Les Zero-Safe Nets pour la Préservation de la TTC dans les Diagrammes d'Activité d'UML

Sabine Boufenara*, Faiza Belala**, Chafia Bouanaka***

*Université Benhamouda de Jijel, Ouled Aissa, Algérie
sabineboufenara@yahoo.com
Université Mentouri de Constantine, Route de Ain Elbey, Algérie
{**belalafaiza@hotmail.com
***c_bouana@yahoo.fr}

Résumé. Avec les extensions d'UML 2.0, les réseaux de Petri, utilisés comme cadre sémantique formel pour les diagrammes d'activité d'UML, ne permettent plus d'exprimer les nouvelles constructions de haut niveau telles que la traverse-to-completion. Cette dernière nécessite une synchronisation globale entre les nœuds fork et join, totalement absente dans les réseaux de Petri ordinaires offrant une synchronisation locale. Afin de préserver le comportement des diagrammes d'activité, nous proposons l'adoption des réseaux de Petri zero-safe, une classe particulière des réseaux de Petri. Nous définissons un passage générique des diagrammes d'activité d'UML vers cette classe des réseaux de Petri. La formalisation que nous proposons assure la préservation de la sémantique opérationnelle des diagrammes d'activité en mettant l'accent sur le principe de la traverse-to-completion et la synchronisation des nœuds fork et join supportant ainsi les flots de contrôle et donnée et la concurrence des threads déclenchés par le noeud fork.

1 Introduction

Le langage UML (*Unified Modelling Language*) (Clark et Evans, 2000) est devenu un standard industriel pour la modélisation des systèmes. C'est un langage visuel à caractère expressif décrivant la structure et le comportement des systèmes orientés objet. Il a une sémantique semi-formelle car celle-ci est spécifiée sous la forme d'un méta-modèle (diagramme de classe) et de règles OCL (OCL : Object Constraint Language) pour la syntaxe abstraite et la sémantique statique, et de commentaires en langage naturel pour la sémantique dynamique. Et de ce fait, il manque de capacités d'analyses et d'automatisation ce qui le rend mal adapté à la simulation, etc. Adopter des modèles formels n'est pas la solution idéale car ils seront mal acceptés par des utilisateurs novices d'UML. La solution adoptée par plusieurs chercheurs est de définir une vraie sémantique d'UML en conservant sa syntaxe, car proche de l'utilisateur et définir des règles de transformation des diagrammes d'UML en des modèles formels pour une éventuelle analyse, simulation ou autres.

Parmi les travaux de formalisation d'UML, nous pouvons mentionner celui de Kim et Carrington (1999) qui propose une formalisation du diagramme de classes basée sur le lan-

gage Z. Beaucoup de travaux (Egyed et Wile, 2001), (Paltor et Lilius, 1999) ont été aussi réalisés pour la formalisation du diagramme état/transition, inspiré des statecharts de David Harel (Harel et Naamad, 1996). Les réseaux de Petri (RdPs) ont largement été utilisés pour la formalisation des diagrammes de collaboration (Baresi et Pezzè, 2001), (Belala et Boufenara, 2009), d'interaction (Saldhana et Shatz, 2000) et d'activité (Störrle, 2005), (Störrle et Hausmann, 2005). De part leur aspect formel, les RdPs sont dotés d'outils automatiques d'analyse des systèmes. Plusieurs travaux de recherche tels que ceux de Bokhari et Poehlman (2006) proposent des algorithmes et/ou des outils logiciels pour la transformation d'un diagramme UML en un RdP. Sarstedt fait l'exception en n'utilisant pas les RdPs pour la formalisation des diagrammes d'activité. Il présente dans sa thèse (Sarstedt, 2006) une approche qui permet d'exécuter ces diagrammes par un composant appelé '*runtime component*' sans avoir à réécrire le flot de contrôle dans un langage de programmation.

Les diagrammes d'activité sont utilisés pour la description du comportement des systèmes sous une sémantique inspirée des RdPs en mettant l'accent sur le flot de contrôle. Le flot explicite d'objet a été introduit dans les diagrammes d'activité d'UML 2.0 donnant lieu à une nouvelle définition de la sémantique de ces diagrammes parue dans OMG Unified Modelling Language (2003). Parmi ces nouveaux concepts, nous citons les exceptions, les streams, la Traverse-To-Completion (TTC) avec tout ce qu'elle implique comme comportements (transformation des jetons, prévention des interblocages, etc.) ainsi que la redéfinition des nœuds fork et join, utilisés pour le lancement et la synchronisation de plusieurs threads.

Ainsi, dans la nouvelle sémantique d'UML, une activité peut offrir un jeton à l'arc sortant via une prise (nœud objet pour les entrées et les sorties des actions), qui à son tour l'offre à ses cibles. La traversée de l'arc est contrainte par l'acceptation de l'offre par la source et la destination. Du côté de la destination, l'acceptation du jeton est régularisée via une acceptation synchronisée de toutes ses prises d'entrée. Cette contrainte est appelée la traversée jusqu'à l'achèvement ou TTC. Ce concept est très intéressant quant à la prévention des interblocages et l'optimisation des implémentations de l'exécution des comportements de la traversée.

Plusieurs tentatives de formalisation des diagrammes d'activité, visant principalement la gestion des workflow patterns, ont été proposées. La plus émergeante est celle basée sur les RdPs (Störrle, 2005), du fait qu'elle préserve le comportement des diagrammes d'activité. Des variantes de RdPs traitant l'aspect réactif des systèmes, sont aussi utilisées à des fins de simulation des workflows (Eshuis and R.J. Wieringa, 2003). Bien que les RdPs autorisent une synchronisation locale des prises d'entrée des activités d'entrée d'un join, ils ne sont pas capables d'exprimer les constructions de haut niveau d'UML 2.0, à savoir la synchronisation globale des nœuds fork et join. Cette dernière est imposée dans certaines situations par le principe de la TTC où le nœud fork ne peut libérer les jetons que si le nœud join est prêt à les accepter. La spécification de la TTC exige que tout le chemin de la source à la destination soit traversé en une étape atomique. Ceci n'étant pas offert par les RdPs ordinaires du fait que le mapping des nœuds de contrôles du diagramme d'activité résulte en plusieurs places intermédiaires dans le RdP. Une solution intuitive à ce problème permettant de préserver l'atomicité du passage imposée par la TTC est la définition elle-même des zero-safe nets. Dans ce papier, nous proposons d'utiliser les RdPs zero-safe ou zero-safe nets (ZSNs) (Bruni et Montanari, 1997), (Bruni, 1999) comme cadre sémantique des diagrammes d'activité d'UML. Nous définissons un ensemble de règles génériques établissant une correspondance entre tous les concepts des diagrammes d'activité et ceux des ZSNs. La formalisation que nous proposons ne prend en considération que le problème de la synchronisation des nœuds

fork et join et donc ne s'applique qu'à une partie du diagramme d'activité que nous appelons *région concurrente*, celle-ci est délimitée par un fork à son début et un join à sa fin. L'utilisation des ZSNs pour la formalisation de la synchronisation du fork et du join dans les diagrammes d'activité d'UML apporte une solution générique à plusieurs problèmes qui ne pouvaient pas être résolus avec les RdPs ordinaires sans avoir à définir des codages complexes. Notre objectif est de définir un codage simple et naturel, structurellement semblable aux diagrammes d'activité d'UML et qui soit le plus fidèle quant à la description de leur sémantique opérationnelle.

La formalisation des diagrammes d'activité par les RdPs Zero-Safe sera faite de manière incrémentale. Dans la section 2 de ce papier, nous rappellerons les concepts de base des diagrammes d'activité UML et ceux des ZSNs. La section 3 présentera les limites des RdPs ordinaires quant à la formalisation des diagrammes d'activité. Notre approche de formalisation des diagrammes d'activité via les ZSNs et ses atouts seront présentés dans la section 4. Enfin, la section 5 conclut le papier.

2 Concepts de base

2.1 Diagramme d'activité

La préoccupation principale des diagrammes d'activité est la représentation graphique du comportement d'une méthode ou d'un cas d'utilisation. Un diagramme d'activité représente une vision temporelle du système modélisé et décrit les événements déclencheurs de changement d'états dans le système. Les diagrammes d'activité sont donc mieux adaptés pour le contrôle de flot de données entre actions.

Selon le standard UML 2 définit par (OMG Unified Modelling Language, 2003), les diagrammes d'activité ont une sémantique similaire à celle des RdPs, leur sémantique opérationnelle est basée sur un jeu de jeton similaire à celui des RdPs.

Dans un diagramme d'activité, nous avons trois types de noeuds : les noeuds de contrôle (incluant les noeuds fork, join, decision, merge, les noeuds de données, généralement appelés jetons, et les noeuds exécutables représentant des actions. Les noeuds sont interconnectés via des arcs permettant le routage des jetons (de contrôle et de données) à travers le franchissement des noeuds de contrôle, en fournissant une donnée ou un contrôle aux autres actions ou en les stockant temporairement avant de les transférer à travers le graphe (voir figure 1).

Pour qu'une action commence son exécution, il lui faudrait connaître quand est-ce qu'elle doit commencer (exprimé par le flot de contrôle) et quelles sont ses valeurs d'entrée (exprimées par des prises). Une action n'ayant pas de contrôle en entrée commence lorsque toutes ses données d'entrée sont disponibles. Les contrôles ne nécessitent pas des prises car ils ne sont pas porteurs de valeurs à part une valeur unique indiquant qu'une action peut commencer son exécution (OMG Unified Modelling Language, 2003). La question posée est : quel est l'élément qui contient cette valeur ? Et si plusieurs contrôles séquentiels étaient en attente : quel est l'élément qui contient la file d'attente des contrôles ? Les prises peuvent porter plus d'une valeur suivant leur capacité indiquée sur le modèle. Ces valeurs sont transmises aux paramètres de l'action au moment de son invocation dans un ordre FIFO par défaut ou en utilisant un système de priorité décrit par le développeur du modèle (OMG Unified Modelling Language, 2003).

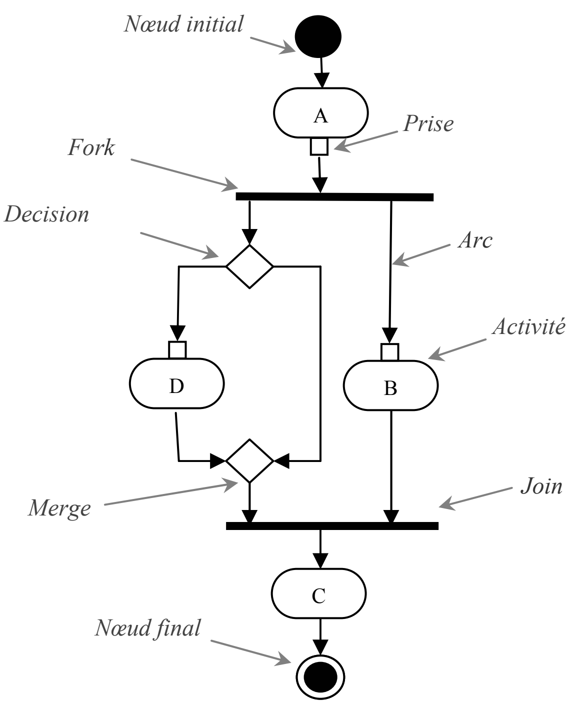

Figure 1. Exemple générique d'un diagramme d'activité

Un diagramme d'activité permet aussi de représenter graphiquement le cycle de vie de plusieurs threads et leur synchronisation, exprimée par les noeuds fork et join. Dans le cas usuel, chaque noeud fork est associé à un noeud join afin de regrouper les threads initiés par le fork. Il y a deux exceptions à cette règle, la première est : il n'est pas nécessaire que les flots concurrents provenant d'un fork soient tous synchronisés par un join. La seconde est : les flots concurrent peuvent aussi se terminer par un nœud final ou un merge, plutôt que synchronisés. Ces cas ne sont pas traités par notre approche. Des flôts provenant de l'extérieur de la région délimitée par un fork et un join peuvent être en entrée de certaines actions des threads ou même du join. Ce cas là est pris en charge par notre approche.

Le noeud fork est une transition spéciale ayant une source et plusieurs destinations. A son franchissement, toutes les cibles sont simultanément atteintes, résultant en une augmentation du nombre de threads concurrents. Inversement, le noeud join est une transition spéciale avec plusieurs sources et une cible. A son franchissement, toutes les actions sources sont terminées donnant lieu à une diminution du nombre de threads concurrents. Et par conséquent, le noeud join permet de synchroniser plusieurs flots. En plus du multithreading, UML 2.0 introduit une nouvelle fonction au nœud fork consistant en la copie des jetons. (voir figure 1).

Afin de garantir un comportement correct des diagrammes d'activité, contenant des noeuds fork et join, une condition de franchissement de ces deux types de noeuds a été définie. La TTC, proposée par Bock (2004), l'un des fondateurs d'UML 2.0, est une sémantique de synchronisation basée sur les RdPs contrôlant le mouvement des jetons entre les noeuds fork et join. Elle stipule que les jetons ne peuvent quitter leurs noeuds actuels vers d'autres noeuds que si le chemin global est ouvert, i.e., chaque jeton doit satisfaire toutes les conditions imposées par le chemin (les valeurs des gardes sont vraies, les buffers destination ne sont pas saturés, etc.). Si plusieurs chemins sont possibles, le moins contraignant est choisi.
La TTC est utile pour trois raisons principales: D'une part, elle assure une transformation des jetons par les comportements de transformation appliqués aux arcs de flots d'objets, avec préservation des jetons détruits dans leurs noeuds sources, i.e., un jeton de sortie d'une activité est maintenu dans sa prise de sortie jusqu'à acceptation du jeton d'entrée par l'activité destination. Ainsi, le jeton source n'est pas perdu au cas où sa transformation n'est pas acceptée. D'un autre côté, une prise d'entrée d'une action ne peut accepter un jeton d'entrée que quand toutes les prises d'entrée de la même action ont accepté leurs jetons d'entrée. Cette propriété est très intéressante pour la prévention de l'interblocage. Finalement, la TTC permet aussi une optimisation du code lors de la phase d'implémentation de l'exécution des comportements de franchissement (traversée) puisque le comportement d'un choix dans les noeuds objet source ne nécessite pas une exécution répétée pendant l'attente de la disponibilité de la destination.

2.2 Les réseaux de Petri Zero-Safe

Les ZSNs ont été introduits par Bruni et Montanari (1997) afin de définir un mécanisme de synchronisation en se contentant des règles ordinaires de transitions dans les réseaux sans introduire de nouveaux mécanismes d'interaction. Leur rôle est d'assurer l'exécution atomique de collections complexes de transitions, qui peuvent paraître comme synchronisées dans le réseau de Petri abstrait où le mécanisme qui contrôle le flot de jetons dans les zero places est caché.
En effet, la coordination de l'exécution atomique des différentes transitions est possible dans les ZSNs grâce à un nouveau type de places appelées *zero places*. Dans un état observable du système, les zero places sont bornées à zéro jetons. Un jeton dans une zero place est équivalent à un état interne du système qui est non-observable. Une évolution synchronisée d'un ZSN doit commencer dans un état observable, évoluer vers un marquage non-observable et se terminer dans un état observable. Les ZSNs définissent deux sortes de places : stables qui correspondent à des places ordinaires d'un RdP et des zero places. Une évolution du ZSN est considérée comme une transaction. Un jeton stable généré dans une transaction est gelé tout au long de l'évolution, il n'est libéré qu'une fois que la transaction est terminée et que l'étape stable ait été atteinte sans problèmes (en franchissant la transition sensibilisée par ce jeton). Nous devons mentionner que la transaction est une activité du système qui peut être composée d'un ensemble de sous activités concurrentes mais atomique.

Exemple. Examinons le RdP de la figure 2.a. Dans l'état observable du système, les places *a* et *b* sont synchronisées. Le RdP ne contient que des places stables et une seule activité atomique (transition) qui consomme les jetons des places *a* et *b* et génère des jetons dans les places *c* et *d*. Le ZSN (modèle raffiné) de ce réseaux abstrait est présenté dans la figure 2.b, où une zero place *z* est introduite. Elle sert à montrer un état interne et une certaine séquence

de franchissements internes (*t0* suivi de *t1*), qui n'était pas visible dans le RdP sous-jacent de la figure 2.a. Le franchissement de *t0* produit un jeton gelé dans *c* et un zero jeton dans *z*. Le franchissement qui suit est celui de *t1* qui consomme le jeton stable de *b* et le zero jeton de *z*, produisant un jeton stable en *d*. Ce nouvel état est stable et, par conséquent, la transaction est terminée libérant ainsi le jeton de la place *c*. A un niveau abstrait, nous ne sommes pas intéressés par l'observation de l'état caché composé d'un jeton stable dans *b*, un zero jeton dans *z* et un jeton gelé dans *c*. Ainsi, le rôle de la zero place z est de coordonner l'exécution atomique de la transition *t*, qui d'un point de vue abstrait apparaîtra comme synchronisée mais qui en réalité, est une transaction faite d'une coordination de sous-transitions.

(a) (b)

Figure 2. (a): Réseau de Petri abstrait. (b): ZSN correspondant ou réseau raffiné.

Définition formelle (Bruni et Montanari, 1997)**.** Un ZSN est le 6-tuplet $B = (S_B, T_B; F_B, W_B, u_B; Z_B)$ où $N_B = (S_B, T_B; F_B, W_B, u_B)$ est le RdP place/transition sous-jacent où S_B est un ensemble non vide de places, T_B est l'ensemble non vide de transitions, $F_B \subseteq (S_B \times T_B) \cup (T_B \times S_B)$ est l'ensemble des arcs, W_B est la fonction poids qui associe un entier positif à chaque arc, u_B est le marquage des places qui associe un nombre de jetons positifs à chaque place et $Z_B \subseteq S_B$ est l'ensemble des zero places (appelées places de synchronisation). Les places de SB \ ZB sont dites stables.
Un marquage stable est un multi-ensemble de places stables. La présence d'une ou de plusieurs zero places dans un marquage, le rend non-observable. Un marquage stable décrit un état observable du système.

Soit B un ZSN et soit $s = u_0[t_1>u_1\ldots u_{n-1}[t_n>u_n$ une séquence de franchissement du réseau sous-jacent N_B de B, tel que $u_i[t_{i+1}>u_{i+1}$ est le franchissement de t_{i+1}, dans l'état u_i, qui défini le nouvel état u_{i+1}.

- La séquence *s* est une étape stable de B si:
  - $\forall\, a \in S_B \setminus Z_B, \Sigma^n_{i=1}\, pre(t_i)(a) \leq u_0(a)$ *(sensibilisation concurrente)*
    pre(t)(a) est l'application d'incidence avant qui définit le poids de l'arc allant de la place *a* entrée de la transition *t* vers celle-ci à l'inverse de l'incidence arrière post(t)(a) qui donne le poids de l'arc de la transition *t* vers sa sortie *a*. La propriété de la sensibilisation concurrente assure la sensibilisation simultanée initiale par les places stables de toutes les transitions de l'étape et non seulement des transitions permettant le déclenchement de la première exécution. Nous pouvons facilement lire entre les lignes que cette propriété interdit la consommation des jetons stables générés à l'intérieur de l'étape par celle-ci.
  - u_0 et u_n sont des marquages stables de B *(équité stable)*

- L'étape stable est une transaction stable de B si en outre:
  - Les marquages $u_1,\ldots, u_{n-1}$ ne sont pas stables *(atomicité)*
  - $\forall\, a \in S_B \setminus Z_B, \Sigma^n_{i=1}\, pre(t_i)(a) = u_0(a)$ *(franchissement parfait)*

> Le franchissement parfait assure que tous les jetons stables initiaux ont été consommés par la transaction. Il rajoute une contrainte à la propriété de sensibilisation concurrente et donc interdit lui aussi la consommation des jetons stables générés à l'intérieur de la transaction par celle-ci.

D'une manière informelle, nous pouvons dire que la propriété de sensibilisation concurrente est satisfaite pour une séquence *s* lorsque toutes ses transitions dont les entrées sont uniquement des places stables, sont sensibles initialement. Le franchissement parfait revient au fait de la consommation totale de tous les jetons stables initiaux.

Dans l'exemple précédant, une étape stable peut être décrite par la séquence s_1 et la séquence s_2 décrit une transaction stable:

s_1 = {a,2b}[t0>{2b, c, z}[t1>{b, c, d}

s_1 est une étape stable, mais pas une transaction stable parce que la propriété du franchissement parfait n'est pas satisfaite du fait qu'un jeton initial de *b* n'a pas été consommé.

s_2 = {a,b}[t0>{b, c, z}[t1>{c, d}

Dans une transaction stable, chaque transition représente une sous étape d'évolution atomique à travers des états invisibles. Les jetons stables produits au cours de la transaction deviennent actifs dans le système seulement à sa fin.

3 La sémantique opérationnelle des diagrammes d'activité UML via les RdPs ordinaires

De nouveaux concepts ont été ajoutés aux diagrammes d'activité dans UML 2.0, ce qui a conduit à une révision de leur sémantique. La sémantique opérationnelle des nouvelles constructions est décrite en langage naturel par un jeu de jetons semblable à celui des RdPs. Le tableau 1 présente des règles de passage des activités vers les RdPs tel que proposées par Störrle (2005). La sémantique proposée est suffisamment simple et le passage assez intuitif pour être facilement compréhensible par les utilisateurs d'UML 2.0, ce qui est un des objectifs de l'auteur. Par contre, cette sémantique ne prend pas en charge les constructions avancées d'UML 2.0. Störrle a conclu ses travaux en mettant en doute l'affirmation du standard OMG concernant les diagrammes d'activité qui suivraient exactement la sémantique intuitive des réseaux de Petri face à la difficulté pour construire un codage simple prenant en compte toutes les constructions de haut niveau introduites dans UML2 dont la TTC, les exceptions et les streams. Tous ces problèmes ont été abordés par l'auteur dans (Störrle et Hausmann, 2005). Dans ce papier nous traitons les problèmes liés au codage de la TTC et proposons une solution.

Dans la spécification d'UML 2.0, les noeuds fork et join sont synchronisés par définition dans le sens où un nœud fork ne peut libérer un jeton que si tous ses chemins sortants peuvent l'accepter. D'autre part, un nœud join ne peut accepter un jeton que si tous ses arcs d'entrée offrent des jetons. Störrle choisit d'assouplir la sémantique du fork pour que ce nœud n'exige l'acceptation de ses jetons que par un seul chemin sortant et stocke les copies des jetons des chemins sortants sans succès, et ainsi évite leur perte. Malgré sa complexité, cette solution transforme la sémantique de la TTC et simplifie la sémantique du fork. Elle l'identifie comme un moyen simple de copie de données au lieu d'un outil de synchronisation

des déclenchements simultanés d'actions. L'objectif de Störrle est de donner une définition simple de la sémantique à base des RdPs classique. Les RdPs avec l'ajout d'arcs inhibiteurs semblent être une solution intéressante notamment dans les situations d'interblocage. Toutefois, cette solution qui ajoute un artifice supplémentaire au RdP correspondant pour exprimer le mécanisme de synchronisation est complexe et ne permettra pas d'intégrer facilement et avec généricité les autres constructions de haut niveau ajoutées à UML 2.0. L'allégement de la condition de franchissement de la TTC par Störrle (Störrle 2005) (voir tableau 1) introduit quelques limites.

L'exemple générique de la figure 3, sous partie de l'exemple de la figure 1, illustre ces problèmes en présentant un diagramme d'activité problématique (figure 3.a) et le RdP correspondant (figure 3.b). L'activité *A* envoie la donnée *d* au nœud fork pour la copier et l'envoyer simultanément aux activités *B* et *C*. L'activité *C* a non seulement besoin de la donnée *d* pour être déclenchée, mais aussi de l'activité *B* qui doit se terminer et lui envoyer un jeton de contrôle *e*. Par conséquent, nous sommes confrontés à une situation de blocage provoquée par une dépendance circulaire. Le noeud fork ne peut libérer ses jetons (ici *d*) que si *B* et le nœud join les acceptent simultanément. La branche droite est ouverte du fait qu'on suppose que la prise d'entrée de *B* est prête pour accepter le jeton *d.* D'autre part, la branche gauche contenant un arc direct de fork vers join n'est pas ouverte du fait que le noeud join ne peut s'exécuter que s'il dispose des jetons transmis par fork et par *B* simultanément car ce sont ses deux entrées directes, donc il nécessite l'offre à la fois des jetons *d* et *e* du fork et de l'activité *B* respectivement. Nous remarquons que l'activité *B* ne peut pas offrir un jeton de sortie du fait qu'elle n'a pas été fournie son jeton d'entrée *d* et donc *B* ne pourra jamais s'exécuter. Nous constatons ainsi que pas tous les chemins à la sortie du fork sont ouverts. En appliquant la contrainte de la TTC, le jeton *d* n'est pas libéré et le fork n'est donc pas exécuté.

Noeuds et arcs	Diagrammes d'activité d'UML	RdPs
Noeuds de contrôle		Fork/Join
Arcs		
Noeuds exécutables	Activité	

Tab. 1 – Règles de passage des diagrammes d'activité d'UML vers les RdPs (Störrle 2005)

L'application des règles de transformation du tableau 1 donne lieu au RdP de la figure 3.b. Le RdP généré ne préserve pas l'interblocage présent dans le diagramme d'activité. Le noeud fork (transition) du RdP de la figure 3.b peut être franchi produisant des jetons de sortie dans les places de sortie *d2* et *d3* indépendamment du nœud join et de sa capacité à accepter les jetons suivant la TTC. Par conséquent, ce RdP exprime une synchronisation locale de *d3* et *e* à la transition join, représentée par l'ellipse à pointillés uniformes, en omettant l'expression d'une synchronisation plus globale des transitions join et fork, symbolisée par une ellipse à pointillés non uniformes.

Dans l'exemple précédant, nous voulions capturer, dans le RdP, l'interblocage observé dans le diagramme d'activité correspondant. Ainsi, le RdP obtenu par application des règles de passage proposées dans (Störrle, 2005) ne garantit pas la préservation du comportement des diagrammes d'activité et n'a pas permis de détecter l'interblocage. Une sémantique plus précise qui offre une synchronisation des transitions et conduit à une synchronisation collective peut être obtenue par les ZSNs.

Dans la section suivante, nous présentons une formalisation des diagrammes d'activité d'UML par les ZSNs préservant la sémantique de la TTC.

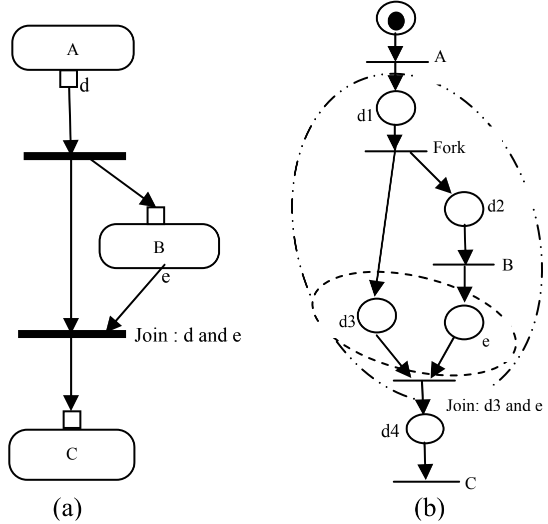

Figure 3. (a) : Diagramme d'activité. (b) : RdP correspondant

4 Préservation de la TTC dans les diagrammes d'activité via les ZSNs

Les RdPs ordinaires utilisés comme cadre sémantique des diagrammes d'activité d'UML peuvent induire des calculs interdits par la TTC. L'objectif de cette contribution est de montrer que le formalisme des ZSNs est plus approprié pour garantir la sémantique de la TTC dans les diagrammes d'activité. En effet, nous proposons par analogie au résultat (tableau 1)

de Störrle (2005) un ensemble de règles permettant de transformer tous les concepts avancés des diagrammes d'activité aux différents éléments syntaxiques des ZSNs en préservant leur sémantique, particulièrement pour la TTC (tableau 2).

Dans la transformation que nous proposons, une distinction est faite entre deux régions dans les diagrammes d'activité: région atomique et région observable. La dernière peut être mappée à un RdP ordinaire suivant les règles de transformation définies par Störrle (2005, voir table 1). Nous appelons région atomique, un sous-diagramme d'activité où les actions sont délimitées par un nœud fork et un noeud join contenant éventuellement d'autres noeuds pouvant être des nœuds forks et/ou des nœuds joins, où tous les chemins conduisent du nœud fork le plus englobant au dernier noeud join qui lui correspond. Ces chemins peuvent contenir tout type de nœuds liés par des arcs. Nous pouvons facilement constater que les régions atomiques que nous définissons sur les diagrammes d'activité d'UML contiennent des threads et donc, nous pouvons les caractériser comme régions concurrentes.

Nous mettons l'accent sur la transformation des régions atomiques (concurrentes) pouvant fonctionner comme un ZSN. En prenant en considération les flots des données et de contrôle ; les noeuds exécutables (les actions : ligne 1 du tableau 2) ainsi que les noeuds concurrents (join et fork : ligne 5 du tableau 2) deviennent des transitions de réseaux, les noeuds de contrôle (decision et merge : ligne 2 du tableau 2) tout comme les prises d'entrée/sortie deviennent des zero places à l'entrée d'une action (ligne 3) et à la sortie d'une action (ligne 4). Deux zero places consécutives générées par les règles de passage sont réduites à une zero place unique évitant d'ajouter une transition auxiliaire, sauf si elles représentent des données différentes, ceci n'étant possible que dans le cas de l'existence d'un comportement de transformation sur l'arc entre deux actions et dans ce cas, on génère une transition auxiliaire qui correspond à ce comportement. Un chemin qui conduit d'un fork à un join sans passer par aucun noeud exécutable sera transformé en une place stable (ligne 9). Les lignes 7 et 10 réduisent deux zero places consécutives, générées par un decision/merge suivi d'une prise ou l'inverse, en une zero place unique. Un autre cas particulier est décrit par la ligne 8, où nous avons deux decisions/merges successifs transformés en deux zero places liées par une transition. Un autre cas problématique, ressemble à celui de la ligne 8, mais cette fois-ci, le chemin englobant les nœuds decision et merge ne contient aucune action. Dans ce cas, une place stable est ajoutée au ZSN correspondant. Une caractéristique très intéressante des ZSNs est que les jetons stables générés à l'intérieur d'une transaction sont gelés jusqu'à la fin de celle-ci où ils sont libérés. Nous utilisons cette notion importante pour créer des situations d'interblocage dans le ZSN qui correspond à un diagramme d'activité présentant un inteblocage non repéré par le RdP classique, en supposant que la région concurrente doit être exécuté en un pas (utilisation des zero places) et en générant par la suite des jetons stables à l'intérieur de la transaction qui d'après le diagramme d'activité sous-jacent doivent être consommés par la transaction elle-même par l'application de la TTC. Dans les cas sans interblocage le ZSN fonctionne parfaitement tout comme le RdP classique.

L'exemple 1 montre le codage du diagramme d'activité de la figure 3.a en utilisant notre approche. Les exemples 2 et 3 représentent deux cas de diagrammes d'activité avec et sans interblocage respectivement ainsi que leur transformation en ZSNs.

Noeuds du diagramme d'activité d'UML	Concepts du ZSN
1. Activité	Transition
2. decision/merge	Zero place
3. Prise d'entrée	Zero place
4. Prise de sortie	Zero place
5. Fork/Join	Transition
6. Arc de flot de donnée Arc de flot de contrôle	Arc Zero place
Sauf: 7.	Zero place
8.	Zero places
9. Fork Join	Place stable
10.	Zero place
À moins que : Il n'y a pas de noeuds exécutables sur le chemin menant du fork au decision ou de merge au join. 11.	

Tab. 2 – La sémantique de la transformation proposée pour les régions concurrentes des diagrammes d'activité d'UML

Exemple 1. Si nous reprenons l'exemple précédent, nous constatons que le ZSN de la figure 4.b, construit en utilisant l'approche de traduction proposée (table 2), préserve l'interblocage, détecté dans le diagramme d'activité de la figure 3.a, par l'exécution des threads déclenchés simultanément d'une façon atomique, et donc par le gel des données sur lesquelles aucun comportement appartenant à la transaction atomique n'est accompli. La région concurrente est délimitée par les transitions fork et join. Pour la génération du ZSN de la figure 4, nous avons suivi les règles de transformations des lignes l, 5, 6, 3, 1, 4, 6, 5 pour la partie droite de la région atomique et la règle 9 pour la partie gauche. Les états invisibles sont composés des zero places z*1* et z*2*, correspondant respectivement aux prises d'entrée/sortie de l'activité *B*. La donnée *d1,* copiée par le nœud fork, pour être passée à la prise d'entrée de l'activité *C*, est associée à une place stable *p2*. Par conséquent, le calcul déclenchant le fork est interdit causant ainsi un blocage juste avant la transition fork. Ce calcul pouvait évoluer dans un état non visible, contenant un zero jeton *z2* et un jeton stable *p2*, ne pouvant plus évoluer. Tout jeton stable généré au cours d'une transaction ne peut être consommé qu'une fois la transaction terminée (propriété de la sensibilisation concurrente des ZSNs : pas de réutilisation des jetons) et cette transaction (fork-join) ne pourra jamais se terminer car sa fin est fonction du jeton stable *p2* généré par la transaction et du zero jeton *z2.*

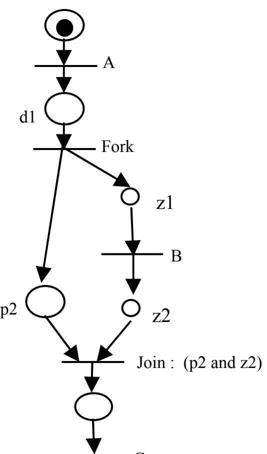

Figure 4. ZSN correspondant au diagramme d'activité de la figure 3.a .

Exemple 2. Dans cet exemple, nous présentons un diagramme d'activité représentant un interblocage dû à l'arc allant du nœud decision au nœud merge et qui sont connectés directement aux nœuds fork et join. Le ZSN correspondant détecte bien l'interblocage en générant un jeton stable non consommable à l'intérieur de la transaction.

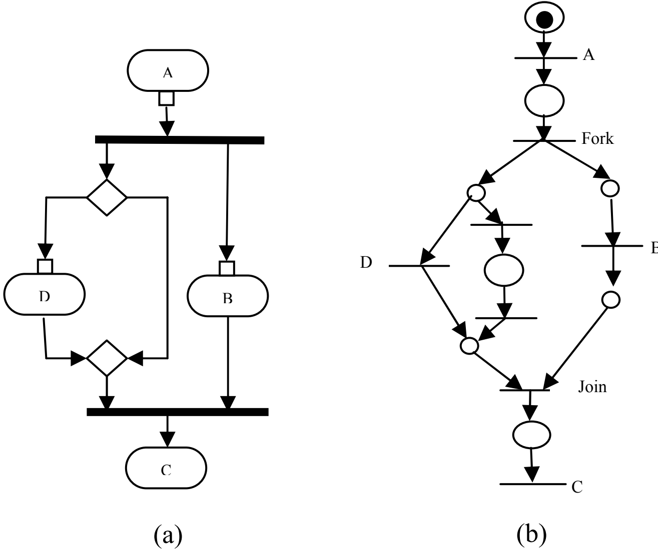

Figure 4. (a) : Diagramme d'activité avec interblocage. (b) : ZSN correspondant

Exemple 3. Dans cet exemple, nous présentons un sous-diagramme d'activité ne représentant aucun interblocage. Deux flots concurrents sont déclanchés par le fork. Une action externe à la région concurrente *F* est traduite par les règles de passage définies dans le tableau 1. Le jeton stable à la sortie de *F* peut très bien être consommé par la transaction car il n'a pas été généré par cette dernière. Le ZSN ne détecte aucun interblocage.

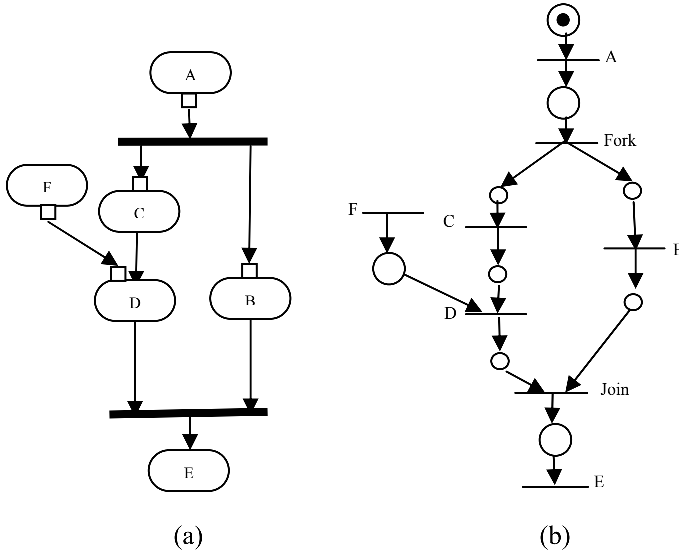

Figure 5. (a) : Diagramme d'activité sans interblocage. (b) : ZSN correspondant

5 Conclusion

Dans UML 2.0, la définition de la syntaxe des constructions a été améliorée. Cependant les formalisations existantes ne prévoient pas l’impact de la nouvelle sémantique sur d'autres concepts des diagrammes d’activité d’UML. Dans ce papier, nous nous sommes concentrés sur la sémantique de la synchronisation dans les diagrammes d’activité d’UML 2.0 en analysant son fondement sur les RdPs ordinaires comme sémantique de base. Nous avons montré que les RdPs ordinaires, censés être un cadre sémantique formel pour les diagrammes d’activité, ne préserve pas la nouvelle sémantique d’UML. Ils définissent une synchronisation locale des transitions, ce qui n’est pas suffisant pour modéliser le concept de la TTC nécessitant une synchronisation globale des nœuds fork et join des diagrammes d’activité.

L'objectif principal de ce travail est de proposer une transformation générique des diagrammes d’activité aux ZSNs. Cette dernière devra préserver la sémantique opérationnelle des diagrammes d’activité, particulièrement le principe de la TTC et la synchronisation entre les noeuds fork et join et supporter ainsi le flot des contrôles/données et la concurrence.

Les streams et les exceptions sont deux autres nouvelles constructions ajoutées aux diagrammes d’activité d’UML 2.0. Les formalisations existantes dont celle de Störrle, ne permettent pas de représenter ces deux concepts. Dans de futurs travaux, nous donnerons une approche de formalisation via les ZSNs plus complète qui permettra d’observer de façon atomique (en introduisant des zero places) ces deux nouveaux concepts tout en préservant leur sémantique.

Afin de permettre la vérification et la validation des diagrammes UML, notre travail a été subdivisé en deux étapes : une première étape a consisté en la transformation de ces diagrammes en ZSNs. Ces derniers ont un cadre sémantique formel basé sur la logique des tuiles, une extension de la logique de réécriture introduisant la notion d'effet de bord et de contraintes dynamiques sur les termes auxquels une règle est applicable. Une deuxième étape, faisant l'objet d'un travail futur, consistera en la projection des ZSNs dans la logique de réécriture afin d'exploiter ses outils de vérification et de validation tel que Maude et son model checker.

Dans ce travail, nous avons permis de voir de façon atomique, la région délimitée par les nœuds fork et join et ce grâce aux ZSNs. Dans de futurs travaux, nous prendrons plus en considération son caractère concurrent par l’utilisation des ZSNs et des RdPs hiérarchiques.

Références

L. Baresi et M. Pezzè. Improving UML with Petri nets. In Proceedings of ETAPS2001 Workshop on Uniform Approaches to Graphical Process Specification Techniques, March 2001.

F. Belala et S. Boufenara. Towards a collaboration diagrams formalization via zero-safe nets. ICMSAO’09, in press.

C. Bock. UML 2 Activity and Action Models: Object Nodes. In J. Object Technology, 3(1):27–41. January/February 2004. Available at www.jot.fm.

A. Bokhari et S. Poehlman. Translation of UML models to object coloured Petri nets with a view to analysis. In. Proc. of the Eighteenth International Conference on Software Engi-

neering and Knowledge Engineering (SEKE'06), San Francisco Bay, USA, July 5-7, 2006, pp.568-571.

R. Bruni et U. Montanari. Zero-safe Nets, or transition synchronization made simple. In C. Palamidessi and J. Parrow, Eds., Proceedings EXPRESS'97, ENTCS 7, Elsevier, 1997

R. Bruni. Tile Logic for Synchronized Rewriting of Concurrent Systems. Ph.D. Thesis: TD-1/99, March 99.

T. Clark et A. Evans. Foundations of the Unified Modeling Language. In NFM97: 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, September 1997.

A. Egyed et D. Wile. Statechart Simulator for Modeling Architectural Dynamics. In Proceeding the second IEEE/IFIP Conference on Software Architecture, pp. 87-96, 2001.

H. Eshuis et R.J. Wieringa. Comparing Petri Net and Activity Diagram Variants for Workflow Modelling: A Quest for Reactive Petri Nets. In Petri Net Technology for Communication-Based Systems: Advances in Petri Nets. Lecture Notes in Computer Science 2472. Springer, Berlin/Heidelberg, Germany, pp. 321-351. ISBN 9783540205388, 2003.

D. Harel et A. Naamad. The STATEMATE Semantics of statecharts. In ACM Transactions on Software Engineering and Methodology, vol. 5, n°4, pp.293-333, 1996.

S.–K. Kim et D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France, B. Rumpe Eds, UML'99- The Unified Modeling Language beyond the standard, Second International Conference, Fort Collins, CO, USA, October 28-30, Proceeding vol. 1723 de LNCS, Springer 1999.

OMG Unified Modelling Language: Superstructure. Final adopted spec, version 2.0, 2003-08-02. Technical report: Object Management Group. November 2003. Available at www.omg.org, downloaded at November 11th, 2003.

L. Paltor et J. Lilius. Formalizing UML state machines for model checking. In R. France, B. Rumpe Eds, UML'99- The Unified Modeling Language beyond the standard, Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceeding vol. 1723 de LNCS, Springer 1999.

J.A. Saldhana et S.M. Shatz. UML diagrams to object Petri nets: An approach for modeling and analysis. In Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE), pp.102-110, July 2000.

S. Sarstedt aus Gummersbach. Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Diagrams. Thèse d'état, 2006. Available at vts.uni-ulm.de/docs/2006/5643/vts_5643_7444.pdf.

H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. In Electr. Notes Theor. Comput. Sci. 127(4): 35-52, 2005. Available at www.pst.informatik.uni-muenchen.de/-stoerrle.

H. Störrle et J. H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In Software Engineering, pp.117-128, 2005.Hölldobler, B. et E-O. Wilson (1990). *The Ant*s. Berlin: Springer Verlag.

Les ZSNs pour la formalisation des activités d'UML

Summary

Petri nets have been used as a formal semantic framework for UML activity diagrams. With UML 2.0 extensions, they are no more able to express high level constructions in activity diagrams, namely the traverse-to-completion concept. This one requires a global synchronization between the fork and join nodes which is totally absent in classical Petri nets which are based on local transition synchronization. To preserve activities behaviors, we propose zero-safe nets as a richer and more adapted semantic framework for activity diagrams by giving a generic mapping from activities to this Petri nets class. The proposed mapping preserves activities operational semantics while focusing on traverse-to-completion principle as well as synchronization between fork and join nodes and therefore, covering control/data flows and fork triggered threads concurrency.