	MDE4Robotique		
	Création du document : 11/11/2013	Date de fin visée : 17/11/2013	Temps restant : 0
Marchal Vincent & Knoertzer Michel		Etat : Finalisé	

Rapport de projet CAR – MDE4Robotique.

Version	Date	Auteurs	Rôle	Modification apportée au document
0.1	11/11/2013	Marchal Vincent & Knoertzer Michel	Rédacteur	Création du document
0.2	15/11/2013	Marchal Vincent & Knoertzer Michel	Rédacteur	Ajout de la présentation du méta-modèle
0.3	16/11/2013	Marchal Vincent & Knoertzer Michel	Rédacteur	Contraintes OCL
0.4	17/11/2013	Marchal Vincent & Knoertzer Michel	Rédacteur	Génération de code
1.0	17/11/2013	Marchal Vincent & Knoertzer Michel	Validateur	Corrections + validation du document

Objectifs : Ce document a pour but de présenter le projet MDE4Robotique des étudiants Marchal Vincent & Knoertzer Michel. Il contient :

- le méta modèle construit ;
- les contraintes OCL ;
- la génération de code ;
- une comparaison avec le code UrbiScript visé.

Sommaire

1. Introduction

Le but de ce projet est de développer un méta-modèle et un générateur de code pour faciliter le développement de petites applications de robotique mobile. Le méta-modèle se base sur les diagrammes dits d'Arkin qui sont le fruit d'une démarche basée sur les comportements.

Voici les étapes que nous avons suivies durant le déroulement du projet :

1. Définissions du méta-model du robot ;
2. Création de contraintes OCL affinant les instances du méta-modèle ;
3. Création d'une instance du robot correspondant à e-puck ;
4. Génération de code UrbiScript avec Jet et le pattern Visitor.

Nous avons dû itérer principalement sur les étapes 1 et 3 pour rectifier certaines incohérences de notre méta-modèle.

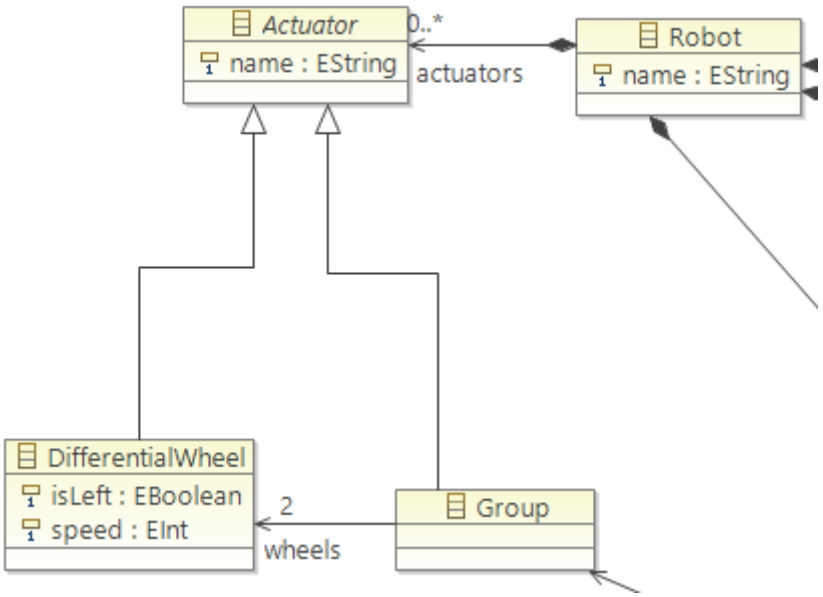
2. Le méta-modèle

a. Le robot

Un robot est constitué d'effecteurs (*Actuator*), de capteurs (*Sensor*) et de comportement (*Behavior*). Nous lui avons également ajouté une liste d'action (*Action*). Les images ci-dessous sont des parties séparées de notre méta-modèle.

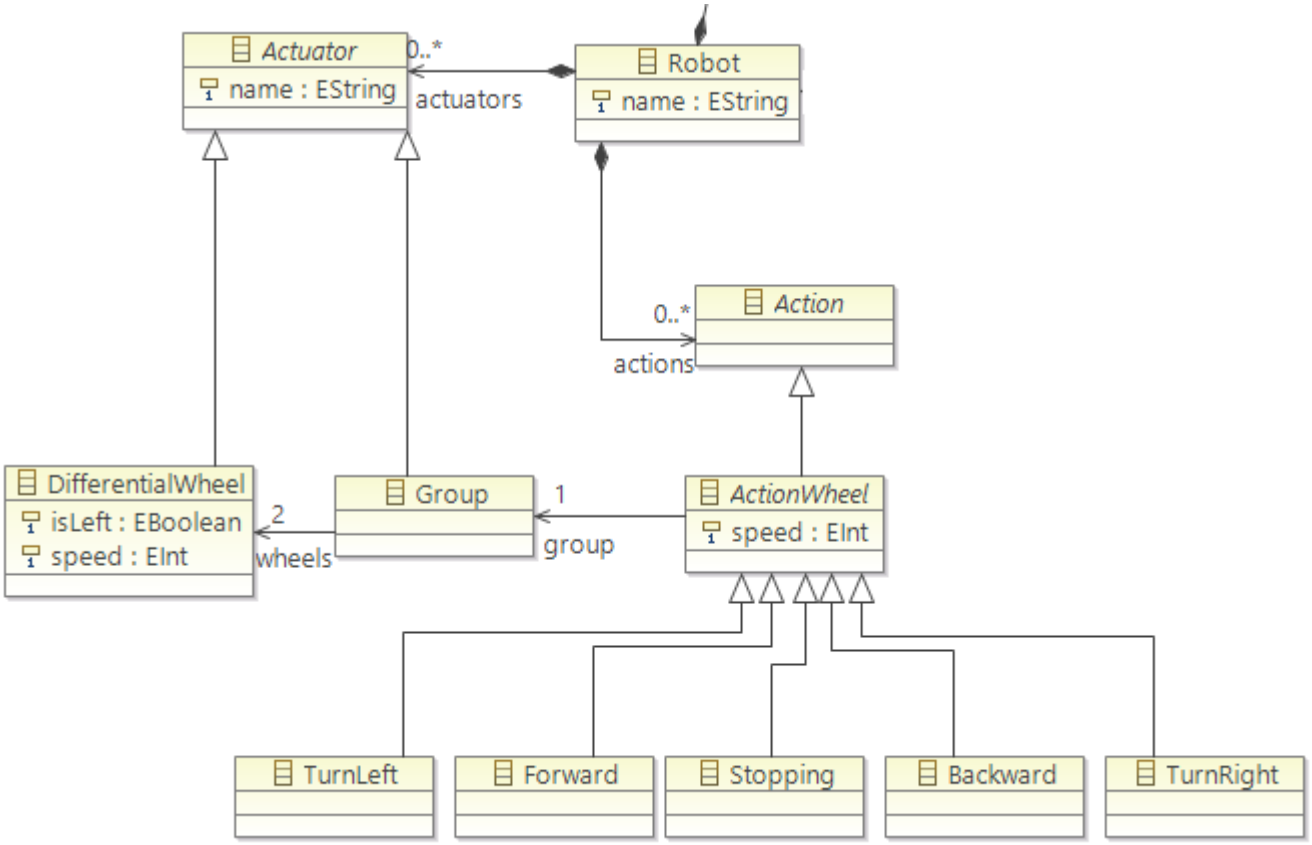
b. Les effecteurs

Dans le cadre d'e-puck nous avons distingué deux effecteurs : les roues (*DifferentialWheel*) et les groupes de roues (*Group*). Il est facilement possible d'ajouter d'autres effecteurs car nous avons une classe abstraite (*Actuator*).



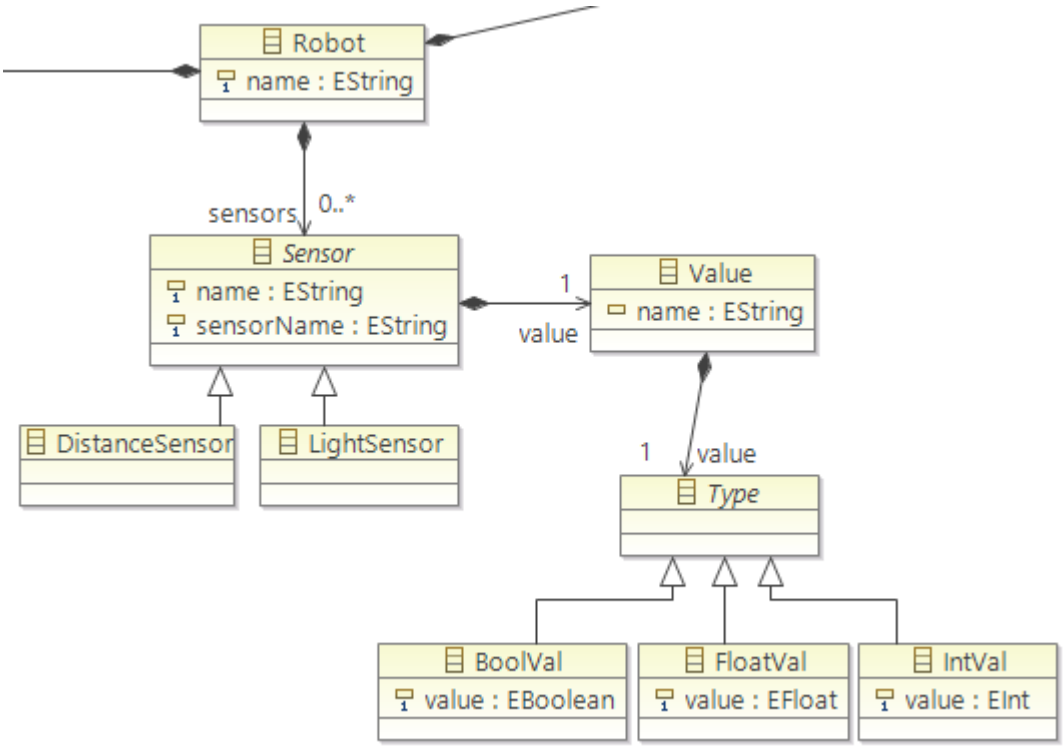
c. Les actions

Nous avons ajouté une liste d'action au robot. Pour notre exemple nous n'avons que des actions sur les effecteurs des roues (reliés à la classe *Group*), mais une classe abstraite (*Action*) permet une extension aisée des actions.



d. Les capteurs

De même que pour les effecteurs, nous avons distingué deux capteurs mais il est facile d'en ajouter d'autres. Nous avons différencié les capteurs de distance (*DistanceSensor*) des capteurs de lumière (*LightSensor*). Chaque capteur est créateur d'une valeur (*Value*), qui peut avoir plusieurs types (*BoolVal*, *FloatVal*, *IntVal*, etc.).

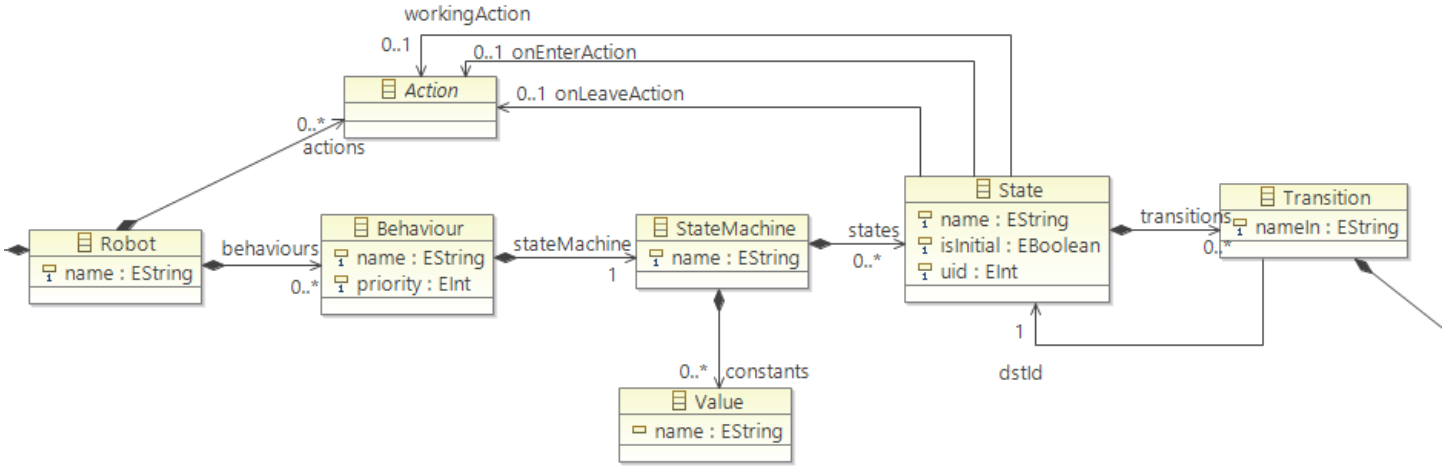


e. Les comportements

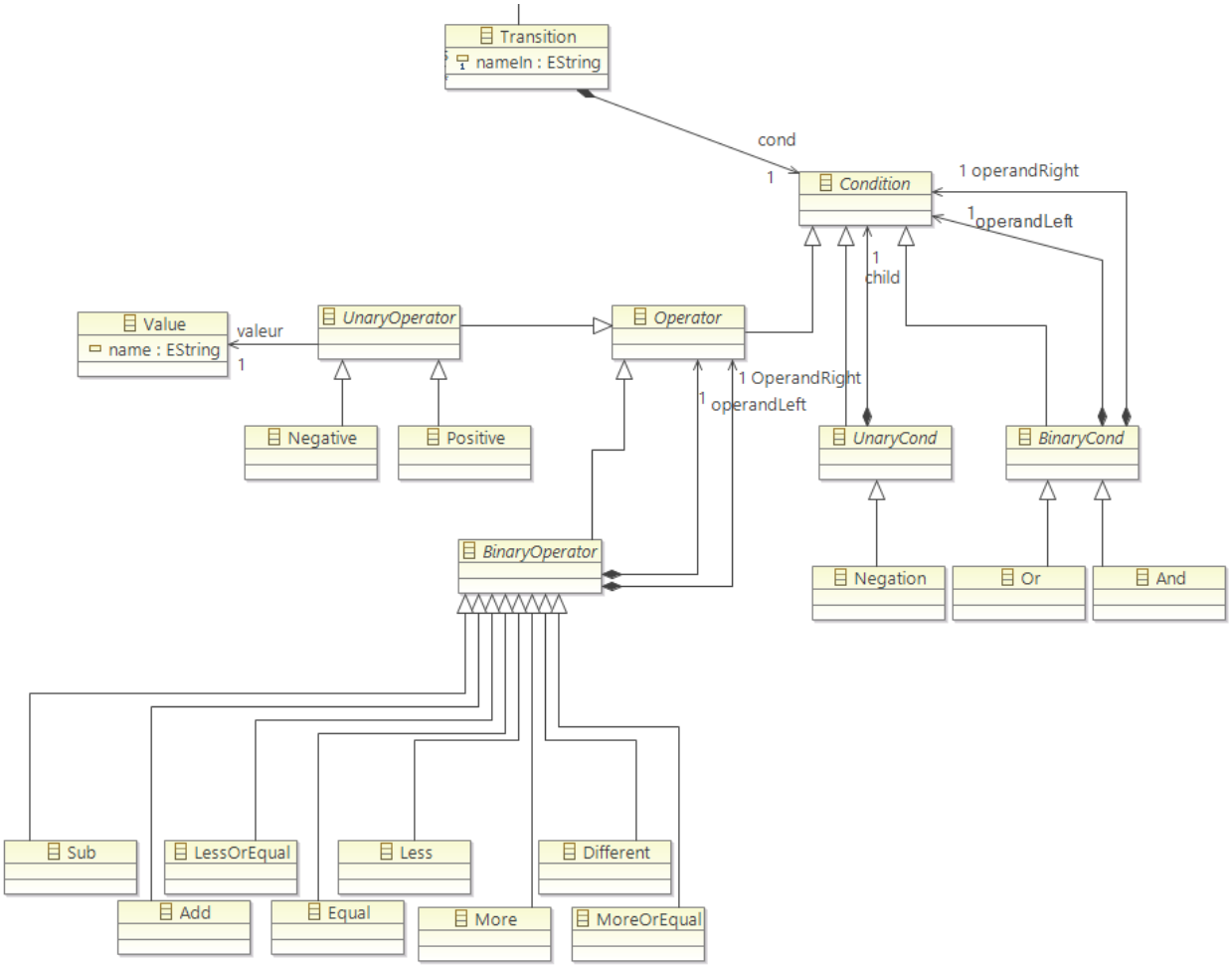
Le robot est composé de plusieurs comportements ayant une priorité (*Behavior*). Chaque comportement possède une machine à état (*StateMachine*) pouvant contenir des variables constantes et des états (*State*). Chaque état possède trois actions :

- Une à exécuter pendant l'état ;
- Une à exécuter à l'entrée de l'état ;
- Une à exécuter à la sortie de l'état.

De plus les états possèdent des transitions (*Transition*) qui permettent de quitter cet état et qui amènent à d'autres états (référence *dstId*).



Et enfin, les transitions possèdent une condition (*Condition*). Les conditions peuvent être des conditions unaires (*UnaryCond*) ou binaires (*BinaryCond*), mais aussi des opérations (*Operator*) unaires (*UnaryOperator*) ou binaires (*BinaryOperator*). Grâce à ce méta-modèle représentant les conditions sous forme d'arbre binaire (Cf. *OperandRight* et *OperandLeft* pour les expressions binaires), nous pouvons exprimer toute sorte de condition.



3. Contraintes OCL

Afin de rendre plus cohérentes nos instances de robot, nous avons défini plusieurs contraintes OCL :

- Contrainte 1 : StateUidUnique. Assure l'unicité des uid (identifiants) de chaque état (*State*).
- Contrainte 2 : TransitionUniqueName. Assure l'unicité des noms des transitions (*Transition*).
- Contrainte 3 : StateTransitionDestination. Assure qu'une transition n'a pas pour destination sa source (ce qui ferait une boucle).

a. Contrainte 1

Nous avons remarqué que dans le code UrbiScript fournit les identifiants des états étaient différents (unamed_1, unamed_2, etc.). C'est pourquoi nous avons ajouté cette contrainte. Voici le code OCL (l'invariant est dans la classe *StateMachine*) :

```
invariant StateUidUnique:
  self.states->collect(s : State | s.uid)->isUnique(n : Integer | n);
```

b. Contrainte 2

De même que pour la contrainte précédente, nous avons défini une vérification sur le nom des transitions. Voici le code OCL (l'invariant est dans la classe *State*) :

```
invariant TransitionsUniqueName:
  transitions->collect(t : Transition | t.nameIn)->isUnique(n : String | n);
```

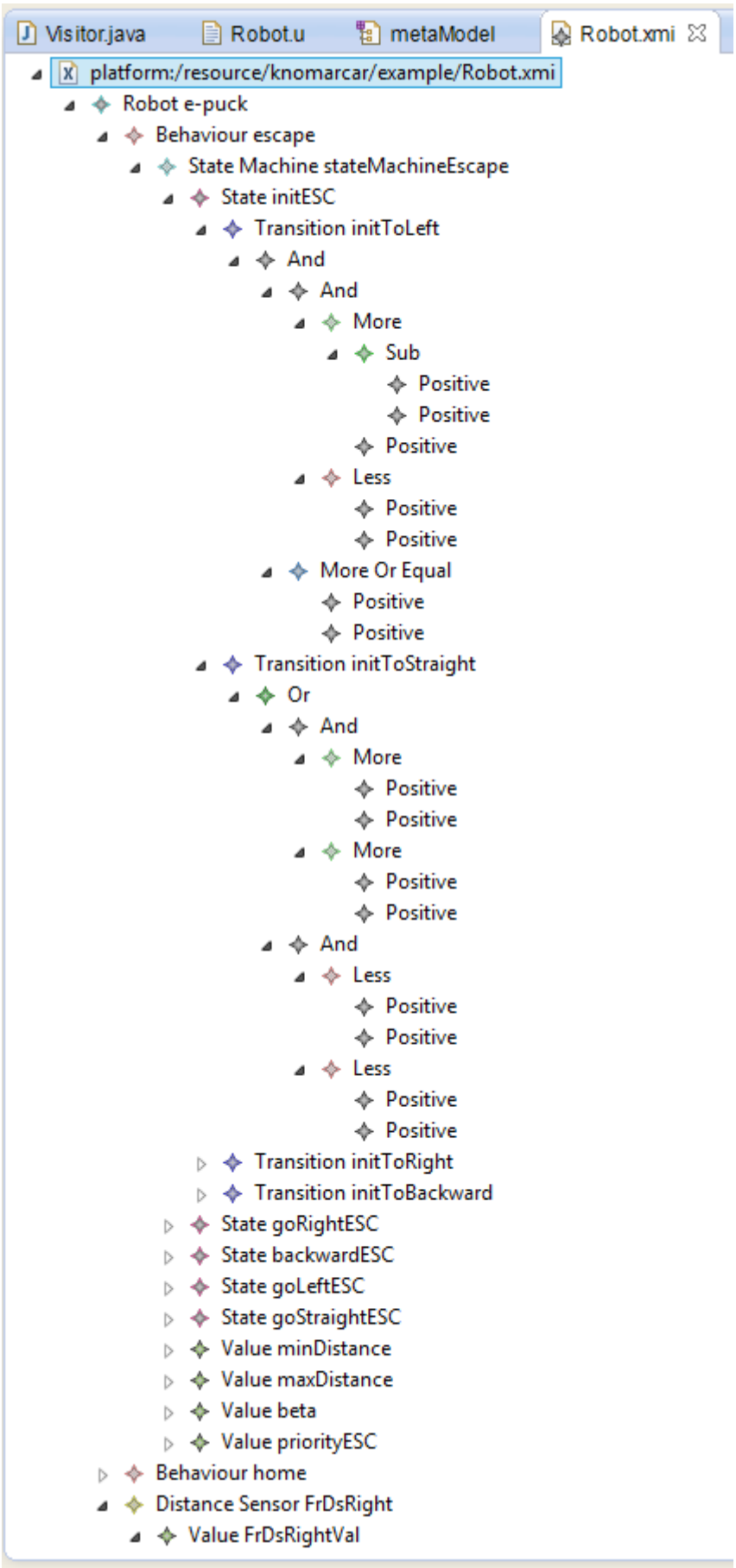
c. Contrainte 3

Afin d'éviter d'avoir une machine à état avec des transitions formant des boucles (la source est égale à la destination), nous avons défini cette contrainte. Voici le code OCL (l'invariant est dans la classe StateMachine) :

```
invariant StateTransitionDestination :
  states->select(s : State | s.transitions->select(t: Transition | t.dstId = s)->notEmpty())
        ->size() = 0;
```

4. Instance du méta-modèle

A partir du code UrbiScript fournit, nous avons pu faire une instance de robot validant notre méta-modèle et les contraintes OCL. Créer cette instance fut une partie longue et éprouvante du projet...



L'instance xmi est dans le dossier « example ».

5. Génération de code

a. Classes Java

Grace au méta-modèle, nous avons généré les classes Java et leurs interfaces correspondantes à l'aide d'un fichier genmodel.

b. Code Urbi

Nous avons utilisé le pattern Visitor pour générer le code UrbiScript du projet *e-puck*. Nous avons modifié chaque classe générée du méta-modèle afin de spécifier qu'elles sont visitable (interface IVisitable) par un visiteur (IVisitor) :

```
public interface IVisitable {
   void accept (IVisitor v);
}
```

Puis nous avons créé le visiteur qui génèrera le code UrbiScript :

```
public interface IVisitor {

   public String getValue ();

   public void visit (Action a);
   ...
   ...
   public void visit (UnaryOperator a);
   public void visit (Value a);
}
```

Nous avions comme objectif de générer le code UrbiScript correspondant à la mission SearchForLight dont voici les fichiers et leurs pourcentages de génération par notre pattern Visiteur :

1. Actuators.u : généré à 100% ;
2. Sensor.u : généré à 100% ;
3. Fsm_escape.u : généré à 90%. La fin du fichier contient une dizaine de ligne concernant les « Channel » que nous n'avons pas eu le temps de comprendre, modéliser et générer automatiquement ;
4. Fsm_home.u : généré à 90% pour les même raisons ;
5. Fsm.u : ce fichier contient la base nécessaire au fonctionnement du robot. Il ne pouvait donc pas être généré;
6. Main.u. Ce fichier contient 8 lignes de code que nous avons directement intégrées à la main.

6. Conclusion

Pour des raisons de problème d'installation, nous n'avons pas pu faire tourner le code UrbiScript fourni sur nos machines. Notre code généré n'a donc pas été testé avec le logiciel Webots. En revanche, nous avons comparé le code manuellement, et il correspondait bien à l'objectif fixé.

Malgré la partie sur les « Channel » que nous n'avons pas pu générer dynamiquement selon l'instance xmi, nous pensons que notre projet est complet et peut être réutilisé pour modéliser d'autres comportements. Notre méta-modèle peut aussi très bien s'appliquer à d'autres robots, à condition que les nouveaux capteurs, effecteurs et actions soient ajoutés au méta-modèle.

Ce projet fut fort intéressant, tant sur le plan modélisation que sur le plan robotique.