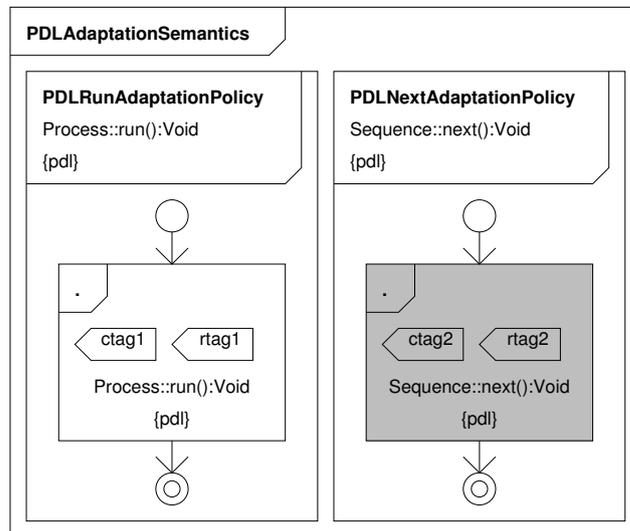


FIGURE 5.4 – Le modèle d'orchestration de l'adaptation pour *PDL*

L'orchestration de l'adaptation sur la figure 5.4 est nommée *PDLAdaptationSemantics*, ce qui a pour avantage de décrire clairement l'intention de ce modèle, qui est de fournir la sémantique d'adaptation pour l'*i-DSML* adaptable *PDL*. Cette orchestration de l'adaptation est composée des deux politiques d'adaptation *PDLRunAdaptationPolicy* et *PDLNextAdaptationPolicy*. Ainsi, ce nommage des politiques

d'adaptation indique que celle de gauche correspond à l'adaptation de l'opération `run` tandis que celle de droite correspond à l'adaptation de l'opération `next`. Évidemment, le choix du nom de l'orchestration de l'adaptation et des politiques d'adaptation appartient à l'ingénieur logiciel qui peut décider de prendre une autre convention de nommage que celle utilisée ici. En effet, le nom de ces éléments n'a aucune incidence sur le comportement de l'application adaptative et joue avant tout un rôle de documentation destinée à l'ingénieur logiciel.

La première politique d'adaptation `PDLRunAdaptationPolicy` définit l'opération d'exécution `Process::run():Void` qui est également l'opération adaptée. Cette opération n'a aucun paramètre et est située à l'intérieur de la méta-classe `Process`. Cette politique d'adaptation étant minimale, elle se contente seulement d'appeler l'opération d'exécution `Process::run():Void`. La deuxième politique d'adaptation `PDLNextAdaptationPolicy`, elle aussi, spécifie une unique opération d'exécution `Sequence::next():Void` qui correspond à l'opération adaptée. Cette opération n'a aucun paramètre et est située à l'intérieur de la méta-classe `Sequence`. Au même titre que la première, c'est une politique d'adaptation minimale qui ne fait qu'appeler l'opération d'exécution `Sequence::next():Void`. Nous pouvons remarquer que le diagramme indique que le pas d'orchestration courant est associé à l'opération d'exécution `Sequence::next():Void`. Cela signifie que la politique d'adaptation `PDLNextAdaptationPolicy` a été sélectionnée et par conséquent nous sommes en train d'adapter l'opération d'exécution `Sequence::next():Void`. Aussi, nous pouvons facilement en déduire que le prochain pas d'orchestration annoncera la fin de l'exécution de la politique d'adaptation.

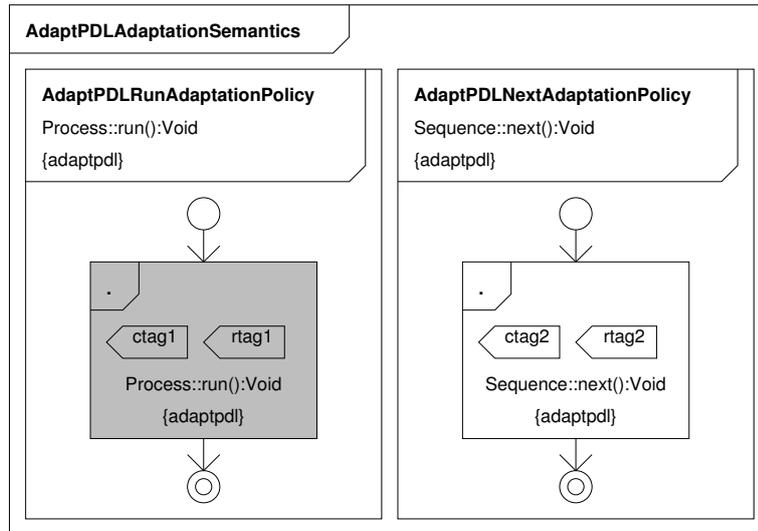
Dans cette sous-section, nous avons présenté le modèle d'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable `PDL`. Cet *i-DSML* adaptable étant trop général, son modèle d'orchestration de l'adaptation se contente d'orchestrer une exécution classique (c'est-à-dire sans adaptation). Dans la sous-section suivante, nous allons descendre dans la hiérarchie de familles `PDLHierarchy` pour se concentrer sur un *i-DSML* adaptable plus spécialisé : `AdaptPDL`. De la même manière que pour `PDL`, nous allons construire le modèle d'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable `AdaptPDL`.

### 5.4.2 Une orchestration de l'adaptation pour `AdaptPDL`

La deuxième famille que nous avons définie dans la sous-section 4.3.2 s'appelle `AdaptPDL`. Elle vient avec son *i-DSML* adaptable de même nom. Grâce à la spécialisation de familles, cet *i-DSML* adaptable offre les opérations d'exécution héritées `run` et `next` mais apporte aussi une nouvelle opération : la vérification d'adaptation `cLate`. Cet *i-DSML* adaptable ne propose aucune action d'adaptation, ce qui signifie qu'il sera impossible d'adapter lorsque la vérification d'adaptation répondra « vrai ».

Encore une fois, si nous voulons systématiquement utiliser l'architecture présentée en section 5.1, même pour les *i-DSML* ne permettant pas l'adaptation, nous devons définir malgré tout un modèle d'orchestration constitué de politiques d'adaptation minimales. Dans cette sous-section, nous allons modéliser l'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable `AdaptPDL`.

Conformément à la convention de nommage expliquée dans la sous-section précédente, l'orchestration de l'adaptation sur la figure 5.5 est nommée `AdaptPDLAdaptationSemantics` et contient les deux politiques d'adaptation `AdaptPDLRunAdaptationPolicy` et `AdaptPDLNextAdaptationPolicy`. Ces politiques d'adaptation sont similaires à celles de la précédente sous-section puisqu'elles définissent les mêmes opérations et qu'elles sont toutes des politiques d'adaptation minimales. Cependant, nous pouvons remarquer que cette fois le diagramme indique que le pas d'orchestration courant est associé à l'opération d'exécution


 FIGURE 5.5 – Le modèle d’orchestration de l’adaptation pour *AdaptPDL*

tion `Process::run():Void`. Cela signifie que la politique d’adaptation `AdaptPDLRunAdaptationPolicy` a été sélectionnée et par conséquent nous sommes en train d’adapter l’opération d’exécution `Process::run():Void`. Aussi, nous pouvons facilement en déduire que le prochain pas d’orchestration annoncera la fin de l’exécution de la politique d’adaptation.

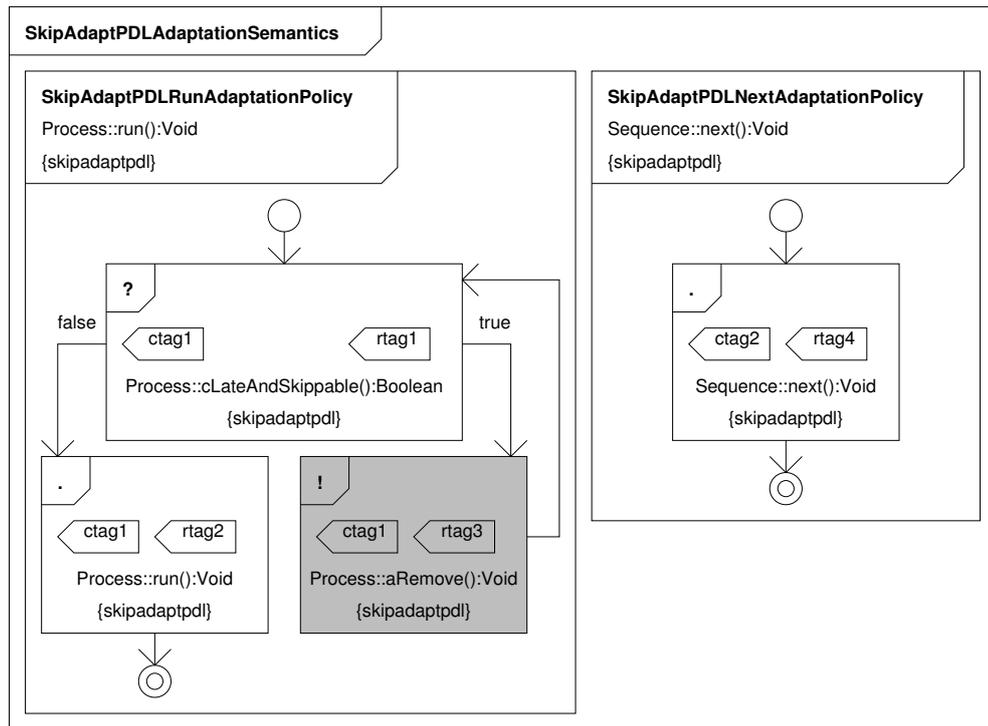
Dans cette sous-section, nous avons présenté le modèle d’orchestration de l’adaptation correspondant à l’*i-DSML* adaptable *AdaptPDL*. Cet *i-DSML* adaptable restant trop général, son modèle d’orchestration de l’adaptation se contente d’orchestrer une exécution classique comme pour celui de *PDL*. Dans la sous-section suivante, nous allons descendre une nouvelle fois dans la hiérarchie de familles *PDLHierarchy* pour se concentrer sur un *i-DSML* adaptable plus spécialisé : *SkipAdaptPDL*. De la même manière que pour *PDL* et *AdaptPDL*, nous allons construire le modèle d’orchestration de l’adaptation correspondant à l’*i-DSML* adaptable *SkipAdaptPDL*.

### 5.4.3 Une orchestration de l’adaptation pour *SkipAdaptPDL*

Dans la sous-section 4.3.3 du chapitre précédent, nous avons défini une troisième famille *SkipAdaptPDL* ainsi que son *i-DSML* adaptable associé. Grâce à la spécialisation de familles, cet *i-DSML* adaptable offre des opérations héritées (par exemple les opérations d’exécution `run` et `next`) mais aussi de nouvelles opérations (par exemple la vérification d’adaptation `cLateAndSkippable` et l’action d’adaptation `a-Remove`). Dans cette sous-section, nous allons modéliser l’orchestration de l’adaptation correspondant à cet *i-DSML* adaptable en combinant les différentes opérations entre elles.

Pour rappel, une adaptation possible avec *SkipAdaptPDL* est que tant que nous sommes en retard et que la prochaine activité d’une séquence peut être sautée, nous supprimons cette activité. Ensuite, nous exécutons l’opération qui passe à la prochaine activité d’une séquence du processus.

Conformément à la convention de nommage expliquée dans la sous-section 5.4.1, l’orchestration de l’adaptation sur la figure 5.6 est nommée `SkipAdaptPDLAdaptationSemantics` et contient les deux politiques d’adaptation `SkipAdaptPDLRunAdaptationPolicy` et `SkipAdaptPDLNextAdaptationPolicy`. La première politique d’adaptation `SkipAdaptPDLRunAdaptationPolicy` définit les trois opérations `Pro-`


 FIGURE 5.6 – Le modèle d’orchestration de l’adaptation pour *SkipAdaptPDL*

`Process::cLateAndSkippable():Boolean`, `Process::run():Void` (l’opération adaptée) et `Process::aRemove():Void`. Ces opérations n’ont aucun paramètre et sont toutes situées à l’intérieur de la méta-classe `Process`. Cette politique d’adaptation commence par effectuer une vérification d’adaptation (`Process::cLateAndSkippable():Boolean`). Si l’opération répond « vrai » alors une action d’adaptation (`Process::aRemove():Void`) sera effectuée, puis la vérification sera de nouveau réalisée et ainsi de suite. Au contraire, si l’opération répond « faux » alors une opération d’exécution (`Process::run():Void`) sera appelée car aucune adaptation n’est nécessaire dans ce cas. La deuxième politique d’adaptation `SkipAdaptPDLNextAdaptationPolicy`, quant à elle, est minimale et similaire à celle présentée dans la sous-section 5.4.1. Nous pouvons remarquer que le diagramme indique que le pas d’orchestration courant est associé à l’action d’adaptation `Process::aRemove():Void`. Cela signifie que la politique d’adaptation `SkipAdaptPDLRunAdaptationPolicy` a été sélectionnée et par conséquent nous sommes en train d’adapter l’opération d’exécution `Process::run():Void`. Aussi, nous pouvons facilement en déduire qu’au prochain pas d’orchestration, la vérification d’adaptation `Process::cLateAndSkippable():Boolean` sera l’opération courante.

Au niveau des étiquettes, nous savons que l’objet qui contient les opérations de la politique d’adaptation `SkipAdaptPDLRunAdaptationPolicy` est toujours le même puisque nous avons à l’exécution une seule instance de la méta-classe `Process`. Ainsi, nous avons décidé d’attacher une étiquette de même nom `ctag1` à toutes les opérations de cette politique d’adaptation afin d’optimiser le nombre d’étiquettes utilisées.

Cette orchestration de l’adaptation n’est pas la seule possible pour notre *i-DSML* adaptable *SkipAdaptPDL* et une autre combinaison des opérations est tout à fait envisageable. Par exemple, l’ingénieur logiciel peut choisir une solution moins radicale en préférant l’action d’adaptation `aSkip` à l’action d’adaptation

**aRemove**. Dans ce cas, il lui suffit de modifier le modèle orchestration de l'adaptation, proposé sur la figure 5.6, en remplaçant la boîte associée à l'opération en question. Ainsi, en cas de retard, au lieu de supprimer les activités facultatives, le comportement correspondant à cette nouvelle orchestration de l'adaptation sera seulement de sauter ces activités.

Dans cette sous-section, nous avons présenté le modèle d'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable *SkipAdaptPDL*. La structure de ce modèle est différente des précédents modèles d'orchestration construits. En effet, nous avons utilisé une politique d'adaptation mettant en œuvre une opération de chaque sorte et servant à adapter d'une façon spécifique lorsqu'une condition particulière se présente, contrairement aux modèles d'orchestration des figures 5.4 et 5.5 dans lesquels il n'y a qu'une seule sorte d'opération. Dans la sous-section suivante, au lieu de descendre plus en profondeur dans la hiérarchie de familles *PDLHierarchy*, nous allons cette fois nous intéresser à la famille sœur de *SkipAdaptPDL* (c'est-à-dire *DependAdaptPDL*). De la même manière que pour *PDL*, *AdaptPDL* et *SkipAdaptPDL* nous allons construire le modèle d'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable *DependAdaptPDL*.

#### 5.4.4 Une orchestration de l'adaptation pour *DependAdaptPDL*

Dans la sous-section 4.3.4 du chapitre précédent, nous avons défini une quatrième famille *DependAdaptPDL* ainsi que son *i-DSML* adaptable associé. Grâce à la spécialisation de familles, cet *i-DSML* adaptable offre des opérations héritées (par exemple les opérations d'exécution **run** et **next**) mais aussi de nouvelles opérations : la vérification d'adaptation **cLateAndMovable** et l'action d'adaptation **aParallelize**. Dans cette sous-section, nous allons modéliser l'orchestration de l'adaptation correspondant à cet *i-DSML* adaptable en combinant les différentes opérations entre elles.

Nous rappelons qu'une adaptation possible avec *DependAdaptPDL* est que tant que nous sommes en retard et que la prochaine activité d'une séquence est déplaçable (en tenant compte de ses dépendances), nous parallélisons les activités dans le but de réduire le temps du processus. Ensuite, nous exécutons l'opération qui passe à la prochaine activité d'une séquence du processus.

Afin de rester en conformité avec la convention de nommage détaillée dans la sous-section 5.4.1, l'orchestration de l'adaptation sur la figure 5.7 est nommée **DependAdaptPDLAdaptationSemantics** et contient les deux politiques d'adaptation **DependAdaptPDLRunAdaptationPolicy** et **DependAdaptPDLNextAdaptationPolicy**. La première politique d'adaptation **DependAdaptPDLRunAdaptationPolicy** définit les trois opérations **Process::cLateAndMovable():Boolean**, **Process::run():Void** (l'opération adaptée) et **Process::aParallelize():Void**. Ces opérations n'ont aucun paramètre et sont toutes situées à l'intérieur de la méta-classe **Process**. Cette politique d'adaptation commence par effectuer une vérification d'adaptation (**Process::cLateAndMovable():Boolean**). Si l'opération répond « vrai » alors une action d'adaptation (**Process::aParallelize():Void**) sera effectuée, puis la vérification sera de nouveau réalisée et ainsi de suite. Au contraire, si l'opération répond « faux » alors une opération d'exécution (**Process::run():Void**) sera appelée car aucune adaptation n'est nécessaire dans ce cas. La deuxième politique d'adaptation **DependAdaptPDLNextAdaptationPolicy**, quant à elle, est minimale et similaire à celle présentée dans la sous-section 5.4.1. Nous pouvons remarquer que le diagramme indique que le pas d'orchestration courant est associé à l'opération d'exécution **Process::run():Void**. Cela signifie que la politique d'adaptation **DependAdaptPDLRunAdaptationPolicy** a été sélectionnée et par conséquent nous sommes en train d'adapter l'opération d'exécution **Process::run():Void**. Aussi,

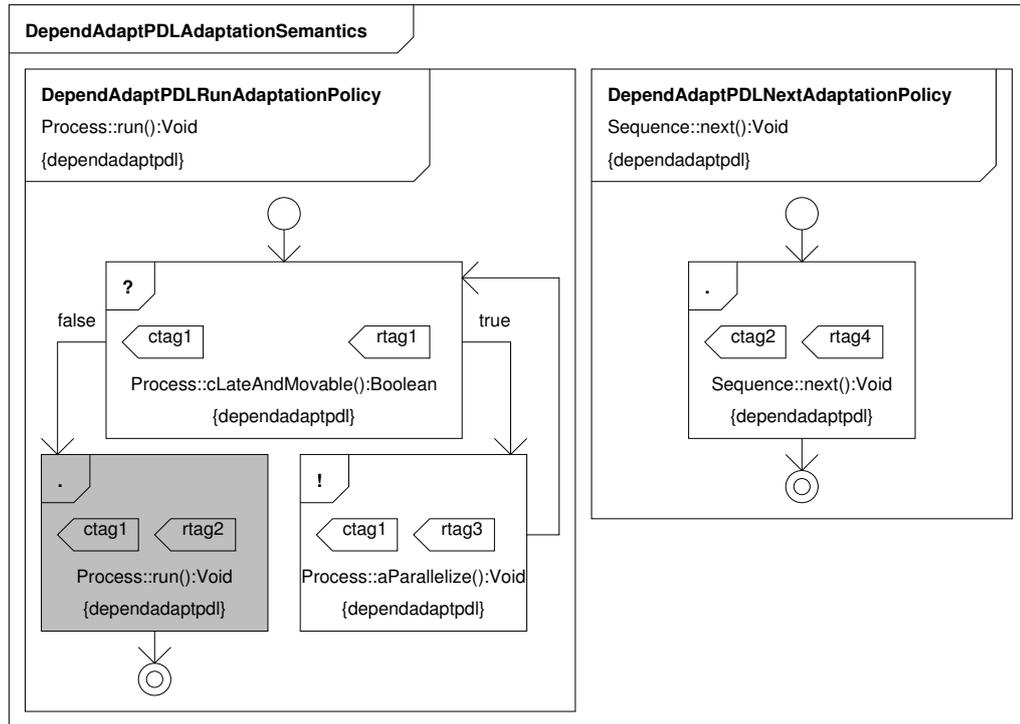


FIGURE 5.7 – Le modèle d’orchestration de l’adaptation pour *DependAdaptPDL*

nous pouvons facilement en déduire que le prochain pas d’orchestration annoncera la fin de l’exécution de la politique d’adaptation.

Dans cette sous-section, nous avons présenté le modèle d’orchestration de l’adaptation correspondant à l’*i-DSML* adaptable *DependAdaptPDL*. La structure de ce modèle est tout à fait semblable à celle pour *SkipAdaptPDL*. En effet, nous avons également utilisé une politique d’adaptation mettant en œuvre une opération de chaque sorte et servant à adapter d’une façon spécifique lorsqu’une condition particulière se présente. Dans la sous-section suivante, nous allons descendre plus en profondeur dans la hiérarchie de familles *PDLHierarchy*, pour nous intéresser au cas de la famille *DependSkipAdaptPDL* et de son *i-DSML* adaptable associé. Contrairement à *SkipAdaptPDL* et *DependAdaptPDL*, nous allons construire cette fois le modèle d’orchestration de l’adaptation correspondant à l’*i-DSML* adaptable *DependSkipAdaptPDL* en utilisant une structure complexe.

#### 5.4.5 Une orchestration de l’adaptation pour *DependSkipAdaptPDL*

Dans la sous-section 4.3.5 du chapitre précédent, nous avons défini une cinquième famille *DependSkipAdaptPDL* ainsi que son *i-DSML* adaptable associé. Grâce à la spécialisation de familles, cet *i-DSML* adaptable offre des opérations héritées (par exemple les opérations d’exécution `run` et `next` ainsi que les vérifications d’adaptation `cLateAndSkippable` et `cLateAndMovable`) mais aussi une nouvelle opération : l’action d’adaptation `aPostpone`. Dans cette sous-section, nous allons modéliser l’orchestration de l’adaptation correspondant à cet *i-DSML* adaptable en combinant les différentes opérations entre elles.

Pour rappel, une adaptation possible avec *DependSkipAdaptPDL* est que tant que nous sommes en retard, que la prochaine activité d’une séquence peut être sautée et que cette activité est déplaçable (en

tenant compte de ses dépendances), nous déplaçons aussi loin que possible cette activité vers la fin du processus afin de l'exécuter seulement si le retard accumulé a été rattrapé depuis. Ensuite, nous exécutons l'opération qui passe à la prochaine activité d'une séquence du processus.

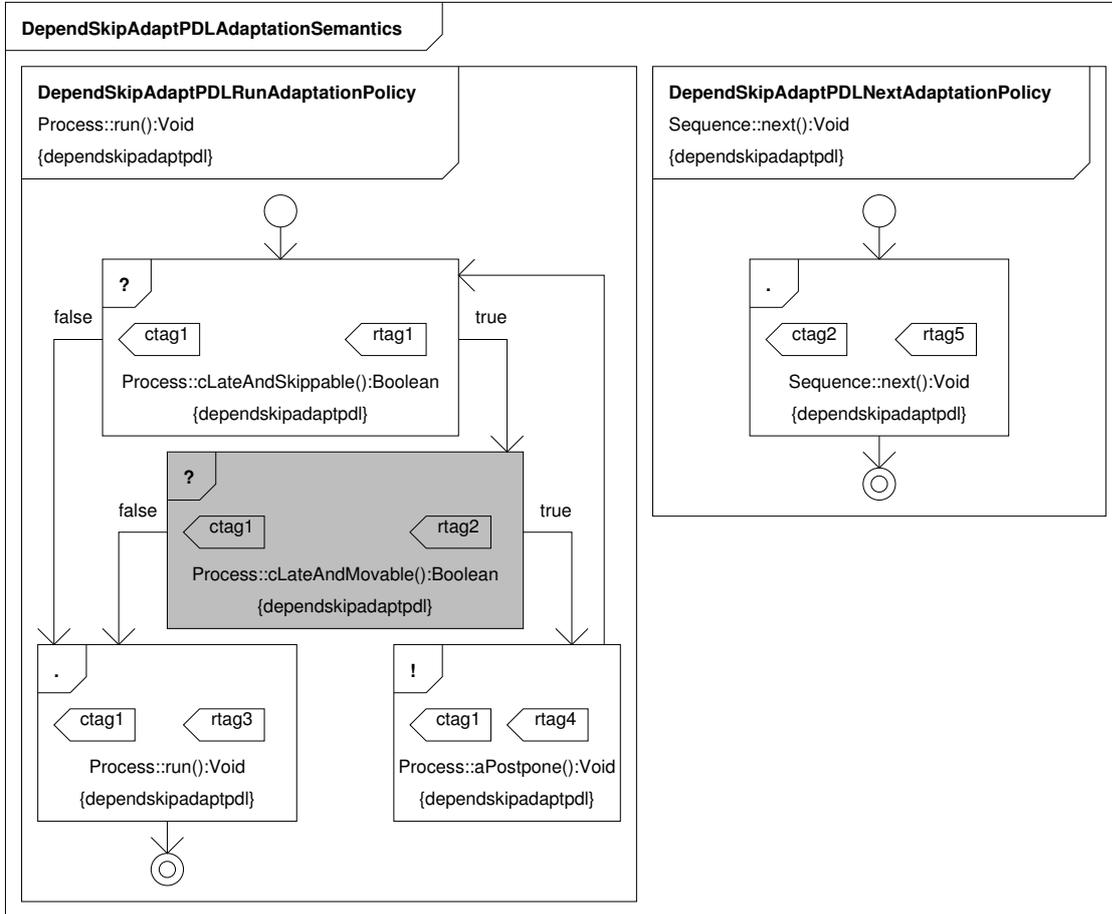


FIGURE 5.8 – Le modèle d'orchestration de l'adaptation pour *DependSkipAdaptPDL*

Comme nous conservons notre convention de nommage décrite dans la sous-section 5.4.1, l'orchestration de l'adaptation sur la figure 5.8 est nommée *DependSkipAdaptPDLAdaptationSemantics* et contient les deux politiques d'adaptation *DependSkipAdaptPDLRunAdaptationPolicy* et *DependSkipAdaptPDLNextAdaptationPolicy*. La première politique d'adaptation *DependSkipAdaptPDLRunAdaptationPolicy* définit les quatre opérations `Process::cLateAndSkippable():Boolean`, `Process::cLateAndMovable():Boolean`, `Process::run():Void` (l'opération adaptée) et `Process::aPostpone():Void`. Ces opérations n'ont aucun paramètre et sont toutes situées à l'intérieur de la méta-classe *Process*. Cette politique d'adaptation commence par effectuer une vérification d'adaptation (`Process::cLateAndSkippable():Boolean`). Si l'opération répond « vrai », alors nous effectuons une deuxième vérification d'adaptation (`Process::cLateAndMovable():Boolean`). Si cette dernière répond également « vrai » alors une action d'adaptation (`Process::aPostpone():Void`) sera effectuée, puis la première vérification sera de nouveau réalisée et ainsi de suite. Au contraire, si l'une des vérifications d'adaptation répond « faux » alors une opération d'exécution (`Process::run():Void`) sera appelée car aucune adaptation n'est nécessaire dans ce cas. La deuxième politique d'adaptation *DependSkipAdaptPDLNextAdaptationPolicy*, quant à elle,

est minimale et similaire à celle présentée dans la sous-section 5.4.1. Nous pouvons remarquer que le diagramme indique que le pas d'orchestration courant est associé à la deuxième vérification d'adaptation `Process::cLateAndMovable():Boolean`. Cela signifie que la politique d'adaptation `DependSkipAdaptPDLRunAdaptationPolicy` a été sélectionnée et par conséquent nous sommes en train d'adapter l'opération d'exécution `Process::run():Void`. Aussi, nous pouvons facilement en déduire qu'au prochain pas d'orchestration, l'opération courante sera soit l'action d'adaptation `Process::aPostpone():Void`, soit l'opération d'exécution `Process::run():Void`.

La politique d'adaptation `DependSkipAdaptPDLRunAdaptationPolicy` est complexe puisqu'elle combine plusieurs vérifications d'adaptation entre elles. Nous avons mis en place l'équivalent d'une conjonction (un « ET » logique) puisque deux vérifications d'adaptation doivent être vraies pour qu'une action d'adaptation se réalise. Toutefois, même si nous venons de montrer que notre langage *ASDL* permet cela, il ne possède pas les concepts nécessaires pour gérer facilement cette logique booléenne. En effet, il manque la notion d'opérateur logique : la conjonction (un « ET » logique), la disjonction (un « OU » logique) ou encore la négation (un « NON » logique). Nous recommandons donc d'implémenter cette logique directement au sein du moteur d'exécution en créant de nouvelles opérations qui réalisent elles-mêmes cette combinaison. C'est ce que nous avons fait pour l'opération `Process::cLateAndSkippable():Boolean` puisque nous avons réutilisé l'opération héritée `Process::cLate():Boolean` pour construire cette nouvelle vérification d'adaptation. L'avantage est que cela permet de concevoir des modèles d'orchestration de l'adaptation simples dont les politiques d'adaptation définissent une seule vérification d'adaptation.

Dans cette sous-section, nous avons présenté le modèle d'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable `DependSkipAdaptPDL`. La structure de ce modèle est différente des précédents modèles d'orchestration construits. En effet, cette fois, nous avons utilisé une politique d'adaptation complexe mettant en œuvre plusieurs vérifications d'adaptation, contrairement aux modèles d'orchestration des figures 5.6 et 5.7, dans lesquels une seule vérification d'adaptation est présente. Dans la sous-section suivante, nous allons nous intéresser à la dernière famille de la hiérarchie de familles `PDLHierarchy : ManagedSkipAdaptPDL`. Nous allons construire le modèle d'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable `ManagedSkipAdaptPDL` associé à cette famille.

#### 5.4.6 Une orchestration de l'adaptation pour `ManagedSkipAdaptPDL`

Dans la sous-section 4.3.6 du chapitre précédent, nous avons défini une sixième famille `ManagedSkipAdaptPDL` ainsi que son *i-DSML* adaptable associé. Grâce à la spécialisation de familles, cet *i-DSML* adaptable offre des opérations héritées (par exemple les opérations d'exécution `run` et `next`, la vérification d'adaptation `cLateAndSkippable` ainsi que l'action d'adaptation `aSkip`) mais aussi de nouvelles opérations : la vérification d'adaptation `cTooManySkips` ainsi que les actions d'adaptation `aIncrNumSkips` et `aChangeProjectManager`. Dans cette sous-section, nous allons modéliser l'orchestration de l'adaptation correspondant à cet *i-DSML* adaptable en combinant les différentes opérations entre elles.

Nous rappelons qu'une adaptation possible avec `ManagedSkipAdaptPDL` est que tant que nous sommes vraiment très en retard (c'est-à-dire que le nombre d'activités sautées est supérieur à une valeur maximale spécifiée), nous créons une activité particulière dans le processus pour changer le chef de projet (ajoutée en parallèle avec les activités existantes du processus) qui est immédiatement activée. Ensuite, nous exécutons l'opération qui passe à la prochaine activité d'une séquence du processus.

Conformément à la convention de nommage expliquée dans la sous-section 5.4.1, l'orchestration

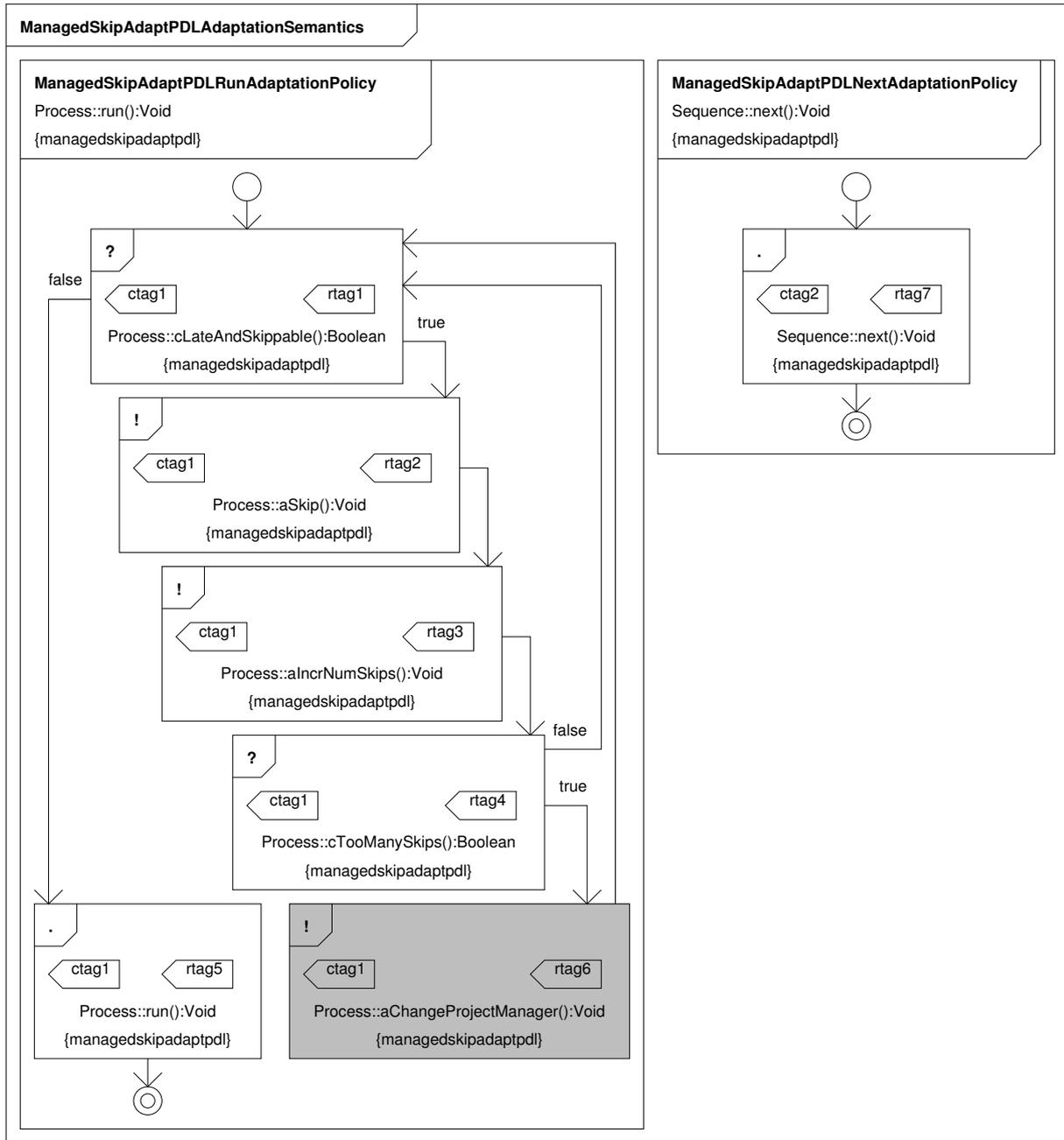


FIGURE 5.9 – Le modèle d’orchestration de l’adaptation pour *ManagedSkipAdaptPDL*

de l’adaptation sur la figure 5.9 est nommée *ManagedSkipAdaptPDLAdaptationSemantics* et contient les deux politiques d’adaptation *ManagedSkipAdaptPDLRunAdaptationPolicy* et *ManagedSkipAdaptPDLNextAdaptationPolicy*. La première politique d’adaptation *ManagedSkipAdaptPDLRunAdaptationPolicy* définit les six opérations `Process::cLateAndSkippable():Boolean`, `Process::aSkip():Void`, `Process::aIncrNumSkips():Void`, `Process::cTooManySkips():Boolean`, `Process::run():Void` (l’opération adaptée) et `Process::aChangeProjectManager():Void`. Ces opérations n’ont aucun paramètre et sont toutes situées à l’intérieur de la méta-classe *Process*. Cette politique d’adaptation commence par ef-

effectuer une vérification d'adaptation (`Process::cLateAndSkippable():Boolean`). Si l'opération répond « vrai », alors nous effectuons l'action d'adaptation `Process::aSkip():Void`, puis l'action d'adaptation `Process::aIncrNumSkips():Void`, puis une deuxième vérification d'adaptation (`Process::cTooManySkips():Boolean`). Si cette dernière répond également « vrai » alors une action d'adaptation (`Process::aChangeProjectManager():Void`) sera effectuée, puis la première vérification sera de nouveau réalisée et ainsi de suite. Au contraire, si la première vérification d'adaptation répond « faux » alors une opération d'exécution (`Process::run():Void`) sera appelée car aucune adaptation n'est nécessaire dans ce cas. Si c'est seulement la deuxième vérification d'adaptation qui répond « faux » alors la première vérification sera de nouveau réalisée et ainsi de suite. La deuxième politique d'adaptation `ManagedSkipAdaptPDLNextAdaptationPolicy`, quant à elle, est minimale et similaire à celle présentée dans la sous-section 5.4.1. Nous pouvons remarquer que le diagramme indique que le pas d'orchestration courant est associé à l'action d'adaptation `Process::aChangeProjectManager():Void`. Cela signifie que la politique d'adaptation `ManagedSkipAdaptPDLRunAdaptationPolicy` a été sélectionnée et par conséquent nous sommes en train d'adapter l'opération d'exécution `Process::run():Void`. Aussi, nous pouvons facilement en déduire qu'au prochain pas d'orchestration, l'opération courante sera la vérification d'adaptation `Process::cLateAndSkippable():Boolean`.

Dans cette sous-section, nous avons présenté le modèle d'orchestration de l'adaptation correspondant à l'*i-DSML* adaptable `ManagedSkipAdaptPDL`. Bien que ce modèle contienne un plus grand nombre d'opérations que celui pour `DependSkipAdaptPDL`, sa structure reste tout à fait semblable. En effet, nous avons également utilisé une politique d'adaptation complexe mettant en œuvre plusieurs vérifications d'adaptation. Nous venons de clôturer cette section avec ce dernier modèle d'orchestration de l'adaptation puisque nous avons traité toutes les familles de la hiérarchie de familles `PDLHierarchy`. Dans la prochaine section, nous donnons une conclusion sur ce chapitre consacré à l'orchestration de l'adaptation des *i-DSML*.

## 5.5 Conclusion

Dans ce chapitre, nous avons proposé une approche pour l'orchestration de l'adaptation des *i-DSML* adaptables. Cette solution, destinée à l'ingénieur de logiciels adaptatifs, repose sur une architecture logicielle particulière et sur un *i-DSML* d'orchestration de l'adaptation. L'architecture de l'application adaptative est séparée en deux moteurs distincts : un moteur d'exécution et le moteur d'orchestration. Cette façon de procéder permet de profiter d'un programme adaptatif semi-générique basé sur la séparation des préoccupations, offrant ainsi une meilleure réutilisation et une maintenance facilitée.

Concernant l'*i-DSML* d'orchestration de l'adaptation, nous avons présenté d'une part sa syntaxe abstraite et d'autre part sa syntaxe concrète. Sa syntaxe abstraite, constituée des différents concepts liés à une telle orchestration, contient des politiques d'adaptation, qui elles-mêmes sont composées de pas d'orchestration associés à une opération. Ces opérations peuvent être des opérations d'exécution, des vérifications d'adaptation ou encore des actions d'adaptation. La syntaxe concrète de notre langage, permettant de représenter un modèle d'orchestration de l'adaptation à l'aide d'un diagramme, est composée de boîtes, de pentagones ainsi que de flèches et permet de représenter n'importe quelle orchestration de l'adaptation.

Nous avons montré que notre approche fonctionne en l'appliquant à chaque *i-DSML* adaptable de la hiérarchie de familles `PDLHierarchy`. En fonction du niveau de spécialisation de la famille, les politiques d'adaptation ainsi créées sont plus ou moins complexes. Ainsi, nous avons construit des politiques

d'adaptation minimales, qui se contentent d'appeler une opération d'exécution, des politiques d'adaptation classiques, qui sont composées d'une opération de chaque sorte et des politiques d'adaptation complexes qui contiennent, quant à elles, plusieurs vérifications d'adaptation.

L'implémentation d'une telle application adaptative peut être réalisée au travers d'outils dédiés à cette tâche et offerts par l'IDM. Dans le chapitre suivant nous allons montrer comment implémenter notre approche au travers de ces technologies disponibles.

# Chapitre 6

## Implémentation

Nous avons implémenté l’approche proposée dans cette thèse en utilisant une technologie orientée modèle. Dans ce chapitre, nous allons expliquer pas à pas comment l’ingénieur logiciel peut parvenir à mettre en œuvre une telle approche au travers des outils mis à sa disposition. Nous partirons d’un modèle adaptable, d’une famille et d’un modèle d’orchestration de l’adaptation déjà existants pour construire progressivement l’application adaptative grâce à *Java/EMF*. Le modèle adaptable considéré est conçu avec l’*i-DSML* adaptable *SkipAdaptPDL* associé à la famille de même nom présentée dans la sous-section 4.3.3. Le modèle d’orchestration dont nous allons nous servir est conçu avec l’*i-DSML ASDL* détaillé dans la section 5.2. Comme expliqué dans le chapitre précédent, notre logiciel adaptatif est composé de deux moteurs distincts : un moteur d’exécution donné et le moteur d’orchestration. Nous allons d’abord implémenter le moteur d’exécution, puis nous chercherons à le coupler au moteur d’orchestration. Dans tout le code source qui sera présenté dans ce chapitre, nous avons volontairement omis la gestion des exceptions dans un souci de clarté.

### 6.1 Un exemple d’implémentation d’un moteur d’exécution

Une fois le méta-modèle de l’*i-DSML* adaptable *SkipAdaptPDL* conçu avec le langage *Ecore*, nous pouvons profiter du générateur de code fourni par l’environnement *Java/EMF* afin d’implémenter le moteur d’exécution associé. En effet, cet outil facilite le travail de l’ingénieur logiciel en construisant un squelette de code à partir d’un méta-modèle donné. Chaque méta-classe du méta-modèle est transformée en une classe *Java* implémentant une interface tandis que les opérations, quant à elles, sont transformées en méthodes *Java*. Cependant, ce code doit être complété par l’ingénieur logiciel afin de rendre le moteur d’exécution fonctionnel puisque d’une part, le corps des méthodes associées aux opérations d’exécution, vérifications d’adaptation et actions d’adaptation est laissé vide par le générateur de code et d’autre part ce générateur ne construit pas la méthode `main` nécessaire à l’exécution de tout programme *Java*.

Concernant les opérations d’exécution, considérons la méthode `next` de la classe `SequenceImpl` dont nous devons compléter le corps. Comme expliqué dans la sous-section 4.1.1, si la séquence vient juste d’être lancée alors la méthode `next` doit exécuter la première activité de la séquence (cf. lignes 3 et 4 du listing 6.1), sinon c’est l’activité suivante de l’activité courante qu’elle doit exécuter (cf. lignes 5 et 6 du listing 6.1).

```
1 @Override public void next()
```

```
2 {
3  if(this.getCurrentActivity() == null)
4    this.currentActivity = this.firstActivity;
5  else if(this.getCurrentActivity().getNextActivity() != null)
6    this.currentActivity = this.getCurrentActivity().getNextActivity();
7 }
```

Listing 6.1 – Le code *Java/EMF* de la méthode `next` de la classe `SequenceImpl`

Au niveau des vérifications d'adaptation, prenons le cas de la méthode `cLate` de la classe `ProcessImpl`. Conformément à ce qui a été écrit dans la sous-section 4.3.2, cette méthode doit déterminer si une exécution d'un processus est en retard en comparant la durée attendue avec le temps réellement écoulé depuis le début du processus (cf. ligne 12 du listing 6.2).

```
1 @Override public Boolean cLate()
2 {
3  Integer expectedDuration = 0;
4  for(ProcessElement processElement : this.getCurrentProcessElements())
5  {
6    if(processElement instanceof Sequence)
7    {
8      Sequence sequence = (Sequence) processElement;
9      expectedDuration = Math.max(expectedDuration, sequence.computeExpectedDuration());
10 }
11 }
12 if(this.elapsedTime > expectedDuration)
13   return true;
14   return false;
15 }
```

Listing 6.2 – Le code *Java/EMF* de la méthode `cLate` de la classe `ProcessImpl`

À titre d'exemple pour les actions d'adaptation, implémentons la méthode `aSkip` de la classe `ProcessImpl`. Pour refléter ce qui a été dit dans la sous-section 4.3.3, cette méthode doit effectuer un saut des activités facultatives (c'est-à-dire mettre à jour la référence de l'activité courante). Pour effectuer ces sauts, nous avons simplement décidé de faire des appels à l'opération d'exécution `next` (cf lignes 11 et 12 du listing 6.3).

```
1 @Override public void aSkip()
2 {
3  for(ProcessElement processElement : this.getCurrentProcessElements())
4  {
5    if(processElement instanceof Sequence)
6    {
7      Sequence sequence = (Sequence) processElement;
8      if(sequence.getCurrentActivity().isCompleted())
9      {
```

```

10     while(sequence.isSkippable() && this.cLate())
11         sequence.next();
12     sequence.next();
13 }
14 }
15 }
16 }

```

Listing 6.3 – Le code *Java/EMF* de la méthode `aSkip` de la classe `ProcessImpl`

Le corps des méthodes étant complété, nous devons maintenant nous intéresser à l'implémentation de la méthode `main` du moteur d'exécution. Nous appelons `SkipAdaptPDLEngine` notre classe pour le moteur d'exécution et comme expliqué dans la sous-section 5.1, le moteur d'exécution est particulier puisqu'il constitue le point d'entrée du logiciel adaptatif. Ainsi, c'est bien cette classe qui contient la méthode principale du programme (`main`). La classe de notre moteur d'exécution ne définit que des méthodes de classe et n'a donc pas besoin d'être instanciée pour être utilisée. Pour cette raison, la classe de notre moteur d'exécution est abstraite.

Le code de la méthode `main` commence par initialiser *EMF* (cf. lignes 5 à 9 du listing 6.4). Ensuite, nous devons charger le modèle. Ici, nous demandons à l'utilisateur de choisir un modèle situé dans le dossier `models/skipadaptpdl`. Une fois le modèle sélectionné, celui-ci est chargé par le moteur d'exécution pour qu'il soit prêt à être exécuté. Après cela, nous entrons dans une boucle (cf. lignes 13 à 29 du listing 6.4) qui appelle la méthode (correspondant à une opération d'exécution) pour réaliser les pas d'exécution. Puisque dans notre implémentation, les activités ne sont rattachées à aucune action tangible, il s'agit ici simplement de simuler le fonctionnement d'un processus par l'intervention d'un utilisateur humain. Bien entendu, nous pourrions facilement modifier le moteur afin de détecter automatiquement la fin d'une activité ce qui déclencherait l'exécution de l'activité suivante. Nous interagissons avec l'utilisateur pour lui demander de saisir soit « c » pour signaler que nous pouvons procéder au prochain pas d'exécution, soit « e » pour signaler que nous souhaitons sortir du programme.

```

1 public abstract class SkipAdaptPDLEngine extends Engine
2 {
3     public static void main(String[] args)
4     {
5         // Initialize EMF
6         SkipadaptpdlPackage.eINSTANCE.eClass();
7         Resource.Factory.Registry registry = Resource.Factory.Registry.INSTANCE;
8         Map<String, Object> map = registry.getExtensionToFactoryMap();
9         map.put("xmi", new XMIResourceFactoryImpl());
10        // Load the model
11        String skipAdaptPDLModel = SkipAdaptPDLEngine.askModelName("models/skipadaptpdl");
12        Process process = (Process) SkipAdaptPDLEngine.load("models/skipadaptpdl/" +
13            skipAdaptPDLModel + ".xmi");
14        // Execution loop
15        Integer integer = 0;
16        while(true)

```

```
16 {
17     // Interact with the user
18     System.out.println("SkipAdptPDLEngine.java(main): Type 'c' to continue (or 'e' to
        exit):");
19     String choice = IO.scanner.nextLine();
20     while(!choice.equals("e") && !choice.equals("c"))
21         choice = IO.scanner.nextLine();
22     if(choice.equals("e"))
23         break;
24     // Do an execution step
25     process.run();
26     // Save the current state of the model
27     SkipAdaptPDLEngine.save("output/skipadaptpdl/" + skipAdaptPDLModel + integer + ".xmi
        ", process);
28     integer++;
29 }
30 }
31 }
```

Listing 6.4 – Le code *Java/EMF* de la classe `SkipAdaptPDLEngine`

## 6.2 Le couplage au moteur d’orchestration

Le moteur d’exécution présenté dans la section précédente doit être couplé au moteur d’orchestration, ce qui demande dans un premier temps d’identifier les méthodes à adapter dans le moteur d’exécution et de les renommer en suivant une convention de nommage particulière. Nous avons décidé pour cela d’ajouter le suffixe `Adapted` au nom de ces méthodes. Ensuite, nous créons une méthode qui prend le nom d’origine de la méthode à adapter. Cette nouvelle méthode contient un code très court qui se contente de remplir le tableau associatif contenant les étiquettes associées aux objets et d’appeler le moteur d’orchestration pour adapter la méthode. Par exemple, la méthode `run` appartenant à la classe `ProcessImpl` est renommée `runAdapted` (cf. ligne 9 du listing 6.5). Ceci nous permet de créer une nouvelle méthode `run` qui ajoute au tableau associatif une étiquette `ctag1` associée à l’objet `this` (cf. ligne 3 du listing 6.5). Cette étiquette référence l’objet contenant la méthode `run` (en l’occurrence, l’instance courante de la classe `ProcessImpl`) et il est inutile d’ajouter des étiquettes supplémentaires pour notre exemple. Cependant, nous montrons comment ajouter d’autres étiquettes au tableau associatif qui sont associées cette fois à des objets correspondant à des valeurs de paramètres. L’appel à la méthode `adaptOperation` du moteur d’orchestration est réalisé en passant en paramètres : le nom de la méta-classe contenant l’opération à adapter (qui en *Java/EMF* correspond en fait au nom complet de la classe `skipadaptpdl.impl.ProcessImpl` contenant la méthode à adapter), le nom de la méthode à adapter et le type de ses paramètres (ici, c’est une liste vide puisque cette méthode est dépourvue de paramètre).

```
1 @Override public void run()
2 {
3     OrchestrationEngine.put("ctag1", this);
```

```

4  OrchestrationEngine.put("ptag1", new Boolean("true"));
5  OrchestrationEngine.put("ptag2", new Integer("3"));
6  OrchestrationEngine.put("ptag3", new String("abc"));
7  OrchestrationEngine.adaptOperation(this.getClass().getCanonicalName(), "run", new
    BasicEList<Object>());
8  }
9  public void runAdapted()
10 {
11 // here is the original code of the run method
12 }

```

Listing 6.5 – Le renommage de la méthode à adapter

En plus de ce processus de renommage, pour coupler le moteur d'exécution au moteur d'orchestration nous avons besoin de retoucher légèrement le code de la méthode `main` du moteur d'exécution. En effet, cette méthode doit dorénavant initialiser *EMF* de manière à prendre en compte le méta-modèle de l'*i-DSML ASDL* et initialiser le moteur d'orchestration pour que ce dernier soit prêt à orchestrer l'adaptation. Pour que le méta-modèle *ASDL* soit pris en considération par l'environnement *Java/EMF*, nous devons ajouter une instruction lors de l'initialisation d'*EMF* (cf. lignes 6 à 10 du listing 6.6). Concernant l'initialisation du moteur d'orchestration, une fois que le modèle à adapter est chargé, nous appelons la méthode `init` du moteur d'orchestration (cf. lignes 14 et 15 du listing 6.6).

```

1  public abstract class SkipAdaptPDLEngine extends Engine
2  {
3  public static void main(String[] args)
4  {
5  // Initialize EMF
6  AsdlPackage.eINSTANCE.eClass();
7  SkipadaptpdlPackage.eINSTANCE.eClass();
8  Resource.Factory.Registry registry = Resource.Factory.Registry.INSTANCE;
9  Map<String, Object> map = registry.getExtensionToFactoryMap();
10 map.put("xmi", new XMIResourceFactoryImpl());
11 // Load the model
12 String skipAdaptPDLModel = SkipAdaptPDLEngine.askModelName("models/skipadaptpdl");
13 Process process = (Process) SkipAdaptPDLEngine.load("models/skipadaptpdl/" +
    skipAdaptPDLModel + ".xmi");
14 // Initialize the orchestration engine
15 OrchestrationEngine.init();
16 // Execution loop
17 Integer integer = 0;
18 while(true)
19 {
20 // Interact with the user
21 System.out.println("SkipAdptPDLEngine.java(main): Type 'c' to continue (or 'e' to
    exit):");

```

```
22 String choice = IO.scanner.nextLine();
23 while(!choice.equals("e") && !choice.equals("c"))
24     choice = IO.scanner.nextLine();
25 if(choice.equals("e"))
26     break;
27 // Do an execution step
28 process.run();
29 // Save the current state of the model
30 SkipAdaptPDLEngine.save("output/skipadaptddl/" + skipAdaptPDLModel + integer + ".xmi
    ", process);
31 integer++;
32 }
33 }
34 }
```

Listing 6.6 – Le code *Java/EMF* de la classe `SkipAdaptPDLEngine` après couplage

### 6.3 Le fonctionnement du moteur d'orchestration

Dans la section précédente, nous avons montré comment coupler le moteur d'exécution au moteur d'orchestration. Nous allons maintenant nous intéresser à ce dernier en commençant par présenter la classe `OrchestrationEngine` qui contient les méthodes nécessaires au moteur d'exécution pour communiquer avec le moteur d'orchestration. Comme expliqué dans la sous-section précédente, le moteur d'orchestration est initialisé par le moteur d'exécution. Ainsi, nous trouvons dans cette classe la méthode permettant cette initialisation (`init`). Nous savons que ce moteur d'orchestration doit gérer des étiquettes qui sont attachées à des objets (cf. section 5.2). Pour cela, nous stockons ces informations à l'intérieur d'une structure de données appropriée : un tableau associatif où les clés sont les étiquettes et les valeurs sont les objets. L'accès en lecture à ce tableau associatif se fait par l'intermédiaire de la méthode `get` tandis que son accès en écriture requiert un appel à la méthode `put`. Ce moteur d'orchestration doit permettre l'adaptation d'une méthode et possède donc une méthode `adaptOperation` dédiée à cette tâche. Dans la même idée que la classe du moteur d'exécution, celle-ci est implémentée de telle sorte que toutes les méthodes qu'elle définit sont des méthodes de classe. Pour cette raison, la classe de notre moteur d'orchestration est elle aussi abstraite. La figure 6.1 nous montre le diagramme de classes pour la classe du moteur d'orchestration.

Le code de la méthode `init` se charge d'initialiser tous les attributs la classe et commence par vider le tableau associatif (cf. ligne 8 du listing 6.7). Ensuite, cette méthode interagit avec l'utilisateur pour lui demander quel modèle d'orchestration charger. Bien entendu, cette étape est facultative puisque le cas où nous disposons que d'un seul modèle à charger peut se présenter. Une fois le modèle sélectionné, celui-ci est chargé en mémoire. Concernant la méthode `adaptOperation`, elle se contente d'appeler la méthode `run` déclarée par l'interface générée `AdaptationSemantics` (cf. ligne 23 du listing 6.7). Cet appel est réalisé sur l'instance du modèle d'orchestration en passant tous les paramètres de la méthode appelante à la méthode appelée. Ainsi, toutes les informations concernant la méthode à adapter sont transmises : sa classe, son nom et ses paramètres.

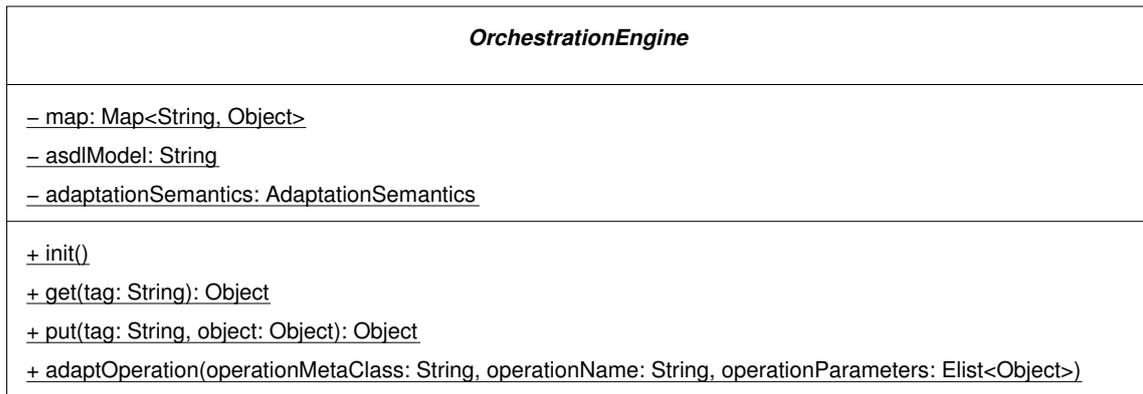


FIGURE 6.1 – Le diagramme de classes pour la classe du moteur d’orchestration

```

1 public abstract class OrchestrationEngine extends Engine
2 {
3     private static Map<String, Object> map = new HashMap<String, Object>();
4     private static String asdlModel;
5     private static AdaptationSemantics adaptationSemantics;
6     public static void init()
7     {
8         OrchestrationEngine.map.clear();
9         OrchestrationEngine.asdlModel = OrchestrationEngine.askModelNumber("models/asdl");
10        // Load the model
11        OrchestrationEngine.adaptationSemantics = (AdaptationSemantics) OrchestrationEngine.
            load("models/asdl/" + OrchestrationEngine.asdlModel + ".xmi");
12    }
13    public static Object get(String tag)
14    {
15        return OrchestrationEngine.map.get(tag);
16    }
17    public static Object put(String tag, Object object)
18    {
19        return OrchestrationEngine.map.put(tag, object);
20    }
21    public static void adaptOperation(String operationMetaClass, String operationName,
            EList<Object> operationParameters)
22    {
23        OrchestrationEngine.adaptationSemantics.run(operationMetaClass, operationName,
            operationParameters);
24    }
25 }

```

Listing 6.7 – Le code *Java/EMF* de la classe `OrchestrationEngine`

Maintenant que nous savons que l’appel à la méthode `adaptOperation` de la classe `Orchestration-`

`Engine` permet de donner la main au moteur d'orchestration pour adapter une opération, nous allons nous pencher sur ce processus permettant l'adaptation d'une opération. Nous avons décrit dans la section 5.1, comment une opération d'exécution doit être adaptée. Une première étape consiste à rechercher des politiques d'adaptation tandis qu'une deuxième étape exécute une politique d'adaptation en invoquant des opérations de rappel. En *Java/EMF*, l'adaptation d'une opération revient en fait à implémenter l'adaptation de la méthode *Java* correspondante.

### 6.3.1 Phase 1 : recherche des politiques d'adaptation

Comme nous l'avons expliqué au début de cette section, la méthode `adaptOperation` du moteur d'orchestration se charge d'appeler la méthode `run` qui est définie dans la classe générée `AdaptationSemanticsImpl` (cf. ligne 23 du listing 6.7). Cette méthode `run` parcourt toutes les politiques d'adaptation définies dans le modèle d'orchestration afin de trouver celles qui correspondent à la méthode à adapter (cf. lignes 6 à 12 du listing 6.8). Pour cela, elle compare la signature de la méthode à adapter avec la signature de l'opération adaptée par chaque politique d'adaptation. Si une politique d'adaptation est trouvée, alors la méthode `run` définie dans la classe générée `AdaptationPolicyImpl` est invoquée sur cette instance (cf. ligne 11 du listing 6.8). Au contraire, si aucune correspondance n'est trouvée, alors cela signifie que le modèle d'orchestration est incomplet et une erreur est générée.

```

1 public void run(String operationMetaClass, String operationName, EList<Object>
    operationParameters)
2 {
3     Collection<String> operationParameterTypes = new ArrayList<String>();
4     for(int index = 0; index < operationParameters.size(); index++)
5         operationParameterTypes.add(operationParameters.get(index).getClass().
            getCanonicalName());
6     AdaptationPolicy selectedAdaptationPolicy = null;
7     for(AdaptationPolicy adaptationPolicy : this.getAdaptationPolicies())
8         if(adaptationPolicy.getAdaptedOperation().getFullyQualifiedName().equals(
            OperationImpl.createFullyQualifiedName(operationMetaClass, operationName,
            operationParameterTypes)))
9     {
10        selectedAdaptationPolicy = adaptationPolicy;
11        selectedAdaptationPolicy.run();
12    }
13    if(selectedAdaptationPolicy == null)
14    {
15        System.err.println("AdaptationSemanticsImpl.java(run): Unable to find an adaptation
            policy for the " + OperationImpl.createFullyQualifiedName(operationMetaClass,
            operationName, operationParameterTypes) + " operation.");
16        System.exit(1);
17    }
18 }

```

Listing 6.8 – Le code *Java/EMF* de la méthode `run` de la classe `AdaptationSemanticsImpl`

### 6.3.2 Phase 2 : exécution d'une politique d'adaptation

La réalisation d'une politique d'adaptation pour adapter une méthode consiste à suivre l'ordre dans lequel les pas d'orchestration sont définis à l'intérieur de cette politique. Ainsi, nous commençons par sélectionner le premier pas d'orchestration de la politique d'adaptation pour qu'il soit le pas d'orchestration courant (cf. ligne 3 du listing 6.9). Ensuite, nous faisons avancer ce pas d'orchestration courant à l'aide d'une boucle jusqu'à atteindre le dernier pas d'orchestration. Pour faire avancer le pas d'orchestration courant, nous appelons la méthode `next` définie dans la classe abstraite générée `StepImpl`.

```

1 public void run()
2 {
3     this.setCurrentStep(this.getStart().getFirstStep());
4     while(!this.getCurrentStep().equals(this.getEnd().getLastStep()))
5         this.getCurrentStep().next();
6     this.getCurrentStep().next();
7 }

```

Listing 6.9 – Le code *Java/EMF* de la méthode `run` de la classe `AdaptationPolicyImpl`

La classe `StepImpl` étant abstraite, sa méthode `next` l'est également. Pour s'intéresser à l'implémentation de cette méthode `next`, nous devons donc étudier les sous-classes de la classe `StepImpl` : `SimpleStepImpl` et `DecisionStepImpl`. Le code de la méthode `next` dans la classe `SimpleStepImpl` consiste à invoquer dynamiquement la méthode associée au pas d'orchestration, puis de mettre à jour le pas d'orchestration courant de la politique d'adaptation (cf. listing 6.10). Le code de cette méthode au sein de la classe `DecisionStepImpl`, quant à lui, est légèrement plus complexe puisque la valeur de retour booléenne de la méthode invoquée dynamiquement est cette fois considérée pour décider du prochain pas d'orchestration courant (cf. listing 6.11). Concernant cette invocation dynamique, elle est réalisée par l'intermédiaire de la méthode `dynamicCall` disponible dans la classe `OperationImpl`.

```

1 @Override public void next()
2 {
3     this.getOperation().dynamicCall();
4     this.getAdaptationPolicy().setCurrentStep(this.getNextStep());
5 }

```

Listing 6.10 – Le code *Java/EMF* de la méthode `next` de la classe `SimpleStepImpl`

```

1 @Override public void next()
2 {
3     if((Boolean) this.getOperation().dynamicCall())
4         this.getAdaptationPolicy().setCurrentStep(this.getIfTrueStep());
5     else
6         this.getAdaptationPolicy().setCurrentStep(this.getIfFalseStep());
7 }

```

Listing 6.11 – Le code *Java/EMF* de la méthode `next` de la classe `DecisionStepImpl`

La première étape pour réaliser l'invocation dynamique d'une méthode consiste à récupérer l'objet qui contient cette méthode (cf. ligne 3 du listing 6.12). Lorsque cette instance de classe est récupérée,

nous créons une variable qui contient le type des paramètres de la méthode et une autre qui contient les valeurs de ces paramètres. La première variable nous sert au moment de retrouver la méthode à invoquer. Dans le cas où la méthode correspond à une opération d'exécution, le suffixe `Adapted` est ajouté pour correspondre à la convention de nommage expliquée au début de cette section. La deuxième variable, quant à elle, nous est utile quand vient le moment d'invoquer ladite méthode. En effet, c'est à cet instant que les valeurs des paramètres doivent être passées. Une fois la méthode invoquée, il ne reste plus qu'à mettre à jour le tableau associatif avec l'étiquette correspondant à la valeur de retour de la méthode et retourner cette valeur de retour (cf. lignes 21 et 22 du listing 6.12).

```

1 public Object dynamicCall()
2 {
3     Object operationContainer = OrchestrationEngine.get(this.getContainerTag().getName());
4     // Create the parameter types
5     ArrayList<Class<?>> parameterTypes = new ArrayList<Class<?>>();
6     for(Parameter parameter : this.getParameters())
7         parameterTypes.add(Class.forName(parameter.getType().getPackageName() + "." +
8             parameter.getType().getName()));
9     // Create the parameter values
10    ArrayList<Object> parameterValues = new ArrayList<Object>();
11    for(Parameter parameter : this.getParameters())
12        parameterValues.add(OrchestrationEngine.get(parameter.getParameterTag().getName()));
13    // Get the method
14    Method method = null;
15    String methodName = this.getName();
16    if(this instanceof ExecutionOperation)
17        methodName += "Adapted";
18    method = operationContainer.getClass().getDeclaredMethod(methodName, parameterTypes.
19        toArray(new Class[parameterTypes.size()]));
20    // Invoke the method
21    Object result = null;
22    result = method.invoke(operationContainer, parameterValues.toArray());
23    OrchestrationEngine.put(this.getReturnTag().getName(), result);
24    return result;
25 }

```

Listing 6.12 – Le code *Java/EMF* de la méthode `dynamicCall` de la classe `OperationImpl`

### 6.3.3 Un exemple du fonctionnement du moteur d'orchestration

Dans les sous-sections précédentes, nous avons présenté les deux phases du fonctionnement du moteur d'orchestration qui effectuent des appels de méthode. La figure 6.2 montre tous ces appels de méthode réalisés pour l'orchestration de l'adaptation de *SkipAdaptPDL* à travers un diagramme de séquence. D'abord, la méthode `adaptOperation` de la classe `OrchestrationEngine` appelle la méthode `run` de la classe `AdaptationSemanticsImpl`, qui à son tour appelle la méthode `run` de la classe `AdaptationPolicyImpl`,

qui elle-même appelle la méthode `next` de la classe abstraite `StepImpl`, qui appelle enfin la méthode `dynamicCall` de la classe abstraite `OperationImpl`.

En considérant la sémantique d'adaptation spécifiée dans le modèle d'orchestration de la figure 5.6, la politique d'adaptation sélectionnée appelle la méthode `cLateAndSkippable` pour le premier pas d'orchestration. Dans cet exemple, nous prenons le cas où cette vérification d'adaptation retourne « vrai » (c'est-à-dire qu'il est temps d'adapter), ce qui explique pourquoi nous voyons dans le diagramme de séquence que le deuxième pas d'orchestration amène à une invocation dynamique de la méthode `aRemove`. Ensuite, les prochains pas d'orchestration appellent dynamiquement la méthode `cLateAndSkippable` qui retourne « faux », puis la méthode `runAdapted` qui effectue le pas d'exécution.

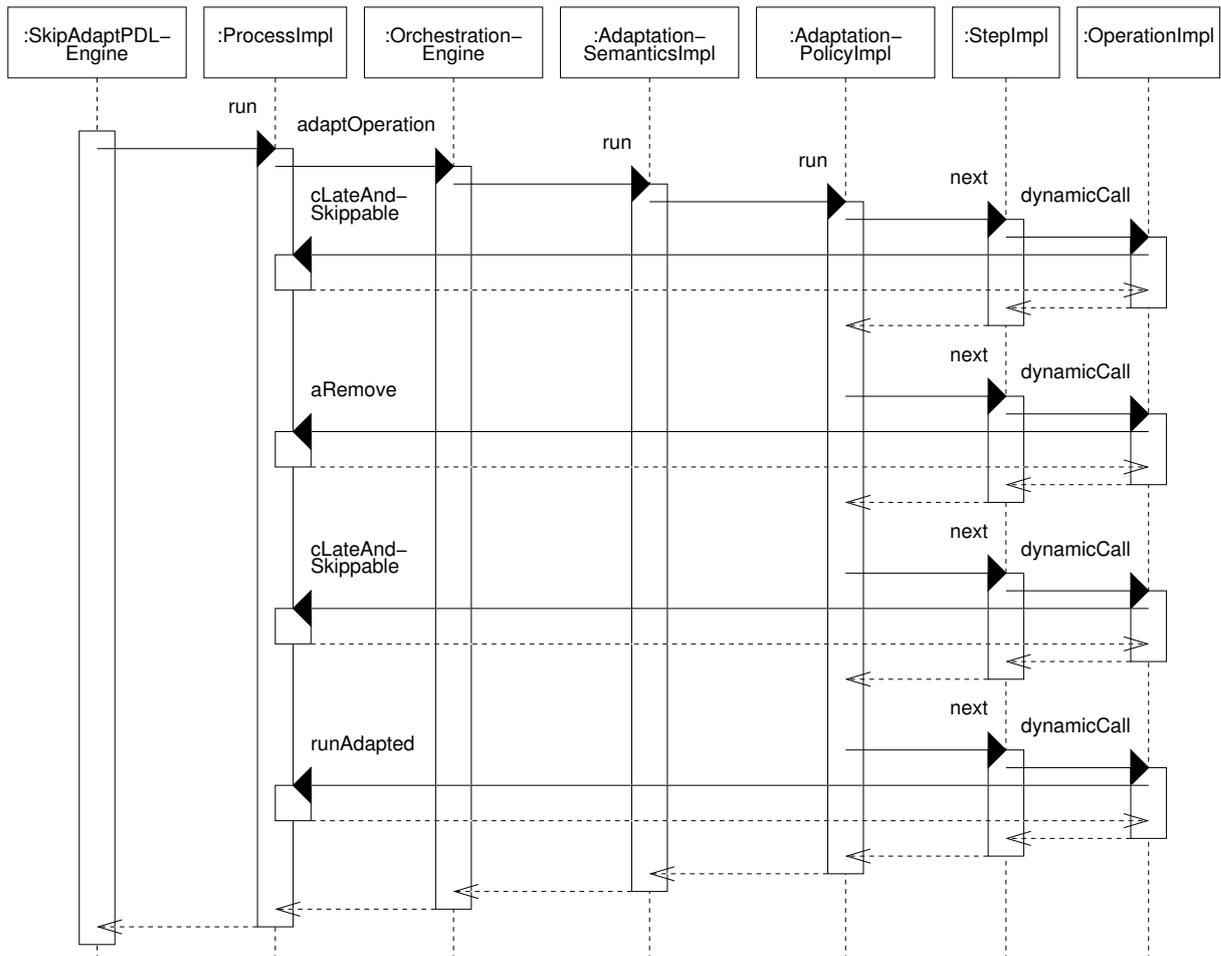


FIGURE 6.2 – Le diagramme de séquence d'un exemple du fonctionnement du moteur d'orchestration

## 6.4 Conclusion

Dans ce chapitre, nous avons montré comment implémenter une application adaptative avec notre approche en utilisant l'environnement *Java/EMF*. Nous avons repris l'exemple d'un modèle adaptable conçu avec l'*i-DSML* adaptable *SkipAdaptPDL* associé à la famille de même nom présentée dans la sous-section 4.3.3. Nous nous sommes servis également d'un modèle d'orchestration conçu avec l'*i-DSML ASDL*

détaillé dans la section 5.2. Nous avons d'abord montré le code du moteur d'exécution en donnant d'une part le corps des méthodes associées aux opérations d'exécution, vérifications d'adaptation et actions d'adaptation et d'autre part le corps de la méthode principale (`main`) qui constitue le point d'entrée du logiciel adaptatif. Cette dernière méthode boucle sur l'appel de la méthode correspondant à l'opération d'exécution afin de réaliser les pas d'exécution.

Ensuite, nous avons expliqué comment coupler le moteur d'exécution au moteur d'orchestration. Une première étape consiste à renommer la méthode correspondant à l'opération d'exécution afin d'en créer une nouvelle qui prend le nom d'origine de la méthode renommée. La deuxième étape consiste à ajouter certaines instructions dans le corps de la méthode `main` afin de permettre la prise en compte du méta-modèle de l'*i-DSML ASDL* par l'environnement *Java/EMF* ainsi que l'initialisation du moteur d'orchestration.

Après cela, nous avons détaillé le fonctionnement du moteur d'orchestration. Il contient les méthodes pour sa propre initialisation (`init`), pour l'accès aux étiquettes ou en ajouter de nouvelles (`get` et `put`) ainsi que pour effectuer l'adaptation d'une méthode (`adaptOperation`). L'appel à la méthode `adaptOperation` permet de donner la main au moteur d'orchestration pour adapter une opération. Nous avons donné les deux phases permettant de mettre en place l'orchestration de l'adaptation d'une opération d'exécution (sachant que les opérations d'exécution correspondent en fait à des méthodes en *Java/EMF*). La méthode `adaptOperation`, appelle `run` sur la sémantique d'adaptation qui appelle à son tour `run` sur les politiques d'adaptation sélectionnées, qui elle-même appelle `next` sur les pas d'orchestration des politiques d'adaptation de la sémantique d'adaptation, qui appelle enfin `dynamicCall` sur l'opération. Cette dernière méthode effectue une invocation dynamique de méthode grâce aux facilités offertes par le langage *Java* en terme de réflexivité. C'est d'ailleurs pour cette propriété de réflexivité dont nous avons besoin que nous avons décidé de choisir une solution basée sur l'environnement *Java/EMF*.

Nous pouvons constater qu'à aucun moment de l'implémentation nous ne nous préoccupons du contenu métier du modèle (cela peut être un processus de développement logiciel ou encore une recette de cuisine). En effet, notre approche étant générique, elle fonctionne pour tout modèle conforme au méta-modèle de l'*i-DSML* adaptable considéré (ici, *SkipAdaptPDL*). Cependant, si nous changeons d'*i-DSML* adaptable nous devons alors modifier le code de notre application adaptative. Toutefois, seule la partie correspondant au moteur d'exécution sera affectée par cette modification, ce qui nous permet de réutiliser la partie correspondant au moteur d'orchestration.

## Chapitre 7

# Conclusion et perspectives

Cela fait cinquante ans que nous savons définir un langage par son arbre syntaxique abstrait, écrire une grammaire *Extended Backus-Naur Form (EBNF)*, parcourir un arbre syntaxique abstrait en mémoire (c'est-à-dire le manipuler), générer du code en sortie, . . . À cet égard, l'IDM n'a pas révolutionné ce qu'il est possible de faire, mais plutôt la façon de le faire. En effet, l'IDM a apporté de la généralisation là où les choses se faisaient essentiellement de manière *ad hoc*. Elle a amené un cadre pour les ingénieurs en logiciel où les différents concepts en jeu ont été désambiguïsés et unifiés, où les préoccupations ont été séparées et où le tout a été parfaitement outillé. Ainsi, avec l'IDM, la création de langages et leur usage productif est devenu moins artisanal et (un peu) plus accessible qu'auparavant.

La situation est aujourd'hui comparable avec les modèles exécutables et la question de leur adaptation. Nous constatons que le domaine des *workflows* par exemple dispose depuis bien longtemps de moteurs d'exécution spécifiques et performants. Bien sûr qu'il est possible d'écrire demain un nouveau moteur d'exécution en *Python* capable de faire tourner des modèles *UML*, en incluant des politiques d'adaptation bien spécifiques. Mais quid de la généralisation de la démarche ? Comment faire en sorte qu'un ingénieur logiciel quelconque puisse, en partant de zéro, concevoir son propre langage de modélisation exécutable et le rendre adaptable en diminuant ses efforts et son risque d'erreur ?

Dans cette thèse, nous avons souhaité apporter une réponse concrète à ces questions grâce à notre approche qui s'inscrit dans la stricte continuité de l'esprit IDM. Les présentes contributions pouvant être prolongées à court ou moyen terme, nous donnons quelques exemples de perspectives dans la section 7.2.

### 7.1 Résumé des contributions

Dans cette thèse, nous avons commencé par caractériser les *i-DSML* adaptables. Puisqu'ils sont une extension des *i-DSML*, ils doivent être de nature exécutable, proposer un moteur et être auto-contenus. La structure de leur méta-modèle est composée de trois parties : les parties structurelles statiques, dynamiques et adaptatives. Leur moteur implémente les sémantiques d'exécution et d'adaptation sous forme opérationnelle. Nous nous sommes efforcés dans cette thèse à définir des politiques d'adaptation dites génériques, en opposition à des politiques d'adaptation dites métier. En effet, les politiques d'adaptation génériques sont capables de s'appliquer sans avoir connaissance du contenu métier du modèle à adapter et sont par conséquent beaucoup plus réutilisables. Néanmoins, nous nous sommes rendu compte qu'il est souvent nécessaire de spécialiser et de contraindre le méta-modèle d'un *i-DSML* pour définir de

telles adaptations génériques. Cette spécialisation apporte avec elle des informations supplémentaires sur lesquelles nous pouvons nous appuyer pour permettre l'adaptation.

Afin d'organiser ces spécialisations, nous avons défini le concept de famille et nous avons conçu le *DSML FHDL* permettant de les spécifier. Une famille rassemble le méta-modèle donné d'un *i-DSML* avec les opérations dédiées à son exécution et à son adaptation que nous nommons « attributs ». Les familles peuvent hériter les unes des autres, nous permettant ainsi de définir des hiérarchies de familles, en partant de la plus générale pour arriver à la plus spécifique. Les attributs, quant à eux, sont soit des opérations d'exécution, soit des vérifications d'adaptation, soit des actions d'adaptation. Ainsi, nous pouvons combiner entre-eux ces attributs contenus dans ces familles afin de construire des politiques d'adaptation. Toutefois, nous avons décidé d'externaliser ces politiques d'adaptation dans un modèle à part au lieu de les coder « en dur » dans le moteur. Ceci permet de pouvoir les remplacer plus facilement par d'autres et offre donc une certaine souplesse à l'application adaptative pour l'ingénieur logiciel.

Pour écrire ces politiques d'adaptation, nous avons créé l'*i-DSML* d'orchestration *ASDL*. Il contient tous les concepts nécessaires à la description d'une sémantique d'adaptation constituée de politiques d'adaptation qui effectuent des pas d'orchestration. Les pas d'orchestration correspondent à des opérations qui sont appelées les unes après les autres par un moteur d'orchestration. Nous nous retrouvons donc avec un logiciel adaptatif constitué de deux moteurs : le moteur d'exécution et le moteur d'orchestration. Cette façon de procéder permet de profiter d'un programme adaptatif semi-générique basé sur la séparation des préoccupations, offrant ainsi une meilleure réutilisation et une maintenance facilitée.

D'un point de vue technique, l'implémentation d'un tel système adaptatif peut être réalisée grâce à l'environnement *Java/EMF*. Les deux modèles sont chargés en mémoire et le moteur d'exécution passe le relais au moteur d'orchestration lorsqu'il est nécessaire de procéder à l'adaptation d'une opération d'exécution. Cette implémentation effectue des invocations dynamiques de méthodes grâce aux facilités offertes par le langage *Java* en terme de réflexivité et s'inscrit complètement dans une approche IDM puisque nous restons dans le cadre d'*EMF*.

## 7.2 Perspectives

Le travail réalisé au cours de cette thèse peut se prolonger à travers plusieurs perspectives qui permettront l'amélioration de notre approche ou l'exploration de nouvelles pistes. Dans cette section, nous allons présenter ces perspectives qui concernent à la fois l'outillage, l'extension du langage *ASDL* et la méta-adaptation.

### 7.2.1 L'outillage

Au niveau de l'ensemble des outils utiles à l'approche proposée dans cette thèse, nous avons d'un côté l'implémentation d'un éditeur de politiques d'adaptation et de l'autre l'implémentation d'un générateur de code qui fournit une automatisation du renommage des opérations à adapter. Dans cette sous-section, nous allons présenter ces deux outils qui faciliteront le travail de l'ingénieur logiciel.

#### 7.2.1.1 L'implémentation d'un éditeur de politiques d'adaptation

Un outil intéressant pour l'ingénieur logiciel qui souhaite concevoir un logiciel adaptatif à l'aide de notre approche serait un éditeur graphique de politiques d'adaptation. Au lancement de cet éditeur,

une interface permettrait de choisir la famille qui nous intéresse. Une fois que l'utilisateur a effectué sa sélection, une palette (cf. figure 7.1, partie droite) s'ouvrirait offrant à l'ingénieur logiciel l'ensemble des attributs associés à cette famille (et à ses super-familles grâce à l'héritage). Ensuite, ces éléments contenus dans cette palette pourraient être glissés vers la zone d'édition (cf. figure 7.1, partie gauche) pour commencer la construction de politiques d'adaptation. En dessinant des liens entre ces attributs dans la zone d'édition, nous pourrions construire progressivement des politiques d'adaptation. Une fois le travail enregistré, il constituerait la sémantique d'adaptation du logiciel adaptatif. Celle-ci prendrait alors la forme d'un modèle, conforme au *i-DSML ASDL*, qui pourrait être interprété par le moteur d'orchestration.

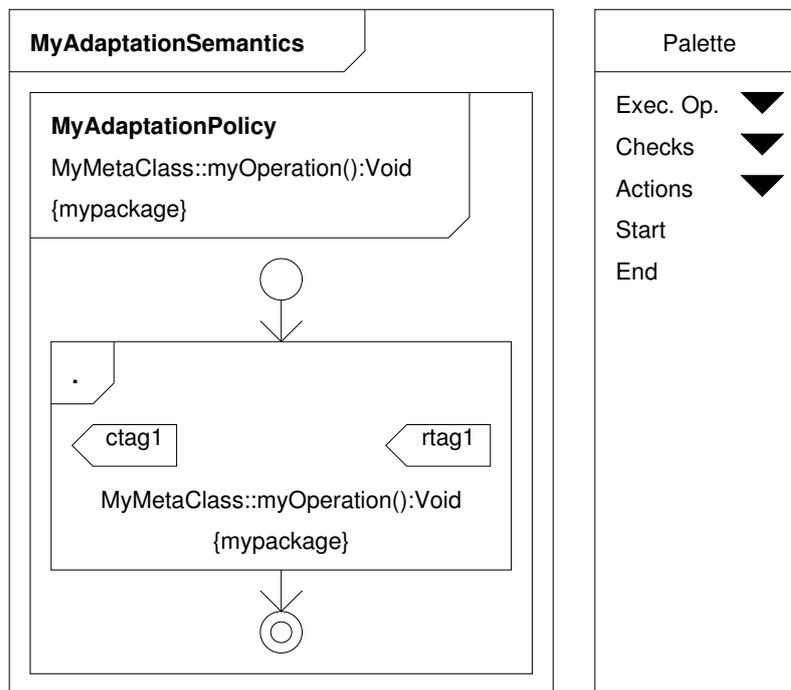


FIGURE 7.1 – Un éditeur de politiques d'adaptation

Pour implémenter un tel outil, nous pourrions envisager de développer une extension pour un environnement de développement intégré tel qu'*Eclipse*. En effet, grâce à son extension destinée à la modélisation graphique (*GMF*<sup>17</sup>), nous pouvons concevoir un éditeur de diagrammes pour un méta-modèle donné. Ainsi, en lui donnant *ASDL* pour méta-modèle, il serait tout à fait envisageable d'éditer des politiques d'adaptation. La difficulté technique réside dans le fait de devoir contraindre l'éditeur à produire seulement des éléments appartenant à la famille sélectionnée.

### 7.2.1.2 L'automatisation du renommage des opérations à adapter

Nous avons vu en section 6.2 que l'ingénieur logiciel doit renommer les méthodes originelles correspondant aux opérations d'exécution à adapter afin d'en créer de nouvelles qui viendront s'y substituer. Ce processus de renommage pourrait être automatisé en modifiant l'extension *Eclipse* qui correspond au générateur de code du projet *EMF*. En effet, la classe `Class` du paquetage `org.eclipse.emf.codegen.`

17. *GMF* : *Graphical Modeling Framework*, <http://eclipse.org/gmf-tooling/>

`ecore.templates.model` contient les chaînes de caractères utiles à la génération des classes à partir du méta-modèle de nos *i-DSML* adaptables. La méthode `generate` contenue dans cette classe s'occupe de la génération du code pour nos classes. En plaçant une condition appropriée (`isAdapted`) dans le corps de cette méthode, nous pourrions automatiser le renommage des méthodes correspondant aux opérations à adapter. Si cette condition est vérifiée alors l'action consiste à ajouter le suffixe `Adapted` au nom de ces méthodes ainsi que de créer les nouvelles méthodes qui ajoutent des étiquettes au tableau associatif et qui appellent le moteur d'orchestration.

## 7.2.2 L'extension du langage *ASDL*

Le langage *ASDL* peut être étendu pour supporter des opérateurs logiques et des politiques d'adaptation composites. Dans cette sous-section, nous allons présenter cette extension qui augmentera le pouvoir d'expression du langage *ASDL* et aidera ainsi l'ingénieur logiciel dans la définition de sa sémantique d'adaptation.

### 7.2.2.1 L'ajout d'opérateurs logiques à *ASDL*

Dans la sous-section 5.4.5, nous avons mis en évidence qu'*ASDL* ne propose pas de concepts correspondant aux opérateurs logiques (comme « ET », « OU » ou encore « NON »). Bien qu'il soit possible de combiner les vérifications d'adaptation entre elles afin d'obtenir l'équivalent d'une conjonction ou d'une disjonction, cela s'avère peu intuitif et l'absence d'opérateurs logiques se fait cruellement ressentir dans notre langage. De plus, ceci ne résout pas le problème de la négation logique qui n'existe tout simplement pas dans *ASDL*. Une solution de contournement que nous proposons en attendant est tout simplement de créer une nouvelle vérification d'adaptation dans le moteur d'exécution qui réalise en interne ces combinaisons.

### 7.2.2.2 Une politique d'adaptation composite

Dans notre approche, chaque politique d'adaptation peut être vue comme la définition d'un flot d'exécution (un début, un ensemble d'opérations à réaliser dans un ordre et une fin). Ainsi, nous avons la présence d'un ordre au sein de chaque politique d'adaptation indiquant qu'une opération doit s'exécuter avant ou après une autre opération. Toutefois, il n'existe pas d'ordre entre les politiques d'adaptation elles-mêmes puisqu'elles sont indépendantes les unes des autres.

Pour prendre un cas où l'ordre entre les politiques d'adaptation a son importance, reprenons la politique d'adaptation `ManagedSkipAdaptPDLRunAdaptationPolicy` de la figure 5.9. Cette politique d'adaptation peut être séparée en deux politiques d'adaptation distinctes : une qui s'occupe d'incrémenter le nombre de sauts tandis que l'autre se charge de changer le chef de projet. Ceci donne naissance à une sémantique d'adaptation alternative où ces deux politiques d'adaptation sont définies. Dans ce nouveau modèle d'orchestration, nous savons que lorsque l'opération `Process::run():Void` est adaptée, ces deux politiques d'adaptation sont sélectionnées successivement par le moteur d'orchestration dans un ordre arbitraire. Or, dans cet exemple nous devons absolument traiter l'incrémentation du nombre de sauts avant le changement du chef de projet. En d'autres termes, la première politique d'adaptation doit être sélectionnée avant la deuxième politique d'adaptation.

Pour définir un tel ordre entre les politiques d'adaptation, nous pouvons définir une politique d'adaptation composite. Celle-ci est destinée à contenir les politiques d'adaptation qui doivent être ordonnées.

Pour spécifier cet ordonnancement, nous nous servons de la même notation que pour les politiques d'adaptation normales qui contiennent des opérations qui doivent être ordonnées (c'est-à-dire un cercle pour le début, une flèche pour l'élément suivant et un double cercle pour la fin). Ces politiques d'adaptation composites offrent l'avantage de pouvoir contrôler plus finement la façon dont est adaptée une opération d'exécution. Ainsi, nous avons deux niveaux de flot d'exécution : celui pour les politiques d'adaptation composites et celui pour les politiques d'adaptation normales.

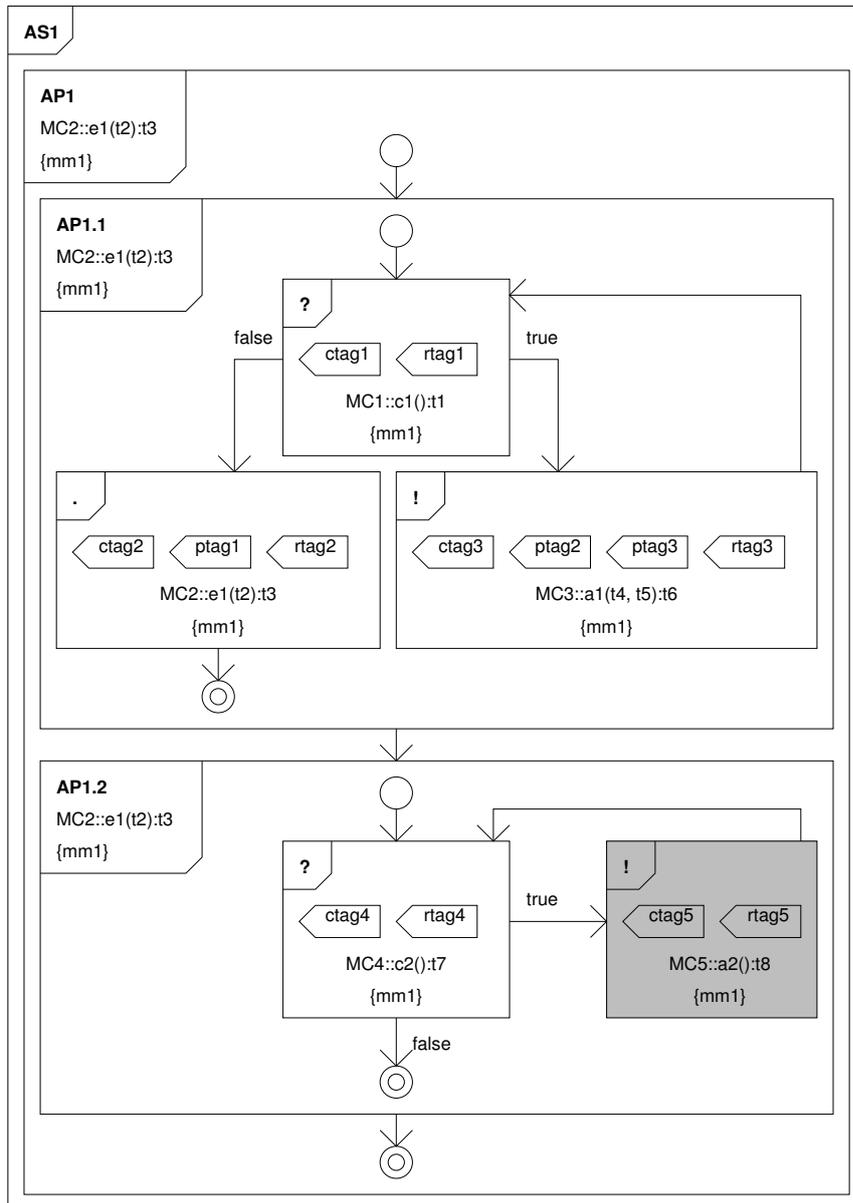


FIGURE 7.2 – Une orchestration de l'adaptation contenant une politique d'adaptation composite

La figure 7.2 montre une sémantique d'adaptation AS1 constituée d'une politique d'adaptation composite AP1 qui elle-même est composée de deux politiques d'adaptation normales (AP1.1 et AP1.2). Ces deux politiques d'adaptation participant à l'adaptation de l'opération d'exécution MC2::e1(t2):t3 contenue dans le paquetage mm1 sont spécifiées de telle manière qu'elles doivent s'enchaîner dans cet ordre : AP1.1,

puis AP1.2.

### 7.2.3 La méta-adaptation

Dans le chapitre 5, nous avons présenté l'*i-DSML ASDL* permettant de spécifier et exécuter une orchestration de l'adaptation. Le moteur d'orchestration est similaire au moteur d'exécution puisqu'il prend en entrée un modèle et effectue des pas. La différence réside dans le fait que ce modèle est issu d'un *i-DSML* d'orchestration au lieu d'un *i-DSML* adaptable et qu'il effectue des pas d'orchestration au lieu de pas d'exécution. Il est alors logique de se demander s'il peut s'avérer intéressant de transformer l'*i-DSML* d'orchestration de l'adaptation en un *i-DSML* adaptable. Si nous effectuons cette transformation, comme expliqué dans le chapitre 3, nous pourrions adapter l'exécution de l'*i-DSML* d'orchestration. Avec cette nouvelle approche, nous avons ainsi deux *i-DSML* adaptables à l'exécution : celui pris en entrée par le moteur d'exécution et celui pris en entrée par le moteur d'orchestration. Ceci aurait pour effet d'accroître les possibilités adaptatives de l'application puisque la sémantique d'adaptation elle-même (c'est-à-dire l'orchestration de l'adaptation) peut être adaptée à l'exécution, ce qui nous amène théoriquement dans une situation où nous réalisons une méta-adaptation (c'est-à-dire que le modèle d'orchestration peut lui-même être modifié durant son exécution, ce qui revient en fait à adapter l'adaptation).

Pour prendre un exemple où la méta-adaptation a une utilité, considérons à nouveau la famille *SkipAdaptPDL* présentée dans la sous-section 4.3.3. Pour l'application adaptative correspondante, nous avons le modèle conforme au méta-modèle *SkipAdaptPDL* ainsi que le modèle d'orchestration de l'adaptation de la figure 5.6. Ces modèles sont interprétés respectivement par le moteur d'exécution et le moteur d'orchestration afin de respecter l'architecture détaillée dans la sous-section 5.1. Bien qu'en l'état actuel, l'*i-DSML ASDL* n'est pas adaptable, nous pouvons trouver dans le code de son moteur une vérification et une action d'adaptation qui sont codées « en dur » aux lignes 13 à 17 du listing 6.8. La vérification consiste à tester s'il n'existe pas de politique d'adaptation pour l'opération d'exécution à adapter tandis que l'action correspond à afficher un message d'erreur et arrêter le logiciel adaptatif. Une autre action pourrait être d'exécuter l'opération à adapter sans effectuer d'adaptation. Ainsi, une fois *ASDL* transformé en *i-DSML* adaptable, nous pouvons construire un modèle d'orchestration de l'adaptation pour *ASDL* lui-même qui sera interprété par un troisième moteur du programme adaptatif (cf. figure 7.3) et permettra de réaliser du *self-healing*. Nous pouvons remarquer que le méta-modèle *ASDL* ainsi que le moteur d'orchestration sont réutilisés puisque notre approche dans cette thèse est générique et fonctionne pour tout *i-DSML* adaptable considéré (*ASDL* compris).

La figure 7.4 montre le modèle d'orchestration de l'adaptation pour *ASDL* que nous pouvons concevoir. Conformément à la convention de nommage expliquée dans la sous-section 5.4.1, cette orchestration de l'adaptation est nommée `ASDLAdaptationSemantics` et contient une seule politique d'adaptation `ASDLRunAdaptationPolicy`. Cette politique d'adaptation définit les trois opérations `AdaptationSemantics::cNoAdaptationPolicy():Boolean`, `AdaptationSemantics::run():Void` (l'opération adaptée) et `AdaptationSemantics::aStop():Void`. Ces opérations n'ont aucun paramètre et sont toutes situées à l'intérieur de la méta-classe `AdaptationSemantics`. La politique d'adaptation `ASDLRunAdaptationPolicy` commence par effectuer une vérification d'adaptation (`AdaptationSemantics::cNoAdaptationPolicy():Boolean`). Si l'opération répond « vrai » alors une action d'adaptation (`AdaptationSemantics::aStop():Void`) sera effectuée, puis la vérification sera de nouveau réalisée et ainsi de suite. Au contraire, si l'opération répond « faux » alors une opération d'exécution (`AdaptationSemantics::run():Void`) sera

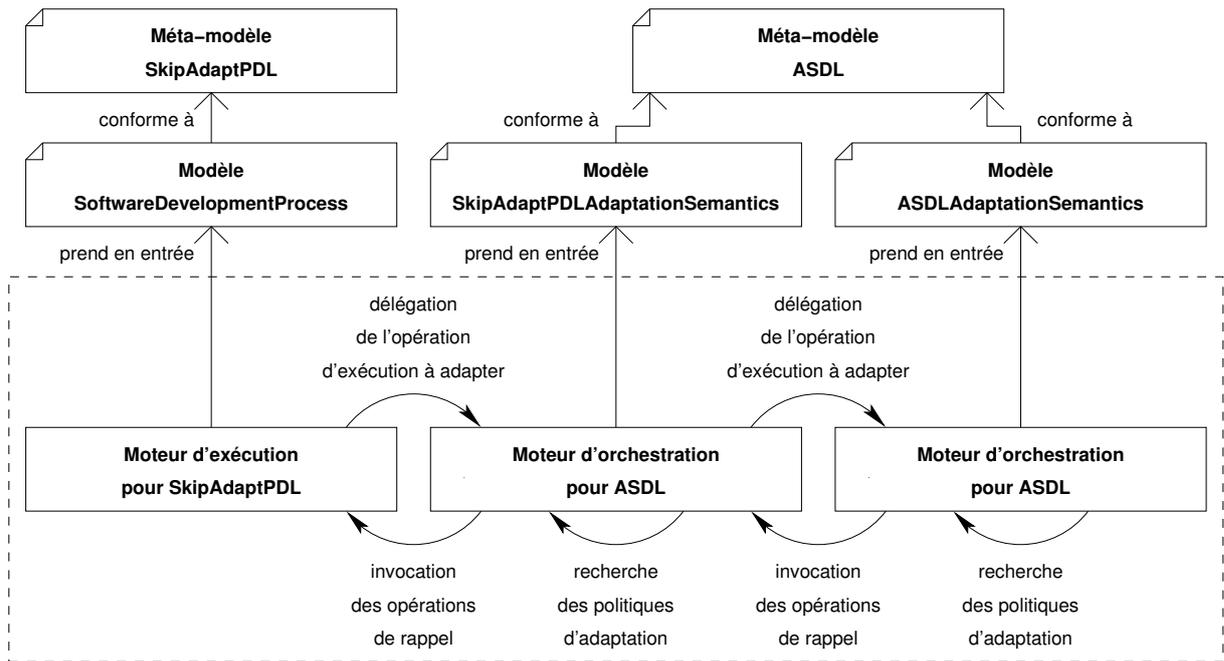


FIGURE 7.3 – Une architecture pour l’adaptation de l’*i-DSML SkipAdaptPDL* avec méta-adaptation

appelée car aucune adaptation n’est nécessaire dans ce cas. Nous pouvons remarquer que le diagramme indique que le pas d’orchestration courant est associé à l’action d’adaptation `AdaptationSemantics::aStop():Void`. Cela signifie que la politique d’adaptation `ASDLRunAdaptationPolicy` a été sélectionnée et par conséquent nous sommes en train d’adapter l’opération d’exécution `AdaptationSemantics::run():Void`. Aussi, nous pouvons facilement en déduire qu’au prochain pas d’orchestration, la vérification d’adaptation `AdaptationSemantics::cNoAdaptationPolicy():Boolean` sera l’opération courante.

Cette orchestration de l’adaptation n’est pas la seule possible pour l’*i-DSML* adaptable *ASDL* et une autre combinaison des opérations est tout à fait envisageable. Par exemple, l’ingénieur logiciel peut choisir une solution moins radicale en préférant l’action d’adaptation `aExecute` à l’action d’adaptation `aStop`. Dans ce cas, il lui suffit de modifier le modèle orchestration de l’adaptation, en remplaçant la boîte associée à l’opération en question. Ainsi, en cas d’absence de politique d’adaptation, lors de l’orchestration de l’adaptation pour *SkipAdaptPDL*, au lieu d’afficher un message d’erreur et d’arrêter l’application adaptative, le comportement correspondant à cette nouvelle orchestration de l’adaptation sera seulement d’exécuter l’opération à adapter sans effectuer d’adaptation.

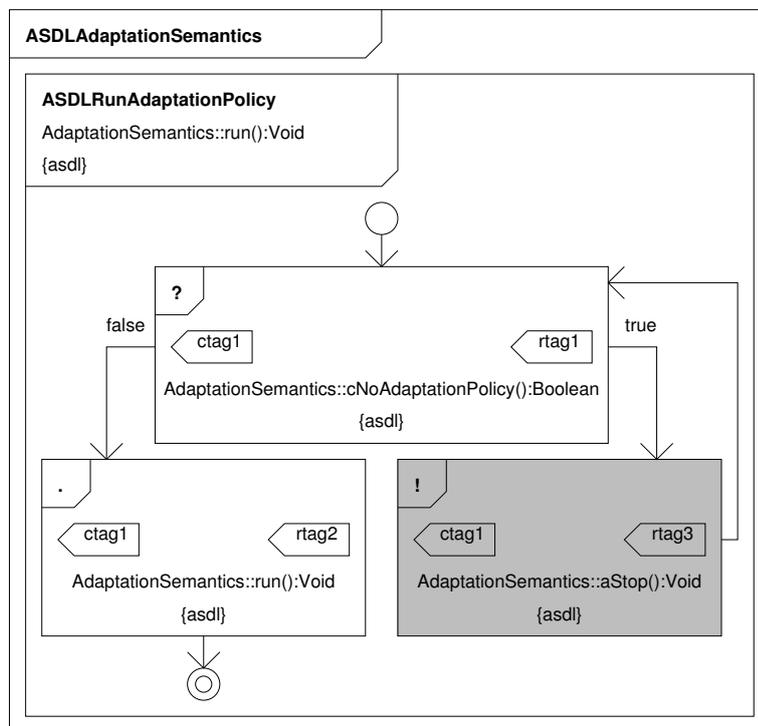


FIGURE 7.4 – Le modèle d’orchestration de l’adaptation pour *ASDL*

# Bibliographie

- [1] W.M.P. van der AALST et A.H.M. ter HOFSTEDE. “YAWL : yet another workflow language”. In : *Information Systems* 30.4. 2005, p. 245–275.
- [2] Francisco Pérez ANDRÉS, Juan de LARA et Esther GUERRA. “Domain Specific Languages with Graphical and Textual Views”. In : *The 3rd international symposium on Applications of Graph Transformations with Industrial relevance (AGTIVE 2007)*. T. 5088. Lecture Notes in Computer Science (LNCS). Springer, 2007, p. 82–97.
- [3] Olivier BALDELLON, Jean-Charles FABRE et Matthieu ROY. “Modeling Distributed Real-Time Systems using Adaptive Petri Nets”. In : *The 1st workshop on Security of System and Software resiliency (3SL)*. 2011, p. 7–8.
- [4] Cyril BALLAGNY, Nabil HAMEURLAIN et Franck BARBIER. “MOCAS : A State-Based Component Model for Self-Adaptation”. In : *The 3rd international conference on Self-Adaptive and Self-Organizing systems (SASO 2009)*. IEEE, 2009, p. 206–215.
- [5] Franck BARBIER, Éric CARIOU, Olivier Le GOAËR et Samson PIERRE. “Software Adaptation : Classification and a Case Study with State Chart XML”. In : *Software* 32.5. IEEE, 2015, p. 68–76.
- [6] Nelly BENCOMO, Gordon S. BLAIR, Carlos A. FLORES-CORTÉS et Peter SAWYER. “Reflective Component-based Technologies to Support Dynamic Variability”. In : *The 2nd international workshop on Variability Modelling of software-intensive Systems (VaMoS 2008)*. Patrick Heymans et al., 2008, p. 141–150.
- [7] Nelly BENCOMO, Paul GRACE, Carlos A. FLORES-CORTÉS, Danny HUGHES et Gordon S. BLAIR. “Genie : Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems”. In : *The 30th International Conference on Software Engineering (ICSE 2008)*. Association for Computing Machinery (ACM), 2008, p. 811–814.
- [8] Gordon S. BLAIR, Nelly BENCOMO et Robert B. FRANCE. “Models@run.time”. In : *Computer* 42.10. IEEE, 2009, p. 22–27.
- [9] Erwan BRETON et Jean BÉZIVIN. “Towards an Understanding of Model Executability”. In : *The international conference on Formal Ontology in Information Systems (FOIS)*. T. 2001. Association for Computing Machinery (ACM), 2001, p. 70–80.
- [10] Licia CAPRA. “Reflective Mobile Middleware for Context-Aware Applications”. Thèse de doct. University of London, 2003.

- [11] Éric CARIOU, Cyril BALLAGNY, Alexandre FEUGAS et Franck BARBIER. “Contracts for Model Execution Verification”. In : *The 7th European Conference on Modelling Foundations and Applications (ECMFA 2011)*. T. 6698. Lecture Notes in Computer Science (LNCS). Springer, 2011, p. 3–18.
- [12] Éric CARIOU, Olivier Le GOAËR et Franck BARBIER. “Model Execution Adaptation ?” In : *The 7th international workshop on Models@Run.Time (MRT 2012) at the 15th international conference on Model Driven Engineering Languages and Systems (MoDELS 2012)*. Association for Computing Machinery (ACM), 2012.
- [13] Éric CARIOU, Olivier Le GOAËR, Franck BARBIER et Samson PIERRE. “Characterization of Adaptable Interpreted-DSML”. In : *The 9th European Conference on Modelling Foundations and Applications (ECMFA 2013)*. T. 7949. Lecture Notes in Computer Science (LNCS). Springer, 2013, p. 37–53.
- [14] Éric CARIOU et Mohamed GRAIET. “Contrats pour la vérification d’adaptation d’exécution de modèles”. In : *La 1ère Conférence en Ingénierie du Logiciel (CIEL 2012)*. 2012.
- [15] Carlos CETINA, Pau GINER, Joan FONS et Vicente PELECHANO. “Autonomic Computing through Reuse of Variability Models at Runtime : The Case of Smart Homes”. In : *Computer* 42.10. IEEE, 2009, p. 37–43.
- [16] Shang-Wen CHENG. “Rainbow : Cost-Effective Software Architecture-Based Self-Adaptation”. Thèse de doct. Carnegie Mellon University, 2008.
- [17] Shang-Wen CHENG, David GARLAN, Bradley R. SCHMERL, João Pedro SOUSA, Bridget SPITNAGEL et Peter STEENKISTE. “Using Architectural Style as a Basis for System Self-repair”. In : *The 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA 2002) : System Design, Development and Maintenance at the 17th International Federation for Information Processing (IFIP) world computer congress*. T. 224. Kluwer, 2002, p. 45–59.
- [18] Peter J. CLARKE, Yali WU, Andrew A. ALLEN, Frank HERNANDEZ, Mark ALLISON et Robert B. FRANCE. “Towards Dynamic Semantics for Synthesizing Interpreted DSMLs”. In : *Formal and Practical Aspects of Domain-Specific Languages : Recent Developments*. Sous la dir. de Marjan MERNIK. IGI Global, 2013. Chap. 9. ISBN : 978-1-4666-2092-6.
- [19] Benoît COMBEMALE, Xavier CRÉGUT, Pierre-Loïc GAROCHE et Xavier THIRIOUX. “Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification”. In : *Journal of Software (JSW)* 4.9. Academy Publisher, 2009, p. 943–958.
- [20] Benoît COMBEMALE, Xavier CRÉGUT et Marc PANTEL. “A Design Pattern to Build Executable DSMLs and associated V&V tools”. In : *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. T. 1. IEEE, 2012, p. 282–287.
- [21] Keling DA. “Kalimuch-A : Autonomic Knowledge-Based Context-Driven Adaptation Platform”. Thèse de doct. Université de Pau et des Pays de l’Adour (UPPA), 2014.
- [22] John DAINITH. *A Dictionary of Science*. Oxford University Press, 2005. ISBN : 978-0-19-280641-3.
- [23] Pierre-Charles DAVID. “Développement de composants Fractal adaptatifs : un langage dédié à l’aspect d’adaptation”. Thèse de doct. Université de Nantes, 2005.

- 
- [24] Gregor ENGELS, Jan Hendrik HAUSMANN, Reiko HECKEL et Stefan SAUER. “Dynamic Meta-Modeling : A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML”. In : *The 3rd international conference on the Unified Modeling Language (UML 2000)*. T. 1939. Lecture Notes in Computer Science (LNCS). Springer, 2000, p. 323–337.
- [25] Franck FLEUREY et Arnor SOLBERG. “A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems”. In : *The 12th international conference on Model Driven Engineering Languages and Systems (MODELS 2009)*. T. 5795. Lecture Notes in Computer Science (LNCS). Springer, 2009, p. 606–621.
- [26] Jacqueline FLOCH, Svein HALLSTEINSEN, Erlend STAV, Frank ELIASSEN, Ketil LUND et Eli GJØRVEN. “Using Architecture Models for Runtime Adaptability”. In : *Software 23.2*. IEEE, 2006, p. 62–70.
- [27] Robert B. FRANCE et Bernhard RUMPE. “The evolution of modeling research challenges”. In : *Software and Systems Modeling 12.2*. Springer, 2013, p. 223–225.
- [28] Henry Laurence GANTT. *Work, wages, and profits ; their influence on the cost of living*. New York, The Engineering magazine, 1910. ISBN : 978-1-144-54677-7.
- [29] David GARLAN, Robert ALLEN et John OCKERBLOOM. “Exploiting Style in Architectural Design Environments”. In : *SIGSOFT Software Engineering Notes 19.5*. Association for Computing Machinery (ACM), 1994, p. 175–188.
- [30] David GARLAN, Robert T. MONROE et David WILE. “Acme : an architecture description interchange language”. In : *The 7th Center for Advanced Studies Conference (CASCON 1997)*. IBM, 1997, p. 7.
- [31] David GARLAN, Bradley SCHMERL et Shang-Wen CHENG. “Software Architecture-Based Self-Adaptation”. In : *Autonomic Computing and Networking*. Sous la dir. d’Yan ZHANG, Laurence Tianruo YANG et Mieso K. DENKO. Springer, 2009, p. 31–55. ISBN : 978-0-387-89827-8.
- [32] Dimitrios GEORGAKOPOULOS, Mark F. HORNICK et Amit P. SHETH. “An Overview of Workflow Management : From Process Modeling to Workflow Automation Infrastructure”. In : *Distributed and Parallel Databases 3.2*. Springer, 1995, p. 119–153.
- [33] John C. GEORGAS, André van der HOEK et Richard N. TAYLOR. “Using Architectural Models to Manage and Visualize Runtime Adaptation”. In : *Computer 42.10*. IEEE, 2009, p. 52–60.
- [34] Dagan GILAT. “Autonomic Computing”. In : *Disappearing Architecture*. Sous la dir. de Georg FLACHBART et Peter WEIBEL. Springer, 2005, p. 32–40. ISBN : 978-3-7643-7275-0.
- [35] Olivier Le GOAËR, Franck BARBIER, Éric CARIOU et Samson PIERRE. “Android Executable Modeling : Beyond Android Programming”. In : *The 2nd international conference on Future internet of things and Cloud (FiCloud 2014)*. IEEE, 2014, p. 411–414.
- [36] Clément GUY, Benoît COMBEMALE, Steven DERRIEN, Jim STEEL et Jean-Marc JÉZÉQUEL. “On Model Subtyping”. In : *The 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*. T. 7349. Lecture Notes in Computer Science (LNCS). Springer, 2012, p. 400–415.
- [37] David HAREL. “Statecharts : A visual Formalism for Complex Systems”. In : *Science of Computer Programming 8.3*. Elsevier, 1987, p. 231–274.

- [38] David HAREL, Asaf KLEINBORT et Shahar MAOZ. “S2A : A Compiler for Multi-modal UML Sequence Diagrams”. In : *The 10th European joint conferences on Theory And Practice of Software (ETAPS 2007), Fundamental Approaches to Software Engineering (FASE 2007)*. T. 4422. Lecture Notes in Computer Science (LNCS). Springer, 2007, p. 121–124.
- [39] Edzard HOFIG. “Interpretation of Behaviour Models at Runtime : Performance Benchmark and Case Studies”. Thèse de doct. Berlin Institute of Technology, 2011.
- [40] IBM. *An architectural blueprint for autonomic computing, third edition*. Rapp. tech. International Business Machines (IBM) corporation, juin 2005. URL : <http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf>.
- [41] Frédéric JOUAULT et Jean BÉZIVIN. “KM3 : A DSL for Metamodel Specification”. In : *The 8th IFIP international conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*. T. 4037. Lecture Notes in Computer Science (LNCS). Springer, 2006, p. 171–185.
- [42] David JUNGER. “Transforming a State Chart at Runtime”. In : *The engineering interactive systems with SCXML workshop at the 6th ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS 2014)*. Association for Computing Machinery (ACM), 2014.
- [43] Dimka KARASTOYANOVA, Alejandro HOUSPANOSSIAN, Mariano CILIA, Frank LEYMAN et Alejandro P. BUCHMANN. “Extending BPEL for Run Time Adaptability”. In : *The 9th IEEE international Enterprise Distributed Object Computing conference (EDOC 2005)*. IEEE, 2005, p. 15–26.
- [44] Jeffrey KEPHART et David CHES. “The Vision of Autonomic Computing”. In : *Computer* 36.1. IEEE, 2003, p. 41–50.
- [45] Grzegorz LEHMANN, Marco BLUMENDORF, Frank TROLLMANN et Sahin ALBAYRAK. “Meta-Modeling Runtime Models”. In : *The 5th international workshop on Models@Run.Time (MRT 2010) at the 13th international conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*. T. 6627. Lecture Notes in Computer Science (LNCS). Springer, 2010, p. 209–223.
- [46] Leonardo A. F. LEITE, Gustavo Ansaldi OLIVA, Guilherme M. NOGUEIRA, Marco Aurélio GEROSA, Fabio KON et Dejan S. MILOJICIC. “A systematic literature review of service choreography adaptation”. In : *Service Oriented Computing and Applications* 7.3. Springer, 2013, p. 199–216.
- [47] Tanja MAYERHOFER, Philip LANGER et Manuel WIMMER. “Towards xMOF : Executable DSMLs based on fUML”. In : *The 12th workshop on Domain-Specific Modeling (DSM 2012) at the 3rd annual conference on Systems, Programming, Languages and Applications : Software for Humanity (SPLASH 2012)*. Association for Computing Machinery (ACM), 2012, p. 1–6.
- [48] Marjan MERNIK, Jan HEERING et Anthony M. SLOANE. “When and how to develop domain-specific languages”. In : *Computing Surveys* 37.4. Association for Computing Machinery (ACM), 2005, p. 316–344.
- [49] Mirjam MINOR, Ralph BERGMANN et Sebastian GÖRG. “Case-based adaptation of workflows”. In : *Information System* 40. Elsevier, 2014, p. 142–152.
- [50] Payal MITTAL, Abhishek SINGHAL et Abhay BANSAL. “A Study on Architecture of Autonomic Computing-Self Managed Systems”. In : *International Journal of Computer Applications* 92.6. IEEE, 2014, p. 6–9.

- 
- [51] Brice MORIN, Olivier BARAIS, Jean-Marc JÉZÉQUEL, Franck FLEUREY et Arnor SOLBERG. “Models@Run.time to Support Dynamic Adaptation”. In : *Computer* 42.10. IEEE, 2009, p. 44–51.
- [52] Adina D. MOSINCAT et Walter BINDER. “Transparent Runtime Adaptability for BPEL Processes”. In : *The 6th International Conference on Service-Oriented Computing (ICSOC 2008)*. T. 5364. Lecture Notes in Computer Science (LNCS). Springer, 2008, p. 241–255.
- [53] Pierre-Alain MULLER, Franck FLEUREY et Jean-Marc JÉZÉQUEL. “Weaving Executability into Object-Oriented Meta-languages”. In : *The 8th international conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*. T. 3713. Lecture Notes in Computer Science (LNCS). Springer, 2005, p. 264–278.
- [54] OASIS. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Rapp. tech. Organization for the Advancement of Structured Information Standards (OASIS), 11 avr. 2007. URL : <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>.
- [55] OMG. *Action Language for Foundational UML (ALF) Version 1.0.1*. Rapp. tech. Object Management Group (OMG), 1<sup>er</sup> sept. 2013. URL : <http://www.omg.org/spec/ALF/1.0.1/PDF>.
- [56] OMG. *Business Process Model and Notation (BPMN) Version 2.0.2*. Rapp. tech. Object Management Group (OMG), 9 déc. 2013. URL : <http://www.omg.org/spec/BPMN/2.0.2/PDF>.
- [57] OMG. *Common Warehouse Metamodel (CWM) Version 1.1*. Rapp. tech. Object Management Group (OMG), 2 mar. 2003. URL : <http://www.omg.org/spec/CWM/1.1/PDF>.
- [58] OMG. *Meta Object Facility (MOF) Version 2.4.2*. Rapp. tech. Object Management Group (OMG), 3 avr. 2014. URL : <http://www.omg.org/spec/MOF/2.4.2/PDF>.
- [59] OMG. *Model to Text Transformation (M2T) Version 1.0*. Rapp. tech. Object Management Group (OMG), 16 jan. 2008. URL : <http://www.omg.org/spec/MOFM2T/1.0/PDF>.
- [60] OMG. *Object Constraint Language (OCL) Version 2.4*. Rapp. tech. Object Management Group (OMG), 3 fév. 2014. URL : <http://www.omg.org/spec/OCL/2.4/PDF>.
- [61] OMG. *Query/View/Transformation (QVT) Version 1.2*. Rapp. tech. Object Management Group (OMG), 1<sup>er</sup> fév. 2015. URL : <http://www.omg.org/spec/QVT/1.2/PDF>.
- [62] OMG. *Semantics of a Foundational Subset for Executable UML Models (FUML) Version 1.1*. Rapp. tech. Object Management Group (OMG), 6 août 2013. URL : <http://www.omg.org/spec/FUML/1.1/PDF>.
- [63] OMG. *Software & Systems Process Engineering Meta-Model Specification (SPEM) Version 2.0*. Rapp. tech. Object Management Group (OMG), 1<sup>er</sup> avr. 2008. URL : <http://www.omg.org/spec/SPEM/2.0/PDF>.
- [64] OMG. *Unified Modeling Language (UML) Version 2.4.1*. Rapp. tech. Object Management Group (OMG), 6 août 2011. URL : <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>.
- [65] Flávio OQUENDO, Brian WARBOYS, Ronald MORRISON, Régis DINDELEUX, Ferdinando GALLO, Hubert GARAVEL et Carmen OCCHIPINTI. “ArchWare : Architecting Evolvable Software”. In : *The 1st European Workshop on Software Architecture (EWSA 2004)*. T. 3047. Lecture Notes in Computer Science (LNCS). Springer, 2004, p. 257–271.

- [66] Peyman OREIZY, Michael M. GORLICK, Richard N. TAYLOR, Dennis HEIMBIGNER, Gregory JOHNSON, Nenad MEDVIDOVIC, Alex QUILICI, David S. ROSENBLUM et Alexander L. WOLF. “An architecture-based approach to self-adaptive software”. In : *Intelligent Systems and their Applications* 14.3. IEEE, 1999, p. 54–62.
- [67] Peyman OREIZY, Nenad MEDVIDOVIC et Richard N. TAYLOR. “Runtime Software Adaptation : Framework, Approaches, and Styles”. In : *The 30th International Conference on Software Engineering (ICSE 2008)*. Association for Computing Machinery (ACM), 2008, p. 899–910.
- [68] Samson PIERRE, Éric CARIOU, Olivier Le GOAËR et Franck BARBIER. “A Family-Based Framework for i-DSML Adaptation”. In : *The 10th European Conference on Modelling Foundations and Applications (ECMFA 2014)*. T. 8569. Lecture Notes in Computer Science (LNCS). Springer, 2014, p. 164–179.
- [69] Claudia PONS et Gabriel BAUM. “Formal Foundations of Object-Oriented Modeling Notations”. In : *The 3rd International Conference on Formal Engineering Methods (ICFEM 2000)*. IEEE, 2000, p. 101–110.
- [70] Youssef RIDENE, Nicolas BELLOIR, Franck BARBIER et Nadine COUTURE. “A DSML for Mobile Phone Applications Testing”. In : *The 10th workshop on Domain-Specific Modeling (DSM 2010) at the 1st annual conference on Systems, Programming, Languages and Applications : Software for Humanity (SPLASH 2010)*. Association for Computing Machinery (ACM), 2010, 3 :1–3 :6.
- [71] Stefanie RINDERLE, Manfred REICHERT et Peter DADAM. “Correctness criteria for dynamic changes in workflow systems - a survey”. In : *Data & Knowledge Engineering* 50.1. 2004, p. 9–34.
- [72] Matthias ROHR, Marko BOSKOVIC, Simon GIESECKE et Wilhelm HASSELBRING. “Model-driven Development of Self-managing Software Systems”. In : *The 1st international workshop on Models@Run.Time (MRT 2006) at the 9th international conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*. 2006.
- [73] Shazia W. SADIQ, Wasim SADIQ et Maria E. ORLOWSKA. “Pockets of Flexibility in Workflow Specification”. In : *The 20th international conference on conceptual modeling at the Entity-Relationship international conference (ER 2001)*. T. 2224. Lecture Notes in Computer Science (LNCS). Springer, 2001, p. 513–526.
- [74] Wasim SADIQ et Maria E. ORLOWSKA. “On Correctness Issues in Conceptual Modelling of Workflows”. In : *The 5th European Conference on Information Systems (ECIS 1997)*. Cork Publishing Ltd, 1997, p. 943–964.
- [75] Mazeiar SALEHIE et Ladan TAHVILDARI. “Self-adaptive software : Landscape and research challenges”. In : *Transactions on Autonomous and Adaptive Systems (TAAS)* 4.2. Association for Computing Machinery (ACM), 2009, 14 :1–14 :42.
- [76] Mary SHAW. “Comparing Architectural Design Styles”. In : *Software* 12.6. IEEE, 1995, p. 27–41.
- [77] Mary SHAW et Paul C. CLEMENTS. “A Field Guide to Boxology : Preliminary Classification of Architectural Styles for Software Systems”. In : *The 21st annual international COMPuter Software and Applications Conference (COMPSAC 1997)*. IEEE, 1997, p. 6–13.
- [78] Jim STEEL et Jean-Marc JÉZÉQUEL. “On model typing”. In : *Software and Systems Modeling* 6.4. Springer, 2007, p. 401–413.

- 
- [79] David STEINBERG, Frank BUDINSKY, Marcelo PATERNOSTRO et Ed MERKS. *EMF : Eclipse Modeling Framework*. Addison-Wesley, 2009. ISBN : 978-0-321-33188-5.
- [80] Roy STERRITT. “Autonomic computing”. In : *Innovations in Systems and Software Engineering* 1.1. Springer, 2005, p. 79–88.
- [81] Wuliang SUN, Benoît COMBEMALE, Steven DERRIEN et Robert B. FRANCE. “Using Model Types to Support Contract-Aware Model Substitutability”. In : *The 9th European Conference on Modelling Foundations and Applications (ECMFA 2013)*. T. 7949. Lecture Notes in Computer Science (LNCS). Springer, 2013, p. 118–133.
- [82] Thomas VOGEL et Holger GLESE. “Language and Framework Requirements for Adaptation Models”. In : *The 6th international workshop on Models@Run.Time (MRT 2011) at the 14th international conference on Model Driven Engineering Languages and Systems (MoDELS 2011)*. T. 794. CEUR Workshop Proceedings (CEUR-WS.org), 2011, p. 1–12.
- [83] Thomas VOGEL, Andreas SEIBEL et Holger GIESE. “The Role of Models and Megamodels at Runtime”. In : *The 5th international workshop on Models@Run.Time (MRT 2010) at the 13th international conference on Model Driven Engineering Languages and Systems (MoDELS 2010)*. T. 6627. Lecture Notes in Computer Science (LNCS). Springer, 2010, p. 224–238.
- [84] W3C. *State Chart XML (SCXML) : State Machine Notation for Control Abstraction*. Rapp. tech. World Wide Web Consortium (W3C), 29 mai 2014. URL : <http://www.w3.org/TR/scxml>.
- [85] Ji ZHANG et Betty H. C. CHENG. “Model-based development of dynamically adaptive software”. In : *The 28th International Conference on Software Engineering (ICSE 2006)*. Association for Computing Machinery (ACM), 2006, p. 371–380.



# Annexe A

## Annexes

### A.1 La construction du moteur d'exécution d'un *i-DSML*

Nous avons vu dans la sous-section 2.2.2 qu'il existe différentes technologies permettant d'ajouter la sémantique d'exécution à la structure du méta-modèle des *i-DSML*. Le tableau A.1 donne un récapitulatif de ces technologies permettant de construire le moteur d'exécution d'un *i-DSML*. Dans cette section, nous allons présenter chacune de ces techniques et implémenter le moteur d'exécution d'un *i-DSML* en ajoutant cette sémantique d'exécution. Pour cela, nous considérons le méta-modèle d'un feu de circulation permettant de modéliser n'importe quelle signalisation, que ce soit pour les véhicules de la route ou pour les piétons qui souhaitent la traverser. La figure A.1 montre ce méta-modèle. Un feu de circulation (**TrafficLight**) possède une ou plusieurs lampes (**lights**) dont une est la lampe active (**currentLight**). Une lampe (**Light**) est d'une couleur (**color**). Un feu de circulation peut changer de lampe courante (**change**). Pour déterminer la prochaine lampe courante, nous nous servons de l'ordre dans lequel les lampes sont stockées dans le modèle.

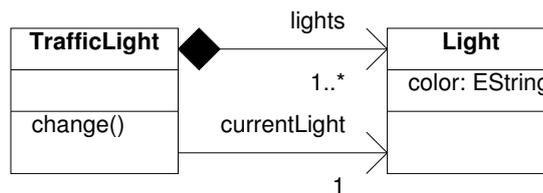


FIGURE A.1 – Le méta-modèle d'un feu de circulation

#### A.1.1 La construction du moteur d'exécution d'un *i-DSML* avec *Kermeta*

Voici les extensions de fichiers utilisées lorsque nous travaillons sur un projet *Kermeta* :

Sémantique d'exécution	Nom	Site Web
Opérationnelle	<i>Kermeta</i>	<a href="http://www.kermeta.org/">http://www.kermeta.org/</a>
	<i>Java/EMF</i>	<a href="http://www.eclipse.org/modeling/emf/">http://www.eclipse.org/modeling/emf/</a>
Translationnelle	<i>ATL</i>	<a href="http://www.eclipse.org/at1/">http://www.eclipse.org/at1/</a>

TABLE A.1 – Les solutions pour construire le moteur d'exécution d'un *i-DSML*

- *\*.kmt* : contient la sémantique d'exécution écrite avec le langage d'action *Kermeta*
- *\*.kp* : contient des méta-données sur le projet *Kermeta*
- *\*.ecore* : contient le méta-modèle conforme au méta-méta-modèle *Ecore*
- *\*.xmi* : contient le modèle conforme au méta-modèle défini dans le fichier *\*.ecore*

Le méta-modèle est enregistré dans un fichier *TrafficLightMM.ecore*. Ensuite, le comportement est enregistré dans un fichier *TrafficLight.kmt*. Le listing A.1 donne ce comportement écrit avec *Kermeta*.

```
1 using kermeta::standard // for Void and Integer
2 using kermeta::io::StdIO => stdio // for stdio
3 using TrafficLightMM // for TrafficLight
4 package TrafficLightMM
5 {
6   aspect class TrafficLight
7   {
8     /**
9      * Changes the current light of a traffic light
10    */
11    operation change() : Void is do
12      var index : Integer init self.lights.indexOf(self.currentLight)
13      if index < self.lights.size() - 1 then
14        self.currentLight := self.lights.at(index + 1)
15      else
16        self.currentLight := self.lights.at(0)
17      end
18    end
19  }
20 }
```

Listing A.1 – Le comportement écrit avec *Kermeta*

Enfin, les modèles conformes à ce méta-modèle sont enregistrés dans des fichiers *StandardTrafficLight.xmi* et *PedestrianTrafficLight.xmi*. Le premier correspond à un feu standard (rouge, vert et orange). Le second correspond à un feu pour piétons (rouge et vert). Les figures A.2 et A.3 montrent ces deux modèles.



FIGURE A.2 – Le modèle du feu standard

### A.1.2 La construction du moteur d'exécution d'un *i-DSML* avec *Java/EMF*

Voici les extensions de fichiers utilisées lorsque nous travaillons sur un projet *Java/EMF* :

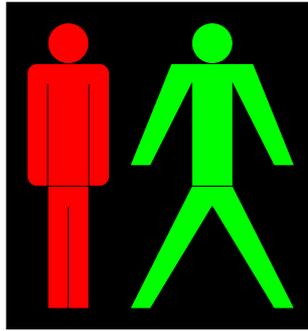


FIGURE A.3 – Le modèle du feu pour piétons

- *\*.java* : contient le code généré écrit avec le langage de programmation *Java*
- *\*.genmodel* : contient le modèle permettant de générer le code *Java* à partir du méta-modèle
- *\*.ecore* : contient le méta-modèle conforme au méta-méta-modèle *Ecore*
- *\*.xmi* : contient le modèle conforme au méta-modèle défini dans le fichier *\*.ecore*

Comme pour l'approche précédente, le méta-modèle est enregistré dans un fichier *TrafficLightMM.ecore* tandis que les modèles sont enregistrés dans des fichiers *StandardTrafficLight.xmi* et *PedestrianTrafficLight.xmi*. Ensuite, grâce à *EMF*, le modèle permettant de générer le code est enregistré dans un fichier *TrafficLightMM.genmodel*. Enfin, le code *Java* généré est complété pour ajouter le comportement désiré. Le listing A.2 donne ce comportement écrit avec *Java/EMF*.

```

129  /**
130   * <!-- begin-user-doc -->
131   * Changes the current light of a traffic light
132   * <!-- end-user-doc -->
133   * @generated NOT
134   */
135  public void change()
136  {
137    int index = this.lights.indexOf(this.currentLight);
138    if(index < this.lights.size() - 1)
139      this.currentLight = this.lights.get(index + 1);
140    else
141      this.currentLight = this.lights.get(0);
142  }
```

Listing A.2 – Le comportement écrit avec *Java/EMF*

### A.1.3 La construction du moteur d'exécution d'un *i-DSML* avec *ATL*

Voici les extensions de fichiers utilisées lorsque nous travaillons sur un projet *ATL* :

- *\*.atl* : contient les règles de transformation écrites avec le langage de transformation *ATL*
- *\*.ecore* : contient le méta-modèle conforme au méta-méta-modèle *Ecore*
- *\*.xmi* : contient le modèle conforme au méta-modèle défini dans le fichier *\*.ecore*

Comme pour les deux approches précédentes, le méta-modèle est enregistré dans un fichier *TrafficLightMM.ecore* tandis que les modèles sont enregistrés dans des fichiers *StandardTrafficLight.xmi* et *PedestrianTrafficLight.xmi*. Ensuite, les règles de transformation sont enregistrées dans un fichier *TrafficLight.atl* pour ajouter le comportement désiré. Le listing A.3 donne ce comportement écrit avec *ATL*.

```
1 module TrafficLight;
2 create OUT: TrafficLightMM from IN: TrafficLightMM;
3 rule trafficlight
4 {
5   from
6     t1 : TrafficLightMM!TrafficLight
7   to
8     t2 : TrafficLightMM!TrafficLight (
9     currentLight <- t1.lights.at(t1.lights.indexOf(t1.currentLight)+1),
10    lights <- t1.lights
11  )
12 }
13 rule light
14 {
15   from
16     l1 : TrafficLightMM!Light
17   to
18     l2 : TrafficLightMM!Light (
19     color <- l1.color
20  )
21 }
```

Listing A.3 – Le comportement écrit avec *ATL*

## A.2 Une étude de quelques moteurs d'exécution

Certaines technologies telles que *PauWare engine*, *Apache Commons SCXML*, *Renew* ou *Tina*, fournissent un moteur d'exécution destiné à exécuter des modèles conformes à un méta-modèle particulier. En effet, les deux premiers outils sont spécifiques à un méta-modèle pour les machines à états tandis que les deux derniers sont dédiés à un méta-modèle pour les réseaux de Petri. Ces solutions sont destinées à être utilisées uniquement avec le méta-modèle imposé qui n'est pas celui d'un *i-DSML* (c'est-à-dire il ne contient pas les parties structurelles requises). Le comportement est intégré dans l'outil, il suffit de s'en servir pour construire le modèle. Le tableau A.2 donne un récapitulatif de ces moteurs d'exécution spécifiques à un méta-modèle. Dans cette section, nous allons présenter chacune de ces technologies en construisant des modèles de feux de circulation qu'elles pourront exécuter.

### A.2.1 La construction d'un modèle pour *PauWare engine*

Une unique extension de fichiers est utilisée lorsque nous travaillons sur un projet *PauWare engine* (*\*.java* : contient la machine à états écrite avec le langage de programmation *Java* et la bibliothèque

Méta-modèle	Nom	Site Web
Machines à états	<i>PauWare engine</i>	<a href="http://www.pauware.com/">http://www.pauware.com/</a>
	<i>Apache Commons SCXML</i>	<a href="http://commons.apache.org/proper/commons-scxml/">http://commons.apache.org/proper/commons-scxml/</a>
Réseaux de Petri	<i>Renew</i>	<a href="http://www.renew.de/">http://www.renew.de/</a>
	<i>Tina</i>	<a href="http://projects.laas.fr/tina/">http://projects.laas.fr/tina/</a>

TABLE A.2 – Des moteurs d’exécution spécifiques à un méta-modèle

*PauWare engine*). Les modèles sont enregistrés dans des fichiers *StandardTrafficLight.java* et *PedestrianTrafficLight.java*. Les figures A.4 et A.5 montrent ces deux modèles. Les machines à états sont représentées par une grande boîte aux bords arrondis, les états par des boîtes similaires mais plus petites et les transitions par des flèches sur lesquelles l’événement est indiqué textuellement. L’état initial, quant à lui, est pointé par un disque noir. Le fonctionnement des machines à états a été expliqué dans la sous-section 2.2.1. Ici, les états correspondent aux différentes couleurs du feu de circulation tandis que les transitions permettent de passer d’une couleur à l’autre. À chaque fois qu’un événement `change` est généré, l’état actif change.

Les listings A.4 et A.5, donnent ces modèles écrits avec *Java* et la bibliothèque *PauWare engine*. À l’intérieur d’une classe *Java*, nous déclarons les états et la machine à états comme des attributs. La classe `AbstractStatechart` permet de déclarer un nouvel état tandis que la classe `AbstractStatechart_monitor` sert à déclarer une nouvelle machine à états. Dans le constructeur de cette classe, nousinstancions ces attributs. La classe `Statechart` permet d’instancier un nouvel état alors que la classe `Statechart_monitor` sert à instancier une nouvelle machine à états. La méthode `inputState` appelée sur les états indique quel est l’état initial de la machine à états. La méthode `fires` appelée sur la machine à états construit une transition et prend trois paramètres :

1. le nom de l’événement associé à la transition
2. l’état source de la transition
3. l’état cible de la transition

*PauWare engine* offre également la possibilité d’attacher un outil de visualisation à la machine à états pour être en mesure de voir graphiquement l’évolution de la machine à états au cours de son exécution. Pour cela, nous devons ajouter un écouteur sur la machine à états en appelant la méthode `add_listener`. Pour permettre à l’utilisateur de générer des événements, une boucle demande à celui-ci de saisir au clavier le nom de l’événement. L’injection d’un événement dans la machine à états se fait par l’intermédiaire de la méthode `run_to_completion`.

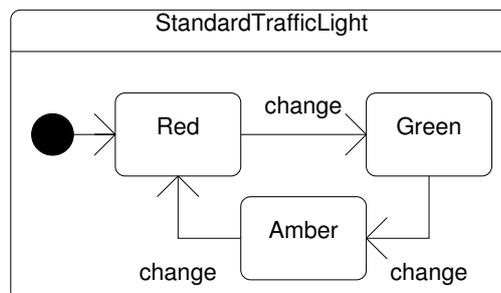


FIGURE A.4 – Le modèle du feu standard (machine à états)

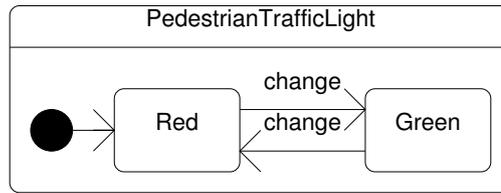


FIGURE A.5 – Le modèle du feu pour piétons (machine à états)

```

9 public class StandardTrafficLight {
10 private AbstractStatechart stateRed, stateGreen, stateAmber;
11 private AbstractStatechart_monitor stateMachine;
12 public StandardTrafficLight() throws Statechart_state_based_exception,
    Statechart_exception {
13 this.stateRed = new Statechart("Red"); // Add states
14 this.stateRed.inputState();
15 this.stateGreen = new Statechart("Green");
16 this.stateAmber = new Statechart("Amber");
17 this.stateMachine = new Statechart_monitor(this.stateRed.xor(this.stateGreen).xor(
    this.stateAmber), "trafficLight");
18 this.stateMachine.fires("change", this.stateRed, this.stateGreen); // Add transitions
19 this.stateMachine.fires("change", this.stateGreen, this.stateAmber);
20 this.stateMachine.fires("change", this.stateAmber, this.stateRed);
21 Statechart_monitor_viewer viewer = new Statechart_monitor_viewer(); // Add a viewer
22 this.stateMachine.add_listener(viewer);
23 this.stateMachine.initialize_listener();
24 viewer.show();
25 String command = ""; // Interact with the user
26 Scanner scanner = new Scanner(System.in);
27 while(!command.equals("exit")) {
28 System.out.println("Type \"change\" or \"exit\":");
29 command = scanner.next();
30 if(command.equals("change"))
31 this.stateMachine.run_to_completion("change");
32 }
33 scanner.close();
34 }
35 public static void main(String[] args) throws Statechart_state_based_exception,
    Statechart_exception {new StandardTrafficLight();}
36 }
  
```

Listing A.4 – Le modèle du feu standard écrit avec *Java/PauWare engine*

```

9 public class PedestrianTrafficLight {
10 private AbstractStatechart stateRed, stateGreen;
  
```

```

11 private AbstractStatechart_monitor stateMachine;
12 public PedestrianTrafficLight() throws Statechart_state_based_exception,
    Statechart_exception {
13     this.stateRed = new Statechart("Red"); // Add states
14     this.stateRed.inputState();
15     this.stateGreen = new Statechart("Green");
16     this.stateMachine = new Statechart_monitor(this.stateRed.xor(this.stateGreen), "
        PedestrianTrafficLight");
17     this.stateMachine.fires("change", this.stateRed, this.stateGreen); // Add transitions
18     this.stateMachine.fires("change", this.stateGreen, this.stateRed);
19     Statechart_monitor_viewer viewer = new Statechart_monitor_viewer(); // Add a viewer
20     this.stateMachine.add_listener(viewer);
21     this.stateMachine.initialize_listener();
22     viewer.show();
23     String command = ""; // Interact with the user
24     Scanner scanner = new Scanner(System.in);
25     while(!command.equals("exit")) {
26         System.out.println("Type \"change\" or \"exit\":");
27         command = scanner.next();
28         if(command.equals("change"))
29             this.stateMachine.run_to_completion("change");
30     }
31     scanner.close();
32 }
33 public static void main(String[] args) throws Statechart_state_based_exception,
    Statechart_exception {new PedestrianTrafficLight();}
34 }

```

Listing A.5 – Le modèle du feu pour piétons écrit avec *Java/PauWare engine*

## A.2.2 La construction d'un modèle pour *Apache Commons SCXML*

Voici les extensions de fichiers utilisées lorsque nous travaillons sur un projet *Apache Commons SCXML* :

- *\*.java* : contient le code, écrit avec le langage de programmation *Java* et la bibliothèque *Apache Commons SCXML*, qui va charger le modèle
- *\*.scxml* : contient le modèle écrit avec le langage de modélisation *SCXML*

Les modèles sont enregistrés dans des fichiers *StandardTrafficLight.scxml* et *PedestrianTrafficLight.scxml*. Ils sont identiques à ceux présentés précédemment et correspondent donc aux figures A.4 ainsi que A.5. Les listings A.6 et A.7, quant à eux, donnent ces modèles écrits avec *SCXML*. La balise racine du document est `<scxml>` et représente la machine à états. Cette balise possède l'attribut `name` indiquant le nom de la machine à états et l'attribut `initialstate` désignant l'état initial de la machine à états. Cette balise contient des éléments `<state>` qui correspondent aux états de la machine à états. Le rôle de l'attribut `id` de la balise `<state>` est de donner l'identifiant unique correspondant à l'état. Chaque balise `<state>`

contient des éléments `<transition>` qui représentent les transitions qui partent de l'état. Les attributs `event` et `target` de cet élément indiquent respectivement le nom de l'événement ainsi que l'état cible associés à la transition.

```
1 <?xml version="1.0"?>
2 <scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initialstate="Red" name="
   StandardTrafficLight">
3 <state id="Red">
4 <transition event="change" target="Green"/>
5 </state>
6 <state id="Green">
7 <transition event="change" target="Amber"/>
8 </state>
9 <state id="Amber">
10 <transition event="change" target="Red"/>
11 </state>
12 </scxml>
```

Listing A.6 – Le modèle du feu standard écrit avec *SCXML*

```
1 <?xml version="1.0"?>
2 <scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initialstate="Red" name="
   PedestrianTrafficLight">
3 <state id="Red">
4 <transition event="change" target="Green"/>
5 </state>
6 <state id="Green">
7 <transition event="change" target="Red"/>
8 </state>
9 </scxml>
```

Listing A.7 – Le modèle du feu pour piétons écrit avec *SCXML*

Ensuite, le code permettant de charger ces modèles est enregistré dans des fichiers *StandardTrafficLight.java* et *PedestrianTrafficLight.java*. Les listings A.8 et A.9 donnent ce code écrit avec *Java* et la bibliothèque *Apache Commons SCXML*. Ces classes *Java* étendent la classe `AbstractStateMachine`. Dans le constructeur de ces classes, nous appelons le constructeur de la super-classe en lui passant en paramètre l'*URL* correspondant au fichier *\*.scxml*. Pour chaque état, une méthode de même nom est définie et décrit l'action à réaliser lorsqu'il est activé. Dans la méthode principale, nous créons une nouvelle instance de la classe et, comme pour *PauWare engine*, une boucle demande à l'utilisateur de saisir le nom de l'événement à générer pour faire avancer d'un pas la machine à états. L'injection d'un événement dans la machine à états se fait par l'intermédiaire de la méthode `fireEvent`.

```
3 public class StandardTrafficLight extends AbstractStateMachine {
4 public void Red() {System.out.println("Red");} // Action methods (for each state)
5 public void Green() {System.out.println("Green");}
6 public void Amber() {System.out.println("Amber");}
```

```

7 public StandardTrafficLight() {super(StandardTrafficLight.class.getClassLoader().
    getResource("StandardTrafficLight.scxml"));} // Constructor
8 public static void main(String[] args) { // Main
9     StandardTrafficLight stateMachine = new StandardTrafficLight();
10    String command = ""; // Interact with the user
11    Scanner scanner = new Scanner(System.in);
12    while(!command.equals("exit")) {
13        System.out.println("Type \"change\" or \"exit\":");
14        command = scanner.next();
15        if(command.equals("change"))
16            stateMachine.fireEvent("change");
17    }
18    scanner.close();
19 }
20 }

```

Listing A.8 – Le code *Java/Apache Commons SCXML* qui charge le modèle du feu standard

```

3 public class PedestrianTrafficLight extends AbstractStateMachine {
4     public void Red() {System.out.println("Red");} // Action methods (for each state)
5     public void Green() {System.out.println("Green");}
6     public PedestrianTrafficLight() {super(PedestrianTrafficLight.class.getClassLoader().
    getResource("PedestrianTrafficLight.scxml"));} // Constructor
7     public static void main(String[] args) { // Main
8         PedestrianTrafficLight stateMachine = new PedestrianTrafficLight();
9         String command = ""; // Interact with the user
10        Scanner scanner = new Scanner(System.in);
11        while(!command.equals("exit")) {
12            System.out.println("Type \"change\" or \"exit\":");
13            command = scanner.next();
14            if(command.equals("change"))
15                stateMachine.fireEvent("change");
16        }
17        scanner.close();
18    }
19 }

```

Listing A.9 – Le code *Java/Apache Commons SCXML* qui charge le modèle du feu pour piétons

### A.2.3 La construction d'un modèle pour *Renew*

Une unique extension de fichiers est utilisée lorsque nous travaillons sur un projet *Renew* (\*.rnw : contient le réseau de Petri écrit à l'aide de l'éditeur fourni). Les modèles sont enregistrés dans des fichiers *StandardTrafficLight.rnw* et *PedestrianTrafficLight.rnw*. Les figures A.6 et A.7 montrent ces deux modèles. Les places sont représentées par des cercles, les transitions par des boîtes, les arcs par des flèches

et les jetons par des petits disques noirs. Le fonctionnement des réseaux de Petri a été expliqué dans la sous-section 2.2.1. Ici, les places correspondent aux différentes couleurs du feu de circulation tandis que les transitions permettent de passer d'une couleur à l'autre.

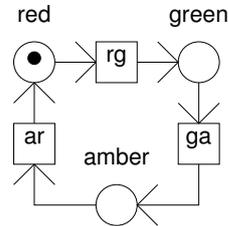


FIGURE A.6 – Le modèle du feu standard (réseau de Petri)

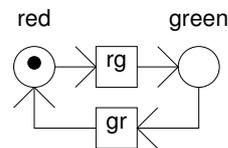


FIGURE A.7 – Le modèle du feu pour piétons (réseau de Petri)

#### A.2.4 La construction d'un modèle pour *Tina*

Une unique extension de fichiers est utilisée lorsque nous travaillons sur un projet *Tina* (\*.ndr : contient le réseau de Petri écrit à l'aide de l'éditeur fourni). Les modèles sont enregistrés dans des fichiers *StandardTrafficLight.ndr* et *PedestrianTrafficLight.ndr*. Ils sont identiques à ceux présentés précédemment et correspondent donc aux figures A.6 ainsi que A.7.

### A.3 Le support du langage *SCXML* pour *PauWare engine*

Dans la section 2.2.1, nous avons signalé l'existence de *SCXML2PauWare* qui est un outil que nous avons développé afin d'ajouter le support du langage *SCXML* pour *PauWare engine*. Ce logiciel se présente sous la forme d'une extension pour *Eclipse* bien qu'il puisse également fonctionner en lignes de commandes. Nous pouvons lui passer un fichier \*.scxml contenant le modèle d'une machine à états écrite avec le formalisme *SCXML*. En parcourant l'arbre du fichier, le programme transforme le modèle vers le langage de programmation *Java/PauWare engine* en construisant progressivement un fichier \*.java. Ce fichier résultat peut alors être exécuté par *PauWare engine*.

Dans l'environnement *Eclipse*, pour demander à *SCXML2PauWare* de transformer un modèle, nous devons d'abord créer un projet *Java*, puis importer le fichier contenant la machine à états. Ensuite, nous pouvons passer soit par la barre d'outils dans laquelle une icône permet de réaliser cette tâche, soit par la barre de menus en choisissant les sous-menus *SCXML2PauWare*, puis *Transform*, soit par le menu contextuel qui s'affiche lors d'un clic droit sur le fichier \*.scxml, soit par le raccourci clavier *Ctrl + T*. En lignes de commandes, il suffit de passer à l'application le fichier à transformer.

*SCXML2PauWare* offre un certain nombre d'options permettant de configurer la façon dont la transformation est réalisée. Nous pouvons ainsi préciser nos préférences concernant la visibilité des classes, des

attributs et des méthodes, le traitement des exceptions et les commentaires. D'autres options permettent également de générer automatiquement une boucle d'interaction avec l'utilisateur, d'ajouter le code permettant de visualiser graphiquement la machine à états (en passant par *PauWare view*) ou encore de créer les méthodes qui correspondent aux événements.



## Résumé

L'un des buts de l'Ingénierie Dirigée par les Modèles (IDM) est de considérer les modèles comme des éléments productifs pour le développement d'applications. Dans cette optique, une nouvelle tendance concerne les modèles exécutables où un modèle produit en phase de conception est réutilisé en tant que tel en phase d'exécution grâce aux *interpreted Domain-Specific Modeling Language (i-DSML)* qui sont interprétés par un moteur d'exécution. Cette façon de procéder permet de gagner du temps lors du développement d'un logiciel et est par conséquent moins coûteux.

D'autre part, les logiciels peuvent être dotés de capacités adaptatives. Ces applications adaptatives sont généralement confrontées à un contexte qui est plus ou moins connu et susceptible de changer au cours de l'exécution et auquel elles vont devoir faire face en modifiant leur comportement dynamiquement, c'est-à-dire sans interruption de service. De telles adaptations dynamiques et automatiques sont censées éviter une phase de maintenance onéreuse pour le logiciel.

Nous avons donc d'un côté les *i-DSML* qui permettent de réduire les coûts de développement d'une application et de l'autre côté l'adaptation logicielle qui permet de réduire les coûts de maintenance d'un programme. Dans cette thèse nous souhaitons prendre le meilleur des deux mondes en fusionnant les deux idées. Le résultat revient *in fine* à directement adapter l'exécution d'un modèle via des *i-DSML* adaptables. Pour cela, nous proposons une caractérisation des *i-DSML* adaptables, la définition du concept de famille pour gérer l'adaptation des *i-DSML*, puis la création d'un langage exécutable d'orchestration dédié à l'adaptation, aboutissant ainsi au fait particulier d'adapter un *i-DSML* par un autre *i-DSML*. Enfin, un prototype à base de deux moteurs d'exécution est proposé avec son implémentation en *Java/EMF*.

**Mots-clés:** IDM, exécution de modèles, adaptation logicielle, *i-DSML*.

## Abstract

One of the goals of Model-Driven Engineering (MDE) is to treat models as productive elements for software development. From this point of view, a new trend is about executable models where a model that is produced at design time is reused as such at runtime through interpreted Domain-Specific Modeling Languages (i-DSMLs) that are interpreted by an execution engine. This way to proceed allows to save time during the software development and consequently is more cost-effective.

On the other hand, software can provide adaptive capabilities. These adaptive applications are often facing a context which is more or less known and which may change during the execution and they will address these various situations by modifying dynamically their own behavior, i.e. without any service disruption. Such dynamic and automatic adaptations ought to avoid a too expensive maintenance stage for the program.

We have on one hand the i-DSMLs that allow to decrease the development costs of a program and on the other hand the software adaptation that allows to decrease the maintenance costs of an application. In this thesis, we must succeed in having the best of both worlds by merging these two ideas. The result is ultimately to directly adapt the model execution through adaptable i-DSMLs. To this end, we propose

a characterization of adaptable i-DSMLs, the definition of the family concept to manage adaptation of i-DSMLs, then the creation of an executable orchestration language for adaptation, thereby leading to the fact that an i-DSML is adapted through an other i-DSML. Finally, a prototype based on two execution engines is proposed with its implementation in Java/EMF.

**Keywords:** MDE, model execution, software adaptation, i-DSML.