# TP : créer son langage de modélisation avec EMF et générer automatiquement le code d'un simulateur

## Objectif du TP

L'objectif de ce TP est d'apprendre à définir le méta-modèle d'un DSL et d'utiliser ce méta-modèle pour (i) créer des exemples d'applications, (ii) vérifier qu'ils respectent des contraintes structurelles, et (iii) générer du code C associé. Le sujet est décomposé en trois parties :

1. Spécification du DSL : cette partie décrit les éléments du DSL, vous devrez compléter un méta-modèle EMF à partir de cette description.
2. Vérifications structurelles : cette partie décrit les contraintes que devra respecter un modèle utilisant votre DSL. Vous devrez implémenter les vérifications décrites dans cette partie.
3. Génération de code C : cette partie décrit les principes de la génération de code C correspondant au DSL proposé. Vous devrez compléter l'implémentation de ce générateur de code.

Dans ce TP, vous utiliserez EMF, le langage de programmation Java, et générerez du code C.

## TH1 - Spécification du DSL

Le DSL que vous allez développer doit permettre de modéliser une application logicielle comme un graphe dirigé acyclique (DAG – Directed Acyclic Graph) de tâches. Un DAG de tâche est un ensemble de tâches et de canaux de communications entre ces tâches.

Une tâche est caractérisée par un nom, une période, et un ensemble de ports. Chaque tâche doit avoir un nom unique. Un port est un point de communications entre tâches et il est caractérisé par un nom, une direction (entrée ou sortie), et un type de donnée. Pour simplifier on ne considérera que des données de type *int* et *float*.

Un canal de communication relie un port de sortie d'une tâche au port d'entrée d'une autre tâche. Un port de sortie peut écrire sur plusieurs canaux alors qu'un port d'entrée ne peut recevoir des données que depuis un unique canal. Cette contrainte est illustrée sur la figure 1.

Profitons de cette figure pour décrire les éléments du modèle :

- Le DAG valide est constitué de trois tâches T1, T2 et T3.
- T1 possède un port de sortie p1, T2 possède un port d'entrée p2 et T3 possède un port d'entrée p3. Le type de donnée associé n'est pas précisé dans cet exemple.
- C1 possède deux canaux : un canal reliant p1 à p2 et un canal reliant p1 à p3.

Pour finir, précisons ce que représente un canal de communication. Un canal de communication représente à la fois la transmission de données entre tâches d'un DAG et une contrainte de précédence entre ces tâches: sur l'exemple de la figure 1, l'existence du canal de communication entre p1 et p2 signifie :

1. Que la donnée produite sur p1 par l'exécution de T1 est passée en paramètre de l'exécution du point d'entrée de T2.
2. Qu'entre deux exécutions de T2, il y a eu au moins une exécution de la tâche T1. En effet, nous considérons que la connexion entre les tâches est immédiate. Cela veut dire que T2 (et T3) ne pourra être exécutée que lorsque T1 aura complété son exécution. Donc T2 doit être exécutée avec une période plus grande ou égale à T1 pour être certain qu'il y aura eu au moins une exécution de T1 durant une période de T2.

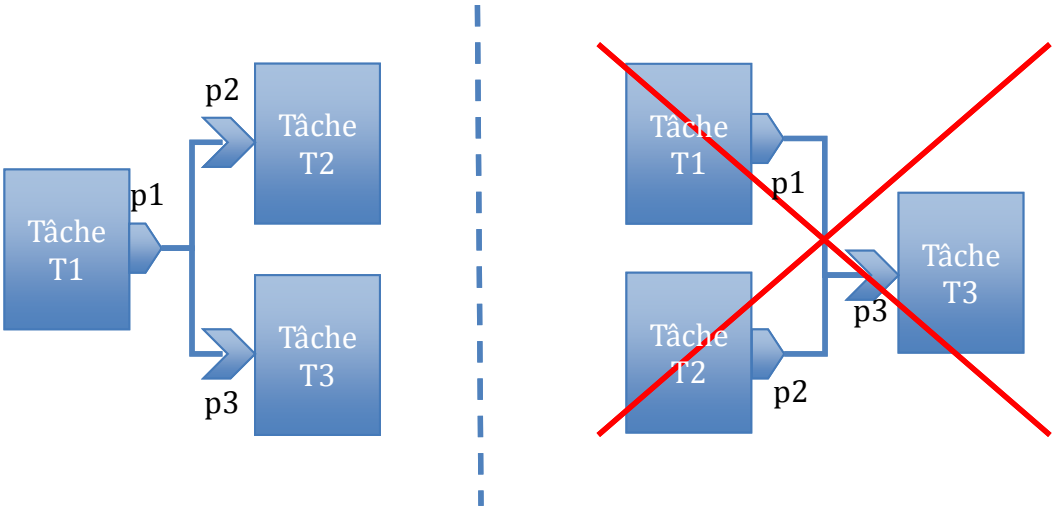



Figure 1: DAG valide (à gauche) et invalide (à droite)

La figure 2 suivante montre un exemple de modèle conforme au méta-modèle du DSL :

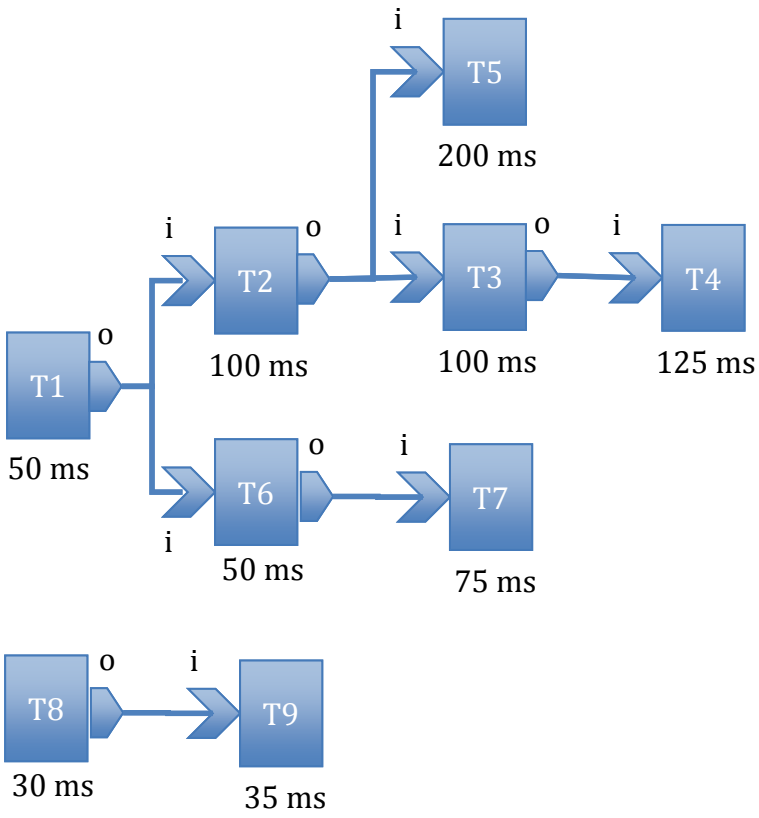
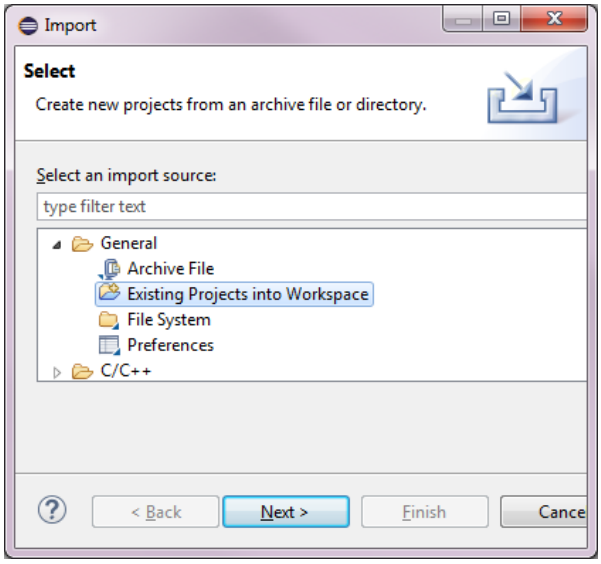


Figure 2: Exemple de DAG

**A faire :**

1. Lancer soit Eclipse Modeling Tools tel qu'installé sur votre ordinateur ou si vous avez accès à un ordinateur de l'école, exécuter les deux commandes suivantes dans une fenêtre de commande :
    a. source /infres/s3/borde/Install/env_osate
    b. /infres/s3/borde/Install/eclipse_emf/eclipse
2. Télécharger l'archive des différents projets sur le site web du cours
3. Décompresser l'archive dans un répertoire
4. Importer les projets suivants en cliquant le menu *File>>Import* puis sélectionner « Existing projects into workspace »

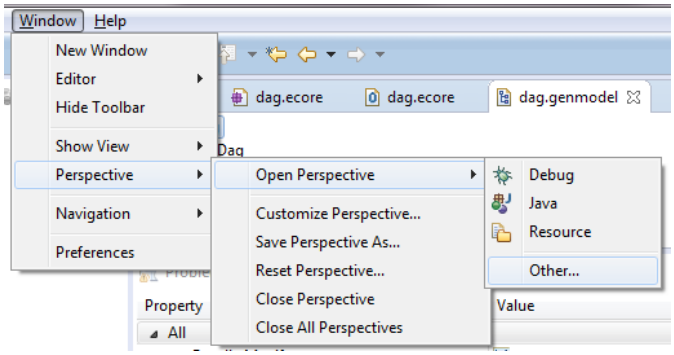



Puis aller dans le répertoire où vous avez décompressé l'archive et ajouter les projets suivants :

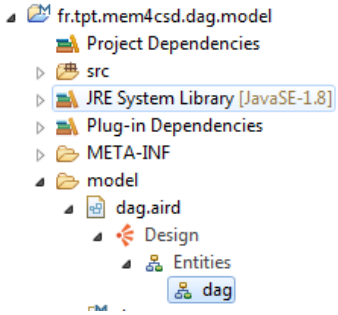
a. fr.tpt.mem4csd.dag.model
b. fr.tpt.mem4csd.dag.model.edit
c. fr.tpt.mem4csd.dag.model.editor
d. fr.tpt.mem4csd.dag.sort
e. fr.tpt.mem4csd.dag.simulator

Le projet « fr.tpt.mem4csd.dag.model » contient le méta-modèle pour le graphe de tâches tel que décrit précédemment. Vous devez compléter ce méta-modèle en y ajoutant une classe « Channel » pour décrire les communications entre les tâches.

a. Passer à la perspective « Modeling ». Pour cela, cliquer sur le menu de la figure suivante et choisir la perspective « Modeling ».




b. Ouvrir le diagramme de classe du fichier « dag.aird » dans le projet du méta-modèle en double-cliquant l'élément « dag » tel qu'illustré ci-dessous :




c. Créer une nouvelle classe nommée « Channel » à l'aide de l'éditeur du diagramme de classe (diapo 25 de la présentation du cours).
d. Choisir la(es) bonne classe(s) dont devra hériter « Channel », sachant que les instances de cette classe devront être identifiables par un nom qualifié.
e. Quelle classe devra contenir les instances de « Channel » ?
f. Définir des références nommées « sourcePort » et « destPort » vers les ports source et destination du canal de communication.
g. Créer des références *dérivées* nommées « sourceTask » et « destTask » dans la classe « Channel » pour accéder directement aux tâches source et destination (diapo 30).
h. Vérifier que le méta-modèle est valide (diapo 35).
i. Générer le code du méta-modèle (diapo 49).
j. Le calcul des références dérivées doit être implémenté dans des méthodes du code généré. Implémenter les méthodes « basicGetSourceTask » et « basicGetDestTask » pour les références « sourceTask » et « destTask » dans la classe générée « ChannelImpl ». N'oubliez pas d'ajouter l'annotation « @generated NOT » dans le commentaire du code des méthodes pour éviter que votre code soit écrasé lors de la régénération. (diapo 49)
k. Lancer une nouvelle instance d'Eclipse (diapos 54 et 55)
l. Créer un modèle simple de DAG comme celui de la partie gauche de la figure 1 de ce document (diapos 56 à 60).
m. Vérifier que les références dérivés « sourceTask » et « destTask » des instances de votre classe « Channel » se calculent tel que prévu en observant la vue propriété.

## TH2 - Vérifications structurelles

Nous souhaitons pouvoir identifier de façon non-ambiguë chaque élément du modèle par son nom qualifié. Voici les règles de construction de ces noms qualifiés :

- Tâche : nom de la tâche ; par exemple « T1 » sur la figure 1.
- Port : nom qualifié de la tâche.nom du port ; par exemple : « T1.p1 » sur la figure 1.
- Canal : nom qualifié du port émetteur -> nom qualifié du port récepteur : « T1.p1 -> T2.p2 » sur la figure 1.

**A faire :**

Implémenter le calcul du nom qualifié pour votre classe « Channel ». Pour ce faire :

1. Dans le code Java généré pour la classe, redéfinir la méthode « getQualifiedName » de la classe « IdentifiedElement » dont devrait hériter votre classe « Channel ».
2. Codez le calcul du nom qualifié selon la règle définie précédemment.
3. Relancez une instance Eclipse de test et vérifiez que le calcul s'effectue correctement.

Pour que le nom qualifié d'un élément soit unique, nous devons respecter les contraintes suivantes :

1. Chaque tâche a un nom différent.
2. Chaque port d'une même tâche un nom différent.

Pour qu'un canal de communication soit valide, il faut vérifier :

1. Que le canal relie des ports ayant les mêmes types de données.
2. Que le canal relie des ports de sortie vers des ports d'entrée (et non pas sortie vers sortie, entrée vers entrée, entrée vers sortie).
3. Que le port de sortie du canal appartient à une tâche dont la période est plus grande ou égale à la période de la tâche du port d'entrée du canal.

**A faire :**

Modifier le méta-modèle pour spécifier les contraintes précédemment mentionnées pour votre classe « Channel ».

1. Ouvrir l'éditeur arborescent de votre méta-modèle (diapo 34).
2. Sélectionner votre classe dans l'éditeur et ajouter une annotation « Ecore » (diapo 39).
3. Spécifiez les noms des contraintes n° 1 à 3 précédemment mentionnées (diapo 40).
4. Régénérer le code du méta-modèle.
5. Implémenter les contraintes structurelles mentionnées précédemment dans la classe de validation « DagValidator » générée (diapo 53). N'oubliez pas d'ajouter l''annotation « @generated NOT » dans le commentaire de la méthode pour éviter que le code soit écrasé lors de la prochaine génération.
6. Relancer une instance Eclipse de test et vérifier le bon fonctionnement des contraintes implémentées en créant des modèles invalides et en lançant la validation (diapos 37 et 60).

## TH3 – Génération de code

Pour générer le code C associé au DSL DAG, nous devons d'abord nous assurer que chaque port est connecté à au moins un canal et également que la contrainte de la figure 1 est satisfaite. Ces contraintes sont déjà implémentées dans le méta-modèle DAG dont vous disposez.

De plus, nous devons réaliser un tri topologique afin de déterminer l'ordre dans lequel exécuter les tâches. Pour cela, il faut d'abord s'assurer qu'il n'y a pas de cycles algébriques dans la définition des canaux de communication. Nous proposons de réaliser cette vérification en même temps que le tri topologique via l'algorithme décrit ci-dessous :

a. Empiler les sommets visités dans l'ordre post-fixe lors d'un parcours en profondeur à partir d'un sommet n'ayant pas de port d'entrée. Si un tel sommet n'existe pas, c'est qu'il existe un cycle algébrique.

b. Si on visite un sommet déjà visité, c'est qu'il existe un cycle algébrique.

c. Itérer jusqu'à avoir visité tous les sommets.

d. Le contenu de la pile constitue le résultat du tri topologique

Le pseudo code de cet algorithme est donné ici :

```
Soit res une pile de tâches
Soit cycle_détecté un booléen initialisé à faux.
Si toutes les tâches ont un port d'entrée, retourner vrai
Pour toute tâche r, sans port d'entrée, exécuter cycle_détecté <- BFS(r, res)
Vérifier que s contient le même nombre de tâches que le dag initial
Si cycle_détecté = vrai, retourner vrai.

      BFS(x, p)
        Marquer x comme ON_STACK
        si il existe un canal de x vers une tâche y avec y marqué ON_STACK
          retourner vrai
        tant que il existe un canal de x vers une tâche y avec y non marqué
          si BFS(y), retourner vrai
        si p ne contient pas x
          Empiler(x, p)
          Marquer x comme VISITED
        retourner faux
```

**A faire :**

Nous allons utiliser une librairie externe contenue dans le projet « fr.tpt.mem4csd.dag.sort » pour réaliser ce tri. Cette librairie utilise un modèle de graphe simple qui ne fait pas appel au méta-modèle du DAG. Il faudra donc convertir le modèle du DAG en créant un autre modèle plus simple de graphe utilisant ces classes.

1. Créer une contrainte nommée « dag » dans la classe « DagSpecification » du méta-modèle.
2. Régénérer le code du méta-modèle.
3. Implémenter la méthode de validation dag dans la classe « DagValidator » en faisant appel aux classes du projet « fr.tpt.mem4csd.dag.sort » :
    a. Créer un modèle de graphe simple à partir du modèle du DAG. Inspirez-vous de la classe « Main » du projet « fr.tpt.mem4csd.dag.sort » montrant un exemple de construction de graphe simple.

   b. Appeler la méthode « sort » de la classe « TopologicalSort » du projet « fr.tpt.mem4csd.dag.sort ». Cette méthode prend deux paramètres : le modèle de graphe simple et une pile vide qui après l'exécution de la méthode contiendra les tâches triées.
   c. La méthode « sort » retourne une valeur booléenne qui indique si le graphe contient un cycle ou non. Utiliser cette valeur pour la validation de la contrainte.
4. Lancer l'instance de test d'Eclipse.
5. Importer dans le workspace de l'instance de test d'Eclipse le projet « test_dag » de l'archive de projets fournie.
6. Valider le modèle et tester l'évaluation de votre contrainte d'absence de cycle.

En plus de valider que le modèle DAG ne contient pas de cycle, il faut également implémenter le calcul du tri des tâches afin de valoriser la propriété « Sorted Tasks » de la classe « DagSpecification » du méta-modèle DAG. Cette propriété est dérivée et il faudra donc implémenter son calcul dans la classe « DagSpecificationImpl » générée.

1. Implémenter le tri dans la méthode « getSortedTasks » de la classe DagSpecificationImpl tel que fait dans la classe de validation.
2. Relancer l'instance de test d'Eclipse.
3. Vérifier que la propriété « Sorted Tasks » est correctement valorisée dans l'éditeur de modèles DAG.

Nous allons maintenant compléter l'implémentation d'un générateur de code servant à automatiser la production de code C à partir du modèle d'un DAG de tâches et ultérieurement à le simuler. Cette génération se fera en deux étapes :

1. Génération de ce qu'on appelle le squelette du code : génération de la signature et de l'implémentation des fonctions correspondant aux tâches. L'implémentation est générée avec un corps de fonctions vide.
2. Génération de la fonction main qui invoque les implémentations des fonctions correspondant aux tâches.

### Génération du squelette de code

Nous proposons une correspondance simple entre une tâche du DSL et une fonction C : par exemple, une tâche T1 avec un port d'entrée p1 (de type « int ») et un port de sortie p2 (de type « float ») correspondra à la fonction C suivante :

```
void T1(int p1, float * p2);
```

Pour tester votre générateur de code, vous pouvez générer l'implémentation du corps des fonctions C comme suit:

- Afficher un message indiquant que la fonction de la tâche est en cours d'exécution.
- Afficher les valeurs reçues sur les ports d'entrée.
- Afficher les valeurs écrites sur les ports de sorties.
- Incrémenter les valeurs de ports de sortie de la tâche.

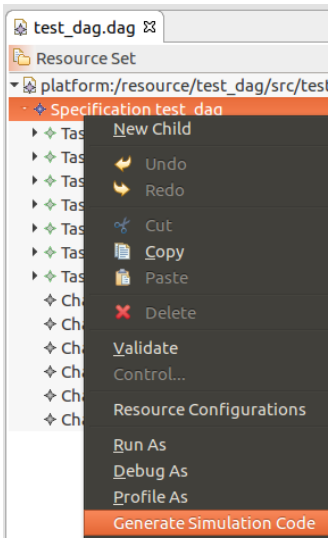
### Génération de la fonction main

La génération de la fonction main s'appuiera sur les étapes suivantes :

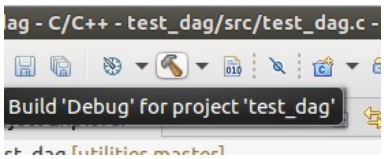
- La fonction main exécute une boucle infinie et appellera à l'intérieur de cette boucle les fonctions associées aux tâches, dans l'ordre défini par le tri topologique.
- Cette boucle infinie doit s'exécuter toutes les X ms où X est la période de base du DAG de tâches : la période de base correspond au pgcd des périodes des tâches du DAG. Chaque tâche du DAG sera activée toute les N exécutions de la boucle infinie de la fonction main, où N correspond à la division de la période de la tâche par la période de base du DAG. A la fin de la boucle, la fonction main est mise en attente jusqu'à la fin de la période de base du DAG.
- La communication entre tâches se fera en définissant une variable globale pour chaque canal de communication dont le type correspond au type des ports. Dans la fonction main, les fonctions correspondant aux tâches sont appelées en passant ces variables globales (ou leurs adresses pour les ports de sortie) en paramètre.

**A faire :**

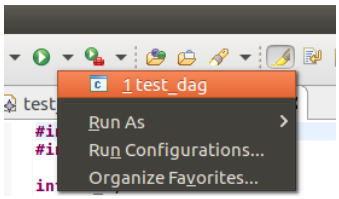
1. Importer le projet « fr.tpt.mem4csd.dag.simulator » dans le workspace Eclipse contenant les projets du méta-modèle DAG.
2. Compléter l'implémentation de la classe « CodeGenerator » du projet du simulateur (les parties à compléter sont identifiées par des « TODO TP » en commentaire dans le code) :
    a. Compléter le code de la méthode « generateDagCode » pour générer le code C de la déclaration des variables globales de chaque port des tâches.
    b. Compléter la méthode « generateTaskBody » qui génère le corps de la fonction C d'une tâche en y ajoutant la génération d'une instruction qui incrémente la valeur de chaque port de sortie de la tâche en fonction de son type :
        i. Pour les ports de type « int », incrémenter la valeur de 1.
        ii. Pour les ports de type « float », incrémenter la valeur de 0,1.
    c. Compléter le code de la méthode « generateTaskCall » :
        i. Générer l'appel à la fonction C de la tâche en cours en passant les bonnes variables globales en paramètres de la fonction.
        ii. Générer l'affectation de la valeur du port de destination du canal de communication en cours avec la valeur du port de la source.
3. Lancer l'instance de test d'Eclipse.
4. Nous allons utiliser l'outil de développement C / C++ d'Eclipse nommé CDT (C Development Tools) pour visualiser le code généré, le compiler et l'exécuter.
    a. Générer le code C de simulation du modèle DAG en exécutant le menu contextuel de génération de code tel qu'indiqué à la figure suivante :




b. Ouvrir la perspective Eclipse de développement C / C++ (voir page 3)
c. Compiler le code généré en cliquant sur le bouton (image de marteau) de la figure suivante :




d. Vérifier qu'il n'y a pas d'erreur de compilation et exécuter le simulateur en lançant l'exécutable par la configuration de lancement CDT Eclipse « test_dag ».




Pour aller plus loin : Que se passe-t-il si le temps d'exécution d'une tâche est supérieur à la période de base ? Proposez une solution pour résoudre ce problème.