Rapport de BE

Ingénierie Dirigée par les Modèles

DANIS Émilie
DALBIN Mickaël
2020

ENSEEIHT - SN App 2A

Sommaire

Introduction

Le projet de BE avait pour objectif d'établir une chaîne de transformation afin de passer d'un modèle de SimplePDL à un réseau de Petri interprétable avec l'outil Tina. Ce BE devait nous permettre de manipuler différents outils de modélisations et de transformation. Malheureusement, la crise du Covid-19 nous aura privé du temps nécessaire pour la mise en pratique de plusieurs de ces outils (Sirius, Xtext, etc.).
Le projet a été réalisé à partir d'Eclipse Modeling Tools, qui est une version de l'IDE Eclipse comportant des outils de modélisations intégrés. Nous verrons que l'outil en lui-même a été une grande source de difficultés pour la réalisation du BE.

Le BE s'est déroulé en plusieurs étapes. Il fallait tout d'abord créer les méta-modèles. Ensuite, il a fallu valider certaines contraintes qui ne peuvent être exprimées par le méta-modèle en lui-même, via l'outil OCL. L'étape suivante consistait à transformer un modèle SimplePDL en réseau de Petri grâce à une transformation ATL. Enfin, il fallait pouvoir représenter graphiquement ce réseau de Petri en utilisant Acceleo, qui permettait de faire une transformation vers un langage interprétable par l'outil Tina.

Conception des méta-modèles

SimplePDL

Un SimplePDL représente un processus de production, de manière générale. Durant le reste du rapport, le modèle de SimplePDL qui nous servira de fil conducteur s'inspire du modèle suivant (tiré du sujet du BE), auquel on ajoutera une ressource "développeur".

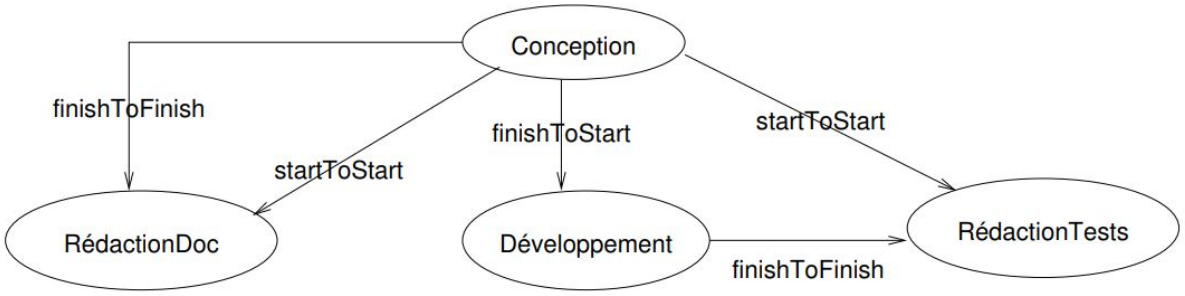


Figure 1 - Exemple de SimplePDL

Pour établir le méta-modèle d'un SimplePDL, nous avons repris celui réalisé durant les TPs. Nous avons ensuite dû y ajouter le concept de ressource. Une ressource peut représenter tout ce qui est nécessaire à la réalisation d'une tâche (ou *WorkDefinition*). Cela peut-être des personnes ou des machines par exemple. Ainsi, nous avons décidé de créer deux nouvelles classes pour compléter le méta-modèle.

La première est la classe *Resource* qui représente un certain type de ressource (par exemple un développeur). Elle hérite de la classe abstraite *ProcessElement* car elle est représente un élémént du processus. Elle possède un attribut *name* qui est son nom, et un attribut *nbAvailableResources* qui représente le nombre de ressources de ce type disponibles dans le processus.

La seconde classe ajoutée est la classe *Requirement*. Celle-ci représente une condition de ressource pour effectuer une tâche. Par exemple, une *WorkDefinition* peut nécessiter deux développeurs et deux machines, et deux instances de *Requirement* seront alors nécessaires. La classe *Requirement* référence la classe *Resource* car elle désigne un type de ressource. Réciproquement, la classe *Ressource* référence la classe *Requirement*, mais elle n'a pas de limite d'instances associées. La classe *Requirement* possède également un attribut *nbResources* qui désigne le nombre de ressources du même type nécessaires. Elle est également associée à une seule *WorkDefinition* et une *WorkDefinition* peut avoir plusieurs instances de *Requirement* associées.

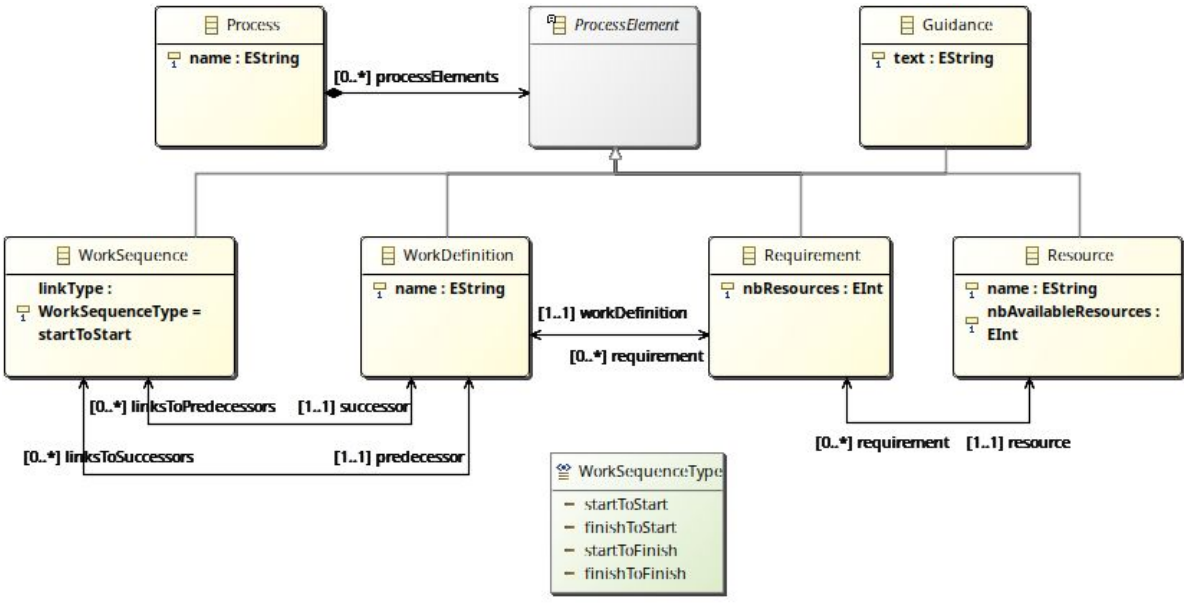


Figure 2 - Méta-modèle de SimplePDL

PetriNet

Un réseau de Petri est composé de places, de transitions et d'arcs. Pour établir notre méta-modèle, nous avons donc décidé de créer une classe *PetriNet* représentant le réseau de Petri dans sa globalité. Celui-ci contient un nombre quelconque de *PetriElement*. Cette classe est abstraite car elle représente tout élément du réseau de Petri, mais ne peut être instanciée directement.

Un *PetriElement* peut donc être un arc, représenté par la classe *Arc*, qui contient un attribut de type entier relatif au poids de l'arc (*weight*) et un attribut de type booléen qui détermine si l'arc est un readarc ou non (*isReadArc*). Un *Arc* possède une origine et une destination, représentés par les attributs *source* et *target* qui référencent la classe abstraite *Node*.

La classe *Node* est une classe abstraite qui représente une place ou une transition. Elle a pour attribut un nom (*name*) ainsi que ses arcs entrants (*incoming*) et sortants (*outgoing*).

Les classes *Transition* et *Place* héritent directement de la classe *Node*. La première n'a pas d'attribut supplémentaire. La seconde comporte un marquage initial (*marking*) qui représente le nombre de jetons initialement présents dans la place.

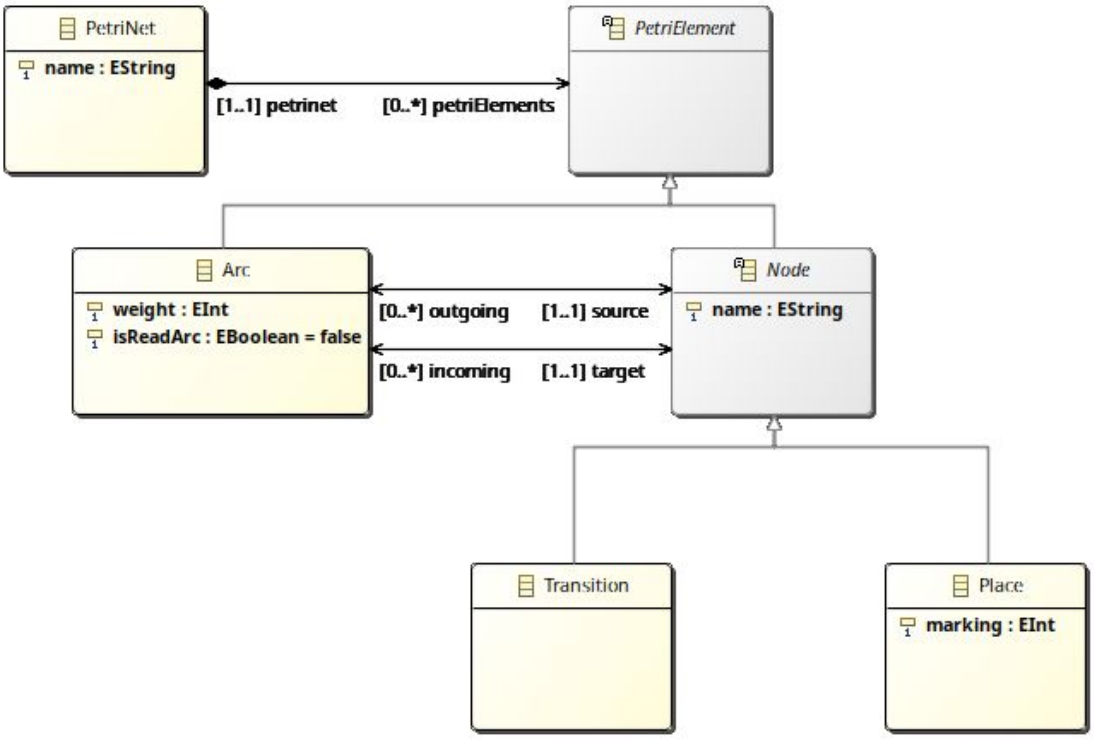


Figure 3 - Méta-modèle d'un réseau de Petri

Validation des méta-modèles avec OCL

OCL permet d'imposer certaines règles supplémentaires sur un méta-modèle, qui ne peuvent en général pas être représentées autrement. Ci-dessous sont détaillées les règles qui complètent les méta-modèles vus plus haut.

SimplePDL

```
context ProcessElement
def: process(): Process =
    Process.allInstances()
        ->select(p | p.processElements->includes(self))
        ->asSequence()->first()
```

Ceci n'est pas une règle, mais une définition qui nous servira pour d'autres règles. Elle permet de récupérer le processus associé à partir de n'importe quelle instance de *ProcessElement*.

```
context Process
inv validName('Invalid name: ' + self.name):
    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
```

Cette règle vérifie que le nom du processus commence par une lettre et est composé seulement de lettres et chiffres.

```
context WorkSequence
inv successorAndPredecessorInSameProcess('Activities not in the same process : '
       + self.predecessor.name + ' in ' + self.predecessor.process().name+ ' and '
       + self.successor.name + ' in ' + self.successor.process().name
):
    self.process() = self.successor.process()
    and self.process() = self.predecessor.process()
```

Cette règle vérifie que toutes les instances de *ProcessElement* reliées appartiennent bien au même processus.

```
context WorkSequence
inv distinctSuccessorAndPredecessor(
'Successor (' + self.successor.name + ') and predecessor (' + self.predecessor.name + ') are the same'):
    self.predecessor <> self.successor
```

Cette règle vérifie qu'aucune instance de *WorkDefinition* n'est reliée à elle-même.

```
context WorkDefinition
inv uniqueName ('Name "' + self.name + '" already exists.'):
    self.process().processElements
    ->select(e | e.oclIsKindOf(WorkDefinition))
    ->collect(e | e.oclAsType(WorkDefinition))
    ->isUnique(name)
```

Cette règle vérifie que chaque nom d'instance de *WorkDefinition* dans un même processus est unique. Nous aurions pu utiliser l'attribut "unique" directement sur le méta-modèle, mais en utilisant cette règle OCL à la place, il est possible d'avoir deux instances de *WorkDefinition* de même nom tant qu'elles ne sont pas dans le même processus. Nous avons donc jugé que c'était plus adapté.

PetriNet

```
context PetriElement
def: net(): PetriNet =
    PetriNet.allInstances()
        ->select(n | n.petriElements->includes(se
        ->asSequence()->first()
```

Comme vu en SimplePDL, ceci n'est pas une règle, mais une définition qui nous servira pour d'autres règles. Elle permet de récupérer le réseau de Petri associé à partir de n'importe quelle instance de *PetriElement*.

```
context PetriNet
inv validNetName('Invalid name: ' + self.name):
    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
```

Cette règle vérifie que le nom du réseau de Petri commence par une lettre et est composé seulement de lettres et chiffres.

```
context Arc
inv isBetweenPlaceAndTransition:
    (self.source.oclIsKindOf(Place)
     and self.target.oclIsKindOf(Transition))
     or (self.target.oclIsKindOf(Place)
     and self.source.oclIsKindOf(Transition))
```

Cette règle vérifie que tous les arcs relient bien une place et un transition, quel que soit le sens.

```
context Arc
inv hasPositiveWeight:
    self.weight > 0
```

Cette règle vérifie que les arcs ont un poids d'au moins un.

```
context Place
inv hasPositiveOrNullMarking:
    self.marking >= 0
```

Cette règle vérifie que les places n'ont pas un nombre de jeton négatif, ce qui n'aurait pas de sens dans le contexte d'un réseau de Petri.

```
context Node
inv validNodeName('Invalid name: ' + self.name):
    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
```

Cette règle fait la même vérification que pour le nom du réseau de Petri, mais au niveau de chaque instance de Node (place ou transition).

```
context Node
inv uniqueName ('Name "' + self.name + '" already exists.'):
    self.net().petriElements
    ->select(e | e.oclIsKindOf(Node))
    ->collect(e | e.oclAsType(Node))
    ->isUnique(name)
```

Cette règle vérifie que chaque nom d'instance de Node dans un même réseau de Petri est unique, de la même façon que la règle OCL expliquée pour les *WorkDefinition* de SimplePDL.

Transformation SimplePDL vers PetriNet avec ATL

Afin de transformer un modèle SimplePDL en un réseau de Petri, nous avons utilisé l'outil ATL.

```
-- @path SimplePDL=/SimplePDL/SimplePDL.ecore
-- @path PetriNet=/PetriNet/PetriNet.ecore
module SimplePDL2PetriNet;
create OUT: PetriNet from IN: SimplePDL;

helper context SimplePDL!ProcessElement
def: process(): SimplePDL!Process =
    SimplePDL!Process.allInstances()
        ->select(p | p.processElements->includes(self))
        ->asSequence()->first();

rule Process2PetriNet {
    from p: SimplePDL!Process
    to pn: PetriNet!PetriNet (
            name <- p.name,
            petriElements <- p.processElements
        )
}
```

Une transformation ATL prend en entrée un modèle de SimplePDL et fournit en sortie un réseau de Petri.

La méthode *process()* permet de récupérer l'instance de *Process* associée à un *ProcessElement* dans un modèle SimplePDL.

```
rule WorkDefinition2PetriNet {
    from wd: SimplePDL!WorkDefinition
    to
        placeReady: PetriNet!Place(
            name <- wd.name + '_ready',
            marking <- 1,
            petrinet <- wd.process()
        ),
        placeRunning: PetriNet!Place(
            name <- wd.name + '_running',
            petrinet <- wd.process()
        ),
        placeFinished: PetriNet!Place(
            name <- wd.name + '_finished',
            petrinet <- wd.process()
        ),
        placeHasStarted: PetriNet!Place(
            name <- wd.name + '_hasStarted',
            petrinet <- wd.process()
        ),
        placeHasFinished: PetriNet!Place(
            name <- wd.name + '_hasFinished',
            petrinet <- wd.process()
        ),
        transitionStarted : PetriNet!Transition (
            name <- wd.name.concat('_started'),
            petrinet <- wd.process()
        ),
        transitionEnded : PetriNet!Transition (
            name <- wd.name.concat('_ended'),
            petrinet <- wd.process()
        ),
```

La première transformation à appliquer est celle d'un *Process* SimplePDL en un *PetriNet*. Le nom du *Process* est passé au *PetriNet* et les instances de *ProcessElement* deviennent des instances de *PetriElement*.

Ensuite, il faut transformer les instances de *WorkDefinition* en *PetriElement*. On se rend tout de suite compte qu'à une instance de *WorkDefinition* ne pourra pas correspondre un seul *Node* de réseau de Petri. Après avoir étudié le problème, nous avons conclu qu'une instance de *WorkDefinition* devait être associée à cinq places, deux transitions et les six arcs les reliant.

En ce qui concerne les places, il doit y en avoir une par état de la *WorkDefinition*. Celle-ci peut être prête, en exécution ou terminée. Mais à cela, on doit ajouter deux places qui permettent de signifier si une place a démarré ou si elle est terminée. C'est en effet très important pour conserver le lien des *WorkSequence* qui nécessite de savoir si une tâche à été démarrée ou terminée pour pouvoir en démarrer ou terminer une autre. Les transitions correspondent au démarrage ou à la terminaison d'une *WorkDefinition*.
Les arcs, quant à eux, relient les places aux transitions adéquates.

```
aPRe2TS : PetriNet!Arc (
    petrinet <- wd.process(),
    isReadArc <- false,
    weight <- 1,
    source <- thisModule.resolveTemp(wd, 'placeReady'),
    target <- thisModule.resolveTemp(wd, 'transitionStarted')
),
aTS2PHS : PetriNet!Arc (
    petrinet <- wd.process(),
    isReadArc <- false,
    weight <- 1,
    source <- thisModule.resolveTemp(wd, 'transitionStarted'),
    target <- thisModule.resolveTemp(wd, 'placeHasStarted')
),
aTS2PRu : PetriNet!Arc (
    petrinet <- wd.process(),
    isReadArc <- false,
    weight <- 1,
    source <- thisModule.resolveTemp(wd, 'transitionStarted'),
    target <- thisModule.resolveTemp(wd, 'placeRunning')
),
aPRu2TE : PetriNet!Arc (
    petrinet <- wd.process(),
    isReadArc <- false,
    weight <- 1,
    source <- thisModule.resolveTemp(wd, 'placeRunning'),
    target <- thisModule.resolveTemp(wd, 'transitionEnded')
),
```

```
aTE2PF : PetriNet!Arc (
    petrinet <- wd.process(),
    isReadArc <- false,
    weight <- 1,
    source <- thisModule.resolveTemp(wd, 'transitionEnded'),
    target <- thisModule.resolveTemp(wd, 'placeFinished')
),
aTE2PHF : PetriNet!Arc (
    petrinet <- wd.process(),
    isReadArc <- false,
    weight <- 1,
    source <- thisModule.resolveTemp(wd, 'transitionEnded'),
    target <- thisModule.resolveTemp(wd, 'placeHasFinished')
)
```

Une *WorkSequence* va être représentée par un readarc, reliant une place *placeHasFinished* ou *placeHasStarted* vers une transition *transitionEnded* ou *transitionStarted.* En effet, dans ces types de places, le jeton n'a pas vocation à être consommé, car une fois la tâche commencée ou terminée, il n'y a pas de raison de ne plus la considérer comme telle une fois le jeton utilisé par une transition.

```
rule WorkSequence2Arc {
    from
        ws: SimplePDL!WorkSequence
    to
        ws2a : PetriNet!Arc (
            isReadArc <- true,
            source <- thisModule.resolveTemp(ws.predecessor,
                if ((ws.linkType = #finishToStart) or (ws.linkType = #finishToFinish))
                then 'placeHasFinished'
                else 'placeHasStarted'
                endif),
            target <- thisModule.resolveTemp(ws.successor,
                if ((ws.linkType = #finishToStart) or (ws.linkType = #startToStart))
                then 'transitionStarted'
                else 'transitionEnded'
                endif),
            weight <- 1,
            petrinet <- ws.process()
        )
}
```

```
rule Resource2Place {
    from
        resource: SimplePDL!Resource
    to
        pResource: PetriNet!Place (
            petrinet <- resource.process(),
            name <- resource.name,
            marking <- resource.nbAvailableResources
        )
}
```

Concernant les instances de *Resource*, elles sont représentées par des places dont le nom correspond à celui de l'instance et le marquage initial reprend le nombre de ressources disponibles pour ce type de ressource.

Les places représentant les ressources sont reliées aux transitions de commencement et de terminaison des *WorkDefinition* par des arcs qui représentent les instances de *Requirement*. Le jeton de la ressource est consommé par la transition de commencement, puis restitué par la transition de terminaison, une fois que la *WorkDefinition* est déclarée terminée. Cela rend de ce fait la ressource à nouveau disponible.

```
rule Requirement2Arc {
    from
        requirement: SimplePDL!Requirement
    to
        aStarted:PetriNet!Arc (
            isReadArc <- false,
            weight <- requirement.nbResources,
            petrinet <- requirement.workDefinition.process(),
            source <- thisModule.resolveTemp(requirement.resource, 'pResource'),
            target <- thisModule.resolveTemp(requirement.workDefinition, 'transitionStarted')
        ),
        aFinished:PetriNet!Arc (
            isReadArc <- false,
            weight <- requirement.nbResources,
            petrinet <- requirement.workDefinition.process(),
            source <- thisModule.resolveTemp(requirement.workDefinition, 'transitionEnded'),
            target <- thisModule.resolveTemp(requirement.resource, 'pResource')
        )
}
```

Après toutes ces transformations, on pouvait, via une configuration d'exécution, transformer un modèle SimplePDL en un réseau de Petri.

Transformation PetriNet vers Tina avec Acceleo

La transformation d'un modèle PetriNet vers le format texte interprétable par Tina a été la dernière étape de ce projet. Nous avons utilisé Acceleo pour la réaliser.

Génération du fichier textuel

```
[query public getPlaces(elements : OrderedSet(PetriElement)) : OrderedSet(Place) =
    elements
    ->select(n | n.oclIsTypeOf(Place))
    ->collect(n | n.oclAsType(Place))
    ->asOrderedSet()/]

[query public getTransitions(elements : OrderedSet(PetriElement)) : OrderedSet(Transition) =
    elements
    ->select(n | n.oclIsTypeOf(Transition))
    ->collect(n | n.oclAsType(Transition))
    ->asOrderedSet()/]
```

Nous avons créé deux méthodes utilitaires afin d'aider à la rédaction du template principal. La première, *getPlaces*, récupère toutes les places parmi une liste d'instances de *PetriElement*.
La seconde, *getTransitions*, réalise la même opération pour récupérer des transitions.

```
[template public Petrinet2Tina(aPetriNet : PetriNet)]
[comment @main/]
[file (aPetriNet.name.concat('.net'), false, 'UTF-8')]
[for (pl : Place | aPetriNet.petriElements->getPlaces())]
pl [pl.name/] ([pl.marking/])
[/for]
[for (tr : Transition | aPetriNet.petriElements->getTransitions())]
tr [tr.name/] [tr.addSource()/]-> [tr.addTarget()/]
[/for]
[/file]
[/template]
```

On peut voir ici le squelette principal de la transformation. Tout d'abord, il faut lister toutes les places existantes avec la syntaxe appropriée pour Tina. Pour cela, nous utilisons la méthode utilitaire *getPlaces* mentionnée plus haut. Ensuite, nous réalisons la même opération pour les transitions. La spécificité ici est qu'il faut, pour chaque transition, indiquer les places sources à gauche de la flèche, et les places destination à droite.
Pour les connaître, nous avons écrit deux sous-templates.

```
[template public addSource(tr : Transition)]
[for (a : Arc | tr.incoming)][if (not a.isReadArc)][if (a.weight > 1)][a.source.name/]*[a.weight/]
[else][a.source.name/] [/if][elseif (a.isReadArc)][a.source.name/]?[a.weight/] [/if][/for]
[/template]

[template public addTarget(tr : Transition)]
[for (a : Arc | tr.outgoing)][if (a.weight > 1)][a.target.name/]*[a.weight/] [else][a.target.name/]
[/if][/for]
[/template]
```

Le premier, *addSource*, récupère tous les arcs entrants d'une transition et les écrit avec la syntaxe appropriée, en fonction de s'ils sont des readarc ou non.
Le second, *addTarget*, réalise une opération similaire pour les arcs sortants. La différence est qu'on ne peut pas avoir ici de readarc.

L'application de ce template sur le modèle PetriNet a donné le fichier .net suivant.

Représentation graphique

Comme le contenu textuel du fichier .net n'est pas forcément très lisible, nous avons réalisé une représentation visuelle de deux modèles convertis de SimplePDL vers .net avec l'outil nd de Tina.

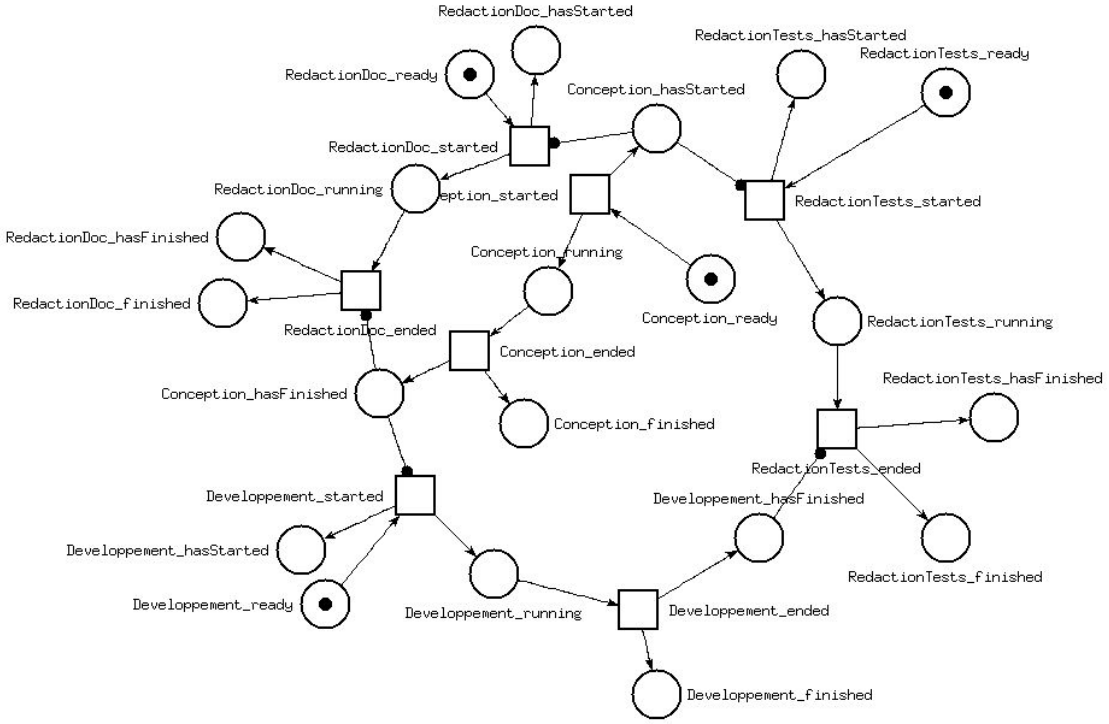


Figure 4 - Modèle "fil rouge" sans ressources

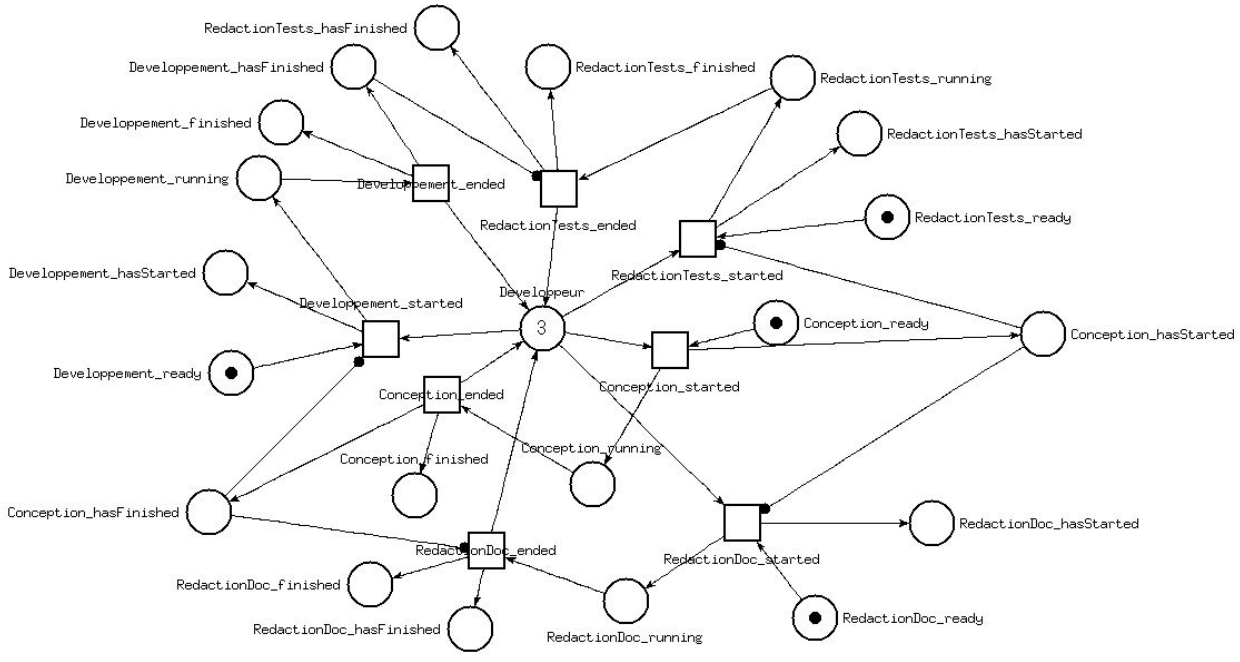


Figure 5 - Modèle "fil rouge" avec ressources

Problèmes rencontrés avec Eclipse

Nous avons rencontré tout au long de ce projet de nombreux problèmes avec Eclipse, certains dus à des mauvaises manipulations de notre part, d'autres dont nous n'avons toujours pas compris la source.

Présentation orale

Comme vous devez vous en souvenir, nous avons eu de gros soucis le jour de la présentation orale. Ce n'est que le lendemain que nous avons réussi à trouver la source du problème qui a empêché le fonctionnement de beaucoup de choses, entre autres la transformation avec Acceleo de PetriNet à Tina.
Il s'est avéré que, alors que la veille encore il était correct, le méta-modèle *PetriNet* contenait le package *PetriNet*, qui lui-même contenait les éléments du méta-modèle (*PetriNet*, *PetriElement*, *Place*, *Transition*…) ainsi qu'un sous package *PetriNet* avec également tous les éléments du méta-modèle à l'intérieur. Nous n'avons toujours pas élucidé le mystère de cette duplication, que nous n'avons malheureusement pas remarqué à temps avant la présentation orale.
Le lendemain, nous avons donc recommencé le projet à partir de zéro (en gardant bien sûr le contenu des fichiers .ocl, .atl et .mtl) dans un nouveau workspace vide et en ayant au préalable désinstallé tous les greffons. Après quelques essais, tout a fonctionné correctement.

Validation OCL

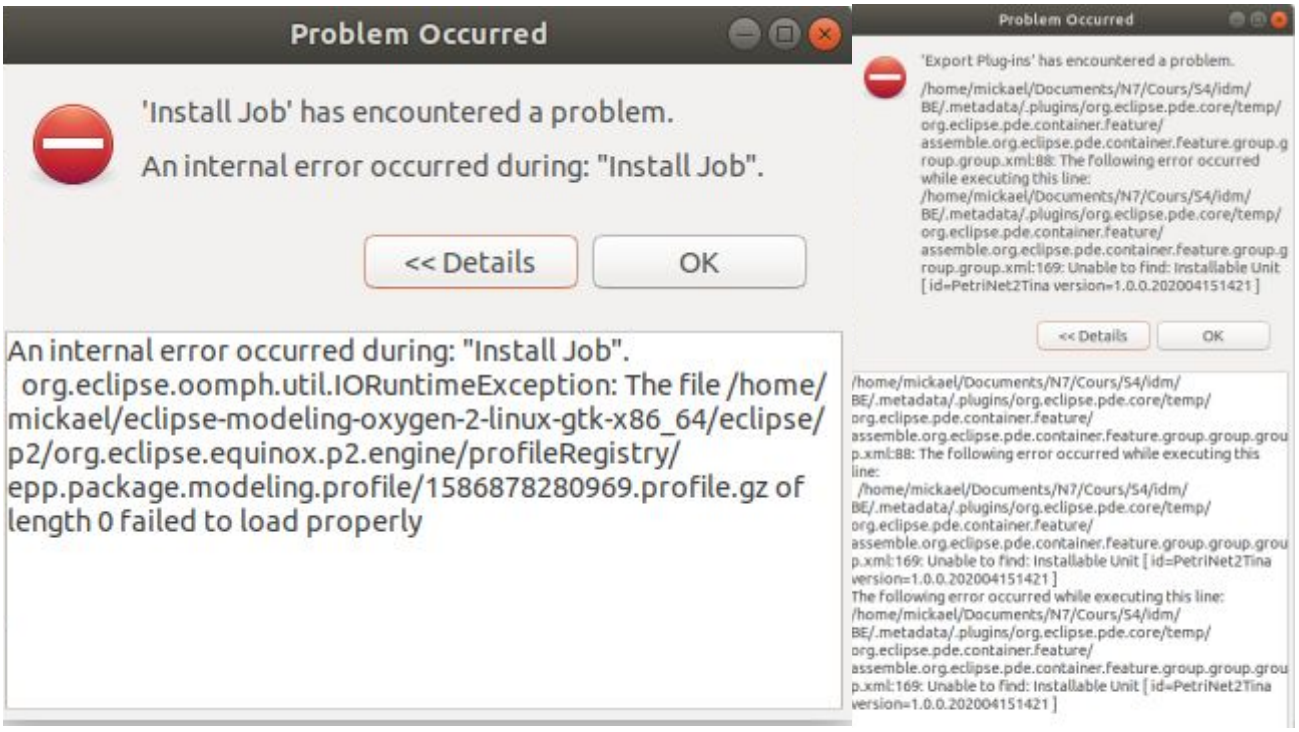
La validation OCL fonctionnait bien au début du projet, mais après le déploiement des différents greffons nécessaires pour continuer le projet, elle n'était plus possible. Pour pallier ce problème, nous avons créé un workspace séparé contenant seulement les méta-modèles basiques et non déployés, et où la validation OCL fonctionnait donc.

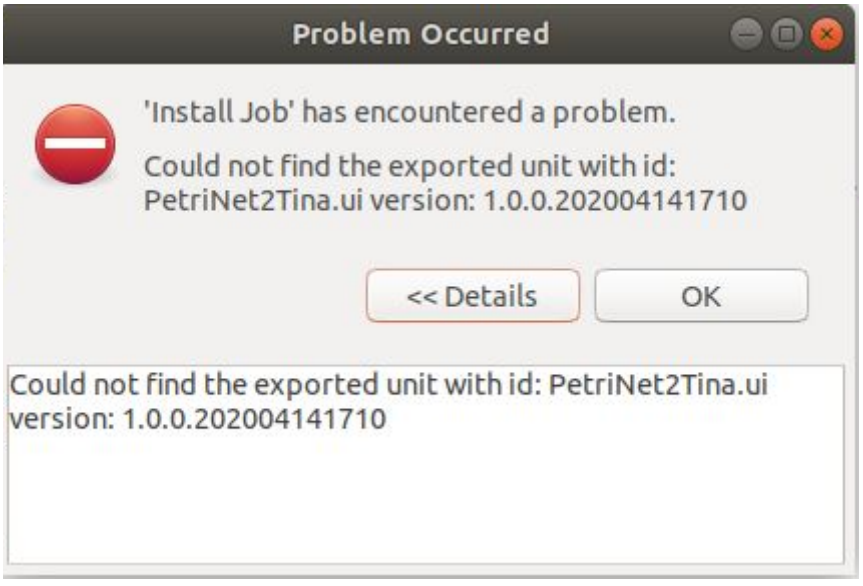
Projet ATL

Nous avons perdu beaucoup de temps lors de l'exécution de la transformation ATL. En effet, elle ne fonctionnait pas car elle ne trouvait vraisemblablement pas le méta-modèle *PetriNet*. Cela semblait incohérent, car elle trouvait bien le SimplePDL que nous avions fait exactement de la même manière.
Nous nous sommes donc rendus compte que, dans le méta-modèle *PetriNet*, l'élément *PetriElement* existait en double, mais seul l'un des deux était visible sur la représentation graphique, ce qui nous avait empêché de le remarquer. L'un des deux *PetriElement* était relié au *PetriNet*, tandis que l'autre était celui dont héritaient

les places, transitions et arc. Après avoir corrigé le méta-modèle, la transformation ATL a fonctionné correctement. Cependant, nous n'avons toujours aucune idée de comment ce problème est arrivé.

Pour compléter cette page blanche, nous vous proposons quelques captures d'écrans du florilège de messages d'erreurs que nous a donnés notre ami Eclipse.





Conclusion

En conclusion, ce projet nous aura appris la persévérance, fait perdre quelques cheveux et aura donné quelques sueurs froides à nos machines en voyant dangereusement s'approcher les battes de baseball. Malgré cela, il a été particulièrement satisfaisant de voir notre travail (tardivement) couronné de succès.

Il a été intéressant de voir toute les étapes de la transformation d'un modèle en un autre modèle d'un tout autre type. Cela nous a permis de mettre en pratique les aspects théoriques du cours et d'utiliser des outils que nous ne connaissions pas. C'était pour nous un premier pas dans le monde de la modélisation, et même si cette expérience s'est avérée frustrante parfois, le résultat final nous permet d'être satisfaits.