A ma Famille

Je tiens à remercier sincèrement :

Marie-Pierre Gervais, Professeur à l'Université Paris X, pour m'avoir dirigé dans mon travail et pour toute l'attention et la patience dont elle a fait preuve durant mon stage.

Xavier Blanc pour son remarquable sens de la critique et sa maîtrise du monde des modèles

Salim Bouzitouna, thésard sous la direction de Marie-Pierre Gervais pour son soutien et ses remarques pertinentes.

Prawee SRIPLAKICH pour sa collaboration dans le cadre de mon stage.

Les partenaires du projet TRAMs : EDF R&D, FT R&D, IRIT et tout particulièrement SofMaint Groupe SODIFRANCE.

Merci à toute l'équipe SRC du LIP6 de m'avoir accueilli et pour la formidable ambiance qui y règne.

Tables des matières :

Chapitre 1 : Introduction

Afin de faire face à un environnement de plus en plus concurrentiel, les grandes et moyennes entreprises cherchent constamment à faire évoluer l'architecture logicielle de leur Système d'Information (S.I) vers des solutions basées sur des technologies nouvelles. Ces évolutions sont conduites comme une reprise des fonctionnalités et des données métier existant dans les applications actuelles, l'entreprise souhaitant réutiliser au mieux son patrimoine logiciel (legacy), en particulier ses données [TRA 01].

Les solutions existantes sur le marché, généralement spécifiques à un type de migration, sont réalisées de manière ad hoc et avec des solutions propriétaires. Elles relèvent plus d'un procédé artisanal qu'industriel et ne proposent pas d'architecture générale et évolutive.

Le projet RNTL **TRAMs** (**T**ransformation **R**eposant sur une **A**rchitecture à base de **M**éta **M**odèles pour la **M**igration des systèmes d'information vers le Web) [RNT 01] qui au cœur de mon stage, vise à réaliser un ensemble de composants logiciels d'aide à la migration de S.I, et ceci dans un cadre architectural (framework) dédié à la réalisation des processus de migration. Ce dernier se devra d'être générique, modulaire et ouvert, indépendant des formalismes de représentation de l'information et des outils support. Il permettra de fédérer les composants logiciels à mettre en œuvre pour réaliser les processus de migration choisis. L'objectif majeur de TRAMs est une réduction importante des coûts et des délais pour les entreprises lors de leurs changements stratégiques, organisationnels ou techniques.

La solution TRAMs repose sur des techniques émergentes qui sont la méta-modélisation et la transformation de modèles.

En effet, celles-ci jouent un rôle de premier plan dans TRAMs, un processus de migration étant décrit comme une succession de transformations à appliquer sur des modèles. Les standards de la méta-modélisation utilisés dans ce projet tel que MOF (*Meta Object Facility*) [OMG 00] offrent un niveau d'abstraction supplémentaire qui favorise la réutilisation des spécifications en facilitant leur interopérabilité, leur fusion et leur transformation. La proposition de systématisation de solutions de migration, concrétisée par le framework ouvert et paramétrable selon la nature de migration, se place comme une avancée significative.

Un processus de migration d'un SI étant décrit comme un ordonnancement d'activités mises en œuvre en vue d'accomplir l'évolution du SI [TRA 02a], le processus-type de migration proposé par Trams se présente comme suit (p2. fig.1.1) :

- Une activité de rétro-modélisation pour analyser le SI source et le modéliser conformément au méta-modèle correspondant,
- Une transformation du modèle source en modèle **Pivot**. Ce dernier étant conforme au **méta-modèle Pivot**
- Une activité d'évolution qui applique l'évolution visée par le processus ; cette activité est facultative (elle n'est par exemple pas nécessaire dans le cas d'une migration de SI pour un changement de langage de programmation). Cette évolution est appliquée sur le modèle pivot.
- Une transformation du modèle pivot en modèle cible
- Une activité de génération de code pour obtenir le SI cible à partir du modèle Cible. Ce dernier étant conforme au **méta-modèle Cible** correspondant.

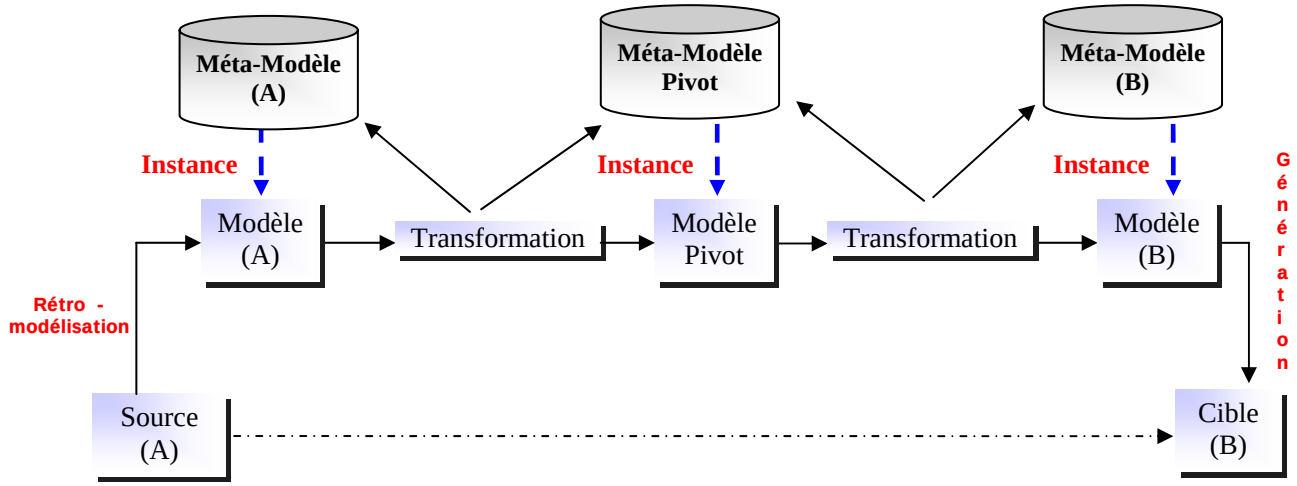


Fig.1.1: Processus de migration selon la proposition TRAMs

Le choix d'un formalisme pivot pour représenter les fonctions et la structure des applications informatiques ainsi que les aspects métiers de celles-ci, se justifie par le fait qu'on peut réduire considérablement le nombre de transformateurs à réaliser en aval (un par type de cible). En effet, lorsque ces transformations se font d'un formalisme vers un autre, i.e. en point à point, si on imagine disposer de n formalismes, il faut alors définir n(n-1) ensembles de règles de transformation. Tandis qu'en passant par un pivot, il n'en faudrait définir que 2n ensembles de règles de transformation. Il s'avère également très coûteux d'ajouter un nouveau formalisme : en effet, passer de n à n+1 formalismes nécessite l'ajout de 2n ensembles de règles, alors qu'il aurait fallu rajouter seulement 2 ensembles de règles de transformation si nous utilisions la technique du pivot [GER 01].

Aussi, les concepts de ce méta-modèle doivent être suffisamment génériques pour être capables de représenter tout ce que les autres formalismes sont capables de représenter. Ils doivent être aussi concrets pour qu'une spécification soit exploitable et utilisable pour générer semi-automatiquement le code correspondant ou pour valider la cohérence de la spécification. Le formalisme pivot doit également être rigoureux, si ce n'est formel.

Ces derniers arguments ont été la motivation principale pour le choix de la norme de traitement réparti ouvert **RM-ODP** (*Reference Model Open Distributed Processing*) [ODP 95][ODP 02], standard de l'**ISO/ITU-T** comme formalisme pivot. Cette norme définit un modèle de référence en proposant un ensemble de concepts et de règles de structuration qui satisfont les critères attendus par un bon formalisme pivot. En effet, les concepts qu'elle définit sont stables et rigoureux. Un grand nombre d'entre eux ont été spécifiés formellement à l'aide de langages tel que Z, LOTOS ou Estelle. De plus, cette norme n'est liée à aucune notation ou méthode, ce qui laisse toute liberté au concepteur dans sa démarche de spécification.

Le modèle architectural ODP est construit sur la notion de **point de vue**. Il définit une architecture par une spécification basée sur une séparation en cinq points de vue. Un point de vue est une subdivision d'une spécification d'un système complexe. Il permet pendant la phase de conception de considérer le système sous un angle particulier en ne s'intéressant qu'à la partie de spécification relative à celui-ci. Chaque point de vue représente une abstraction différente du même système. RM-ODP définit cinq points de vue : (Entreprise :"Le Pourquoi?", Information : "Le Quoi", ***Traitement*** : "Le Comment Fonctionnel", Ingénierie :"Ou" et Technologie : Les choix technologiques d'implantation

Mon stage consiste à spécifier le méta-modèle ***Pivot*** ainsi que les correspondances entre le méta-modèle source vers le méta-modèle pivot et de ce dernier vers le méta-modèle cible. Ces correspondances serviront de base pour l'écriture des règles de transformation qui seront appliquées sur les modèles (source -> pivot ->cible) dans la procédure de migration. Pour cela, nous adoptons une approche incrémentale. En effet, le fait de partir directement du code source, ne nous permet pas de ressortir les différents aspects de l'application i.e. Métier et Informations. Ces derniers étant enfouis dans le code. Ainsi nous nous focalisons dans une première étape, sur le point de vue ***Traitement*** de la norme RM-ODP. Les étapes suivantes consisteront à enrichir cet ensemble de règles par la prise en compte des autres points de vue d'ODP [ODP 95].

Afin de valider notre méta-modèle, nous avons utilisé l'étude de cas réalisée dans le projet TRAMs en 2002, qui consiste à migrer une application Cobol vers Java directement sans utilisation du méta-modèle pivot. Nous avons alors proposé les règles de transformation entre le méta-modèle Cobol et le méta-modèle Traitement de RM-ODP et de ce dernier vers le méta-modèle Java[BEN 03]. Elles reposent sur l'identification des correspondances entre concepts du méta-modèle source Cobol et ceux du méta-modèle Pivot, et du Pivot vers Java. Le méta- modèle Java considéré est celui proposé par NetBeans[NET].

L'analyse d'une première ébauche réalisée a démontrée que seule *l'**Architecture Logicielle*** de notre application (Objets, Méthodes …etc.) a pu être migrée et non les détails d'implémentation (instructions, boucles, variables…etc.).

En effet, étant donné le niveau trop élevé d'abstraction offert par RM-ODP [PUT 01][LEY 97], le méta-modèle du point de vue Traitement spécifié [TRA 02b] ne nous permet pas de spécifier le comportement lié aux objets et par conséquent, n'offre pas la possibilité de faire migrer le détail des instructions reliées aux Opérations ainsi que la sémantique sous jacente.

D'où le besoin qui s'est fait ressentir d'étendre le méta modèle du point de vue Traitement d'RM-ODP de notre pivot afin qu'il autorise la spécification d'un tel niveau de détail, tout en restant standard. L'idée était de trouver une approche qui nous permettrait de formaliser ces instructions, tout en restant standard et indépendant de tout choix d'implémentation.

Action Semantics d'UML [OMG 02] s'est avéré un candidat potentiel qui répondait à ces critères. Action Semantics est une extension à UML et se présente sous forme d'un package nommé ***Actions***. Le standard présente l'avantage d'avoir une sémantique bien fondée permettant une spécification précise du comportement. Par ailleurs Action Semantics offre un niveau d'abstraction qui se révèle être un bon compromis entre les spécifications de haut niveau d'UML et les concepts d'implémentation.

Pour permettre une migration complète de Cobol vers le Pivot, nous avons alors réalisé le couplage de RM-ODP avec Action Semantics.

Une fois notre méta modèle pivot spécifié, il nous restait à reconsidérer le méta modèle cible (Le langage Java) afin qu'à son tour, il puisse aller plus loin que la simple spécification de l'architecture logicielle et prendre en considération les détails des instructions.

Une autre contribution de ce stage a été d'étendre le méta-modèle du langage Java proposé par NetBeans. Pour se faire, nous nous sommes approfondis dans les concepts propres au langage afin d'avoir une spécification la plus complète et la plus fidèle possible permettant ainsi de spécifier tout type d'applications Java.

Les applications de gestion se basant principalement sur une interaction Homme/Machine à travers une interface, il nous a paru primordial de prendre en considération ce point lors du processus de migration. Pour cela, nous avons définit le moyen d'inclure un méta-modèle IHM dans notre méta-modèle Traitement et dans le méta-modèle Java, permettant ainsi une spécification complète incluant l'architecture logicielle, le comportement et l'IHM.

Enfin, une fois les correspondances établies entre les différents méta modèles, pour la phase de transformation de modèles, nous avons utilisé un langage de transformation de modèle nommé Simple TRL [PRA 03] conçu dans le cadre d'un stage au sein mon équipe d'accueil, l'équipe SRC du LIP6. Simple TRL est inspiré de la proposition française TRL[TRL 02] suite à l'RFP (Request For Proposal) MOF 2.0 QVT [QVT 02] de l'OMG.

Le présent rapport est organisé comme suit :

Le chapitre 2 décrit le méta modèle du point de vue Traitement de la norme RM – ODP que nous avons spécifié pour notre Pivot. Nous présenterons les règles de correspondances que nous avons établi entre le MM COBOL, MM Traitement et le MM Java. Nous aborderons ensuite, la problématique à laquelle nous avons été confrontés.

Le chapitre 3 détaille la solution que nous avons proposé afin de palier au problème posé. Nous décriront les principaux concepts d'Action Semantics d'UML qui sont à la base de notre approche et comment nous avons réussi le couplage des deux standards : RM-ODP et Action Semantics pour la spécification d'un méta-modèle du point de vue Traitement complet. Enfin, nous présenterons les nouvelles correspondances que nous avons identifiées entre le MM COBOL et le MM Traitement.

Le chapitre 4 présente le méta-modèle du langage Java que nous avons élaboré pour les besoins du prototype COBOL vers Java. Nous nous sommes basés sur la proposition de NetBeans [NET] que nous avons étendu et ajuster selon nos contraintes. Nous présenterons aussi comment nous avons pu inclure la spécification de l'IHM dans le MM Traitement ainsi que dans le MM Java. Enfin, nous détaillerons les résultats obtenus ainsi que les correspondances nouvellement identifiées entre le méta-modèle Traitement de RM-ODP et le méta-modèle Java

Le chapitre 5 illustre une étude de cas que nous avons réalisé afin de valider les solutions que nous avons proposé. Nous présenterons brièvement le langage Simple TRL qui nous a servi à écrire nos règles de transformation ainsi que des exemples afin de mieux concrétiser les résultats de l'approche suivie.

Le chapitre 6 présente notre conclusion et nos perspectives.

Chapitre 2 : Le point de vue Traitement de RM-ODP

La migration de l'aspect ***métier*** des applications COBOL vers le Web, repose sur l'identification des règles de correspondance entre le méta-modèle *Cobol* et le méta-modèle *Java* en passant par le méta-modèle ***RM-ODP***. Pour cela, nous adoptons une approche incrémentale. Ainsi nous nous focalisons dans une première étape, sur le point de vue ***Traitement*** de la norme RM-ODP. Les étapes suivantes consisteraient à enrichir cet ensemble de règles par la prise en compte des autres points de vue d'ODP. Ce point de vue fait partie des cinq points de vue définis par la norme RM-ODP [ODP 95] que nous avons choisi comme pivot pour notre démarche de migration. Le choix de RM-ODP comme pivot s'est basé sur les nombreux avantages qu'elle pourrait nous apporter, tant en terme de stabilité et de rigueur de ses concepts, qu'en le fait qu'elle soit assez générique, et de ce fait, offre un bon niveau d'abstraction, permettant ainsi un passage plus aisé vers d'éventuelles cibles.

Nous commencerons par présenter le méta-modèle du point de vue Traitement de RM-ODP. Nous donnerons les détailles concernant les classes, relations et propriétés propres à ce méta-modèle. Ensuite nous décrirons les correspondances établies entre le MM COBOL vers le MM Traitement et de ce dernier vers le MM JAVA. Enfin, nous conclurons par les problèmes rencontrés lors de la réalisation du prototype COBOL vers Java.

1. Le Méta-Modèle du point de vue Traitement de RM-ODP

Pour la spécification de notre méta-modèle Traitement, nous nous sommes basés sur une proposition qui a été déjà faite au sein de l'équipe SRC du LIP6[CHI 01] que nous avons amélioré. Pour se faire, nous nous sommes approfondis sur le standard RM-ODP afin d'en ressortir les concepts qui nous ont parus essentiels à notre processus de migration

Présentation

Le point de vue **Traitement** s'intéresse à la décomposition fonctionnelle d'un système ODP en objets interagissant à travers leurs interfaces. Il fournit les concepts relatifs à un modèle de programmation répartie abstrait, basé objet, pour décrire la structure et l'exécution d'applications réparties dans un environnement ODP. Dans ce point de vue sont définis un modèle d'interaction, des interfaces de traitement et un modèle de liaison

Le Typage

RM-ODP définit des concepts relatifs au typage. RM-ODP définit le concept de **type** comme étant un prédicat vérifiable sur n'importe quel élément du modèle. Elle définit aussi le concept de **Template** (gabarit) qui correspond à la spécification de propriétés communes à un ensemble d'éléments de façon à ce qu'il soit possible d'en construire un. Les templates et les types sont les deux concepts RM-ODP à traiter des notions de typage. La principale différence est qu'il est possible de construire un élément à partir de son template. Un élement construit à partir d'un template est alors appelé une **instantiation** alors qu'un élément qui satisfait un type est appelé **instance**.

1.3. Le modèle d'interaction

Dans le point de vue Traitement, les interactions entre objets sont essentiellement asynchrones, reflétant en cela les hypothèses sur la nature des environnements répartis (absence d'horloge globale, d'état global, délais arbitraires des communications, possibilité de défaillance, etc.). Une interaction peut être de la forme suivante :

- Une **opération** est similaire à une procédure et est invoquée à une interface désignée. L'opération reflète le paradigme client-serveur. C'est une interaction entre un objet client et un objet serveur par laquelle l'objet client demande l'exécution d'une fonction par l'objet serveur. Il y a deux types d'opérations :

  - L'**interrogation** est une fonction qui retourne un résultat. Une interrogation peut être synchrone (bloquante) au sens où le client est suspendu en attente du résultat ou asynchrone (non bloquante) ;
  - L'**annonce** est une invocation d'opération sans retour de résultat. Elle est non bloquante.
- Un **flux** est une abstraction de séquences d'interactions aboutissant à l'acheminement d'informations d'un objet producteur (source des informations) vers un objet consommateur (collecteur des informations). Par exemple, un flux peut être utilisé pour modéliser le débit d'information audio ou vidéo d'une application multimédia ou d'un service de télécommunication. Ce peut être également un transfert de fichiers. Sous Unix, on pourrait assimiler un flux à une Socket. Un flux est caractérisé par son nom et son type, qui spécifie la nature et le format des données échangées.

- Un **signal** est une interaction atomique élémentaire aboutissant à une communication unilatérale d'un objet initiateur vers un objet répondeur (c'est-à-dire acceptant la communication). Il correspond par exemple à un événement ou une interruption. Sous Unix, on pourrait assimiler le signal à un signal Unix.

1. Les interfaces de traitement

Un type d'interaction appartient à une unique interface de traitement. Il y a donc trois catégories d'interfaces de traitement : interface opération, interface flot et interface signal. Une interface d'une catégorie donnée ne contient que des interactions de cette catégorie. Par exemple, une interface opération est une interface dont toutes les interactions sont des opérations.

La **signature** d'une interface est définie en fonction de la catégorie de l'interface. La seule notation prescrite dans le modèle de référence RM-ODP est celle utilisée pour les signatures d'interface opération, qui sont décrites par un langage de définition d'interface (Interface Definition Language -- IDL) qui est celui de CORBA.

Un objet peut avoir plusieurs interfaces de différentes catégories. Son nombre d'interfaces peut varier. Une spécification de traitement définit un ensemble initial d'objets de traitement et leur comportement. Le comportement d'un objet est décrit par un ensemble d'actions. Une ***Action*** : "*est quelque chose qui ce passe*" [ODP 95].

La configuration change selon que les objets de traitement instancient d'autres objets de traitement ou d'autres interfaces de traitement, exécutent des actions de liaisons, remplissent des fonctions de contrôle sur des objets de liaison et suppriment des interfaces de traitement ou des objets de traitement.

1.3.2. Le modèle de liaison

Les interactions entre interfaces de traitement nécessitent préalablement l'établissement d'une liaison (**binding**) entre elles, c'est-à-dire un chemin de Communication. L'exemple type de liaison statique en programmation est celui de l'éditeur de liens résolvant la référence externe "appel de procédure" dans un programme principal par la procédure en question.

On distingue les liaisons implicites (cf. l'exemple ci-dessus) des liaisons explicites. De plus, pour les liaisons explicites, on distingue les liaisons primitives (entre deux interfaces) des liaisons composites (entre deux ou plus interfaces). Dans ce dernier cas, un objet de liaison est utilisé.

Pour finir, RM-ODP définit le concept de **Contrat d'Environnement** comme étant un ensemble de règles qui s'applique sur un élément.

Nous représentons ci-dessous (figure 2.1) le méta-modèle du point de vue Traitement, où nous exprimons tous les concepts de ce point de vue.

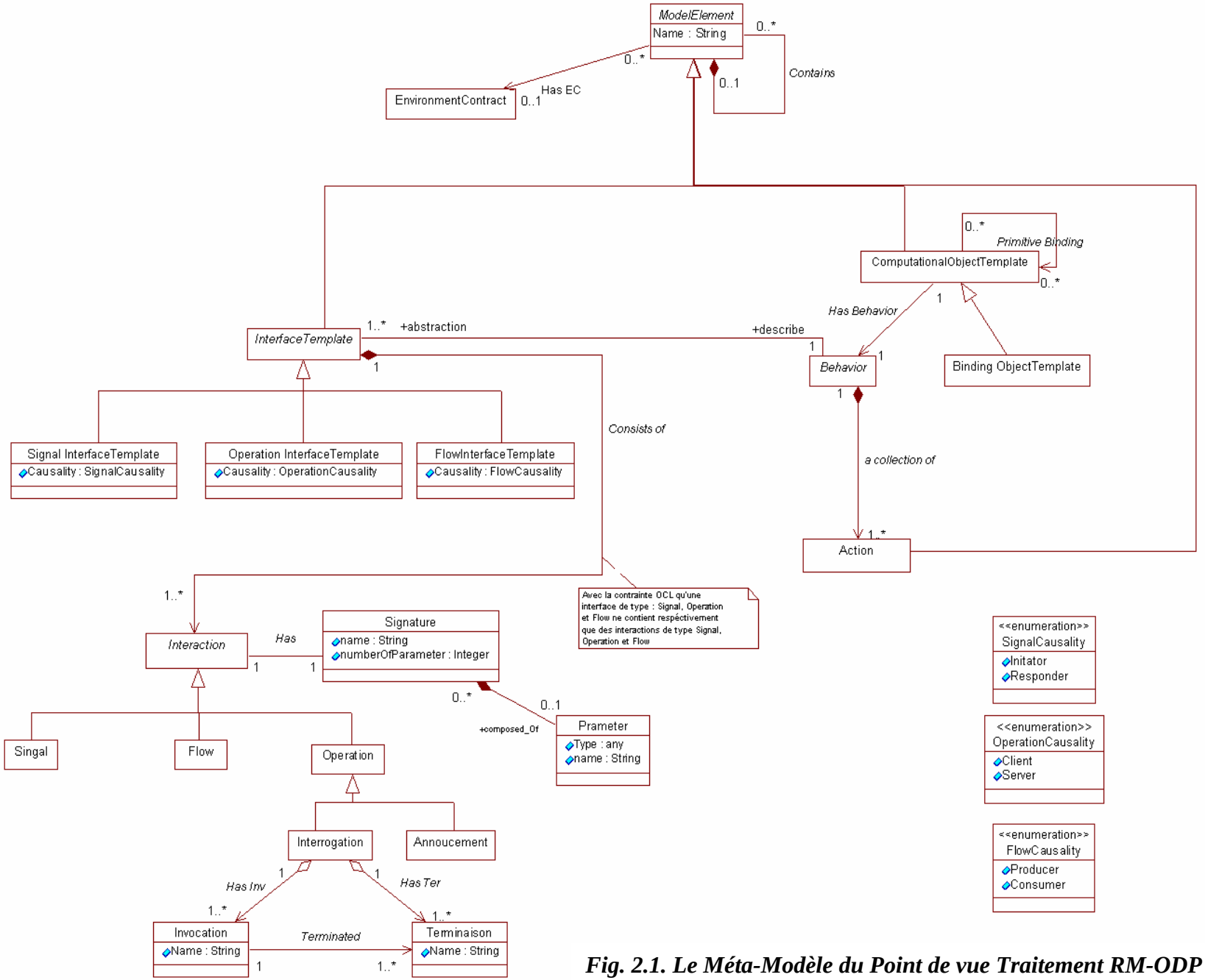


Fig. 2.1. Le Méta-Modèle du Point de vue Traitement RM-ODP

Structure du méta-modèle Traitement

1.4.1. Classes

Les classes du méta-modèle Traitement RM-ODP sont décrites en précisant leurs super-classes, leurs attributs et les références qu'elles possèdent vers d'autres classes comme suit :

ModelElement (abstract class)
Super-classes : Aucune
Attributs : **name** de type string: *le nom de l'élément du modèle*
Références : contractEnvironment de type **ContractEnvironment**: un *contrat d'environnement*

Environnement Contract c'est un Contrat d'environnement qui régie le comportement d'un élément
Super-classes : Aucun
Attributs : Aucun
Références : Aucun

Behavior (abstract class) une collection d'actions qu'un élément doit prendre part en respectant son contrat d'environnement
Super-classes : Aucun
Attributs : Aucun
Références : interfaces de type **Interface** : toutes les interfaces qui représentent l'abstraction du Comportement
actions de type **Action** : le comportement d'un objet est dicté par une collection d'actions

Action "*quelque chose qui se passe* " selon le standard
Super-classes : ModelElement
Attributs : Aucun
Références Aucun

ComputationalObjectTemplate Le gabarit (moule) de l'objet de traitement
Super-classes : ModelElement
Attributs : Aucun
Références : behavior de type **Behavior** : le comportement de l'objet
computationalObjectsTempalte de type **ComputationalObjectTemplate** : Toutes liaisons primitives entreprises avec d'autres objets.

InterfaceTemplate (abstract class) Interface d'objet de traitement
Super-classes : ModelElement
Attributs : Aucun
Références : interactions de type **Interaction** : une Interface de type (Signal, Operation, ou Flow) est composée d'interactions de type (Signal, Operation, ou Flow) respectivement

Signal InterfaceTemplate Le gabarit d'une interface de type Signal

Super-classes : Interface
Attributs : Causality de Type **SignalCausality** : Soit Initiateur, soit Répondeur.
Références : Aucun

Operation InterfaceTemplate Le gabarit d'une interface de type Opération
Super-classes : Interface
Attributs : Causality de Type **OperationCausality** : Soit Client, soit Serveur.
Références : Aucun.

Flow InterfaceTemplate Le gabarit d'une interface de type Flow
Super-classes : Interface
Attributs : Causality de Type FlowCausality : Soit Producteur, soit Consommateur.
Références : Aucun

Interaction (abstract class) Interactions contenues dans une interface
Super-classes : Aucun
Attributs : Aucun
Références signature de type **Signature** : la signature de l'interaction

Signal Une interaction de type Signal
Super-classes : Interaction
Attributs : Aucun
Références Aucun

Operation Une interaction de type Operation
Super-classes : Interaction
Attributs : Aucun
Références Aucun

Flow Une interaction de type Flow
Super-classes : Interaction
Attributs : Aucun
Références Aucun

Interrogation Opération de Type Interrogation
Super-classes : Operation.
Attributs : Aucun.
Références : invocations de Type **Invocation** : les invocations possibles de cette interrogation.
terminaisons de Type **Terminaison** : Les terminaisons possibles de cette interrogation.

Annoucement Opération de type Annonce
Super-classes : Operation
Attributs : Aucun.
Références : Aucun

Invocation (les invocations dans le cas d'une Interrogation)
Super-classes :
Attributs : Name de type String.

Références : terminaisons de Type **Terminaison**: Les terminaisons associées à cette invocation. L'attribut isOrdered est à true pour exprimer l'ordre des ces Terminaisons.

Signature La signature de l'Interaction
Super-classes : Aucun
Attributs : name: nom de l'interaction, numberOfParameter :le nombre de paramètres
Références parameter de type **Parameter** : les paramètres de la signature

Parameter les paramètres de la Signature
Super-classes : Action
Attributs : Type : le type du paramètre, name : son nom
Références Aucun

Terminaison Les terminaisons possibles d'une invocation
Super-classes : Aucun
Attributs : name : le nom de la terminaison
Références Aucun

1.4.2. Types

Les Types définis dans ce méta-modèle sont :

OperationCausality : Enumération qui représente les deux type de causalité pour une opération soit serveur (Server) ou Client(Client).

FlowCausality : Enumération qui représente les deux type de causalité pour un Flow soit Producteur (Producer) ou Consommateur (Consumer).

SignalCausality : Enumération qui représente les deux type de causalité pour un Signal soit initiateur (Initiator) ou Répondeur (Responder).

***String** : le type de base String.

***Int** : le type de base Integer.

1.4.3. Associations

Les associations du méta-modèle traitement sont décrites en précisant les deux classes qu'elles joignent ainsi que les multiplicités associées à celles-ci. Pour les multiplicités, la convention utilisée ici est celle définie par le MOF.

HasEC permet de définir le contrat de l'environnement de l'élément du modèle.
Source : Type : ModelElement
Multiplicité : 0..* (A un Contrat d'environnement peut correspondre plusieurs éléments du modèle)
Destination : Type : EnvironementContract
Multiplicité : 0..1 (Un élément peut être avoir un contrat avec son environnement)

HasBehavior permet de définir le comportement de l'objet.
Source : Type : ComputationalObject
Multiplicité : 1 (A un Comportement correspond un objet)
Destination : Type : Behavior
Multiplicité : 1 (Un objet a un comportement)

Abstraction Une Référence permettant de définir les Interfaces qui représentent l'abstraction du comportement
Source : Type : Behavior.
Multiplicité : 1 (Une Interface est l'abstraction d'un comportement)
Destination : Type : Interface
Multiplicité : 1..* (Le comportement d'un objet peut être décrit par plusieurs Interfaces)

A collection of Une Composition pour dire que le comportement d'un objet est décrit par une collection d'actions
Source : Type : Behavior.
Multiplicité : 1 (Une action est relative à un comportement)
Destination : Type : Interface
Multiplicité : 1..* (Le comportement d'un objet peut être décrit par une ou plusieurs actions)

Consists of Une Composition pour exprimer qu'une interface est composée de plusieurs Interactions
Source : Type : Interface.
Multiplicité : 1 (Une interaction appartient à une interface)
Destination : Type : Interaction
Multiplicité : 1..* (Une interface est composée de plusieurs interactions)

Has Une Référence pour dire qu'une Interaction a une signature
Source: Type : Interaction
Multiplicité : 1 (Une interaction a une seule signature)
Destination : Type : Signature
Multiplicité : 1 (Une signature correspond à une seule interaction)

Has Param Une Composition. Une signature est composée de paramètres
Source: Type : Signature
Multiplicité : 1 (Un paramètre appartient à une signature)
Destination : Type : Parameter
Multiplicité : 1 (Une signature comprend zéro ou plusieurs paramètres)

PrimoitiveBinding Une association réflexive permettant d'exprimer les liaisons primitives entre objets de qui est une liaison directe entre deux Objet sans passer par un Objet de liaison(BindingObject).
Source : Type : ComptationalObject.
Multiplicité : 1(Une Interface appartient à un seul Objet de traitement)
Destination : Type : Interface
Multiplicité : 1..* (un Objet de traitement est composé de plusieurs Interfaces)

HasInv Une agrégation permettant de définir les Invocations d'une Intérrogation.
Source : Type : Interrogation
Multiplicité : 1(Une Invocation est associée à une seule Interrogation)
Destination : Type : Invocation
Multiplicité : 1..* (Une Interrogation peut avoir une ou plusieurs Invocations)

HasTer Une agrégation permettant de définir les Terminaisons d'une Intérrogation.
Source : Type : Interrogation
Multiplicité : 1(Une Terminaison est associée à une seule Interrogation)
Destination : Type : Terminaisaon
Multiplicité : 1..* (Une Interrogation peut avoir une ou plusieurs Terminaisons associées à ses Invocations)

Terminated permettant de définir toutes les Terminaisons possibles d'une Invocation.
Source : Type : Invocation
Multiplicité : 1(Une Terminaison est associée à une seule Invocation)
Destination : Type : Terminaisaon
Multiplicité : 1..* (Une Invocation peut référencer une ou plusieurs Terminaisons associées à ses Invocations)

2. Correspondances entre le MM COBOL / MM Traitement / MM JAVA

L'approche que nous avons choisi dans ce document afin d'établir les règles de correspondance, consiste tout d'abord, en partant du pivot, à mettre en évidence les concepts clés du point de vue Traitement de RM-ODP, les définir, et enfin, identifier au fur et à mesure, leurs correspondants en Java (cible), puis en Cobol (source). Mais avant, nous présenterons les différents méta-modèles cible, source et pivot qui nous ont servi comme base pour notre mapping. Et enfin, le tableau des correspondances entre les méta-modèles COBOL/Traitement/JAVA.

2.1. Les méta-modèles source, pivot et cible

Afin d'établir les règles de correspondances entre source (COBOL) et pivot (Point de vue Traitement de RM-ODP) on s'est basé sur les méta-modèles définis respectivement dans le lots Méta-modèle Cobol [TRA 03] (voir annexe 1) et celui qu'on vient de définir.. Pour le méta-modèle cible (Java), on a utilisé le méta-modèle proposé par la communauté ***netBeans*** (voir annexe 1).

La motivation de netBeans était de permettre l'échange de structures de données entre outils et leur réutilisation par des systèmes plus larges. Les structures de données de ces outils, qui sont définies sous forme de classes et d'interfaces, étaient propriétaires et de ce fait, limitaient la portabilité de celles-ci vers d'autres langages.

L'idée était de créer un méta-modèle permettant la définition de structures de données qui décrivent – suffisamment- un programme écrit en langage Java. netBeans, a choisit le standard MOF (Meta Object Facility) adopté par l'OMG afin de décrire ce méta-modèle.

Tous ces arguments nous ont incité à le choisir comme base pour notre correspondance entre Pivot et cible (Java).

Nous avons choisit de détailler certains concepts définis dans le méta-modèle que nous avons jugé pertinents pour l'établissement de nos règles de correspondance.

La méta classe **JavaClass** peut représenter une *Classe Java* ou une *Interface* selon la valeur de l'attribut booléen *isInterface*, ainsi que des méthodes d'introspection de type : *getXXX()*.

La méta classe abstraite ***ClassMember***, comme son nom l'indique, représente tous les membres, qu'une classe peut contenir : d'autres classes (la notion de classes internes), des attributs définis par la méta classe **Field** ainsi que les éléments décrivant son comportement. Ces derniers sont regroupés sous la méta classe abstraite ***BehavioralFeature***. Cette méta classe englobe les constructeurs d'une classe – la méta classe Constructor - ainsi que les *Méthodes* représentées par la méta classe **Method**. Les méthodes ayant des paramètres, la méta classe **FeatureParameter** les définit. Chaque paramètre a un type, celui ci est décrit dans la méta classe abstraite ***TypeDescriptor*** qui regroupe les différents types possibles. Un paramètre peut être de type primitif (La méta classe ***PrimitiveType***) ou de type non primitif représenté par la méta classe ***ClassDescriptor***.

2.2. Du point de vue Traitement de RM-ODP vers Java

Le point de vue traitement se focalise sur la définition de l'***Architecture Logicielle*** de l'application. Cela se fait par la décomposition d'un système en une collection d'objets qui interagissent via leurs interfaces afin d'assurer les fonctionnalités requises du système. Les concepts relatifs à ce point de vue sont d'un niveau d'abstraction tel, que le passage d'une application d'un système vers un autre se résume à changer sa représentation selon le langage de programmation cible, sans pour autant toucher à l'application elle-même.

Le premier concept clé est celui d'***Objet de traitement***. Chaque objet de traitement représente le modèle d'une entité, possède un état, une identité et un comportement. Il est spécifié à travers son gabarit (***Template***) qui décrit ses caractéristiques et permet son *instanciation*. En Java, un objet a aussi un état, un comportement, une identité et modélise une entité. ***L'objet*** est instancié à partir de sa ***Classe.*** Une Classe d'objets, regroupe les caractéristiques communes à plusieurs objets, qui décrivent la structure et le comportement communs de ces objets.

On peut en déduire les deux premières règles suivantes :

Règle 1 : Un objet de traitement ODP correspond à un objet Java.

Règle 2 : Le Template de l'objet de traitement ODP est équivaut à une Classe en Java.

Un objet de traitement offre des services via des ***Interfaces*** et utilise les services proposés par d'autres objets. Il peut avoir plusieurs interfaces, qui sont les points d'accès à ses services. Une interface a un ***type*** et définit une ou plusieurs ***interactions*** toutes du même type (*Signal*, *Stream* ou *Opération*). Une interaction étant une action entre deux objets. Les interfaces sont définies à travers des ***Templates d'Interface***.

Pour chaque type d'interface, RM-ODP identifie le nom de chaque extrémité de l'interface de l'objet participant aux interactions de celle-ci, ce nom correspond à la ***Causalité*** assumée par l'objet [PUT 01]. Par exemple pour les interfaces de type Opération, les causalités sont *Client* et *Serveur*.

Nous nous intéressons ici tout particulièrement à l'interface de type ***Opération***.

Une interface de type opération peut être soit une ***Annonce*** (sans retour de résultat) ou une ***Interrogation*** (retourne un résultat). Chaque opération est caractérisée par :

- Une ***Signature*** (*nom, arguments* et *résultats*) par laquelle l'opération est invoquée par un utilisateur du service (objet *client*)

- Un corps, qui représente le code exécuté par le fournisseur du service (objet *Serveur*) une fois l'opération invoquée.

Une ***Interrogation*** peut avoir un ou plusieurs résultats et l'objet serveur utilise les ***Terminaisons*** pour répondre aux invocations de l'objet client. Chaque terminaison a un *nom* et une *signature* (liste des arguments de sortie).

En Java et pour tout langage orienté objet, un objet contient des données décrivant son *état*, une identité unique et invariable et des ***Méthodes*** associées afin de guider son comportement. Chaque méthode possède une *Signature* correspondant à son *nom*, *paramètres* et *type* de retour. Le détail d'implémentation des méthodes est privé tandis que leurs signatures sont publiques et apparaissent dans les ***Interfaces***. En Java, une classe peut implémenter plusieurs interfaces.

Règle 3 : Les Templates d'Interfaces Opérationnelles de causalité Serveur dans le point de vue traitement de RM –ODP correspondent aux interfaces en Java.

Règle 4 : Les Templates d'Interfaces Opérationnelles de causalité Client dans le point de vue traitement de RM –ODP n'ont pas de correspondance en Java.

Règle 5 : Les Opérations dans le point de vue traitement de RM –ODP correspondent aux Méthodes en Java.

Dans le point de vue traitement de RM-ODP, une opération peut avoir un ou plusieurs paramètres (arguments). Chaque paramètre est typé. Dans le méta-modèle de traitement RM-ODP le type d'un paramètre est décrit comme étant de type « any » qui peut représenter n'importe quel type. Pareillement en Java, une méthode peut avoir un ou plusieurs paramètres comme elle peut ne pas en avoir du tout. Chaque paramètre a un type qui peut être -comme décrit dans le méta-modèle ci-dessous Fig. 1- soit un *PrimitiveType* (Boolean, Byte, Integer,…etc.), soit *ClassDescriptor* pour un type non primitif.

Règle 6 : Un paramètre et son type dans le point de vue traitement de RM -ODP correspondent aux même concepts en Java.

2.3. De COBOL vers le point de vue Traitement de RM-ODP

Dans les applications de gestion, le COBOL s'est longtemps imposé comme étant le langage le plus adapté, d'ailleurs l'idée de sa création était partie de ce point : concevoir un langage spécialement dédié aux applications des entreprises et des administrations [MOE 82]. Une application COBOL, comme toute autre, a un objectif à atteindre. Cela est assuré par un ou plusieurs programmes interagissant ensemble afin de réaliser des tâches bien particulières. Un ***Programme*** COBOL est composé de 4 parties nommées *divisions* :

- Identification division : Sert à l'identification du programme et à sa documentation;
- Environment division : Description des communications avec les fichiers;
- Data division : Définit la structure des entrées et des sorties ainsi que les zones de travail;
- ***Procedure*** division : Décrit le traitement à effectuer (instructions).

Les trois premières divisions sont plus au moins spécifiques à tout ce qui est informations concernant le programme (nom, auteur…etc.), entrées/sorties ainsi que les structures et les données utilisées (variables, constantes... etc.). En fait, cela correspond à la ***partie Données*** (Data) du programme.

La *Procedure division* quant à elle, contient les instructions COBOL, nommées ***Énoncés***, qui prescrivent le traitement à effectuer. Un énoncé est formé de plusieurs mots. Le premier mot doit être un verbe.

Les énoncés sont regroupés en ***Paragraphes***. Cette partie représente la partie ***Traitement*** du programme et agit sur les données décrites dans la partie Data (Donnée). Le comportement du programme, quant à lui, dépend fortement des données traitées (entrées).

On peut en déduire qu'en fait, un programme COBOL est un moule, spécifiant un ensemble d'actions à réaliser et les différentes initiatives à entreprendre selon les données reçues en entrées.

Si on fait la comparaison avec RM-ODP, on retrouve la même notion avec les objets de traitement, dans le sens où, un *template d'un objet de traitement*, décrit les services offerts à travers *un template d'interface* afin de réaliser une tâche particulière. Ces services agissent sur des données, le plus souvent, passées en paramètres par d'autres objets via des *interactions* (invocations) mais le résultat de leur traitement (le comportement) dépend des valeurs prises par ces données.

D'où la règle suivante :

Règle 7 : Un Programme COBOL correspond à un Template d'objet de traitement

En COBOL, comme cité ci-dessus, le traitement est assuré par des instructions (énoncés) qui sont regroupées dans des *paragraphes*. L'appel à ces paragraphes se fait à partir de *Procedure Division*. Les paragraphes en COBOL n'admettent pas de paramètres. Pour palier à ce problème, la norme ANSI 85 COBOL a introduit la notion de *Nested Programs*

(programmes nichés). Ce sont des programmes à l'intérieur même du programme, et on peut les invoquer en leur passant des paramètres. Le passage de paramètres se fait soit par valeur, soit par référence [Ank 01].

Règle 8 : Un Paragraphe en COBOL correspond à une Opération dans le point de vue Traitement de RM-ODP

Et à partir de la norme ANSI 85 COBOL :

Règle 9 : Un Programme niché en COBOL correspond à un Template d'Objet de Traitement et à une Classe Interne dans un langage Orienté Objet [BUD 99].

2.4. Table des correspondances entre le langage COBOL, Point de vue Traitement RM-ODP et le langage Java

COBOL	Point de vue Traitement de RM-ODP	Le langage Java
CobolProgram / CobolNestedProgram	Template d'Objet de Traitement	JavaClass (avec l'attribut *isInterface* = False)
	Objet de Traitement	Objet Java
	Template d'Interface Opérationnelle de causalité Serveur	JavaClass (avec l'attribut *isInterface* = True)
CobolProcedure (Paragraphe)	*Opétation* (Annoucement, Interrogation)	Méthod (une méthode Java)
	Parameter	FeatureParameter (Paramètres)
	Parameter.Type	*TypeDescriptor* (ClassDescriptor, PrimitiveType)

2.5. Problématique

Le résultat de cette première étape laisse apparaître, que seule l'architecture logicielle de l'application pourra être migrée et non pas le détail des instructions, ce qui s'éloigne des objectifs majeurs fixés par le projet TRAMs. La prochaine initiative serrait de trouver un moyen, qui soit standard, de les spécifier.

3. Conclusion

Le but de cette première approche était de définir des règles de correspondance entre le méta modèle COBOL, le point de vue Traitement de la norme RM-ODP et le Langage Java. Le travail effectué ici représente une première ébauche pour la migration de l'aspect ***métier*** des applications. En effet, comme résultat de cette étape, nous arrivons à faire migrer *l'Architecture Logicielle* de notre application (Objets, Méthodes …etc.) et non pas les détails d'implémentation (instructions, boucles, variables…etc.). On pourra aussi noter qu'établir des correspondances entre COBOL et le point de vue traitement de RM-ODP, était moins évident qu'entre RM-ODP et

Java, du fait que la notion d'objet est commune à ces deux derniers alors qu'elle est totalement absente en COBOL.

Chapitre 3

Introduction

Le travail présenté dans le chapitre 2, avait pour objectif de définir une première ébauche pour la migration de l'aspect ***métier*** des applications. En effet, comme résultat de cette étape, nous arrivions à faire migrer *l'Architecture Logicielle* de notre application (Objets, Méthodes …etc.) et non pas les détails d'implémentation (instructions, boucles, variables…etc.). D'où le besoin qui s'est fait ressentir d'étendre le méta modèle du point de vue Traitement de RM-ODP de notre pivot afin qu'il autorise la spécification d'un tel niveau de détail, tout en restant standard. Notre choix s'est porté sur ***Action Semantics*** d'UML.[OMG 02].

Dans ce qui va suivre, nous commencerons d'abord par présenter Action Semantics ainsi que les arguments qui nous ont poussés à opter pour son intégration dans notre méta modèle. Ensuite, nous donnerons une brève description des possibilités offertes par ce package (**Actions**). Nous nous attarderons tout particulièrement sur les éléments qui nous seront utiles à notre démarche de migration et nous mettrons l'accent sur les concepts communs aux deux méta modèles et qui nous ont permis d'intégrer Action Semantics au MM de Traitement RM-DOP. Nous établirons au fur et à mesure, les correspondances entre les concepts de Action Semantics et ceux de MM COBOL.

Enfin, comme résultat de notre étude, nous présenterons la nouvelle version de notre méta modèle Traitement intégrant Action Semantics ainsi que les nouvelles règles de correspondance entre ce dernier et le Méta modèle COBOL.

1. Le standard Action Semantics de UML

1.1. Pourquoi Action Semantic?

Il est vrai que les points forts de RM-ODP sont la stabilité et la rigueur de ses concepts, c'est d'ailleurs ce qui justifie notre choix de la prendre comme formalisme pivot pour notre démarche de migration. Cependant, vu la généricité de RM-ODP, elle ne permet pas de spécifier les actions reliées aux Opérations ainsi que la sémantique sous jacente. Or, lors de la migration du code la question s'est clairement posée.

L'idée était de trouver une approche qui nous permettrait de formaliser ces instructions, tout en restant standard et indépendant de tout choix d'implémentation. Action Semantics s'est avéré un candidat potentiel qui répondait à ces critères.

Plutôt qu'en parler longuement, on pourra citer ses avantages sous forme de points :

- Permet une spécification précise du comportement ;
- Une sémantique bien fondée ;
- Son niveau d'abstraction est un bon compromis entre les spécifications de haut niveau d'UML et les concepts d'implémentation ;

- Permet l'exécution de plusieurs Threads à la fois ainsi que des mécanismes de synchronisation, faisant en sorte que des procédures peuvent s'exécuter dans un ordre spécifié. La Sémantique d'exécutions concurrentes peut être facilement mappée vers une implémentation distribuée ;
- Action Semantics s'inscrit dans la nouvelle vision de l'OMG : le **MDA** [OMG 01]. Du fait qu'il offre la possibilité de faire évoluer, raffiner et transformer des modèles à un stade relativement tôt de la conception jusqu'à leur implémentation [JEZ 02] ;

1.2. Présentation d'Action Semantics

L'initiative de l'OMG était de proposer une extension au formalisme UML incluant un modèle d'Actions de Traitement avec une sémantique complète permettant de modéliser le comportement de n'importe quel type d'action qu'on pourrait trouver dans une procédure ainsi que les données accessibles par celle-ci, une procédure étant un groupe d'actions réunies et qui s'exécutent comme une unité.

Cette extension se présente sous forme d'un package, appelé package : **Actions** (voir Figure. 3.1). Ce dernier contient sept autres package que nous détaillerons par la suite.

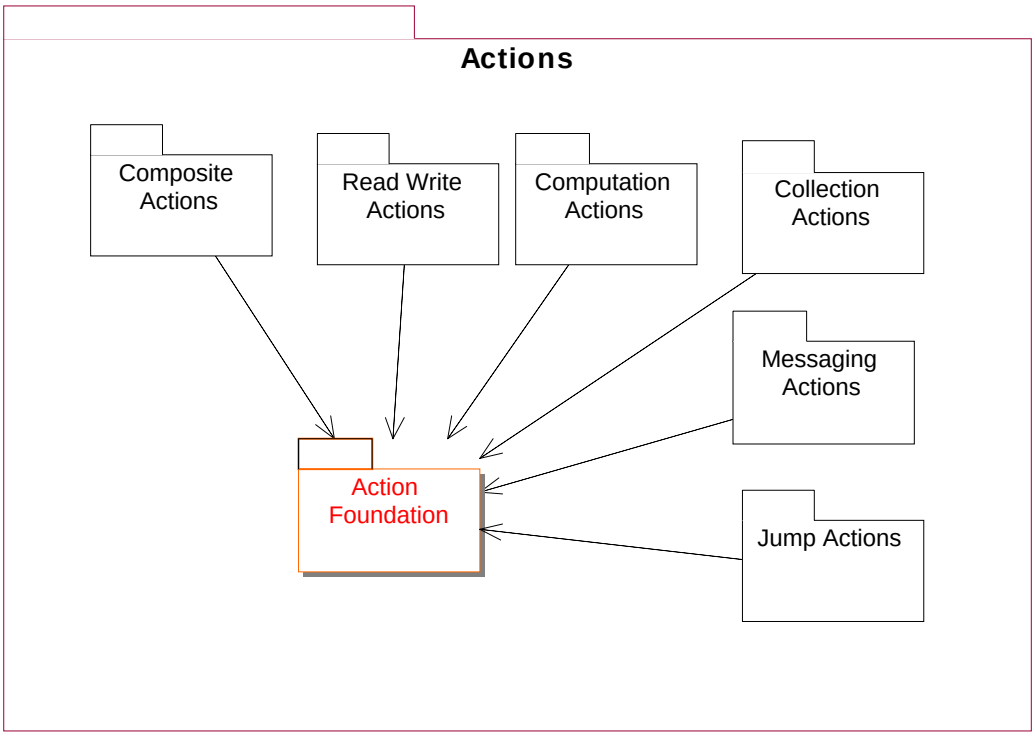


Fig. 3.1 : Le package Actions

1.2.1. Le package Action Foundation

Ce package spécifie la structure des procédures et des actions dans le méta modèle UML. Les concepts définis dans ce package s'appliquent à tout type d'actions.

La brique de base pour la spécification du comportement est l'***Action*.** Elle prend en entrée, un ensemble d'***inputs*** et produit un ensemble d'***outputs***, tous deux peuvent être des ensembles vides et représentent les ***Pins*** de l'action. Un ***Pin*** a un type et une multiplicité en rapport des valeurs contenues par celui-ci.

Dans le méta modèle Cobol, on retrouve la même notion du ***Pin*** avec l'élément ***CobolVariableData***. Une *CobolVariableData* est associée à un champ BMS qui, à son tour, est assigné à une ***CobolOperation***. Celle-ci peut soit, l'utiliser (la valeur de la variable associée au champ) comme input d'une opération, soit comme récepteur d'un output d'une opération d'où la similitude avec le concept de ***Pin*** du package *Action Foundation*.

Une ***action primitive*** est une action indivisible. Elle permet de spécifier des fonctions purement mathématiques, des fonctions sur les strings, les Read and Write actions…etc.

Action Semantics considère que toutes les actions sont exécutées concurremment à moins d'imposer explicitement un séquencement de leur exécution par un flux de données ou de contrôle (Data Flow et Control Flow). Un ***Data Flow*** d'un output pin vers un input pin signifie que le résultat d'une action pourra être utilisé comme entrée d'une autre action. En absence d'un flux de données explicite entre actions, le ***Control Flow*** détermine une contrainte d'ordre entre une action précédente et celle qui la succède. Le méta modèle qui suit (Figure.3.2) représente ces concepts de bases et qui sont utilisés par tous les autres packages.

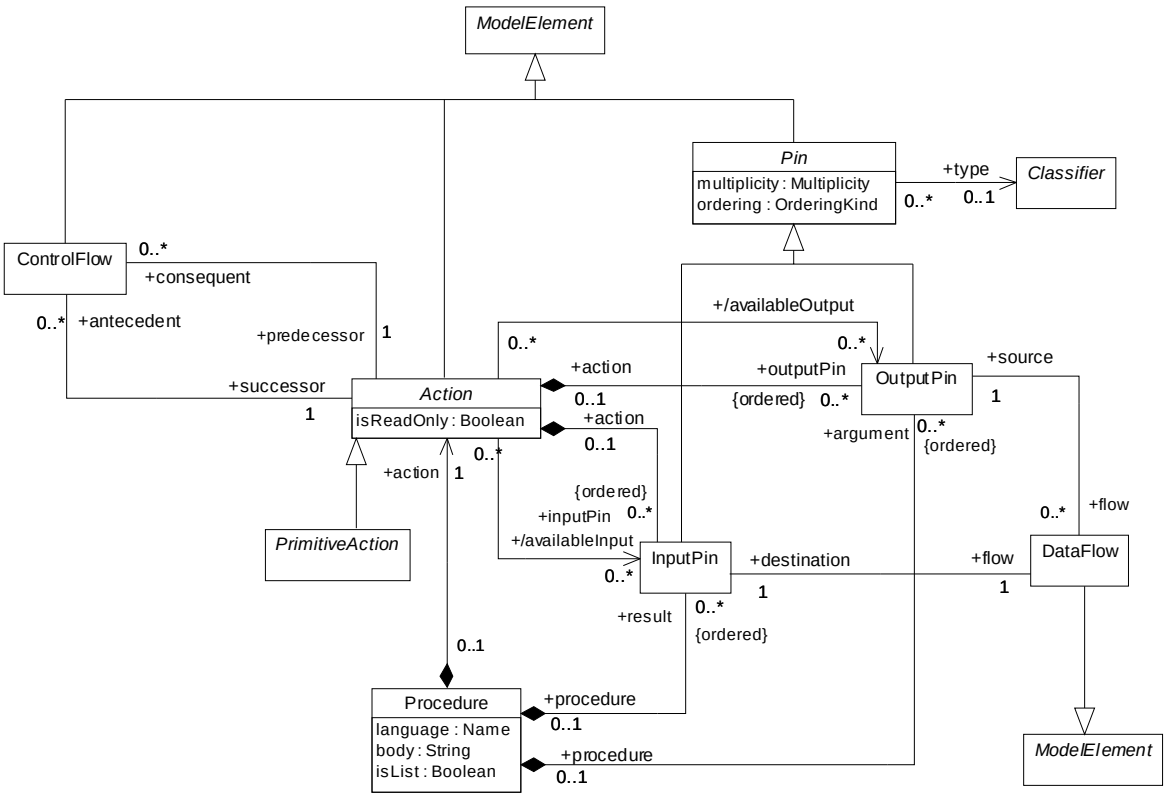


Fig. 3.2. Le package Action Foundation : brique de base d'Action Semantics

1.2.2. Composite Actions

Les Composites Actions sont des structures récursives permettant la composition d'actions complexes à partir d'actions plus simples, qui à leur tour, peuvent être des actions primitives aussi bien que composées. Ce package définit trois types d'actions composites :

1.2.2.1. Group Action

C'est un block qui regroupe un ensemble d'actions en une seule unité et qu'on retrouve dans les procédures. On pourra facilement faire la correspondance avec les blocks d'opérations en Cobol, représentés par l'élément ***CobolStatementBlock*** du méta modèle COBOL (voir annexe 1). Aussi, un ensemble de *variables locales* peut être associé au groupe d'actions et dont la portée ne dépassera pas celle du groupe. L'élément ***Variable*** du modèle présenté en figure.3.3.correspond aux ***Variables locales*** d'une procédure.

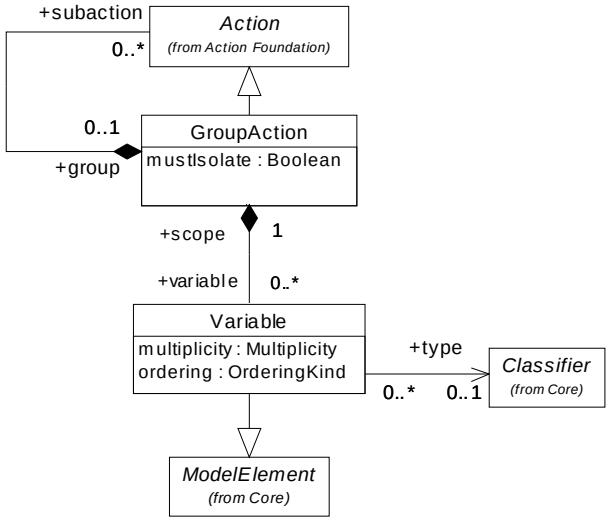


Fig. 3.3 : Group Action.

1.2.2.2. Conditional Action

Une action conditionnelle (figure. 3.4) consiste en un certain nombre de ***clauses***, chacune comprend une ***Test*** action et une ***Body*** action (toutes deux sont des ***Action***). La *test* action produit un output pin. S'il est évalué à *vrai*, l'action *Body* est exécutée. L'action *body* étant une action, elle peut être simple ou complexe (***Group Action***). Au cas ou plusieurs *test actions* sont évaluées à vrai, UML ne définit pas comment choisir la *body* action qui s'exécutera.

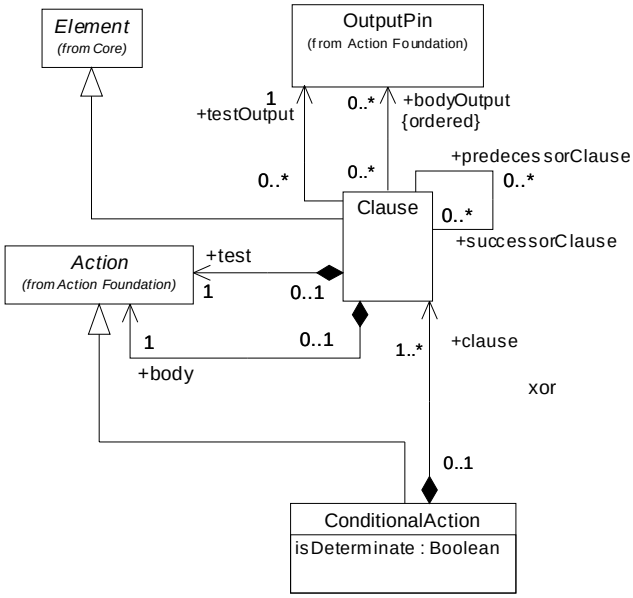


Fig.3.4: Conditional Action du package Composite Actions

Les concepts de ***Conditional Action***, ***Clause*** et ***Body Action (Group Action)***, correspondent respectivement aux éléments : ***If***, ***Condition*** *et* ***CobolStatementBlock*** du méta modèle Cobol. Ces mêmes concepts peuvent être mappés en Java en tant que structures conditionnelles (If et Case). *Test Action* pour Action Semantics est toute action (primitive, groupe d'actions…etc.) qui hérite d'***Action*** et qui nous permettra de spécifier le test en question. Ça reste donc un choix d'implémentation que nous n'aborderons pas ici.

1.2.2.3. Loop Action

Le dernier type d'action composite est ***Loop Action***. Elle permet l'expression d'une exécution répétitive d'une action aussi longtemps que le résultat de la *test Action* le permettra. La *loop action* ne contient qu'une seule clause avec une *test action* et une *body action*.

Cette action représente la boucle ***While***, tant en Cobol qu'en Java et aussi la boucle ***FOR*** en java.

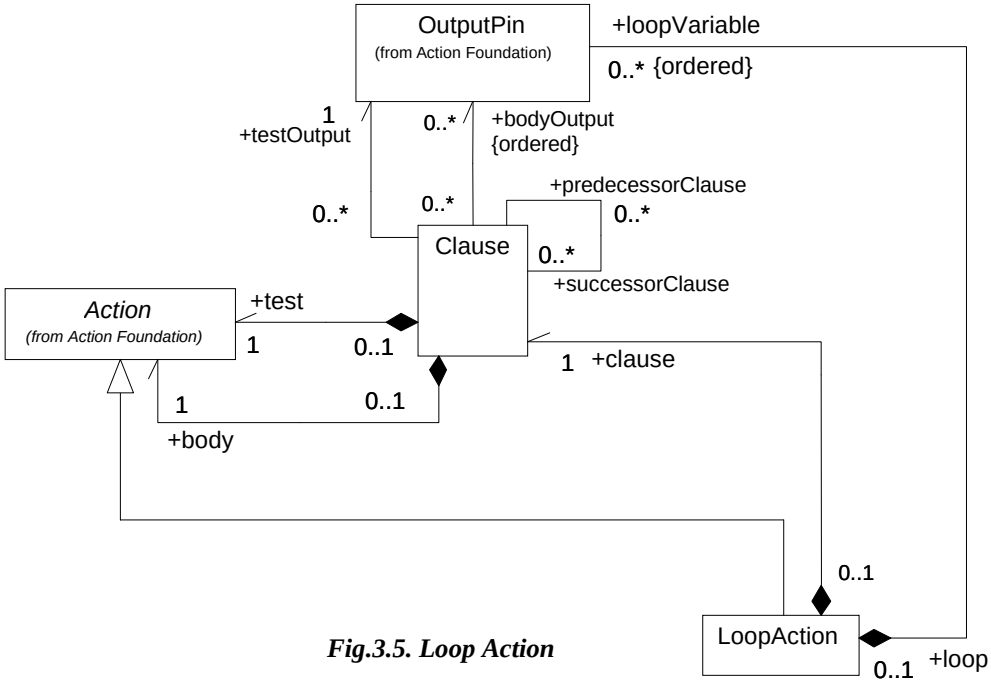


Fig.3.5. Loop Action

1.2.3. Read et Write Actions

Dans ce package, on retrouve toutes les actions permettant de manipuler les valeurs relatives aux objets, attributs, liens et variables. Vu que nous nous intéressons tout particulièrement à établir des règles de correspondance entre le méta modèle cobol et le méta modèle traitement de RM-ODP, certaines actions nous ont parues inutiles à notre démarche et de ce fait, nous avons décidé de ne pas les aborder ici.

1.2.3.1. Object Actions

Ces actions couvrent les propriétés directes des objets. On pourra citer : ***CreateObjectAction*** qui crée un objet conforme à un Classifier et ***DestroyObjectAction***

supprime l'objet fourni en entrée (input pin). ***ReclassifyObjectAction*** ajoute et supprime des classifiers à et à partir des objets donnés en input (au moment de l'exécution).

1.2.3.2. Attribute Actions

Idem pour les attributs, la figure.6 présente les différentes actions possibles sur ces derniers.

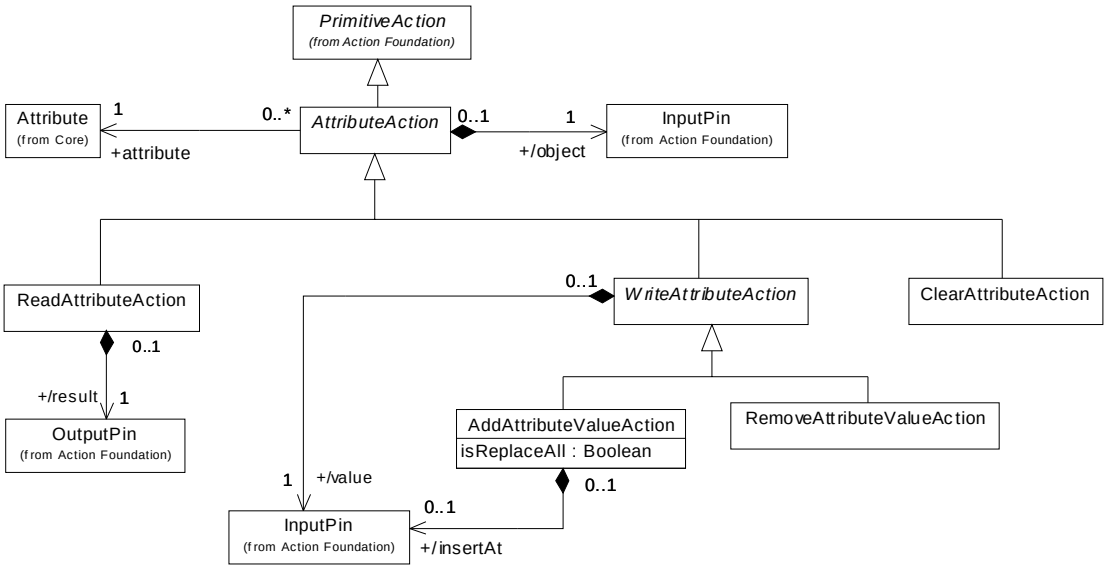


Fig.3.6 les actions possibles sur les attributs.

1.2.3.3. Variables Actions

Les *Variables Actions* peuvent avoir accès seulement aux variables à l'intérieur de la procédure dont fait partie l'action. Les valeurs des variables peuvent être lues et modifiées comme le montre le modèle de la figure 3.7.

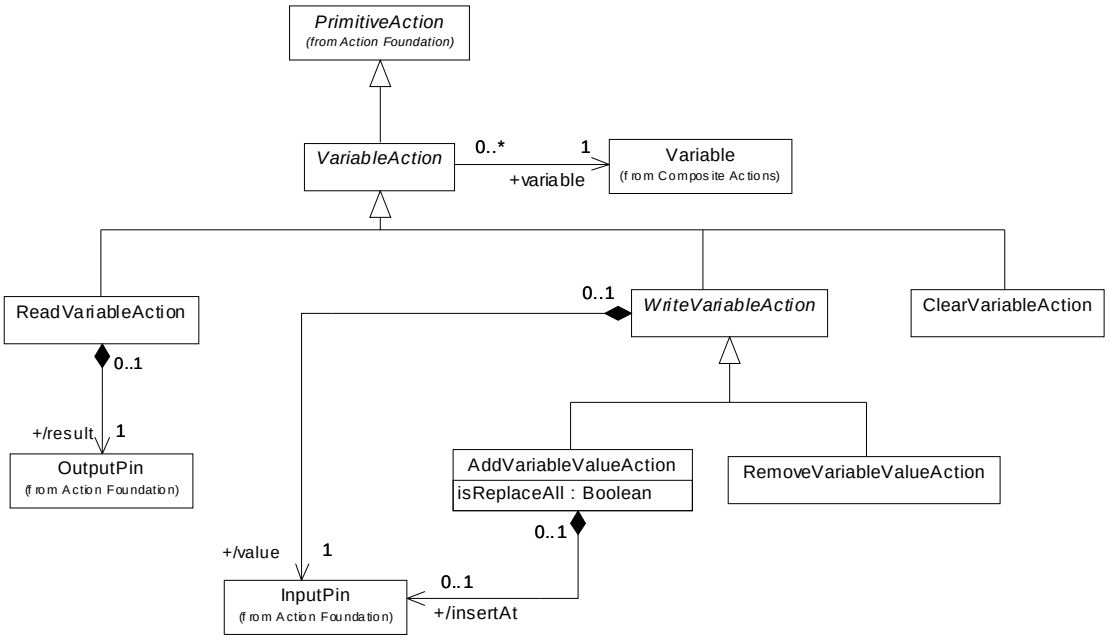


Fig. 3.7. Les actions sur les variables

Dans le méta modèle Cobol, l'élément ***Affectation*** qui hérite de *CobolStatement* représente une instruction d'affectation de variables de type *MOVE* ou *COMPUTE* par exemple. Ce qui revient au fait à un *Read* d'une variable suivi d'un *Write* sur une autre. Donc on pourra faire correspondre une affectation par une action de ***ReadVariableAction*** suivie de ***AddVariableValueAction***. Si l'affectation n'engage qu'une seule variable, *AddVariableValueAction* suffira du fait qu'elle offre la possibilité de remplacer une ancienne valeur par une nouvelle fournie en entrée.

Ici, nous insistons tout particulièrement sur l'affectation de variables du fait qu'en Cobol, la notion d'objet est totalement absente et du coup celle d'attribut aussi. Mais on peut tout aussi bien représenter une instruction d'affectation d'attributs dans des langages objet par ***ReadAttributeAction*** et ***AddAttributeValueAction***.

1.2.4. Messaging Action

Ces actions permettent l'échange de messages entre objets. L'objet qui envoie le message peut tout simplement continuer son exécution sans se soucier du comportement invoqué par sa requête. Ce qui est désigné comme *Invocation Asynchrone*. Ou bien, l'objet initiateur du message, choisit de suspendre son exécution jusqu'à ce que le résultat de sa requête lui soit renvoyé par l'objet invoqué (*Invocation Synchrone*).

Ce package définit les actions ***CallOperationAction***, ***SendSignalAction*** et ***BroadcastSignalAction*** qui fournissent les fonctionnalités habituelles qu'on retrouve dans les langages de programmation traditionnels sans la spécification du mécanisme complet de messaging et peuvent donc être implémentées directement.

L'action *CallOperationAction* comporte l'attribut ***isSynchronous*** tel que, s'il est évalué à ***Vrai***, l'invocation est synchrone et l'objet invocateur attend la réponse de l'objet invoqué. Si elle est égale à ***faux***, l'invocateur n'attend pas.

CallOperationAction avec l'attribut *isSynchronous* égal à *vrai*, correspond au ***Perfom*** de Cobol et au ***Goto*** s'il est égal à *faux*.

Même si les concepts de *Signal* ou *BroadcatsSignal* (envoi vers multiples cibles) n'apparaissent pas en Cobol, nous avons jugé qu'il était tout à notre avantage d'avoir un pivot qui soit le plus complet possible, couvrant ainsi d'autres sources cibles qui pourraient les manipuler.

Ce même package définit deux autres actions : ***SynchronousInvocationAction*** et ***AsynchronousInvocationAction*** qui peuvent formaliser les mêmes fonctionnalités que celles assurées par *CallOperationAction* à la différence près que les arguments ainsi que le type d'invocation (Opération ou signal) sont encapsulés dans un objet (le *Requester*). Alors que pour *CallOperationAction*, les arguments sont explicites i.e. des tuples de valeurs avec pour chacun, un type qui correspond à celui attendu par la méthode invoquée.

Par souci de simplicité et surtout pour le côté pratique de la chose, on a préféré ne garder que les invocations explicites (*CallOperationAction*, *SendSignalAction* et *BroadcastSignalAction*). Ces actions sont présentées ci-dessous (figure 3.8).

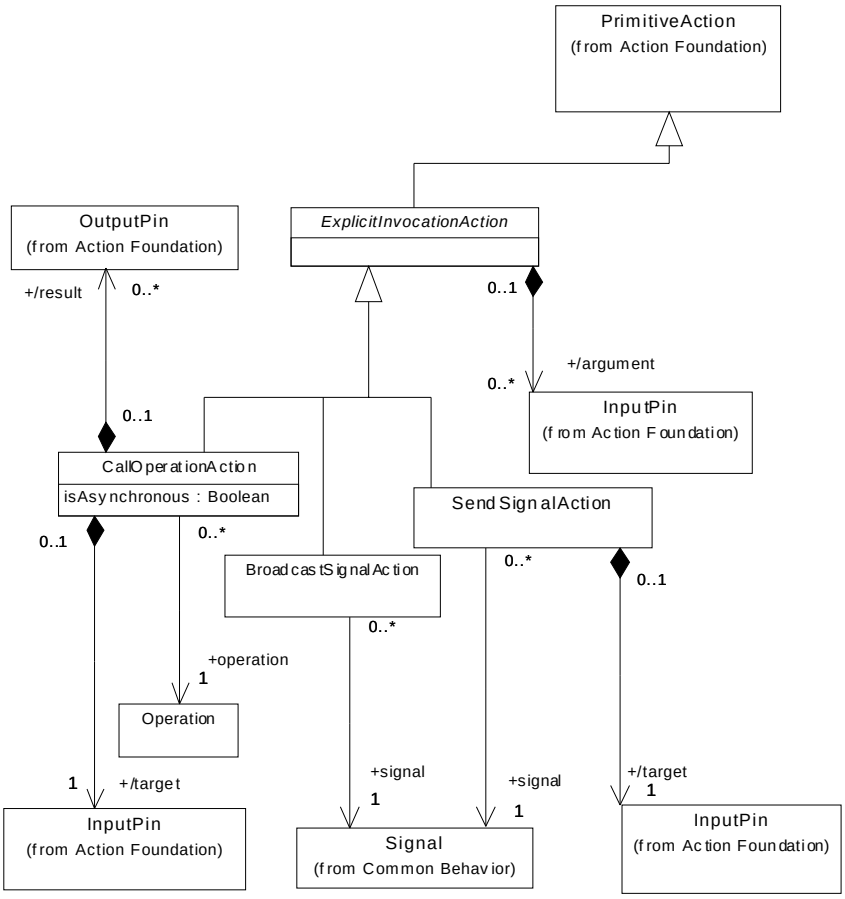


Fig.3.8. Les échanges de messages en Action Semantics

1.2.5. Computation Action

Nous arrivons à présent aux actions de calculs. Ces actions transforment un ensemble de valeurs en input vers un autre ensemble de valeurs en output. Elles emboîtent les fonctions mathématiques et fournissent des ***fonctions primitives*** sur lesquelles la plupart des calculs sont construis.

Action Semantics ne définit pas ces fonctions primitives par souci de généricité et laisse libre choix au développeur de le faire selon l'usage pré requis de celles-ci. Cela se fera sans doute en utilisant des profils.

En ce qui nous concerne, nous pensons que le mieux serait de suivre cette philosophie, dans le sens où nous devons laisser un maximum de souplesse quant aux fonctions primitives à expliciter. Nous pourrions en définir quelques-unes mais cela réduirait la pertinence de notre méta modèle. L'idéal serait que les règles de transformation et la partie de génération de code puissent assurer ce point là. Cependant, il ne faudrait pas perdre de vue ces fonctions (la tracabilité) lors du passage de la source vers la cible, en passant par le pivot.

De ce fait, les éléments du méta modèle Cobol : ***Plus***, ***Minus***, ***Times*** et ***Quotient*** pourraient correspondre à des ***PrimitiveFunction*** du package Computation Action. Tout en sachant que pour certains langages, l'addition est une méthode ayant comme arguments les valeurs à additionner.

Il est à signaler que l'élément ***PrimitiveFunction*** qui apparaît dans le modèle du package Computation (voir figure 3.9), n'est plus ni moins qu'une signature et non pas une action. Cette signature comporte le nom de la fonction primitive à appliquer et des inputs pins comme arguments (les valeurs sur lesquelles porte l'action).

La solution que nous proposons pour la migration des éléments cités ci-dessus (***Plus***, ***Minus***, ***Times*** et ***Quotient***) du MM Cobol vers Pivot, est de les faire correspondre avec l'élément ***PrimitiveFunction*** du pivot. Prenons l'exemple de ***Plus*** (du MM Cobol), elle correspondra dans le pivot à une ***primitive action*** dont le nom (attribut *name* hérité de Model Element) est *Plus* et les valeurs qu'on voulait additionner seront des ***inputs pin***. Ce que nous voulons expliciter est qu'au niveau instance de ce méta modèle, on aura : des *Plus*, des *Minus*…etc. et il ne restera aux règles de transformation qu'à faire la correspondance avec le langage cible et de prévoir la transformation appropriée, puis la génération du code.

L'action *MarshalAction* permet de créer un objet et d'initialiser ses attributs. Les valeurs d'attributs seront fournies en tant qu'inputs de l'action. A l'inverse de *MarshalAction*, *UnMarshalAction* dispatche les valeurs des attributs d'un objet vers des outputs. Pour ces actions là, nous n'avons pas vraiment pu en retirer quelque chose qui pourrait nous servir, en tous les cas, pas pour le moment. C'est pourquoi nous avons pris le choix de les ignorer.

Reste d'autres actions de calcul spécifiées dans ce package mais qui n'ont pas encore été bien définies. Par conséquent, nous ne nous aventurons pas à les inclure dans notre méta modèle.

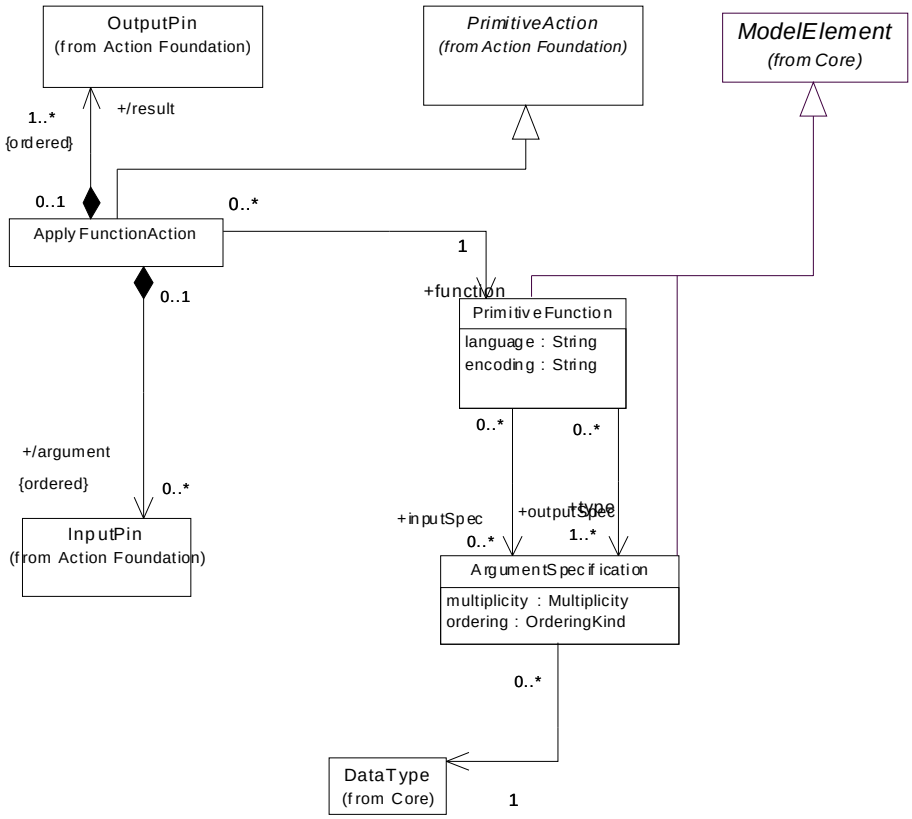


Fig. 3.9. PrimitiveAction de Computational Actions

1.2.6. Jump Actions

Le modèle contenu dans ce package (Figure 3.10) formalise le fait que, sous certaines conditions, l'exécution d'une action courante soit abandonnée pour transférer (*Jump*) le contrôle à une autre action alternative (ou groupe d'actions). L'action alternative est nommée ***HandlerAction*** et l'action qui provoque le *Jump* : ***JumpAction***. Une action peut avoir zéro ou plusieurs *actions handler* rattachées à elle. Le mapping de cette action vers les langages de programmations traditionnels serait la représentation des *break*, *continue* et *exception*. En UML l'occurrence d'une exception est modélisée par un *Jump object*. Le type de l'objet Jump représente le genre de l'exception et les arguments désignent les paramètres de l'exception.

Une levée d'exception explicite en UML est modelée à l'aide d'une ***JumpAction***. Ce qui nous amène à en déduire que l'élément ***Error*** du MM Cobol correspond à une ***JumpAction*** dans notre MM modèle Traitement du pivot.

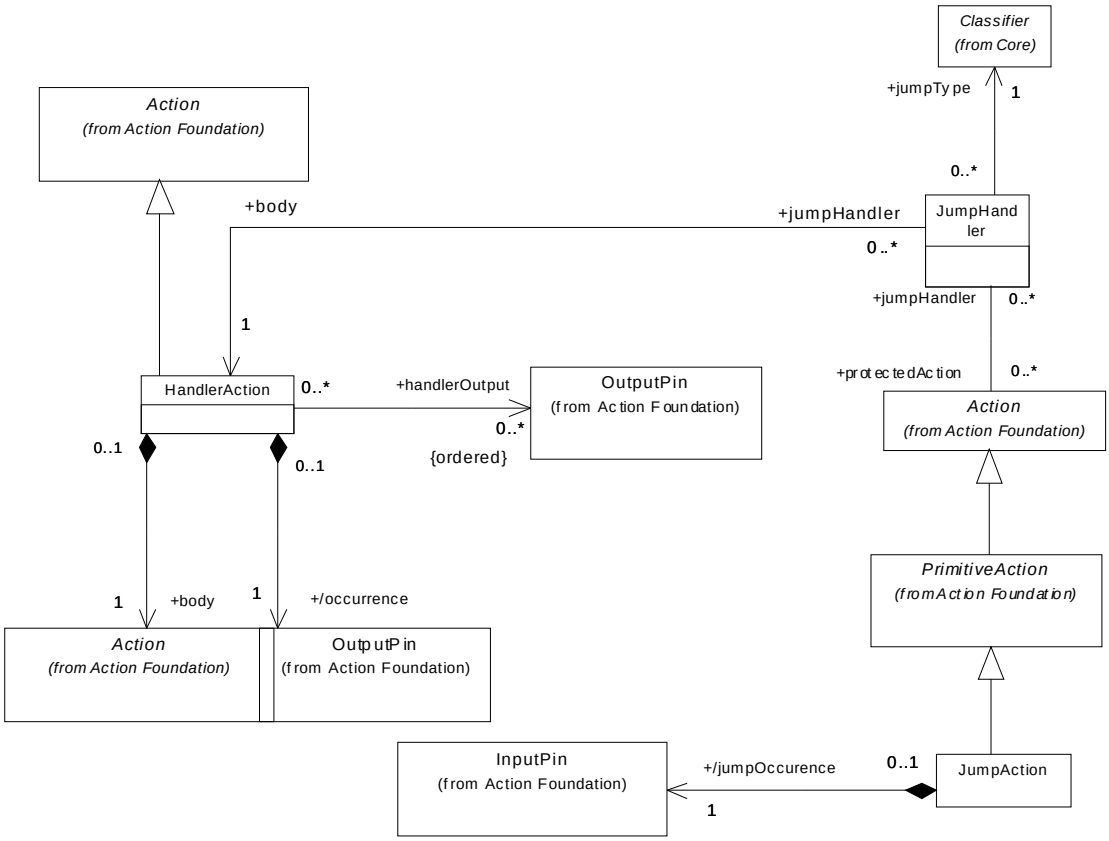


Fig. 3.10 les Jump Actions

1.2.7. Collection Actions

Une *collection action* applique une sous-action aux éléments d'une collection. Ce package définit quatre types d'action. ***MapAction, Filter Action, IterateAction et ReduceAction***. En regardant bien le standard, nous avons pu comprendre que ces actions étaient plus faites pour formaliser une façon de faire ou un algorithme plutôt qu'une action.

D'ailleurs dans la spécification du standard, une *FilterAction* est assimilé, à titre d'exemple, au moyen de vérifier si un compte bancaire est au-dessus d'une certaine balance, la *ReduceAction* : à la récapitulation des balances de tous les comptes…etc.

Il est vrai que ces actions seraient d'une grande utilité si on avait à modéliser une application à l'aide d'Action Semantics pour une génération directe et complète du code. Or, pour notre usage, ces actions nous semblent inutiles et peuvent être remplacées par d'autres actions plus simples même si c'est plus contraignant, l'objectif étant toujours d'avoir un méta modèle le plus expressif possible mais aussi le plus simple même si c'est un peu paradoxal.

2. Couplage de RM-ODP et Action Semantics

L'idée de départ comme citée ci-dessus, était de trouver le moyen d'apporter à notre méta modèle, un pouvoir d'expression qui lui permettrait d'exprimer les détails des instructions qu'on trouverait dans n'importe quelle application tout en restant standard, permettant ainsi, une migration de la source vers la cible (en passant par le pivot) sans pertes d'informations.

Action Semantics avait tout pour répondre à nos besoins. Le problème qu'il faut alors résoudre est son intégration dans notre méta modèle Traitement d'RM-ODP. Le but est de trouver des concepts communs aux deux standards et de ce qui pourrait être une continuité logique de l'un vers l'autre dans le sens où on part du plus abstrait vers le plus spécifié.

Pour cela, on a pris les concepts d'***Action*** et ***Opération*** de RM-ODP.

Dans la spécification du Standard RM-ODP, un ***objet*** a un ***comportement*** qui est décrit par un ensemble d'actions. Une ***Action*** : "*est quelque chose qui se passe*" [ODP 95]. En Action Semantics, tout est conçu autour de l'***Action*** qui est la brique de base du package ***Actions***. Une ***action*** étant une spécification du comportement et qui agit sur des inputs pour produire des outputs.

Même si les deux définitions ne sont pas exactement les mêmes, nous pourrons remarquer que l'une englobe l'autre i.e. la définition donnée par RM-ODP est plus abstraite et elle peut inclure celle donnée par le standard Action Semantics.

D'autre part, en RM-ODP, une ***Interface*** est composée d'***Interactions***, toutes du même type (***Signal***, ***Operation*** ou ***Flow***). Chaque interaction a une ***Signature*** qui correspond à son nom ainsi que des paramètres. En UML, une ***Operation*** (du package Core) correspond à la *spécification* d'une ***Méthode***. Une ***Méthode*** étant implémentée par une ***Procédure*** (du package Action/Action Foundation).

Donc, on peut facilement mapper le concept d'***Operation*** de RM-ODP à celui d'*Operation* d'UML car ils correspondent exactement à la même notion.

Le méta modèle résultant du couplage du point de vue Traitement RM-ODP et d'Action Semantics est joint à ce document en **annexe 2**.

3. Les Règles de correspondances

Le méta modèle résultant du couplage du point de vue Traitement RM-ODP et d'Action Semantics est complet, dans le sens où on peut représenter, et l'Architecture

Logicielle des applications, et les détails des instructions spécifiant leur comportement. De ce fait, dans le cadre du prototype "*Cobol vers le Web*", le méta modèle Traitement/AS nouvellement spécifié, nous a permis d'enrichir considérablement notre table de correspondances entre le MM Cobol et ce dernier, ouvrant ainsi la voie vers une génération complète du code (Voir Table 3.1).

COBOL	Point de vue Traitement de RM-ODP/AS
CobolProgram/CobolNestedProgram	Template d'Objet de Traitement *(RM-ODP)*
CobolProcedure (Paragraphe)	Method *(from package Core de UML)*
Perform	*CallOperationAction* (*from Messaging Action de AS*) avec *CallOperationAction.isSynchronous = vrai*
GoTo	*CallOperationAction* avec *CallOperationAction.isSynchronous = faux*
CobolVariableData	Pin *(from Action Foundation de AS)*
CobolStatementBlock	Groupe Action *(from Composite Actions de AS)*
If	Conditional Action *(from Composite Actions de AS)*
While	Loop Action *(from Composite Actions de AS)*
Condition	Clause *(from Composite Actions de AS)*
Plus	PrimitivesFunction *(from Computation Actions de AS)*
Minus	
Times	
Quotient	
Error	JumpAction *(from Jump Actions de AS)*

Table 3.1. Règles de correspondance entre Cobol et Point de vue Traitement/AS

4. Conclusion

Au cours de cette étude, nous avons tout d'abord exposé le problème auquel on a été confronté lors des premières phases de migration et qui consistait au fait, que notre MM Traitement était trop abstrait pour qu'il puisse autoriser une migration complète du code. La solution que nous avons proposée se base sur le standard Action Semantics de UML, qui nous a permit de spécifier le comportement des actions qu'on pourrait trouver dans les langages de programmation traditionnels. Le résultat étant un MM du point de vue Traitement intégrant Action Semantics et une table de correspondance plus complets. La prochaine étape serait d'étendre le MM Java proposé par netBeans afin, qu'à son tour, il puisse prendre en considération le détail des instructions nous permettant ainsi une génération complète du code au niveau de la cible.

Chapitre 4 : Le Méta-Modèle JAVA

Une fois que nous avions étendu notre méta-modèle Traitement en incluant la spécification détaillée du comportement à l'aide de Action Semantics, le besoin de modifier notre méta-modèle cible, à savoir le MM du langage Java, est devenu inévitable.

Dans ce qui suit, nous vous présenterons le méta-modèle du langage Java que nous avons spécifié. Nous détaillerons les concepts clés de ce dernier. Ensuite, nous décrirons la manière dont nous avons inclus la spécification de l'IHM ainsi que l'enchaînement des écrans dans les deux méta-modèles : Traitement de notre Pivot et le Langage Java. Enfin, nous présenterons le tableau des règles de correspondances que nous avons établi entre le MM Traitement et le MM Java.

1. Le Méta-Modèle du langage JAVA

L'idée de netBeans était de créer un méta-modèle permettant la définition de structures de données qui décrivent – suffisamment- un programme écrit en langage Java. netBeans, a choisit le standard MOF (Meta Object Facility) adopté par l'OMG afin de décrire ce méta-modèle. Cependant ce dernier s'arrête à l'architecture logicielle (Classes et Méthodes) et ne spécifie pas le détail des instructions que pourrait inclure une méthode Java (figure 4.1).

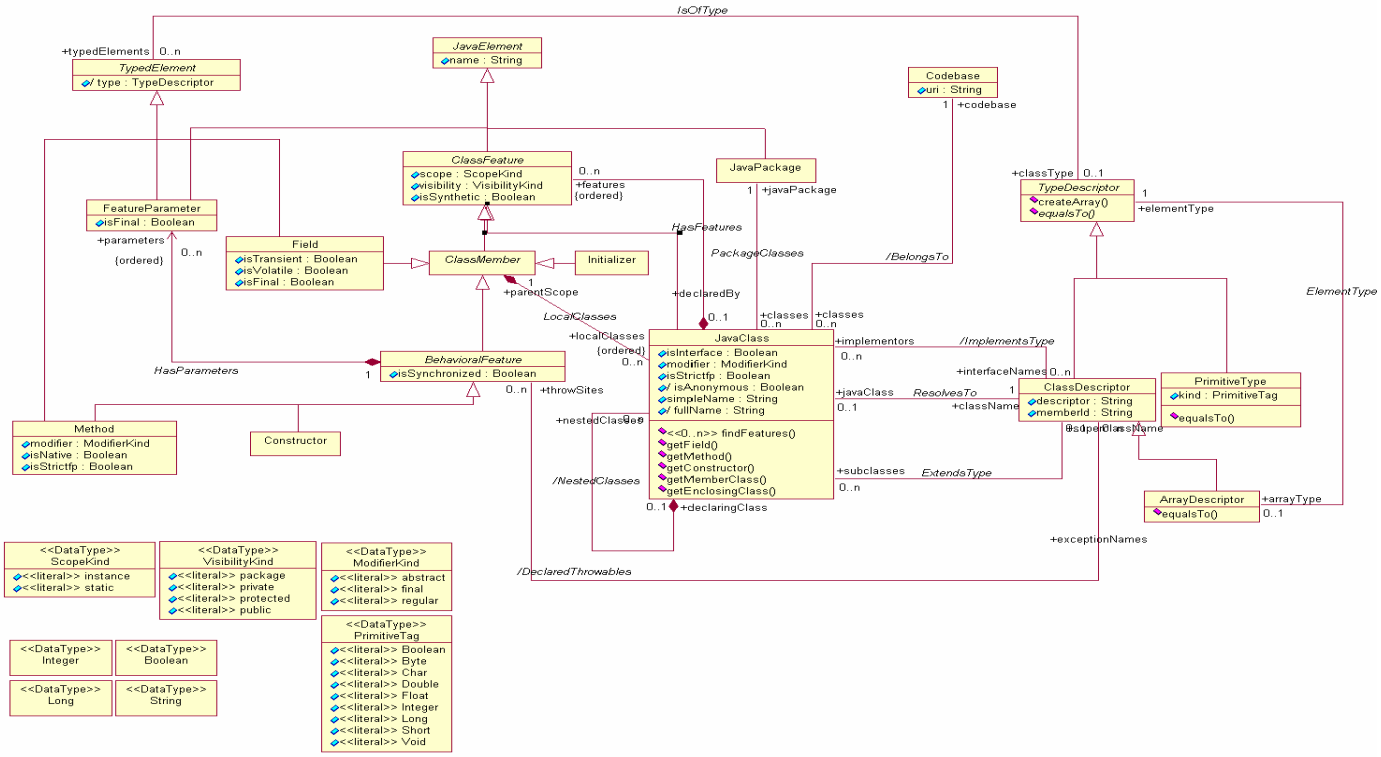


Fig.4. 1 : Méta-Modèle Java proposé par netBeans

1.1. Présentation du méta-modèle Java

Afin d'étendre le méta-modèle Java proposé par netBeans, il nous a fallu nous approfondir dans les concepts propres au langage. La spécification résultante, nous permet de spécifier n'importe quel type d'application Java avec toute la sémantique sous-jacente.

Le méta-modèle que nous avons spécifié se présente comme suit (figure 4.2.) :

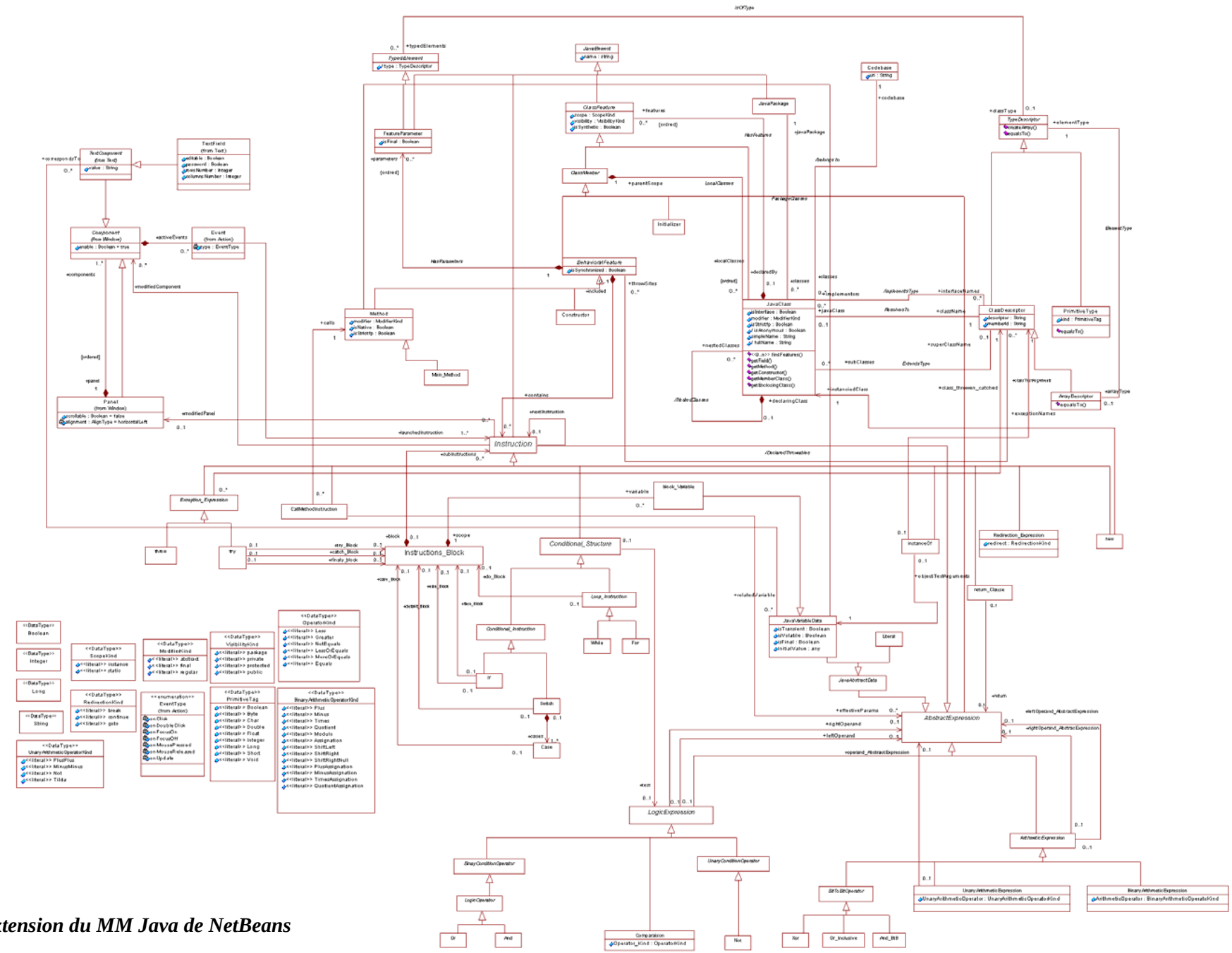


Fig. 4.2. Extension du MM Java de NetBeans

1.1.1 Classes

Les classes du méta-modèle Java sont décrites en précisant leurs super-classes, leurs attributs et les références qu'elles possèdent vers d'autres classes comme suit :

JavaElement (abstract class)
Super-classes : Aucune
Attributs : **name** de type string: *le nom de l'élément du modèle*
Références : Aucune

ClasseFeature (abstract class) Les caractéristiques de la classe.
Super-classes : JavaElement
Attributs : scope de type **ScopeKind** (énumération)
visibility de type **VisibilityKind** (énumération)
isSynthetic de type Boolean
Références : Aucun

ClassMember (abstract class) Les composants d'une classe
Super-classes : ClassFeature
Attributs : Aucun
Références : javaclass de type **JavaClass**

Initializer
Super-classes : ClassMember
Attributs : Aucun
Références : Aucun

BehavioralFeature (abstract class) Caractéristiques liées au comportement de la classe.
Super-classes : ClassMember
Attributs : isSynchronized de type Boolean (garantissant qu'un seul thread utilise la méthode –le block- en même temps)
Références : featureparameter de type **FeatureParamater** : les caractéristiques du paramétrer de la méthode/constructeur.
instruction de type **Instruction** : les instructions contenues dans le block

JavaPackage Le package contenant la classe
Super-classes : JavaElement
Attributs : Aucun
Références : Aucune

Contructor Le constructeur de la classe
Super-classes : BehavioralFeature
Attributs : Aucun
Références : Aucun

Method Une méthode Java
Super-classes : BehavioralFeature, TypedElement
Attributs : modifier de type ModifierKind (énumération)
isNative de type Boolean (indiquant que la méthode est implémentée dans un autre fichier source et dans un autre langage)
isStrictfp : Boolean. (pour contrôler certains aspetcs liés à la virgule flottante)
Références : Aucun

Main_Method La méthode Main de la classe
Super-classes : Method
Attributs : Aucun
Références : Aucun

FeatureParameter Le paramètre d'une méthode
Super-classes : TypedElement, JavaElement
Attributs : isFinal : Boolean (permet de définir qu'une entité ne pourra pas être changée ou dérivée)
Références : Aucune

JavaClass Une classe Java (interface selon l'attribut isInterface :Boolean)
Super-classes : ClassFeature
Attributs : isInterface : Boolean (définir s'il s'agit d'une classe ou d'une interface)
modifier de type **ModifierKind** (énumération)
isStrictfp : Boolean (concept lié à la virgule flottante)
isAnonymous : Boolean
simpleName : String
fullName : String
Opérations getXXX (des accesseurs)
Références : Aucune

TypedElement (abstract class) Le type.
Super-classes : Aucune
Attributs : type de type **TypeDescriptor**
Références : Aucune

TypeDescriptor (abstract class)
Super-classes : Aucune
Attributs : Aucun
Références : Aucun
Opérations: createArray() et equalsTo()

ClassDescriptor si le type est complexe
Super-classes : TypedDescriptor
Attributs : descriptor de type String
memberId de type String
Références : Aucun

PrimitiveType Pour spécifier les types primitifs
Super-classes : TypedDescriptor
Attributs : Kind de type PrimitiveTag (énumération)
Références : Aucun
Opération : equalsTo()

ArrayDescriptor Pour les tableaux
Super-classes : ClassDescriptor
Attributs : Aucun
Références : Aucun
Opération : equalsTo()

Instruction (abstract class) Une instruction Java
Super-classes : JavaElement, AbstractExpression
Attributs : Aucun
Références : instruction de type **Instruction** : la prochaine instruction.
panel de type **Panel** : Le panel sur lequel l'action agit.
component de type Component : le composant sur lequel l'action agit

Exception_Expression (abstract class) Pour spécifier les exceptions
Super-classes : Instruction
Attributs : Aucun
Références : classedescriptor de type **ClassDescriptor** : le nom de la classe d'exception.

throw mot clé java pour spécifier l'exception.
Super-classes : Exception_Expression
Attributs : Aucun
Références : Aucun

try mot clé java pour ouvrir le block d'exception.
Super-classes : Exception_Expression
Attributs : Aucun
Références : tryblock de type **Instructions_Block** : le block try.
catchblock de type Instructions_Block : le block catch
finaly de type Instructions_Block : le block finaly.

Instructions_Block Un block d'instructions
Super-classes : Instruction
Attributs : Aucun
Références : instruction de type Instruction : les sous-instructions du block.

CallMethodInstruction Un appel de méthode en Java.
Super-classes : Instruction
Attributs : Aucun
Références : method de type Method : la méthode appelée.
abstractexpressionparam de type **AbstractExpression** : paramètre de l'appel (variable, résultat d'un test, résultat d'un calcul…etc.)

Conditional_Structure (abstract class) une structure conditionnelle
Super-classes : Instruction
Attributs : Aucun
Références : logicexression de type **LogicExression :** le test étant toujours une expression logique.

Conditional_Instruction (abstract class) le If ou le Switch.
Super-classes : Conditional_Structure
Attributs : Aucun
Références : Aucune

Loop_Instruction (abstract class) le for ou le while
Super-classes : Conditional_Structure
Attributs : Aucun
Références : doblock de type **Instrcutions_Block**

while La boucle while
Super-classes : Loop_Instruction
Attributs : Aucun
Références : Aucune

For La boucle For
Super-classes : Loop_Instruction
Attributs : Aucun
Références : Aucune

If La condition If
Super-classes : Conditional_Instruction
Attributs : Aucun
Références : thenblock de type **Instrcutions_Block** : le block then.
elseblock de type **Instrcutions_Block** : le block else.

Switch Le Switch
Super-classes : Conditional_Instruction
Attributs : Aucun
Références : defaultblock de type **Instrcutions_Block** : le block default.
case de type **Case** : les différents Cases du switch.

Case Le case du Switch
Super-classes : Aucun
Attributs : Aucun
Références : caseblock de type **Instrcutions_Block** : le block default.
.

new Pour obtenir un objet instance d'une classe.
Super-classes : Instruction
Attributs : Aucun
Références : classinstanciée de type **JavaClass** : la classe instanciée.

return_Clause Pour spécifier les return
Super-classes : Instruction
Attributs : Aucun
Références : abstractexression de type **AbstractExression** : la classe instanciée.

Redirection_Expression Pour spécifier les redirections
Super-classes : Instruction
Attributs : redirect de type RedirectionKind (énumération)
Références : abstractexression de type **AbstractExression** : la classe instanciée.

instanceOf Permet de vérifier si un objet est instance d'une classe
Super-classes : Instruction
Attributs : Aucun
Références : javavariabledata de type **JavaVariableData**.
classdescriptor de type **ClassDescriptor**

AbstractExpression (abstract class) Permet de spécifier une expression en Java
Super-classes : ClassMember

Attributs : Aucun
Références : Aucune

JavaAbstractData (abstract class) Permet de spécifier les données
Super-classes : AbstractExpression
Attributs : Aucun
Références : Aucune

Literal Un littéral
Super-classes : JavaAbstractData
Attributs : Aucun
Références : Aucune

JavaVairiableData Une variable Java
Super-classes : JavaAbstractData, TypedElement
Attributs : isTransient : Boolean (pour indiquer q'un attribut fait -ou pas- partie de la partie serializable de l'objet)
isVolatile : Boolean (pour indiquer qu'une variable java peut être modifiée par deux threads qui s'exécutent concurremment)
isFinal : Boolean (si sa valeur restera inchangée ou pas)
InitialValue de type any (la valeur initiale de la variable)
Références : Aucune

block_Variable Une variable de portée block
Super-classes : JavaVariableData
Attributs : Aucun
Références : Aucune

LogicExpression (abstract class) Permet de définir toute expression logique (Test qui retourne un boolean)
Super-classes : AbstractExpression
Attributs : Aucun
Références : abstractexpressionright de type **AbstractExpression**
abstractexpressionleft de type **AbstractExpression**

BinaryConditionOperator (abstract class)
Super-classes : LogicExpression
Attributs : Aucun
Références : Aucune

UnaryConditionOperator (abstract class)
Super-classes : LogicExpression
Attributs : Aucun
Références : Aucune

LogicOperator (abstract class)
Super-classes : BinaryConditionOperator
Attributs : Aucun
Références : Aucune

Or L'opérateur logique OU
Super-classes : LogicOperator
Attributs : Aucun
Références : Aucune

And L'opérateur logique ET
Super-classes : LogicOperator
Attributs : Aucun
Références : Aucune

Not L'opérateur logique de négation
Super-classes : UnaryConditionOperator
Attributs : Aucun
Références : Aucune

Comparaison Permet de faire une comparaison entre deux AbstractExpression.
Super-classes : LogicExpression
Attributs : Operator_Kind de type **OperatorKind** (le <, <=, >, >=…etc.)
Références : Aucune

ArithmeticExpression (abstract class) pour les expressions arithmétiques
Super-classes : AbstractExpression
Attributs : Aucun
Références : rightabstractexpression de type **AbstractExpression**
leftabstractexpression de type **AbstractExpression**

BinaryArithmeticExpression Pour toutes les opérations arithmétiques binaires
Super-classes : **ArithmeticExpression**
Attributs : ArithmeticOperator de type BinaryArithmeticOperatorKind
Références : Aucune

UnaryArithmeticExpression Pour toutes les opérations arithmétiques unaires
Super-classes : **ArithmeticExpression**
Attributs : ArithmeticOperator de type UnaryArithmeticOperatorKind
Références : abstractexpressionunary de type **AbstractExpression.**

BitToBitOperator (abstract class) Pour toutes les opérations bit à bit
Super-classes : **ArithmeticExpression**
Attributs : Aucun
Références : Aucune

Xor Le OU exclusif
Super-classes : BitToBitOperator
Attributs : Aucun
Références : Aucune

Or_Inclusive Le OU inclusif
Super-classes : BitToBitOperator
Attributs : Aucun
Références : Aucune

And_BtB Le ET bit à bit
Super-classes : BitToBitOperator
Attributs : Aucun
Références : Aucune

Codebase
Super-classes : Aucune
Attributs : uri de type String.
Références : Aucune

Component (abstract class) tout composant graphique
Super-classes : GraphicElement
Attributs : enable de type Boolean (selon que le composant soit actif ou pas)
Références : Aucune

Panel
Super-classes : Component
Attributs : scrollable de type boolean (pour le défilement)
alignment de type AlignType (pour l'alignement)
Références : Aucune

TextComponent (abstract class)
Super-classes : Component
Attributs : scrollable de type boolean (pour le défilement)
alignment de type AlignType (pour l'alignement)
Références : Aucune

TextField Champ de texte
Super-classes : TextComponent
Attributs : editable : de type boolean
password de type boolean
rowsNumber de type integer
columnsNumber de type Integer
Références : Aucune

Event L'évènement
Super-classes : Component
Attributs : type de type EventType
Références : instruction de type Instruction.

1.1.2. Type

Les Types définis dans ce méta-modèle sont :

ScopeKind : Enumération qui représente la portée (scope) qui peut être soit du niveau Instance, soit static (classifier level)

RedirectionKind : Enumération qui représente les instructions de redirection ou ceux permettant l'interruption de l'exécution en cours pour sortir de la boucle. Exp. *goto, break et continue*

ModifierKind : Enumération qui représente le type l'entité qui peut être : *abstract, regular ou final*

VisibilityKind : Enumération qui représente la visibilité de l'entité qui peut être soit : *private, protected, public ou package*.

EvenType : Enumération qui représente les types d'évènements (onClick, onDoubleClick…etc.)

PrimitiveTag : Enumération qui représente tous types primitifs : *char, double, byte*…etc.

OperatorKind : Enumération qui représente les opérateurs arithmétiques de comparaison : <, >,<=,…etc.

BinaryArithmeticOperatorKind : Enumération qui représente les opérateurs arithmétiques binaires : +,-,/,*…etc.

UnaryArithmeticOperatorKind : Enumération qui représente les opérateurs arithmétiques unaires : ++,--,~…etc.

Boolean : représente le type de base booléen

Integer : représente le type de base Integer

Long : représente le type de base Long

String : représente le type de base String.

1.1.3. Assiotiation

Les associations du méta-modèle Java sont décrites en précisant les deux classes qu'elles joignent ainsi que les multiplicités associées à celles-ci. Pour les multiplicités, la convention utilisée ici est celle définie par le MOF.

LocalClasses	Présente les classes locales.
Source:	Type : ClassMember
	Multiplicité : 1
Destination	Type : JavaClass
	Multiplicité : 0..*
PackageClasses	Les classes contenues dans un package
Source:	Type : JavaPackage
	Multiplicité : 1 (une classe appartient à un seul package)
Destination	Type : JavaClass
	Multiplicité : 0..* (un package contient plusieurs classes)
belongsTo	Le code base auquel appartient la classe.
Source:	Type : JavaClass
	Multiplicité : 0..* (à une uri peut appartenir plusieurs classes)
Destination	Type : Codebase
	Multiplicité : 1 (une classe appartient à un seul uri)

isOfType	Pour décrire le type de l'entité
Source:	Type : TypedElement
	Multiplicité : 0..* (un type peut être attribué à plusieurs entités)
Destination	Type : TypeDescriptor
	Multiplicité : 0..1 (une entité à un seule type)
ElementType	Présente le type des éléments d'un tableau
Source:	Type : ArrayDescriptor
	Multiplicité :0..1 (un type peut correspondre à un ou aucun élément)
Destination	Type : TypeDescriptor
	Multiplicité : 1 (un élément n'a qu'un seul type)
DeclaredThrowables	Présente les classes d'exceptions que l'entité pourrait éventuellement lever
Source:	Type : BehavioralFeature
	Multiplicité :0..* (une exception peut être levée par plusieurs entités)
Destination	Type : TypeDescriptor
	Multiplicité : 0..*(une entité peut lever une ou plusieurs classes d'exceptions)
TextVariable	Présente la variable attachée à un composant Texte (label ou champ)
Source:	Type : TextComponent
	Multiplicité :1 (une variable correspond à un seul composant texte)
Destination	Type : JavaVariableData
	Multiplicité : 0..* (à un composant texte peut être affecté une ou plusieurs variables)

2. L'IHM (Inetrface Homme Machine)

L'interface Homme/Machine faisant partie intégrante de la majorité des applications, le besoin de la spécifier au niveau du pivot s'est tout de suite avéré comme une priorité. Le méta modèle fourni par le partenaire industriel du projet, Sof-Maint est complet dans le sens où il décrit tout ce qui pourrait être nécessaire à la spécification d'une interface graphique indépendamment du choix du langage d'implémentation. L'étape suivante, consiste à identifier le point d'interconnexion du MM IHM avec le modèle Traitement du pivot et le MM Java spécifiant ainsi, L'IHM, le traitement et le comportement sous-jacent.

Le but de cette section est, de présenter l'intégration du méta modèle IHM réalisé par Soft-Maint au méta modèle Traitement de notre pivot.

2.1. Le méta modèle IHM

Dans ce qui suit, nous n'allons pas présenter le MM IHM mais juste les concepts qui ont permis son intégration au MM Traitement. Pour cela, on a considéré le package ***Action*** du MM IHM. Fig. 4.3.

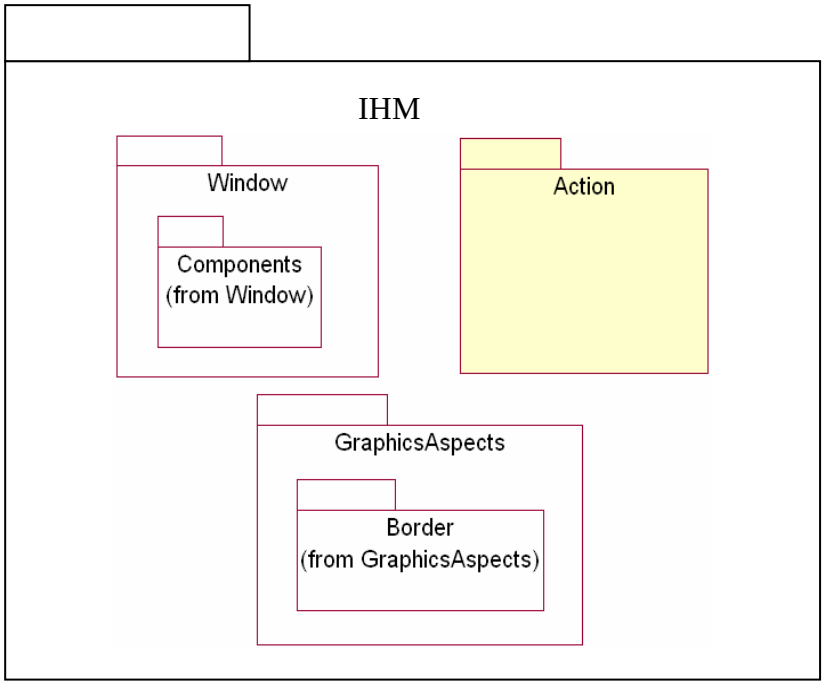


Fig. 4.3 : Package Action du méta modèle IHM

Le méta modèle inclus dans le package **Action** (voir Fig. 4.4.) spécifie, qu'à chaque composant graphique peut être associé plusieurs événements ou le cas échéant aucun. Un événement, une fois déclenché, provoque l'exécution d'une ou plusieurs actions. Une action peut être soit une action interne modifiant l'élément *Panel* ou un composant tel que les boutons, les champs de texte ou bien les listes, soit une action externe agissant sur les valeurs contenues dans les composants de l'interface graphique.

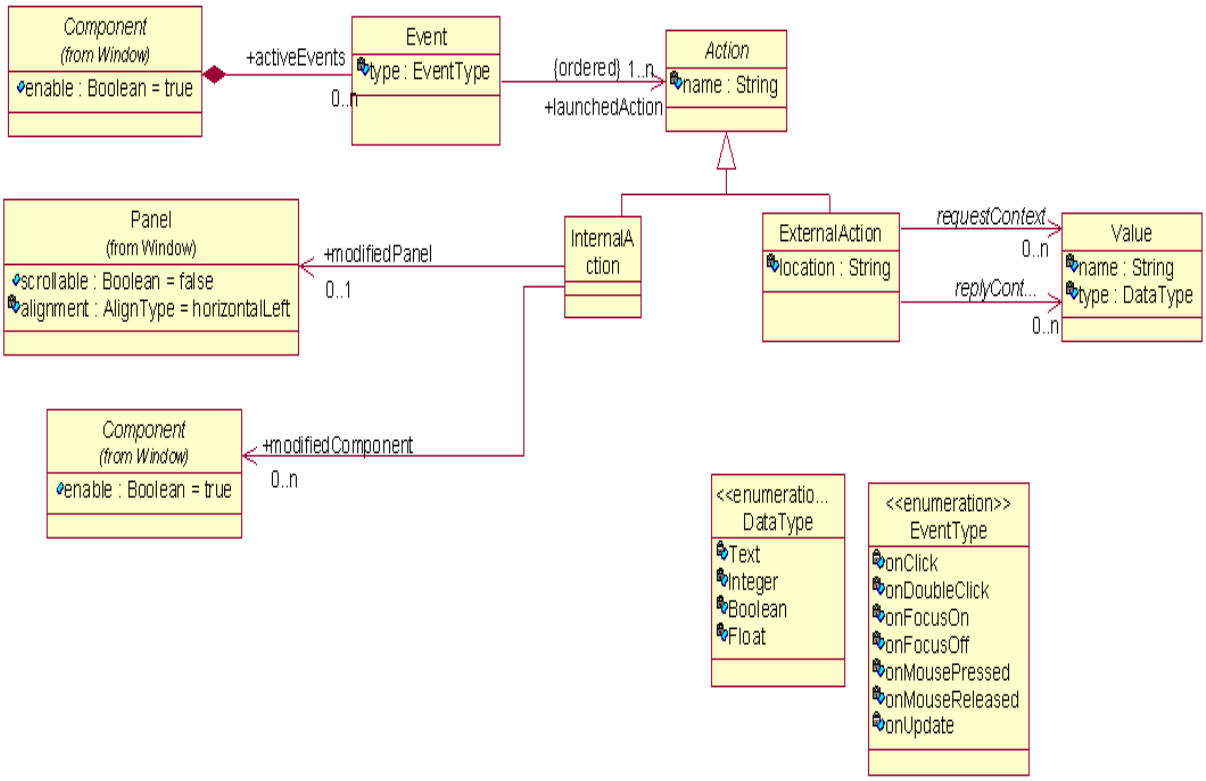


Fig. 4.4. : Lien entre les actions et les composants de l'interface graphique

2.2. Intégration du MM IHM au MM Traitement

Comme vous pourriez l'imaginer, le point de connexion entre les deux méta-modèles ne pourrait être autre que le concept d'Action. En effet, nous avons remplacé celui défini dans le MM IHM par celui défini dans le package *Action Foundation* du standard ***A**ction **S**emantics*. Ce dernier étant plus complet et surtout plus détaillé, couvre tous les cas de figures. Du fait que tous les packages définis dans *AS* héritent de *Action Foundation*, une action pourra alors être soit : un groupe d'actions, un appel de méthode avec *CallOperationAction* du package *Messaging Action* (exp. Pour activer l'affichage d'une fenêtre, appel d'une fonction de calcul sur les valeurs contenues par les champs de texte ou des méthodes propres aux composants afin d'agir sur leurs aspects graphiques), la création d'objets avec l'action *CreateObjectAction* de *ObjectAction* (permettant l'instanciation de nouvelles fenêtres ou composants)…etc.

De ce fait, il s'avère inutile de garder les éléments *InternalAction* et *ExternalAction* qui spécialisaient *Action* dans le MM IHM.

Pour ce qui est de l'élément *Value* qui représente soit les valeurs à passer en paramètres à l'action, soit les résultats obtenus par l'exécution de celle-ci, il correspond aux concept de *Pin* défini dans Action Semantics. L'élément *Pin* est spécialisé en *Input pin* et *Output pin* qui correspondent respectivement aux paramètres et résultats de l'action. Donc, le concept de *Pin* remplace celui de *Value* dans le nouveau méta-modèle.
Le méta-modèle résultant de cette intégration se présente comme suit :

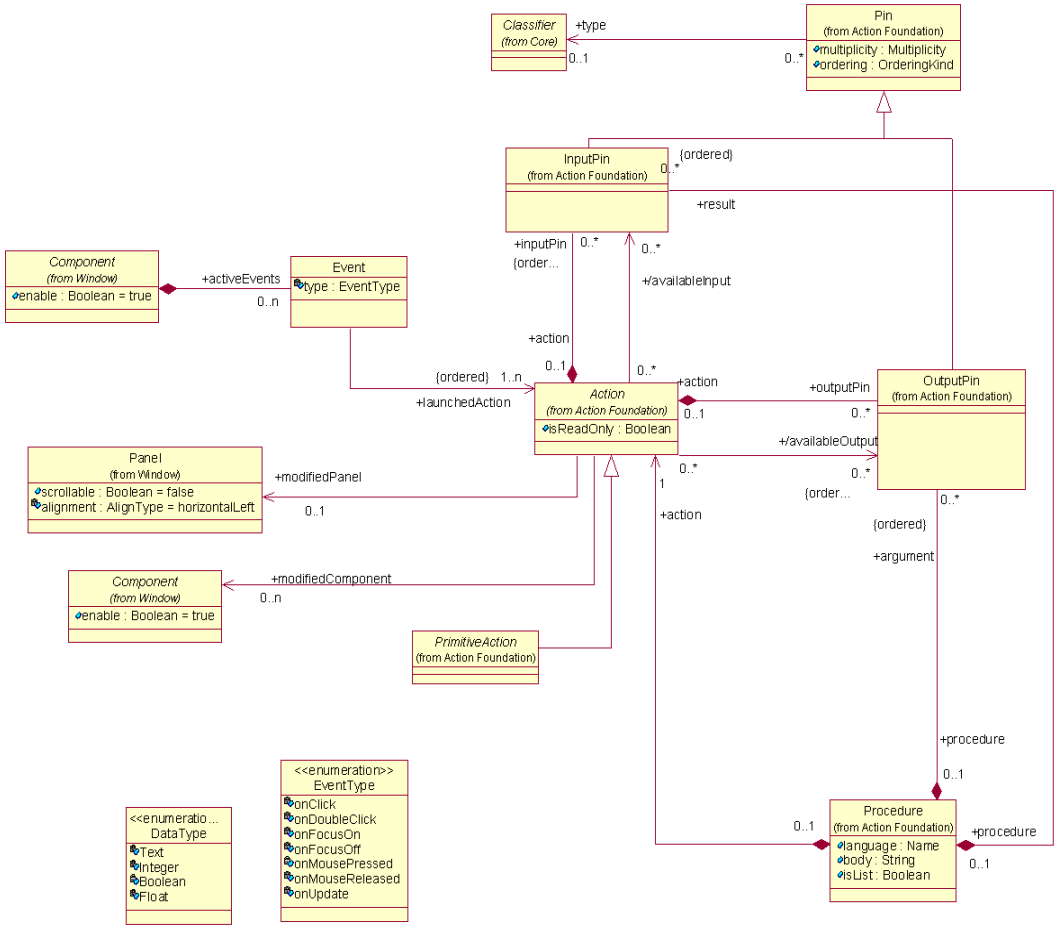


Fig. 4.5 : Intégration du MM IHM au MM Traitement

2.3. Détails du méta-modèle IHM/Traitement

Par souci de lisibilité et afin d'éviter de renvoyer le lecteur vers d'autres documents, nous avons préféré reprendre ici, l'explication ainsi que les détails des classes et des associations du méta-modèle IHM/Traitement. Bien sûr, nous n'aborderons que les concepts dont nous nous sommes servis pour notre couplage.

2.3.1 Classes

Action

Unité fondamentale de spécification du comportement. Une action prend un ensemble de valeurs en Input, les utilise et fournit en Output des valeurs correspondantes à l'exécution de l'action. Les ensembles d'Input et d'Output peuvent être nuls.

Super Classe *saucune*

Attributs

isReadOnly Booléen s'il est égal à Vrai, l'action ne modifie aucune variable en dehors de l'action

2.3.2. Associations

availableInput	InputPin [0..*]	L'ensemble de tous les inputs fournis en entrée à l'action
availableOutput	OutputPin [0..*]	L'ensemble de tous les outputs en sortie de l'action
inputPin	InputPin [0..*]	L'ensemble ordonné d'inputs contenu par l'action agissant comme points de connexions, procurant ainsi les valeurs consommées par l'action. Il est à noter qu'une action peut être aussi bien primitive que complexe (un groupe d'action)
outputPin	OutputPin [0..*]	L'ensemble ordonné d'outputs contenu par l'action agissant comme points de connexions pour l'obtention des valeurs générées par l'action. Il est à noter qu'une action peut être aussi bien primitive que complexe (un groupe d'action)

Event

Les éléments "Event" correspondent aux événements pouvant être associés aux composants (Boutons, Listes déroulantes, Cases à cocher, etc.).

Super Classes

aucun

Attributs

EventType Détermine le type d'événement représenté par cet objet. Exemple : Clique, Double clique, Composant activé, etc.

Références

LaunchedAction Action Liste ordonnée d'actions à effectuer lorsque l'événement se produit.

2.3.3. Types

DataType

Liste de types de données pouvant être passées en paramètre.

Valeurs

Text, Integer, Boolean, Float.

EventType

Liste de types d'évènement.

Valeurs

onClick, onDoubleClick, onFocusOn, onFocusOff, onMousePressed, onMouseReleased, onUpdate.

Vous trouverez en **annexe 3**, le méta-modèle complet (Traitement RM-ODP/Action Semantics/IHM) avec :

- En couleur beige : les concepts propres à Traitement RM-ODP ;
- En jaune : les concepts liés à Action Semantics ;
- En bleu : l'IHM (juste les concepts utilisés pour le couplage)
- En blanc : les points d'interconnexion entre les différents méta-modèles.

Pour le méta-modèle Java, le principe est le même, notre point de connexion entre le méta-modèle Java et le méta-modèle IHM se situe au niveau du concept **Instruction** (voir le méta-modèle complet **en annexe 3**). Une instruction étant lancée en réaction à un déclenchement d'un événement, lui-même rattaché à un composant graphique.

3. Correspondances entre le MM Traitement RM-ODP/ActionSemantics UML/IHM et le MM Java/IHM

Une fois que nous avons spécifié les détails des instructions, l'IHM ainsi que le traitement sous-jacent au niveau du Pivot et de la cible (Java), il nous est maintenant possible de prétendre à une migration complète du code. La table qui suit (Voir Table 4.1) présente les correspondances nouvellement identifiées entre les MM Traitement de notre Pivot et le MM Java que nous avons spécifiés.

RM-ODP/ActionSemantics/IHM Java	MM Java
Event	Event
TextField	TextField
Panel	Panel
Template	JavaClass avec isInterface = False
ComputationalObject	Objet Java
OperationInterface avec l'attribut Causality = Serveur	JavaClass avec isInterface = True
Method	Method
	Main_Method
	Constructor avec comme contrainte que le nom du constructeur soit le même que celui de la classe
GroupAction	Instruction_Block
Variable	block_Variable
ControlFlow & DataFlow servent à déterminer l'ordre d'exécution des actions	Ce qui correspond à la relation "***nextInstruction***"
OutputPin & InputPin héritent de Pin mais ne contiennent pas d'informations supplémentaires. Du coup la correspondance se fait directement avec la Classe **Pin**	JavaVariableData & Literal qui héritent **JavaAbstractData**
ArgumentSpecification	
Attribute	
Classifier une classe abstraite Dont héritent **Class** et **DataType**. Ici l'élement Class correspond à Class Descriptor	ClassDescriptor. L'élément ArrayDescriptor qui en héritent permet de traiter les tableaux (un peu spéciaux en Java)
DataType	PrimitiveType
Parameter	FeatureParameter
ConditionalAction	If
	Switch
LoopAction	While
	For
Clause : L'élément Clause est composé d'un "**test**" qui est une **Action** (l'élément) et d'un "**Body**" qui est aussi une Action (ou un GroupAction). Donc l'Action du "test" correspond à LogicExression de Java du fait que ça retourne un Boolean.	Le test dans une structure conditionnelle est une expression logique (l'élément **LogicExpression**) et donc peut être tout élément qui héritent de LogicExpression avec les relations qui existent entre ces élément : Or, And, Comparaison, Not et qui à leurs tour utilisent les ArithmeticExpression
	Case : comme en AS une Structure conditionnelle peut comprendre plusieurs Clauses, chaque clause peut correspondre à un Case du Switch.

Action	Instruction (dans le principe même si c'est une classe abstraite
PrimitiveFunction : AS n'entre pas dans les détails des : Or, and et les opérations arithmétique dans le sens où ils peuvent être implémentés différemment selon le language (le Or ou le and peuvent être des méthodes dans d'autres languages avec des inputs et un output)	Or
	And
	Comparaison
	Xor
	Or_Inclusive
	And_BtB
	UnaryArithmeticExpression
	BinaryArithmeticExpression
	InstanceOf
CreateObjectAction	new
Procedure	correspond au corps d'une methode et correspond le plus à l'élément Instructions_Block
JumpAction	throw
	Redirection_Expression
AddVariableValueAction	Ces actions correspondent à une UnaryArithmeticExpression avec l'attribut UnaryArithmeticOperator = à ++ ou -- selon qu'on incremente ou qu'on decremente la variable de 1. Ou bien elles peuvent aussi bien correspondre à une BinaryArithmeticExpression avec l'attribut BinaryArithmeticOpérator = Assignation avec comme arguments la variable à laquelle on veut rajouter/enlever une valeur et la valeur en question
RemoveVariableValueAction	
AddAttributeValueAction	
RemoveAttributeValueAction	
ClearVariableAction	ces actions se font implicitement
ReadVariableAction	
ReadAttributeAction	
ClearAttributeAction	
DestroyObjectAction	
ReclassifyObjectAction	
ReadClassifiedObjectAction	
JumpHandler	
HandlerAction	
Operation	Cet élément de contient pas d'information pértinante.
State	Correspond à la valeur des attributs de l'objet à un instant donné
BindingObject	/
Flow	/
Signal	/
SendSignalAction	/
SignalInterface	/
FlowInterface	/
Type	/
EnvironmentContract	/
/	JavaPackage
/	Initializer
/	return_Clause
/	try

4. Conclusion

Le méta modèle Traitement de notre pivot est maintenant complet et assure la spécification de tous les aspects que se soit ceux liés à l'architecture logicielle de l'application, son interface graphique ou bien le traitement ainsi que le comportement sous-jacent à celle-ci. Reste maintenant à le tester et voir ce que cela donne réellement en pratique. Le méta-modèle Java que nous avons spécifié, est lui aussi complet et se place au même niveau d'abstraction que celui du méta-modèle Traitement permettant ainsi, une migration du code de la source vers la cible en passant par le pivot, sans perte d'informations.

Chapitre 5 Etude de Cas.

Une fois les méta-modèles Traitement de notre Pivot et celui du langage Java spécifiés, les correspondances identifiées, il nous restait à valider nos travaux par un exemple concret. La tâche de spécification des règles de transformation ainsi que le transformateur étant à la charge du partenaire industriel SofMaint, nous avons jugé préférable de le faire nous-même. Cela nous permettrait d'une part de tester nos résultats et d'autre part de vérifier le bon fonctionnement du transformateur ainsi que le langage de transformation de modèles que nous avons utilisé. Ce dernier, nommé SimpleTRL [PRA 03], a été conçu dans le cadre d'un stage de DEA au sein de l'équipe SRC du LIP6.

Dans ce qui suit, nous allons commencer par présenter le langage de transformation de modèles SimpleTRL ainsi que le méta-modèle correspondant.

1. SimpleTRL

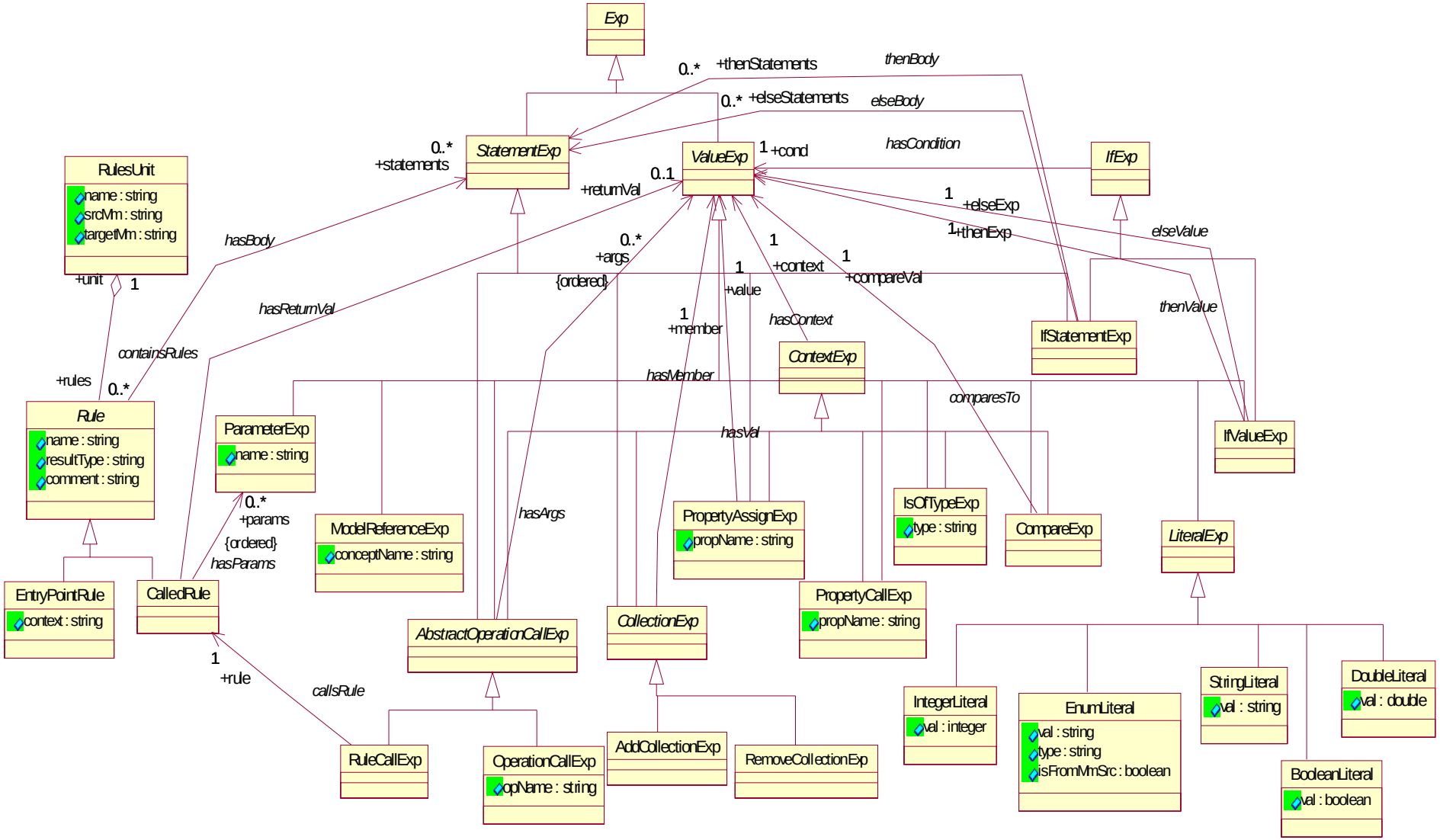
Simple TRL est un langage de transformation de modèles inspiré de la proposition française TRL (**T**ransformation **R**ules **L**anguage) [TRL 02] suite à l'RFP (Request For Proposal) MOF 2.0 QVT [QVT 02] de l'OMG. Une transformation de modèles consiste en une activité de production d'un modèle cible à partir d'un modèle source.

1.2. Méta-modèle Simple TRL

La structure du formalisme SimpleTRL est spécifiée sous forme d'un méta-modèle MOF [OMG 00], chaque concept de ce dernier est représenté par une classe dans le méta-modèle. Une instance de ce méta-modèle serait une spécification des correspondances entre les concepts d'un MM source et ceux d'un MM cible. SimpleTRL est accompagné d'une syntaxe textuelle permettant d'exprimer les règles de transformation de façon plus lisible par rapport à la forme objet ou la syntaxe XMI.

Le Méta-modèle SimpleTRL se présente comme suit :

Le Méta-modèle SimpleTRL.



Au risque de déborder du cadre de notre rapport, nous ne détaillerons pas les concepts de ce méta-modèle. Vous trouvez plus d'information dans [PRA 03].

2. La Transformation de modèles

Approche

Selon la vision de l'MDA [OMG 01], une transformation de modèles se divise en deux étapes :

- La première étape consiste à spécifier les règles de transformation. Ces dernières se basent sur les correspondances identifiées entre les concepts du MM source et les concepts du MM cible.

- La deuxième étape consiste à appliquer les règles de transformation sur le modèle source (conforme au méta-modèle de la source) afin de produire le modèle cible (lui aussi conforme au méta-modèle de la cible).

Dans notre cas, nous devons passer par deux transformations de modèles (Fig. 5.1). La première vise à transformer le modèle source COBOL vers un modèle pivot et la deuxième, d'un modèle pivot vers le modèle cible correspondant en JAVA.

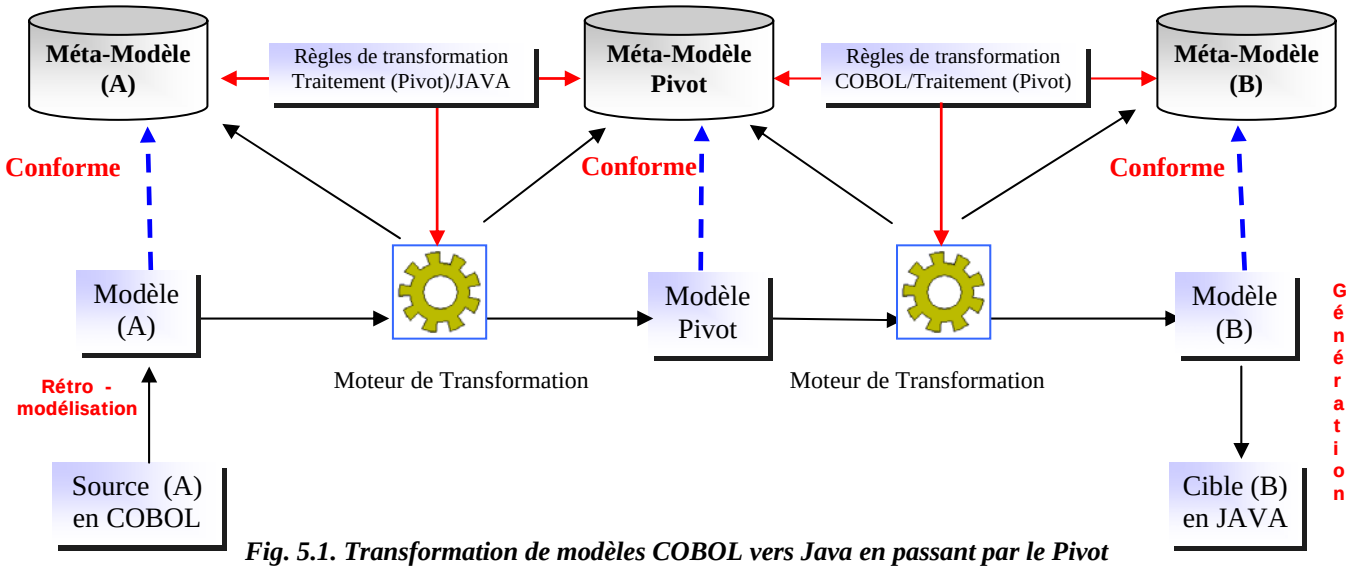


Fig. 5.1. Transformation de modèles COBOL vers Java en passant par le Pivot

Inputs

Les inputs de notre démarche sont :

1- Les méta-modèles Source (COBOL), Pivot et Cible (Java).
2- Un modèle Source (COBOL) conforme au méta-modèle de la Source (COBOL).
3- Les règles de transformation établies à partir des correspondances identifiées entre les concepts des méta-modèles (Source => Pivot => Java).

Résultat

Le résultat de la transformation de modèle est un modèle Cible (Java) conforme au méta-modèle de la Cible.

3. Exemple

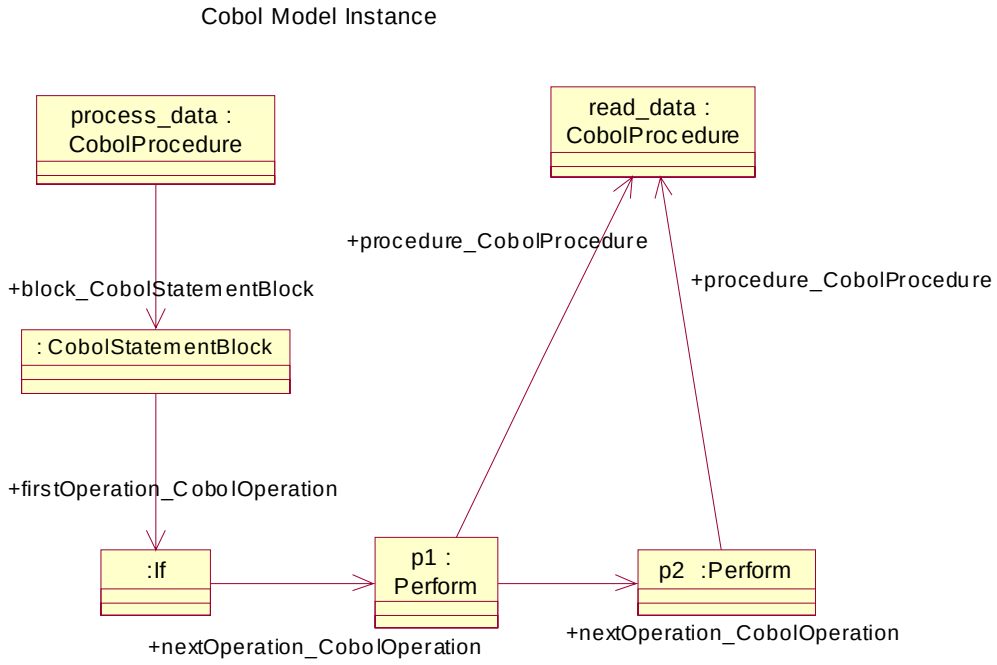
L'exemple que nous avons réalisé est très simple. Il a pour but ici étant juste d'illustrer notre démarche et de concrétiser les résultats de nos travaux.

Inputs

1)- Les méta-modèles que nous avons utilisés sont présentés en annexes :

- Le méta-modèle COBOL (annexe 1)
- Le méta-modèle Traitement RM-ODP/ActionSemantic/IHM (annexe 3)
- Le méta-modèle Java (annexe 3).

2)- Le modèle COBOL conforme au méta-modèle COBOL (Fig. 5.2) :



Grossièrement, ce modèle décrit qu'une méthode (Procédure en COBOL) nommée *process_data* est composée d'un block (CobolStatementBlock) qui correspond à son implémentation. Ce block contient une instruction conditionnelle : le *If*. Les instructions qui suivent le *If* sont des appels de méthodes : le *Perfom*. La méthode appelée est : read_data (concept CobolProcedure sur le MM COBOL).

3) – Les règles de transformation que nous avons spécifiées se basent sur les correspondances que nous avons identifiées comme le montre le tableau suivant (Table 5.1). Les concepts en gras et de couleur bleu sont ceux que nous avons utilisé dans notre exemple.

COBOL	RM-ODP/ActionSemantics/IHM Java	MM Java
CobolEvent_OtherEvent, CobolEvent_NextScreen, CobolEventValidation	Event	Event
BMSField	TextField	TextField
BMSMap	Panel	Panel
CobolProgram / CobolNestedProgram	ComputationalObjectTemplate	JavaClass avec isInterface = False
/	ComputationalObject	Objet Java
/	OperationInterface avec l'attribut Causality = Serveur	JavaClass avec isInterface = True
CobolProcedure (Paragraphe)	**Method**	**Method**
		Main_Method
		Constructor avec comme contrainte que le nom du constructeur soit le même que celui de la classe
CobolStatementBlock	**GroupAction**	**Instruction_Block**
/	Variable	block_Variable
/	ControlFlow & DataFlow servent à déterminer l'ordre d'exécution des actions	Ce qui correspond à la relation "***nextInstruction***"
/		
CobolVariableData	OutputPin & InputPin héritent de Pin mais ne contiennent pas d'informations supplémentaires. Du coup la correspondance se fait directement avec la Classe **Pin**	JavaVariableData & Literal qui héritent **JavaAbstractData**
	ArgumentSpecification	
	Attribute	
/	Classifier une classe abstraite Dont héritent **Class** et **DataType**. Ici l'élément Class correspond à Class Descriptor	ClassDescriptor. L'élément ArrayDescriptor qui en héritent permet de traiter les tableaux (un peu spéciaux en Java)
/	DataType	PrimitiveType
/	Parameter	FeatureParameter
If	**ConditionalAction**	**If**
		Switch
While	LoopAction	While
		For
Condition	**Clause** : L'élément Clause est composé d'un "**test**" qui est une **Action** (l'élément) et d'un "**Body**" qui est aussi une Action (ou un GroupAction). Donc l'Action du "test" correspond à LogicExression de Java du fait que ça retourne un Boolean.	Le test dans une structure conditionnelle est une expression logique (l'élément **LogicExpression**) et donc peut être tout élément qui héritent de LogicExpression avec les relations qui existent entre ces élément : Or, And, Comparaison, Not et qui à leurs tour utilisent les ArithmeticExpression
		Case : comme en AS une Structure conditionnelle peut comprendre plusieurs Clauses, chaque clause peut correspondre à un Case du Switch.

	CallOperationAction avec l'attribut isSynchronized egal à True ou False selon que l'appel est synchrone ou pas. ApplyFunctionAction (dans le cas ou on fait appel à des fonctions primitives sur les string exp. comparTo(), equals()…)	CallMethodInstruction
Perform et GoTo		**CallMethodInstruction**
CobolOperation	**Action**	**Instruction (dans le principe même si c'est une classe abstraite)**
Plus	PrimitiveFunction : AS n'entre pas dans les détails des : Or, and et les opérations arithmétique dans le sens où ils peuvent être implémentés différemment selon le language (le Or ou le and peuvent être des méthodes dans d'autres languages avec des inputs et un output)	Or
		And
Minus		Comparaison
		Xor
		Or_Inclusive
Times		And_BtB
		UnaryArithmeticExpression
Quotient		BinaryArithmeticExpression
		InstanceOf
/	CreateObjectAction	new
/	**Procedure**	correspond au corps d'une methode et correspond le plus à l'élément Instructions_Block
Error	JumpAction	throw
		Redirection_Expression
/	AddVariableValueAction	Ces actions correspondent à une UnaryArithmeticExpression avec l'attribut UnaryArithmeticOperator = à ++ ou -- selon qu'on incremente ou qu'on decremente la variable de 1. Ou bien elles peuvent aussi bien correspondre à une BinaryArithmeticExpression avec l'attribut BinaryArithmeticOpérator = Assignation avec comme arguments la variable à laquelle on veut rajouter/enlever une valeur et la valeur en question
/	RemoveVariableValueAction	
/	AddAttributeValueAction	
/	RemoveAttributeValueAction	
/	ClearVariableAction	ces actions se font implicitement
/	ReadVariableAction	
/	ReadAttributeAction	
/	ClearAttributeAction	
/	DestroyObjectAction	
/	ReclassifyObjectAction	
/	ReadClassifiedObjectAction	
/	JumpHandler	
/	HandlerAction	
/	**Operation**	Cet élément de contient pas d'information pértinante.
/	State	Correspond à la valeur des attributs de l'objet à un instant donné
/	BindingObject	/
/	Flow	/
/	Signal	/
/	SendSignalAction	/

/	SignalInterface	/
/	FlowInterface	/
/	Type	/
/	EnvironmentContract	/
/	/	JavaPackage
/	/	Initializer
/	/	return_Clause
/	/	try

Table 5.1 Correspondances COBOL/IHM => Traitement RM-ODP/AS/IHM => Java/IHM

Il est à noter qu'un concept d'un méta-modèle source peut correspondre à un ou plusieurs concepts dans le méta-modèle cible et vice versa. Exp. le concept *Cobolprocedure* en COBOL correspond à trois concepts dans le Pivot : *Operation* (la signature de la méthode), *Method* (qui est sémantiquement parlant la méthode) et la *Procedure* (qui représente l'implémentation de la méthode). Ces derniers concepts correspondent à un seul concept en Java : *Method*.

Les règles de transformation se basent sur ces correspondances. Elles sont écrites selon la syntaxe propre à SimpleTRL [PRA 03] et conformes au méta-modèle de SimpleTRL. Nous donnons ici des exemples de règles de transformation sans trop entrer dans les détails car cela sortirait largement du cadre de notre rapport. Les règle présentées ci-dessus permettent la transformation de l'élément *CobolProcedure* du MM COBOL vers l'élément *Method* du MM Java en passant par les trois élément *Operation, Method et Procedure* du MM Traitement.

Règles de transformation COBOL => Pivot

Règle 1 :
EntryPointRule CreateOperations
context # CobolProcedure {
#context.Operation_frm_CobolProcedure ()

EntryPointRule correspond à notre point d'entrée. Elle prend en paramètre un contexte (un élément du méta-modèle cible) sur lequel s'appliqueront récursivement d'autres règles. Le contexte ici est l'élément *CobolProcedure* et la règle invoquée pour le transformer est *Operation_frm_CobolProcedure().*

Règle 2 :
Rule Operation_frm_CobolProcedure( )
creates Operation
return #result {
# result.name = # context.name_string
# result.specify = # context.Methode_frm_CobolProcedure ( )

Cette règle prend tout élément *CobolProcedure* et crée un élément *Operation* dans le modèle cible. Une fois l'élément crée, le nom de la procédure lui est affecté. A partir du rôle "specify", la règle *Method_frm_CobolProcedure()* est appelée afin de créer l'élément *Method* ainsi que l'association qui le lie à Operation dans le modèle cible.
Règle 3 :

Rule Method_frm_CobolProcedure()

```
 creates Method
 return #result {
   #result.name = #context.name_string
   #result.body = #context.Procedure_frm_CobolProcedure()
```

Cette règle crée un élément *Method* dans le modèle cible, lui affecte un nom correspondant au même nom de l'élément contexte à partir du quel la règle a été appelée. A partir du rôle "body", la règle *Procedure_frm_CobolProcedure()* est invoquée afin de créer l'élément *Procedure* ainsi que l'association qui le lie à *Method* dans le modèle cible.

Règle 4 :

```
Rule Procedure_frm_CobolProcedure
 creates Procedure
 return #result {
   #result.name = #context.name_string
   #result.action
#context.block_CobolStatementBlock.GroupAction_frm_CobolStatementBlock()
```

Enfin, cette règle crée l'élément Procédure et appel une autre règle que nous ne détaillerons pas.

Le résultat de cette première transformation se présente sous la forme du modèle suivant (voir Fig. 5.3) et qui est conforme au MM Traitement de notre Pivot.

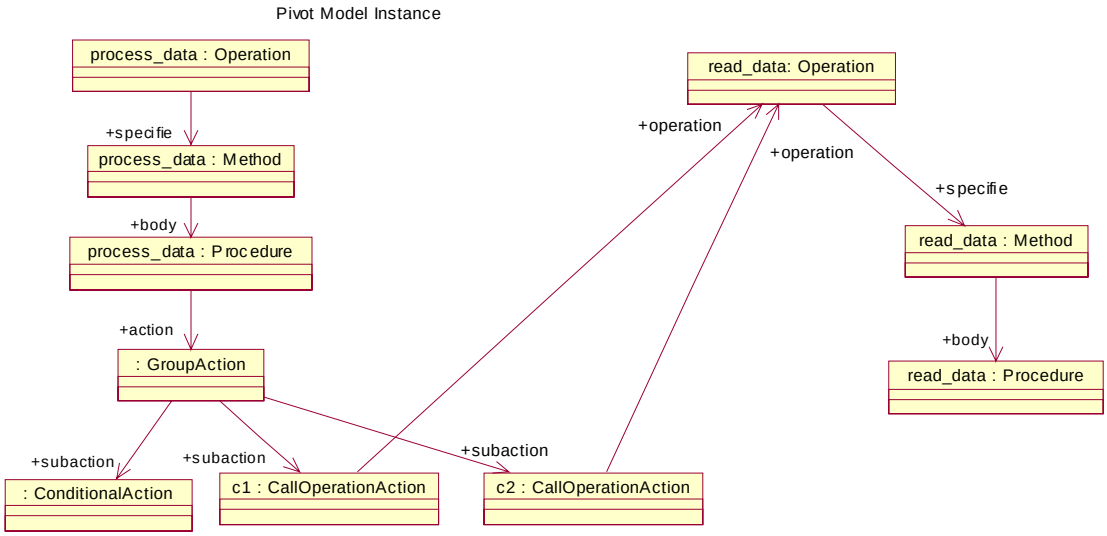


Fig. 5.3. Le modèle Traitement du Pivot résultat de la transformation COBOL =>Traitement

Règles de transformation Pivot => Java

Règle 1 :

```
EntryPoint CreateMethods()
  context Method
  return #result {
    #context.Method_frm_MethodPivot()
```

Cette règle est le point d'entrée de notre transformation Pivot=> Java. Elle prend tout élément *Method* du modèle pivot et applique récursivement la règle *Method_frm_MethodPivot()* sur ce dernier.

Règle 2 :

```
Rule Method_frm_MethodPivot()
  creates Method
  return #result {
    #result.name = #context.name
    #result.contains = #context.body.action.Instructions_Block_frm_GroupAction()
```

Cette règle crée un élément *Method* dans le modèle cible java pour tout élément *Method* du Pivot. L'instruction "context.body.action.Instructions_Block_frm_GroupAction()" traverse l'élément *Procedure* du MM Pivot à travers le rôle "body" pour récupérer directement les instructions (l'élément GroupAction) contenues dans la procédure. L'élément *Procedure* est ignoré du fait qu'il n'y a pas de correspondance entre ce dérnier et un élément dans le MM Pivot.

Résultat

Le modèle Java résultant de la transformation se présente comme suit :

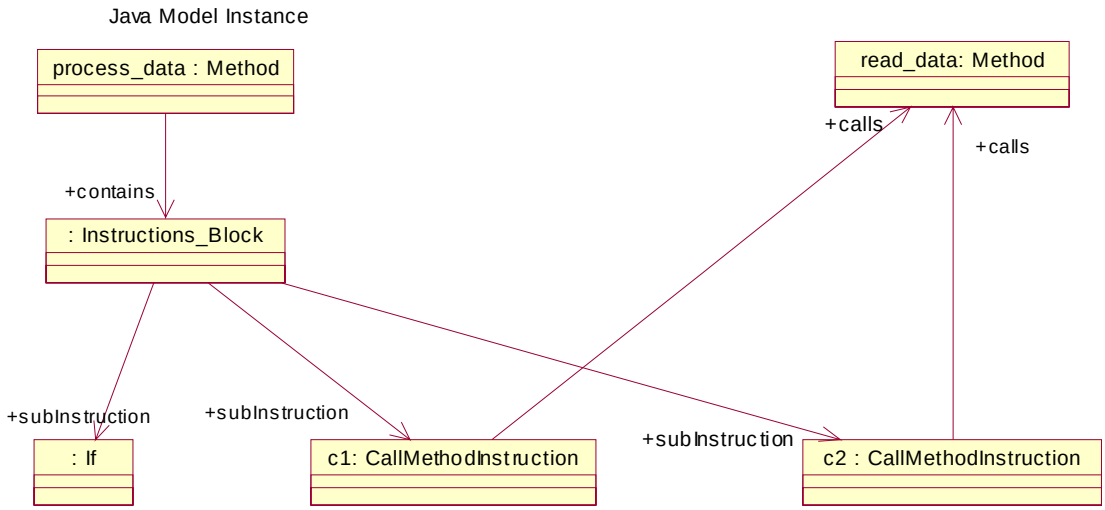


Fig. 5.4. Le modèle Java résultant de la transformation COBOL =>Pivot =>Java

4. Conclusion

Le travail présenté dans cette section avait pour objectif de tester la validité de nos travaux. Pour cela , nous avons présenté l'approche adoptée pour la transformation de modèles ainsi que langage que nous avons utilisé. Nous avons donné un exemple de transformation de modèles se basant sur les méta-modèles que nous avons spécifiés, les correspondances que nous avons identifiées entre ces méta-modèle et un modèle source fourni comme exemple. Cet exercice nous a permis de valider et nos travaux et les travaux se relatant à un autre stagiaire de l'équipe SRC du LIP6.

Bibliographie

[Ank 01] T.S. Ankrum, COBOL Nested Programs, 2001

[BEN 03] R.Bendraou, Modèle Pivot, Livrable Trams Lot2.3 Modèle Pivot V1.7 juin 2003

[BUD 99] Mo Budlong, Teach Yourself COBOL in 21 Days, Sams, 1999, http://cobolreport.com/columnists/mo/07172000.htm

[CHI 01] A.Chibani. Rapport de stage de DEA. Titre : Instrumentation de la Méthodologie ODACpour une Spécification Comportementale. DEA SIR Promotion 2001

[GER 01] M. P. Gervais et X. Blanc, Interopérabilité de spécifications, Revue des sciences et technologies de l'information (RSTI) - série L'Objet, Vol. 8, n° 4, 2002, pp121-144, 2002

[JEZ 02] : G. Sunyé, A. LeGuennec, and J.-M. Jézéquel. -- Using UML action semantics for model execution and transformation. -- *Information Systems, Elsevier*, 27(6):445--457, July 2002

[LEY 97] Leydekkers Peter: Multimedia Services in Open Distributed Telecommunications Environments 1997. CTIT.

[[NET] NetBeans communty, http://java.netbeans.org/models/java/java-model.html

MOE 82] G. Moeckli, COBOL Manuel de Programmation, Université de Genève, 1982

[OMG 00] OMG "Meta-Object Facility (MOF) Specification v1.3", Object Management Group, Mars 2000, OMG. TC Document formal/00-04-03. www.omg.org

[OMG 01] OMG, "Model Driven Archiecture (MDA)", Object Management Group, juillet 2001, OMG TC Document ormsc/2001-07-01. www.omg.org

[OMG 02] OMG, "OMG Unified Modeling Language Specification", Objetc Management Group, Janvier 2002, OMG TC Document UML1.4 (Action Semantics) / 02-01-09, www.omg.org

[ODP 95] ISO/IEC IS 10746-x — ITU-T Rec. X90x, *RM-ODP Reference Model Part x*, 1995

[ODP 02] ISO/IEC JTC1/SC7 FDIS 15414, *RM-ODP Reference Model : Enterprise Viewpoint*, Mai 2002

[PRA 03] Prawee SRIPLAKICH. Rapport de stage de DEA. Titre : Techniques de transformations de modèles basées sur la méta-modèlisation. DEA SIR 2003.

[PUT 01] J.R. Putman, *Architecting with RM-ODP*, Prentice Hall PTR, 2001

[QVT 02] OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10. Document disponible sur le site www.omg.org

[RNT 01] site Web du projet : http://www.telecom.gouv.fr/rntl/AAP2001/Fiches_Resume/TRAMS.htm

[TRA 01] Annexe technique du projet TRAMs V4 08 Novembre 2001.

[TRA 02a] Trams Lot1 Définition du processus de migration mai 2002

[TRA 02b] Trams Lot2.3 Modèle Pivot V1.5 juillet 2002.

[TRA 02c] TRAMs/Lot2.3/DT/ModèlePivotV1.6

[TRL 02] Soumission à l'RFP OMG ad/2003-08-05. Soumission initiale 03/07/2002

[TRA 03] Trams Lot 4.1. Méta Modèle COBOL fournit par SODIFRANCE dans le cadre du projet. Réf. TRAMs/Lot4.1/DT/Méta-modèleSource-BMS-Cbl_v2.0 25 février 2003.

News : comp.lang.cobol, fj.comp.lang.cobol

Annexe 1

Méta Modèle JAVA proposé par NetBeans.

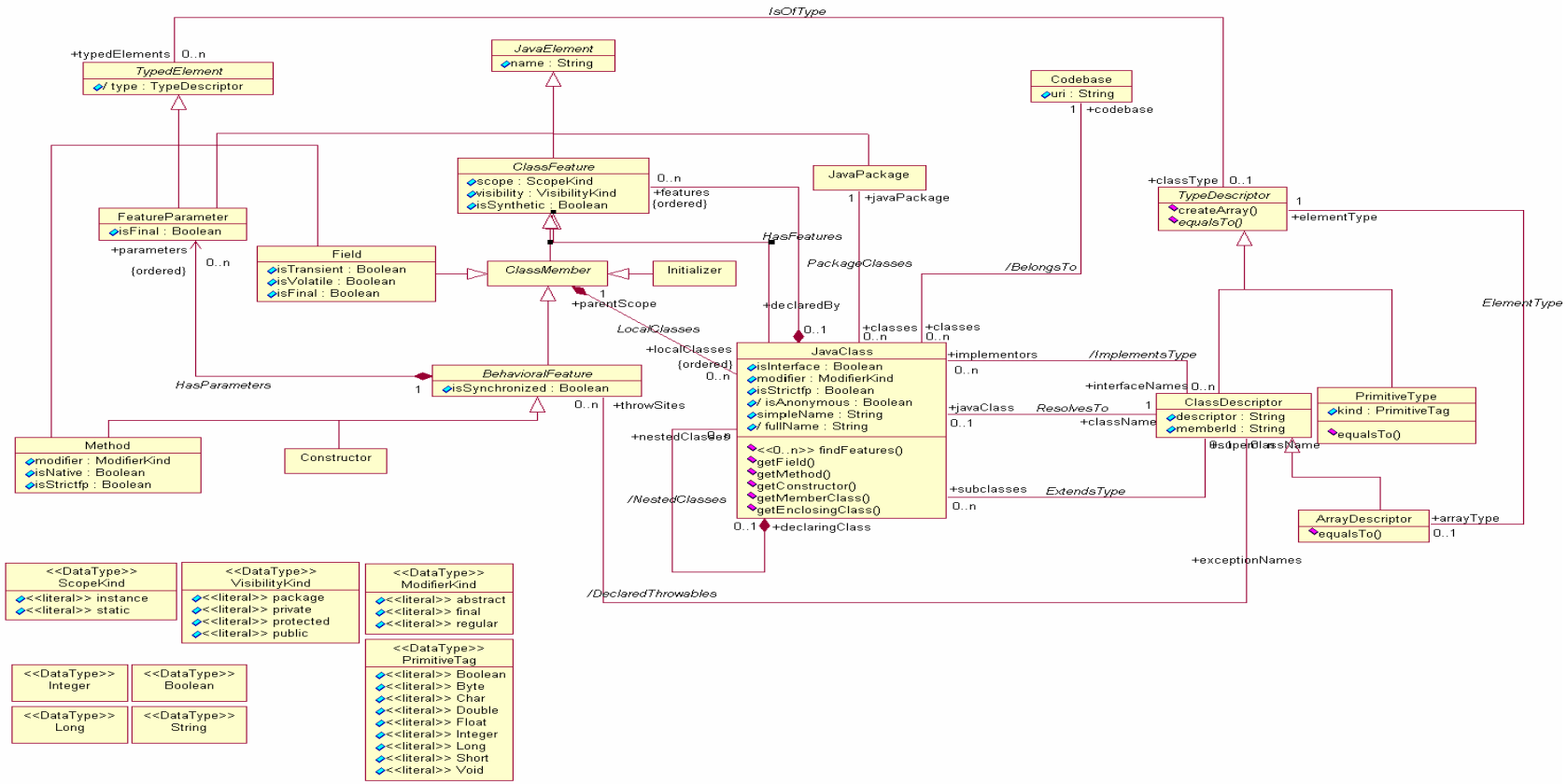
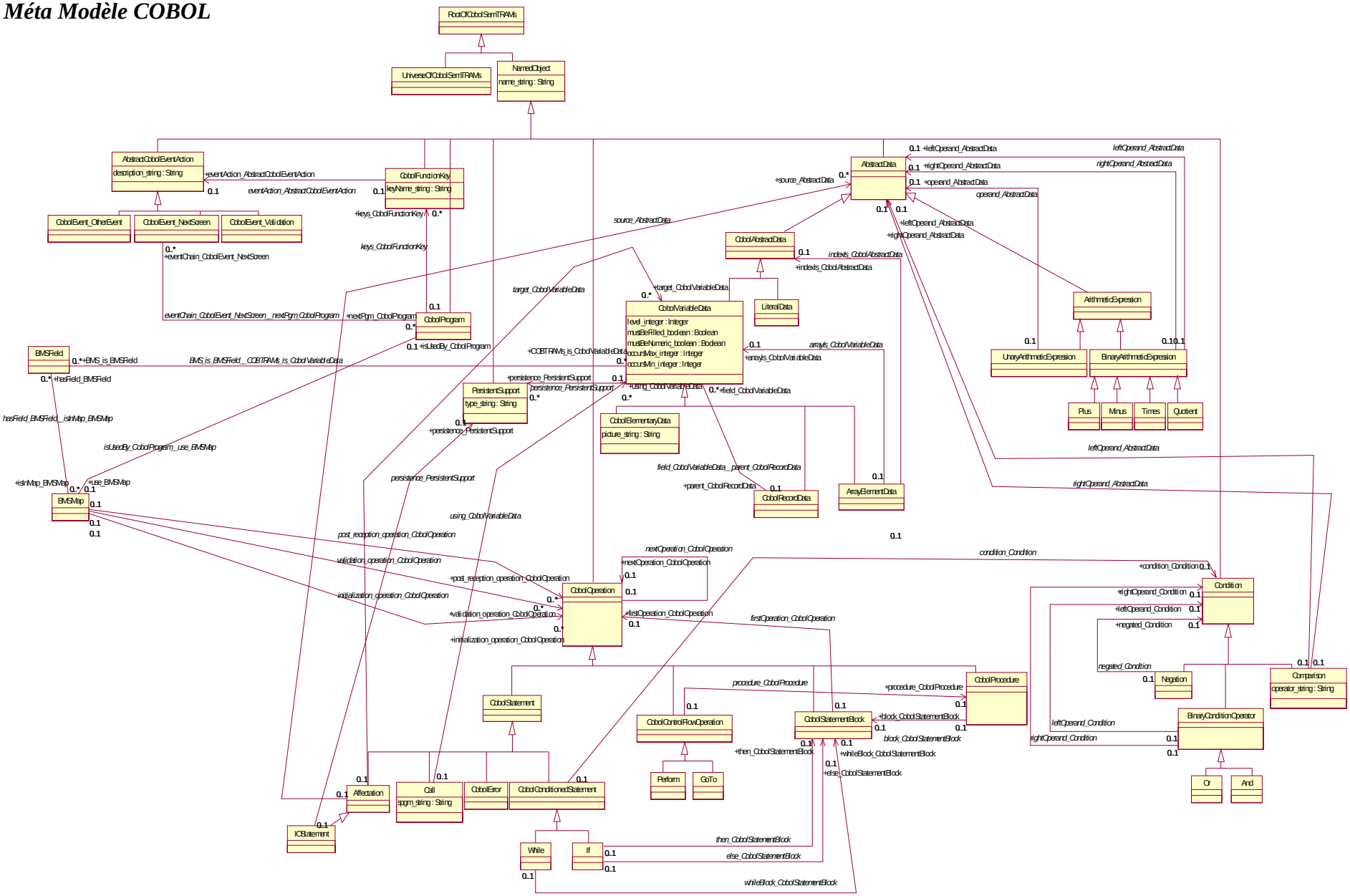


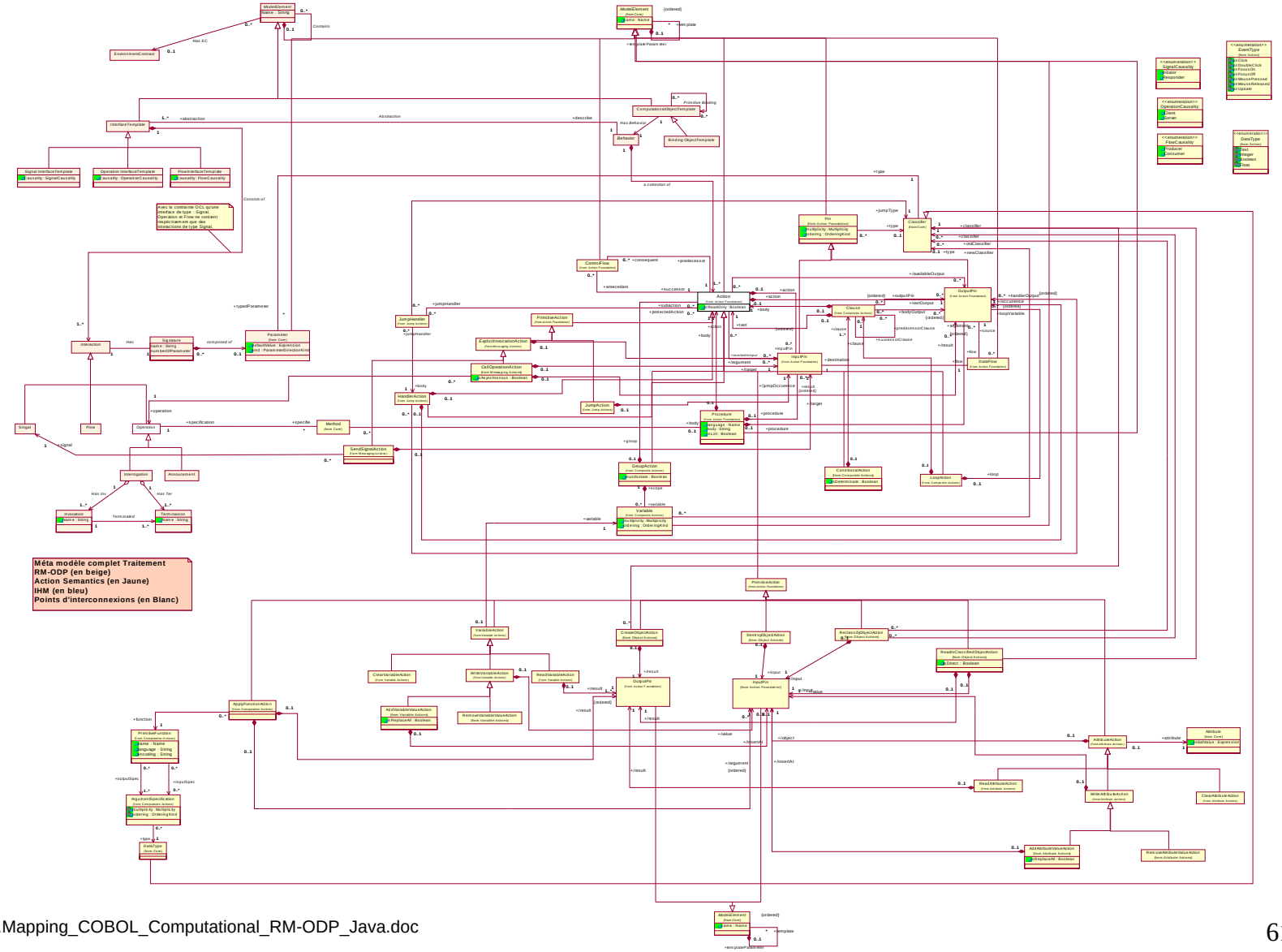
Fig.4. 1 : Méta-Modèle Java proposé par netBeans

Méta Modèle COBOL



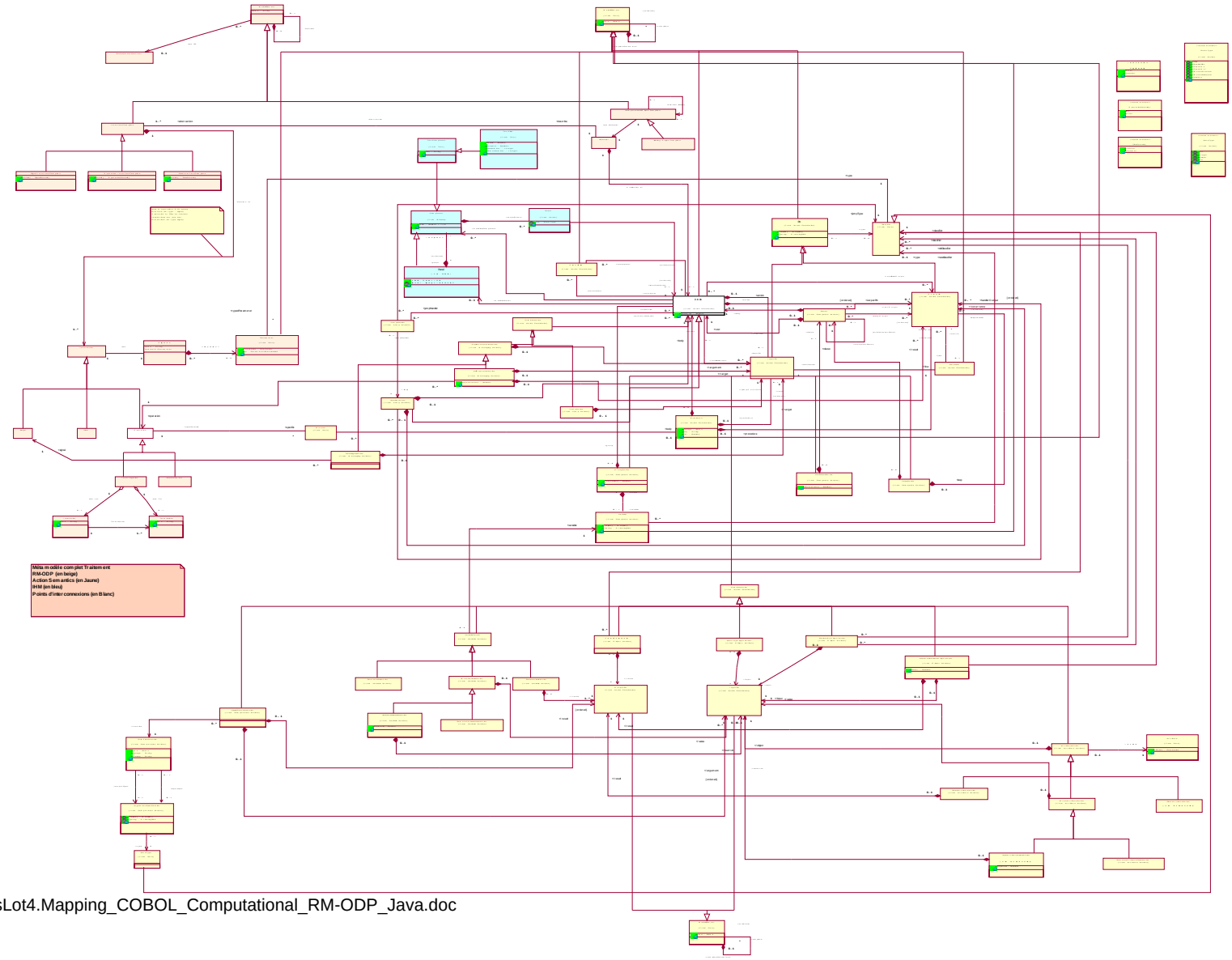
Annexe 2

Méta-Modèle résultant du couplage du point de vue Traitement de RM-ODP et de Action Semantics de UML. (Voir la version éléctronique jointe à ce document en format MDL pour plus de lisibilité.



Annexe 3

Le MM Traitement RM-ODP/ActionSemantics UML/IHM complet. (Vu l'importance du MM et pour plus de lisibilité, voir le MM en format électronique en fichier joint à ce document format .MDL)



Le JAVA/IHM complet. (Vu l'importance du MM et pour plus de lisibilité, voir le MM en format électronique en fichier joint à ce document format .MDL)