Ingénierie des Modèles

Eric Cariou

Université de Pau et des Pays de l'Adour – France
UFR Sciences Pau – Département Informatique

Eric.Cariou@univ-pau.fr

Contenu des enseignements

- Cours (4h)
  - Introduction générale
    - Création et but de l'ingénierie des modèles
    - Concepts de modèle et associés
  - Méta-modélisation
  - Transformation de modèles
- TP (4h)
  - Méta-modélisation dans le contexte d'EMF
  - Transformations de modèles avec ATL et Kermeta

Introduction et définitions générales

Plan

- Pourquoi l'ingénierie des modèles (IDM)
  - Constat sur l'évolution des technologies
  - Approche MDA de l'OMG
- Pour quoi faire
  - But, problématique de l'IDM
- Définitions générales
  - Modèles, méta-modèles, transformations

Évolution des technologies

- Évolution permanente des technologies logicielles
- Exemple : systèmes distribués
  - Faire communiquer et interagir des éléments distants
- Évolution dans ce domaine
  - C et sockets TCP/UDP
  - C et RPC
  - C++ et CORBA
  - Java et RMI
  - Java et EJB
  - C# et Web Services
  - A suivre ...

Évolution des technologies

- Idée afin de limiter le nombre de technologies
  - Normaliser un standard qui sera utilisé par tous
- Pour les systèmes distribués
  - Normaliser un intergiciel (middleware)
  - C'était le but de CORBA
- CORBA : Common Object Request Broker Architecture
  - Norme de l'OMG : Object Management Group
    - Consortium d'industriels (et d'académiques) pour développement de standards

Évolution des technologies

- Principes de CORBA
  - Indépendant des langages de programmation
  - Indépendant des systèmes d'exploitation
  - Pour interopérabilité de toutes applications, indépendamment des technologies utilisées
- Mais CORBA ne s'est pas réellement imposé en pratique
  - D'autres middleware sont apparus : Java RMI
  - Plusieurs implémentations de CORBA
    - Ne réalisant pas tous entièrement la norme
  - Les composants logiciels sont arrivés
    - OMG a développé un modèle de composants basé sur CORBA : CCM (Corba Component Model)
    - Mais Sun EJB, MS .Net, Web Services sont là ...

Évolution des technologies

- De plus, « guerre » de la standardisation et/ou de l'universalité
  - Sun : Java
    - Plate-forme d'exécution universelle
    - Avec intergiciel intégré (RMI)
  - OMG : CORBA
    - Intergiciel universel
  - Microsoft et d'autres : Web Services
    - Interopératibilité universelle entre composants
      - Intergiciel = HTTP/XML
- Middleware ou *middle war ** ?

* citation de Jean-Marc Jézéquel

Évolution des technologies

- Evolutions apportent un gain réel
  - Communications distantes
    - Socket : envoi d'informations brutes
    - RPC/RMI/CORBA : appel d'opérations sur un élément distant *presque* comme s'il était local
    - Composants : meilleure description et structuration des interactions (appels d'opérations)
  - Paradigmes de programmation
    - C : procédural
    - Java, C++, C# : objet
      - Encapsulation, réutilisation, héritage, spécialisation ...
    - EJB, CCM : composants
      - Meilleure encapsulation et réutilisation, déploiement ...

Évolution des technologies

- Conclusion sur l'évolution des technologies
  - Nouveaux paradigmes, nouvelles techniques
  - Pour développement toujours plus rapide, plus efficace
  - Rend difficile la standardisation (désuétude rapide d'une technologie)
    - Et aussi car combats pour imposer sa technologie
- Principes de cette évolution
  - Évolution sans fin
  - La meilleure technologie est ... celle à venir

Évolution des technologies

- Quelles conséquences en pratique de cette évolution permanente ?
- Si veut profiter des nouvelles technologies et de leurs avantages :
  - Nécessite d'adapter une application à ces technologies
- Question : quel est le coût de cette adaptation ?
  - Généralement très élevé
    - Doit réécrire presque entièrement l'application
    - Car mélange du code métier et du code technique
    - Aucune capitalisation de la logique et des régles métiers

Évolution des technologies

- Partant de tous ces constats
  - Nécessité de découpler clairement la logique métier et de la mise en oeuvre technologique
  - C'est un des principes fondamentaux de l'ingénierie des modèles
    - Séparation des préoccupations (*separation of concerns*)
- Besoin de modéliser/spécifier
  - A un niveau abstrait la partie métier
  - La plate-forme de mise en oeuvre
  - De projeter ce niveau abstrait sur une plateforme

Model Driven Architecture

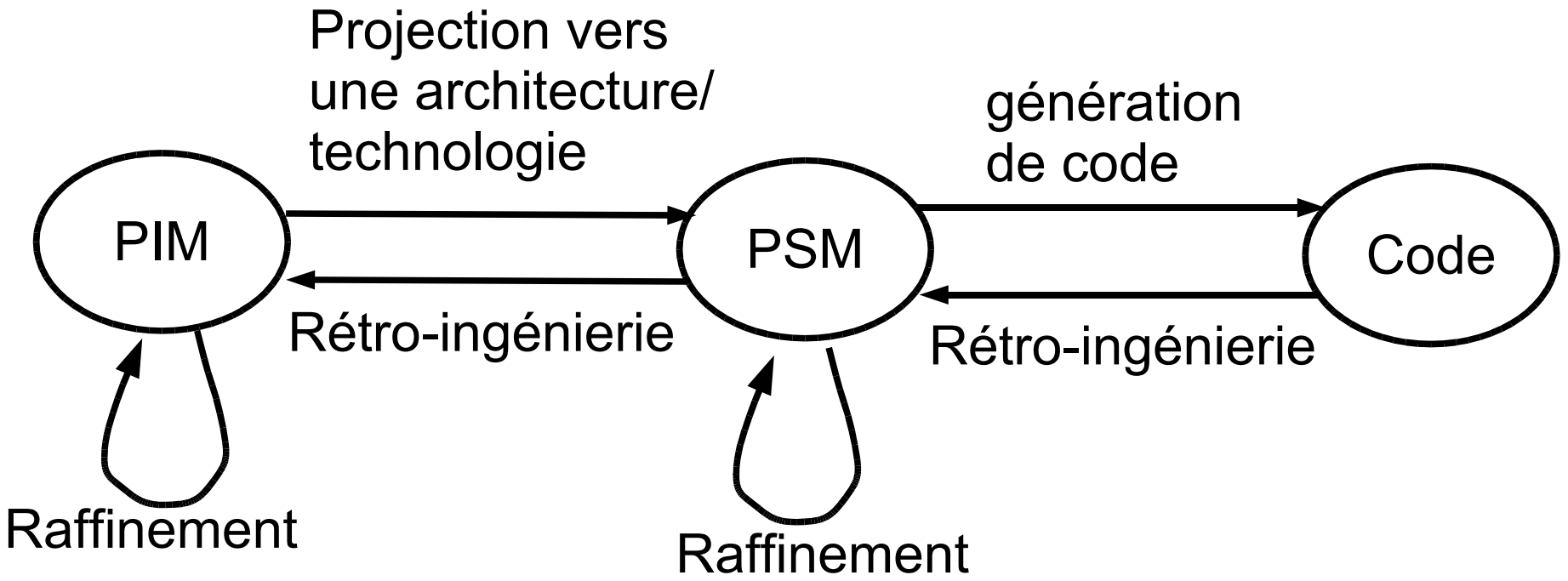
- Approche Model-Driven Architecture (MDA) de l'OMG
  - Origine de l'ingénierie des modèles
  - Date de fin 2000
- Le MDA est né à partir des constatations que nous venons de voir
  - Évolution continue des technologies
- But du MDA
  - Abstraire les parties métiers de leur mise en oeuvre
  - Basé sur des technologies et standards de l'OMG
    - UML, MOF, OCL, CWM, QVT ...

Model Driven Architecture

- Le MDA définit 2 principaux niveaux de modèles
  - PIM : Platform Independent Model
    - Modèle spécifiant une application indépendamment de la technologie de mise en oeuvre
    - Uniquement spécification de la partie métier d'une application
  - PSM : Platform Specific Model
    - Modèle spécifiant une application après projection sur une plate-forme technologique donnée
- Autres types de modèles
  - CIM : Computation Independent Model
    - Spécification du système, point de vue extérieur de l'utilisateur
  - PDM : Plateform Deployment Model
    - Modèle d'une plate-forme de déploiement

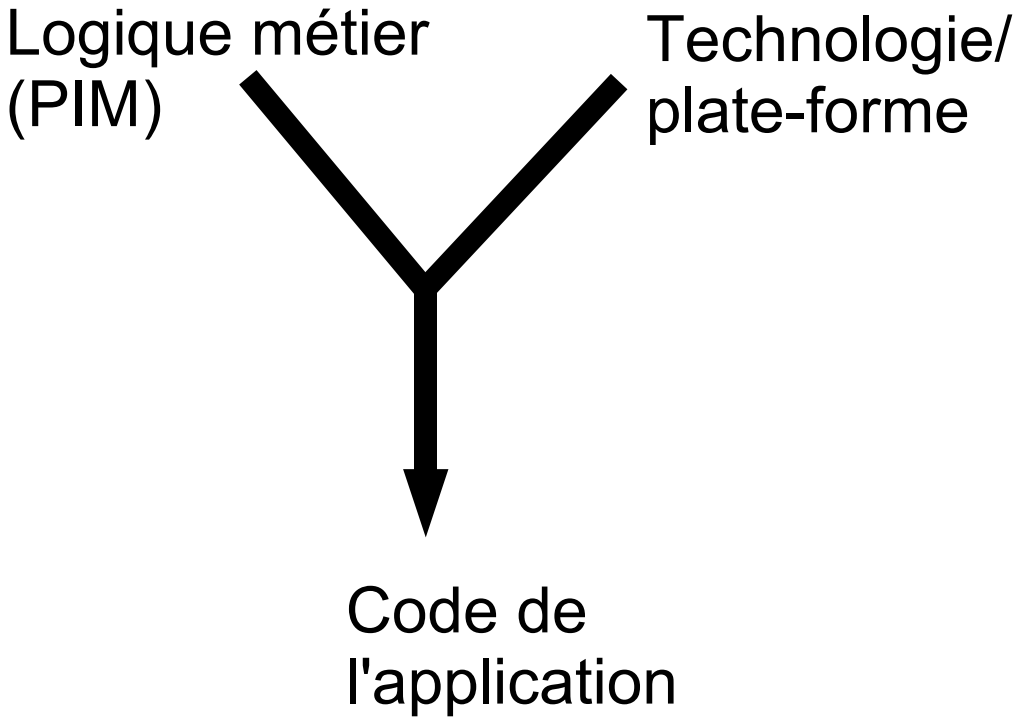
Model Driven Architecture

- Relation entre les niveaux de modèles



Model Driven Architecture

- Cycle de développement d'un logiciel selon le MDA
  - Cycle en Y
  - Plus complexe en pratique
    - Plutôt un cycle en épi



Model Driven Architecture

- Outils de mise en oeuvre du MDA
  - Standards de l'OMG
- Spécification des modèles aux différents niveaux
  - Langage de modélisation UML
    - Profils UML
  - Langage de contraintes OCL
  - Langages dédiés à des domaines particuliers (CWM ...)
- Spécification des méta-modèles
  - Meta Object Facilities (MOF)
- Langage de manipulation et transformation de modèles
  - Query/View/Transformation (QVT)

Model Driven Engineering

- Limites du MDA
  - Technologies OMG principalement
- Ingénierie des modèles
  - MDE : Model Driven Engineering
    - IDM : Ingénierie Dirigée par les Modèles
  - Approche plus globale et générale que le MDA
- Appliquer les mêmes principes à tout espace technologique et les généraliser
  - Espace technologique : ensemble de techniques/principes de modélisation et d'outils associés à un (méta)méta-modèle particulier
    - MOF/UML, EMF/Ecore, XML, grammaires de langages, bases de données relationnelles, ontologies ...
  - Le MDA est un processus de type MDE

Principes du MDE

- Capitalisation
  - Approche objets/composants
    - Capitalisation, réutilisation d'éléments logiciels/code
  - MDE
    - Capitalisation, réutilisation de (parties de) modèles : logique métier, règles de raffinements, de projection ...
- Abstraction
  - Modéliser, c'est abstraire ...
  - Par exemple abstraction des technologies de mise en oeuvre
    - Permet d'adapter une logique métier à un contexte
    - Permet d'évoluer bien plus facilement vers de nouvelles technologies

Principes du MDE

- Modélisation
  - La modélisation n'est pas une discipline récente en génie logiciel
  - Les processus de développement logiciel non plus
    - RUP, Merise ...
  - C'est l'usage de ces modèles qui change
  - Le but du MDE est
    - De passer d'une vision plutôt contemplative des modèles
      - A but de documentation, spécification, communication
    - A une vision réellement productive
      - Pour générer le code final du logiciel pour une technologie de mise en oeuvre donnée

Principes du MDE

- Séparation des préoccupations
  - Deux principales préoccupations
    - Métier : le coeur de l'application, sa logique
    - Plate-forme de mise en oeuvre
  - Mais plusieurs autres préoccupations possibles
    - Sécurité
    - Interface utilisateur
    - Qualité de service
    - ...
  - Chaque préoccupation est modélisée par un ... modèle
  - Intégration des préoccupations
    - Par transformation/fusion/tissage de modèles
      - Conception orientée aspect

Principes du MDE

- Pour passer à une vision productive, il faut
  - Que les modèles soient bien définis
    - Notion de méta-modèle
  - Pour que l'on puisse les manipuler et les interpréter via des outils
    - Avec traitement de méta-modèles différents simultanément
      - Pour transformation/passage entre 2 espaces technologiques différents
    - Référentiels de modèles et de méta-modèles
    - Outils et langages de transformation, de projection, de génération de code
    - Langages dédiés à des domaines (DSL : Domain Specific Language)

Définitions

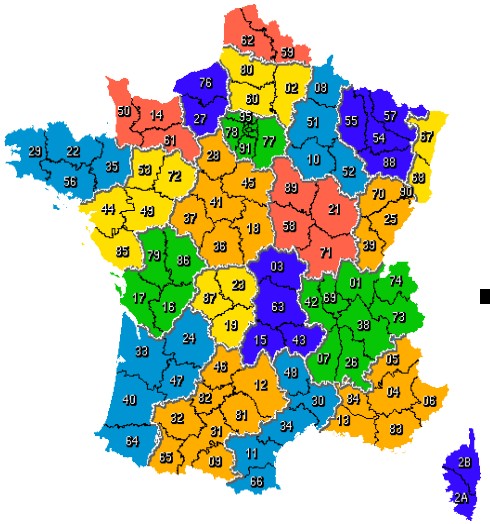
- Notions fondamentales dans le cadre du MDE
  - Modèle
  - Méta-modèle
  - Transformation de modèle
- Nous allons donc les définir précisément
  - En reprenant les notations et exemples des documents réalisés par l'AS CNRS MDA
    - *"L'ingénierie dirigée par les modèles. Au-delà du MDA",* collectif sous la direction de JM. Favre, J. Estublier et M. Blay-Fornarino, 2006, Hermes / Lavoisier

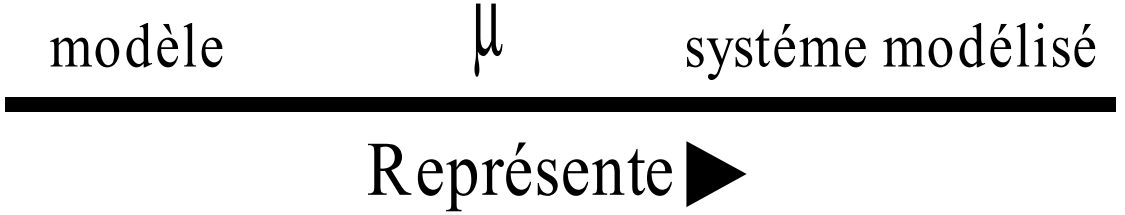
Modèle

- Un modèle est une description, une spécification partielle d'un système
  - Abstraction de ce qui est intéressant pour un contexte et dans un but donné
  - Vue subjective et simplifiée d'un système
- But d'un modèle
  - Faciliter la compréhension d'un système
  - Simuler le fonctionnement d'un système
- Exemples
  - Modèle économique
  - Modèle démographique
  - Modèle météorologique

Modèle

- Différence entre spécification et description
  - Spécification d'un système *à construire*
  - Description d'un système *existant*
- Relation entre un système et un modèle
  - RepresentationDe (notée μ)

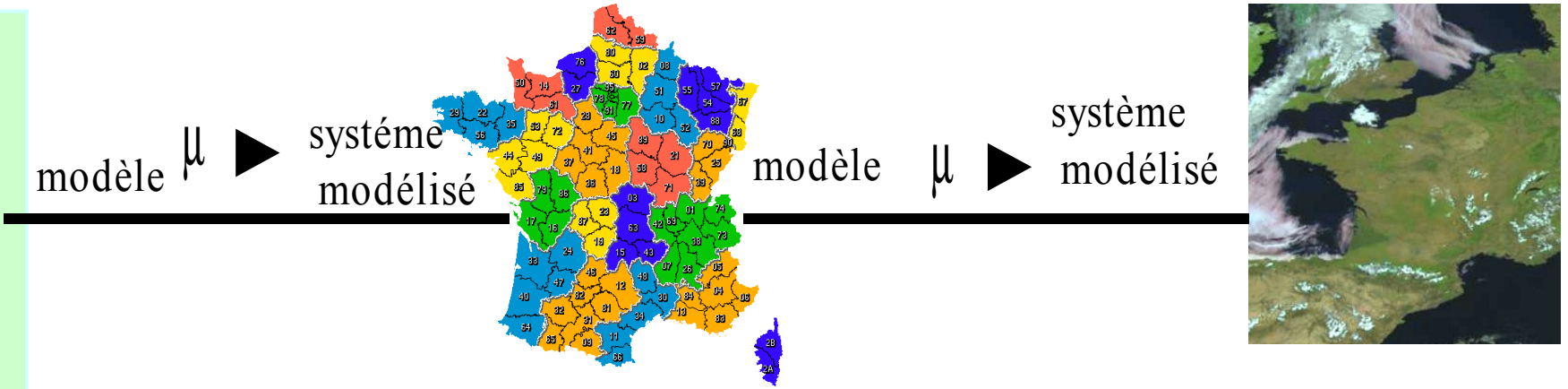




Modèle

- Un modèle représente un système modélisé
  - De manière générale, pas que dans un contexte de génie logiciel ou d'informatique
  - Un modèle peut aussi avoir le rôle de système modélisé dans une autre relation de représentation

```
<MAP name="france"
      taille="20x20">
  <region>
    <departement>
      38
    </departement>
    <departement>
      73
     </depart ement>
    ...
  </region>
...
```



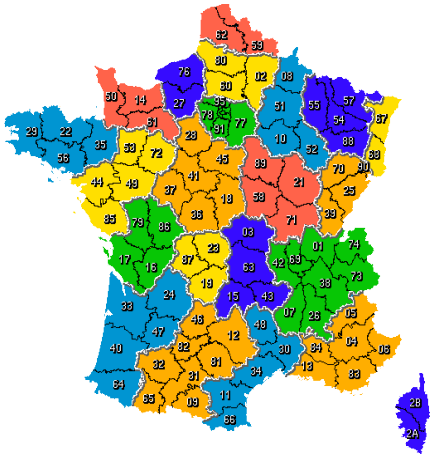
- Modèle XML de la carte de la France administrative qui est un modèle de la France « réelle »

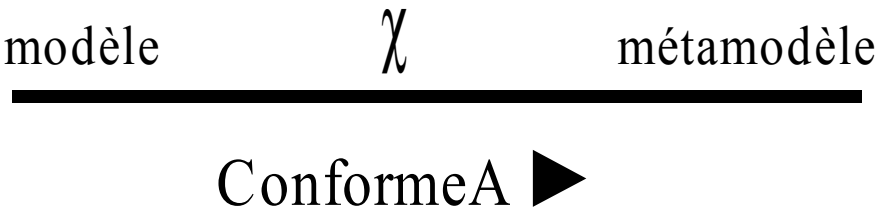
Modèle

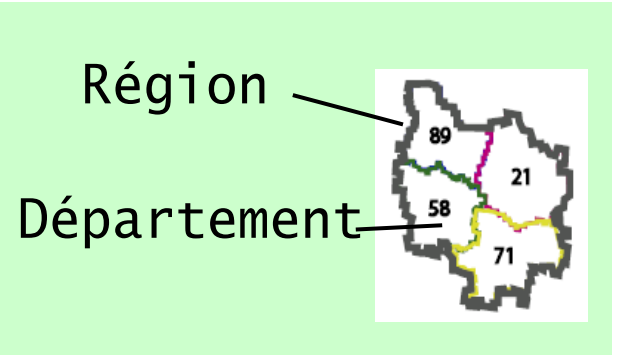
- Un modèle est écrit dans un langage qui peut être
  - Non ou peu formalisé, la langue naturelle
    - Le français, un dessin ...
  - Formel et bien défini, non ambigu
    - Syntaxe, grammaire, sémantique
    - On parle de méta-modèle pour ce type de langage de modèle
- Pour les modèles définis dans un langage bien précis
  - Relation de conformité
    - Un modèle est conforme à son méta-modèle
    - Relation EstConformeA (notée χ)

Méta-modèle

- Un modèle est conforme à son méta-modèle







- Notes
  - On ne parle pas de relation d'instanciation
    - Un modèle n'est pas une instance d'un méta-modèle
      - Instanciation est un concept venant de l'approche objet
      - Approche objet qui ne se retrouve pas dans tous les espaces technologiques
  - Un méta-modèle n'est pas un modèle de modèle
    - Raccourci ambigü à éviter
    - Le modèle XML de la carte administrative de la France n'est pas le méta-modèle des cartes administratives

Méta-modèle

- Cette relation de conformité est essentielle
  - Base du MDE pour développer les outils capables de manipuler des modèles
  - Un méta-modèle est une entité de première classe
- Mais pas nouvelle
  - Un texte écrit est conforme à une orthographe et une grammaire
  - Un programme Java est conforme à la syntaxe et la grammaire du langage Java
  - Un fichier XML est conforme à sa DTD
  - Une carte doit être conforme à une légende
  - Un modèle UML est conforme au méta-modèle UML

Méta-modèle et Langage

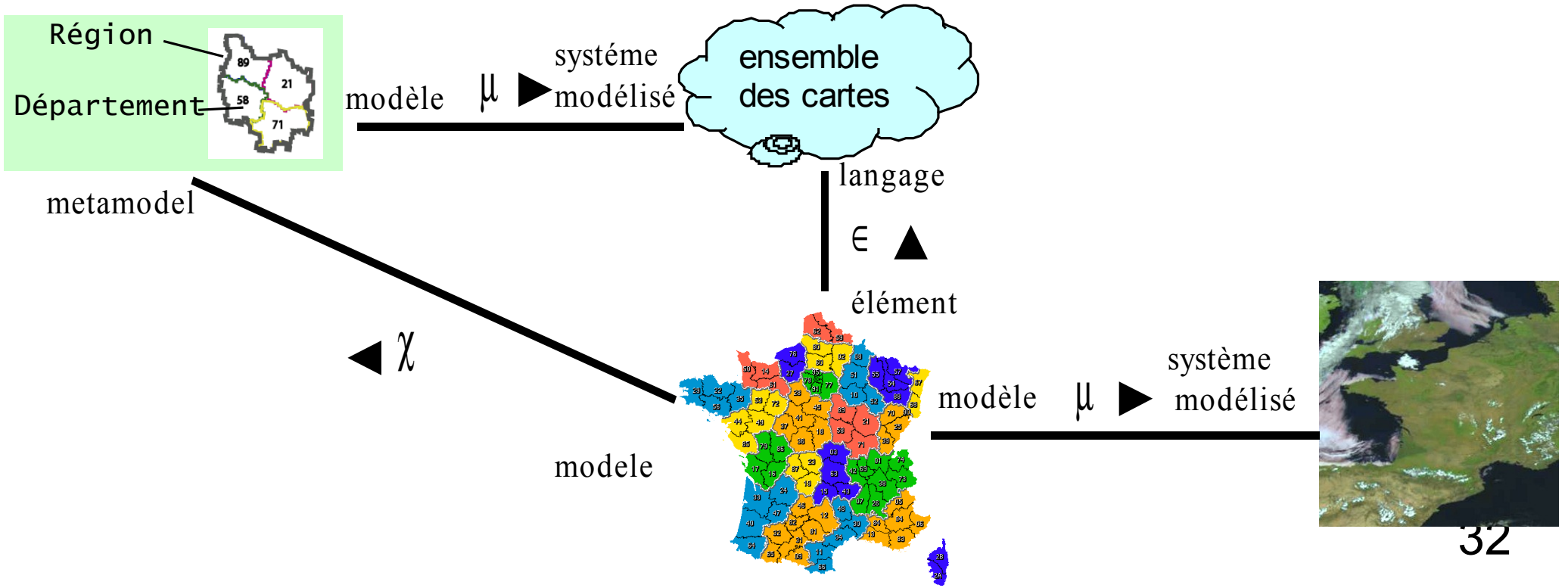
- Lien entre méta-modèle et langage
  - Un méta-modèle est un modèle qui définit le langage pour définir des modèles
  - Langage
    - Système abstrait
  - Méta-modèle
    - Définition explicite et concrète d'un langage
    - Un méta-modèle modélise alors un langage
  - Un méta-modèle n'est donc pas un langage

Méta-modèle et Langage

- En linguistique
  - Un langage est défini par l'ensemble des phrases valides écrites dans ce langage
  - Une grammaire est un modèle de langage
    - Une grammaire est un méta-modèle
- Relation entre langage et modèle
  - Un modèle est un élément valide de ce langage
    - Une phrase valide du langage en linguistique
  - Relation d'appartenance
    - AppartientA, notée ϵ

Relations générales

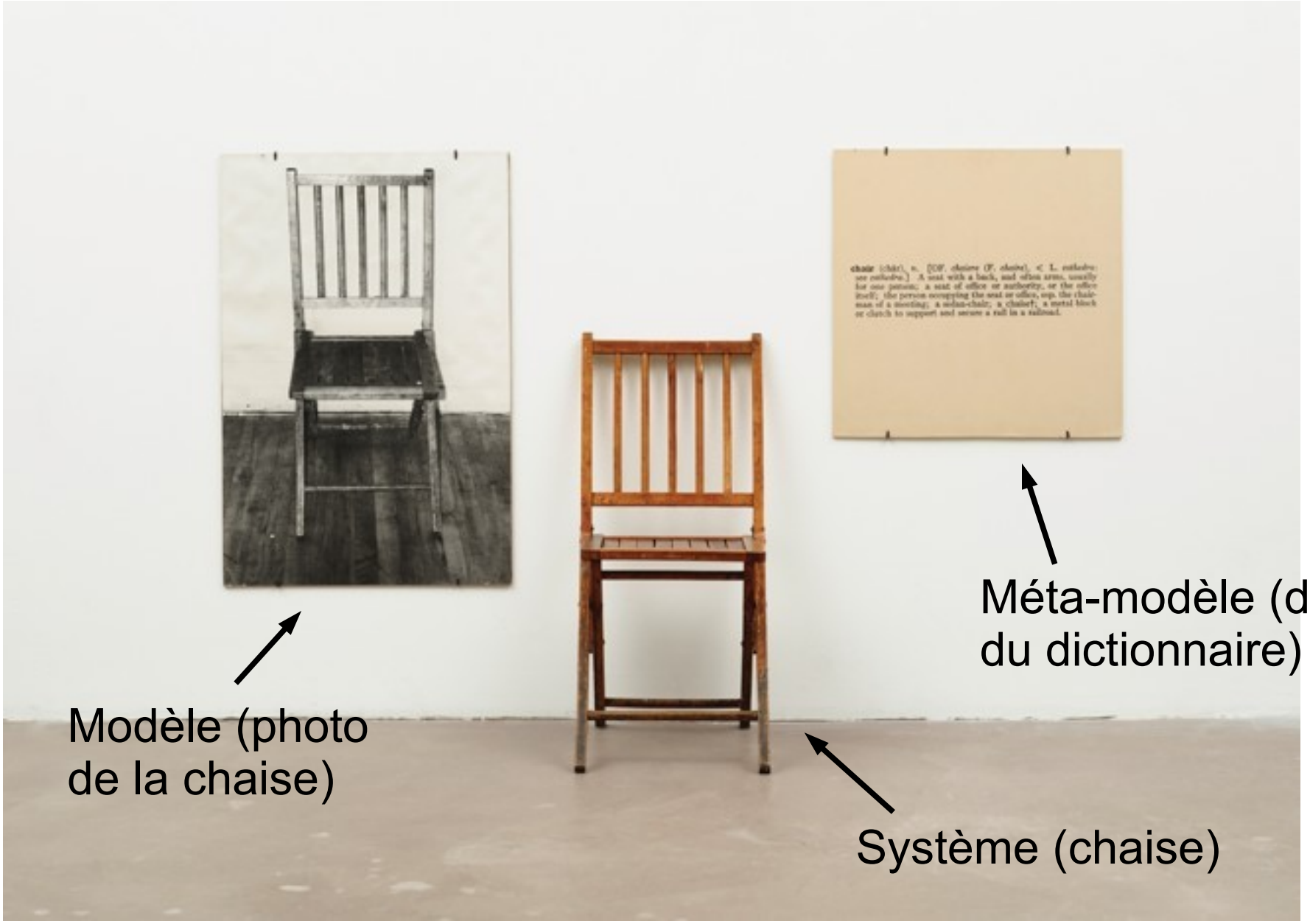
- Exemple de la carte
  - Une carte modélise un pays selon un point de vue
  - Le méta-modèle de la carte est sa légende
  - La légende définit un ensemble de cartes valides
  - Une carte conforme à une légende appartient à cet ensemble



Relations générales

- Exemple avec un langage de programmation
  - Un programme Java modélise/simule un système (le programme à l'exécution)
  - Un programme Java est conforme à la grammaire du langage Java
  - La grammaire de Java modélise le langage Java et donc tous les programmes valides
  - Le programme Java appartient à cet ensemble
- Exemple avec UML
  - Un diagramme UML modélise un système
  - Un diagramme UML est conforme au méta-modèle UML
  - Le méta-modèle UML définit l'ensemble des modèles UML valides
  - Un modèle UML appartient à cet ensemble

Les concepts de l'IDM en une image ...



« One and three chairs », Joseph Kossuth, 1965

Conclusion

- Le MDE est une nouvelle approche pour concevoir des applications
  - En plaçant les modèles et surtout les méta-modèles au centre du processus de développement dans un but productif
    - Les modèles sont depuis longtemps utilisés mais ne couvrait pas l'ensemble du cycle de vie du logiciel
    - Les méta-modèles existent aussi depuis longtemps (grammaires, DTD XML, ...) mais peu utilisés en modélisation « à la UML »
    - Nouvelle vision autour de notions déjà connues : le méta-modèle devient le point clé de la modélisation
      - Automatisation de la manipulation des modèles
      - Création de DSL : outillage de modélisation dédié à un domaine
  - Automatisation du processus de développement
    - Application de séries de transformations, fusions de modèles
  - Les grands éditeurs de logiciel suivent ce mouvement
    - IBM, Microsoft ...

Conclusion

- Problématiques, verrous technologiques, besoin en terme d'outils dans le cadre du MDE
  - Définition précise de modèles et de méta-modèles
  - Langage et outils de transformations, fusion de modèles, tissage de modèles/aspects
  - Traçabilité dans le processus de développement, reverse-engineering
  - Gestion de référentiels de modèles et méta-modèles
    - Y compris à large échelle (nombre et taille des (méta)modèles)
  - Exécutabilité de modèles
    - Connaissance et usage du modèle à l'exécution, modèle intégré dans le code
  - Validation/test de modèles et de transformations

Méta-modélisation

Principales normes modélisation OMG

- MOF : Meta-Object Facilities
  - Langage de définition de méta-modèles
- UML : Unified Modelling Language
  - Langage de modélisation
- CWM : Common Warehouse Metamodel
  - Modélisation ressources, données, gestion d'une entreprise
- OCL : Object Constraint Language
  - Langage de contraintes sur modèles
- XMI : XML Metadata Interchange
  - Standard pour échanges de modèles et méta-modèles entre outils

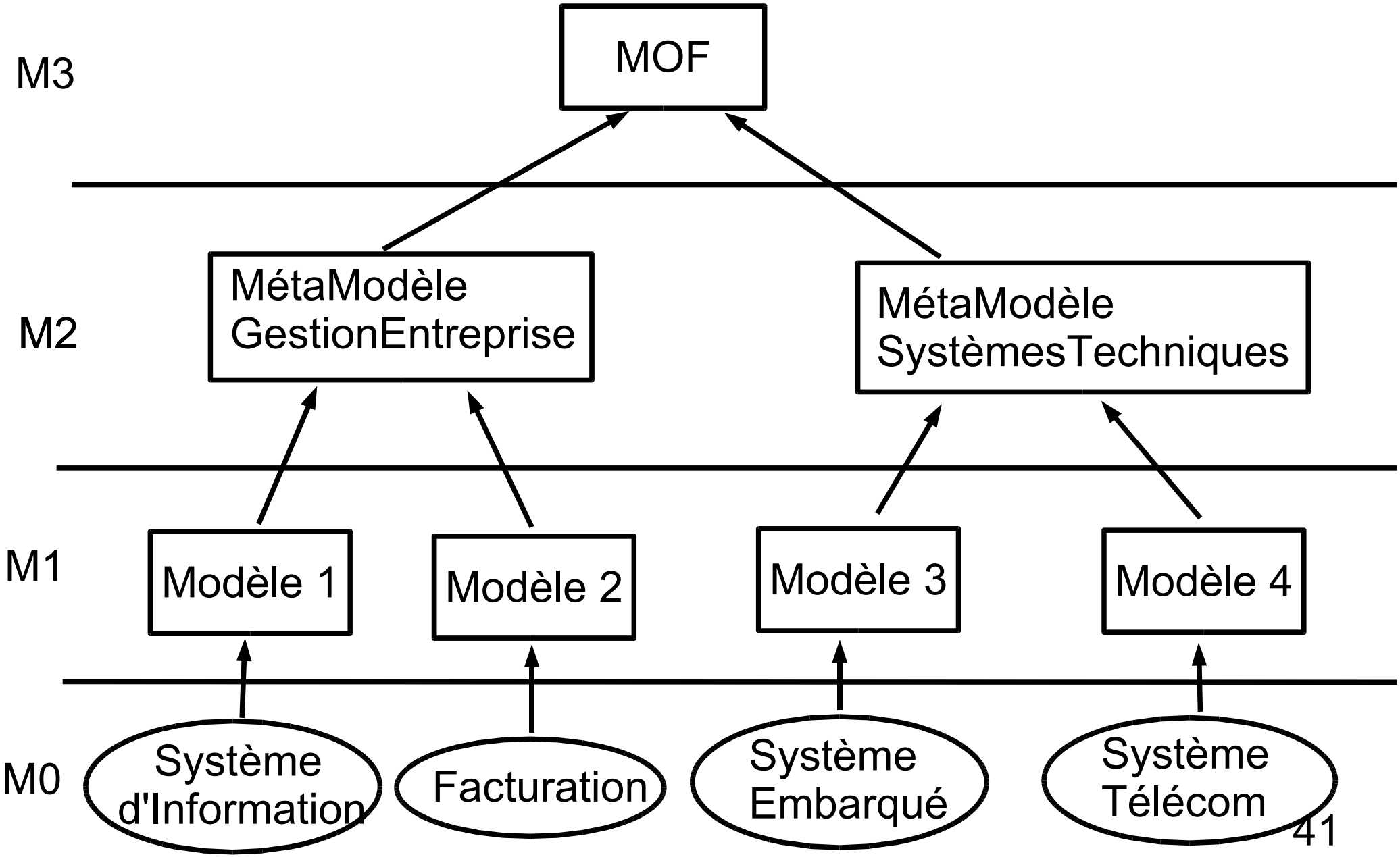
Normes OMG de modélisation

- Plusieurs de ces normes concernent la définition et l'utilisation de méta-modèles
  - MOF : but de la norme
  - UML et CWM : peuvent être utilisés pour en définir
  - XMI : pour échange de (méta-)modèles entre outils
- MOF
  - C'est un méta-méta-modèle
    - Utilisé pour définir des méta-modèles
  - Définit les concepts de base d'un méta-modèle
    - Entité/classe, relation/association, type de données, référence, package ...
  - Le MOF peut définir le MOF

Hiérarchie de modélisation à 4 niveaux

- L'OMG définit 4 niveaux de modélisation
  - M0 : système réel, système modélisé
  - M1 : modèle du système réel défini dans un certain langage
  - M2 : méta-modèle définissant ce langage
  - M3 : méta-méta-modèle définissant le méta-modèle
    - Le niveau M3 est le MOF
    - Dernier niveau, il est méta-circulaire : il peut se définir lui même
- Le MOF est – pour l'OMG – le méta-méta-modèle unique servant de base à la définition de tous les méta-modèles

Hiérarchie de modélisation à 4 niveaux



Hiérarchie de modélisation à 4 niveaux

- Hiérarchie à 4 niveaux existe en dehors du MOF et d'UML, dans d'autres espaces technologiques que celui de l'OMG
  - Langage de programmation
    - M0 : l'exécution d'un programme
    - M1 : le programme
    - M2 : la grammaire du langage dans lequel est écrit le programme
    - M3 : le concept de grammaire EBNF
  - XML
    - M0 : données du système
    - M1 : données modélisées en XML
    - M2 : DTD / Schema XML
    - M3 : le langage XML

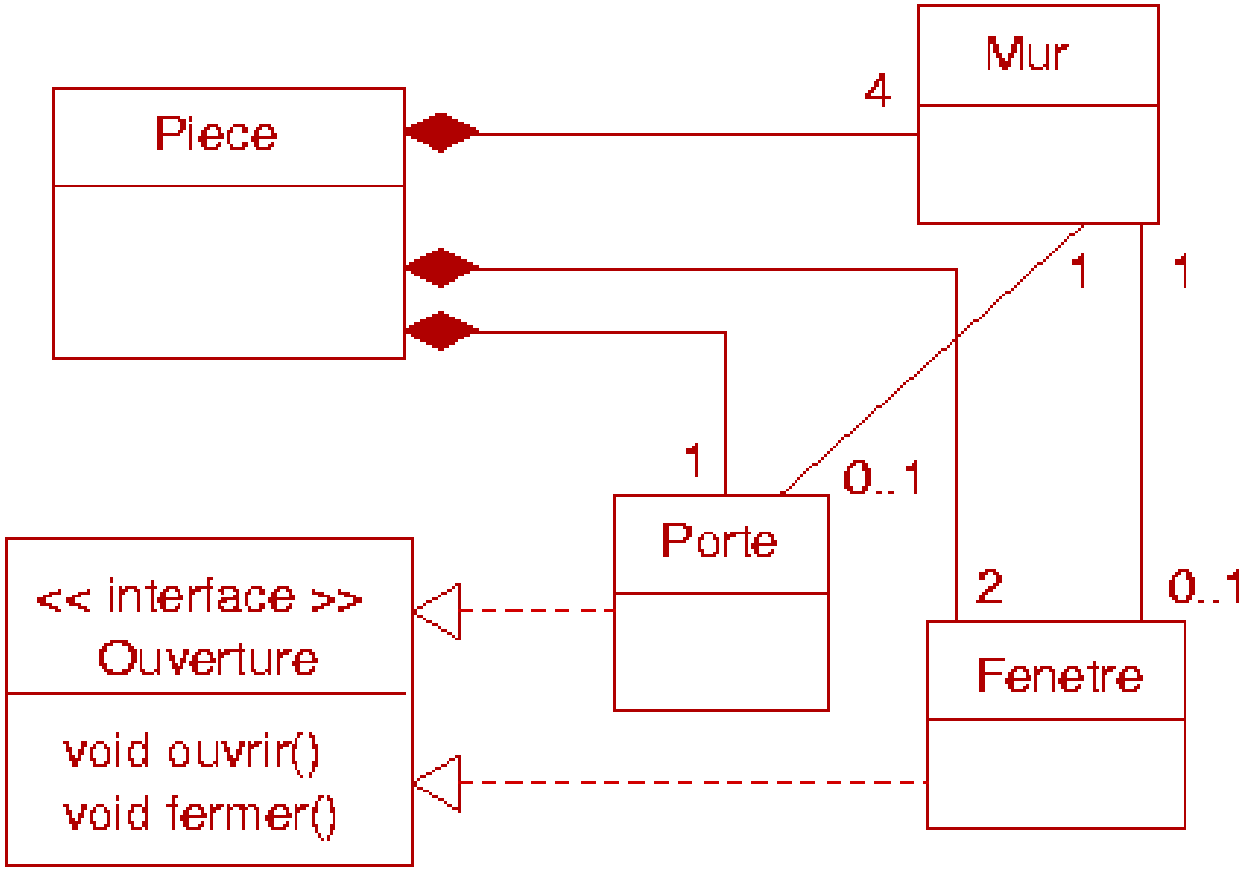
Méta-modélisation UML

- Avec UML, on retrouve également les 4 niveaux
  - Mais avec le niveau M3 définissable en UML directement à la place du MOF
- Exemple de système réel à modéliser (niveau M0)
  - Une pièce possède 4 murs, 2 fenêtres et une porte
  - Un mur possède une porte ou une fenêtre mais pas les 2 à la fois
  - Deux actions sont associées à une porte ou une fenêtre : ouvrir et fermer
  - Si on ouvre une porte ou une fenêtre fermée, elle devient ouverte
  - Si on ferme une porte ou une fenêtre ouverte, elle devient fermée

Méta-modélisation UML

- Pour modéliser ce système, il faut définir 2 diagrammes UML : niveau M1
  - Un diagramme de classe pour représenter les relations entre les éléments (portes, murs, pièce)
  - Un diagramme d'état pour spécifier le comportement d'une porte ou d'une fenêtre (ouverte, fermée)
  - On peut abstraire le comportement des portes et des fenêtres en spécifiant les opérations d'ouverture fermeture dans une interface
    - Le diagramme d'état est associé à cette interface
  - Il faut également ajouter des contraintes OCL pour préciser les contraintes entre les éléments d'une pièce

M1 : spécification du système



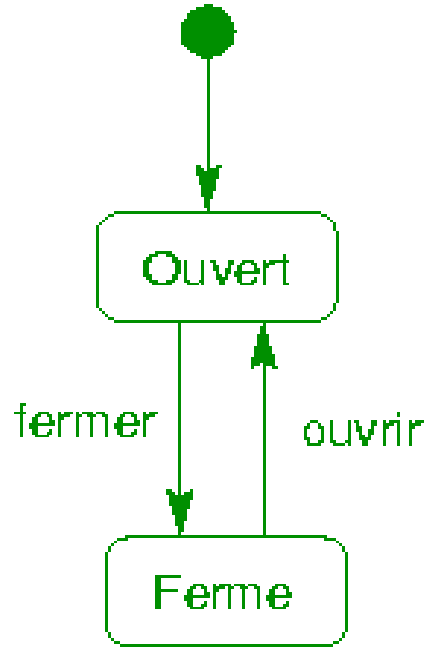


diagramme d'etat associe a l'interface Ouverture

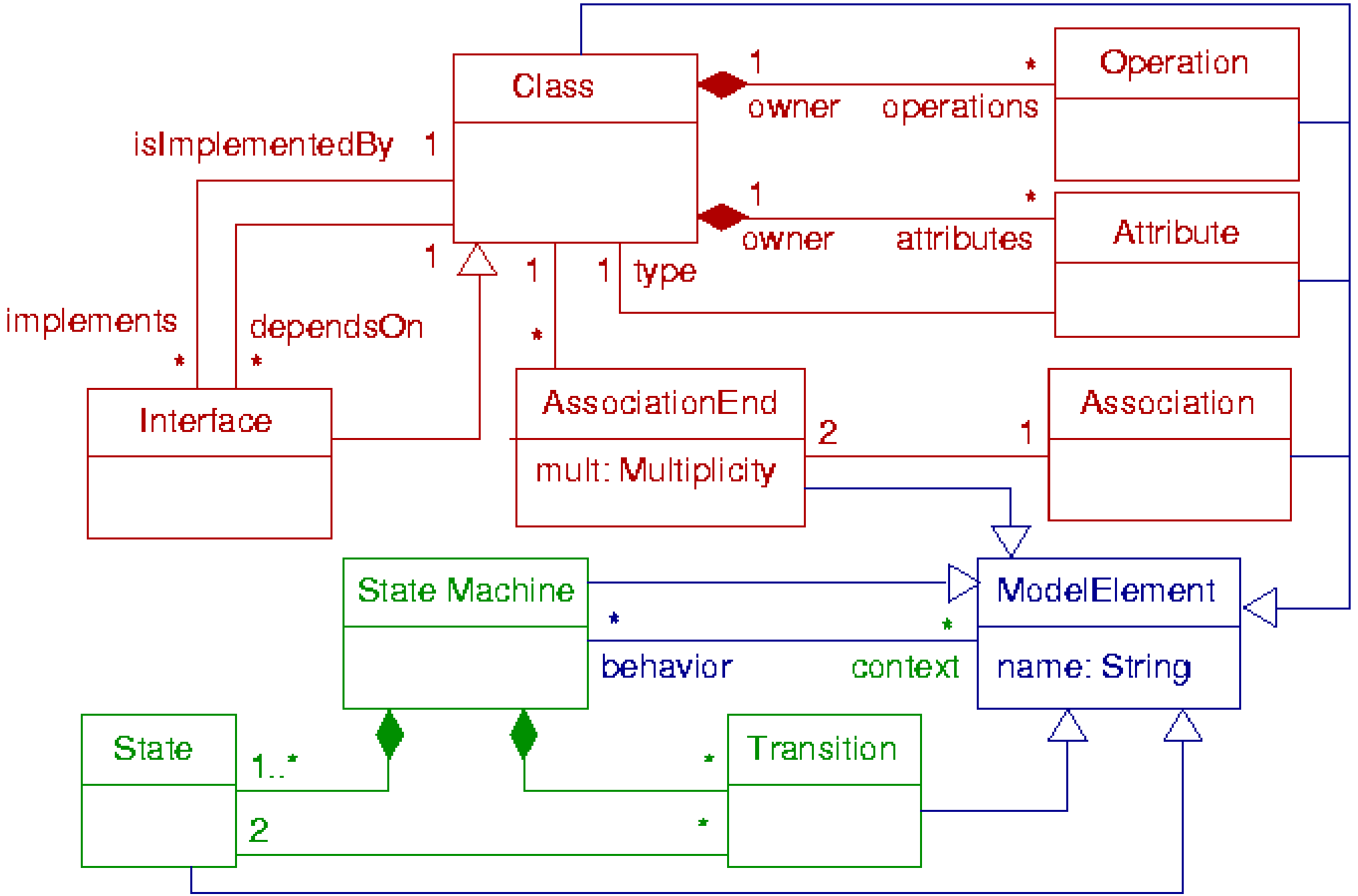
- **context** Mur **inv**: fenetre -> union(porte) -> size() <= 1
  -- un mur a soit une fenêtre soit une porte (soit rien)

- **context** Piece **inv**:
  mur.fenetre -> size() = 2 -- 2 murs de la pièce ont une fenêtre
  mur.porte -> size() = 1 -- 1 mur de la pièce a une porte

Méta-modélisation UML

- Les 2 diagrammes de ce modèle de niveau M1 sont des diagrammes UML valides
- Les contraintes sur les éléments des diagrammes UML et leurs relations sont définies dans le méta-modèle UML : niveau M2
  - Un diagramme UML (de classes, d'états ...) doit être conforme au méta-modèle UML
- Méta-modèle UML
  - Diagramme de classe spécifiant la structure de tous types de diagrammes UML
    - Diagramme de classe car c'est le diagramme UML permettant de définir des éléments/concepts (via des classes) et leurs relations (via des associations)
  - Avec contraintes OCL pour spécification précise

M2 : Méta-modèle UML (très simplifié)



M2 : Méta-modèle UML (très simplifié)

- Contraintes OCL, quelques exemples
  - **context** Interface **inv**: attributes -> isEmpty()

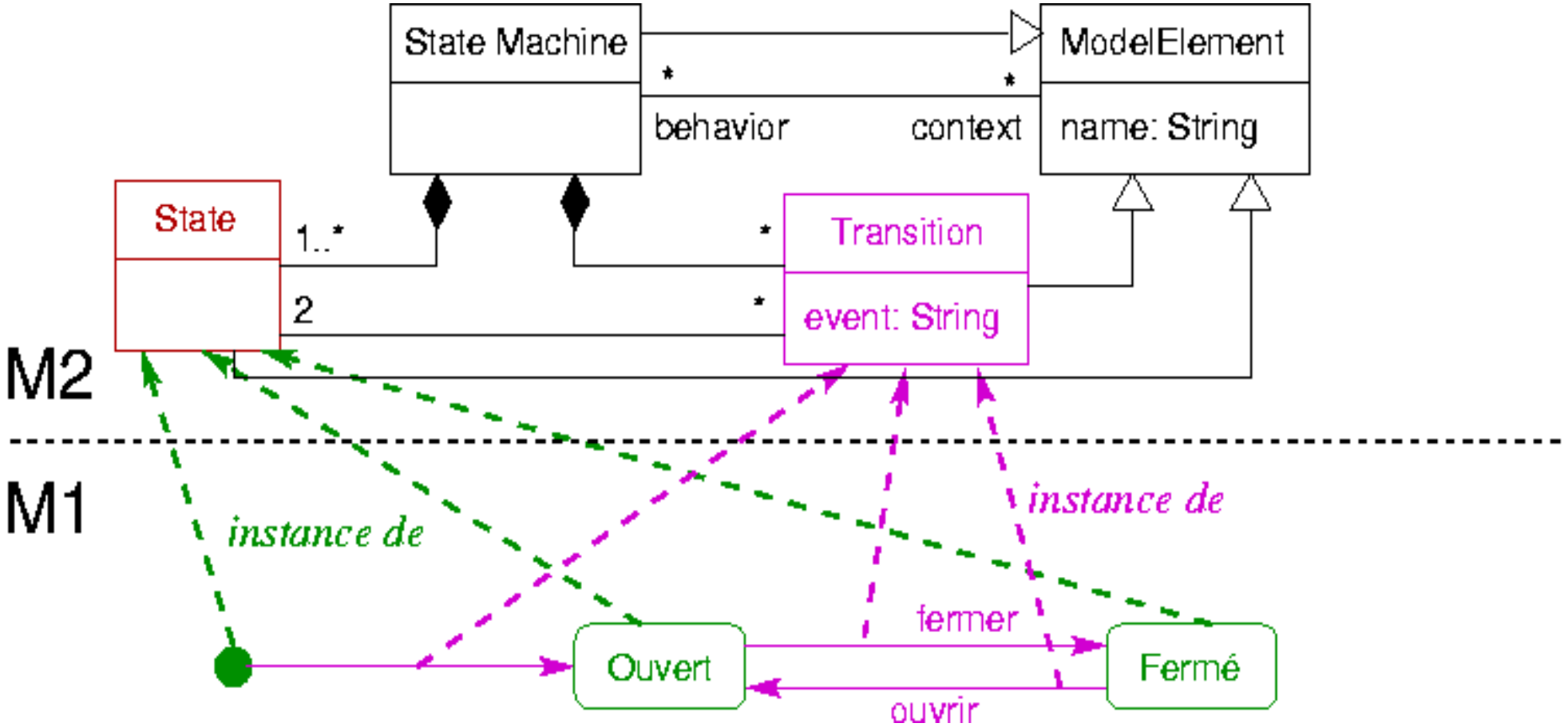
    Une interface est une classe sans attribut
  - **context** Class **inv**: attributes -> forAll (a1, a2 |
    a1 <> a2 **implies** a1.name <> a2.name)

    2 attributs d'une même classe n'ont pas le même nom
  - **context** StateMachine **inv**: transition -> forAll (t |
    self.state -> includesAll(t.state))

    Une transition d'un diagramme d'état connecte 2 états de ce diagramme d'état

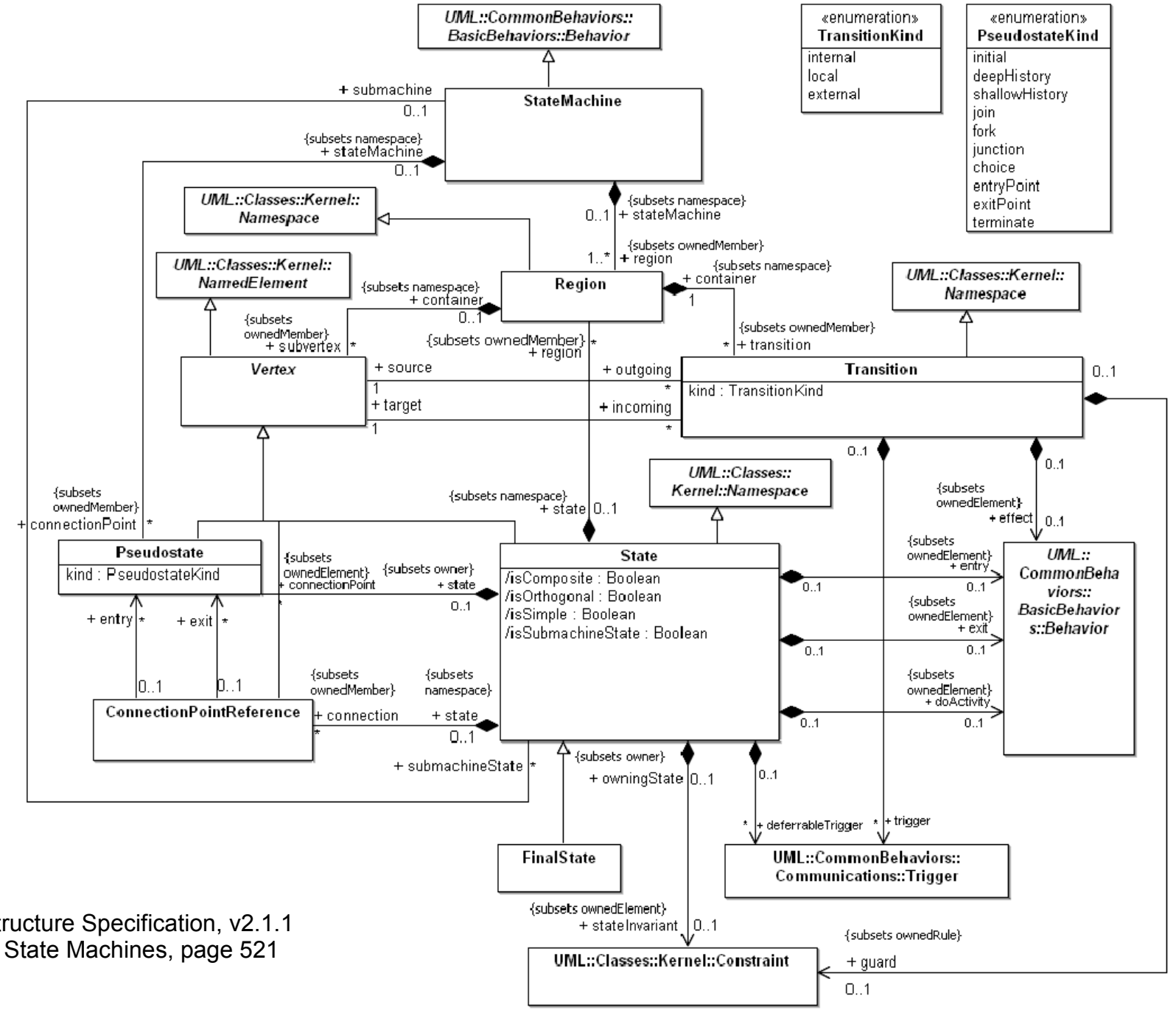
Liens éléments modèle/méta-modèle

- Chaque élément du modèle
  - Est une « instance » d'un élément du méta-modèle (d'un méta-élément)
  - En respectant les contraintes définies dans le méta-modèle
- Exemple avec diagramme état



Extrait méta-modèle UML 2.0

- Partie spécifiant les machines à états



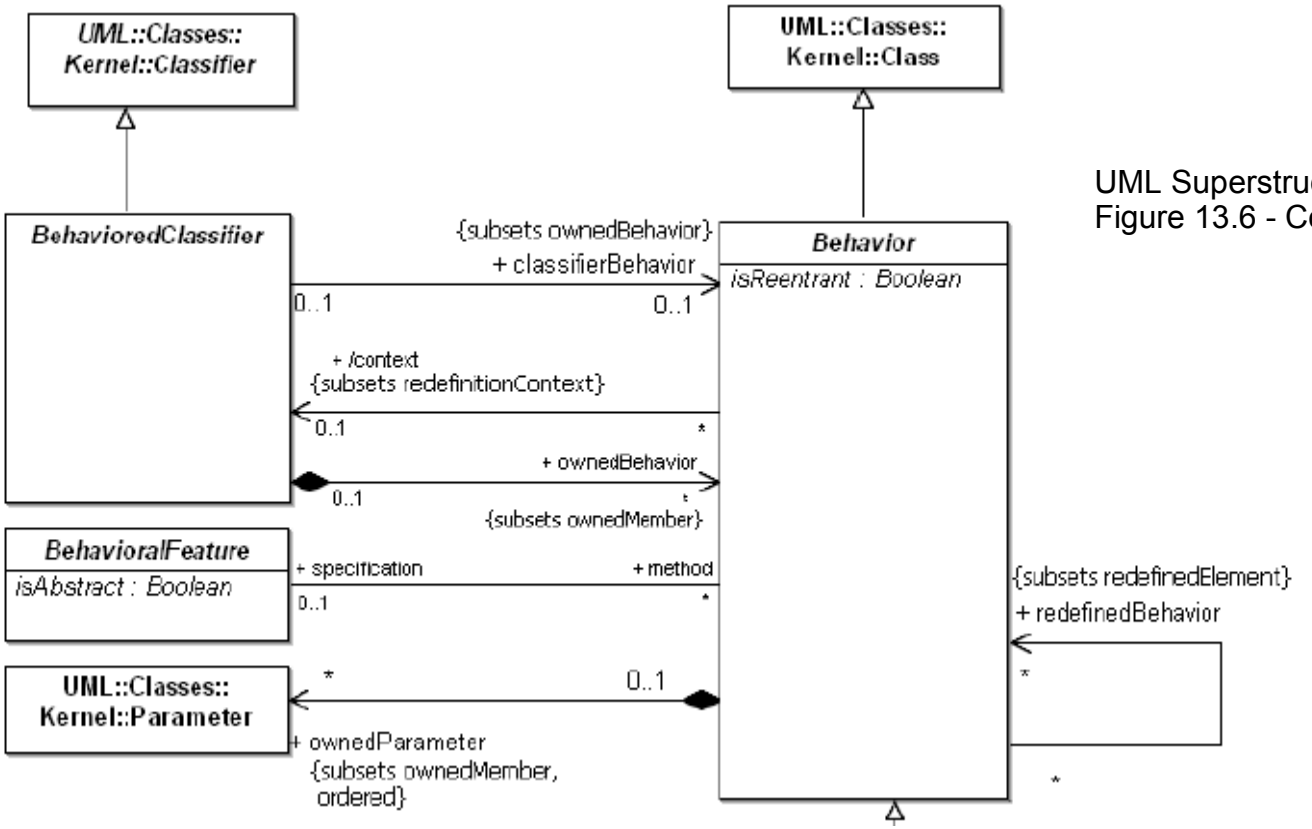
UML Superstructure Specification, v2.1.1
Figure 15.2 - State Machines, page 521

Extrait méta-modèle UML 2.0

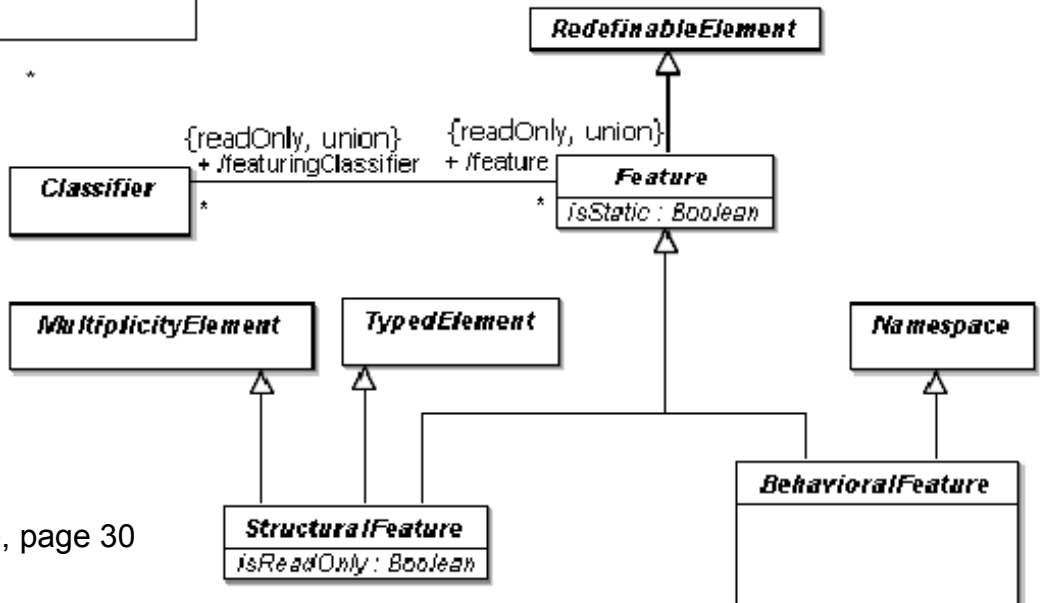
- Exemples de contraintes OCL pour la specification des machines à états
  - Invariants de la classe StateMachine
    - The classifier context of a state machine cannot be an interface.
      - `context->notEmpty() implies not context.oclIsKindOf(Interface)`
    - The context classifier of the method state machine of a behavioral feature must be the classifier that owns the behavioral feature.
      - `specification->notEmpty() implies (context->notEmpty() and specification->featuringClassifier->exists (c | c = context))`
    - The connection points of a state machine are pseudostates of kind entry point or exit point.
      - `connectionPoint->forAll (c | c.kind = #entryPoint or c.kind = #exitPoint)`

Extrait méta-modèle UML 2.0

- Pour comprendre les contraintes OCL



UML Superstructure Specification, v2.1.1
Figure 13.6 - Common Behavior, page 424

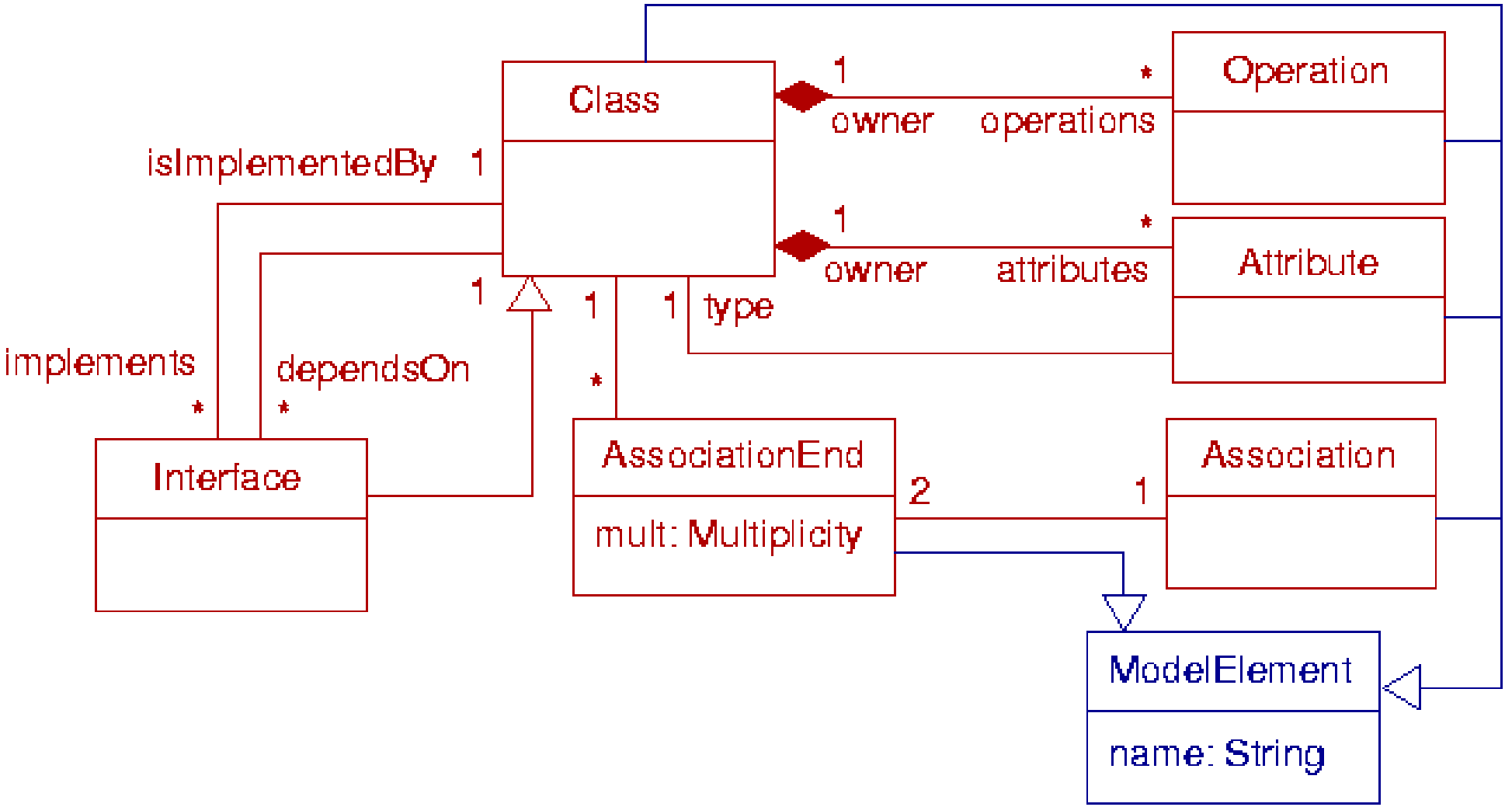


UML Superstructure Specification, v2.1.1
Figure 7.10 - Features diagram of the Kernel package, page 30

Méta-modélisation UML

- Le méta-modèle UML doit aussi être précisément défini
  - Il doit être conforme à un méta-modèle
  - C'est le méta-méta-modèle UML
    - Niveau M3
- Qu'est ce que le méta-modèle UML ?
  - Un diagramme de classe UML (avec contraintes OCL)
- Comment spécifier les contraintes d'un diagramme de classe UML ?
  - Via le méta-modèle UML
  - Ou plus précisément : via la partie du méta-modèle UML spécifiant les diagrammes de classes
- Méta-méta-modèle UML = copie partielle du méta-modèle UML

M3 : Méta-méta-modèle UML (simplifié)



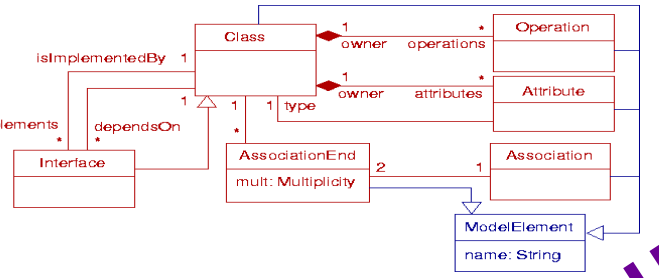
Méta-modélisation UML

- Méta-méta-modèle UML doit aussi être clairement défini
  - Il doit être conforme à un méta-modèle
- Qu'est ce que le méta-méta-modèle UML ?
  - Un diagramme de classe UML
- Comment spécifier les contraintes d'un diagramme de classe ?
  - Via la partie du méta-modèle UML spécifiant les diagrammes de classe
    - Cette partie est en fait le méta-méta-modèle UML
- Le méta-méta-modèle UML peut donc se définir lui même
  - Méta-circulaire
  - Pas besoin de niveau méta supplémentaire

Hiérarchie de modélisation

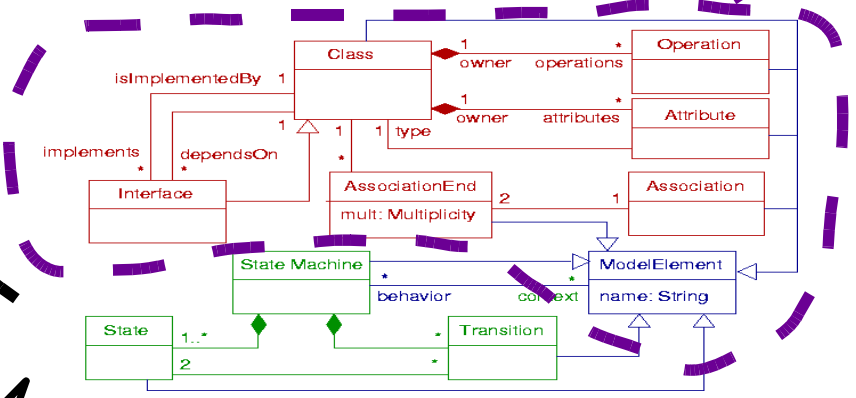
Niveau M3

conforme à



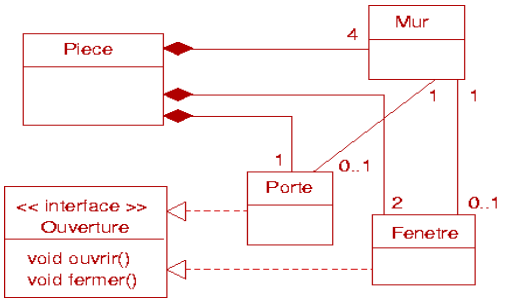
Niveau M2

conforme à



Niveau M1

conforme à



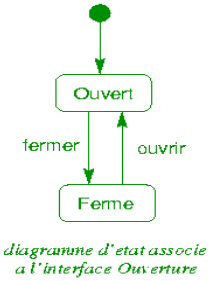


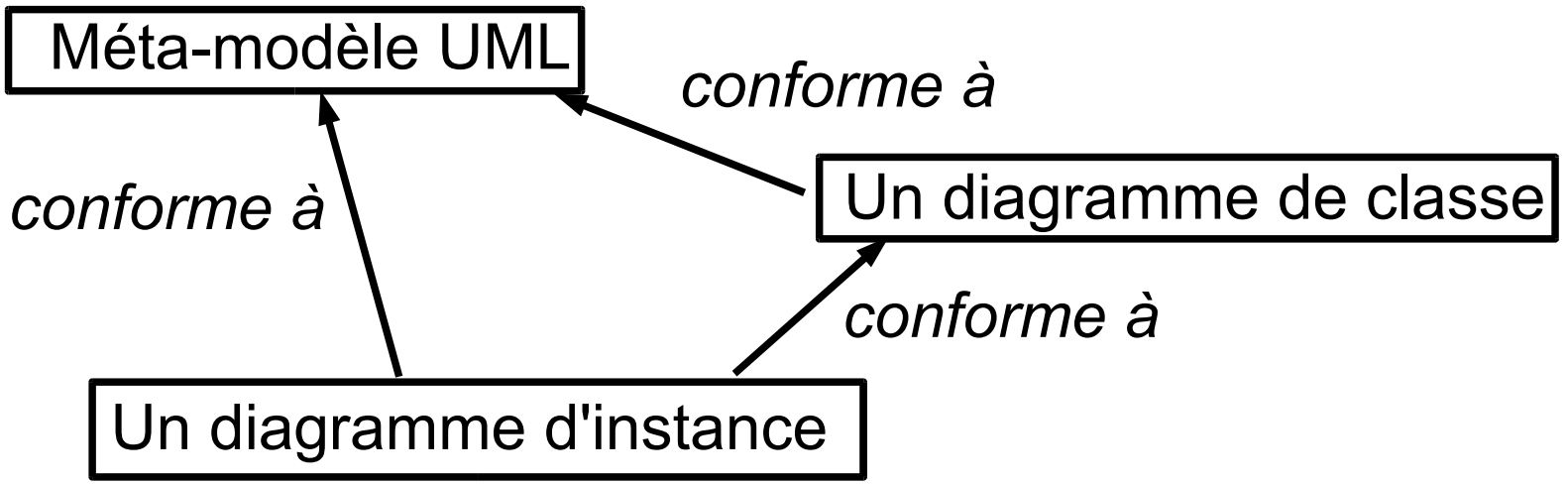
diagramme d'etat associe a l'interface Ouverture

Niveau M0

conforme à

Diagrammes d'instance UML

- Un diagramme d'instance est particulier car
  - Doit être conforme au méta-modèle UML
    - Qui définit la structure générale des diagrammes d'instances
  - Doit aussi être conforme à un diagramme de classe
    - Ce diagramme de classe est un méta-modèle pour le diagramme d'instance
    - Diagramme de classe est conforme également au méta-modèle UML



Hiérarchie de modélisation

- Architecture à 4 niveaux
  - Conceptuellement pertinente
- En pratique
  - Certains niveaux sont difficiles à placer les uns par rapport aux autres
  - Cas du diagramme d'instance
    - Diagramme d'instance de l'application : niveau M1
    - Méta-modèle UML : niveau M2
    - Diagramme de classe de l'application : niveau M1 ou M2 ?
      - M1 normalement car modélise l'application et conforme au méta-modèle de niveau M2
      - Mais devrait être M2 rapport au diagramme d'instance qui est de niveau M1
  - Conclusion
    - Pas toujours facile ni forcément pertinent de chercher à placer absolument les modèles à tel ou tel niveau
    - L'important est de savoir quel rôle (modèle / méta-modèle) joue un modèle dans une relation de conformité

Syntaxe

- Un langage est défini par un méta-modèle
- Un langage possède une syntaxe
  - Définit comment représenter chaque type d'élément d'un modèle
    - Elément d'un modèle = instance d'un méta-élément
- Syntaxe textuelle
  - Ensemble de mots-clé et de mots respectant des contraintes défini selon des règles précises
    - Notions de syntaxe et de grammaire dans les langages
    - Exemple pour langage Java :
      public class MaClasse implements MonInterface { ... }
    - Grammaire Java pour la déclaration d'une classe :
      class_declaration ::= { modifier } "class" identifier
      ["extends" class_name] ["implements" interface_name
      { "," interface_name }] "{" { field_declaration } "}"

Syntaxe

- Syntaxe graphique
  - Notation graphique, chaque type d'élément a une forme graphique particulière
  - Exemple : relations entre classes/interfaces dans les diagrammes de classe UML
    - Trait normal : association
    - Flèche, trait pointillé : dépendance 
    - Flèche en forme de triangle, trait en pointillé : implémentation 
    - Flèche en forme de triangle, trait plein : spécialisation 

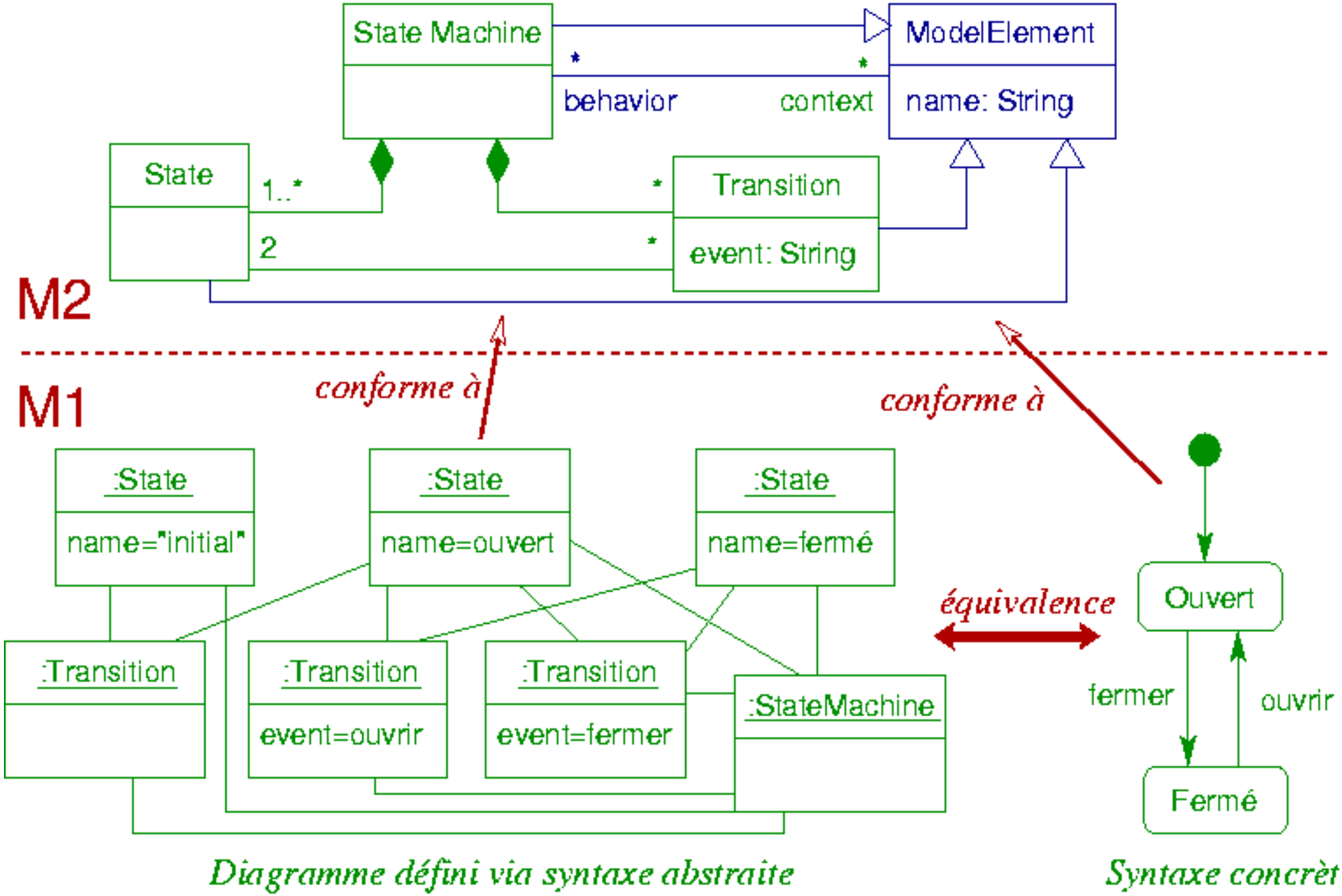
Syntaxe

- Syntaxe abstraite/concrète
  - Abstraite
    - Les éléments et leurs relations sans une notation spécialisée
    - Correspond à ce qui est défini au niveau du méta-modèle, à des instances de méta-éléments
  - Concrète
    - Syntaxe graphique ou textuelle définie pour un type de modèle
    - Plusieurs syntaxes concrètes possibles pour une même syntaxe abstraite
- Un modèle peut être défini via n'importe quelle syntaxe
  - L'abstraite
  - Une des concrètes
- MOF : langage pour définir des méta-modèles
  - Pas de syntaxe concrète définie

Syntaxe

- Exemple de la modélisation de la pièce
  - Syntaxe concrète
    - 2 diagrammes UML (classes et états) avec syntaxes graphiques spécifiques à ces types de diagrammes
  - Via la syntaxe abstraite
    - Diagramme d'instance (conforme au méta-modèle) précisant les instances particulières de classes, d'associations, d'états...
  - Pour la partie diagramme d'états
    - Diagramme défini via syntaxe concrète : diagramme d'états de l'exemple
    - Diagramme défini via syntaxe abstraite : diagramme d'instances conforme au méta-modèle UML

Syntaxe : exemple diagramme état



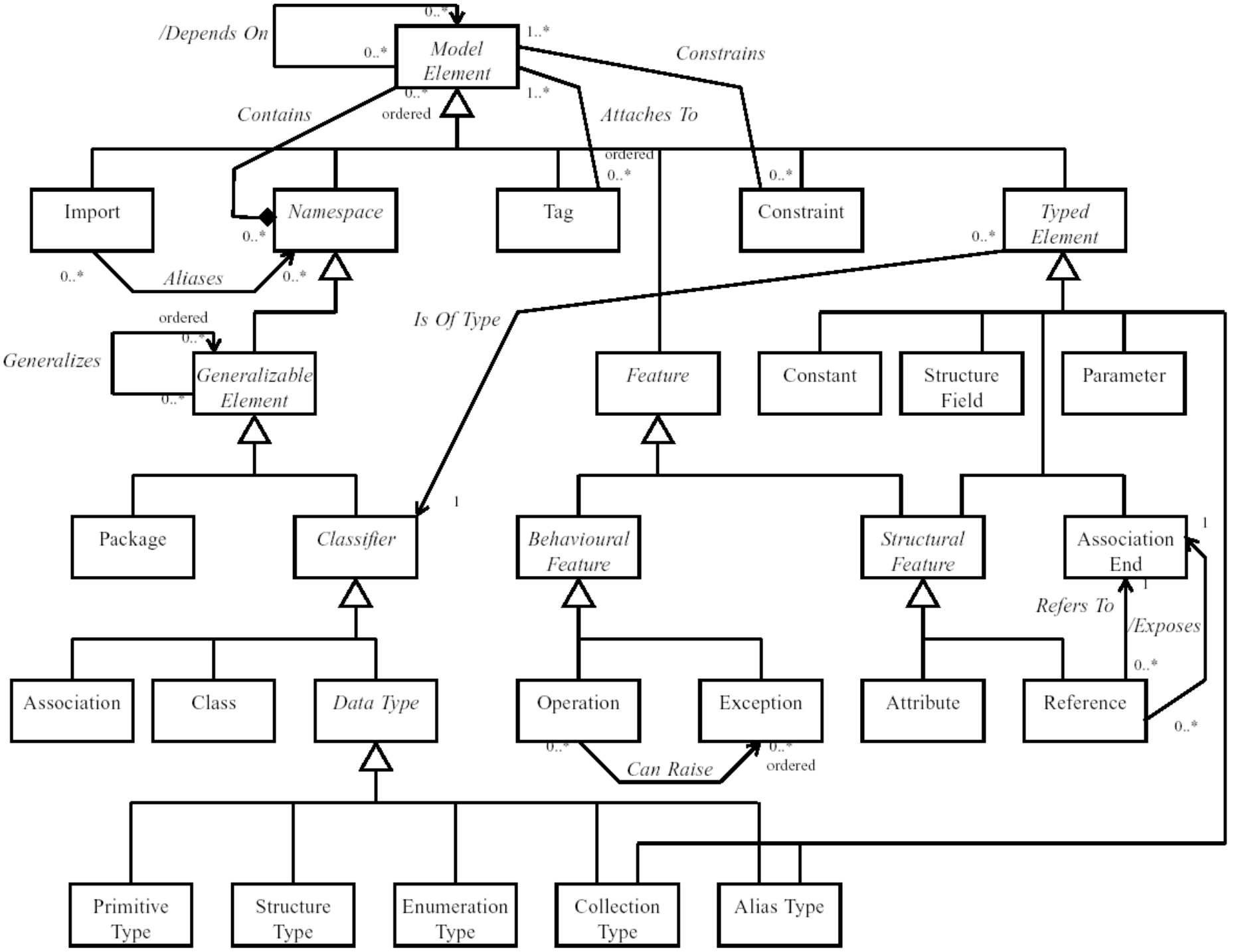
Définition de méta-modèles

- But : définir un type de modèle avec tous ses types d'éléments et leurs contraintes de relation
- Plusieurs approches possibles
  - Définir un méta-modèle nouveau à partir de « rien », sans base de départ
    - On se basera alors sur un méta-méta-modèle existant comme MOF ou Ecore
  - Modifier un méta-modèle existant : ajout, suppression, modification d'éléments et des contraintes sur leurs relations
  - Spécialiser un méta-modèle existant en rajoutant des éléments et des contraintes (sans en enlever)
    - Correspond aux profils UML

MOF

- Meta Object Facilities (MOF) (version 2.0)
  - Méta-méta-modèle de référence pour l'OMG
  - Pas de syntaxe concrète
  - Aligné depuis la version 2.0 avec le « noyau » du méta-modèle d'UML 2.0
  - Décomposé en 2 parties
    - E-MOF : essential MOF
      - Méta-modèle noyau
    - C-MOF : complete MOF
      - Méta-modèle plus complet
    - E-MOF et C-MOF peuvent se définir mutuellement et chacun eux-mêmes
  - Définition d'un méta-modèle via le MOF
    - Méta-modèle = ensemble d'instances de méta-éléments du MOF associées entre elles

MOF : méta-modèle (version 1.4)

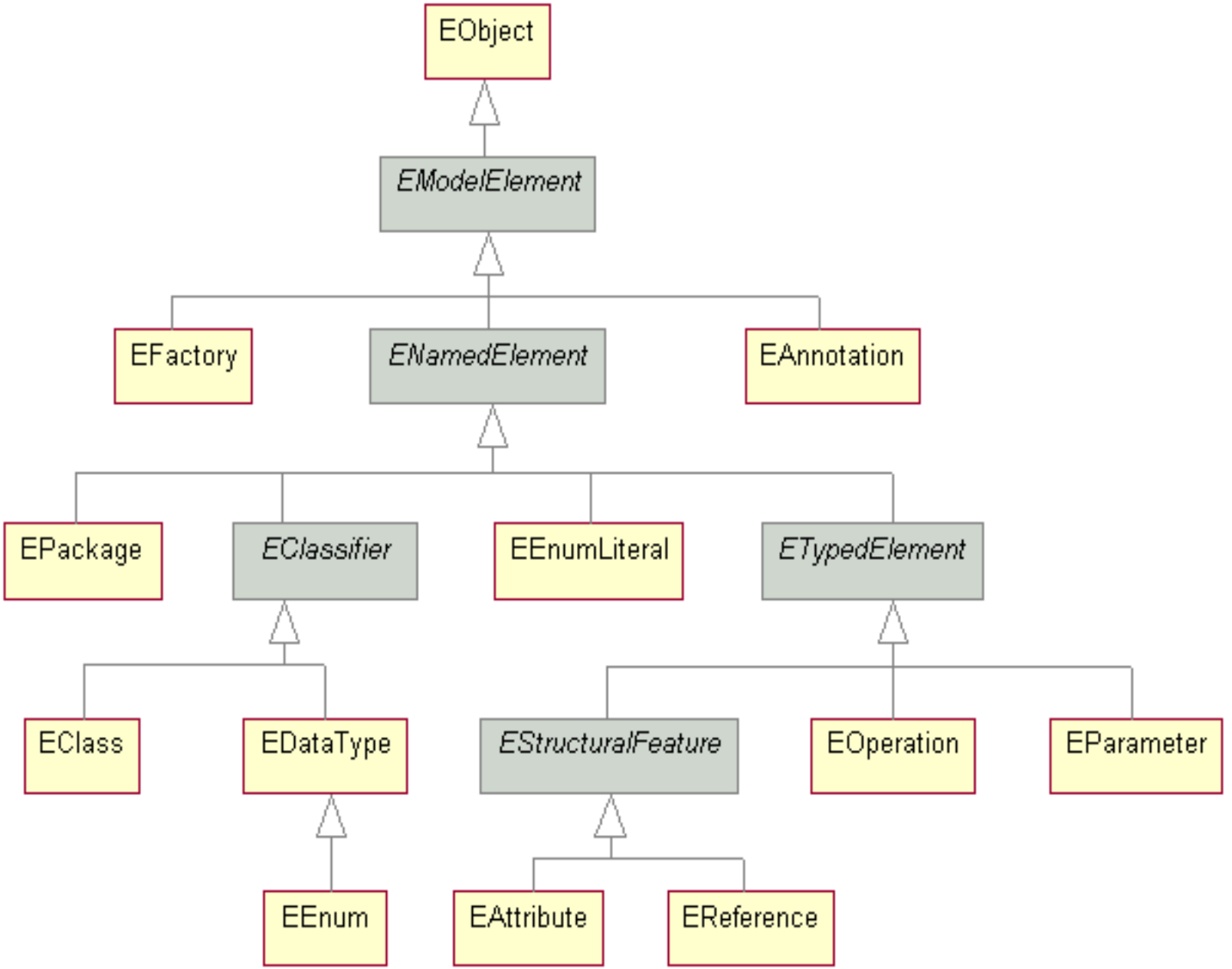


Ecore

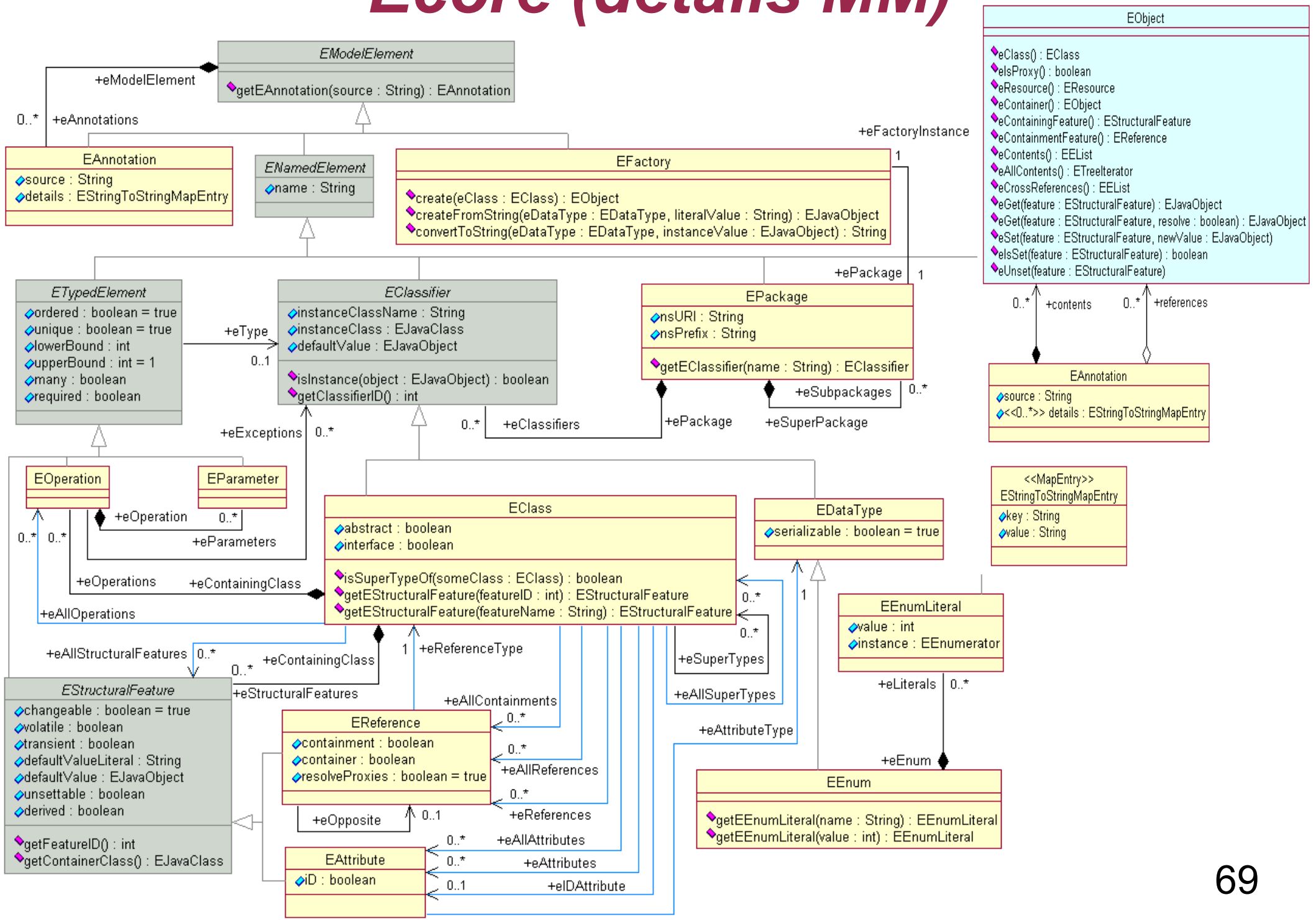
- Eclipse Modeling Framework (EMF)
  - Framework de modélisation intégré dans l'atelier de développement Eclipse (IBM) pour le langage Java
  - But : modéliser des programmes Java et travailler au niveau modèle en plus du code
  - Mais peut aussi servir à créer des modèles et des méta-modèles
- Ecore
  - Méta-modèle intégré dans EMF
  - Méta-modèle « minimal » aligné sur E-MOF

Ecore

- Eléments du méta-modèle Ecore



Ecore (détails MM)



Transformations de modèles

Transformations

- Une transformation est une opération qui
  - Prend en entrée des modèles (source) et fournit en sortie des modèles (cibles)
    - Généralement un seul modèle source et un seul modèle cible
- Transformation endogène
  - Dans le même espace technologique
  - Les modèles source et cible sont conformes au même méta-modèle
    - Ex : transformation d'un modèle UML en un autre modèle UML
- Transformation exogène
  - Entre 2 espaces technologique différents
  - Les modèles source et cible sont conformes à des méta-modèles différents
    - Ex : transformation d'un modèle UML en programme Java
    - Ex : transformation d'un fichier XML en schéma de BDD

Autre vision des transformations

- Depuis longtemps on utilise des processus de développement automatisé et basé sur les transformations
  - Rien de totalement nouveau
    - Adaptation à un nouveau contexte
  - Exemple : compilation d'un programme C
    - Programme C : modèle abstrait
    - Transformation de ce programme sous une autre forme mais en restant à un niveau abstrait
      - Modélisation, représentation différente du programme C pour le manipuler : transformation en un modèle équivalent
      - Exemple : arbres décorés
    - Génération du code en langage machine
      - Avec optimisation pour une architecture de processeur donnée

Outils pour réaliser des transformations

- Outils de mise en oeuvre
  - Exécution de transformations de modèles
    - Nécessité d'un langage de transformation
    - Qui pourra être défini via un méta-modèle de transformation
  - Les modèles doivent être manipulés, créés et enregistrés
    - Via un *repository* (dépôt, référentiel)
    - Doit pouvoir représenter la structure des modèles
      - Via des méta-modèles qui devront aussi être manipulés via les outils
      - On les stockera également dans un repository
- Il existe de nombreux outils ou qui sont en cours de développement (industriels et académiques)
  - Notamment plusieurs moteurs/langages de transformation

Transformations : types d'outils

- Langage de programmation « standard »
  - Ex : Java
  - Pas forcément adapté pour tout
  - Sauf si interfaces spécifiques
    - Ex : JMI (Java Metadata Interface) ou framework Eclipse/EMF
- Langage dédié d'un atelier de génie logiciel
  - Ex : J dans Objecteering
  - Souvent propriétaire et inutilisable en dehors de l'AGL
- Langage lié à un domaine/espace technologique
  - Ex: XSLT dans le domaine XML, AWK pour fichiers texte ...
- Langage/outil dédié à la transformation de modèles
  - Ex : standard QVT de l'OMG, ATL
- Atelier de méta-modélisation avec langage d'action
  - Ex : Kermeta

Transformations : types d'outils

- 3 grandes familles de modèles et outils associés
  - Données sous forme de séquence
    - Ex : fichiers textes (AWK)
  - Données sous forme d'arbre
    - Ex : XML (XSLT)
  - Données sous forme de graphe
    - Ex : diagrammes UML
    - Outils
      - Transformateurs de graphes déjà existants
      - Nouveaux outils du MDE et des AGL (QVT, J, ATL, Kermeta ...)

Techniques de transformations

- 3 grandes catégories de techniques de transformation
  - Approche déclarative
    - Recherche de certains patrons (d'éléments et de leurs relations) dans le modèle source
    - Chaque patron trouvé est remplacé dans le modèle cible par une nouvelle structure d'élément
    - Ecriture de la transformation « assez » simple mais ne permet pas toujours d'exprimer toutes les transformations facilement
  - Approche impérative
    - Proche des langages de programmation usuels
    - On parcourt le modèle source dans un certain ordre et on génère le modèle cible lors de ce parcours
    - Ecriture transformation peut être plus lourde mais permet de toutes les définir, notamment les cas algorithmitiquement complexes
  - Approche hybride : à la fois déclarative et impérative
    - La plupart des approches déclaratives offrent de l'impératif en complément car plus adapté dans certains cas

Repository

- Référentiel pour stocker modèles et méta-modèles
  - Les (méta)modèles sont stockés selon le formalisme de l'outil
    - XML par exemple pour les modèles
    - Et DTD/Schema XML pour les méta-modèles
  - Forme de stockage d'un modèle
    - Modèle est formé d'instances de méta-éléments, via la syntaxe abstraite
      - Stocke ces éléments et leurs relations
    - L'affichage du modèle via une notation graphique est faite par l'AGL
  - Les référentiels peuvent être notamment basés sur
    - XML
      - XMI : norme de l'OMG pour représentation modèle et méta-modèle
    - Base de données relationnelle
    - Codage direct dans un langage de programmation

Transformations en EMF

- Plusieurs langages pour le framework EMF/Ecore
  - Directement via le framework Java EMF
    - Chaque élément du méta-modèle Ecore correspond à une classe Java avec ses interfaces d'accès aux éléments de modèles
    - Pb : reste de la progammation en Java, pas totalement adapté à la manipulation de modèles
- Langages de plus haut niveau, dont ceux utilisés en TP
  - ATL : approche déclarative
    - Une règle de transformation sélectionne un élément dans le source et génère un (ou plusieurs) élément(s) dans le cible
    - La sélection se fait par des query OCL
  - Kermeta : approche impérative
    - Langage de programmation orienté-objet
    - Primitives dédiées pour la manipulation de modèles
    - Ajout d'opérations/attributs sur méta-éléments par aspect
    - Intègre aussi un langage de contraintes similaire à OCL
      - Invariant sur méta-élément et pré/post sur leurs opérations

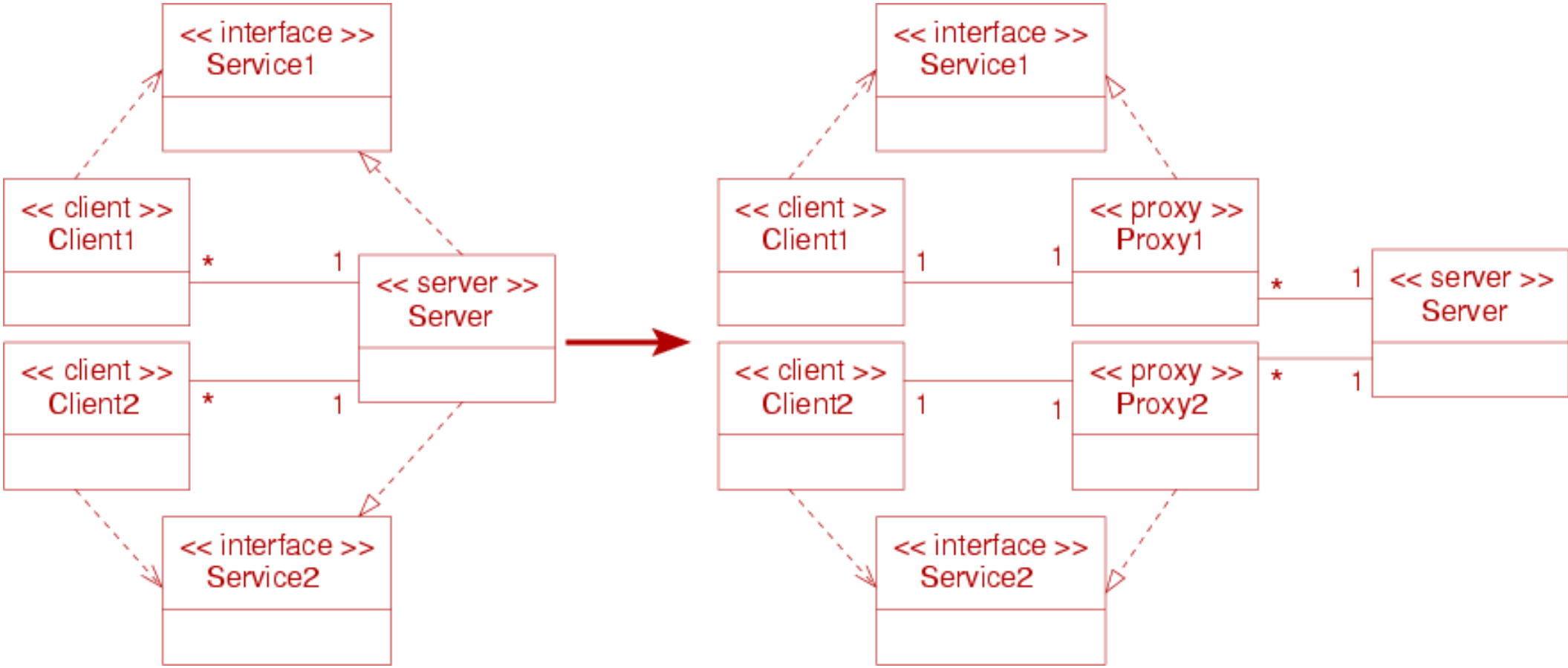
Transformation : exemple de l'ajout de proxy

Ajout de proxy

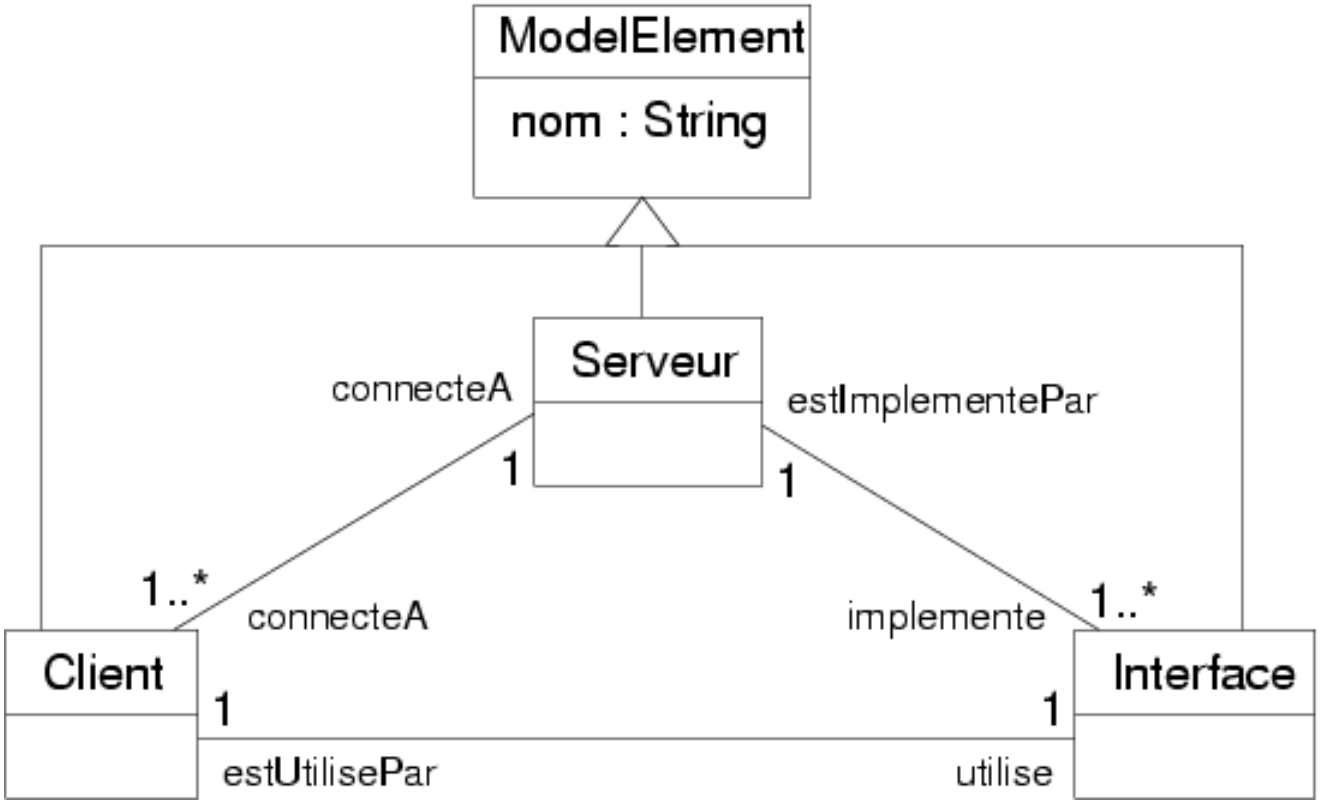
- Modélisation très simplifiée d'architecture logicielle
  - Modèles source : clients et serveurs
    - Un type de client est connecté à un serveur
    - Chaque type de client requière une interface qui est implémentée par le serveur
  - Modèles cible : clients, serveurs et proxy
    - Entre un type de client et le serveur, on rajoute un proxy dédié
    - Le client est maintenant connecté à son proxy
    - Le proxy implémente l'interface de services requise par le client
  - Méta-modèles source et cible
    - Proches mais différents : transformation exogène
    - Contraintes OCL dont une qui exprime qu'il n'y a un qu'un seul serveur par modèle

Ajout de proxy : exemple

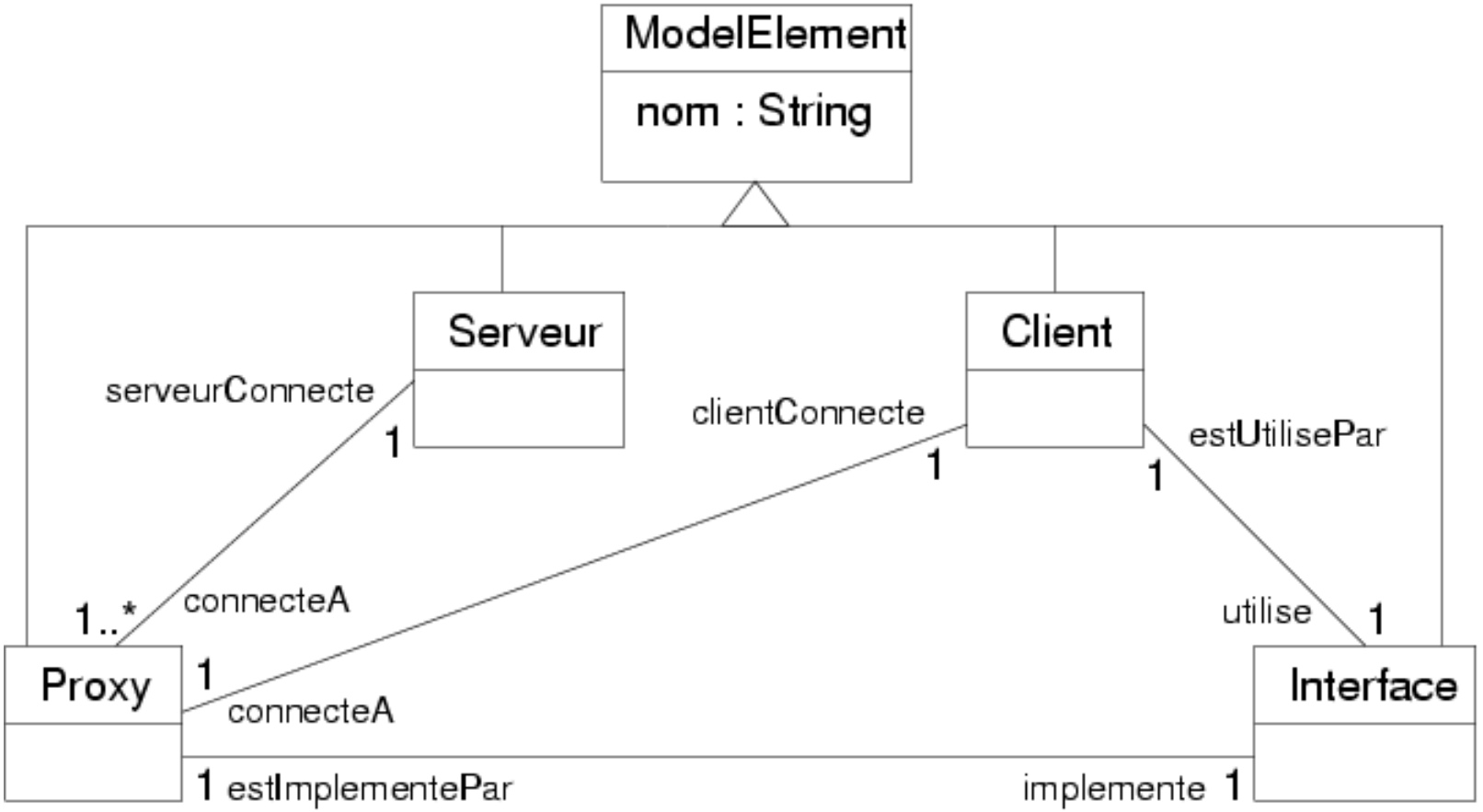
- Exemple de transformation
  - En utilisant un diagramme de classes UML avec stéréotypes



Ajout de proxy : MM source



Ajout de proxy : MM cible



Ajout de proxy : transfo ATL

- Duplication du serveur
  - On lui associe tous les proxys qui existent coté cible
  - ```
    rule LeServeur {
    from
          serveurSource : ClientServeur!Serveur
    to
          serveurCible : ClientProxyServeur!Serveur (
                nom <- serveurSource.nom,
                connecteA <- ClientProxyServeur!Proxy.allInstances()
          )
    }
    ```
- Duplication des interfaces
  - On associe une interface au proxy qui a été associé à son client
  - ```
    rule Interfaces {
    from
          interSource : ClientServeur!Interface
    to
          interCible : ClientProxyServeur!Interface (
                nom <- interSource.nom,
                estUtiliseePar <- interSource.estUtiliseePar,
                estImplementeePar <-
                      (ClientProxyServeur!Proxy.allInstances() -> any (p |
                                  p.implemente = interCible)) )
    }
    ```

Ajout de proxy : transfo ATL

- Duplication des clients
  - On rajoute en plus un proxy associé à chaque client
  - Nom du proxy : concaténation des noms du client et du serveur
  - ```
    rule Clients {
    from
          clientSource : ClientServeur!Client
    to
          clientCible : ClientProxyServeur!Client (
                nom <- clientSource.nom,
                utilise <- clientSource.utilise,
                connecteA <- proxy
          ),
          proxy : ClientProxyServeur!Proxy (
                nom <- clientSource.nom +
                      ClientServeur!Serveur.allInstances().first().nom,
                clientConnecte <- clientCible,
                serveurConnecte <-
                       ClientProxyServeur!Serveur.allInstances().first(),
                implemente <- clientCible.utilise
          )
    }
    ```
    ```

Ajout de proxy : transfo Kermeta

```
operation transformerModele(serveur : ClientServeur::Serveur) :
                                                    ClientProxyServeur::Serveur is do
    // on cree un serveur cible avec le même nom
    var serveurCible : ClientProxyServeur::Serveur init
                                                    ClientProxyServeur::Serveur.new
    serveurCible.nom := serveur.nom

    // pour chaque client auquel le serveur est connecté, on crèe
    // une interface, un client et un proxy cible avec les bonnes associations
    serveur.connecteA.each { client |
        var clientCible : ClientProxyServeur::Client init
                                                    ClientProxyServeur::Client.new
        var interfaceCible : ClientProxyServeur::Interface init
                                                    ClientProxyServeur::Interface.new
        var proxy : ClientProxyServeur::Proxy init ClientProxyServeur::Proxy.new

        clientCible.nom := client.nom
        interfaceCible.nom := client.utilise.nom
        proxy.nom := client.nom + serveur.nom

        clientCible.connecteA := proxy
        clientCible.utilise := interfaceCible

        interfaceCible.estImplementeePar := proxy
        interfaceCible.estUtiliseePar := clientCible

        proxy.implemente := interfaceCible
        proxy.clientConnecte := clientCible
        proxy.serveurConnecte := serveurCible

    serveurCible.connecteA.add(proxy)
    }
    result := serveurCible
end
```

Ajout de proxy : comparaison

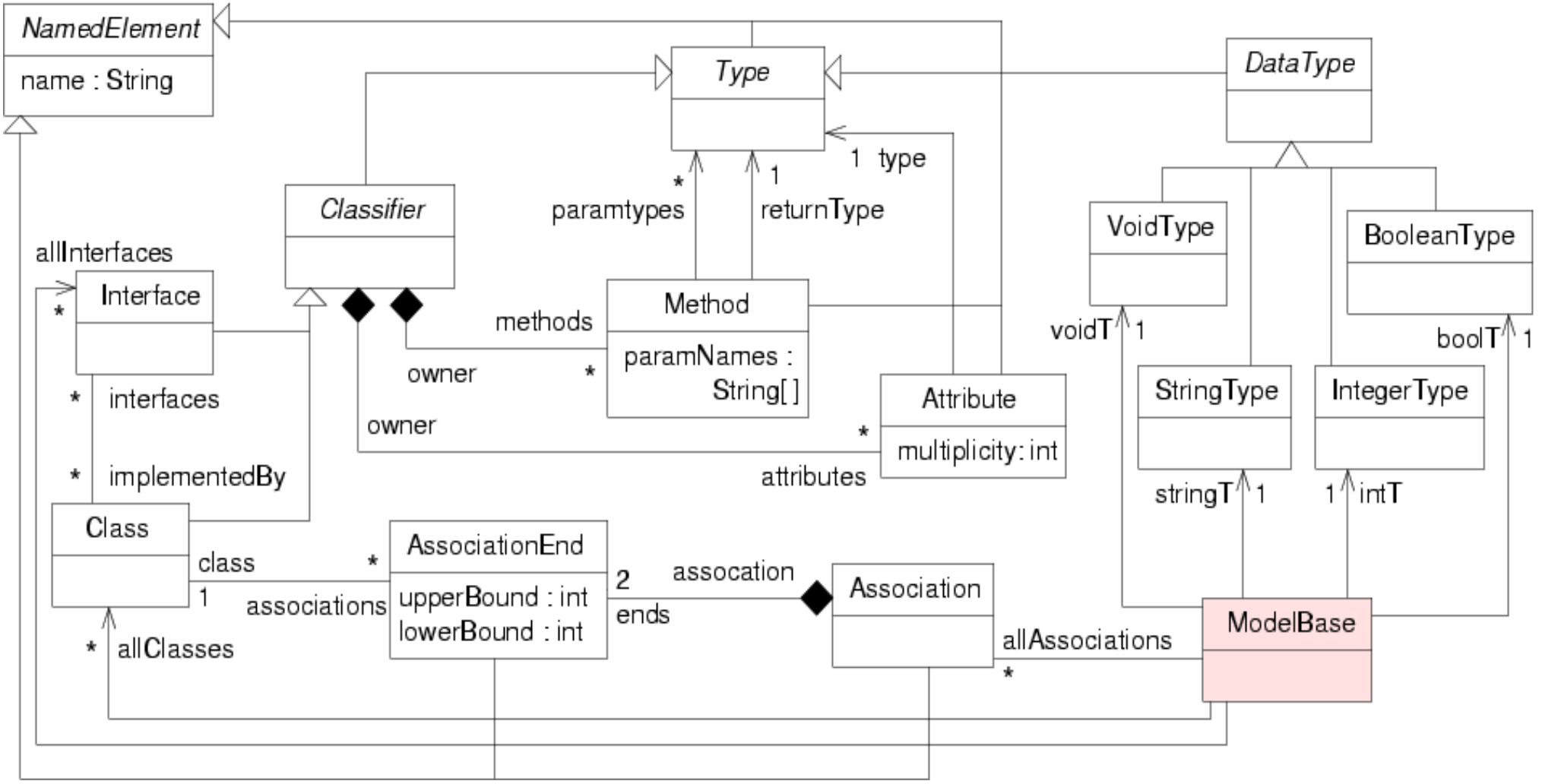
- ATL
  - Purement déclaratif
  - A un type d'élément source, on associe un (ou deux) type(s) d'élément cible
    - Ne se préoccupe pas de savoir comment les éléments sont retrouvés dans le modèle
    - Accède à des éléments crées dans d'autres règles sans de préoccuper de l'ordre de leurs créations
- Kermeta
  - Approche impérative
  - Doit explicitement naviguer sur les modèles
    - Sur le source pour récupérer les éléments à dupliquer avec modifications requises
    - Sur le cible, pour en créer les éléments un par un

Transformation : exemple de privatisation d'attributs

Privatisation d'attributs

- Pour un diagramme de classe, passer les attributs d'une classe en privé et leur associer un « getter » et un « setter »
- Deux transformations étudiées
  - Transformation en Kermeta, style impératif
  - Transformation en ATL, style déclaratif
- Utilisation d'un méta-modèle simplifié de diagramme de classe
  - Méta-modèle UML trop complexe pour cet exemple
- Transformation endogène
  - Modèle source et cible sont conformes au même méta-modèle
  - Concrètement, c'est un raffinement
    - C'est le même modèle mais avec plus de détails (les getter et setter pour les attributs)

MM simplifié de diagramme de classe



- Note: les visibilités ne sont pas définies, on ne les gérera donc pas pendant la transformation

MM simplifié de diagramme de classe

- Contraintes OCL pour compléter le méta-modèle
  - Unicité des noms de type

```
context Type inv uniqueTypeNames:
Type.allInstances() -> forAll (t1, t2 |
          t1 <> t2 implies t1.name <> t2.name)
```

  - Une interface n'a pas d'attributs

```
context Interface inv noAttributesInInterface:
attributes -> isEmpty()
```

  - Une méthode à une liste de noms de paramètres et une liste de types de paramètres de même taille

```
context Method inv sameSizeParamsAndTypes:
paramNames -> size() = paramTypes -> size()
```

  - ...

Règles générales de la transformation

- Pour un attribut *att* de type *type*, forme des accesseurs
  - Getter : *type getAtt()*
  - Setter : *void setAtt(type value)*
    - Le nom du paramètre est quelconque, on choisit value systématiquement
- Règles de transformations
  - Pour chaque attribut de chaque classe
    - On ajoute, s'ils n'existaient pas déjà, un setter et un getter dans la classe qui possède l'attribut
  - Doit donc prévoir des fonctions de vérification de la présence d'un getter ou d'un setter

Transformation en Kermeta

- Vérification de la présence d'un getter

```
operation hasGetter(att : Attribute) : Boolean is do
  result := att.owner.methods.exists { m |
     if (m.name.size <=3)
     then
       false
     else
       m.name.substring(0,3).equals("get") and
       m.name.substring(3,4).equals(att.name.substring(0,1).toUpperCase) and
       m.name.substring(4,m.name.size).equals(
                                        att.name.substring(1,att.name.size)) and
       m.returnType == att.type and
       m.paramNames.empty() and
       m.paramTypes.empty()
     end
  }
end
```

- Pour un attribut, on récupère les méthodes de sa classe (att.owner.methods) et on vérifie qu'il existe un getter
  - Nom « getAtt » (vérifie taille chaine > 3 sinon les substring plantent)
  - Même type de retour que l'attribut
  - Liste de paramètres vide

Transformation en Kermeta

- Vérification de la présence d'un setter

```
operation hasSetter(att : Attribute) : Boolean is do
  result := att.owner.methods.exists { m |
     if (m.name.size <=3)
     then
       false
     else
       m.name.substring(0,3).equals("set") and
       m.name.substring(3,4).equals(att.name.substring(0,1).toUpperCase) and
       m.name.substring(4,m.name.size).equals(
                                        att.name.substring(1,att.name.size)) and
       m.returnType.name.equals("void") and
       m.paramNames.size() == 1 and
       m.paramTypes.includes(att.type)
     end
  }
end
```

- Vérification d'un setter
  - Nom « setAtt »
  - Un seul paramètre, du même type que l'attribut
  - Type de retour est « void »

Transformation en Kermeta

```
operation addAccessors(base : ModelBase): ModelBase is do
  base.allClasses.each { cl |
    cl.attributes.each { att |
      var met : Method

      if not(hasGetter(att)) then
         met := Method.new
         met.name := "get"+ att.name.substring(0,1).toUpperCase +
                                              att.name.substring(1,att.name.size)
         met.returnType := att.type
         met.owner := att.owner
         cl.methods.add(met)
      end

      if not(hasSetter(att)) then
        met := Method.new
        met.name := "set" + att.name.substring(0,1).toUpperCase +
                                              att.name.substring(1,att.name.size)
        met.returnType := base.voidT
        met.paramNames.add("value")
        met.paramTypes.add(att.type)
        met.owner := att.owner
        cl.methods.add(met)
      end
    }
  }
  result := base
end
```

Transformation en Kermeta

- Réalisation de la transformation
  - A partir de la base du modèle, on parcourt l'ensemble des classes et pour chacun de leur attribut, s'il ne possède pas un getter ou un setter
    - Crée la méthode en instanciant le méta-élément Method
    - Positionne son nom en « setAtt » ou « getAtt »
    - Positionne son type de retour
      - Référence sur le type void (voidT) ou le type de l'attribut
    - Positionne les listes des paramètres pour un getter à partir du type de l'attribut et le nom « value »
      - Sinon reste vide par défaut (cas d'un setter)
    - Positionne les associations entre la méthode créée et la classe qui possède l'attribut
  - Retourne ensuite la base du modèle modifiée

Transformation en ATL

- Pour vérification des présences des getter et setter, utilise des helpers écrits en OCL (légère diff de syntaxe)

```
helper context CDSource!Attribute def: hasGetter() : Boolean =
self.owner.methods -> exists ( m |
   m.name = 'get' + self.name.firstToUpper() and
   m.paramNames -> isEmpty() and
   m.paramTypes -> isEmpty() and
   m.returnType = self.type
);
```

```
helper context CDSource!Attribute def: hasSetter() : Boolean =
self.owner.methods -> exists ( m |
   m.name = 'set' + self.name.firstToUpper() and
   m.paramNames -> size() = 1 and
   m.paramTypes -> includes( self.type ) and
   m.returnType = thisModule.voidType
);
```

- Fonction pour gérer le premier caractère d'une chaine en majuscule

```
helper context String def: firstToUpper() : String =
self.substring(1, 1).toUpper() + self.substring(2, self.size());
```

- Référence sur le type void

```
helper def: voidType : CDSource!VoidType =
CDSource!VoidType.allInstances() -> any(true);
```

Transformation en ATL

- Transformation ATL qui effectue le raffinement
  - Duplication à l'identitique de tous les éléments du modèle, à l'exception des attributs car doit gérer les getter/setter
    - Une règle par type d'élément
  - Quatre règles de transformation pour attributs
    - Pour chaque attribut, on a des régles qui créent l'attribut coté cible et les éventuels méthodes accesseurs manquantes
    - Selon qu'il possède déjà un getter ou un setter, 4 cas différents
      - Possède un getter et un setter (règle « hasAll »)
        - Duplique l'attribut à l'identique, sans plus
      - Possède un setter mais pas un getter (règle « hasSetter »)
        - Rajoute un getter en plus de la duplication de l'attribut
      - Possède un getter mais pas un setter (règle « hasGetter »)
        - Rajoute un setter en plus de la duplication de l'attribut
      - Ne possède ni l'un ni l'autre (règle « hasNothing »)
        - Rajoute un getter et un setter en plus de la duplication de l'attribut

Transformation en ATL

```
module AddAccessorRefining;
create cible : CDTarget refining source : CDSource;

... liste des helpers ...

rule duplicateModelBase {
from
   sourceBase : CDSource!ModelBase
to
   cibleBase : CDTarget!ModelBase (
      allClasses <- sourceBase.allClasses,
      allInterfaces <- sourceBase.allInterfaces,
      allAssociations <- sourceBase.allAssociations,
      voidT <- sourceBase.voidT,
      intT <- sourceBase.intT,
      stringT <- sourceBase.stringT,
      boolT <- sourceBase.boolT )
}
...
rule attributeHasAll {
from
  attSource : CDSource!Attribute (
     attSource.hasSetter() and attSource.hasGetter())
to
   attTarget : CDTarget!Attribute (
      name <- attSource.name,
      owner <- attSource.owner,
      type <- attSource.type,
      multiplicity <- attSource.multiplicity )
}
```

Transformation en ATL

```
rule attributeHasSetter {
from
   attSource : CDSource!Attribute (
      attSource.hasSetter() and not(attSource.hasGetter())
)
to
   attTarget : CDTarget!Attribute (
      name <- attSource.name,
      owner <- attSource.owner,
      type <- attSource.type,
      multiplicity <- attSource.multiplicity
   ),
   getter : CDTarget!Method (
      name <- 'get' + attSource.name.firstToUpper(),
      returnType <- attTarget.type,
      owner <- attTarget.owner
   )
}
```

- Pour un attribut du source, 2 éléments sont créés coté cible
  - L'attribut équivalent
  - La méthode getter associée
    - Elle n'est pas ajoutée explicitement dans la liste des méthodes de la classe de l'attribut car cela est automatiquement réalisé via l'association bi-directionnelle entre classe et méthode : comme on a déjà positionnéle owner de la méthode getter, cela fera la liaison dans l'autre sens

Transformation en ATL

```
rule attributeHasGetter {
from
   attSource : CDSource!Attribute (
      not(attSource.hasSetter()) and attSource.hasGetter()
)
to
   attTarget : CDTarget!Attribute (
      name <- attSource.name,
      owner <- attSource.owner,
      type <- attSource.type,
      multiplicity <- attSource.multiplicity
   ),
   setter : CDTarget!Method (
      name <- 'set' + attSource.name.firstToUpper(),
      returnType <- thisModule.voidType,
      owner <- attTarget.owner,
      paramNames <- Set { 'value' },
      paramTypes <- Set { attTarget.type }
   )
}
```

Transformation en ATL

```
rule attributeHasNothing {
from
   attSource : CDSource!Attribute (
      not(attSource.hasSetter()) and not(attSource.hasGetter())
   )
to
   attTarget : CDTarget!Attribute (
      name <- attSource.name,
      owner <- attSource.owner,
      type <- attSource.type,
      multiplicity <- attSource.multiplicity
   ),
   setter : CDTarget!Method (
      name <- 'set' + attSource.name.firstToUpper(),
      returnType <- thisModule.voidType,
      owner <- attTarget.owner,
      paramNames <- Set { 'value' },
      paramTypes <- Set { attTarget.type }
   ),
   getter : CDTarget!Method (
      name <- 'get' + attSource.name.firstToUpper(),
      returnType <- attTarget.type,
      owner <- attTarget.owner
   )
}
```

Exécution de modèles

Modèles et exécution

- But de l'IDM : passer pour les modèles
  - D'une vision plutôt contemplative
  - A une vision réellement productive
- Conséquence
  - Le fossé entre le modèle et le code est (partiellement) supprimé via une correspondance (partiellement) automatique entre le modèle et le code
  - Le modèle est donc, totalement ou partiellement, directement ou indirectement, le programme qui est à exécuter
- Models at runtime (Models@runtime)
  - Partie de l'IDM qui s'intéresse à la problématique de gérer les modèles pendant l'exécution

Modèles et exécution

- Models at runtime
  - Deux types de gestion du modèle à l'exécution
    - Le modèle est directement le programme, ou une partie du, à exécuter
      - Exécutabilité/exécution de modèle
    - Le modèle est « présent » dans le code qui s'exécute
      - Modèle à l'exécution
      - Le programme s'appuie sur le modèle pendant son exécution pour prendre des décisions
        - Adaptation dynamique au contexte, utilisation d'un modèle de gestion des erreurs/problèmes ...
  - Troisième type d'intérêt, pour l'exécution de modèle
    - Exécution/simulation (partiellement) du modèle pour vérifier des propriétés
    - Vérification, validation, model-checking, prototypage ...

Exécution de modèle

- Exécution ou simulation de modèle
  - Nécessite une sémantique du modèle non ambiguë et complète
- Modèles UML
  - Ambigus sur certains points
  - UML est un langage de modélisation général
    - Laisse des « points de variation sémantique »
    - (et aussi des parties mal ou pas très bien définies ...)
  - Exemple : diagramme de classe spécifiant un programme objet
    - Notion de surcharge, redéfinition de méthodes en orienté-objet avec spécialisation et liaison dynamique
    - Sémantique variable selon le langage utilisé (Java, Eiffel, C++ ...)
    - Ce point est donc laissé non spécifié en UML pour ne pas contraindre l'usage à la spécification de programmes pour un seul langage
  - Suppression ambiguités d'UML
    - Spécialiser via un profil les points ambigus ou non spécifié
    - Restreindre l'usage à certains diagrammes ou de leurs éléments

Spécification complète d'un modèle

- Pour pouvoir exécuter un modèle, on doit spécifier complétement plusieurs aspects
  - Aspects statiques
    - Diagramme de type classe, composants ... + contraintes OCL
  - Aspects dynamiques
    - Diagramme de type machines à état, séquence, activités ...
  - Spécification complète des opérations
    - Pour l'opération d'une classe, une partie de ce qu'elle doit faire est spécifiée dans les autres diagrammes
      - Ex : diagramme de séquence spécifie que l'appel d'une opération est imbriquée dans une autre
    - Mais insuffisant, on doit disposer d'un langage exprimant le détail de l'opération : langage d'action
      - Proche d'un langage de programmation
      - UML intégre un méta-modèle d'action (pas de syntaxe concrète)

Sémantiques

- Pour un modèle, 3 sémantiques exprimables
  - Sémantique axiomatique
    - Contraintes statiques entre méta-éléments
      - Concrètement, c'est le méta-modèle : relations entre éléments, contraintes de type OCL ...
      - Sémantique exprime relation de conformité modèle/méta-modèle
  - Sémantique opérationnelle
    - Définit comportement dynamique des instances de méta-éléments
    - Décrit alors les règles d'exécution des éléments du modèle
  - Sémantique dénotationnelle
    - Utilise un autre formalisme (autre espace technologique/méta-modèle) pour lequel on a une sémantique opérationnelle ou d'exécution
    - Relation entre chaque type d'élément avec types d'éléments de l'autre formalisme donne alors la sémantique par « transitivité »

Modes d'exécution/simulation

- Deux modes d'exécution ou de simulation
  - Exécution « directe »
    - Nécessite une sémantique opérationnelle
    - Le modèle est directement exécuté, via un outil ou une plate-forme dédiée
    - Exemple : Kermeta
      - Kermeta = EMOF + langage d'action
      - Les actions ajoutées aux méta-éléments sont directement exécutables par Kermeta dans Eclipse ou via la génération de code Java standalone
  - Exécution « par traduction »
    - Nécessite une sémantique dénotationnelle
    - Traduit le modèle en un modèle équivalent pour un autre formalisme
      - Ex trivial : génération de code

Modes d'éxécution/simulation

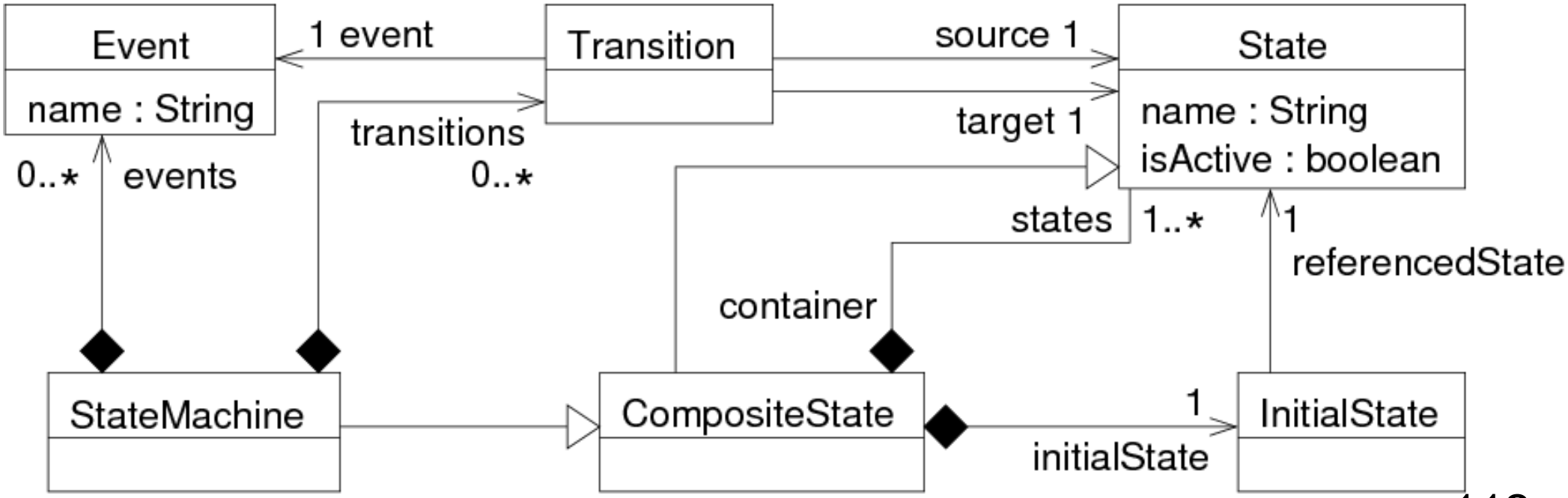
- Exécution directe
  - Avantages
    - Reste dans le même formalisme/espace technologique/outil
  - Inconvénients
    - Le fait de rester dans le même formalisme/outil peut être limitatif en terme de fonctionnalités, d'expressivité
- Exécution par traduction
  - Avantages
    - Peut utiliser les fonctionnalités de l'autre formalisme
    - Très utile notamment pour simulation/vérification
      - Peut utiliser les méthodes de preuves, les outils de l'autre formalisme (ex : réseau de pétri, logiques temporelles, model-checking ...)
  - Inconvénients
    - Nécessite une correspondance formellement définie entre les deux formalismes

Définition d'un i-DSML

- I-DSML
  - Interpreted Domain Specific Modeling Language
  - Type de modèle interprété (exécuté) par un moteur d'exécution
- Trois parties à définir pour créer le i-DSML
  - Dans le méta-modèle
    - Eléments statiques
      - Le contenu métier du modèle
    - Eléments dynamiques
      - Permet de représenter l'état du modèle pendant son exécution
    - Avec leurs règles de bonne-formation (well-formedness rules)
      - Invariants OCL
  - Dans le moteur d'exécution
    - Sémantique opérationnelle implémentée par le moteur
      - Définit comment le modèle évolue en fonction des interactions avec le système/utilisateur ou le temps

Exemple : machines à états simplifées

- Caractéristiques
  - Etats composites avec état initial
  - Transitions associées à un événement
  - Machine à états = état composite particulier + transitions + événements



Exécution machine à états

- Initialisation
  - Activer l'état initial de la machine
  - Si cet état est composite, activer son état initial et ainsi de suite
- Exécution : traitement d'une suite d'événements
  - A chaque événement, on cherche une transition associée à l'événement partant de l'état racine
  - Si non trouvé, on cherche à partir de son état composite (s'il y en a un) et ainsi de suite
  - Si trouve une transition : désactive l'état initial et active l'état cible
    - Avec les hiérarchies d'états associées
  - Si pas trouvé de transition : ne fait rien

Exemple : exécution d'un micro-onde