TOULOUSE INP N7

SCIENCE DU NUMÉRIQUE

GÉNIE DU LOGICIEL ET DES SYSTÈMES

Projet Ingénierie Dirigée par les Modèles

Auteur :
Timothée BLANCHY
Romain PEYREMORTE

Groupe :
Image et Multimédia
Référent :
Prénom NOM

14 novembre 2022

Résumé

Pour mettre en application notre cours d'Ingénierie Dirigée par les Modèles, nous avons réalisé un projet où nous avons manipulé deux méta-modèles : SimplePD, basé sur un système de processus avec des activités reliées par des séquences, utilisant des ressources, et PétriNet, un méta-modèle de réseaux de Pétri, basé sur un ensemble de places et de transitions reliées par des Arcs.

Cette manipulation de modèles s'est faite par la création des méta-modèles dans un éditeur de modèle, Ecore, pour déterminer les éléments basiques des modèles, et par la définition de règles de sémantique statique via OCL. Nous avons ensuite utilisé des outils graphiques et textuels pour créer les modèles SimplePDL avec les outils Sirius et Xtext. Malheureusement, suite à des erreurs, la conception textuelle via Xtext n'a pas aboutie durant notre projet.

Puis nous avons élaboré des transformateurs en différents formats (HTML et DOT) pour permettre une meilleur visualisations des modèles SimplePDL via l'outil Acceleo. Ensuite, nous avons rédigé en Java et en ATL deux codes permettant de convertir les modèles SimplePDL en modèles PetriNet. Avec les modèles PetriNet obtenus, nous avons utilisé l'outil Acceleo pour les transposer en format graphique (Dot), mais aussi en langages (Tina et LTL) permettant l'application de ces réseaux grâce aux outils NetDraw et Selt afin de déterminer leur bon fonctionnement ou non.

Table des matières

1 Introduction et fil rouge **1**
1.1 Introduction . . . 1
1.2 Fil rouge . . . 1

2 SimplePDL **3**
2.1 Présentation . . . 3
2.2 Méta-modèle et règles OCL . . . 3
2.2.1 Méta-modèle . . . 3
2.2.2 Règles OCL . . . 3
2.3 Construction . . . 4
2.3.1 Dans Eclipse . . . 4
2.3.2 Avec Sirius . . . 4
2.4 Représentation . . . 4
2.4.1 Dot . . . 4
2.4.2 HTML . . . 4

3 Réseau de Pétri **6**
3.1 Présentation . . . 6
3.2 Méta-modèle et règles OCL . . . 6
3.2.1 Méta-modèle . . . 6
3.2.2 Règles OCL . . . 7
3.3 Créer un réseau de Pétri à partir d'un SimplePDL . . . 7
3.3.1 Avec Java . . . 7
3.3.2 Avec ATL . . . 8
3.4 Représentation . . . 8
3.4.1 Format dot . . . 8
3.4.2 Format Tina . . . 8
3.5 Vérification - Règles LTL . . . 8

4 Conclusion **10**
4.1 Notions travaillées . . . 10
4.2 Difficultés particulières . . . 10

Annexes **12**

Annexe 1 : Liste détaillée des fichiers de l'archive **12**
1 SimplePDL . . . 12
2 PetriNet . . . 12

Annexe 2 **13**

Chapitre 1

Introduction et fil rouge

1.1 Introduction

Durant ce projet, nous avons manipulé, grâce au logiciel Eclipse sous cadre Modeling Framework, différents méta-modèles ainsi que réalisé une chaîne permettant de partir d'une interface graphique ou textuelle pour réaliser un premier modèle suivant un méta-modèle facilement compréhensible par l'utilisateur, pour ensuite le transformer en un autre modèle qui suit un méta-modèle possédant des outils de vérification d'application.

Nous commencerons donc ce rapport par observer le premier méta-modèle, SimplePDL, avec ses éléments et ses règles, sa construction de modèle, ainsi que ses moyens de représentation. Puis ensuite, nous parleront du deuxième méta-modèle, PetriNet, avec ses éléments et ses règles, sa création à partir d'un modèle de processus SimplePDL, puis de sa représentation et son application. Enfin, nous conclurons.

1.2 Fil rouge

Pour plus facilement comprendre ce travail, et pour tester, nous avons appliqué toute la chaîne créée à un exemple spécifique : la préparation d'une galette (ou crêpe) au jambon et au fromage râpée. Cette préparation est grossièrement divisée en deux étape : la préparation de la pâte à crêpe, et la préparation d'une galette.

Ressources nécessaires :
— 1kg de farines
— 6 oeufs
— 1L de lait
— 100g de gruyère râpé
— 4 tranches de jambon
— 1L d'huile
— Plat
— Louche
— Fouet
— Paire de ciseaux
— Poêle
— Assiette

Préparation pâte :

1. Mettre 175g de farine dans un plat
2. Cassez 1 oeuf dedans
3. Ajouter du lait 20cL
4. Mélanger avec le fouet

Préparation galette

1. Mettre la poêle à chauffer avec l'huile
2. Cuire la première face de la galette en mettant la pâte dedans
3. Couper 1 tranche de jambon en petit morceau avec la paire de ciseaux dans l'assiette

4. Cuire la seconde face de la galette
5. Mettre le jambon sur la galette
6. Mettre le gruyère râpé
7. Mettre la galette dans l'assiette
8. Déguster la galette

FIGURE 1.1 – Galette jambon fromage préparée

Chapitre 2

SimplePDL

2.1 Présentation

SimplePDL est un modèle qui permet de facilement établir une suite d'activités (WorkDefinition). Ces activités sont liées par des connecteurs (*WorkSequence*) qui permettent de préciser les conditions sur le début et/ou la fin de ces activités (*StartToStart*, *FinishToStart*, ...). Chaque activité peut également demander une certaine quantité de ressources (*AskedRessource*), chacune des ressources (*Ressource*) ayant un stock limité. Enfin, un système de commentaire (*Guidance*) est inclut, pour détailler un ou plusieurs éléments du modèle.

2.2 Méta-modèle et règles OCL

2.2.1 Méta-modèle

Nous avons défini le méta-modèle de SimplePDL avec l'éditeur de modèle d'ecore. Le fichier qui le défini est disponible sous le nom **SimplePDL.ecore** dans l'archive rendue ! Voici ci-dessous la représentation graphique.

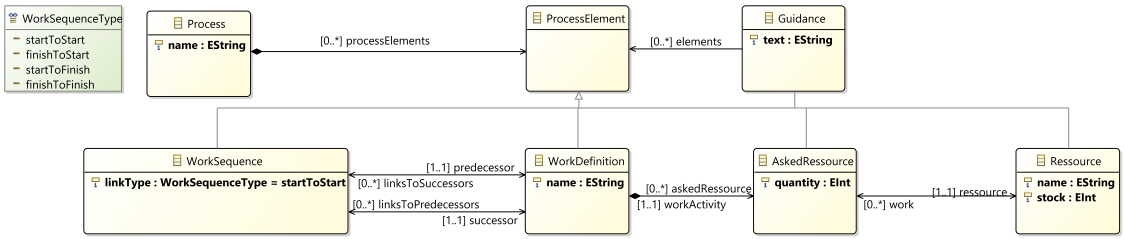

FIGURE 2.1 – Méta-modèle de SimplePDL [Vue graphique]

2.2.2 Règles OCL

Nous avons mis en place des règles pour compléter et valider ce méta-modèle via OCL, qui nous permet de mettre en place une sémantique statique. Ces règles sont dans le fichier **simplePDL.ocl** disponible dans l'archive rendue. Voici les principales créées :

— Dès qu'un élément a un attribut nom (*"name"*), celui-ci doit, pour être accepter, faire plus de deux caractère, commencer par une lettre et être composé ensuite de lettres, de chiffres ou de _.
— Un nom ne peux être utilisé qu'une seule fois pour décrire des éléments.
— Le nombre de ressource demandé et disponible doit être positif.

2.3 Construction

2.3.1 Dans Eclipse

Il est possible d'utiliser l'éditeur intégré d'Eclipse pour créer un modèle SimplePDL. Voici une capture d'écran d'un exemple d'édition d'un élément d'un modèle SimplePDL.

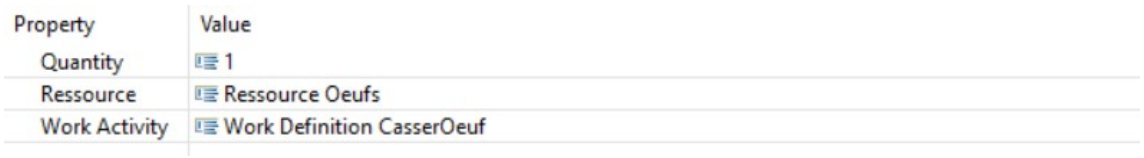

FIGURE 2.2 – Édition d'un élément d'un modèle SimplePDL avec l'éditeur d'Eclipse

Malheureusement, cet éditeur n'est pas très pratique, surtout pour des utilisateurs non développeur, et c'est pour cela qu'une autre solution existe : l'éditeur graphique de Sirius.

2.3.2 Avec Sirius

Cet éditeur permet de créer et modifier un modèle SimplePDL graphiquement et très facilement. Plusieurs boutons y sont intégrés, permettant de créer aisément de nouvelles activités, transitions, commentaires et ressources. Il est également très simple de changer ou supprimer les éléments déjà en place ainsi que les liens entre eux.

Voici les boutons "elements" présents :
— **WorkDefinition** : permet de créer une activité
— **Guidance Node** : pour créer un commentaire (sans lien)
— **Ressource** : permet de créer une nouvelle ressource

Et voici les boutons "links" :
— **Work Sequence** : pour créer les liens entre les activités
— **Guidance Link** : permet de relier les commentaires à certains éléments
— **Asked Ressource** : permet de définir les ressources dont dépendent les activités

Une capture d'écran de notre recette de galette créé via cet éditeur graphique est disponible en Annexe 2 à la figure 4.1.
Pour aider à la compréhension et mieux visualiser les différents types d'éléments, nous avons décidé de séparer dans différents calques. Donc pour visualiser l'ensemble, il faut activer tous les calques.

2.4 Représentation

2.4.1 Dot

"Le langage **DOT** est un langage de description de graphe dans un format texte. Il fait partie de l'ensemble d'outils open source Graphviz créés par les laboratoires de recherche d'AT&T." (Source : Wikipedia)

Pour représenter facilement les modèles créés, nous avons mis en place une exportation des modèles SimplePDL vers ce format universel (sans les commentaires). Le fichier **pdl-PreparationGaletteJambonFromage-dot.dot**, disponible dans l'archive rendue, est le résultat de cette exportation pour notre fil conducteur.

Ensuite, de multiples outils peuvent ensuite être utilisés pour afficher graphiquement le fichier produit. Par exemple nous avons utilisé l'outil en ligne *graphvizOnline* pour visualiser notre recette.

2.4.2 HTML

Nous avons également une exportation possible vers le format **HTML**, même si elle resterait à peaufiner (n'étant pas demandée par le sujet, nous n'avons pas rajouté les ressources et les commentaires).

Voici le rendu pour notre fil conducteur (le fichier **PreparationGaletteJambonFromage.html** est disponible dans l'archive rendue). :

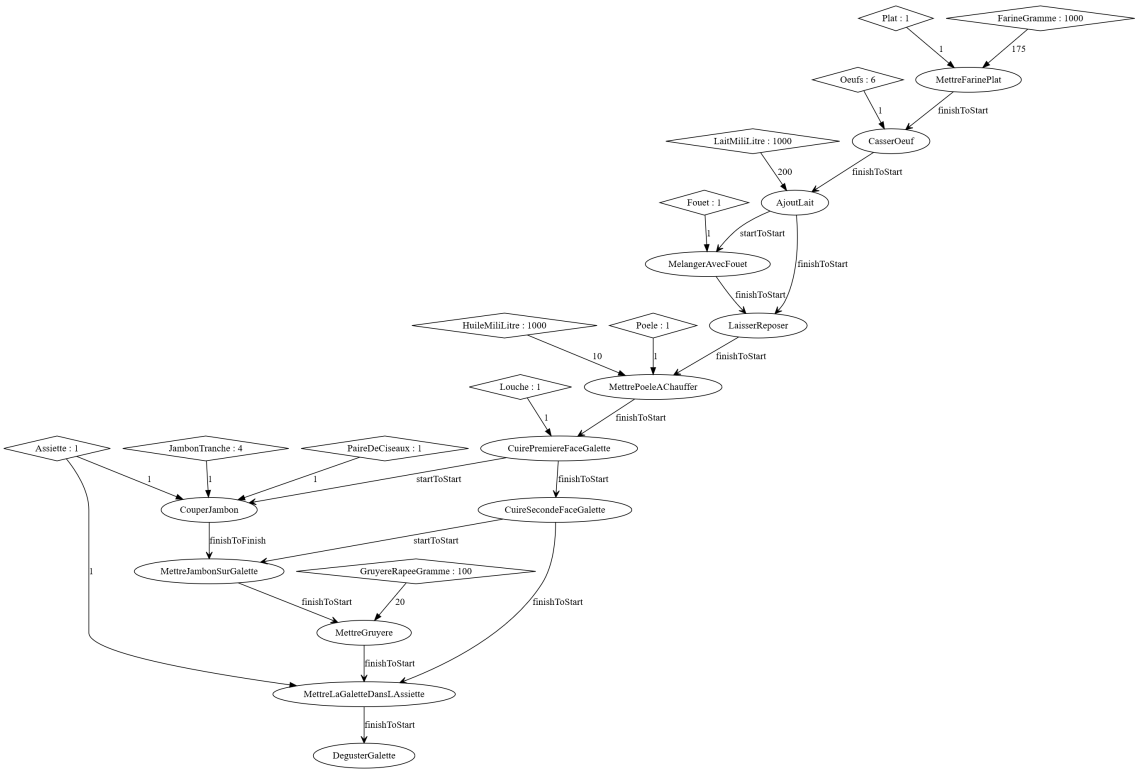

FIGURE 2.3 – Rendu de la recette de galette SimplePDL [fichier .dot via graphvizOnline]

Process "PreparationGaletteJambonFromage"

Work definitions

- MettreFarinePlat
- CasserOeuf requires MettreFarinePlat to be finished.
- AjoutLait requires CasserOeuf to be finished.
- MelangerAvecFouet requires AjoutLait to be started.
- LaisserReposer requires MelangerAvecFouet to be finished.
- MettrePoeleAChauffer requires LaisserReposer to be finished.
- CuirePremiereFaceGalette requires MettrePoeleAChauffer to be finished, MettreLaGaletteDansLAssiette to be finished.
- CouperJambon requires CuirePremiereFaceGalette to be started.
- CuireSecondeFaceGalette requires CuirePremiereFaceGalette to be finished.
- MettreJambonSurGalette requires CuireSecondeFaceGalette to be started, CouperJambon to be finished.
- MettreGruyere requires MettreJambonSurGalette to be finished.
- MettreLaGaletteDansLAssiette requires CuireSecondeFaceGalette to be finished.
- DegusterGalette requires MettreLaGaletteDansLAssiette to be finished.

FIGURE 2.4 – Visuel de la recette de galette [fichier .html]

Chapitre 3

Réseau de Pétri

3.1 Présentation

Un réseau de pétri est un modèle mathématique utilisé pour représenter différents systèmes. Ce modèle de réseau a été créé en 1962 par Carl Adam Petri. En voici une définition.

Un réseau de Pétri est un 6-uplet (S, T, F, M_0, W) où :
— S définit une ou plusieurs places.
— T définit une ou plusieurs transitions.
— F définit un ou plusieurs arcs (flèches). Un arc ne peut pas connecter deux places ni deux transitions, il ne peut connecter que des paires place-transition ; plus formellement : $F \subseteq (S \times T) \cup (T \times S) F \subseteq (S \times T) \cup (T \times S)$.
— $M_0 : S \to \mathbb{N}$ appelé marquage initial, où, pour chaque place $s \in S$, il y a $n \in \mathbb{N}$ jetons.
— $W : F \to \mathbb{N}^+$ appelé ensemble d'arcs primaires , assignant à chaque arc $f \in F$ un entier positif $n \in \mathbb{N}^+$ qui indique combien de jetons sont consommés depuis une place vers une transition, ou sinon, combien de jetons sont produits par une transition et arrivent pour chaque place.

3.2 Méta-modèle et règles OCL

3.2.1 Méta-modèle

Nous avons défini le méta-modèle du réseau de Pétri avec l'éditeur de modèle d'ecore que l'on peut retrouver dans le fichier **PetriNet.ecore**. Graphiquement, cela donne :

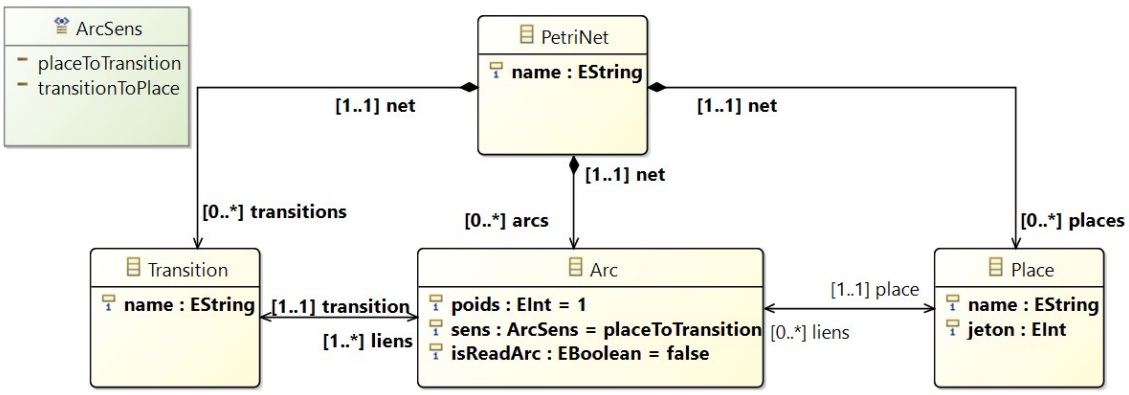

FIGURE 3.1 – Méta-modèle du réseau de Pétri [Vue graphique]

3.2.2 Règles OCL

Les règles OCL du méta-modèle PetriNet sont retrouvables dans le fichier **petriNetOCL.ocl**. Voici une brève liste de ces règles :

— Pour chaque élément ayant un attribut nom (name), nous avons mis en place les règles suivantes : le nom commence par une lettre de l'alphabet latin, ne comporte que des lettres de l'alphabet latin ou des chiffres ou le symbole _, a au moins deux caractères, et chaque élément a un nom unique dans son type (chaque place a un nom différent, chaque transition aussi)
— Pour chaque attribut comptant les jeton (jeton dans Place et poids dans Arc), nous avons vérifié qu'ils étaient positifs.
— Chaque transition doit avoir au moins une Place en entrée et au moins une Place en sortie.

3.3 Créer un réseau de Pétri à partir d'un SimplePDL

3.3.1 Avec Java

La transformation d'un processus suivant le SimplePDL en un réseau suivant le modèle PetriNet se fait par le programme **SimplePDL2PetriNet.java**. La transformation en Java se base sur les classes Java générées à partir des méta-modèles ecore de SimplePDL et PetriNet.

Pour utiliser ce programme, il faut mettre en argument le chemin vers le modèle SimplePDL à transformer, sans cela l'algorithme ne pourra pas tourner.

Voici ensuite les différentes étapes du programme pour produire le réseau de pétri de sortie :

— Tout commence par la recherche et l'ouverture du modèle SimplePDL en entrée, et la préparation de la lecture des différents objets de ce modèle.
— Ensuite le programme crée le fichier de sortie (le réseau de Pétri) et prépare la création des différents objets de ce réseau. Puis on rentre dans le coeur du programme.
— Pour chaque activité (WorkDefiniton), nous devons créer une série de 4 places, 2 transitions et 5 arcs, pour chaque Ressource, une place, pour chaque WorkSequence, 1 arc, et pour chaque AskedRessource, 2 arcs.

Cependant, en parcourant le fichier de départ, lorsque nous devons par exemple relier les différentes séries de WorkDefinition en fonction des WorkSequence, il faut que toutes les transitions et places de WorkDefinition soient créées pour réaliser les liaisons, de même avec les Ressources, AskedRessources et WorkDefinitions.

C'est pourquoi nous avons décidé de d'abord créé les places, transitions et arcs pour les Ressources et WorkDefinition, puis les relier ("réalisation des liaisons entre les séries") entre elles en créant ceux pour représenter les AskedRessources et les WorkSequences.

En appliquant cette solution, nous sommes tombés face à un problème : comment, lorsque l'on réalise les liaisons entre les séries, peut-on retrouver à quelle place ou quelle transition les arcs doivent être connectés ? En effet, Java créé des objets pour chaque élément du nouveau réseau de Pétri, on a donc rencontré des difficultés pour retrouver les objets déjà créés lors des liaisons (création d'arc entre transition et place créé à deux WorkDefinitions différent par exemple).

Nous avons donc utilisé des Map (dictionnaire) pour stocker les places et transitions utiles : chaque WorkSequence utilise une transition et une place. Ainsi,il possible, lors de la création de ces places et transition propres à une WorkDefinition, de les mémoriser pour, après coup, les relier. De même pour les Ressources qui ont une place et donc on peut associer à chaque Ressource une place via une Map.

La mémorisation des places et transitions pour les liaisons entre WorkDefinition, Ressource a été notre plus grand difficulté sur cette partie, qu'on a réussi à surmonter. Une fois que l'on a parcouru chaque élément du processus d'origine et associé les places, transitions et arcs correspondant, on finit le programme en sauvegardant le fichier obtenu.

En appliquant ce programme Java au processus de notre exemple fil rouge, on obtient le fichier **petri-PreparationGaletteJambonFromage-java.xmi**, disponible dans l'archibe rendue.

3.3.2 Avec ATL

La conversion d'un processus en réseau de Pétri peut aussi se faire grâce au programme **SimplePDL2PetriNet.atl** en réalisant une transformation ATL. La transformation ATL se base directement des méta-modèles ecores contrairement au Java qui a besoin que l'on génère les classes Java correspondantes.

L'avantage d'ATL à Java permet de résoudre le problème rencontré en Java plus facilement. En effet, en indiquant à ATL quelle est l'élément du processus dont on cherche la place ou la transition, puis le nom générique (commun à chaque élément du processus) de cette objet, ATL le retrouve de lui même grâce au code *"thisModule.resolveTemp"*. ATL permet face à Java de ne pas utiliser de dictionnaire et ainsi potentiellement de gagner de la mémoire et/ou du temps d'exécution.

En appliquant ce programme ATL au processus de notre exemple fil rouge, on obtient le fichier **petri-PreparationGaletteJambonFromage-atl.xmi**, pour le distinguer de celui générer par le Java.

3.4 Représentation

3.4.1 Format dot

Comme pour SimplePDL, nous avons mis en place une exportation des modèles PetriNet vers le format **DOT** pour mieux visualiser le résultat avant de passer à l'outil Tina et au logiciel NetDraw.
On a alors, pour notre exemple, le fichier **petri-PreparationGaletteJambonFromage.dot**, donnant le graphique se trouvant sur les figures 4.2 et 4.3 en Annexe 2 (grâce à l'outil en ligne *graphvizOnline*).

3.4.2 Format Tina

Pour pouvoir contrôler le réseau de Pétri ainsi que le visualiser sous une forme que le Dot, nous avons, via l'outil Acceleo, transformé les modèles suivant le méta-modèle en réseau pouvant être manipulé par la boîte à outils Tina. Pour cela, nous avons appliqué le programme **PetriNet2Tina.mtl** qui prend en entrée un modèle suivant le méta-modèle PetriNet et qui donne en sorti un réseau de Pétri en format **.net**.

Le résultat obtenu pour notre exemple fil rouge est **PreparationGaletteJambonFromage.net**, disponible aussi dans l'archive rendue.

3.5 Vérification - Règles LTL

A partir du fichier Tina, obtnu, il est possible de faire des tests, notamment le test d'accessibilité qui confirme l'absence d'interblocages sur notre exemple fil rouge. Le test arrive à la conclusion que notre réseau est limité ("bounded"), non vivant ("not live") et non reversible ("not reversible").

```
ANALYSIS COMPLETED -----------------------------------------------

# net PreparationGaletteJambonFromage, 64 places, 26 transitions       #
# bounded, not live, not reversible                                    #
# abstraction        count      props      psets       dead       live #
#      states           43         64          ?          1          1 #
# transitions           60         26          ?          0          0 #
```

FIGURE 3.2 – Retour du test d'accessibilité

Pour vérifier si le réseau fonctionne correctement, nous avons aussi développé des règles LTL à partir du méta-modèle SimplePDL : **SimplePDL2LTL.mtl**. Ce programme, grâce à l'outil Acceleo, rédige deux séries de propriétés pour le réseau : les propriétés des invariants et les propriétés de terminaison. Ces propriétés sont testées via le programme *selt* de l'outil Tina.
Les propriétés des invariants sont les règles vérifiés à tout moment de l'exécution du réseau. Nous vérifions donc que pour chaque WorkDefinition, cette dernière est à un seul état :

1. *ready*
2. *running*
3. *finished*

De plus, chaque état indiquant aux autres élément du process ou réseau l'avancement doit resté activé : les états *started* et *finished* doivent rester actif une fois qu'ils le sont. De même qu'un WorkDefinition *finished* doit être d'abord *started* par logique de la chose. Un autre invariant que nous vérifions est qu'à la fin d'une WorkDefinition, cette dernière "rend" les ressources utilisées : cette règle n'est pas vérifiée tout le temps car une ressource peut être utilisée par une autre WorkDefinition alors que son premier utilisateur a fini avec.

Les invariant de terminaison sont vérifiés lorsque le réseau est terminé, c'est à dire *dead*. Nous vérifions alors que chaque WorkDefinition ne tourne pas : l'état d'une WorkDefinition doit être *finished*. Indirectement, chaque WorkDefinition ne doit pas être à l'état *running*. De plus, chaque ressource doit avoir à la fin récupérer tout son stock car le modèle de processus SimplePDL est de telle sorte que les ressources sont justes occupées durant le fonctionnement. Cela donne donc un autre invariant terminale.

Les fichiers LTL pour notre exemple fil rouge sont **PreparationGaletteJambonFromageInv.ltl** et **PreparationGaletteJambonFromageFin.ltl**. Les tests appliqués ont donner que des résultats *TRUE*.

Chapitre 4

Conclusion

4.1 Notions travaillées

Ce projet nous a d'abord permis d'appréhender le fonctionnement des modèles et méta-modèles, de comprendre certains de leurs usages et des fonctionnalités qui leurs sont associés, d'appliquer les éléments vu durant les cours et travaux dirigés.

Il nous a également permis de manier plus particulièrement eclipse, et certains des framework qui lui sont associés, même si cela n'a pas été sans difficulté, comme précisé dans la partie suivante.

4.2 Difficultés particulières

Une première et principale difficulté a été le travail à deux :

— Premièrement, nous avons pas forcément la même manière de fonctionner, donc il a fallu prendre du temps pour apprendre à travailler ensemble et à communiquer sur ce que nous faisions.

— Deuxièmement, nous avons commencé le travail sur l'ordinateur de Romain, tournant sous Windows et avec une certaine version de java, d'Eclipse,... Ainsi il n'était pas possible de travailler à plusieurs et à distance sur le code source, car Timothée n'avait aucun de moyen de le faire tourner (sur un ordinateur Linux de l'école ou son ordinateur personnel). Nous avons donc décider de ne coder que sur l'ordinateur de Romain, mais de quand même utiliser un répertoire github pour versionner notre projet, et que Timothée puisse consulter ce code.

— Pour le rapport, nous avons décidé d'utiliser le site *overleaf.com* pour pouvoir modifier en même temps le fichier LaTeX source de celui-ci, et cela a été très pratique, surtout quand nous devions traviller à distance.

La deuxième difficulté était l'utilisation d'eclipse et de ses multiples frmaework qui n'était pas forcément très claire pour nous. Nous avons eu de nombreux bogues avant de réussir à finir notre projet. L'outil semble très puissant, mais parfois un peu capricieux (un exemple concret est que certains problèmes ne se résolvaient qu'en relançant totalement eclipse).

Nous vous remercions pour le temps que vous avez passé à lire ce rapport.

Annexes

Annexe 1 : Liste détaillée des fichiers de l'archive

Voici la liste détaillée des fichiers présents dans l'archive rendue.

1 SimplePDL

1. *SimplePDL.ecore* : Le méta-modèle de SimplePDL.
2. *SimplePDL.ocl* : Conditions OCL du méta-modèle SimplePDL.
3. *SimplePDL.aird* : Représentation graphique du méta-modèle SimplePDL.
4. *simplepdl.odesign* : Code de représentation et construction sous sirius des modèles SimplePDL.
5. *pdl-process-ko.xmi* : Exemple de modèle SimplePDL ne respectant pas les contraintes OCL.
6. *pdl-PreparationGaletteJambonFromage.xmi* : Modèle SimplePDL de la recette fil rouge.
7. *SimplePDL2Dot.mtl* : Programme Acceleo transformant le modèle SimplePDL au format DOT.
8. *pdl-PreparationGaletteJambonFromage-dot.dot* : Représentation graphique du modèle SimplePDL de la recette au format dot.
9. *SimplePDL2HTML.mtl* : Programme Acceleo transformant le modèle SimplePDL au format HTML.
10. *pdl-PreparationGaletteJambonFromage-html.html* : Représentation textuelle du modèle SimplePDL de la recette au format html.

2 PetriNet

1. *PetriNet.ecore* : Le méta-modèle de PetriNet.
2. *PetriNet.ocl* : Conditions OCL du méta-modèle PetriNet.
3. *PetriNet_test_1.xmi* : Modèle de test de PetriNet conçu de 3 places, 3 arcs et 1 transition.
4. *PetriNet_test_2.xmi* : Modèle de test de PetriNet pour vérifier le *ReadArc*.
5. *PetriNet_test_3.xmi* : Modèle de test de PetriNet ne respectant pas les contraintes OCL.
6. *SimplePDL2PetriNet.java* : Programme Java de transformation d'un modèle de processus SimplePDL en réseau de Pétri (modèle PetriNet).
7. *petri-PreparationGaletteJambonFromage-java.xmi* : Modèle PetriNet de la recette suite à la transformation par le programme Java.
8. *SimplePDL2PetriNet.atl* : Programme Java de transformation d'un modèle de processus SimplePDL en réseau de Pétri (modèle PetriNet).
9. *petri-PreparationGaletteJambonFromage-atl.xmi* : Modèle PetriNet de la recette suite à la transformation par le programme ATL.
10. *PetriNet2Dot.mtl* : Programme Acceleo transformant le modèle PetriNet au format DOT.
11. *petri-PreparationGaletteJambonFromage-dot.dot* : Représentation graphique du modèle PetriNet de la recette au format dot.
12. *PetriNet2Tina.mtl* : Programme Acceleo transformant un modèle PetriNet en réseau de Pétri en format net.
13. *PreparationGaletteJambonFromage.net* : Réseau de Pétri de la recette manipulable par Tina.
14. *SimplePDL2LTL.mtl* : Programme Acceleo transformant un modèle SimplePDL en propriétés invariants et propriétés de terminaison pour Tina.
15. *PreparationGaletteJambonFromageInv.ltl* : Invariants du réseau de Pétri de la recette.
16. *PreparationGaletteJambonFromageFin.ltl* : Propriétés terminales du réseau de Pétri de la recette.

Annexe 2 : Figures supplémentaires

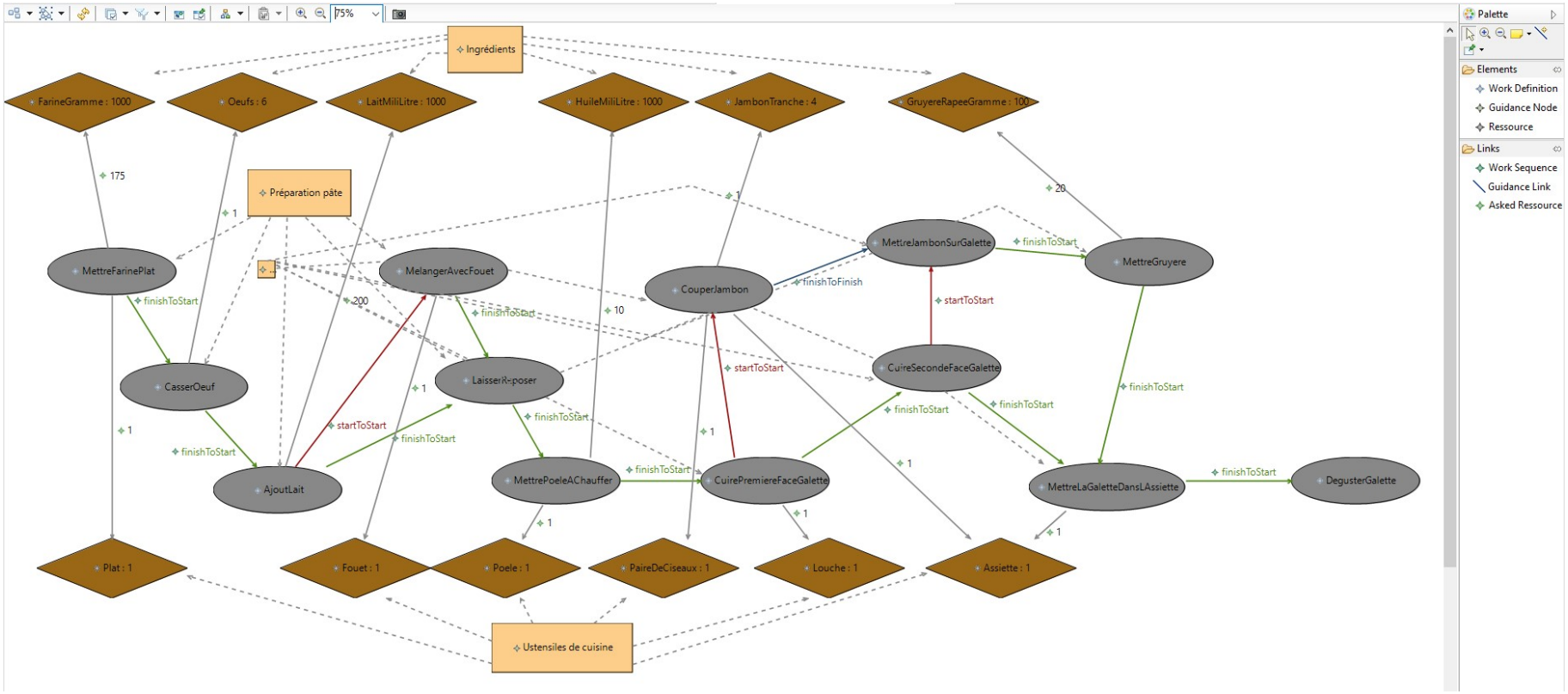

FIGURE 4.1 – Édition d'un élément d'un modèle SimplePDL avec l'éditeur d'Eclipse

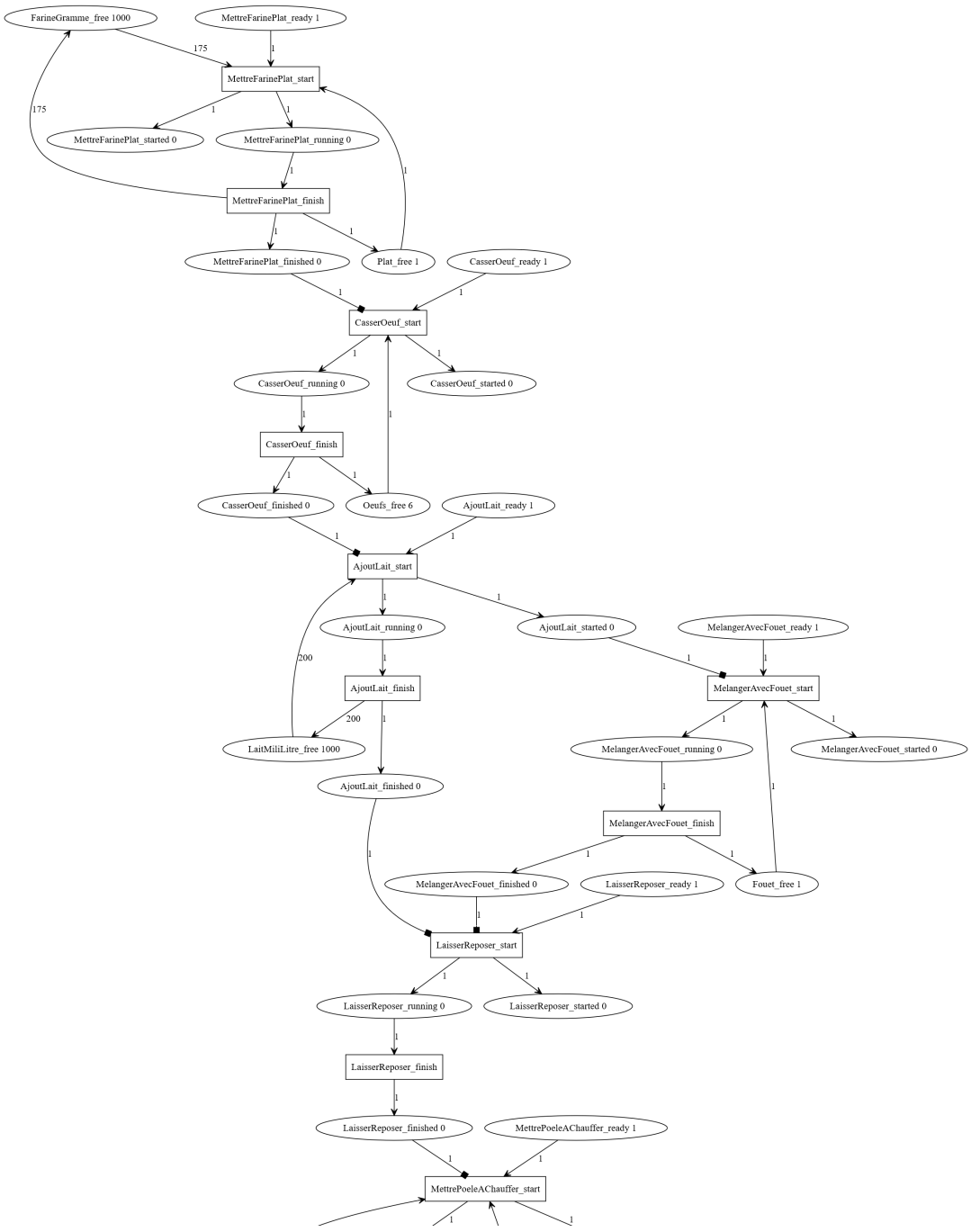

FIGURE 4.2 – Rendu de la recette de galette PetriNet part 1 [fichier .dot via graphvizOnline]

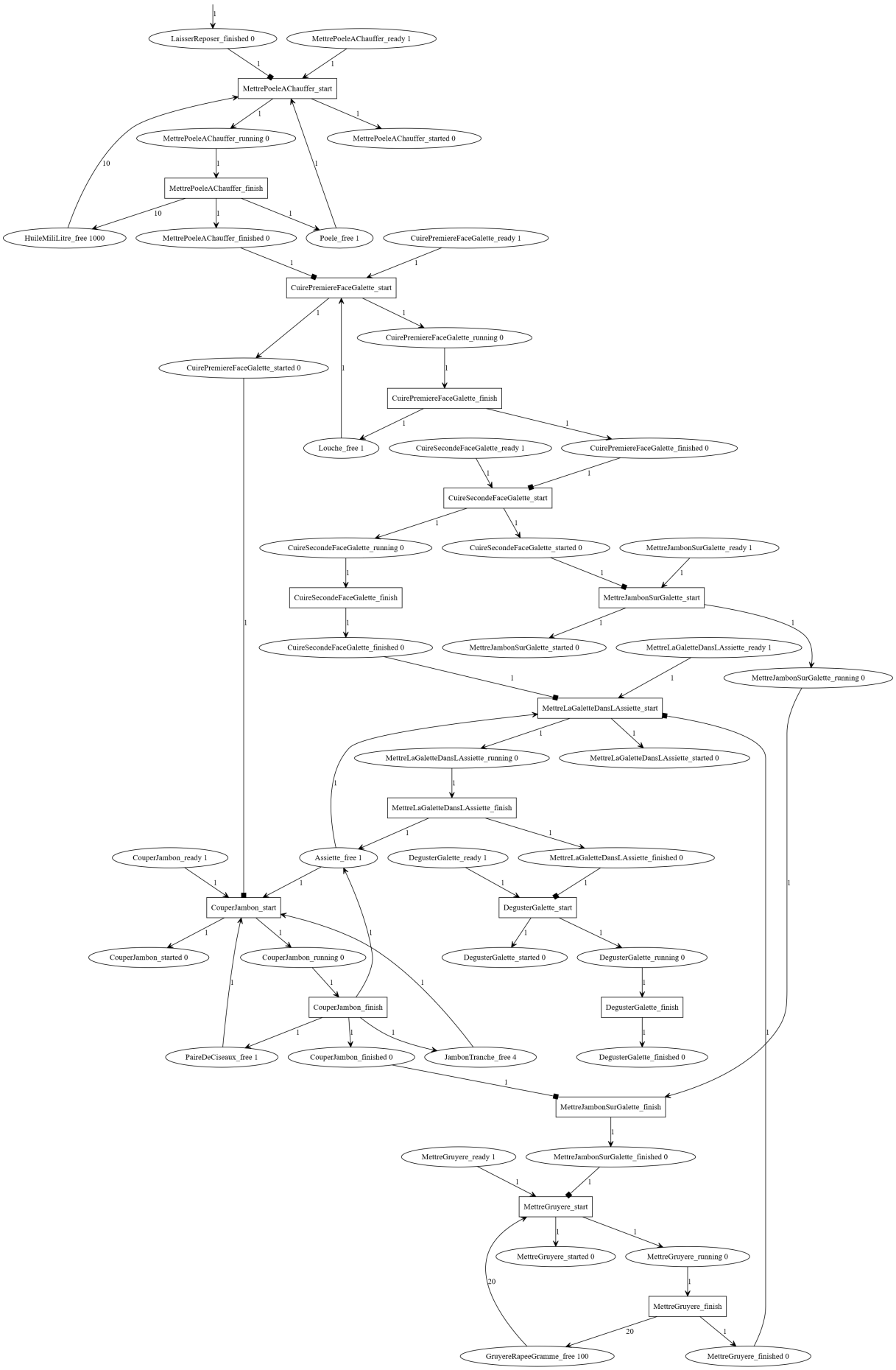

FIGURE 4.3 – Rendu de la recette de galette PetriNet part 2 [fichier .dot via graphvizOnline]