BILLEBAULT Benjamin
LEBEAU Élise
SN APP 2A

Rapport de projet

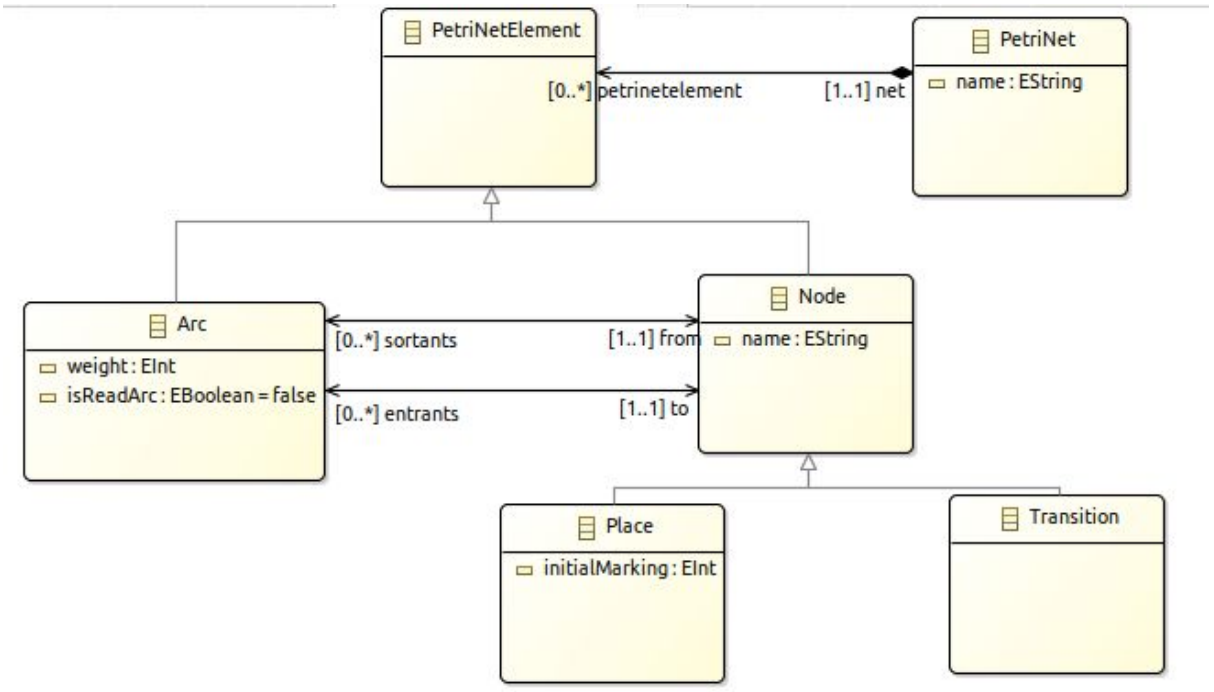
Projet IDM 2020

Table des matières

Méta-modèles

PetriNet

Le premier méta-modèle que nous avons modélisé est PetriNet. Il permet de représenter un réseau de Pétri, dont voici une image :



Étant donné que ce méta-modèle n'était pas donné, nous allons l'expliquer un peu.
Tout d'abord, l'élément principal de notre version du réseau de Pétri est la classe PetriNet. Cette dernière représente le réseau de Pétri entier, en lui attribuant un nom.
On peut constater ensuite qu'un réseau de Petri (PetriNet) est constitué de plusieurs PetriNetElement. Cette classe représente un élément du réseau de Pétri parmi tous les éléments possibles (arcs ou noeuds). Nous aurions pu nous passer de cette classe et éviter deux héritages, mais cela facilite l'utilisation du méta-modèle dans OCL.

De cette classe héritent deux sous-classes :

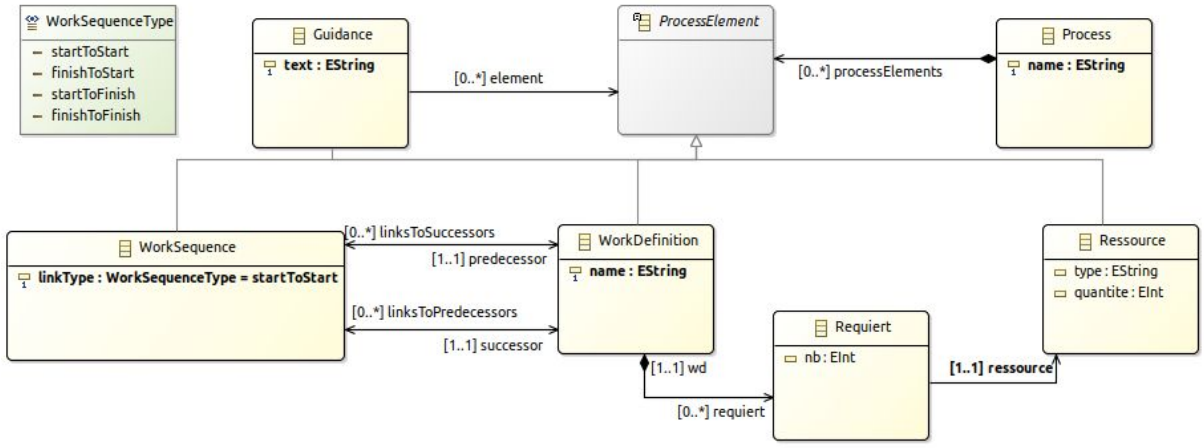
- Arc : représente un arc orienté entre une transition et une place (from et to indiquent respectivement le noeud de départ et le noeud d'arrivée de l'arc)
- Node : représente un noeud du réseau de Pétri. Nous avons considéré que tout ce qui peut être au départ ou à l'arrivée d'un arc est un noeud :
    - La classe Place représente une place dans le réseau de Pétri. En raison de son héritage de la classe Node, elle possède un nom, mais aussi un attribut initialMarking, qui indique le nombre de jetons initial dans la place représentée.
    - La classe Transition représente une transition dans le réseau de Pétri. Tout comme pour la classe Place, cette dernière possède un nom.

Nous pouvons également remarquer que nous avons fait le choix de mettre des références à double sens entre la classe Arc et la classe Node, afin de faciliter l'utilisation du méta-modèle plus tard.

Concernant les cardinalités, chaque élément d'un réseau de Pétri doit faire partie d'un et d'un seul réseau de Pétri, mais un réseau de Pétri peut être constitué de zéro, un, ou plusieurs éléments.
Il peut y avoir zéro, un, ou plusieurs arcs entrants ou sortants d'un noeud, mais un arc ne peut avoir qu'un seul noeud à chaque bout.

SimplePDL

Le second méta-modèle sur lequel nous nous sommes penchés est SimplePDL. Ce dernier était donné sans la partie sur les ressources, que nous avons dû ajouter pour les besoins de ce projet. Voici une image de notre solution :



Nous avons donc rajouté :

- La classe Requiert
- La classe Ressource
- L'association "requiert" entre WorkDefinition et Requiert
- L'association "ressource" entre Requiert et Ressource

Tout d'abord, il est important de noter que comme pour le méta-modèle PetriNet, nous avons choisi de faire hériter Ressource de la classe ProcessElement afin de faciliter le traitement dans OCL. Nous considérons donc qu'une ressource est un élément d'un processus.

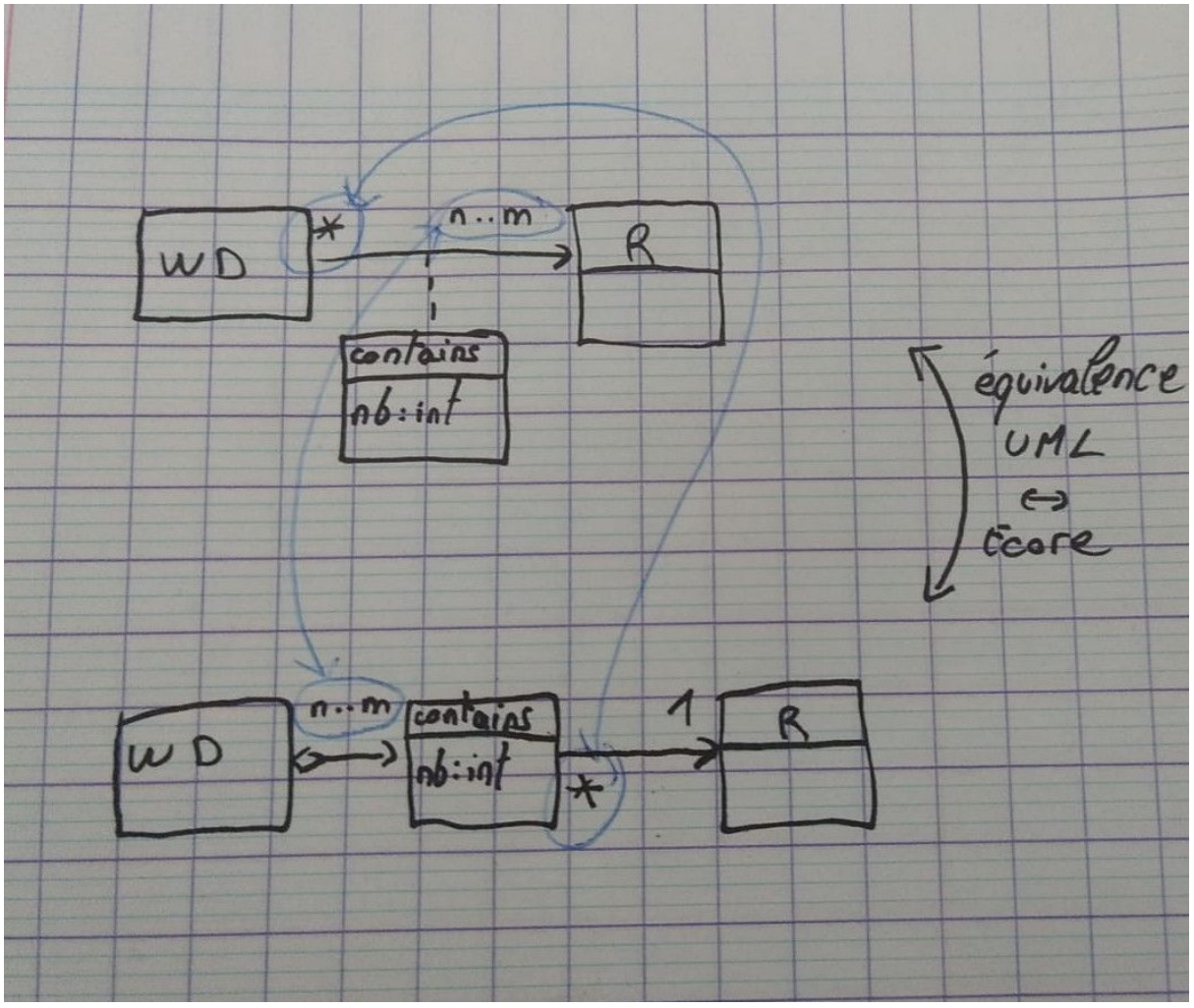
Une ressource est d'un certain type (nous avons attribué le type EString à cet attibut). Dans l'exemple du sujet, cet attribut type peut valoir "concepteur", "rédacteur", "machine", "développeur", ou "testeur" (nous n'avons pas mis de contraintes de nommage des ressources, mais une convention intéressante serait de mettre les noms de ressource en minuscule, sans accents ni caractères spéciaux, et en remplaçant les espaces par des underscores).

Une ressource est présente en certaine quantité. Dans l'exemple du sujet, nous avons 3 concepteurs, 2 développeurs, etc. Cette quantité sera donc inscrite dans cet attribut.

La relation de composition entre ProcessElement et Process permettra de décrire les ressources allouées à un processus.

Un processus a ensuite besoin de ces ressources pour réaliser certaines activités (WorkDefinition). Pour représenter ce besoin, nous avions imaginé une classe d'association entre WorkDefinition et Ressource pour pouvoir stocker la quantité de la ressource nécessaire. Cependant, le concept de classe d'association d'existant pas dans le

méta-méta-modèle Ecore, nous avons dû créer la classe Requiert, qui est l'équivalent de cette classe d'association. Voici une illustration de cette transformation, donnée par notre professeur, Nicolas Hili :



Ainsi, pour chaque ressource nécessaire à l'accomplissement d'une activité, il y aura une classe Requiert créée, qui indiquera la quantité nécessaire d'une ressource. De l'autre côté, cette classe contient une référence à la ressource pointée.

Attention ! La classe Ressource est utilisée non seulement pour décrire le processus en indiquant de quelles ressources on dispose et en quelle quantité, mais elle est aussi utilisée par la classe Requiert pour décrire les ressources nécessaires pour accomplir une activité. Dans ce cas, il ne faut pas oublier que l'attribut "quantite" continue de représenter la quantité de la ressource dont on dispose dans le processus, et non pas la quantité de ressource nécessaire pour la WorkDefinition !

Contraintes OCL

Les méta-modèles définis plus haut ne permettent malheureusement pas d'assurer totalement la création de modèles valides. Les contraintes OCL complètent donc les méta-modèles.
Outre les quelques contraintes OCL "évidentes" telles que la validité des noms sur les différents éléments qui ne peut être null ou doit être unique, nous avons ajouté également un certain nombre de contraintes que l'on espère exhaustive pour valider totalement les modèles.
Pour chacune des règles que nous avons ajoutée, nous avons créé un modèle contre-exemple. Ces contre-exemple sont présents dans les dossiers be.petrinet.exemple et be.simplepdl.exemple et les noms de fichiers sont de type "<simplepdl | petrinet>_err_<type de l'erreur>"
Les fichiers dont le nom ne suit pas ce formalisme sont des modèles à priori valides.
Pour tous les fichiers contre-exemple, nous somme partis d'un modèle que l'on sait valide en ne modifiant que le strict minimum afin de ne lever qu'une seule erreur à la fois.

SimplePDL

- On s'assure donc de la validité et de l'unicité des noms
- On s'assure que le prédécesseur et le successeur d'un WorkSequence sont bien dans le même process
- On vérifie qu'un WorkSequence n'est pas réflexif. En effet cela n'aurait pas de sens d'avoir une dépendance startToStart par exemple qui boucle sur une WorkDefinition.
- Pour chaque ressource, on s'assure qu'elle existe en quantité non négative et que son type soit unique.
- On s'assure enfin que chaque WorkDefinition ayant besoin d'une ressource en requiert une quantité positive ET au plus la quantité déclarée dans la ressource concernée.

PetriNet

- On s'assure là encore de l'unicité et de la validité des noms des différents éléments
- On vérifie que chaque Arc possède un poids strictement positif
- On a pensé à vérifier qu'un Arc a bien un prédécesseur et un successeur. En effet, on ne peut avoir un arc qui pointe nulle part dans un réseau de pétri. Cependant, on s'est rendu compte après coup que cette contrainte était déjà existante de par les cardinalités présentes dans le méta-modèle. On a laissé la contrainte OCL mais elle n'est donc finalement pas nécessaire.
- On s'assure surtout que les arcs ne puissent pas passer d'une Transition à une autre ou d'une Place à une autre car ce n'est pas valide dans un réseau de pétri.
- On vérifie également qu'un arc provenant d'une Transition ne soit pas un readArc.

- On vérifie qu'une place a un marquage initial au moins égal à 0, et qu'elle a au moins soit un arc entrant, soit un arc sortant.
- Les transitions ne pouvant pas non plus être isolées, elle doivent quant-à elles posséder au moins un arc entrant ET un arc sortant

Transformation SimplePDL → PetriNet avec ATL

Pour transformer un modèle SimplePDL en modèle PetriNet, nous utilisons plusieurs règles de transformation :

- Tout d'abord, pour chaque Ressource, nous créons une place, dont le nombre de jetons initial correspond à la quantité de cette ressource.

- Ensuite, pour chaque WorkDefinition, nous créons 3 places (ready, running, finished) et deux transitions (start, finish). Cela nous permet de représenter tous les états possibles d'une activité. Il y a toujours un jeton dans la place "ready" au départ, et aucun jeton dans les deux autres. Nous créons finalement 4 arcs :
    - L'un va de la place "ready" à la transition "start" et a un poids de 1.
    - L'un va de la transition "start" à la place "running" et a un poids de 1.
    - L'un va de la place "running" à la transition "finish" et a un poids de 1.
    - L'un va de la transition "finish" à la place "finished" et a un poids de 1.

  Ainsi, une activité possède un jeton dans la place correspondant à son état actuel (en attente, en cours, terminée).

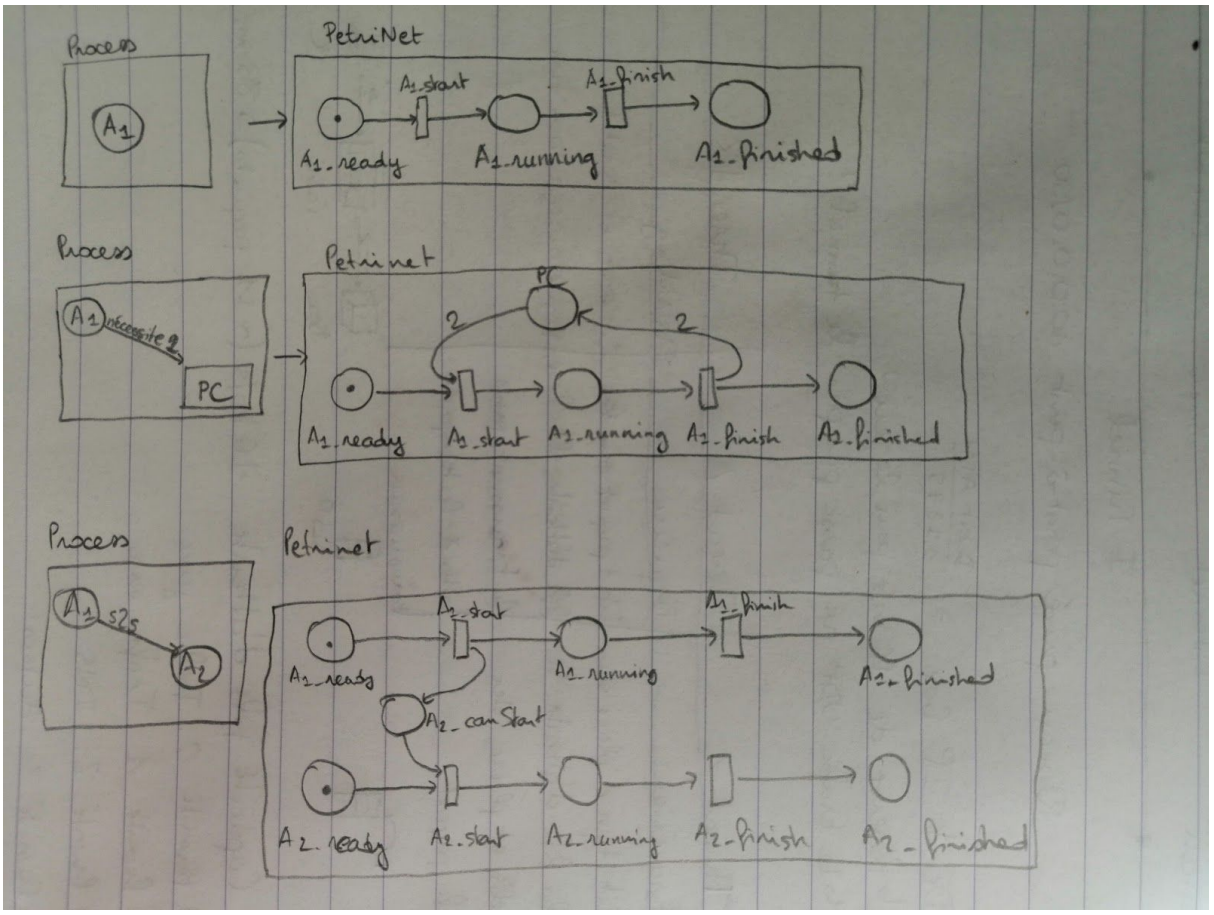
  Par la suite, pour chaque ressource requise par l'activité, nous créons un arc allant de la place représentant la ressource à la transition "start" de l'activité, avec un poids correspondant à la quantité de cette ressource nécessaire (attribut nb de la classe Requiert). Un arc dans le sens inverse est créé, partant de la transition finish de l'activité vers la place correspondant à la ressource, et ayant le même poids. Ces arcs permettent respectivement d'emprunter la ressource et de la rendre.

- Ensuite, dans le cas où cette activité a besoin qu'au moins une autre activité ait démarré (startToStart) ou terminé (finishToStart), nous créons une place "canStart" avec un nombre de jetons initial de 0. Nous créons également un arc allant de cette place à la transition canStart de l'activité, avec un poids égal au nombre d'activités dont le démarrage de cette activité dépend.

- Nous effectuons la même chose pour dans le cas où l'activité a besoin qu'une autre activité ait démarré (startToFinish) ou terminé (finishToFinish) pour terminer. La place créée sera donc nommée "canFinish". Il est important de rappeler que dans notre version de la transformation, ces places et arcs ne sont créés que s'il y en a besoin. Il n'y aura pas de place canStart ou canFinish sans arc entrant.

- Pour chaque WorkSequence, nous faisons le lien via un arc depuis la transition de l'activité dont dépend l'activité à l'arrivée et la place concernée. Par exemple, si la terminaison de l'activité A dépend du démarrage de l'activité B, et donc qu'on a une dépendance de type "startToFinish", on aura un arc qui part de la transition B_start, et arrivant dans la place A_canFinish.

La transformation est résumée en 3 étapes dans ce schéma :



Transformation PetriNet → Tina avec Acceleo

Depuis un fichier petrinet, il est aisé d'obtenir un fichier tina (.net) grâce à Acceleo.

On commence par indiquer dans le fichier cible le nom de ce réseau de pétri grâce à " net [aPetriNet.name/]"

Ensuite, il faut indiquer chaque place. Pour cela, on veut obtenir une ligne par place de sorte que le nom de chaque place soit préfixé par "pl " et suivi, entre parenthèses, de son marquage initial . On a donc une première boucle sur les places que l'on récupère grâce à une query getPlaces.

Il faut ensuite indiquer quelles sont les transitions. On boucle donc sur les transitions récupérées grâce à la query getTransitions qui renvoie l'ensemble des transitions du réseau de pétri.
Dans un fichier tina, une transition est indiqué sur une ligne préfixée par "tr". On indique ensuite son nom. Puis on doit indiquer chaque place précédant cette transition avec "?" puis le poids de l'arc si l'arc est un read-arc ; ou "*" suivi du poids de l'arc si c'est un arc simple. Pour faciliter cet affichage, nous avons créé un template getFromPlaces(arcs) qui renvoie la chaîne de caractères correspondant aux places entrantes de chacun des arcs entrants dans la transition, ces derniers étant récupérés grâce à une query getArcsEntrants.
Enfin, il en est fait de même pour les places sortantes de la transition.

Transformation processus “Développement” en réseau de pétri