Un DSL pour un ADL

1 Création du méta-modèle Ecore de l'ADL

Lancer *Obeo Designer Community*, choisir un dossier de projets *workspace* sur un disque accessible via le réseau.

1.1 Création d'un Ecore Modeling Project

Créer un nouvel **Ecore Modeling Project**, l'appeler fr.ensma.idm.votrenom.adl. C'est un nommage typique de projet de meta-modèle (MM), avec domaine de premier niveau (ici fr), puis de second niveau (ensma), et enfin des noms de dossier séparés par un point. La seconde fenêtre est aussi importante puisque nous allons donner le nom de paquetage, le Ns Uri et le Ns Prefix. Ici Ns signifie Namespace, ce n'est pas grave si l'URI n'existe pas vraiment, ce qui est important est qu'il soit unique. Nous allons choisir comme NS Uri http://www.ensma.fr/idm/votrenom/adl, le nom du paquetage reste adl, ainsi que le préfixe. Le point de vue sera le Design qui permet d'éditer le MM à l'aide d'un diagramme de classes.

Dans le projet créé, les éléments les plus importants sont :

- model/adl.ecore, le fichier qui contiendra notre MM ;
- model/adl.aird qui est le fichier qui permettra les visualisations ;
- model/adl.genmodel qui servira à générer une instance d'Obeo qui connaît notre MM et le *viewpoint* spécifique que nous allons créer. A la fin, toute instance d'Eclipse ou Obeo qui contiendrait notre plugin permettrait à un utilisateur d'éditer directement dans un viewpoint des instances de notre MM.
- META-INF/MANIFEST.MF contient de nombreuses informations de configuration, comme la version, les dépendances, etc.

1.2 Création du méta-modèle

Un diagramme de classe apparaît par défaut lorsqu'on déroule la hiérarchie soit du fichier adl.aird, soit du fichier adl.ecore. Dans les deux cas, nous ouvrons le même diagramme de classes.

Nous allons créer un « mini-AADL » (voir Figure 1) : un *System*, est composé de 0..* *Process*, qui sont eux-mêmes constitués de 0..* *Thread*, eux même constitués de 0..* *Function*. Une EClass *Communication* possède deux relations 1..1 nommées *dest* et *source* vers *function*. Cela permet d'identifier quelle fonction est source, et quelle est destination d'un échange fonctionnel représenté par une *Communication*. Afin de permettre de parcourir des chaînes fonctionnelles par la suite, et étant donné qu'une communication n'a pas d'existence hors d'une fonction, nous ajoutons une relation permettant de spécifier qu'une communication appartient à une fonction, celle qui est la source de la communication. Une relation permet de lier aussi les communications entrantes. Un attribut *Synchronous* permet de spécifier si la communication est synchrone ou non. Enfin, puisque chaque élément est nommé, nous créons une classe abstraite *NamedElement*, avec un attribut *name*, dont toutes les classes créées héritent. Notre entité racine sera l'EClass *System*.

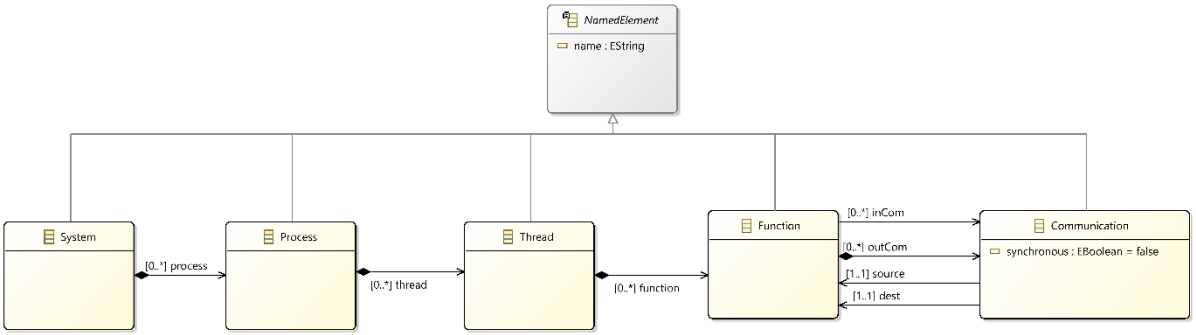

Figure 1 : Première version du MM de l'ADL

1.3 Création d'une instance test

Dans le Model Explorer (fenêtre gauche), dérouler adl.ecore jusqu'à trouver l'EClass racine System, faire un clic droit et créer une instance dynamique. Double cliquer sur le fichier System.xmi ainsi créé, ce qui ouvre l'éditeur réflexif par défaut, permettant de créer un modèle sous forme d'arbre. Dérouler platform:/resource/fr.ensma.idm.votrenom.adl/model/System.xmi afin de voir *System*. Nous allons avec le clic droit de la souris commencer à créer une instance de notre MM. Nous pouvons par exemple créer l'instance telle que sur la Figure 2. Notez qu'il faut ajouter manuellement dans les propriétés de F2 que la Communication C1-2 fait partie de ses communications *in*.

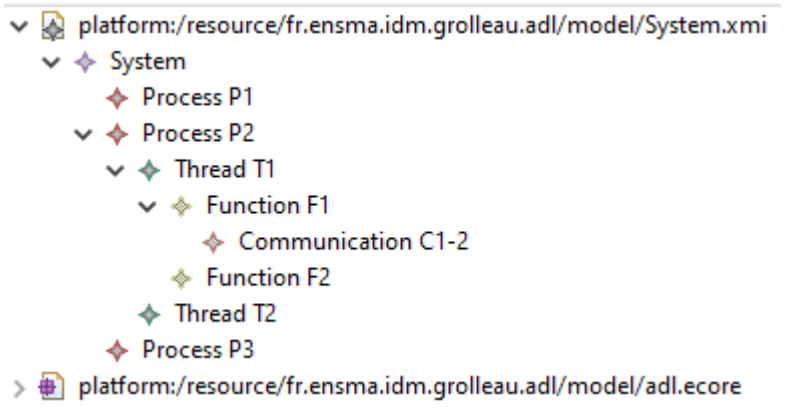

Figure 2 : Création d'une instance du MM de l'ADL

2 Création d'un diagramme ADL

2.1 Création d'un projet *Viewpoint Specification Project*

Créer un nouveau projet de type *Viewpoint Specification Project*, nommé fr.ensma.idm.votrenom.adl.design et l'appeler Adl.odesign. Ouvrir le fixhier plugin.xml de ce projet, et dans l'onglet Dependencies, ajouter le projet fr.ensma.idm.votrenom.adl. Penser à enregistrer. Ouvrir dans le dossier description le fichier adl.odesign, dérouler jusqu'à MyViewpoint. Là changer l'Id* en adlViewpoint (ce sera le nom du point de vue créé). Changer *Model file extension* en adl, de façon à ce que ce point de vue soit proposé sur les fichiers d'extension .adl.

Faire un clic droit sur adlViewpoint, et dans *New representation*, choisir *Diagram description*. Cliquer dessus, et dans les propriétés, entrer dans le champ *Id** la valeur AdlDiagram (ce sera le nom du diagramme) et dans le champ *Domain Class** la valeur adl::System (en tout cas, avec ctrl+espace, choisir le nom de votre EClass racine). Le diagramme que nous allons créer représentera le contenu de cette classe racine, ainsi un diagramme AdlDiagram représentera un adl::System. Cocher la case *Initialisation**, ainsi lorsqu'un nouveau projet de type Adl sera créé, ce diagramme sera créé

automatiquement. Dans l'onglet Metamodels, cliquer sur Add from workspace, et choisir notre MM adl.ecore.

Dans un diagramme on peut trouver plusieurs *Layers* (couches), et la couche Default est créée automatiquement. Nous allons représenter notre diagramme dans cette couche. Cliquer sur cette couche Default, et faire un clic droit. Nous allons d'abord créer un container (sous menu de *New diagram element*) qui représentera les *Process*. On renseigne donc le champ *Id** par ProcessContainer, et le champ *Domain Class** par adl::Process (bien utiliser Ctrl+espace pour s'assurer de ne pas faire d'erreur en donnant le nom de l'EClass). Faire un clic droit sur le conteneur ainsi créé et lui donner un style. Pour aller vite, nous choisissons un parallélogramme. Cliquer sur le style parallélogramme ainsi créé, et dans ses propriétés, modifier l'onglet Color pour choisir la couleur qui vous plait.

Remonter sur le nœud ProcessContainer, afin de faire un clic droit et faire de même pour créer un ThreadContainer, pour lequel on créera aussi un style – parallélogramme pour commencer. Cette fois le *Border Line Style* sera *Dashed* pour représenter les bords en pointillés. Remarquer comment notre description de diagramme permet de représenter System qui contient Process qui contient Thread.

A partir d'un clic droit sur le nœud ThreadContainer, cette fois nous allons choisir un Sub Node qui représentera l'EClass Function. En effet, les fonctions ne contiennent pas d'autre élément directement, les liens n'étant pas internes à leur conteneur. Nous pouvons lui donner l'Id* FunctionNode, et le Domain Class* adl::Function (utiliser ctrl+espace, maintenant cela devrait commencer à être une habitude). Pour le style, on choisira une ellipse.

Il nous reste maintenant à représenter les communications en tant que relations. Comme nous avons une EClass représentant nos communications, nous allons créer un *Element Based Edge* (nous aurions choisi un *Relation Based Edge* si notre lien était représenté par une relation dans notre méta-modèle). Un lien se crée au niveau Layer, il faut donc faire un clic droit sur *Default* (le nom de notre *Layer*), afin de créer ce lien. La Figure 3 montre comment renseigner les différents champs des propriétés de cet élément. La combinaison magique Ctrl+espace ou à défaut le bouton « … » sur la droite du champ, ont été utilisés pour chaque champ. Malheureusement, le Ctrl+espace a été de peu d'aide pour remplir Source finder et Target finder, pour lesquels il a fallu voir que comme on parle ici d'une EClass Communication, l'attribut source (resp. dest) permettent de lier la fonction source (resp. destination).

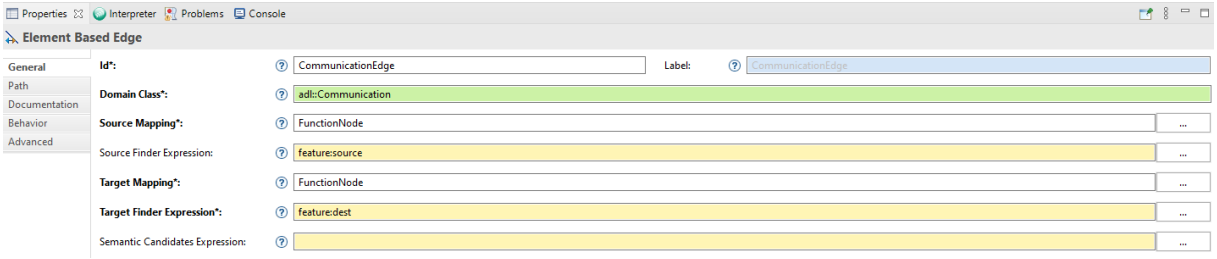

Figure 3 : propriétés du lien représentant la communication entre fonctions

2.2 Test du point de vue nouvellement créé sur une instance de notre MM

Si un petit M apparaît sur l'icône du dossier du projet originel fr.ensma.idm.votrenom.adl où nous avons créé le MM adl.ecore, et une instance de test System.xmi, vous avez déjà un projet de type modélisation. Sinon, dans le menu Configure vers le bas, cliquer sur *Convert to modeling project*. Le M apparu sur l'icône indique que c'est un projet de modélisation. On pourra dorénavant associer des points de vue spécifiques. En haut du menu contextuel obtenu par un clic droit sur le dossier de projet, les entrées de menu Viewpoint selection puis Create representation apparaissent. Choisir *Viewpoint selection*. Une boîte permettant de sélectionner les quatre points de vue standards apparaît, mais

notre point de vue AdlViewpoint n'apparaît pas : c'est parce-que nous avons dit que ce point de vue permettait de visualiser les fichiers d'extension .adl. Nous allons donc renommer notre instance System.xmi en System.adl. En faisant à nouveau *Viewpoint selection*, notre point de vue personnalisé est dorénavant proposé, nous le sélectionnons et validons. En déroulant le fichier System.adl, on peut dorénavant ouvrir le diagramme AdlDiagram. Après un peu de réorganisation, nous obtenons le diagramme donné Figure 4.

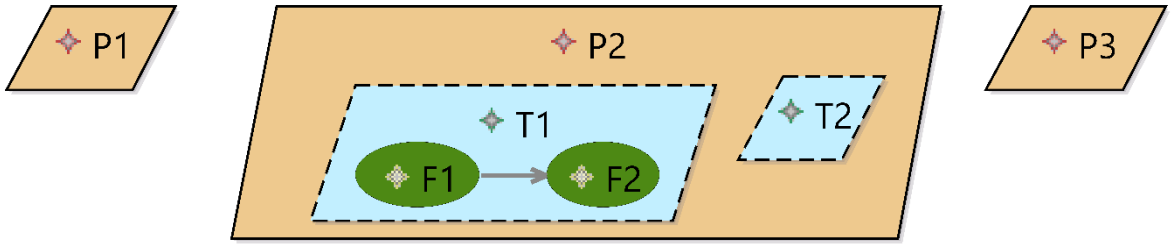

Figure 4 : notre diagramme personnalisé, permettant de visualiser un modèle d'ADL

Organiser les fenêtres de façon à visualiser en même temps System.adl dans un *Sample Reflective Ecore Editor* et le diagramme côte à côte (voir Figure 5) et s'amuser à modifier le modèle System.adl, enregistrer et regarder le diagramme se modifier en conséquence. Nous pourrions changer les icônes de façon à remplacer les petites formes en diamant par des icônes plus représentatives en remplaçant les images contenues dans le dossier icons du projet fr.ensma.idm.adl.edit généré automatiquement lorsqu'on ouvre le fichier adl.genmodel, et qu'on fait un clic doirt pour tout générer sur la racine Adl, mais nous le laissons en exercice à la maison.

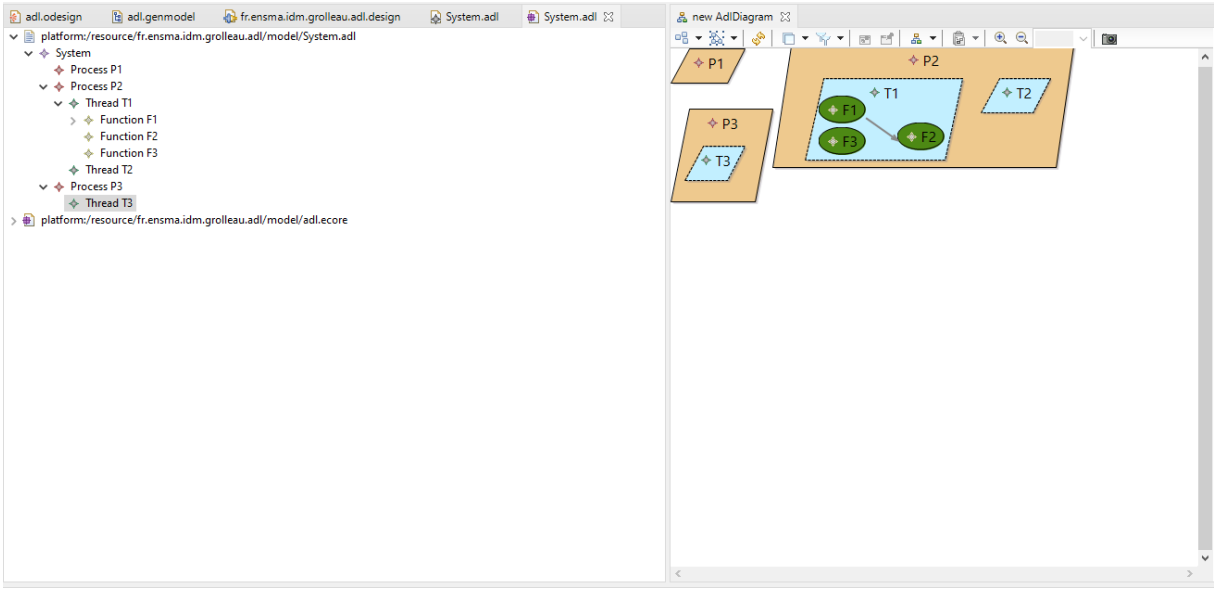

Figure 5 : visualisation côte à côte du modèle ADL via l'éditeur Ecore classique et le point de vue AdlDiagram

2.3 Ajout de contraintes

Nous allons d'abord tester des contraintes aql (Acceleo Query Language) de sorte à forcer le fait que pour une *Communication*, la *Function* donnée par *dest* possède bien la communication en question dans l'ensemble de ses communications entrantes. Pour ce faire, jouer un peu avec l'interpréteur Sirius. Pour l'afficher, passer en vue Sirius (icone en haut à droite). En bas à côté des onglets Properties, Problems et Console, un nouvel onglet apparaît. Passer en mode Sirius interpréter. On peut saisir des contraintes aql en tapant [/]. Il convient de taper d'abord ces trois caractères, puis de placer le curseur

avant le slash, avant de pouvoir utiliser la complétion automatique (ctrl+espace). Sélectionner dans le modèle d'instance du MM System.adl la communication C1-2, puis taper dans la fenêtre Expression de l'interpréteur Sirius votre expression aql.

Vous devriez arriver à une expression comme [self.dest.inCom->includes(self)/], évaluée à vrai si vous avez bien pensé dans votre instance à entrer C1-2 dans les *inCom* de la fonction F2.

Nous allons entrer cela dans l'adlViewpoint : ouvrir adl.odesign, faire un clic droit sur le nœud diagramme AdlDiagram, et ajouter une *New validation*. Sélectionner cette validation, et la nommer via ses propriétés : son nom sera FunctionReferencesInputCommunications. Par un clic droit sur la validation créée, nous pouvons ajouter une *Semantic Validation Rule*, son *Id** peut être le même que le nom de la validation, son *Level* sera *Error*, sa *Target Class** sera adl::Communication, et le message sera le message d'erreur affiché lorsque la contrainte est violée, par exemple « The element is not referenced winthin the input communications in its destination function ». Nous pouvons entrer l'invariant aql testé précédemment dans l'audit que nous pouvons créer via un clic droit sur la *Semantic Validation Rule*. Malheureusement, dans la version actuelle, la syntaxe [contrainteaql/] ne semble pas fonctionner, il nous faut l'écrire sous la forme aql:contrainteaql. Ainsi par exemple si la contrainte dans l'interpréteur sirius était [self.dest.inCom->includes(self)/], alors dans l'audit il faut l'entrer sous la forme aql:self.dest.inCom->includes(self)

Pour cette même règle nous pouvons même proposer une correction qui pourra être appliquée par l'utilisateur, un *Fix*, appelé par exemple FixFunctionReferencesInputCommunications. En faisant un clic droit sur ce *Fix*, nous pouvons insérer un *Begin* qui correspond à démarrer une action. Il nous faut ajouter la *Communication* dans la liaison 0..* *inCom* (qui est donc un *OrderedSet*) de la *Function* pointée par la liaison 1..1 dest (qui est donc directement la *Function* en question).de la *Communication*.Or, dans un VSM (*Viewpoint Specification Model*), on ne peut modifier qu'un attribut de l'objet courant (contexte). Il va donc nous falloir changer le contexte à la *Function* à modifier. Avant cela, il nous faut mémoriser la *Communication* courante qui a levé l'erreur. Par conséquent, sous le Fix, nous commençons par créer un nœud Begin, ce qui est obligatoire, ce nœud va démarrer par un nœud *Let* qui va mémoriser la valeur *self* de la Communication, en disant que la variable, nommée par exemple, *instance* (champ *Variable Name**), doit recevoir la valeur *Value Expression** aql:self. Ensuite, on crée un nœud *Change Context*, qui va mettre la valeur de self à aql:self.dest, c'est-à-dire que le contexte est dorénavant la *Function* à changer. Au fur et à mesure que vous créerez les nœuds, faire Validate de façon à vérifier que la syntaxe est correcte. Il reste maintenant à ajouter *instance* à la valeur de la liaison inCom, pour ce faire nous créons un nœud Set, où *Feature Name** est inCom, et Value expression est aql:self.inCom->prepend(instance).

Afin de tester votre règle et son fix, s'arranger pour que dans System.adl, une communication existe sans que la fonction destination possède son nom dans sa liaison inCom. Sur le diagramme AdlDiagram correspondant, un symbole d'erreur doit apparaître sur la communication en question, de plus dans l'onglet Problems qui regroupe tous les avertissements, informations, et erreurs, l'erreur telle qu'elle a été décrite dans la *Semantic Validation Rule* doit apparaître dans la liste d'erreurs. En faisant un clic droit, un fix es proposé. L'appliquer, sauvegarder le diagramme AdlDiagram, et observer dans la vue Ecore de System.adl que le problème a bien été résolu sur le modèle (la communication apparaît maintenant dans la liste des inCom de la fonction destination).

2.4 Création d'une palette de commandes permettant de modifier un diagramme directement

Dans le *Sirius Specification Editor* d'adl.odesign, faire un clic droit sur le nœud Default (Layer), et choisir *New tool->section*. Comme Id*, nous allons choisir par exemple ModelElements. Cette section de la

palette d'outils permettra de créer des Process, des Threads et des Functions. Notez qu'on pourrait là encore personnaliser les icônes mais là encore, c'est laissé en exercice à la maison.

Les *Process* peuvent être créés n'importe où dans le diagramme. Par un clic droit sur la Section ModelElements, on va choisir *New Element Creation* puis *Container Creation*. Son *Id** sera ProcessCreation, Dans *Container Mappings**, le bouton à droite permet de choisir *ProcessContainer*. En regardant les propriétés de Node Creation Variable container, on voit que le nom du conteneur du Process créé se trouve dans la variable container. Un clic droit sur *Begin* nous permet de dire que l'on souhaite créer une instance de *Process*par le menu *New operation->Create instance*. Sur cette nouvelle instance créée (propriétés de du nouveau nœud *Create Instance*), dans *Reference Name**, nous entrons donc la valeur *container*. Dans le champ *Type Name*, la combinaison ctrl+espace permet de choisir adl::Process. Si on le souhaite, on peut nommer l'instance créée par défaut, le contexte passe automatiquement sur cette nouvelle instance, il suffit donc d'ajouter un nœud *Set*, qui permet de modifier le *Feature Name** name, et lui donner la Value Expression Process.

La création de l'outil *ThreadCreation* fonctionne sur le même principe, un copier/coller de *ProcessCreation*, modifié pour l'adapter au Thread, simplifie l'opération.

Mis à part que c'est un *Node Creation* à la place d'un *Container Creation*, la création de *Function* repose sur le même principe que les deux autres.

Reste à créer le lien, c'est l'opération la plus délicate. A partir de Section ModelElements, choisir *New Element Creation -> Edge Creation*. Nous allons lui donner l'*Id* CommunicationCreation*, et le seul *Edge Mappings** possible est *CommunicationEdge* (bouton … à droite). Noter, en regardant les propriétés de *Source* (resp. *Target*) *Edge Creation Variable* que le nœud source (resp. destination) est dans la variable source (resp. *target*). En faisant un clic droit sur le nœud *Begin*, nous allons choisir *Change Context* de façon à nous placer dans var:source (la fonction source). En effet, un lien appartient (lien de *containement*) à la fonction source. A partir de là, nous créons donc une instance (*Create Instance*) qui est contenu dans *Reference Name** qui est *outCom*. Le *Type Name* est bien entendu adl::Communication, et l'instance ainsi créée sera mémorisée dans la variable *instance*. La Communication créée appartient donc à la liaison outCom de la fonction source. Il nous reste à mettre à jour les liaisons source et dest de la Communication créée et à mettre celle-ci dans la liaison inCom de la fonction destination. Par un clic droit sur le nœud *Change Context var:source*, on peut créer un *Change Context* sur *var:instance*, le contexte est donc maintenant l'instance de Communication que nous venons de créer. Sous ce Change Context, nous ajoutons donc deux nœuds *Set* : l'un met la valeur de la liaison *source* à var:source, et l'autre met la valeur de *dest* à *var:target*. Reste à faire un *Change Context* sur var:target, la fonction destination, et faire un Set qui met inCom à la valeur aql:self.inCom->prepend(instance) comme nous l'avions fait dans le Fix de l'erreur correspondante.

Et voilà, nous avons maintenant un diagramme AdlDiagram complètement opérationnel, qui nous permet de modifier à loisir des instances de notre MM. Le plugin généré peut être intégré dans un Eclipse, ou Obeo, pour que les modèles d'extension .adl soient édités avec notre diagramme.

3 Génération de code POSIX

Nous allons générer un fichier POSIX par processus, avec un main qui va créer les tâches, une déclaration de chaque fonction qui se contente d'afficher son nom à l'écran, et des threads tous de période 1 seconde. Voici un fichier généré pour l'un des processus : le processus contient deux threads, T1 et T2. Le thread T1 appelle quatre fonctions F1, F2, F3, Function. Le fichier est compilable et à l'exécution, le nom des fonctions s'affiche chaque seconde.

```c
#include <pthread.h>
#include <stdio.h>

void add_us(struct timespec *t,unsigned us)  {
    t->tv_nsec+=us%1000000;
    if (t->tv_nsec>1000000000) {
        t->tv_sec++;
        t->tv_nsec-=1000000000;
    }
    t->tv_sec+=us/1000000;
}

void func_F1() {
    // Start of user code fun_F1
    // TODO should be implemented
    // End of user code
    printf("func_F1\n");
}
void func_F2() {
    // Start of user code fun_F2
    // TODO should be implemented
    // End of user code
    printf("func_F2\n");
}
void func_F3() {
    // Start of user code fun_F3
    // TODO should be implemented
    // End of user code
    printf("func_F3\n");
}
void func_Function() {
    // Start of user code fun_Function
    // TODO should be implemented
    // End of user code
    printf("func_Function\n");
}
void * thread_T1() {
    struct timespec release;
    clock_gettime(CLOCK_MONOTONIC,&release);
    for (;;) {
        // Start of user code thread_T1
        // TODO should be implemented
        // End of user code
        // Should call functions :
        func_F1();
        func_F2();
        func_F3();
        func_Function();
        add_us(&release,1000000);
```

```
        clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &release,0);
    }
}
void * thread_T2() {
    struct timespec release;
    clock_gettime(CLOCK_MONOTONIC,&release);
    for (;;) {
        // Start of user code thread_T2
        // TODO should be implemented
        // End of user code
        // Should call functions :
        add_us(&release,1000000);
        clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &release,0);
    }
}
void main() {
    pthread_t pthread_t_T1;
    pthread_create(&pthread_t_T1,0,thread_T1,0);
    pthread_t pthread_t_T2;
    pthread_create(&pthread_t_T2,0,thread_T2,0);
    pthread_join(pthread_t_T1,0);
    pthread_join(pthread_t_T2,0);
}
```

Afin de générer ce fichier, nous allons passer en perspective Acceleo. De là, créer un projet Acceleo Project, nommé fr.ensma.idm.votrenom.toPOSIX. La première page de création peut rester telle quelle, cependant, sur la seconde page, il faut cocher Generate file et Main template afin qu'un fichier MTL soit créé. Il faut aussi ajouter comme *Metamodel URIs*, radio bouton *Runtime Version* : http://www.eclipse.org/emf/2002/Ecore .
http://www.ensma.fr/idm/votrenom/adl (l'URI associée au projet contenant le MM Adl)

Dans le sous dossier src/fr.ensma.idm.votrenom.toPOSIX.main se trouve le fichier generate.mtl. C'est le fichier Acceleo à modifier pour générer du code à partir des modèles Adl.

La ligne `[template public generateElement(anEClass : EClass)]` sera appliquée pour toute EClass, cependant nous voulons créer un fichier pour chaque Process. Il faut donc modifier cette ligne en :

```
[template public generateElement(aProcess : Process)]
```

Comme le fichier généré devrait s'appeler nom_du_Process.c, il faut changer la command [file…] en :

```
[file (aProcess.name.concat('.c'), false, 'UTF-8')]
```

Ensuite nous entrons en plein texte la partie en-tête : les inclusions, la définition de la fonction add_us. Pour chaque fonction contenue dans un thread, nous devons créer une fonction func_nomFunction de profil void* void*, qui affichera le nom de la fonction.

L'idée est donc de faire une boucle for sur les threads du Process dont on génère le fichier (variable *aProcess* qui est le paramètre de la fonction generateElement), puis une boucle pour chaque Function contenue dans le Thread. Après la définition des fonctions, on définit le thread, qui s'appelera thread_nomThread. Celui-ci initialise une horloge *release*, puis fait une boucle sans fin qui appelle

chaque fonction (nécessite une autre boucle sur les fonctions), avant d'ajouter un million du microsecondes à l'horloge release, et d'attendre cette date avant de terminer l'itération de boucle.

Enfin, la fonction main déclare une variable de type pthread_t utilisée pour garder l'identifiant d'un thread POSIX lors de sa création, puis lance la création du thread avec des attributs par défaut, et un argument null passé au thread. Lorsque tous les threads sont ainsi créés, il reste à effectuer un pthread_join sur chaque variable de type pthread_t.

Essayer soi-même avant de regarder la solution :

```
[comment encoding = UTF-8 /]
[module
generatePOSIX('http://www.eclipse.org/emf/2002/Ecore','http://www.ensma.fr/idm/gro
lleau/adl')]


[template public generateElement(aProcess : Process)]
[comment @main/]
[file (aProcess.name.concat('.c'), false, 'UTF-8')]
#include <pthread.h>
#include <stdio.h>

void add_us(struct timespec *t,unsigned us)  {
	t->tv_nsec+=us%1000000;
	if (t->tv_nsec>1000000000) {
		t->tv_sec++;
		t->tv_nsec-=1000000000;
	}
	t->tv_sec+=us/1000000;
}

	[for (t : Thread | aProcess.thread)]
		[for (f: Function | t.function)]
void *func_[f.name/](void *) {
	// [protected ('fun_'+f.name)]
	// TODO should be implemented
	// [/protected]
	printf("func_[f.name/]\n");
}
		[/for]
void * thread_[t.name/]() {
	struct timespec release;
	clock_gettime(CLOCK_MONOTONIC,&release);
	for (;;) {
		// [protected ('thread_'+t.name)]
		// TODO should be implemented
		// [/protected]
		// Should call functions :
		[for (f: Function | t.function)]
		func_[f.name/]();
		[/for]
		add_us(&release,1000000);
		clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME, &release,0);
	}
}
	[/for]
void main() {
	[for (t : Thread | aProcess.thread)]
	pthread_t pthread_t_[t.name/];
```

```
        pthread_create(&pthread_t_[t.name/],0,thread_[t.name/],0);
        [/for]
        [for (t : Thread | aProcess.thread)]
        pthread_join(pthread_t_[t.name/],0);
        [/for]
}
[/file]
[/template]
```

Afin de générer du code, dans le projet initial cotnenant le MM Adl, changer la nature du projet via le menu contextuel (clic droit) *Configure->Toggle Acceleo Nature*. Remarque : dans l'onglet *Properties->Project Natures*, les natures du projet sont visibles. Notre projet doit être de nature Modeling, Acceleo, Java et Plug-in development. Créer un dossier Src dans le projet, il nous servira à générer les fichiers POSIX.
Sur le bouton Run (comme un bouton *Play* vert de lecteur de musique ou video), la flèche permet d'ouvrir différentes façons de lancer le projet. Cliquer dessus afin de créer une *Run Configuration*. Sélectionner l'entrée *Acceleo Application*, et créer une nouvelle configuration d'exécution de cette nature. Comme *Project*, choisir fr.ensma.idm.votrenom.toPosix. Comme *Main class*, choisir Generate, qui correspond au nom de votre fichier mtl. Pour le *Model*, ce sera notre instance exemple de MM Adl, System.adl, pour la *Target*, sélectionner le dossier Src que vous avez créé dans le projet. Il suffit de lancer cette commande pour générer un fichier POSIX par *Process*. Ce fichier pourra être compilé par la commande gcc -o testposix nomfichier.c -lpthread. Ensuite, lancer le fichier avec la commande ./testposix.

4 A vous de jouer

Il existe des capteurs et actionneurs, que nous allons représenter par une EClass *Device*. Il existe deux classes différentes de *Device* : les *PullDevice* (type capteur analogique par exemple) que l'on va scruter, caractérisés par un temps de réponse donné en microsecondes, et les *PushDevice*, caractérisés par une période minimale et une période maximale. Il existe une et une seule communication sortante ou bien une et une seule communication entrante. S'il existe une communication entrante, c'est un actionneur, et cette communication doit être de nature synchrone. S'il existe une communication sortante, celle-ci ne peut être que de nature asynchrone si c'est un *PullDevice*, mais elle doit être de nature synchrone si c'est un PushDevice. Un organe de dialogue sera donc représenté par deux devices différents, l'un pour sa fonction d'actionneur, l'autre pour sa fonction de capteur. Un processus, en plus de contenir des threads, peut contenir des ISR (Interrupt Service Routine). Celles-ci sont typiquement utilisées lorsqu'on ne souhaite pas utiliser de façon synchrone la sortie d'un Push dans un thread. Elles sont activées par une et une seule communication provenant d'un PushDevice, elles peuvent contenir des fonctions (elles en contiennent normalement au moins une activée par une communication provenant d'un PushDevice). Les PushDevice sont munis d'une période minimale et d'une période maximale.

Chaque *thread* possède un et un seul rythme d'activation, celui-ci peut être de nature :

- *Periodic*, de période donnée en microsecondes. Dans ce cas aucune fonction ne peut posséder de *Communication* entrante synchrone provenant d'une fonction hors du même *thread*. Graphiquement en AADL, cela est représenté par la période dans un ovale en haut à gauche du parallélogramme.
- Sporadic, de période minimale donnée en microsecondes. Dans ce cas il existe une et une seule communication de nature synchrone provenant de l'extérieur du thread. Cette communication est celle qui active la première fonction du *thread*. La période minimale est obtenue soit à partir de la période minimale du PushDevice délivrant la Communication, si

c’est le cas, moins 2% arrondi à la microseconde inférieure, soit c’est la période de la sporadique le précédant si c’est un thread qui lui délivre la communication synchrone.

- *Timed*, de délai de watchdog donné en microsecondes, affiché comme la période, avec un texte « T : duréewatchdogenus ». Dans ce cas il existe au moins une communication entrante nommée *reset*, arrivant sur la première fonction, permettant de réinitialiser le watchdog.