
SIMPLEPDL2TINA : Mise en oeuvre d'une Validation de Modèles de Processus

Benoît Combemale* — **Xavier Crégut***
Bernard Berthomieu** — **François Vernadat****

* *IRIT-CNRS, INPT-ENSEEIH*
Université de Toulouse
{benoit.combemale, xavier.cregut}@enseeiht.fr

** *LAAS-CNRS*
Université de Toulouse
{Bernard.Berthomieu, Francois.Vernadat}@laas.fr

RÉSUMÉ. L'Ingénierie Dirigée par les Modèles (IDM) place les modèles au centre du développement d'un système. Il est donc important de pouvoir valider les premiers modèles construits à partir de DSL (Domain Specific Language). Partant d'un langage simplifié de description de procédé, SIMPLEPDL, nous avons mené dans (Combemale et al., 2006c) des expérimentations pour définir une sémantique opérationnelle en utilisant un langage de méta-programmation ou des transformations endogènes. Une autre approche consiste à s'appuyer sur un langage formellement défini pour exprimer la sémantique d'un langage donné. On parle alors de sémantique dénotationnelle. Dans cet article, nous reprenons SIMPLEPDL pour expérimenter cette approche vers les réseaux de Petri temporels.

ABSTRACT. Model-Driven Engineering (MDE) emphasizes models in the center of a lifecycle. Thus, it is important to be able to validate models. With a simplified Process Description Language, SIMPLEPDL, we carried out in (Combemale et al., 2006c) experiments to define an operational semantics by using a meta-programming language or endogenous transformations. Another approach to express semantics consists in translating our meta-model into a well defined language. It is called a denotational semantics. In this article, we use SIMPLEPDL to experiment this approach towards the technological space of the time Petri nets.

MOTS-CLÉS : IDM, transformations, sémantique, validation, ATL, Réseaux de Petri, Tina

KEYWORDS: MDE, transformations, semantics, validation, ATL, Petri Nets, Tina

1. Introduction

L'Ingénierie Dirigée par les Modèles (IDM) considère les modèles comme des entités de première classe. Les modèles sont définis par des méta-modèles qui en définissent la syntaxe et certaines propriétés structurelles, généralement sous la forme d'une syntaxe abstraite (MOF, Ecore, KM3, etc) complétée de contraintes (invariants, pré-, post-conditions) spécifiées à l'aide d'un langage de requête tel qu'OCL. Un développement est alors une succession de transformations, automatiques ou assistées. L'IDM favorise donc la définition de langages dédiés (*Domain Specific Language*) qui ont de plus l'avantage de permettre aux utilisateurs de se concentrer sur leur métier en manipulant un formalisme spécifique à leur activité (UML, CWM, SPEM, etc.).

Dans un cycle de vie, il est reconnu que plus tôt une erreur est détectée, moins élevé est son coût de correction. Il est donc important de pouvoir valider les premiers modèles construits à partir de DSL. Cette validation peut être obtenue par la simulation des modèles construits. Il s'agit alors de définir explicitement la sémantique opératoire qui est sous-jacente au méta-modèle considéré. Il est également possible de vérifier que le modèle respecte certaines propriétés de cohérence, de vivacité ou de terminaison. Un outil de type model-checking permet alors de faire une analyse des exécutions possibles pour confirmer l'hypothèse ou mettre en évidence un ou plusieurs contre-exemples. Les projets en cours traitant des aspects vérification et simulation de modèles dans une approche IDM comme par exemple TOPCASED (Farail *et al.*, 2006) montrent l'intérêt de cet aspect d'un point de vue industriel.

Partant d'un langage simplifié de description de procédé, SIMPLEPDL, nous avons mené dans (Combemale *et al.*, 2006c) des expérimentations pour définir une sémantique opérationnelle en utilisant un langage de méta-programmation (p.ex. Kermeta (Muller *et al.*, 2005)) ou des transformations endogènes (exprimées en ATL (Jouault *et al.*, 2005) ou AGG (Taentzer, 2003)). Nous avons pu dans les deux cas simuler un modèle de procédé mais un enrichissement du méta-modèle était nécessaire pour capturer l'état global du processus. La principale différence est qu'une approche par méta-programmation consiste à enrichir le méta-modèle d'opérations équipées d'une implantation qui, dans une approche par transformations endogènes, sont décrites en dehors du méta-modèle, dans la transformation elle-même.

Une autre approche consiste à s'appuyer sur un langage formellement (c.-à-d. mathématiquement) défini pour exprimer la sémantique d'un langage donné. On parle alors de sémantique dénotationnelle (Mosses, 1990, Gunter *et al.*, 1990). Appliquée à l'IDM, il s'agit d'exprimer des transformations vers un autre espace technologique (Jean-Marie Favre, 2006) dans lequel il est possible d'utiliser des outils de simulation, vérification ou exécution disponibles. On parle aussi de sémantique de traduction (Clark *et al.*, 2004).

Dans cet article, nous reprenons SIMPLEPDL pour expérimenter cette approche. L'espace technologique que nous avons choisi est celui des réseaux de Petri temporels. Il s'agit alors de traduire un modèle de procédé en réseaux de Petri pour utiliser les outils de simulation et de vérification proposés par la boîte à outils TINA (Ber-

thomieu *et al.*, 2004). Nous commençons par décrire SIMPLEPDL et les propriétés que l'on souhaite vérifier (section 2) puis le meta-modèle PETRINET des réseaux de Petri utilisés (section 3). La section 4 explique le schéma de transformation d'un modèle SIMPLEPDL en PETRINET, puis la transformation de ce dernier dans la syntaxe concrète d'entrée de TINA. Ces deux transformations sont décrites en utilisant ATL. Nous décrivons dans la section 5 comment les propriétés identifiées sur SIMPLEPDL peuvent être vérifiées dans le formalisme des réseaux de Petri en utilisant des expressions de la logique LTL (*Linear Temporal Logic*). Nous terminons cet article par une discussion (section 6) sur l'approche suivie pour définir la sémantique comportementale de notre DSL et une conclusion (section 7) sur nos travaux et perspectives.

2. Présentation de SIMPLEPDL

SIMPLEPDL est un langage de description de procédé. Il s'inspire du standard SPEM¹ (*Software Process Engineering Metamodel*) (Obj 2005) proposé par l'OMG mais aussi du méta-modèle UMA (*Unified Method Architecture*) utilisé par le plug-in Eclipse EPF² (*Eclipse Process Framework*), dédié à la modélisation de procédé. Il est volontairement simplifié pour ne pas surcharger inutilement les expérimentations.

Le méta-modèle SIMPLEPDL est donné figure 1. Il définit le concept de processus (*Process*) composé d'un ensemble d'activités (*WorkDefinition*) représentant les différentes tâches à réaliser durant le développement. Une activité peut dépendre d'une autre (*WorkSequence*). Une contrainte d'ordonnancement sur le démarrage ou la fin de la seconde activité est précisée (*linkType*) grâce à l'énumération *WorkSequenceType*. Par exemple, deux activités A_1 et A_2 reliées par une relation de précédence de type *finishToStart* signifie que A_2 ne pourra commencer que quand A_1 sera terminée.

La figure 4 donne un exemple de modèle de processus composé de quatre activités. Le développement ne peut commencer que quand la conception est terminée. La rédaction de la documentation ou des tests peut commencer dès que la conception est commencée (*startToStart*) mais la documentation ne peut être terminée que si la conception est terminée (*finishToFinish*) et les tests si le développement est terminé.

Notons que SIMPLEPDL fait également apparaître la notion de ressource (*Resource*) nécessaire pour la réalisation d'une activité (concepteur, machine, serveur...) et de temps (*min_time* et *max_time* sur *WorkDefinition* et *Process*) pour préciser le temps minimum (resp. maximum) nécessaire pour la réalisation d'une activité (resp. d'un processus). Ces éléments ne seront pas utilisés dans un premier temps dans notre exemple afin de ne pas le surcharger mais nous y reviendrons à la fin de l'article.

D'autre part, nous n'avons modélisé ici ni les produits (*WorkProduct*) que manipulent une activité, ni les rôles (*Role*) qui peuvent être assimilés à des ressources.

1. Nous proposons une analyse du standard SPEM 1.1 dans (Combemale *et al.*, 2006b)

2. <http://www.eclipse.org/epf/>

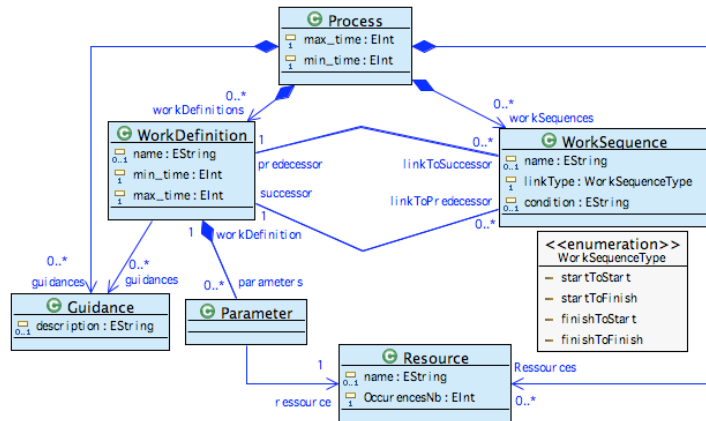


Figure 1. Meta-modèle de SIMPLEPDL en Ecore

Cette description structurelle des modèles de procédé doit être complétée par la description du comportement et des propriétés dynamiques de ces modèles. Par exemple, une activité peut être démarrée et terminée. L'utilisateur (c.-à-d. la personne qui définit un processus) peut alors se demander si un modèle de procédé peut effectivement se terminer, c'est-à-dire si toutes ses activités peuvent se terminer.

Quand on rajoute les ressources et le temps, on peut affiner ces questions. Par exemple, pour un jeu de ressources données, est-il possible de terminer un processus ? Est-il possible de respecter les contraintes temporelles (*min_time* et *max_time*) ajoutées sur les activités et le processus ? Si non, quelles sont les activités qui ne respectent pas les contraintes de temps, qui n'obtiennent pas les ressources nécessaires à leur réalisation, etc.

3. Réseaux de Petri, Logique Temporelle et Vérification

Cette section présente les différents éléments permettant de mettre en oeuvre la vérification de modèles SIMPLEPDL. Parmi ceux-ci, sont utilisés les réseaux de Petri temporels pour donner une sémantique par traduction à la description du modèle SIMPLEPDL, la logique temporelle SE-LTL pour exprimer les propriétés à vérifier sur le modèle SIMPLEPDL, et l'environnement de description et de vérification TINA qui supporte ces différents éléments. Faute de place, nous ne décrivons pas dans le détail chacune de ces parties et nous renvoyons le lecteur intéressé à (Chaki *et al.*, 2004) pour une présentation de SE-LTL, (Berthomieu *et al.*, 2006) pour une présentation des réseaux temporels et (Berthomieu *et al.*, 2004) pour une description de TINA.

3.1. TINA (Time Petri Net Analyzer)

Il s'agit d'un environnement logiciel permettant l'édition et l'analyse de réseaux de Petri et réseaux temporels (Berthomieu *et al.*, 2004). Les différents outils constituant l'environnement peuvent être utilisés de façon combinée ou indépendante. Parmi les outils utilisés dans le cadre de cette étude on citera :

- *nd* (*NetDraw*) : *nd* est un outil d'édition de réseaux temporels et d'automates, sous forme graphique ou textuelle. Aussi, il intègre un simulateur « pas à pas » (graphique ou textuel) pour les réseaux temporels et permet d'invoquer les outils ci-dessous sans sortir de l'éditeur.

- *tina* : cet outil construit des représentations de l'espace d'états d'un réseau de Petri, temporel ou non. Aux constructions classiques (graphe de marquages, arbre de couverture), *tina* ajoute la construction d'espaces d'états abstraits, basés sur les techniques d'ordre partiel. Pour les réseaux temporels, *tina* propose toutes les constructions de graphes de classes discutées dans (Berthomieu *et al.*, 2006).

- *selt* : en plus des propriétés générales d'accessibilité vérifiées à la volée par *tina* (caractère borné, présence de blocage, pseudo-vivacité et vivacité), il est le plus souvent indispensable de pouvoir garantir des propriétés spécifiques relatives au système modélisé. L'outil *selt* est un vérificateur de modèle (*model-checker*) pour les formules d'une extension de la logique temporelle *SE-LTL* (State/Event *LTL*) de (Chaki *et al.*, 2004). En cas de non-satisfaction d'une formule, *selt* peut fournir une séquence contre-exemple en clair ou sous un format exploitable par le simulateur de TINA.

3.2. Réseaux Temporels

Les réseaux temporels, introduits dans (Merlin, 1974), sont obtenus depuis les réseaux de Petri en associant deux dates *min* et *max* à chaque transition. Supposons que *t* soit devenue sensibilisée pour la dernière fois à la date θ , alors *t* ne peut être tirée avant la date $\theta + min$ et doit l'être au plus tard à la date $\theta + max$, sauf si le tir d'une autre transition a désensibilisé *t* avant que celle-ci ne soit tirée. Le tir des transitions est de durée nulle. Les réseaux temporels expriment nativement des spécifications « en délais ». En explicitant débuts et fins d'actions, ils peuvent aussi exprimer des spécifications « en durées ». Leur domaine d'application est donc large.

La vérification de réseaux temporels nécessite d'obtenir une abstraction finie de l'espace d'états temporels associé qui est en général infini. Les méthodes permettant d'obtenir ces abstractions finies sont basées sur la technique des classes d'états, et ont été initiées par (Berthomieu *et al.*, 1983, Berthomieu *et al.*, 1991). Une présentation des résultats les plus récents peut-être trouvée dans (Berthomieu *et al.*, 2006).

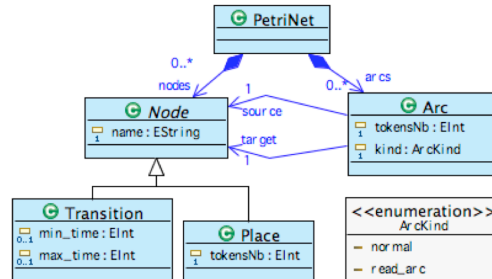


Figure 2. Meta-modèle des réseaux de Petri temporel en Ecore

3.3. Meta-modèle des réseaux temporels

Le meta-modèle des réseaux temporels est présenté sur la figure 2. Un réseau de Petri (*PetriNet*) est composé de noeuds (*Node*) pouvant être des places (*Place*) ou des transitions (*Transition*). Les noeuds sont reliés par des arcs (*Arc*) pouvant être des arcs normaux ou des read-arcs (*ArcKind*). Le nombre de jetons d'un arc indique le nombre de jetons consommés dans la place source ou ajoutés à la place destination (sauf dans le cas d'un read-arc où il s'agit simplement de tester si la place source contient le nombre spécifié de jetons). Le marquage du réseau de Petri est capturé par l'attribut *nbJetons* d'une place. Enfin, nous rajoutons la possibilité d'exprimer un intervalle de temps sur une transition.

Ce meta-modèle permet de construire des modèles incorrects. Par exemple on peut mettre un arc entre deux places ou deux transitions. Il a donc été complété par des contraintes OCL pour restreindre les instances valides. Une approche identique a été appliquée pour restreindre le meta-modèle SPEM (Combemale *et al.*, 2006a).

3.4. La logique temporelle SE-LTL

La logique *LTL* étend le calcul propositionnel en permettant l'expression de propriétés spécifiques sur les séquences d'exécution d'un système. *SE-LTL* est une variante de *LTL* récemment introduite (Chaki *et al.*, 2004), qui permet de traiter de façon homogène des propositions d'états et des propositions de transitions. Les modèles pour la logique *SE-LTL* sont des structures de Kripke étiquetées (ou *SKE*), aussi appelées systèmes de transitions de Kripke (ou *KTS*). En voici quelques formules :

- (Pour tout chemin)
- P P vraie au départ du chemin (pour l'état initial),
 - $\square P$ P vraie tout le long du chemin,
 - $\diamond P$ P vraie une fois au moins le long du chemin,
 - $P U Q$ Q sera vraie dans le futur et P est vraie jusqu'à cet instant,
 - $\square \diamond P$ P vraie infiniment souvent.

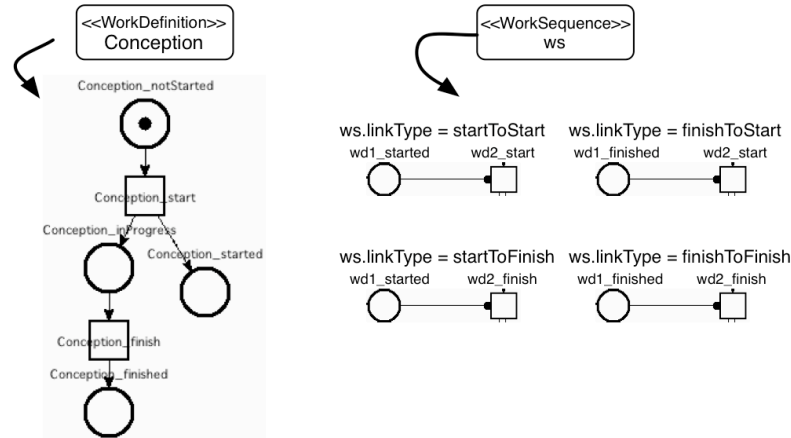


Figure 3. Schéma de traduction de SimplePDL en PetriNet

Les structures de Kripke étiquetées que nous manipulons sont obtenues à partir d'un réseau de Petri. Afin de conserver l'information de multiplicité de marques exprimée par les marquages, selt travaille sur des structures de Kripke enrichies, basées sur des multi-ensembles de propriétés atomiques plutôt que des ensembles. Afin d'exploiter au mieux l'information contenue dans ces structures de Kripke enrichies, on substitue au calcul propositionnel (bi-valué par $\{true, false\}$), un calcul propositionnel multi-valué et on étend le langage d'interrogation de selt avec des opérateurs logico-arithmétiques. Le model-checker selt permet aussi la déclaration d'opérateurs dérivés ainsi que la redéclaration des opérateurs existants ; un petit nombre de commandes sont aussi fournies pour contrôler l'impression des résultats ou encore permettre l'utilisation de bibliothèques de formules.

Quelques formules de selt :

- $\square (p2 + p4 + p5 = 2)$ un invariant de marquage linéaire,
- $\square (p2 * p4 * p5 = 0)$ un invariant de marquage non linéaire,
- infix $q R p = \square (p \Rightarrow \diamond q)$ déclare l'opérateur « répond à », noté R .

4. Description de SIMPLEPDL2PETRINET

Le schéma de traduction pour transformer un modèle de processus en un modèle de réseaux de Petri (fig. 4) est présenté figure 3. Le code ATL correspondant comprend trois règles déclaratives (listing 1). La première traduit chaque activité en quatre places caractérisant son état (*NotStarted*, *Started*, *InProgress* et *Finished*) et deux transitions (*Start* et *Finish*). L'état *Started* permet de mémoriser qu'une activité a démarré.

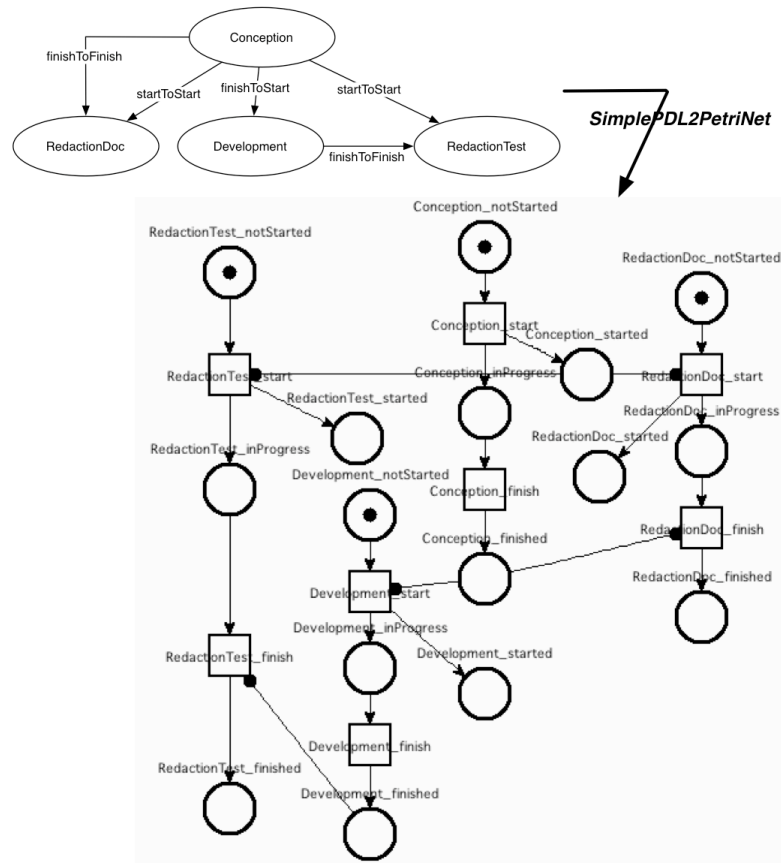


Figure 4. Exemple de transformation d'un processus en réseau de Petri

Ces places et transitions sont utilisées dans la deuxième règle pour traduire chaque *WorkSequence* en un *read-arc* entre une place de l'activité source et une de l'activité cible, la place et la transition dépendant de la valeur de l'attribut *linkType*. L'opération *resolveTemp* d'ATL est nécessaire car plusieurs places et transitions sont créées pour une même *WorkDefinition*.

La dernière règle traduit *Process* en un *PetriNet* regroupant les nœuds et arcs construits par les autres règles.

Afin de pouvoir manipuler le réseau de Petri dans l'outil TINA, nous avons complété la précédente transformation par une transformation *PETRI NET2TINA* qui traduit un modèle de réseau de Petri dans la syntaxe concrète textuelle requise par TINA.

Nous avons illustré la transformation *SIMPLEPDL2TINA* sur des méta-modèles simplifiés pour ne pas surcharger la présentation. D'autres éléments, ressources, temps

Listing 1: Extrait de la transformation ATL de SIMPLEPDL vers PETRINET

```

module SimplePDL2PetriNet;
create OUT: PetriNet from IN: SimplePDL;

rule WorkDefinition2PetriNet {
  from wd: SimplePDL!WorkDefinition
  to
    -- création des places
    p_notStarted : PetriNet !Place (name <- wd.name + '_notStarted', tokensNb <- 1),
    p_started : PetriNet !Place (name <- wd.name + '_started', tokensNb <- 0),
    p_inProgress : PetriNet !Place (name <- wd.name + '_inProgress', tokensNb <- 0),
    p_finished : PetriNet !Place (name <- wd.name + '_finished', tokensNb <- 0),
    -- création des transitions
    t_start : PetriNet !Transition (name <- wd.name + '_start'),
    t_finish : PetriNet !Transition (name <- wd.name + '_finish'),
    -- création des arcs :
    a_nsed2s : PetriNet !Arc (kind <- #normal, tokensNb <- 1,
                             source <- p_notStarted, target <- t_start ),
    ...
}
rule WorkSequence2PetriNet {
  from ws: SimplePDL!WorkSequence
  to
    a_ws: PetriNet !Arc (kind <- #read_arc, tokensNb <- 1,
                        source <-
                          if ((ws.linkType = # finishToStart ) or (ws.linkType = # finishToFinish ))
                            then thisModule.resolveTemp(ws.predecessor, ' p_finished ')
                            else thisModule.resolveTemp(ws.predecessor, ' p_started ')
                          endif,
                        target <-
                          if ((ws.linkType = # finishToStart ) or (ws.linkType = # startToStart ))
                            then thisModule.resolveTemp(ws.successor, ' t_start ')
                            else thisModule.resolveTemp(ws.successor, ' t_finish ')
                          endif )
}
rule Process2PetriNet {
  from p: SimplePDL!Process
  to pn: PetriNet !PetriNet (nodes <- ..., arcs <- ...)
}

```

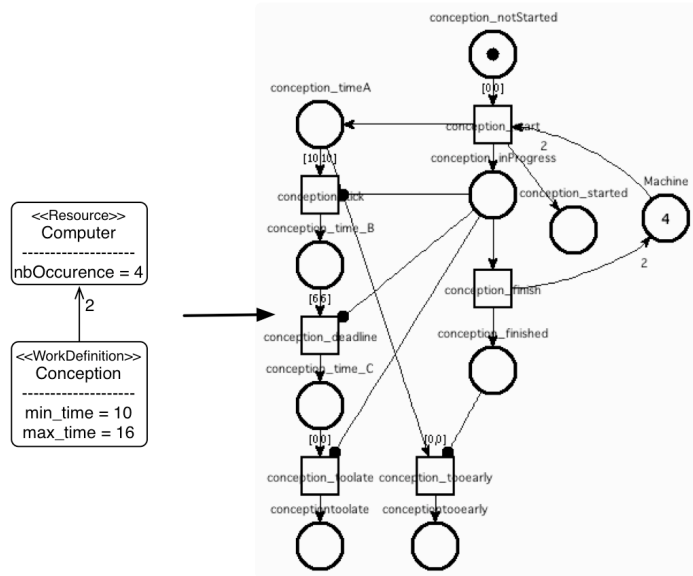


Figure 5. Traduction d'un *WorkDefinition* temporel en réseaux de Petri temporel

et décomposition d'une activité en sous-activités ont été pris en compte. La figure 5 montre la traduction d'un *WorkDefinition* temporel utilisant une ressource.

5. Analyse sur SIMPLEPDL

Après la transformation du modèle de processus en réseaux de Petri nous souhaitons vérifier les propriétés associées à SIMPLEPDL en utilisant l'outil Tina.

Le principe est d'engendrer les propriétés LTL en utilisant des requêtes ATL. Nous engendrons les propriétés LTL au format texte. Il faut donc connaître le schéma de traduction de SIMPLEPDL vers PETRINET et Tina pour être cohérent. Travailler directement au niveau PETRINET aurait signifié s'appuyer également sur le méta-modèle de LTL ce qui ne semblait pas pertinent ici.

Nous décrivons ci-dessous les types de propriétés que nous avons vérifiées.

5.1. Consistance des modèles SIMPLEPDL

L'utilisateur, lorsqu'il élabore son modèle SIMPLEPDL, décrit les contraintes qui vont régir son procédé. Celles-ci peuvent être de nature causale (c.à-d. stipuler qu'une

Listing 2: Génération des propriétés pour l'étude de la terminaison

```

query finished =
  'op finished = true'.concat(
    pdl!WorkDefinition.allInstances()->iterate(wd; acc : String=' ' |
      acc.concat(' \\\ ' + wd.name + '_finished ')) + ';\n\n'
  + '[] <> dead ;\n'
  + '[] (dead => finished) ;\n'
  ...
  .writeTo('/tmp/finished . ltl ');

```

activité ne peut débiter qu'après la fin d'une autre), liées à la possession de ressources nécessaires pour la réalisation d'une activité, ou encore temporelles... Une fois décrit l'ensemble des contraintes associé au procédé, l'utilisateur doit être capable de savoir si le procédé qu'il vient de décrire est satisfaisable (réalisable), en d'autres termes, existe-t-il une exécution permettant de réaliser, en respectant les contraintes temporelles fixées, les différentes activités en accord avec les contraintes de causalité ?

Les techniques de model-checking vont lui permettre de résoudre ce problème. La question de savoir si un procédé est réalisable peut se ramener à une étude de la terminaison de l'exécution du procédé. Dans le contexte de cette étude, on considérera que le procédé est terminé si chacune des activités prévues dans le procédé ont effectivement été exécutés³. A cet effet, la macro-proposition *finished* est générée automatiquement lors de la traduction du modèle SIMPLEPDL (listing 2).

On peut distinguer la correction partielle « tout procédé terminé est dans son état final » (exprimé par $\square (dead \Rightarrow finished)$) et la terminaison en tant que telle ($\square \diamond dead$) qui assure que toute exécution se termine.

À ce stade, on a une « consistance forte » : toute exécution termine et toute exécution terminée est dans son état final. Dans la pratique, l'ensemble des contraintes exprimées dans le modèle *SimplePDL* peuvent être telles que toute exécution de celui-ci ne termine pas systématiquement.

On peut donc s'intéresser à une notion de « consistance faible » permettant de s'assurer de l'existence d'au moins une exécution du procédé. On évalue la propriété $\neg \diamond finished$ qui, littéralement, exprime qu'aucune exécution ne termine. Si le procédé est faiblement consistant, cette propriété est évaluée à *False* et le contre-exemple produit par selt donne une exécution correcte du procédé. Si la propriété est évaluée à *True* alors le procédé est inconsistant et n'admet aucune solution.

Il est à noter que l'étude de la consistance est générique. Le seul paramètre à fournir est la donnée de la proposition *finished*. Dans le cas de descriptions temporisées, la démarche est du même type.

3. D'autres alternatives pourraient bien sur être considérés, la définition de cette notion de « terminaison » devant se faire au niveau de SIMPLEPDL.

5.2. Correction de la transformation :

Le langage SIMPLEPDL ne possédant pas de sémantique formellement définie il n'est bien sûr pas possible de valider formellement la correction de la transformation réalisée. Dans le contexte de cette étude, la transformation vers les réseaux temporels revient à proposer pour SIMPLEPDL une sémantique par traduction.

Si la transformation exprimée en ATL est à un niveau d'abstraction suffisant, il est toutefois important de pouvoir vérifier que la transformation est consistante et produit une traduction en réseau de Petri conforme à l'« esprit » de la spécification SIMPLEPDL. La complexité du réseau de Petri obtenu à partir de la spécification SIMPLEPDL rend difficile cette analyse.

Pour faciliter la mise au point de la transformation ATL et nous assurer de la « conformité » du réseau de Petri (transformé) nous générons automatiquement avec ATL un ensemble de formules LTL qui doivent être satisfaites par la traduction en réseau de Petri du procédé SIMPLEPDL.

Ces formules permettent par exemple de s'assurer que les différents états d'une activité sont mutuellement exclusifs, ou de s'assurer des contraintes de précédence entre les différentes activités du procédé, exprimé en SIMPLEPDL par l'attribut *linkType* de *WorkSequence*. Ainsi pour tout couple d'activités $(A1, A2)$ et tout couple de statut $\in \{finish, start\}^2$, la contrainte $A1_status1 TO_status2 A2$ exprimée en SIMPLEPDL donnera lieu à la propriété LTL suivante : $A1_status1 Precede A2_status2$.

L'ensemble de ces propriétés correspond à des « obligations de preuves » qui doivent être vérifiées. Ces propriétés sont engendrées lors de la phase de transformation du modèle SIMPLEPDL en réseaux de Petri et s'appuient sur les conventions de nommage utilisées pour les places et les transitions dans le mapping SIMPLEPDL2PETRINET. Dans le cadre de cette étude, nous avons uniquement mis en oeuvre une vérification de ces propriétés par model-checking (pour chaque instance transformée), il serait aussi possible/intéressant d'établir (une fois pour toutes) une preuve structurelle de leur satisfaction.

6. Discussion

Une première conclusion de l'approche adoptée est qu'elle est opérationnelle. Nous avons effectivement pu traduire nos modèles SIMPLEPDL en réseaux de Petri et générer les propriétés exprimées en LTL pour ensuite utiliser TINA pour vérifier la consistance de la transformation et le modèle de processus lui-même. Nous présentons ci-dessous un retour d'expérience.

6.1. Intérêt de la transformation

La définition d'une transformation explicite est un point positif. En effet, si le schéma de traduction de SIMPLEPDL vers les réseaux de Petri est relativement simple, il nécessite de construire de nombreux motifs répétitifs correspondant à la modélisation d'une activité, d'une précedence, d'une ressource, etc. Ces correspondances sont définies sous la forme de règles ATL qui seront automatiquement appliquées sur tous les éléments du modèle SIMPLEPDL en entrée de la transformation.

Nous avons utilisé un meta-modèle intermédiaire, PETRINET, entre les modèles SIMPLEPDL et le code pour TINA. Ceci permet de simplifier les transformations et d'abstraire le changement de formalisme au niveau des modèles (Kleppe, 2006). La transformation PETRINET2TINA peut ainsi être capitalisée pour plusieurs DSL.

Le code ATL n'est cependant pas nécessairement très lisible car nous n'avons pas une correspondance un à un entre les concepts du méta-modèle d'entrée et ceux du méta-modèle de sortie. Par exemple, une activité est traduite par quatre places et deux transitions. Il faut alors définir une convention pour nommer les états (et les transitions). D'autre part, lorsque l'on veut traduire la relation de précedence, on ne peut pas utiliser la simple affectation d'ATL mais nous devons passer par *resolveTemp* pour retrouver la place de l'activité source et la transition de l'activité cible qui devront être reliées par un arc. Cette place et cette activité dépendent du type de précedence.

Notons que pouvoir définir la transformation dans un langage de haut niveau devient encore plus intéressant si nous compliquons notre méta-modèle SIMPLEPDL en autorisant par exemple un choix entre plusieurs ressources pour démarrer une activité ou si nous enrichissons sa sémantique, par exemple en donnant la possibilité d'interrompre et reprendre une activité, autorisant ainsi la libération momentanée des ressources utilisées. Ces extensions ont été développées mais ne sont pas présentées ici car elles apportent peu et engendrent des réseaux de Petri d'une taille telle qu'ils deviennent difficilement lisibles.

Un inconvénient concerne cependant la génération des propriétés LTL. En effet, ces propriétés ne font pas partie des réseaux de Petri. Nous avons donc défini des requêtes ATL pour traduire une propriété sur un modèle SIMPLEPDL en un texte LTL. Ce processus s'appuie sur les conventions utilisées pour nommer les places et non sur le réseau de Petri lui-même.

6.2. Nécessité d'une maîtrise de bout en bout

Pour l'approche mise en place, nous récapitulons ici l'effort à investir par un concepteur SIMPLEPDL et celui à déployer par le « transformateur » qui va mettre au point la chaîne de transformations allant du modèle SIMPLEPDL aux réseaux de Petri pour représenter son comportement et à la logique temporelle pour s'assurer de sa consistance.

L'utilisateur niveau SIMPLEPDL dispose de moyens de s'assurer automatiquement de la consistance de son modèle. Il peut invoquer la chaîne d'outils de façon transparente. Cependant, en cas de consistance faible, le scénario fourni par TINA est exprimé directement dans la traduction Petri et devrait être remonté dans les éléments de modélisation SIMPLEPDL. Nous n'avons pas encore développé cet aspect là, mais il nous semble possible de définir l'équivalent d'attributs dérivés (ou de requêtes) sur le modèle SIMPLEPDL qui permettraient d'obtenir les informations obtenues sur les réseaux de Petri. Par exemple, les états d'une activité (ressource non allouées, prête, en cours, suspendue, terminée, etc.) pourraient être déduits de la place contenant le jeton. Dans notre schéma de transformation, il suffit de décoder le nom de la place : avant le souligné, on récupère le nom de l'activité, à droite son état.

Pour faciliter le diagnostic, en cas d'inconsistance, des propriétés générales relatives à la vivacité des activités ont été développées et n'ont pas été présentées ici faute de place. Plus généralement, on pourrait envisager de fournir à l'utilisateur un langage « métier » lui permettant d'interroger son modèle. Il suffirait alors au « transformateur » de proposer une traduction de celui-ci dans le langage de propriétés.

Le « transformateur » doit maîtriser les différents éléments de la chaîne de transformations : SIMPLEPDL, réseaux de Petri, LTL, schéma de traduction, ATL et les outils de vérification disponibles sur les réseaux de Petri.

6.3. Limitation du stockage des modèles en XMI

Lors de cette expérimentation, nous avons enrichi progressivement notre meta-modèle SIMPLEPDL. Nous avons donc fait évoluer en conséquence le modèle correspondant. Malheureusement, les modifications faites sur notre méta-modèle ont souvent rendu la version du modèle illisible et ont obligé à la resaisir complètement. Il est donc indispensable de mettre en place une approche consistant à définir, quand cela est possible, une transformation entre chaque version successive du meta-modèle afin de mettre à jour automatiquement les modèles.

6.4. Sémantique opérationnelle ou sémantique par traduction

Nous avons étudié l'expression de sémantiques pour les langages de modélisation dans cet article (sémantique par traduction) et dans (Combemale *et al.*, 2006c) (sémantique opérationnelle). Une sémantique opérationnelle permet de rester dans le même espace technologique alors qu'une sémantique par traduction, par définition, met en œuvre une traduction vers un autre espace technologique formellement défini.

Une sémantique par traduction permet de bénéficier de tous les outils disponibles dans l'espace cible, par exemple simulateurs, model-checkers... Se pose toutefois le problème de la remontée des résultats obtenus vers l'espace d'origine. Telle qu'expérimentée dans (Combemale *et al.*, 2006c) avec Kermeta, ATL ou AGG, une sémantique opérationnelle nous semble plus simple à mettre en œuvre, en particulier parce que,

restant dans le même espace technologique, les concepts manipulés sont plus proches du domaine métier.

Ainsi, dans un objectif d’animation/simulation une approche opérationnelle est préférable, en particulier si le modèle de calcul est simple (p.ex. événements discrets). Pour de la vérification/validation, une approche par traduction est plus adaptée pour tirer partie des outils de vérifications disponible dans l’espace technologique cible.

7. Conclusion et perspectives

Nous présentons dans cet article la mise en oeuvre d’une validation de modèles de processus exprimés à partir d’un DSL simplifié, SIMPLEPDL. Cette validation est faite par le biais de transformations vers les réseaux de Petri temporels et une manipulation avec la boîte à outils TINA. Nous utilisons également un méta-modèle intermédiaire, PETRINET, afin d’abstraire la transformation réalisant le changement de formalisme. Cette approche opérationnelle permet ainsi de réutiliser tous les outils développés dans des domaines formellement définis comme les réseaux de Petri.

Des travaux comme (Cicchetti *et al.*, 2006) proposent des transformations vers l’espace technologique des *Abstract State Machine*. (Chen *et al.*, 2005) développent cette approche sous le nom de *Semantic Anchoring* correspondant à l’ancrage au niveau du DSL d’unités sémantiques définies comme des abstractions de comportement traduit en *Abstract State Machine*.

Cette expérimentation nous permet de mettre en avant de nombreuses perspectives indispensables pour un aboutissement favorable de cette approche. La remontée des informations au niveau du DSL source, la validation de la transformation et la lourdeur des manipulations sont autant de questions encore ouvertes à ce jour.

NOTE. — Ces travaux ont été réalisés dans le cadre du projet Topcased (<http://www.topcased.org>)

8. Bibliographie

- Berthomieu B., Diaz M., « Modeling and Verification of Time Dependent Systems Using Time Petri Nets. », *IEEE Transactions on Software Engineering*, vol. 17, n° 3, p. 259-273, 1991.
- Berthomieu B., Menasche M., « An Enumerative Approach for Analyzing Time Petri Nets. », *IFIP Congress Series*, vol. 9, p. 41-46, 1983.
- Berthomieu B., Ribet P.-O., Vernadat F., « The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », *International Journal of Production Research*, vol. 42, n° 14, p. 2741-2756, 15 juillet, 2004.
- Berthomieu B., Vernadat F., *Réseaux de Petri temporels : méthodes d’analyse et vérification avec TINA*, Traité IC2 1, Ed. Nicolas Navet, Hermes, 2006.

- Chaki S., E M., Clarke, Ouaknine J., Sharygina N., Sinha N., « State/Event-based Software Model Checking », *4th International Conference on Integrated Formal Methods (IFM'04)*, Springer LNCS 2999, p. 128-147, avril, 2004.
- Chen K., Sztipanovits J., Abdelwalhed S., Jackson E., « Semantic Anchoring with Model Transformations », in , L. 3748 (ed.), *Model Driven Architecture - Foundations and Applications, First European Conference (ECMDA-FA)*, p. 115-129, 2005.
- Cicchetti A., Ruscio D. D., Pierantonio A., « Weaving concerns in model based development of data-intensive web applications », *Proceedings of the Symposium on Applied Computing (SAC '06)*, ACM Press, New York, USA, p. 1256-1261, 2006.
- Clark T., Evans A., Sammut P., Willans J., « Applied Metamodelling - A Foundation for Language Driven Development », 2004, version 0.1.
- Combemale B., Crégut X., Caplain A., Coulette B., « Modélisation rigoureuse en SPEM de procédé de développement », in , Lavoisier (ed.), *12ième conférence sur les Langages et Modèles à Objets (LMO'06)*, Hermès Sciences, Nîmes, France, p. 135-150, mars, 2006a.
- Combemale B., Crégut X., Ober I., Percebois C., « Evaluation du standard SPEM de représentation des processus », *Génie Logiciel, Modèles et Processus de développement*, vol. 77, p. 25-30, Juin, 2006b.
- Combemale B., Rougemaille S., Crégut X., Migeon F., Pantel M., Maurel C., « Sémantique dans la méta-modélisation », in , H. Sciences/Lavoisier (ed.), *2ieme journées sur l'Ingénierie Dirigée par les Modeles (IDM'06)*, Lille, France, juin, 2006c.
- Farail P., Gauffillet P., Canals A., Camus C. L., Sciamma D., Michel P., Crégut X., Pantel M., « The TOPCASED project : a Toolkit in OPEN source for Critical Aeronautic SystEms Design », *Embedded Real Time Software (ERTS'06)*, Toulouse, France, 25-27 janvier, 2006.
- Gunter C. A., Scott D. S., « Semantic Domains », *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics (B)*, p. 633-674, 1990.
- Jean-Marie Favre Jacky Estublier M. B., *L'Ingénierie Dirigée par les Modèles : au-delà du MDA*, Informatique et Systèmes d'Information, Hermes Science, Lavoisier (ed.), février, 2006.
- Jouault F., Kurtev I., « Transforming Models with ATL », *Proceedings of the Model Transformations in Practice Workshop at MoDELS'05*, Montego Bay, Jamaica, 2005.
- Kleppe A., « MCC : A Model Transformation Environment », *ECMDA-FA*, p. 173-187, 2006.
- Merlin P. M., *A Study of the Recoverability of Computing Systems.*, Irvine : Univ. California, PhD Thesis, 1974.
- Mosses P. D., « Denotational Semantics », *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics (B)*, p. 575-631, 1990.
- Muller P.-A., Fleurey F., Jézéquel J.-M., « Weaving Executability into Object-Oriented Meta-Languages », in , S. K. L. Briand (ed.), *MoDELS'05*, LNCS, Montego Bay, Jamaica, octobre, 2005.
- Obj, *Software Process Engineering Metamodel (SPEM) 1.1 Specification.* janvier, 2005, formal/05-01-06.
- Taentzer G., « AGG : A Graph Transformation Environment for Modeling and Validation of Software. », in , J. L. Pfaltz, , M. Nagl, , B. Böhlen (eds), *AGTIVE*, vol. 3062 of LNCS, p. 446-453, 2003.