ANR programme ARPEGE 2008

Systèmes Embarqués et Grandes Infrastructures

Projet SELKIS : Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

ANR-08-SEGI-018

Février 2009 - Décembre 2011

Demonstration of the prototype (translator + V&V tools)

Livrable 3.3bis

Akram Idani
Yves Ledru
Jean-Luc Richier
Mohamed-Amine Labiadh

Aout 2012

Table des matières

1 Introduction **2**

2 Outil de multi-transformation **5**

2.1 Introduction 5

2.2 Usage de l'IDM pour l'intégration de méthodes 6

2.2.1 Présentation générale 6

2.2.2 Le module de transformation 6

2.3 Mise en œuvre de la traduction d'UML vers B 10

2.3.1 Mise en place des environnements de modélisation 10

2.3.2 Spécification de la transformation de modèles 10

2.3.3 Implémentation des règles abstraites 12

2.4 Configuration et application de la transformation 15

2.4.1 Configuration 1 15

2.4.2 Configuration 2 16

2.4.3 Configuration 3 17

2.5 Conclusion et travaux futurs 17

3 B4MSecure : une plateforme intégrée
Mode d'emploi et visite guidée **19**

3.1 Téléchargement et lancement 20

3.2 Création d'un modèle 22

3.2.1 Projet Topcased 22

3.2.2 Edition et structuration du modèle 23

3.3 Génération des spécifications B 28

3.3.1 Configuration 28

3.3.2 Traduction du modèle fonctionnel 30

3.3.3 Extraction de l'instance du méta-modèle de sécurité 30

3.3.4 Traduction du modèle de sécurité 32

Chapitre 1

Introduction

Ce livrable est dédié à la mise en pratique des concepts théoriques que l'équipe VASCO a développés dans le cadre du projet Selkis. Il s'attache au fonctionnement des outils de génération de spécifications B à partir de modèles fonctionnels renforcés par des politiques de contrôles d'accès. Rappelons que le processus de traduction que nous avons proposé passe par deux étapes majeures : (*i*) traduction du modèle fonctionnel, et (*ii*) traduction du modèle de sécurité. Dans le livrable 6.1.2 nous avons présenté les principes fondamentaux de notre approche.

Dans le cadre de la traduction du modèle fonctionnel, nous avons été motivés par la multitude de travaux de traduction d'UML en B. En vue d'une approche générique qui s'adapte à ces travaux nous avons proposé de les intégrer dans une plateforme dirigée par les modèles. Ladite plateforme de multi-modélisation permet de combiner des règles provenant d'approches différentes, de personnaliser leurs règles, et d'implémenter de nouvelles transformations. Cette première étape du processus de transformation est représentée par la figure 1.1.

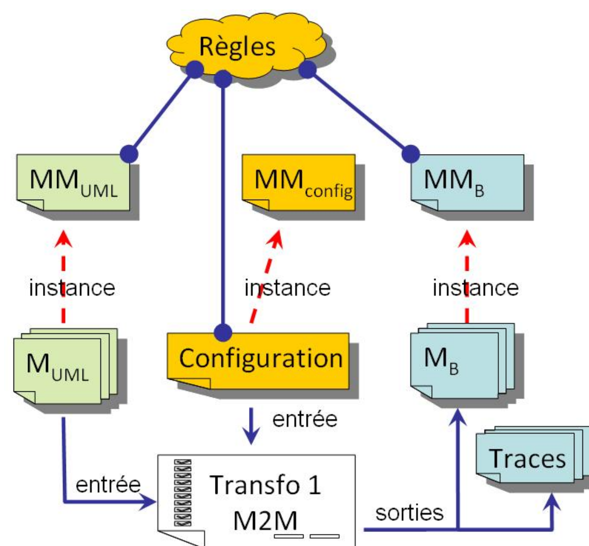


Fig. 1.1 – Première étape de la transformations

Les règles de transformation du modèle fonctionnel (représentées par un nuage de règles dans la figure 1.1) ont été implémentées dans l'outil sous forme de fonctions Java indiquant pour chaque élément du méta-modèle d'UML (MM_{UML}) ses contre-parties dans le méta-modèle de B (MM_{B}). Une configuration de transformation, spécifiée par un méta-modèle (MM_{config}), est un sous-ensemble de règles ordonnées issues du nuage de règles. La première étape du processus de transformation est donc une transformation de type Modèle-Vers-Modèle (M2M) qui prend

en entrée (i) un modèle UML (M_{UML}) instance du méta-modèle d'UML (MM_{UML}) et (ii) une configuration de transformation instance du méta-modèle MM_{config}, et produit un modèle B (M_{B}) instance du méta-modèle de B (MM_{B}) ainsi que les traces de la transformation.

Quant à la traduction du modèle de sécurité, elle est fortement liée à l'approche adoptée pour la transformation du modèle fonctionnel. En effet, le lien entre les deux modèles est très étroit. C'est pourquoi nous avons implémenté, dans notre outil de multi-transformation, une suite de règles permettant de traduire de façon spécifique la partie fonctionnelle du modèle. Rappelons que ces règles sont décrites dans le livrable 6.1.2. Sur cette base nous avons été confrontés à deux principales problématiques concernant le modèle de sécurité :

1. La syntaxe concrète du modèle : la syntaxe concrète correspond à celle de SecureUML, c'est-à-dire basée sur des classes UML stéréotypées. D'un point de vue technique le modèle graphique est réalisé dans l'outil Topcased et correspond donc à une instance du méta-modèle d'UML enrichi par un profil. Toutefois, notre modèle de sécurité se doit d'être conforme au méta-modèle de sécurité que nous avons proposé. Nous avons donc été amenés à implémenter des règles de transformation, qui génèrent une instance de notre méta-modèle de sécurité à partir d'une instance (stéréotypée) du méta-modèle d'UML. Ces règles sont implémentées en QVTo et permettent d'effectuer certaines vérifications structurelles sur le modèle graphique. Cette deuxième étape est représentée par la figure 1.2. L'environnement de modélisation utilisé pour élaborer le modèle fonctionnel et le modèle de sécurité est TopCased. Celui-ci permet de modéliser graphiquement le modèle de départ ($M_{\text{UML étendu}}$) grâce à la notion de profil ($\text{Profil}_{\text{UML}}$). Ce faisant, nous avons mis en place une transformation de type Modèle-Vers-Modèles (M2M) en vue de produire, à partir du modèle graphique de départ ($M_{\text{UML étendu}}$), le modèle de sécurité ($M_{\text{Sécurité}}$) instance du méta-modèle de sécurité ($MM_{\text{Sécurité}}$).

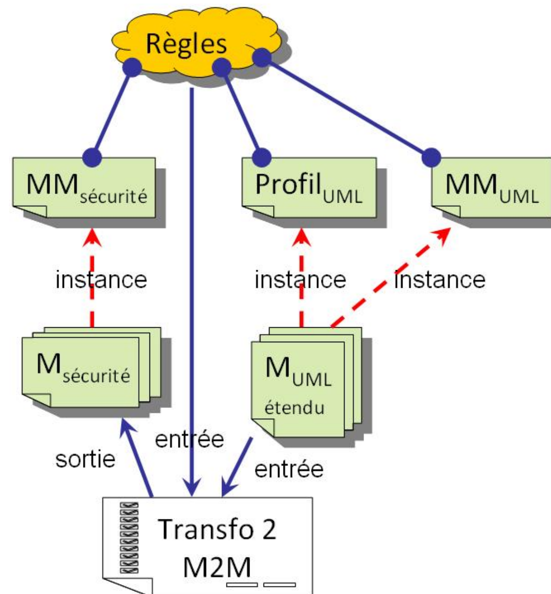


Fig. 1.2 – Deuxième étape de la transformation

2. Les types de transformation à mettre en place pour la génération des spécifications B : une fois l'instance du modèle de sécurité obtenue, l'enjeu majeur est de générer les spécifications B à partir de cette instance tout en analysant la trace de la transformation du modèle fonctionnel. En effet, l'appel aux opérations fonctionnelles est encapsulé dans les opérations sécurisées. Il faut donc retrouver les éléments de modélisation issus de la première étape de la traduction. La solution adoptée se base sur des templates xPand dont les éléments

d'entrée sont gérés par le moteur de workflow de la plateforme oAW. Cette transformation de type Modèle-Vers-Texte (M2T) est illustrée par la figure 1.3. Elle prend en entrée un modèle B (M_B), les traces générées lors de la première étape ainsi que le modèle de sécurité ($M_{Sécurité}$) instance du méta-modèle de sécurité ($MM_{Sécurité}$) et génère le code B correspondant.

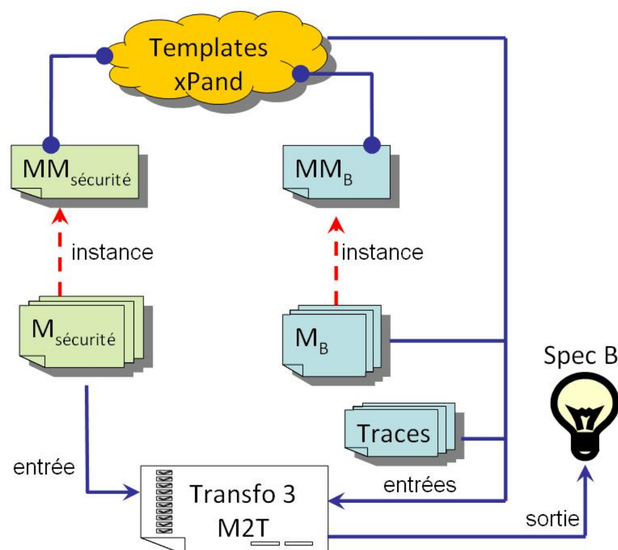


Fig. 1.3 – Troisième étape de la transformations

Le chapitre 2 rappelle les principes de notre plateforme de multi-transformation et illustre son fonctionnement sur la base d'un exemple simple. Le chapitre 3 constitue un manuel utilisateur couvrant toutes les étapes de la transformation et passe en revue les technologies utilisées lors de son implémentation.

Chapitre 2

Outil de multi-transformation

2.1 Introduction

L'intégration d'approches formelles et semi-formelles permet une utilisation plus abordable des notations formelles et plus précise des notations semi-formelles. Bien que ce sujet soit largement traité dans la littérature, son application dans la pratique reste limitée. Comme l'explique [KBC05], l'un des obstacles est le flou sémantique entourant ces approches et qui est dû au fait que les transformations entre notations sont souvent définies implicitement. Le manque d'outils pour supporter ces techniques met, également, un frein considérable à leur adoption.

La solution émergente à cette problématique, comme proposé par [KBC05, GMK+06, ZJBZ08, NOW09], prêche l'utilisation de l'Ingénierie Dirigée par les Modèles (ou IDM) pour définir et mettre en œuvre les transitions entre langages. En effet, ce paradigme requiert une définition explicite des transformations sur la base de méta-modèles définissant les modèles sources et cibles et d'un langage de transformation de modèles. L'avantage majeur en est de disposer d'un cadre conceptuel clair, de par l'utilisation des méta-modèles, pour fonder les transformations d'un langage en un autre. De ce fait, les règles de transformation qui en découlent sont souvent qualifiées de précises. Aussi, offrir une plateforme IDM pour supporter l'activité d'intégration d'UML et de méthodes formelles, est-il un moyen pragmatique pour contribuer à populariser le couplage de méthodes. Toutefois, l'implication de la communauté (aussi bien scientifique qu'industrielle) reste le moteur principal pour diffuser ces approches et relever les défis restants avant le passage à l'échelle. Pour ce faire, disposer d'une plateforme de transformation commune qui permet l'implémentation et l'expérimentation d'approches diverses, devient de plus en plus incontournable.

Dans le cadre de nos travaux de recherche nous abordons cette question en mettant en oeuvre une plateforme IDM qui se doit d'être évolutive pour supporter une multitude d'approches de traduction d'UML en des notations formelles. Cette plateforme se veut ouverte en permettant l'intégration de plusieurs langages de modélisation et de plusieurs transformations entre ces langages. Notre moteur de transformation permet de traiter une spécification abstraite d'un processus de transformation en faisant le lien avec les règles concrètes qui en découlent. L'originalité de ce travail se traduit par la possibilité de faire cohabiter plusieurs approches de transformation d'UML au sein d'un même environnement avec des règles de transformation pouvant être exprimées dans des langages variés. Cela permet une grande souplesse dans l'expression des transformations d'une part, et l'évolution et la personnalisation des approches existantes d'autre part. Les concepts théoriques sous-jacents sont étayés dans [ILL10] et ne seront pas développés davantage ici. Notons que l'intention de ce chapitre est de présenter l'usage de notre plateforme tout en mettant l'accent sur ses contributions pratiques.

Dans la section 2 nous présentons notre usage de l'Ingénierie Dirigée par Modèles pour l'intégration

de méthodes. La section 3 présente des implémentations de transformations dans le cadre du couplage d'UML et B. Dans la section 4, nous illustrons l'utilisation de notre outil sur un exemple pédagogique. Nous terminons finalement, dans la section 5, par une conclusion et des perspectives.

2.2 Usage de l'IDM pour l'intégration de méthodes

2.2.1 Présentation générale

Une architecture IDM est souvent une architecture à trois composantes qui sont outillées par trois modules intégrables à la plateforme EMF[1] : deux modules pour l'édition des modèles sources et cibles et un module d'expression et d'exécution de règles de transformation. L'idée principale de ce type d'architecture est de définir dans un langage de transformation la transition entre un modèle d'entrée et un modèle de sortie. Dans le cadre de nos travaux une telle architecture nous permettra de disposer d'un socle unique proposant la modélisation UML, la modélisation formelle et la transformation de modèles. Notre plateforme IDM est donc constituée de :

- Un module de modélisation UML avec le méta-modèle correspondant et un éditeur de diagrammes pour pouvoir manipuler graphiquement les modèles. Plusieurs éditeurs UML existent en tant que plugins Eclipse (TopCasedUML, Papyrus, etc). Dans notre plateforme nous avons intégré la spécification de UML2.1 disponible dans la distribution "Modeling" d'Eclipse [EMP10]. Cette dernière fournit entre autres un éditeur de diagrammes UML utile pour visualiser et éditer les modèles UML.
- Un module de modélisation formelle avec le méta-modèle correspondant et un éditeur de syntaxe concrète. Pour le cas des transformations d'UML en B nous avons considéré le méta-modèle de B proposé par [Ida06][2]. Pour passer de cette syntaxe abstraite de B vers la syntaxe concrète, nous avons intégré une amélioration de l'outil de génération de code proposé dans [IBP09].
- Un module de transformation de modèles dans lequel nous pouvons définir et exécuter les transformations entre UML et les notations formelles. Ce module constitue le cœur de notre travail et sera présenté en détail dans la suite du chapitre. Notons qu'il existe une diversité d'outils/langages de transformation de modèles compatibles avec la plateforme EMF d'Eclipse. Toutefois ils n'offrent pas les mêmes fonctionnalités et sont basés sur des paradigmes différents. Le choix du langage de transformation dépend des besoins de la transformation et de l'expérience de l'utilisateur.

2.2.2 Le module de transformation

Aperçu et contributions

Classiquement, dans les environnements IDM, le module de transformation est composé d'un moteur de transformation qui interagit avec les deux autres modules en vue de réaliser la transformation d'un modèle source vers un modèle cible. Cependant, cela suppose qu'une transformation est encodée de façon statique et il devient par conséquent difficile de changer les règles utilisées dans une transformation. Ce besoin se ressent fortement quand l'analyste est face à plusieurs solutions alternatives lors de la transformation d'un même élément source. Par exemple, en parcourant les divers travaux qui ont cherché à définir des règles de transformation d'un modèle objet en un modèle relationnel, nous avons constaté que certains désirent obtenir autant de tables relationnelles que de classes dans le modèle objet, tandis que d'autres ne produisent

[1] Eclipse Modeling Framework
[2] Ce méta-modèle (ecore) est disponible sous http ://membres-liglab.imag.fr/idani/BMeta/BMethod.ecore

des tables que pour les classes concrètes du modèle. Pour ce faire, dans les environnements IDM existants, il faut agir directement sur le code de la transformation et proposer autant de blocs de règles que d'alternatives. Ainsi, l'intérêt de notre module de transformation en comparaison avec les modules existants est qu'il permet de décomposer une transformation en règles élémentaires (dites abstraites) qui pourront être instanciées par diverses règles concrètes . L'utilisateur pourra par la suite sélectionner la séquence de règles souhaitée. Dans notre approche, une transformation de modèles devient configurable et le choix d'une solution parmi d'autres n'est autre qu'une configuration particulière de la transformation.

Rappelons que notre objectif principal est de pouvoir, d'une part, combiner aisément différentes règles issues d'approches diverses et d'autre part, d'étendre ces approches par de nouvelles règles. Cela nécessite la composition de règles pour pouvoir définir une transformation configurable selon un choix de règles déterminé par l'utilisateur. L'originalité de notre approche est, de ce fait, l'introduction d'une couche supplémentaire permettant de spécifier, au moyen d'un méta-modèle, la notion de transformation configurable.

Outre le fait que notre plateforme permet à l'utilisateur de sélectionner des règles diverses provenant d'approches variées, les règles elles-mêmes peuvent être codées dans divers langages de transformation. Cela procure non seulement une souplesse dans le choix de la transformation mais également une souplesse dans l'implémentation des transformations. Nous verrons par la suite un exemple de règle écrite en Java et en xTend/oAW [3]. Notre moteur de transformation interprète une configuration de transformation et délègue l'exécution des règles concrètes à d'autres outils de transformation. Pour ce faire, la transformation est guidée par un processus de transformation abstrait. L'outil de transformation que nous proposons ne se présente donc pas sous la forme d'un langage de transformation assisté par un moteur de transformation. En effet, notre intention n'est pas de proposer un nouveau language, mais plutôt de développer un invocateur de moteurs de règles (*e.g.* ATL [JABK08] ou QVT [OMG08], ...) existants en vue d'exécuter les règles référencées dans notre module de transformation lors d'une configuration de transformation. Notons que notre proposition n'est pas spécifique au couplage de UML et B ; elle a été définie dans un cadre général et sera expérimentée également sur des transformations alternatives de UML vers Java.

Le processus de transformation

La notion de processus abstrait permet de subdiviser une transformation de modèle en un ensemble de phases séquentielles dont le comportement est variable selon l'implémentation. Chacune de ces phases, que nous appelons règle abstraite, transforme un type d'élément donné. Le vrai comportement de ces règles, notamment les éléments produits, dépend des implémentations qui les réalisent. En effet, une même règle abstraite peut être implémentée de plusieurs manières différentes et dans des langages de transformation différents. Ces implémentations sont appelées règles concrètes. Ainsi, chaque règle abstraite est réalisée par une ou plusieurs règles concrètes. Par exemple, transformer un diagramme de classe, comme l'illustre la figure 2.1, revient à transformer successivement les packages, les classes, les associations, les attributs et les opérations de ce diagramme.

Spécification/exécution des transformations

La configurabilité des transformations et le méta-modèle de transformations configurables correspondant ont été discutés dans [ILL10] ; nous n'allons donc pas les reprendre en détail dans le présent chapitre. La figure 2.2 en présente un fragment utile pour la suite. Au niveau de ce méta-modèle de configuration, une transformation de modèle est définie par trois types d'entités : (a)

[3]openArchitectureWare (http ://www.openArchitectureWare.org)

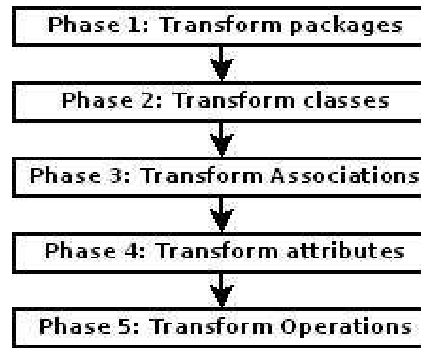


Fig. 2.1 – Un processus abstrait pour la transformation d'un diagramme de classes UML.

les règles abstraites, (b) les types d'éléments sources et (c) les règles concrètes.

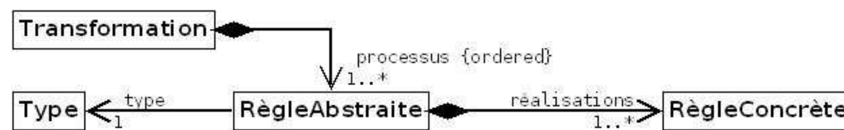


Fig. 2.2 – Fragment du méta-modèle de transformation configurable.

Une transformation est spécifiée par un ensemble ordonné de règles abstraites que nous appelons processus abstrait. Chacune de ces règles est associée au type d'éléments du modèle d'entrée qu'elle transforme. Elle est, également, associée à un ensemble (non vide) de règles concrètes qui la réalisent. Les types d'éléments sources sont les types d'éléments transformés par les règles abstraites et définis dans le méta-modèle d'entrée (e.g. Class, Association, etc.). La figure 2.3 présente globalement l'architecture de notre module de transformation. On y voit le moteur de transformation (TransformationEngine) qui fait appel à trois autres composants (interfaces). En effet, le moteur de transformation interprète un modèle de transformation configurable (CTM) pour transformer un modèle d'entrée (InputModel) en un modèle de sortie (OutputModel). De plus, il stocke les liens entre le modèle d'entrée et le modèle de sortie dans un modèle de trace (TraceModel). L'ensemble des modèles manipulés par le moteur de transformation, que nous appelons contexte de la transformation, sont regroupés au sein du composant *TransformationContext*. Le moteur de transformation reconnaît les types des éléments en entrée grâce à une interface (appelée *InstanceProvider*) et peut de ce fait en extraire automatiquement les instances pour les transformer. L'implémentation par défaut de l'interface *InstanceProvider* ne reconnaît que les types prédéfinis dans le méta-modèle d'entrée. Ainsi, pour prendre en compte de nouveaux types en entrée il suffit d'étendre le méta-modèle d'entrée et de surcharger par conséquent l'*InstanceProvider*.

Contrairement aux règles abstraites, les règles concrètes sont des règles exécutables. Elles permettent de réaliser les règles abstraites auxquelles elles sont associées. Notre outil de transformation n'offre pas de langage pour l'implémentation de ces règles mais propose la réutilisation des langages et outils existants. En effet, les règles concrètes permettent de référencer, au moyen d'identifiants, des règles externes implémentées avec des outils tiers. Concrètement, le moteur de transformation récupère la référence de la règle externe à appliquer et l'exécute à l'aide d'une interface particulière appelée invocateur de règles (*RuleInvoker*). L'implémentation actuelle de *RuleInvoker* permet d'invoquer des règles écrites en tant que méthodes Java ou en tant qu'extensions (opérations) en xTend[4] à travers leurs API respectives (ConcrèteTransformationEngine). Nous envisageons, en perspective, de l'étendre à d'autre langages comme QVT. Cette dissocia-

[4]langage de transformation de modèles de l'outil oAW

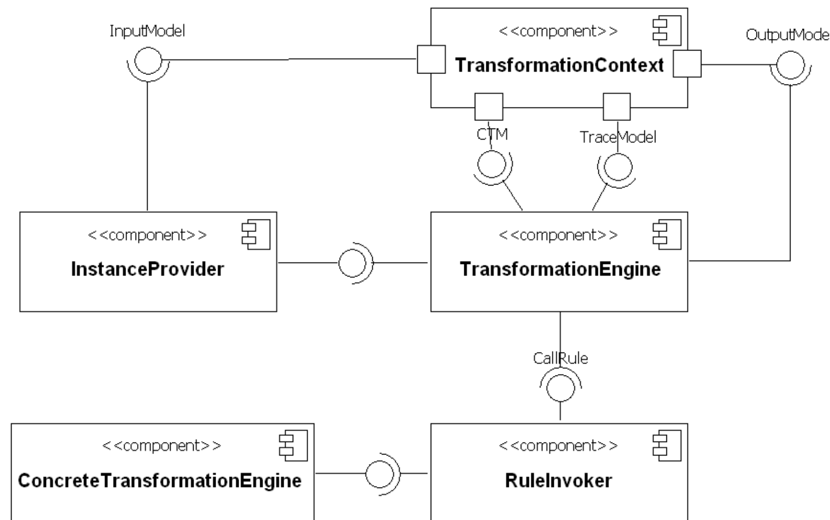


FIG. 2.3 – Architecture du module de transformation.

tion entre règles abstraites et règles concrètes (spécification/exécution) permet de faire varier dynamiquement le comportement des règles de la transformation et ainsi le comportement global de la transformation. Le moteur de transformation déterminera dynamiquement, à partir des paramètres de configuration, la règle concrète qui réalisera chacune des règles abstraites.

Configuration de la transformation

Pour un processus de transformation, la réalisation de chaque règle abstraite devient un point de décision qu'il faut résoudre en fonction de la configuration. Pour ce faire, nous proposons deux méthodes (manuelles) de configuration pour guider ces décisions : (*i*) la méthode de sélection par **règle** et (*ii*) la méthode de sélection par **approche**.

La première consiste à référencer explicitement la règle concrète à utiliser pour chaque règle abstraite. Cette méthode offre une grande flexibilité dans la configuration de la transformation. Néanmoins, l'application de cette méthode devient difficile lorsque le nombre de points de décision augmente considérablement. Pour remédier à ce problème, nous proposons une deuxième méthode plus intuitive pour l'utilisateur. Il s'agit de permettre à l'utilisateur de pré-définir des combinaisons de règles concrètes produisant des résultats différents selon les besoins. Ces combinaisons pré-définies que nous appelons **approches de transformation** correspondent généralement à des critères de classification du résultat obtenu. Par exemple, pour le cas de la traduction d'UML vers B [LM00, Led01, SB06] l'un des critères utilisable est la modularité de la spécification B obtenue. Nous pouvons, par exemple, proposer trois approches de transformation d'un diagramme de classes basées sur ce critère :

(*i*) génération d'une seule machine B pour l'ensemble du diagramme de classes,

(*ii*) génération d'une machine par classe UML ou

(*iii*) génération d'une machine par paquetage.

Les deux méthodes de configuration sont, en fait, complémentaires. En effet, la méthode de sélection par règle permet de personnaliser les approches existantes tandis que la méthode de sélection par approche permet de pérenniser ces approches et d'en proposer de nouvelles dont l'intention est d'être réutilisées ultérieurement sur d'autres modèles.

Nous distinguons en plus des deux méthodes de configuration deux niveaux pour leur application. Le premier niveau — que nous appelons niveau **méta-modèle** ou $\mathcal{M}^2$ par référence aux quatre

niveaux de méta-modélisation de l'[OMG09, section 7.10] — correspond à l'application de la configuration uniformément sur l'ensemble du modèle source. C'est à dire que le choix de la règle concrète qui réalisera chacune des règles abstraites est récupéré exclusivement à partir des paramètres de configuration. Concrètement, tous les éléments du modèle d'entrée ayant le même type seront traduits par la même règle concrète désignée dans la configuration et produiront donc le même type de résultat. Ce niveau d'application est connu sous le nom de Model Type Mapping dans la spécification MDA de l'[OMG03, section 3.4].

Le deuxième niveau, que nous appelons niveau **modèle** ou $\mathcal{M}^1$, permet de personnaliser la transformation du niveau $\mathcal{M}^2$ en forçant certains éléments du modèle en entrée à être transformés de façon différente de la configuration. Nous proposons pour cela le marquage de ces éléments au moyen de stéréotypes indiquant la règle à leur appliquer.

2.3 Mise en œuvre de la traduction d'UML vers B

On se propose de mettre en œuvre la traduction d'UML vers B en utilisant notre outil de transformation afin d'en illustrer l'utilisation par un exemple concret. Nous considérons une traduction simple permettant de produire un squelette à partir des classes d'un modèle UML et pour laquelle il y a un consensus entre les diverses approches d'intégration d'UML et B. Dans la spécification B qui résulte de la transformation d'une classe, nous obtenons, comme illustré ci-dessous, deux ensembles : l'ensemble des instances effectives $\mathcal{C}_E$ et l'ensemble d'instances possibles $\mathcal{C}_P$ avec un invariant d'inclusion indiquant que $\mathcal{C}_E$ est inclus dans $\mathcal{C}_P$.

MACHINE	C
SETS	$\mathcal{C}_P$
VARIABLES	$\mathcal{C}_E$
INVARIANT	$\mathcal{C}_E \subseteq \mathcal{C}_P$

2.3.1 Mise en place des environnements de modélisation

Pour pouvoir définir la transformation de modèles UML vers B nous devons disposer des méta-modèles EMF correspondants. Nous présentons dans la figure 2.4 une version très simplifiée du méta-modèle d'UML contenant les concepts que nous allons prendre en compte dans notre exemple de transformation. Pour le méta-modèle de la notation B nous utilisons celui proposé par [Ida06]. Ce méta-modèle modélise les données des clauses usuelles en B et est pleinement suffisant pour l'exemple de traduction d'UML vers B que nous envisageons.

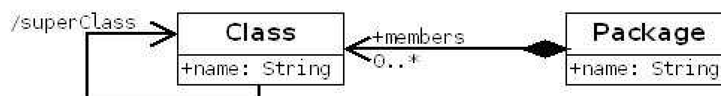


FIG. 2.4 – Concepts UML à transformer

2.3.2 Spécification de la transformation de modèles

Une fois nos méta-modèles mis en place, il devient alors possible de définir une configuration de transformation de modèles. Pour ce faire, nous allons procéder en deux étapes. Lors de la première étape nous définissons les types d'éléments UML qui sont pris en compte dans la transformation ainsi que les approches transformationnelles que nous allons implémenter. Dans

la deuxième étape, nous définissons un processus abstrait pour transformer ces éléments et nous implémentons ses règles concrètes.

Types d'éléments à transformer En vue de pouvoir spécifier la transformation dans notre environnement, il est primordial de déterminer le sous ensemble des éléments de modélisation UML qui seront visités lors de la transformation. Dans notre exemple de transformation, ces éléments sont : le *Package* racine représentant le modèle en question ainsi que les *classes* contenues dans ce paquetage. Concrètement, nous définissons au niveau de la spécification, un type d'élément source, que nous appelons *This*, pour faire référence au package racine du modèle. Toutefois, ce type ne sera pas reconnu par le moteur de transformation puisqu'il n'existe pas dans UML. Nous devons donc le considérer dans l'implémentation du fournisseur d'instances (*InstanceProvider*). Nous définissons ensuite le deuxième type d'éléments, nommé *Class*, qui correspond aux classes UML. Celles-ci seront donc systématiquement reconnues par le moteur de transformation.

Approches considérées Afin de mettre en évidence la configurabilité de la transformation nous considérons deux approches de transformation de classes UML. La première, inspirée des travaux de [SB06], produit une machine B unique pour tout le diagramme de classes en entrée. Cette approche, que nous appellerons *UniqueMachine* , vise à limiter la complexité liée aux mécanismes de modularité en B et à simplifier, ainsi, l'activité de preuve sur la spécification B. Cependant, les avantages de la structuration B (*e.g.* inclusions de machines) tels que la modularité de la spécification et sa lisibilité sont perdus. La deuxième approche, inspirée des travaux de [Mey01], propose de produire autant de machines que de classes afin de préserver la modularité de la spécification UML, et ce, en incluant ces machines dans une machine racine généralement nommée *Machine Système*. Nous appellerons cette approche *MachineByClass*.

Processus abstrait Rappelons que le concept de base de notre approche de transformation configurable est la notion de processus de transformation abstrait. Cette dernière définit une transformation comme une séquence de règles abstraites dont le comportement est déterminé dynamiquement en fonction de la configuration. Pour la transformation considérée dans notre exemple, nous proposons de définir un processus en deux phases (règles abstraites) :

– la première phase que nous appelons *CreateSystemMachine* se charge d'explorer la racine du modèle UML et de produire la machine B du modèle (machine système). Une fois cet élément créé et nommé au niveau du modèle de transformation, nous lui associons un type d'élément en entrée (qui sera le type This).
– la deuxième phase que nous appelons *ClassTo* explore les classes pour produire les données B correspondantes. Les éléments d'entrée de cette phase sont donc de type *Class*.

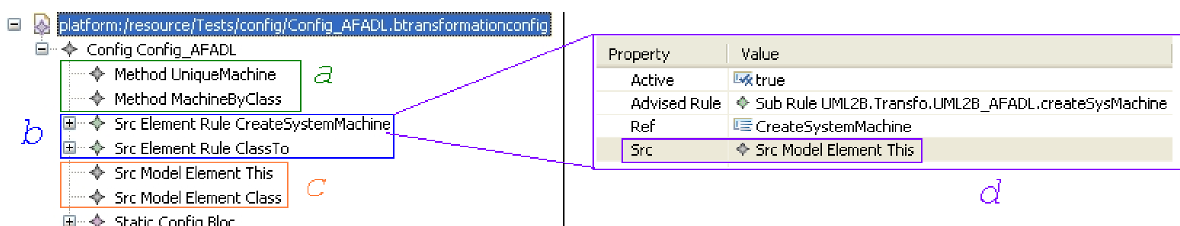


FIG. 2.5 – Modèle de configuration

A ce stade, le modèle de transformation se présente comme illustré dans la figure 2.5 avec (a) les approches, (b) le processus abstrait et (c) les types. Dans la partie droite (d) de la figure 2.5 nous

présentons les propriétés de la règle abstraite *CreateSystemMachine* qui indiquent l'élément de modélisation *This* comme type d'entrée. Les autres propriétés sont spécifiques à l'instanciation du méta-modèle de configuration et ne font pas l'objet du présent chapitre.

2.3.3 Implémentation des règles abstraites

Dans cette sous-section nous présentons une manière d'implémenter le processus abstrait défini précédemment. Nous proposons, pour chaque règle abstraite du processus de transformation, autant d'implémentations que de comportements différents espérés ou possibles de cette règle. Concrètement, si nous considérons seulement les deux approches proposées alors la transformation devra avoir deux comportements distincts. Dans notre cas, cela se traduira par deux règles concrètes différentes pour la règles abstraite *ClassTo*.

Règle 1 : implémentation de la règle abstraite *CreateSystemMachine*

La première phase, consistant à créer une machine B pour le diagramme de classes, est commune aux deux approches. En effet, cette machine servira à regrouper l'ensemble des données du modèle dans le cas de l'approche *UniqueMachine* et à inclure les autres machines dans le cas de l'approche *MachineByClass*. Pour cela, la première règle abstraite ne sera implémentée que par une seule règle concrète.

Rappelons qu'il est possible d'implémenter les règles de transformation dans divers langages. A ce stade, nous faisons le choix d'implémenter cette règle concrète en utilisant directement le langage Java sans passer par un langage de transformation particulier. Pour ce faire, nous devrons respecter la signature de méthodes supportée par le moteur de transformation et plus précisément l'invocateur de règles externes qui peut éventuellement être personnalisé.

La signature par défaut exige en entrée une méthode de classe statique et publique qui prend en entrée un objet de type *ConfigContext* ainsi que l'élément à transformer avec le type *EObject* et retourne le(s) élément(s) crée(s) dans un objet de type *EObject*. La classe *ConfigContext*, qui fait partie de l'API de notre outil de transformation, représente le contexte de la transformation configurable. Elle contient le modèle d'entée (UML), le modèle de sortie (B), le modèle de transformation configurable et le modèle de trace qui mémorise les relations entre les éléments traités par la transformation et les éléments produits. La Classe *EObject*, à l'instar de *Object* en Java, est la racine de tout objet modélisé en EMF et peut donc être n'importe quel élément du modèle d'entrée ou du modèle de sortie.

```
public class UML2B_AFADL {
   static public EObject  createSysMachine(ConfigContext ctx,EObject o){
       if (o instanceof Package ){//the input element must be a package
               Package p = (Package)o;//create a new B Machine using the factory
               Bmachine m=BFactory.eINSTANCE.createBMachine();
               m.setName(p.getName());//assign the name of the model to the machine
               m.setSpec(ctx.getBSpec());       //add the machine to the collection
                                                //of the B specification's machines
               ctx.getBSpec().setSystemMachine(m);// set it as the system machine
               return m;
               } else{return null;}
       }
}
```

FIG. 2.6 – Corps de la règle *createSystemMachine*

La méthode *createSysMachine* de la classe *UML2B_AFADL*, illustrée par la figure 2.6, est la règle concrète qui implémente la première règle abstraite conformément à la signature exigée.

Cette méthode prend en entrée le contexte de la transformation et la racine du modèle UML, qui doit être de type *Package*, et retourne en sortie la machine B correspondante en lui affectant le nom du modèle et en la référençant en tant que machine système.

Comme illustré dans la figure 2.7, nous créons, sous la première règle abstraite du modèle de transformation, une entité de type *SubRule* correspondant à la règle concrète. Puis nous référençons la méthode *createSysMachine*, grâce l'attribut *Ref*, par son chemin absolu. L'attribut *Comply With* permet d'indiquer les approches qui utilisent cette règle. Il s'agit ici des deux approches citées précédemment et pour lesquelles cette règle est bien commune.

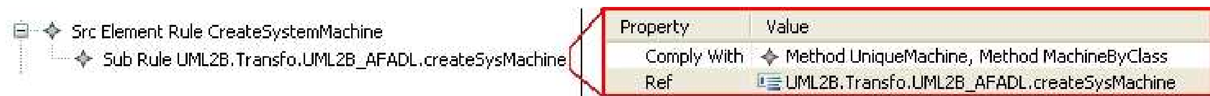


FIG. 2.7 – Spécification de la règle concrète *createSystemMachine*

Règle 2 : implémentation de la règle abstraite *ClassTo*

La deuxième phase du processus, correspondant à la deuxième règle abstraite, se charge de produire les structures de données B à partir des classes UML et de les structurer selon nos deux approches. Dans l'approche *UniqueMachine* ces structures de données sont encapsulées dans la machine système créée lors de la première phase alors que dans l'approche *ClassByMachine* les structures de données de chaque classe sont encapsulées dans une machine à part. Les structures de données B modélisant une classe UML sont composées d'une variable représentant les instances existantes de cette classe, d'un ensemble abstrait représentant les instances possibles de cette classe si celle-ci n'hérite d'aucune autre classe et d'un invariant de typage de la variable. Nous avons ainsi deux règles concrètes qui implémentent cette même règle abstraite.

Pour illustrer la capacité de l'outil à supporter plusieurs langages de transformation, nous proposons de les implémenter à l'aide du langage de transformation xTend d'oAW qui permet une écriture plus synthétique inspirée des expressions OCL. Le but n'étant pas de présenter ce langage d'expression de transformations mais de mettre en évidence l'utilisation de plusieurs langages dans une même transformation. De même que pour l'écriture de règles en Java, nous devons respecter la signature exigée par l'invocateur de règles xTend.

Implémentation de la règle 2 selon l'approche *UniqueMachine*

La figure 2.8 correspond à l'implémentation, sous la forme d'une extension xTend, de la règle concrète selon l'approche *UniqueMachine*. Cette extension, dont la syntaxe est similaire à celle d'une opération, prend en entrée la classe à transformer et retourne la variable B créée. L'expression *create*, devançant le type de retour de l'extension (*BVariable*), permet de créer une variable B et de la définir comme le contexte de l'extension. De même que pour une classe Java, la variable B créée est référençable, dans le corps de l'extension, par le mot clé *this*. En xTend, les expressions chaînées par l'opérateur `->` sont évaluées séquentiellement et le résultat de la chaine est celui de la dernière expression. C'est pour cela que l'élément this, représentant le résultat de l'extension, est la dernière expression dans l'extension. La première expression (ligne 1) consiste à affecter, moyennant l'opération *setName* du type *BVariable*, le nom de la classe à celui de la variable B créée. La deuxième expression (ligne 2) permet d'ajouter la variable B aux données de la machine système. L'élément *BSPEC* est une variable globale qui représente le modèle B. En effet, pour les règles écrites en xTend, nous avons fait le choix de passer le contexte de la transformation comme variable globale et non pas comme paramètre de règles.

```
  create BVariable _ClassToBVariableInSysMachine(uml::Class c)://this is a comment
1 setName(c.getCannonicalName())-> //assign the name of the class to the variable
2 BSPEC().systemMachine.data.add(this)->      //add the B variable to the
                                              //machine's data
3 c.superClass.isEmpty?(//if the class doesn't inherit from any other class then
       let as=  new BAbstractSet:// create a new abstract set "as"
       as.setName(name.toUpperCase())-> //assign the variable name to the abstract
                                        //set name
       BSPEC().systemMachine.data.add(as)->//add the as to the system machine
       this.setType(as,c) // add a typing invariant : this subset_of as
):(//else do ...)->
4 this;
```

FIG. 2.8 – Implémentation de la règle *ClassTo* selon l'approche *UniqueMachine*

Dans la troisième expression (ligne 3), la structure conditionnelle Condition ? AlorsExpr : sinonExpr permet de différencier le traitement selon la présence ou non d'héritage. En effet, si la classe n'hérite d'aucune autre classe alors un ensemble abstrait as est créé en lui affectant le nom de la classe (en majuscules) puis en créant un invariant de typage exprimant la relation d'inclusion entre la variable B, représentant les instances existantes de la classe, et l'ensemble abstrait (as) représentant les instances possibles ($\mathcal{C}_E \subseteq \mathcal{C}_P$). Dans notre exemple nous réalisons ce typage par la fonction setType(_,_). Dans le cas où la classe hérite d'une autre classe alors le typage se fera avec la variable B représentant les instances existantes de la classe parente ($fils_E \subseteq Parent_E$). Le code correspondant à ce cas de figure n'apparait pas dans la figure 2.8.

Une fois la règle concrète implémentée, nous pouvons la référencer dans la configuration de la transformation avec son nom (*_ClassToBVariableInSysMachine*) tout en spécifiant l'approche conforme à cette règle à savoir *UniqueMachine* .

Implémentation de la règle 2 selon l'approche *MachineByClass*

Cette règle concrète, appelée $_ClassToMachine_Inh$ et présentée dans la figure 2.9, prend en entrée une classe UML et retourne une machine B. Vu que dans cette règle l'élément créé est une machine B, alors la variable B représentant les instances existantes de la classe est créée localement à l'aide de l'expression *Let*. Après avoir été nommée avec le nom de la classe, la machine créée est ajoutée à l'ensemble des machines du modèle B. La suite du traitement est similaire à la règle précédente, sauf pour l'avant dernière expression qui permet de créer un lien d'inclusion entre la machine créée et la machine système.

```
create  b::BMachine _ClassToMachine_Inh(uml::Class c):
let v = new BVariable: //create a new variable v
this.setName(c.getCannonicalName())->//set machine name from the class name
BSPEC().machines.add(this)->//add the machine to the B model
v.setName(c.name)->//assign the name of the class to the variable
this.data.add(v)->//add the variable v to the current machine
                      //if no inheritance
c.superClass.isEmpty?(/*then*/ ...):(/*else*/...)->
              //include this machine to the system machine if the latter exist
(BSPEC().systemMachine!=null)?BSPEC().systemMachine.addInclude(this) : null )->
this;
```

FIG. 2.9 – Implémentation de la règle *ClassTo* selon l'approche *MachineByClass*

Une fois la règle concrète implémentée, nous l'intégrons à la configuration de la transformation en la référençant avec son nom ($_ClassToMachine_Inh$) et en spécifiant l'approche conforme à

cette règle à savoir *MachineByClass*.

Création du lien entre règles abstraites et concrètes

La figure2.10 présente les deux règles concrètes, qui implémentent la règle abstraite *ClassTo*, ainsi que leurs propriétés au niveau du modèle de transformation.

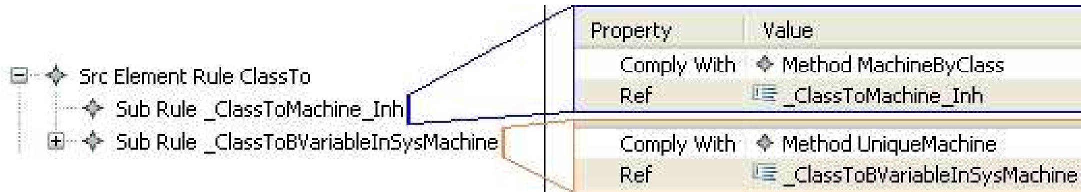


FIG. 2.10 – Spécification des implémentations de la règle abstraite *ClassTo*

Cette dernière étape de l'implémentation marque la fin de la phase de spécification de la transformation configurable. A ce stade, la transformation est complète et peut donc être appliquée. Nous proposons, dans ce qui suit, d'expérimenter cette transformation sur un cas d'étude pédagogique afin d'illustrer les capacités de l'outil.

2.4 Configuration et application de la transformation

Pour illustrer l'utilisation pratique de notre outil nous allons prendre en entrée un diagramme de classes UML représentant une vidéothèque (figure2.11) tiré de [Lal02]. Ce diagramme comporte sept classes, trois associations et deux héritages simples.

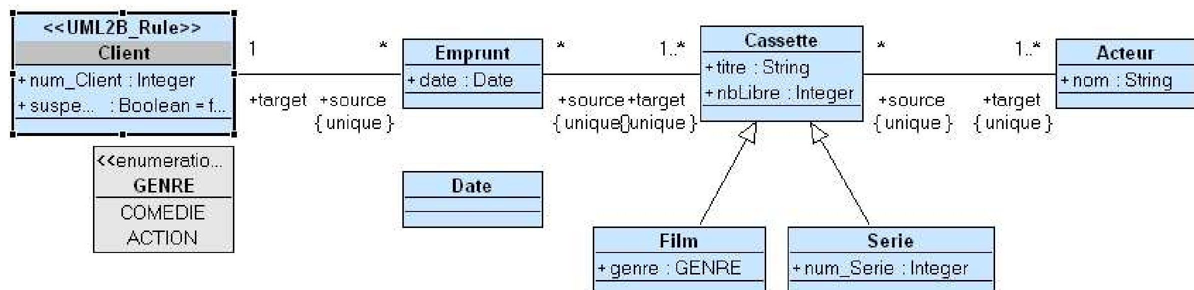


FIG. 2.11 – Diagramme de classe d'une vidéothèque [Lal02]

Dans la suite, nous proposons trois configurations de transformations différentes :

- Configuration 1 : application de l'approche *UniqueMachine* sur l'ensemble du modèle.
- Configuration 2 : application de l'approche *MachineByClass* sur l'ensemble du modèle.
- Configuration 3 : application de l'approche *UniqueMachine* sur l'ensemble du modèle hormis la classe Client qui sera transformée par une machine à part.

2.4.1 Configuration 1

Pour transformer le modèle UML selon la configuration 1 nous devons paramétrer la transformation au niveau de la racine du modèle de transformation. Les paramètres à éditer sont, comme illustré dans la figure 2.12 :

– la méthode de configuration (ChoiceBase) : nous choisissons la méthode de configuration par approche (METHOD) induisant une transformation conforme à une approche de transformation.
– L'approche sélectionnée (DefaultMethod) : nous choisissons l'approche *UniqueMachine*.
– Le niveau de configuration : nous choisissons le niveau méta-modèle ($\mathcal{M}^2$) qui permet d'appliquer la transformation, de façon uniforme sur l'ensemble du modèle d'entrée.

Property	Value
Choice Based	METHOD
Default Method	Method UniqueMachine
Level	M2
Ref	Config_AFADL

Fig. 2.12 – Paramètres de configuration de la transformation

Le résultat de cette transformation, comme illustré par la figure 2.13, produit une seule machine B contenant toute l'information du diagramme de classes en entrée.

```
MACHINE
        VideoClub
SETS
        CLIENT, EMPRUNT, CASSETTE, ACTEUR, DATE
VARIABLES
         Client, Emprunt, Cassette, Acteur, Film, Serie, Date
INVARIANT
        Client ⊆ CLIENT & Emprunt ⊆ EMPRUNT & Cassette ⊆ CASSETTE & Acteur ⊆ ACTEUR
        & Film ⊆ Cassette & Serie ⊆ Cassette & Date ⊆ DATE
END
```

Fig. 2.13 – Résultat de la configuration 1

2.4.2 Configuration 2

Si nous modifions cette approche par l'approche *ClassByMachine* le résultat sera autant de machines que de classes et une machine (VideoClub) incluant ces dernières. Il suffit de modifier le paramètre *DefaultMethod*. Ci-dessous un aperçu du résultat :

```
MACHINE
  VideoClub
INCLUDES
  Client,
  Emprunt,
  Cassette,
  Acteur,
  Film,
  Serie,
  Date
END
```

```
MACHINE
  Cassette
SETS
  CASSETTE
VARIABLES
  Cassette
INVARIANT
  Cassette ⊆ CASSETTE
END
```

```
MACHINE
  Film
USES
  Cassette
VARIABLES
  Film
INVARIANT
  Film ⊆ Cassette
END
```

```
MACHINE
  Client
SETS
  CLIENT
VARIABLES
  Client
INVARIANT
  Client ⊆ CLIENT
 END
```

```
MACHINE Emprunt ...
```

```
MACHINE Acteur ...
```

...

Fig. 2.14 – Résultat de la configuration 2

2.4.3 Configuration 3

Nous proposons maintenant d'illustrer l'application d'une configuration au niveau modèle ($\mathcal{M}^1$). A cet effet, nous appliquons l'approche *UniqueMachine* sur l'ensemble du modèle UML en entrée et nous transformons la classe Client en une machine à part. Pour ce faire, nous paramétrons la transformation comme illustré dans la figure 2.12, sauf que le niveau appliqué sera $\mathcal{M}^1$. Nous appliquons le stéréotype *UML2B_Profile*::*UML2B_Rule* sur la classe Client en spécifiant la règle à utiliser ($_ClassToMachine_Inh$) au niveau de l'attribut *Rule* du stéréotype. Nous obtenons, comme attendu, une spécification B correspondant à une approche *UniqueMachine* mais avec une machine Client distincte comme illustré par la figure 2.15.

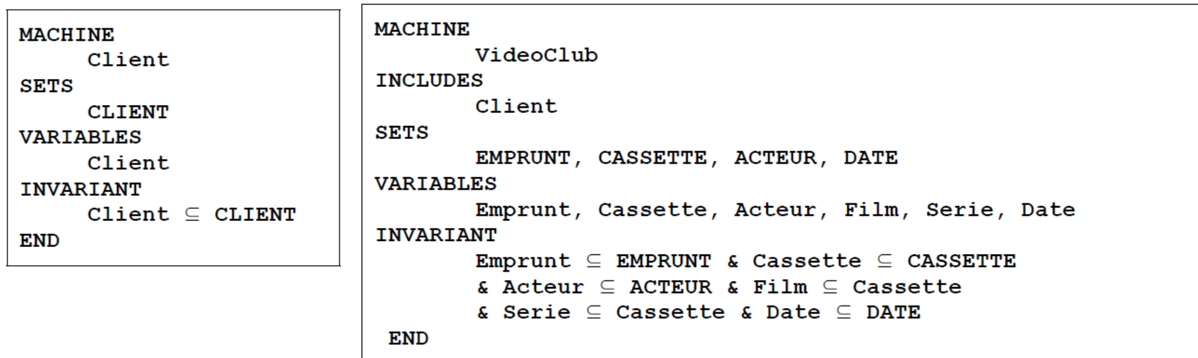


FIG. 2.15 – Résultat de la configuration 3

2.5 Conclusion et travaux futurs

Une multitude de travaux de recherche se sont intéressés à l'intégration de UML et de notations formelles et ont abouti à diverses règles de transformation, souvent complémentaires. Cependant, et bien que ces recherches disposent de plusieurs bénéfices provenant de l'apport des méthodes formelles et semi-formelles, elles restent dédiées à des cadres applicatifs bien spécifiques. Par conséquent, elles ne sont pas applicables à plus grande échelle. Dans nos travaux actuels, nous proposons une plateforme IDM, d'intégration d'UML et de langages formels, qui se veut générique de par le fait qu'elle peut s'adapter à des approches transformationnelles diverses. L'avantage en est de pouvoir faire cohabiter, dans un même environnement des règles de transformation provenant d'approches complémentaires, ce qui permettra de les étendre et de les personnaliser. Ce chapitre a présenté notre plateforme en prenant le cadre particulier des approches de couplage de UML et B. Cependant, il est à noter que notre contribution s'inscrit dans un contexte plus global et peut s'appliquer au couplage d'UML et d'autres langages formels (comme Z).

Dans ce chapitre nous avons distingué les notions de règles abstraites et concrètes qui permettent d'avoir une souplesse dans la définition du processus de transformation d'UML. En effet, nous avons montré comment des règles concrètes différentes peuvent implémenter la même règle abstraite ainsi que la possibilité d'utiliser plusieurs langages de transformation pour exprimer les règles concrètes. Outre ces contributions liées à la spécification de la transformation, notre plateforme permet de paramétrer et de configurer les transformations selon le niveau d'abstraction $\mathcal{M}^2$ ou $\mathcal{M}^1$. Le niveau $\mathcal{M}^2$ permet l'application uniforme de l'ensemble du processus de transformation sur tous les éléments UML en entrée, alors que le niveau $\mathcal{M}^1$ permet de forcer l'application de certaines règles pour des éléments particuliers du modèle. Cela permet de combiner des règles alternatives provenant d'approches diverses.

En perspective, nous songeons à améliorer notre outil pour la prise en compte de plus de règles de transformation et considérer d'autres diagrammes UML en entrée, comme par exemple les diagrammes d'états/transitions. Cela fera apparaître des défis intéressants portant sur la gestion adéquate des dépendances entre règles de transformation. Par exemple, transformer un attribut de classe dépend de la transformation de la classe dans laquelle cet attribut est encapsulé. Dans l'état actuel de notre outil les dépendances entre règles sont intrinsèques au processus de transformation et supposent que ce processus n'implique que des règles non-conflictuelles. Finalement, nous pensons qu'il est judicieux d'ajouter aux règles abstraites des éléments détaillés de spécification de règles. L'idée en est de pouvoir vérifier que les règles concrètes préservent la spécification des règles abstraites.

Dans le projet Selkis, la traduction du lien entre modèle fonctionnel et modèle de sécurité peut prendre plusieurs formes. Il serait intéressant d'exprimer cette diversité de choix par une multi-transformation.

Chapitre 3

B4MSecure : une plateforme intégrée Mode d'emploi et visite guidée

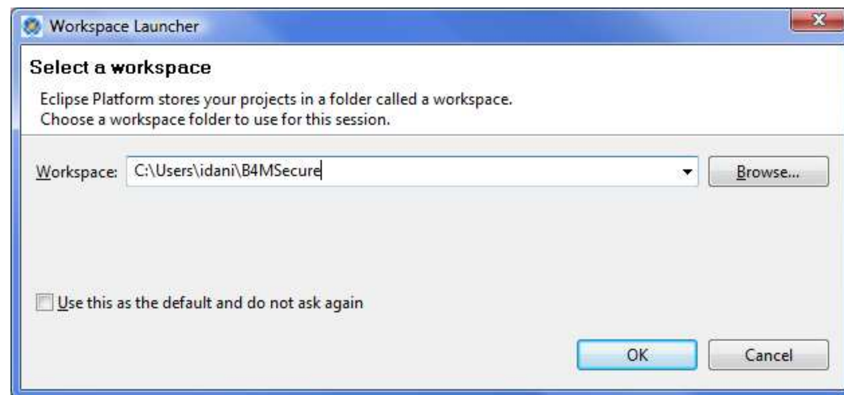
Dans le chapitre précédent nous nous sommes focalisés sur les principes fondateurs de l'approche de multi-transformation qui régit la traduction d'un modèle UML source. L'outil de multi-transformation a été utilisé dans la plateforme B4MSecure que nous avons développée en vue de générer les spécifications B pour un modèle fonctionnel étendu par un modèle RBAC. Comme mentionné dans le chapitre 1 le lien entre le modèle fonctionnel et le modèle de sécurité est très étroit. Nous avons donc choisi une transformation particulière du modèle fonctionnel que nous avons intégrée dans l'outil de multi-transformation, et ce sous forme d'une configuration. Les specifications B issues de cette configuration constituent une base pour l'approche de transformation du modèle de sécurité. Dans la suite, nous n'allons pas revenir sur les principes de cette traduction (déjà décrits et mis à jour dans les livrables 6.1.2, 3.2 et 3.3). Nous allons plutôt présenter une visite guidée de B4MSecure. Notons qu'une version de ce chapitre est en ligne (http ://b4msecure.forge.imag.fr/UserManual.pdf) et sera actualisée suite aux évolutions futures de notre plateforme.

3.1. Téléchargement et lancement

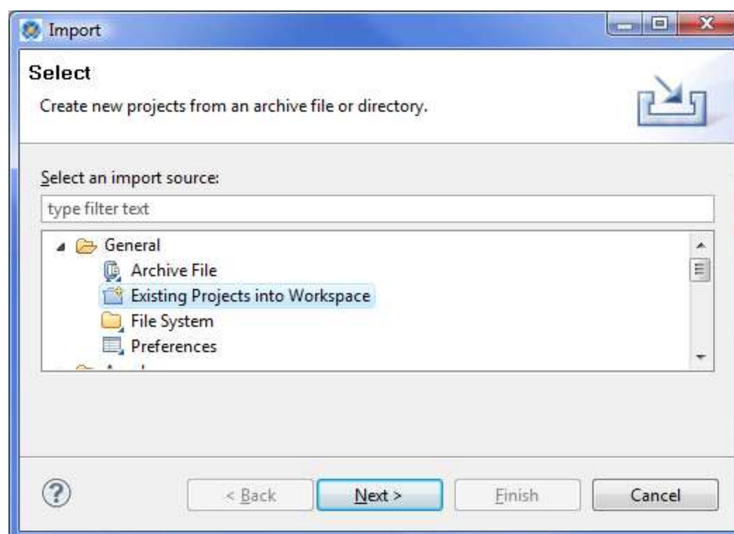
L'outil B4MSecure se présente sous la forme d'une plateforme Eclipse intégrant tous les plugins nécessaires et nécessitant Java 1.6. Il peut être téléchargé à partir du lien suivant :

http://b4msecure.forge.imag.fr/

1. Décompresser le fichier B4MSecure.zip
2. Lancer eclipse.exe
3. Créer un workspace (ici on nommera le workspace B4MSecure)

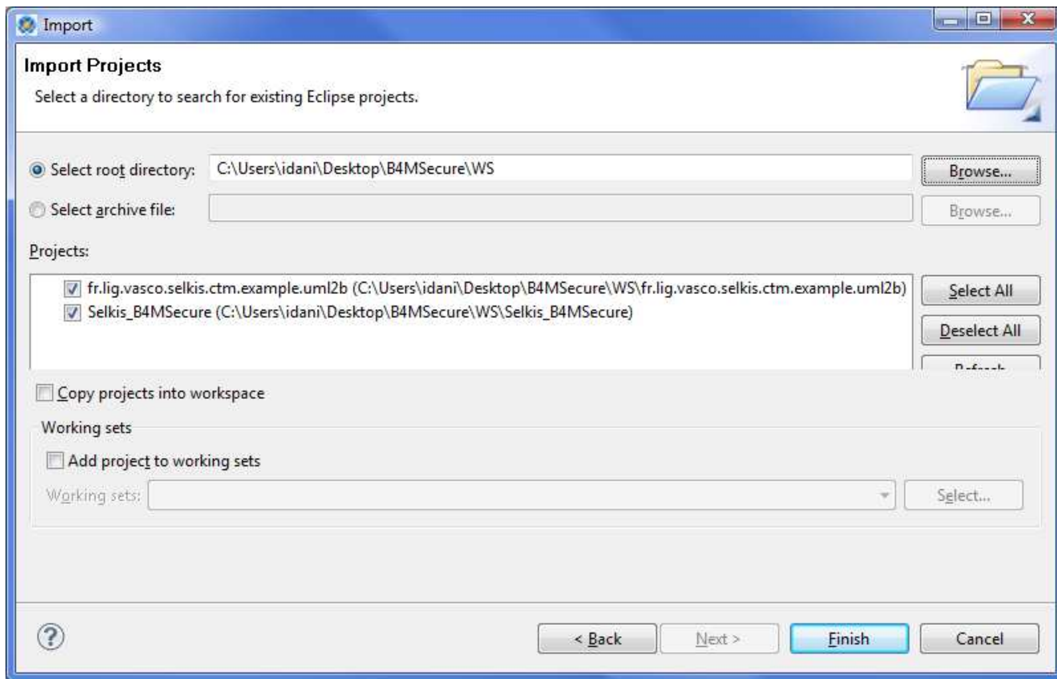


4. Dans le menu : File / Import… sélectionner General / Existing Projects into Workspace, ensuite cliquer sur Next.



5. Dans « Select root directory » sélectionner le répertoire WS se trouvant dans le dossier où vous avez décompressé le fichier B4MSecure.zip ; ensuite, cochez les deux projets associés :
   - fr.lig.vasco.selkis.ctm.example.uml2b
   - Selkis_B4MSecure

Comme il s'agit de projets Java, il est nécessaire de cocher la case « Copy projects into workspace ». Cliquer ensuite sur Finish.



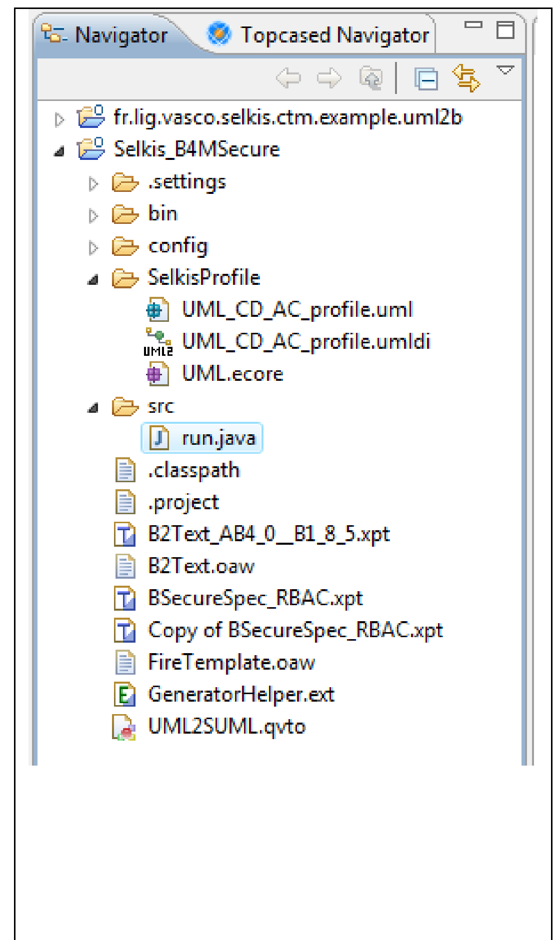
L'outil B4MSecure est Open Source, et se présente sous la forme de deux projets Eclipse :

1. fr.lig.vasco.selkis.ctm.example.uml2b : contient les règles Java de la transformation du diagramme de classes fonctionnel vers B. Cette partie de la transformation a été illustrée par la figure 1.1 (page 2).

2. Selkis_B4MSecure :

– contient les règles QVTo d'extraction de l'instance du méta-modèle de sécurité à partir d'un modèle UML stéréotypé (fichier UML2SUML.qvto). Cette partie de la transformation a été illustrée par la figure 1.2 (page 3).
– contient les templates xPand de génération de code B à partir d'une instance du méta-modèle de B (fichier B2Text_AB4_0__B1_8_5.xpt) ainsi que modèle de sécurité (fichier BSecureSpec_RBAC.xpt). Cette partie de la transformation a été illustrée par la figure 1.3 (page 4).
– permet d'invoquer le moteur de transformation du modèle fonctionnel vers B, ainsi que la traduction du modèle de sécurité.

Le fichier java run.java permettra d'agencer l'ensemble de cette transformation.



3.2. Création d'un modèle

L'édition des différents modèles se fait en Topcased. Ce manuel, ne décrivant pas en profondeur les spécificités de cet éditeur, une certaine maîtrise de son fonctionnement est donc nécessaire. Pour récupérer la documentation de Topcased vous pouvez vous rendre sur : http://www.topcased.org

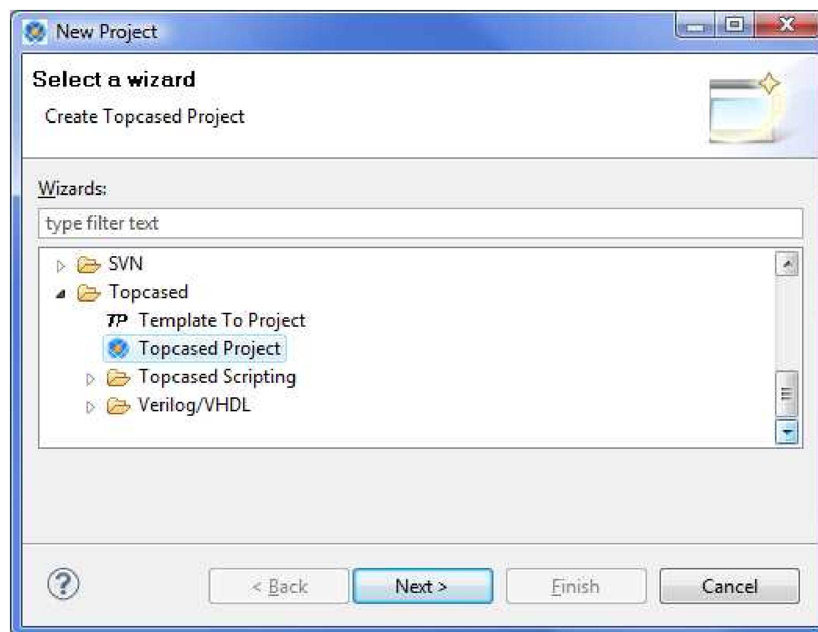
Notez que dans ce qui suit un exemple de modèle UML édité par Topcased est présenté. Ce modèle peut aussi être téléchargé à partir du lien ci-dessous et importé comme un projet Eclipse :

http://b4msecure.forge.imag.fr/TopcasedB4MSecure.zip.

3.2.1. Projet Topcased

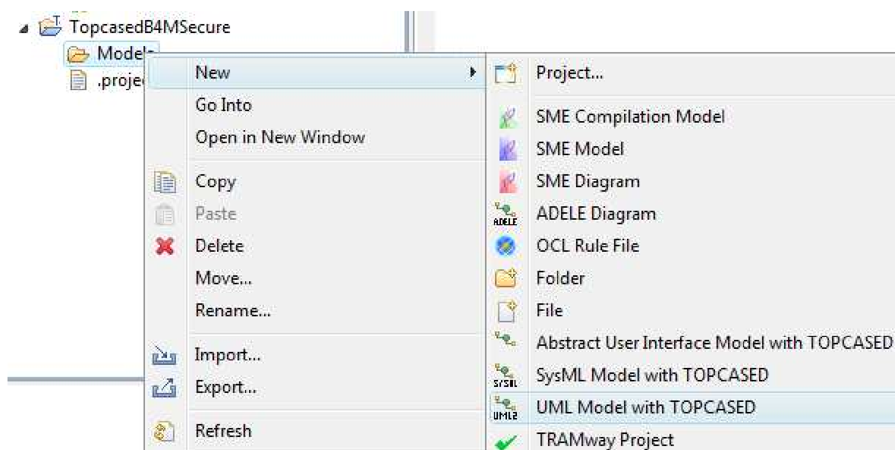
Créer un nouveau projet Topcased :

1. Menu File / New / Project…
2. Sélectionner Topcased / Topcased Project



On appellera ici le projet **`TopcasedB4MSecure`**

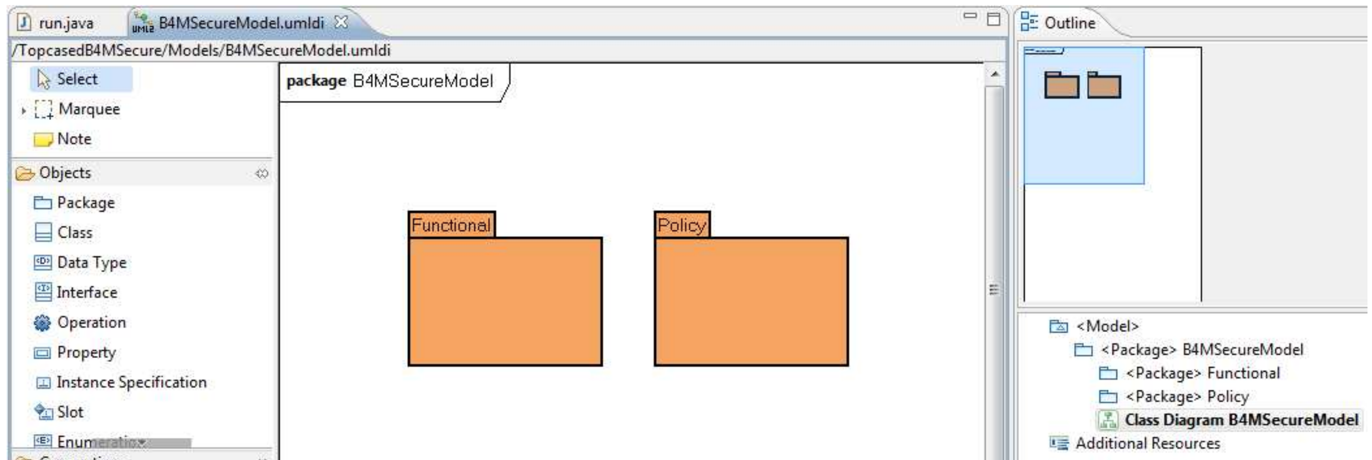
3. Faire un clic droit sur le dossier **`Model`** du projet **`TopcasedB4MSecure`** ensuite cliquer sur « UML Model with TOPCASED »



4. Nommer le modèle (ici on le nommera B4MSecureModel)
5. Dans l'option Diagram, sélectionner « Class diagram »
6. Cliquer sur Finish.

3.2.2. Edition et structuration du modèle

Commencer par créer deux packages : l'un pour le modèle fonctionnel, l'autre pour le modèle de sécurité. On nommera ici ces packages respectivement « Functional » et « Policy ».



A. Création du modèle fonctionnel

Faire un double clic sur le package « Functional » et sélectionner « Class Diagram ». Construire le diagramme de classes suivant :

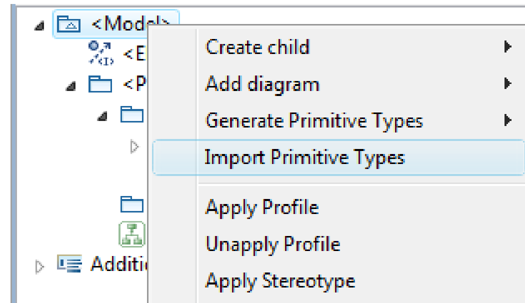
Dans le diagramme de classes, l'attribut Data est public et optionnel, et l'attribut Valid est privé, obligatoire et initialisé à false. L'étiquette {readOnly} de l'attribut Valid indique qu'on ne générera pas l'opération B de modification. L'association mRecords indique qu'un acte de soin doit être rattaché à un patient, et qu'un patient peut disposer de plusieurs actes de soin. Cette association est navigable dans les deux sens. L'édition de toutes ces informations se fait dans la fenêtre « Properties ».

N.B.

1. S'assurer que, dans la fenêtre « Properties » de l'association mRecords, la case « isUnique » est cochée pour les deux extrémités de l'association. L'outil ne considère pas la transformation de collections dont les éléments sont redondants.

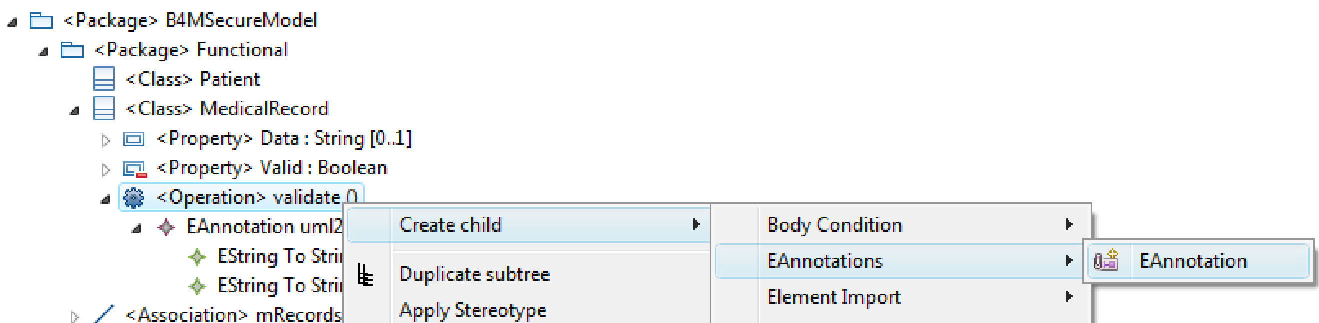
2. Un attribut, dont la propriété « isUnique » est cochée, est un attribut qui dispose d'une valeur unique pour toutes les instances de sa classe. Lors de la transformation vers B cette propriété est prise en compte. Pour l'exemple ci-dessus, décocher la case « isUnique » pour les deux attributs Data et Valid de la classe MedicalRecord. En effet, des instances différentes de MedicalRecord peuvent avoir les mêmes valeurs pour ces attributs.

3. Pour utiliser les types primitifs dans Topcased il faut faire un clic droit sur <Model> dans l'éditeur outline et choisir « Import Primitive Types »

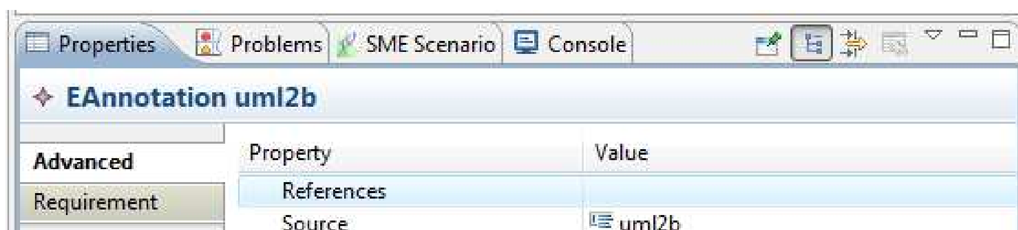


Opérations prédéfinies :

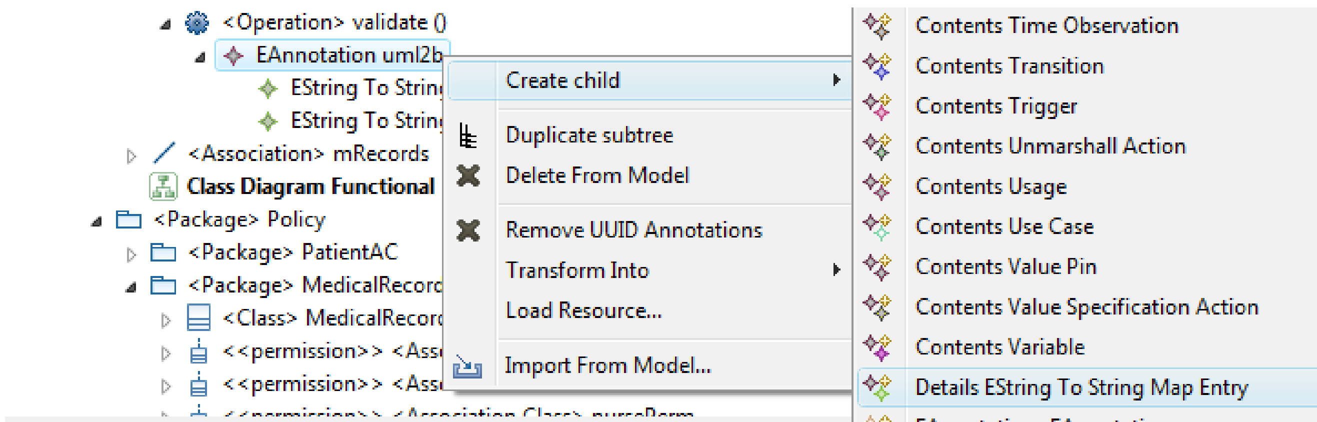
Outre les getters et les setters qui sont générés automatiquement par l'outil, certaines opérations, comme l'opération validate peuvent être introduites et spécifiées. Pour indiquer la pré-condition et le corps de cette opération on utilisera les annotations. Pour ce faire, dans l'éditeur Outline, faire un clic droit sur l'opération validate, ensuite sélectionner « Create child », ensuite « EAnnotations », ensuite cliquer sur « EAnnotations ».



Dans la fenêtre « Properties » de l'annotation que vous venez de créer, marquer « uml2b » dans le champ « Source » :

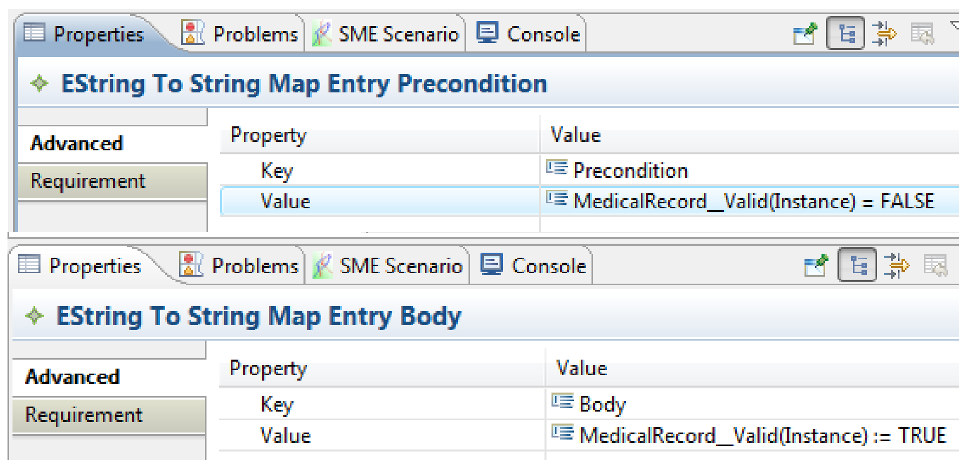


Dans l'éditeur Outline, faire un clic droit sur l'annotation uml2b ensuite sélectionner le menu « Create child » et cliquer sur « Details EString To String Map Entry ».



Cette action permet d'ajouter des détails (couples clé/valeur) à l'annotation. On ajoutera successivement les couples :

- Precondition / MedicalRecord__Valid(Instance) = FALSE
- Body / MedicalRecord__Valid(Instance) := TRUE



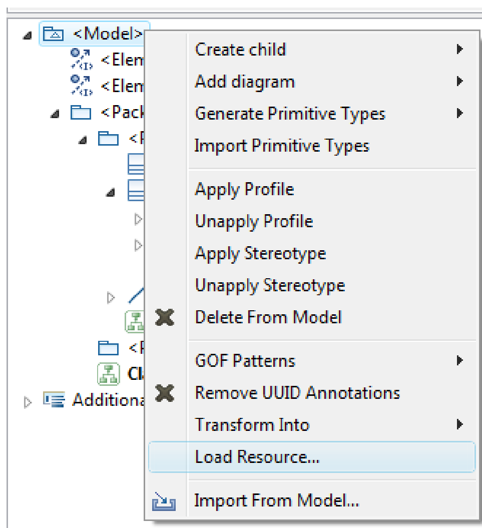
Ceci permet de générer la spécification de l'opération « validate » comme suit :

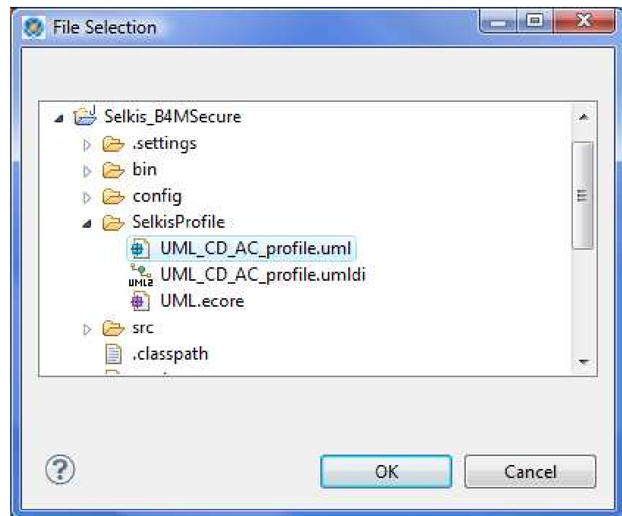
```
MedicalRecord__validate(Instance)=
PRE
      Instance : MedicalRecord
      & MedicalRecord__Valid(Instance) = FALSE
        /* Precondition generated from annotation*/
THEN
      MedicalRecord__Valid(Instance) := TRUE
      /* Body generated from annotation */
END;
```

B. Création du modèle de sécurité

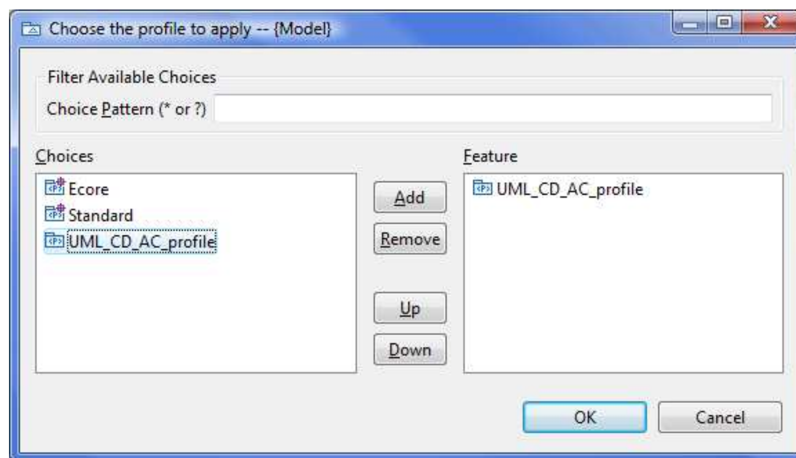
<u>Profil de sécurité</u>

Il faut d'abord charger le profil de sécurité. Pour ce faire, dans l'éditeur Outline, faire un clic droit sur <Model> et choisir « Load Resource ». Ensuite cliquer sur « Browse Workspace » et choisir le fichier **`UML_CD_AC_profile.uml`** se trouvant dans **`Selkis_B4MSecure/SelkisProfile`**.



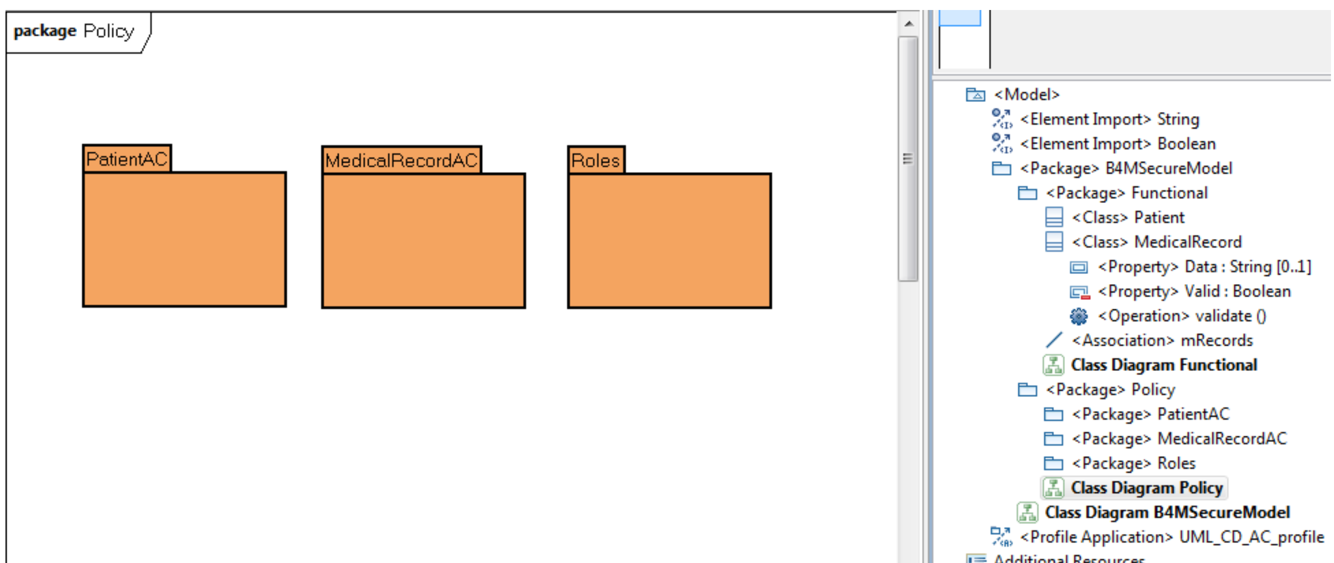


Il faut maintenant appliquer le profil de sécurité au modèle. Pour ce faire, dans l'éditeur Outline, faire un clic droit sur <Model> et choisir « Apply Profile ». Ajouter le profil UML_CD_AC_profile :

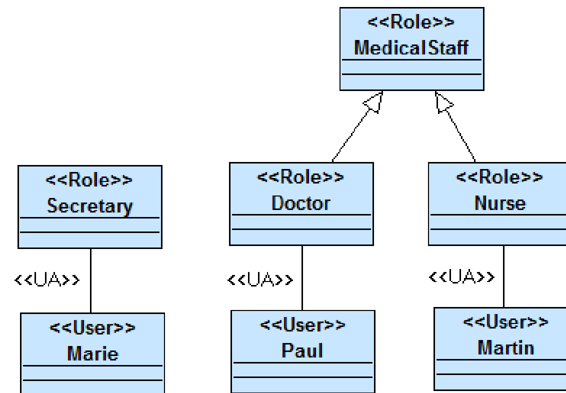


Edition du modèle de sécurité

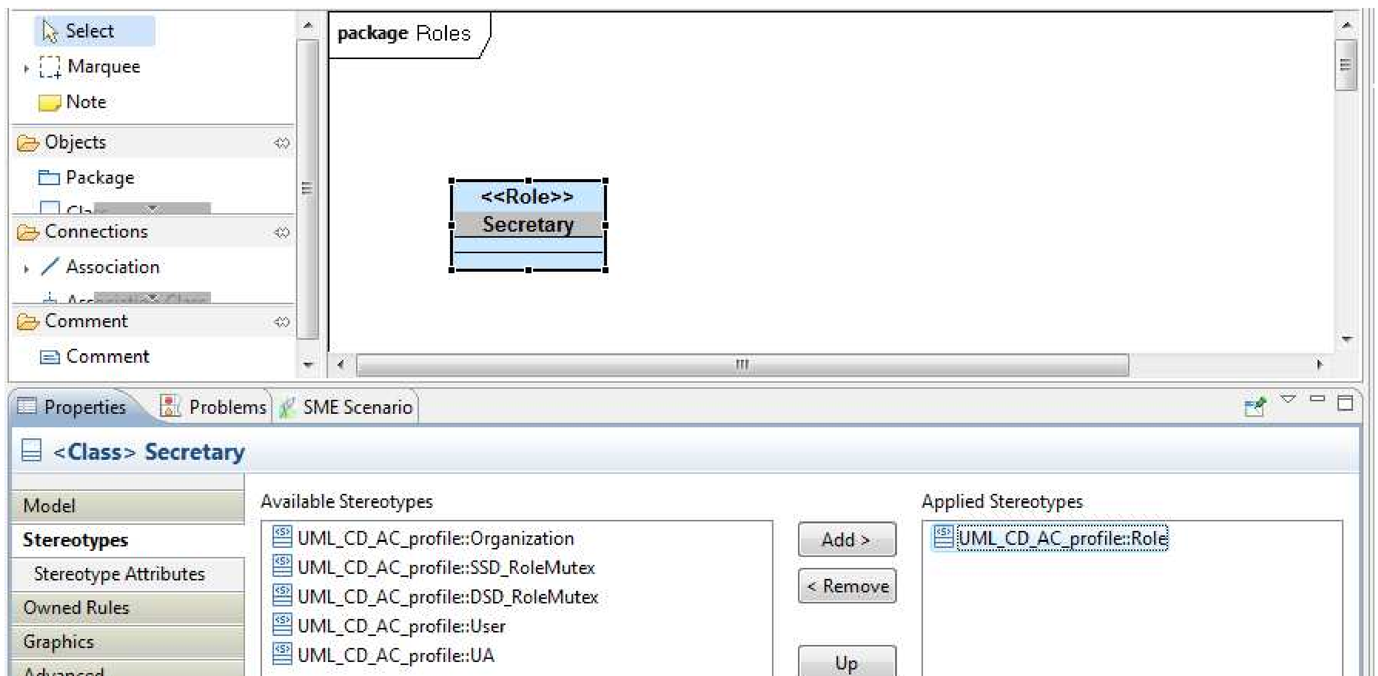
Faire un double clic sur le package « Policy » et sélectionner « Class Diagram ». Construire trois packages PatientAC, MedicalRecordAC et Roles.



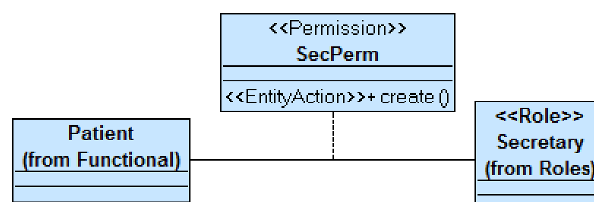
Les deux premiers packages vont contenir respectivement les politiques de contrôle d'accès associées aux classes Patient et MedicalRecord. Le package Role contiendra le modèle de rôles ainsi que l'affectation des utilisateurs aux différents rôles :



N.B. Pour ajouter un stéréotype, sélectionner une classe et choisir l'onglet stéréotypes dans la fenêtre « Properties ».

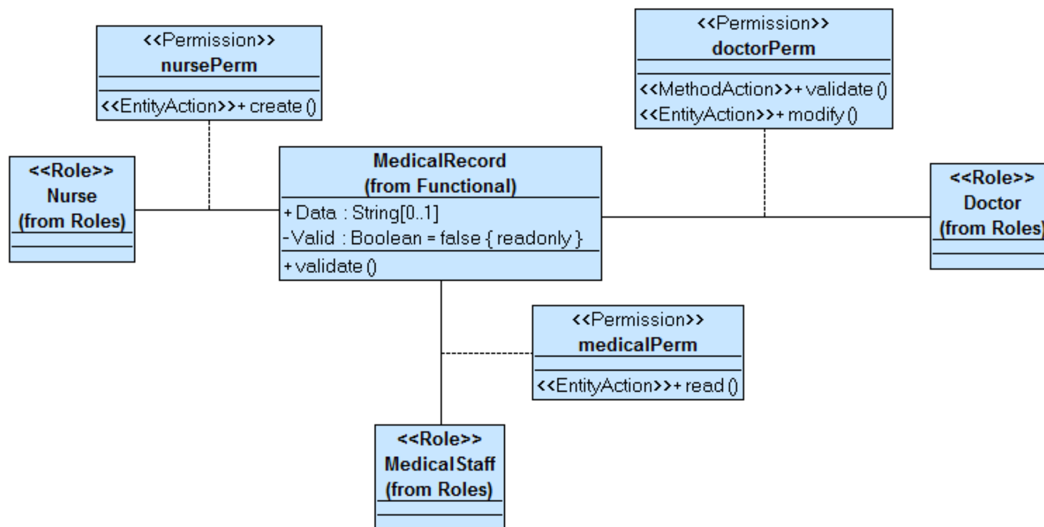


Le package PatientAC, contiendra la politique de contrôle d'accès à l'entité Patient. Celle-ci indique qu'une secrétaire peut créer un patient.



N.B. Dans cet exemple, la permission SecPerm est associée au package PatientAC, alors que la classe Patient et le rôle Secretary sont issus respectivement des packages Functional et Roles.

Le package MedicalRecordAC, contiendra la politique de contrôle d'accès à l'entité MedicalRecord :



Les règles de contrôle d'accès considérées dans ce modèle sont les suivantes :

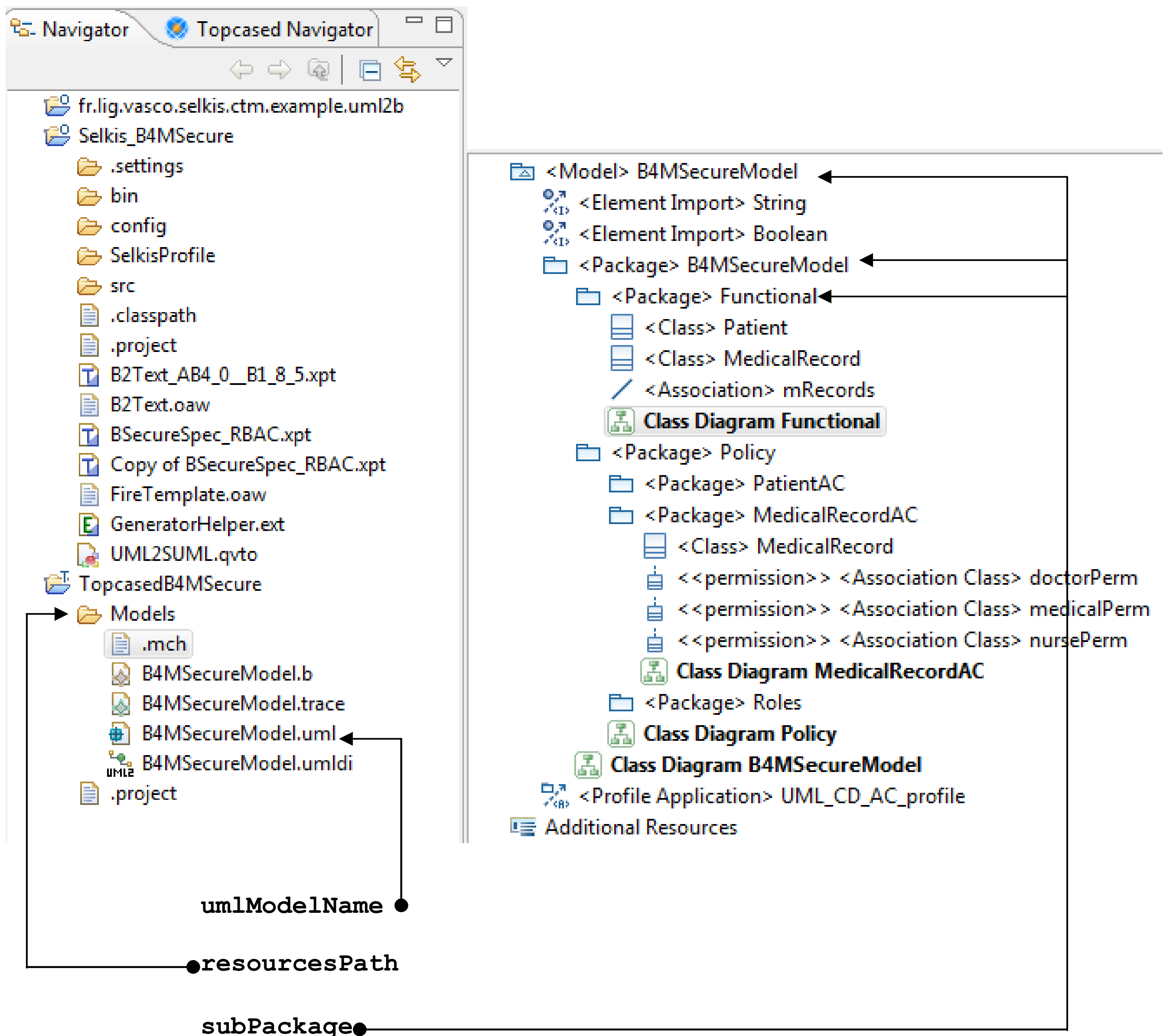
- Toute personne du corps médical peut lire les informations d'un acte de soin. Concrètement les infirmières et les médecins ont un accès en lecture de l'attribut Data.
- Les infirmières peuvent uniquement créer des actes de soin.
- Les docteurs peuvent modifier les données d'un acte de soin et le valider.

3.3. Génération des spécifications B

3.3.1. Configuration

Dans la version actuelle de l'outil, cette étape de configuration passe par l'instanciation de certaines variables directement dans le code source de l'invocateur de transformations. Ces variables sont à éditer dans le fichier Selkis_B4MSecure/src/run.java et sont : **`umlModelName`**, **`subPackage`** et **`resourcesPath`**.

- umlModelName : fichier UML (**`umlModelName = "B4MSecureModel"`**)
- subPackage : le sous-package contenant le modèle fonctionnel (**`subPackage = "B4MSecureModel::B4MSecureModel::Functional"`**)
- resourcesPath : dossier contenant le modèle TopCased et dans lequel les spécifications B seront générées (**`resourcesPath= "../TopcasedB4MSecure/Models/"`**).



Topcased Modeling - Selkis_B4MSecure/src/run.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Selkis Scripts SmartQVT Bad Smells Run Window Help

Navigator | Topcased Navigator

- fr.lig.vasco.selkis.ctm.example.uml2b
- Selkis_B4MSecure
  - .settings
  - bin
  - config
  - SelkisProfile
  - src
  - .classpath
  - .project
  - B2Text_AB4_0__B1_8_5.xpt
  - B2Text.oaw
  - BSecureSpec_RBAC.xpt
  - Copy of BSecureSpec_RBAC.xpt
  - FireTemplate.oaw
  - GeneratorHelper.ext
  - UML2SUML.qvto
- TopcasedB4MSecure

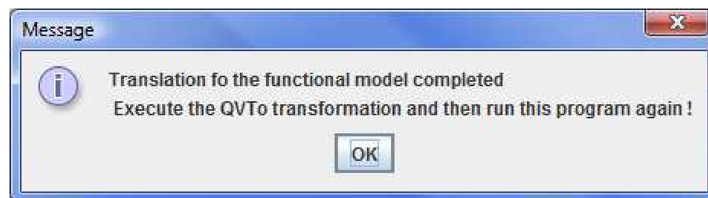
run.java | B4MSecureModel.umldi

```
import java.io.File;

public class run {
/*
umlModelName : fichier UML sans l'extension .uml
subPackage : le sous-package contenant le modèle fonctionnel
resourcesPath : dossier contenant le modèle TopCased et dans lequel les spécific
*/

    static String umlModelName  = "B4MSecureModel" ;
    static String subPackage    = "B4MSecureModel::B4MSecureModel::Functional";
    static String resourcesPath = "../TopcasedB4MSecure/Models/";

    public static void main(String[] args) {
        File bFile = new File(resourcesPath + umlModelName + ".b");
        if(!bFile.exists())
        {
            runMFunctional() ;
```

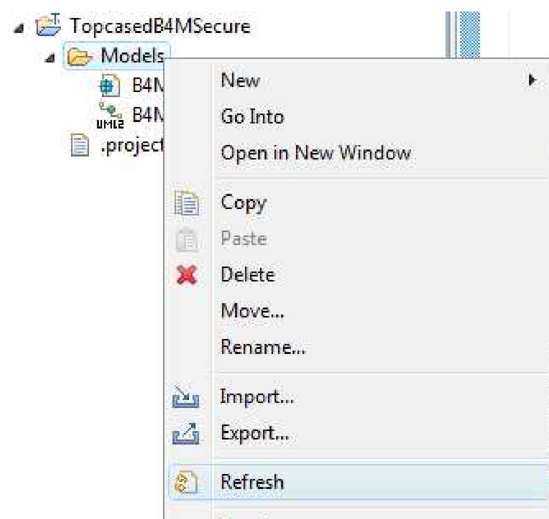
3.3.2. Traduction du modèle fonctionnel

Exécuter le fichier Selkis_B4MSecure/src/run.java ; ensuite cliquer sur : « Translate functional model ».

Un message d'affichera à la fin du processus de génération des spécifications B à partir du modèle fonctionnel :



La console affiche la trace de la transformation ainsi que les erreurs éventuelles. En cas d'erreur vérifier le modèle UML. Si la traduction s'est déroulée correctement, alors trois fichiers sont produits dans le dossier « TopcasedB4MSecure/Models ». Si ces fichiers n'apparaissent pas, faire un clic droit sur ce dossier ensuite cliquer sur « Refresh ».

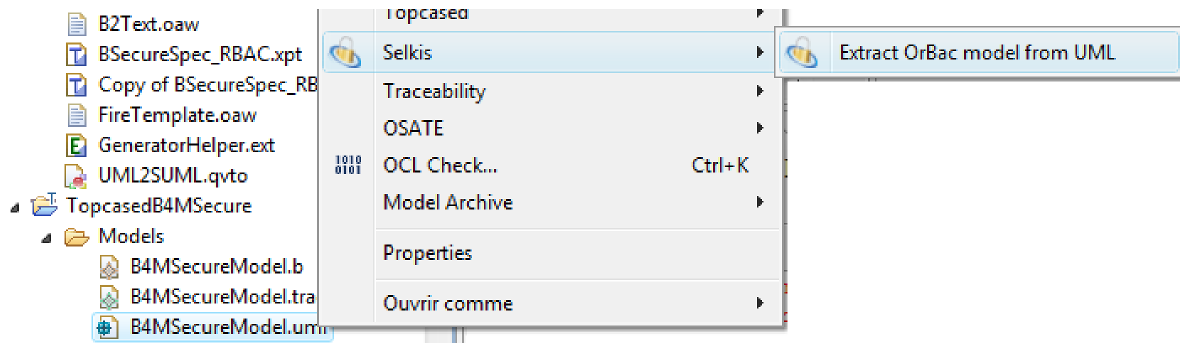


Ces fichiers sont :

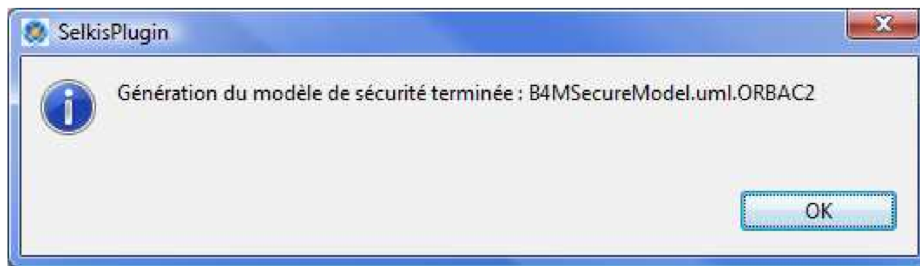
- **`B4MSecureModel.b`** : instance du méta-modèle de B
- **`B4MSecureModel.trace`** : trace de la traduction d'UML vers B
- **`Functional.mch`** : machine B du modèle fonctionnel au format textuel.

3.3.3. Extraction de l'instance du méta-modèle de sécurité

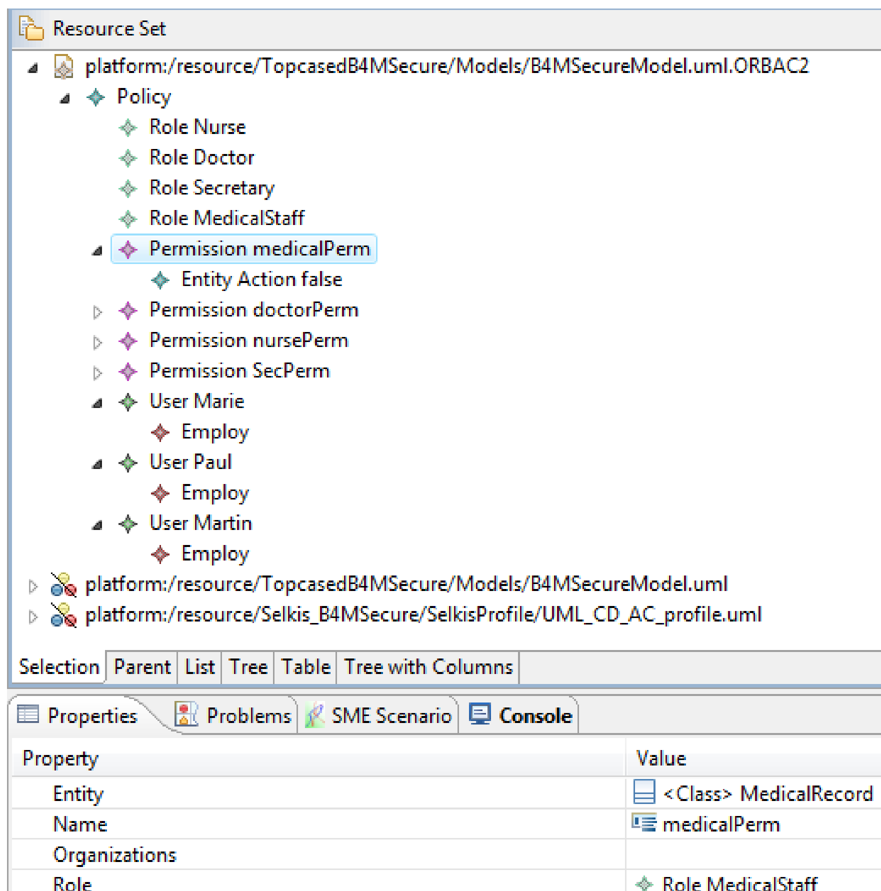
Rappelons que cette étape a été décrite à la page 3 (figure 1.2). Faire un clic droit sur le fichier **`B4MSecureModel.uml`**, ensuite cliquer sur Selkis / Extract OrBac model from UML.



Cette étape invoque les règles QVTo du fichier **UML2SUML.qvto** sur **B4MSecureModel.uml** en vue de produire une instance du méta-modèle de sécurité (variante du méta-modèle de SecureUML intégrant le concept d'organisation). A la fin de ce processus d'extraction, un message s'affiche :



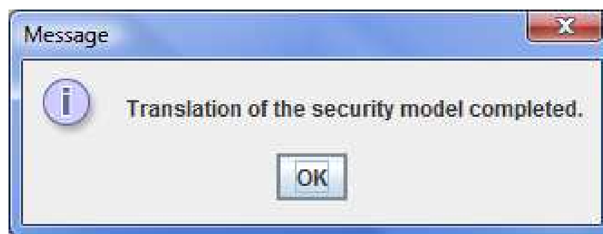
L'instance du méta-modèle de sécurité correspond au fichier **B4MSecureModel.uml.ORBAC2** généré dans le dossier « TopcasedB4MSecure/Models ».



3.3.4. Traduction du modèle de sécurité

Exécuter le fichier Selkis_B4MSecure/src/run.java ; ensuite cliquer sur : « Translate security model ».

Un message d'affichera à la fin du processus de génération des spécifications B à partir du modèle de sécurité :



La génération des spécifications B du modèle de sécurité nécessite l'existence des fichiers **`B4MSecureModel.b`** et **`B4MSecureModel.uml.ORBAC2`**. Si cette étape s'est déroulée correctement, alors trois machines B sont produites dans le dossier « TopcasedB4MSecure/Models » :

- ContextMachine.mch : contient l'ensemble des utilisateurs et des organisations du modèle
- UserAssignments.mch : contient les rôles, leurs liens et leurs contraintes, ainsi que l'affectation entre utilisateurs et rôles. Cette machine contient également les opérations de connexion d'un utilisateur dans une session.
- RBAC_Model.mch : contient la traduction B de la politique de contrôle d'accès.

Bibliographie

[EMP10] EMP. Eclipse modeling project, 2010.

[GMK+06] Jörn Guy Süß, Tim McComb, Soon-Kyeong Kim, Luke Wildman, and Geoffrey Watson. MDA-Based Re-engineering with Object-Z. In *International conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *LNCS*, pages 291–305. Springer, 2006.

[IBP09] Akram Idani, Jean-Louis Boulanger, and Laurent Philippe. Linking paradigms in safety critical systems. *International Journal of Computers and their Applications. Special Issue on the Application of Computer Technology to Public Safety and Law Enforcement*, 16(2) :111–120, 2009.

[Ida06] Akram Idani. *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B.* PhD thesis, Université de Grenoble 1, November 2006.

[ILL10] Akram Idani, Mohamed-Amine Labiadh, and Yves Ledru. Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *RSTI - Ingénierie des Systèmes d'Information (ISI)*, 15(3), 2010.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL : A model transformation tool. *Sci. Comput. Program.*, 72(1-2) :31–39, 2008.

[KBC05] Soon-Kyeong Kim, Damian Burger, and David-A. Carrington. An MDA Approach Towards Integrating Formal and Informal Modeling Languages. In *International Symposium of Formal Methods (FM 2005)*, volume 3582 of *LNCS*, pages 448–464. Springer, 2005.

[Lal02] Régine Laleau. Conception et développement formels d'applications bases de données. Hdr, Université d'Evry, 2002.

[Led01] Hung Ledang. Automatic translation from uml specifications to b. In *16th IEEE international conference on Automated software engineering*, page 436, 2001.

[LM00] Régine Laleau and Amel Mammar. An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations. In *15th IEEE international conference on Automated software engineering*, pages 269–272, 2000.

[Mey01] Eric Meyer. *Développements formels par objets : utilisation conjointe de B et d'UML.* PhD thesis, Université de Nancy 2, Mars 2001.

[NOW09] Francisco Assis Nascimento, Marcio F. S. Oliveira, and Flávio Rech Wagner. Using MDE for the formal verification of embedded systems modeled by UML sequence diagrams. In *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design : Chip on the Dunes.* ACM, 2009.

[OMG03] OMG. Mda guide version 1.0.1, 2003. www.omg.org/docs/omg/03-06-01.pdf.

[OMG08] OMG. Meta object facility (mof) 2.0 query/view/transformation specification version 1.0. http ://www.omg.org/spec/QVT/1.0/, April 2008.

[OMG09] OMG. *Unified Modeling Language : infrastructure (version 2.2)*. Object Management Group, February 2009. http ://www.omg.org/spec/UML/2.2/Infrastructure/PDF/.

[SB06] Colin Snook and Michael Butler. UML-B : Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :92–122, 2006.

[ZJBZ08] Tian Zhang, Frédéric Jouault, Jean Bézivin, and Jianhua Zhao. A MDE Based Approach for Bridging Formal Models. In *IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 113–116. IEEE CS Press, 2008.