# Vers l'Exécutabilité des Modèles de Procédés Logiciels[i]

Reda Bendraou*, Marie-Pierre Gervais **, Xavier Blanc **, Jean-Marc Jézéquel*, ***

* INRIA-Rennes Bretagne Atlantique
Campus de Beaulieu
F-35042 Rennes Cedex (FRANCE)
prénom.nom@inria.fr

**Laboratoire d'Informatique de Paris 6
104 Avenue du Président Kennedy
Paris F-75016 (FRANCE)
prénom.nom @lip6.fr

*** IRISA, Université Rennes 1
Campus de Beaulieu
F-35042 Rennes Cedex (FRANCE)
Jean-Marc.Jezequel@irisa.fr

**Résumé.** L'un des enjeux majeurs de l'ingénierie dirigée par les modèles est d'augmenter la productivité des logiciels à travers la manipulation de modèles dès les premières phases de développement. La finalité étant de pouvoir utiliser les modèles non seulement pour des fins de compréhension et de description mais aussi de production. Les modèles de procédés de développement logiciels sont au cœur de la démarche de construction du logiciel. Cependant, à ce jour, ils ne sont utilisés que pour documenter les procédés et demeurent des modèles contemplatifs. Le but de nos travaux est de les rendre productifs, permettant ainsi une meilleure coordination entre les équipes de développement, l'automatisation des tâches répétitives et non interactives et une gestion plus efficace des moyens utilisés pendant les phases de développement. A cet effet, nous proposons UML4SPM, un langage exécutable et orienté modèle pour la modélisation de procédés de développement logiciel.

## 1 Introduction

Depuis qu'il a été largement établi que la qualité et la rapidité de construction des logiciels pouvaient être directement influencées par les procédés de développement suivis [Montangero (1999)], la communauté du génie logiciel ne cesse de voir fleurir ce que l'on appelle communément, les *Langages de Modélisation de Procédés* (LMPs). Les éditeurs de logiciels ont très vite compris cet enjeu et leur intérêt à vouloir capturer leur *Procédés de Développement Logiciel (PDL)* sous forme de *Modèles de Procédés (MP)* n'a cessé de s'accroître. L'objectif sous-jacent est bien sûr de pérenniser une connaissance précieuse et répétitive de leurs procédés mais aussi de s'en servir à des fins de compréhension, d'analyse et d'exécution.

[i] Ce travail est financé en partie par le réseau d'excellence AOSD-Europe ainsi que par le projet européen Modelplex, contrat IST-3408

Cependant, si on regarde de plus près les LMPs proposés par la littérature, force est de constater que, jusqu'à, présent aucun d'entre eux n'a réussi à s'imposer comme le *Langage* pour la modélisation de PDL. De plus, rares sont les langages pouvant concilier à la fois compréhensibilité, flexibilité et exécutabilité, critères pourtant si importants lors de la définition d'un PDL [Curtis (1992)]. Dans la littérature, deux familles de LMPs sont proposées, descriptifs et exécutables. Les LMPs dits exécutables sont le plus souvent basés sur des langages de programmation, (par ex, APPL/A (Ada), MERLIN (Prolog)) [Sutton (1995), Junkermann (1994)], d'autres sont plus similaires à des formalismes tels que les Réseaux de Petri (par ex, SPADE) [Bandinelli (1993)] ou bien sont basés sur des règles d'inférences (par ex, MARVEL) [Kaiser (1990)]. Ces langages connus pour être les LMPs de première génération, imposent de représenter un modèle de procédé logiciel sous forme d'un programme informatique. Cela implique une connaissance parfaite du langage de la part des modélisateurs de procédés, ce qui n'est pas toujours le cas. Même si ces modèles (programmes) de procédés offrent l'avantage d'être exécutables, ils sont de trop bas niveau pour être utilisés, compris ou modifiés par les acteurs du procédé de développement.

Les LMPs descriptifs sont apparus afin de pallier ce manque d'abstraction qui caractérisait les LMPs exécutables. En effet, en élevant le niveau d'abstraction et en utilisant des notations graphiques et des diagrammes au lieu du code, les modèles de procédés devenaient plus lisibles et la compréhension ainsi que le raisonnement autour de ceux-ci étaient rendus plus aisés. Cependant, ces langages sont utilisés afin de documenter les procédés et non pas pour coordonner leur exécution. Ce dernier besoin n'étant pas satisfait, ces LMPs n'ont rencontré à leur tour qu'une faible adoption par les acteurs industriels. Un exemple de ces LMPs descriptifs est le standard SPEM (Software Process Engineering Metamodel) de l'OMG (Open Management Group) [OMG (2002)] [OMG (2007c)] qui, même dans sa nouvelle version, ne satisfait pas le critère d'exécutabilité [Bendraou (2007c)].

Dans cet article nous tentons d'allier compréhensibilité et exécutabilité pour une meilleure efficacité des LMPs. Nous pensons qu'un LMP doit être non seulement utilisé pour documenter les procédés logiciels mais aussi pour assurer leur exécution. Par exécution, nous visons l'automatisation des taches répétitives et non interactives du procédé telles que la vérification de pré et de post conditions propres à l'exécution de chaque activité, la régulation ainsi que le contrôle automatique des flux de données et d'artefacts, l'affectation des agents aux rôles, la connexion avec des applications métier, etc. A cet effet, nous proposons UML4SPM, un *Langage de Modélisation de Procédés* de *Développement Logiciel* conçu selon une approche normative et visant à répondre aux nouvelles attentes de la communauté logicielle. Ce dernier devra satisfaire un certain nombre d'exigences telles que l'expressivité, la compréhensibilité, la modularité, et comme priorités premières, une simplicité d'utilisation ainsi qu'une prédisposition des modèles instances de ce langage à être simulés et exécutés.

Pour ce faire, dans la prochaine section, nous commençons par introduire notre langage, UML4SPM. Ce dernier vient sous forme d'un méta-modèle MOF qui étend un sous-ensemble du standard UML2.0 Superstructure [OMG (2007b)], un ensemble de diagrammes et de notations ainsi qu'une sémantique propre à la modélisation des PDLs. Pour l'exécution des modèles de procédés UML4SPM, nous avons exploré deux approches. Une première approche dite *compilée* et qui est présentée en détails dans [Bendraou (2007a)], consiste à projeter les modèles UML4SPM vers l'espace technologique du langage WS-BPEL, un standard pour l'exécution des modèles de procédés métiers (*i.e., Business Processes*). Une seconde approche, dite *interprétée*, que nous présentons ici en section 3, consiste quant à elle à définir un *Modèle d'Exécution* pour UML4SPM. Ce dernier est utilisé comme base pour

l'implantation du moteur d'exécution d'UML4SPM. Pour cela, nous avons étudié deux possibilités. La première consiste en une implantation Java en utilisant le pattern *Visiteur*. La seconde utilise *Kermeta*, un méta-langage exécutable [Muller (2005)]. L'objectif derrière étant de pouvoir exécuter les modèles UML4SPM directement et sans aucune étape intermédiaire. Une discussion sur notre retour d'expérience concernant ces approches d'exécution est donnée en section 4. Nous verrons que même si l'approche compilée offre l'avantage de réutiliser les outils performants du domaine des procédés métiers, elle manque énormément de flexibilité. Cela se traduit par un manque de support de l'interaction humaine ainsi qu'un besoin de régénérer du code à chaque modification du modèle de procédé UML4SPM, nécessitant d'interrompre l'exécution. L'approche interprétée quant à elle permet de remédier à cela en proposant un couplage fort entre le modèle et son exécution rendant ainsi possible sa modification pendant le déroulement du procédé. Finalement, la section 5 conclut cet article et dessine les futures perspectives de ce travail.

# 2 UML4SPM

Notre langage UML4SPM (UML2.0-Based Language For Software Process Modeling) vient sous la forme d'un méta-modèle MOF qui étend un sous-ensemble du standard UML2.0, une notation graphique et un ensemble de diagrammes ainsi qu'une sémantique appropriée pour la modélisation de procédés logiciels. Avant de présenter le méta-modèle, nous introduisons les motivations qui nous ont poussés à choisir UML comme socle de notre langage.

## 2.1 Réutilisation d'UML2.0 comme base pour UML4SPM

Le choix de prendre UML2.0 comme base de notre langage a été motivé par les raisons suivantes:

- Dans l'un de nos précédents travaux [Bendraou (2005)], nous avons évalué UML2.0, plus précisément, les packages *Activity* et *Actions* avec les principales exigences de LMP qui sont: expressivité, compréhensibilité, modularité et exécutabilité [Jaccheri (1999)]. UML2.0 a démontré de réels avantages concernant les critères d'expressivité, de compréhensibilité et d'exécutabilité. En effet, les *Activités* dans UML2.0 ont radicalement changé par rapport aux précédentes versions du standard (i.e., UML1.x). Désormais, elles offrent un ensemble de concepts assez riche qui pourra être utilisé non seulement pour représenter les procédés mais aussi pour les automatiser [Hausmann (2005)]. Le standard fournit quatre packages d'actions avec une sémantique opérationnelle qui pourront être utilisés pour modéliser les actions automatiques du procédé (par ex, invocation d'activités et d'outils, gestion des exceptions, traitement des événements, etc.). Pour ce qui est de la compréhensibilité, UML fournit un ensemble de notations graphiques et de diagrammes, aujourd'hui assez bien accepté dans l'industrie du logiciel. Dans UML4SPM, nous réutilisons ces digrammes afin de représenter le procédé selon différents points de vue. Le diagramme d'activité est utilisé pour donner une représentation du flux de travail, le diagramme de classe pour montrer les différentes relations liant les éléments du procédé (i.e., héritage, association, dépendances, etc.) et le diagramme de cas d'utilisation pour souligner les responsabilités de chaque rôle dans le procédé.

- La deuxième raison qui nous a poussés à opter pour UML vient du fait que c'est un langage standard. En effet, aucun langage n'a réussi à s'imposer comme *le langage* de modélisation que se soit au niveau de l'industrie ou bien dans le domaine académique. De plus, de nombreux outils UML ainsi que des supports techniques sont proposés et bon nombre de personnes sont déjà formées à ce langage. Tout cela représente un atout non négligeable à prendre en considération. Dans ce qui suit, nous présentons le méta-modèle de notre langage.

## 2.2 Le Méta-modèle UML4SPM

Le méta-modèle UML4SPM est composé de deux packages. Le premier package, "UML4SPM Process Structure", introduit les concepts de base pour la modélisation de procédés (figure 1). Le package "UML4SPM Foundation" quant à lui, contient un sous-ensemble de concepts du standard UML2.0, utiles pour assurer tous les mécanismes liés à la coordination des activités, à la gestion des exceptions et à l'expression de concepts avancés (i.e., itérations, conditions, etc.). Dans ce qui suit, nous présentons brièvement le package "UML4SPM Process Structure".

### 2.2.1 Le package UML4SPM *ProcessStructure*

La brique de base de tout modèle UML4SPM est la notion de *Software Activity* (cf. fig 1). Elle représente tout effort à réaliser pendant le développement. Elle a une propriété "description" qui décrit brièvement le but de cette activité, "kind" pour statuer si l'activité est automatisable ou bien sous la responsabilité d'un agent et "priority" afin de souligner l'importance de l'activité dans le procédé. La propriété "isInitial" est utile pour spécifier si l'activité en question représente le contexte courant (i.e., conteneur) du procédé ou non. Ainsi, tout modèle UML4SPM devra impérativement avoir une *Software Activity* ayant comme valeur de sa propriété "isInitial" égale à vrai. Celle-ci contiendra toutes les autres activités du procédé.

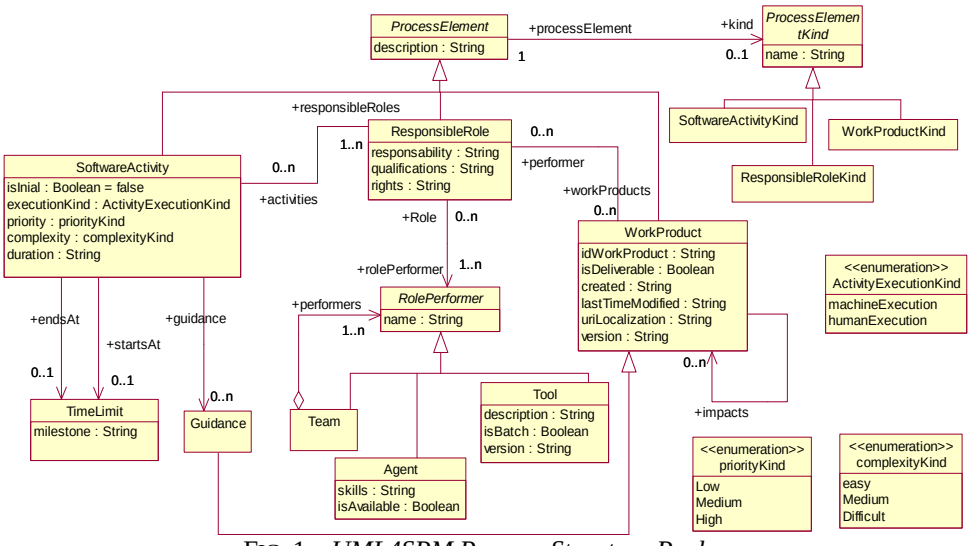


FIG. 1 – *UML4SPM Process Structure Package*

Une *Software Activity* est sous la responsabilité d'un ou plusieurs *Responsible Role* qui devront assurer toutes les étapes de l'activité. Un *Responsible Role* est décrit en termes de ses responsabilités pendant la réalisation de l'activité, de ses qualifications ainsi que de ses droits d'accès aux différents artefacts utilisés pendant le développement. Un *Responsible Role* peut être affecté à un ou plusieurs *Role Performer*. Ceux-ci seront désignés durant les phases d'exécution du procédé. Cependant, s'ils sont connus par avance, ils peuvent être définis durant la phase de spécification du procédé. Un *Role Performer* peut être un *Agent*, un outil (*Tool*) ou bien une équipe (*Team*) qui à son tour peut être composée d'autres agents ou bien d'autres équipes. Une contrainte OCL capture le fait qu'une équipe ne puisse pas contenir d'outil (`context Team inv: self.performers->forAll (roleperformer|not roleperformer.isKindOf (Tool)`).

Un autre concept de base, est celui de *WorkProduct*. Il représente toute entité créée, utilisée ou modifiée pendant les différentes étapes du procédé. Il a un identifiant unique "idWorkProduct", une propriété booléenne "isDeliverable" pour indiquer si le *WorkProduct* est juste un artefact intermédiaire ou bien un livrable du procédé, une propriété "uriLocalization" permettra d'atteindre le *WorkProduct* pendant l'exécution ainsi que d'autres propriétés concernant la dernière date de modification du *WorkProduct*, sa version, etc.

Pour les trois éléments essentiels du procédé i.e. *Software Activity*, *Responsible Role* et *WorkProduct*, il est possible d'en définir différents types selon le projet ou la méthodologie utilisée pendant le développement. Ceci est assuré grâce à l'association entre la méta-classe abstraite *Process Element* et *Process Element Kind*. Cette association est redéfinie ensuite (elle n'est pas présentée ici) afin de contraindre l'affectation des types par élément du procédé (i.e. afin qu'un type de *Software Activity*, par ex, "Itération", ne puisse pas être affecté comme type d'un *WorkProduct* ou d'un *Responsible Role*).

Toutes les méta-classes présentées jusqu'ici représentent les éléments essentiels à la définition d'un modèle de procédé. Ces méta-classes, leurs propriétés ainsi que leurs associations sont le résultat d'une étude détaillée de la littérature que nous avons menée dans le domaine de la modélisation de procédés. C'est aussi une prise en compte de toutes les exigences auxquelles nous avons été confrontées de la part de nos partenaires industriels dans le projet Modelplex [ModelPlex (2007)] ainsi que dans notre participation aux efforts de standardisation à l'OMG dédiés à la définition du standard SPEM2.0 [OMG (2007c)].

Cependant, à ce niveau, il n'est pas possible de représenter tous les mécanismes censés exprimer l'ordre/synchronisation d'exécution des activités, la gestion des exceptions, les itérations, les événements, etc. Tous ces concepts sont regroupés dans le package "UML4SPM Foundation". Il s'agit d'un sous-ensemble du standard UML2.0 Superstructure que nous avons identifié et que nous réutilisons dans UML4SPM. Nous le décrivons dans la section ci-dessous.

### 2.2.2 Le package UML4SPM *Foundation*

Dans cette section nous donnons un bref aperçu des concepts UML2.0 que nous avons réutilisés dans UML4SPM. Pour ce faire, nous utilisons la relation d'héritage afin de lier les méta-classes des deux méta-modèles. L'ensemble des méta-classes d'UML2.0 que nous avons identifié comme base pour UML4SPM regroupe principalement des éléments des packages *Activity* et *Actions* ainsi que d'autres éléments tels que la notion *d'Artifact*. Dans ce qui suit, nous présentons les méta-classes *Activity* et *Artifact*.

- **Activity:** dans UML2.0, une *Activity* est définie comme étant un comportement paramétré, spécifié en terme de coordination d'actions [OMG (2007b)]. Cette coordination est assurée grâce à deux notions: le *contrôle de flux* et le *contrôle de données*. Il est aussi possible d'exprimer des nœuds de contrôle à l'intérieur d'une activité. Ces derniers sont utilisés pour décrire des choix (*Decision Node*), du parallélisme (*Fork Node*), de la synchronisation (*Join Node*), etc.

En faisant hériter *Software Activity* de la méta-classe UML2.0 *Activity*, nous prenons avantage de l'expressivité du standard UML. Il est maintenant possible de composer une *Software Activity* avec d'autres *Software Activity*, de lui spécifier des pré et post conditions, d'y inclure des actions avec une sémantique opérationnelle, etc. Du fait qu'une *Software Activity* hérite indirectement de UML2.0 *Classifier*, elle pourra être enrichie avec de nouvelles propriétés ou bien de nouvelles opérations.

- **Artifact:** le standard UML2.0 définit un artifact comme étant un *Classifier* représentant une entité physique. Cela peut être un modèle, un fichier de code source, des règles de transformation, un exécutable, etc. La méta-classe UML4SPM *WorkProduct* hérite d'*Artifact*. De ce fait, les *WorkProducts* peuvent être utilisés comme paramètres d'entrée et de sortie des actions mais aussi des *Software Activities* en tant que *ActivityParameterNode*. Il est aussi possible de leur définir de nouvelles propriétés et opérations.

Par manque d'espace, nous ne pourrons pas aborder tous les aspects liés à notre langage. Une présentation détaillée de chacun de ses concepts ainsi que l'ensemble de la notation et des diagrammes UML4SPM sont donnés dans [Bendraou (2007b)].

Concernant l'évaluation d'UML4SPM, dans [Bendraou (2006)] nous comparons dans un premier lieu notre langage aux principales exigences des LMP [Curtis (1992), Jaccheri (1999)]. Par la suite nous évaluons son expressivité on utilisant le benchmark "ISPW-6 Software Process Example" [Kellner (1991)]. L'exemple vient sous la forme d'une partie obligatoire et une autre optionnelle. Avec UML4SPM nous avons réussi à modéliser tous les aspects du procédé, exception faite de quelques éléments de la partie optionnelle. Ceux-ci exigent la présence de concepts qui prennent en compte la mesure des temps d'exécution, l'identification des activités les plus critiques, la prise en compte des réactions et des communications entre agents, etc.

Dans la partie qui suit, nous présentons notre démarche pour l'exécution des modèles de procédés UML4SPM.

# 3 Exécution des modèles de procédés UML4SPM

Pour l'exécution des modèles de procédés UML4SPM, nous avons abordé deux approches. La première, présentée dans [Bendraou (2007a)], consiste à explorer la faisabilité de projeter les modèles UML4SPM vers des formalismes de procédés métiers (i.e., Business Processes) puis de les exécuter (i.e., approche compilée). L'idée derrière étant de réutiliser la maturité du domaine des procédés métiers. Pour ce faire, nous avons défini des règles de correspondances d'UML4SPM vers WS-BPEL (Web Services Business Process Execution Language) [WSBEPL (2007)]. Notre choix pour BPEL n'était pas complètement naïf étant donné que ce dernier est le standard pour l'exécution des procédés métiers. Cela permet la réutilisation des outils ainsi que de la documentation et supports techniques qui gravitent autour de ce standard.

La seconde initiative, que nous détaillons ci-dessous, consiste à définir un modèle de comportement décrivant la sémantique d'exécution des concepts UML4SPM (i.e., approche interprétée). Ainsi, les modèles de procédés UML4SPM pourront être exécutés directement. Une discussion résumant nos retours d'expériences sur ces deux approches d'exécution est proposée en section 4.

### *Le Modèle d'exécution d'UML4SPM*

Dans cette approche, afin d'exécuter les modèles de procédés UML4SPM, nous définissons un modèle de comportement visant à décrire la sémantique d'exécution des concepts de notre langage. Ce modèle de comportement, nommé *modèle d'exécution*, reprend le méta-modèle UML4SPM en lui rajoutant les opérations nécessaires à l'exécution des éléments du modèle. Ainsi, le comportement de chaque concept est décrit par un ensemble d'opérations. L'enchaînement d'exécution de ces opérations représente la sémantique comportementale du concept en question.

Le but de cette approche est de permettre une exécution directe des modèles UML4SPM dès que ceux-ci seront édités, sans aucune étape intermédiaire. UML4SPM étant basé sur UML2.0, pour la spécification de ce modèle de comportement, nous nous sommes basés sur les efforts de standardisation en cours à l'OMG nommés *Foundation of Executable UML* et auxquels nous participons [OMG (2005)]. L'objectif de cette initiative est d'identifier un sous-ensemble d'UML2.0 ayant une sémantique opérationnelle qui serait assez complet et suffisant pour la définition de modèles UML exécutables soit par interprétation, soit par transformation. Cette spécification était encore à l'état de draft lors de l'écriture de cet article et aucune implantation n'est encore fournie ou prévue pour le moment.

Pour la réalisation du modèle d'exécution d'UML4SPM, nous avons exploré deux approches. La première tend à séparer la définition des modèles de procédés de la spécification de leur exécution. Pour cela nous fournissons une implantation Java du modèle d'exécution en utilisant le pattern *Visiteur* [Gamma (1994)]. La seconde, une solution plus orientée modèle, a pour but de rajouter du comportement directement au niveau des éléments du méta-modèle UML4SPM. Pour cela, nous utilisons *Kermeta*, un méta-langage exécutable [Muller (2005)]. Par la suite, nous décrivons chacune de ces deux approches.

## 3.1 Réalisation du modèle d'exécution: l'approche Visiteur

Cette approche s'inspire du pattern *Visiteur* [Gamma (1994)] et vise à découpler la définition des éléments UML4SPM de la spécification de leur comportement au moment de l'exécution. Ainsi, pour chaque concept du méta-modèle UML4SPM ayant une sémantique opérationnelle, nous définissons une *Classe d'Exécution* dans le modèle d'exécution (figure 2). Chaque classe d'exécution contient l'ensemble des opérations qui une fois implanté, reproduit le comportement de l'élément UML4SPM. Le pattern *Visiteur* requiert d'implanter une opération "visiter" dans la classe visiteur et une opération "accepter" dans la classe visitée. Dans le modèle d'exécution, les classes d'exécution ont une association dirigée vers l'élément du méta-modèle UML4SPM pour lequel elles définissent un comportement opérationnel. Ceci reste en accord avec le pattern *Visiteur* et permet d'ajouter du comportement aux éléments UML4SPM sans pour autant modifier le méta-modèle.

A l'exécution, si une classe d'exécution a besoin de récupérer une valeur d'une propriété d'un élément du modèle UML4SPM, elle le fait directement au niveau du modèle à travers

l'association qui lie les deux éléments. Les propriétés ainsi que leurs valeurs ne sont pas redéfinies au niveau de la classe d'exécution. Cette approche permet donc au modèle d'exécution d'être toujours à jour avec le modèle de procédé UML4SPM.

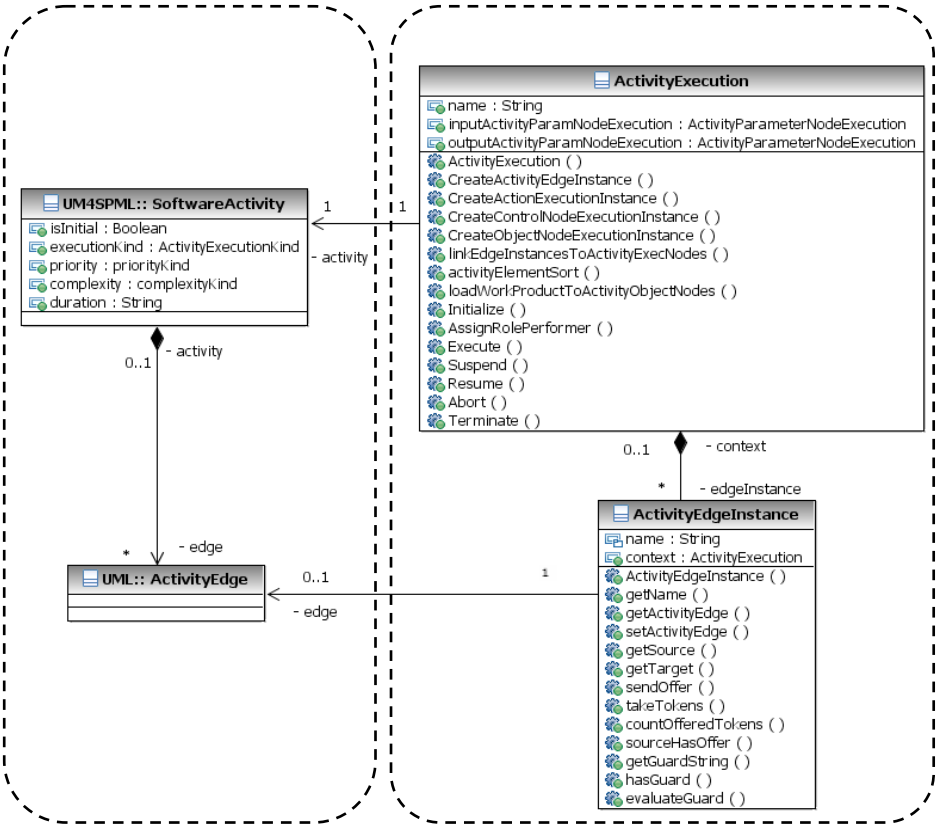


FIG. 2 – *Le modèle d'exécution d'UML4SPM: un exemple*

En plus de la définition du modèle de comportement, nous avons implanté l'ensemble des opérations des principales classes d'exécutions en Java. Celles-ci regroupent les classes relatives aux éléments UML4SPM mais aussi celles des *Activités* et *Actions* UML2.0. Le résultat est un interpréteur (moteur d'exécution) UML4SPM qui prend en entrée des modèles de procédés UML4SPM et qui les exécutent selon le comportement décrit par le modèle d'exécution que nous avons défini. Le tout sans aucune configuration ni étapes intermédiaires. Tous les aspects liés aux actions, flux de contrôle, flux d'objets, concurrence, choix, ont été implantés selon la sémantique décrite par UML2.0. Cela rend donc notre interpréteur parfaitement réutilisable pour l'exécution des diagrammes d'activités UML2.0. Nous avons aussi introduit le moyen d'interagir avec l'utilisateur en ajoutant des actions permettant de spécifier l'interaction en cas de choix, ainsi que les actions non automatisables.

*Exemple d'un modèle de procédé UML4SPM et de son exécution selon l'approche interprétée Visiteur.*

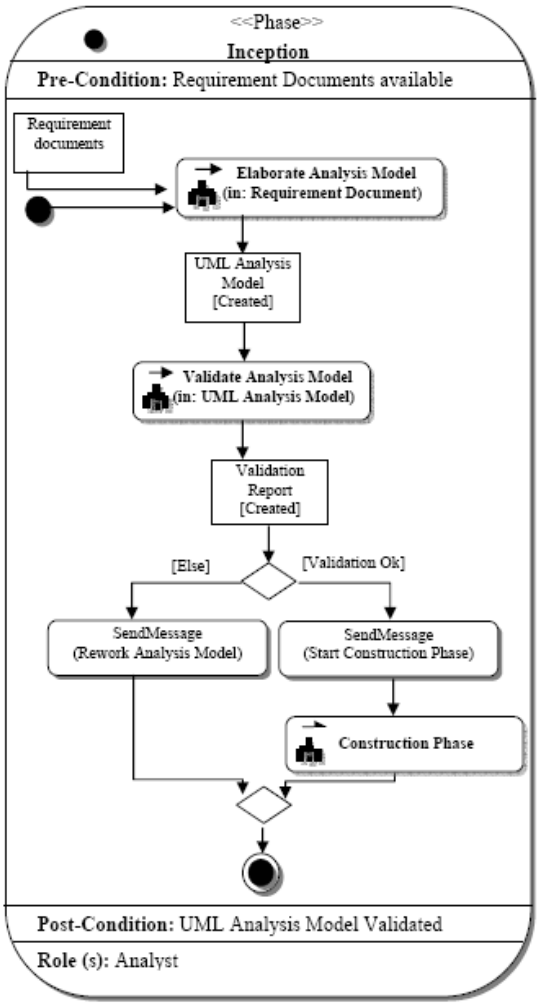


FIG. 3 – *Un exemple de modèle de procédé modélisé avec UML4SPM.*

La figure 3 montre un exemple concret d'une partie d'un procédé de développement logiciel modélisé avec UML4SPM et qui regroupe un ensemble de routines pouvant être automatisées.

Comme on peut le remarquer, l'activité « Inception » représente le contexte courant du procédé (celle orchestrant toutes les autres activités et WorkProducts du procédé). Cela se distingue grâce au point noir sur le coin gauche en haut de l'activité. L'activation ainsi que les appels vers les autres activités sont assurés grâce à l'utilisation de *CallBehaviorActions*. Les appels peuvent être synchrones (flèche pleine, ex. « EloborateAnalysisModel ») ou bien

asynchrones (flèche coupée, ex. « Construction Phase »). Les paramètres de l'appel sont notés avec le nom de l'activité appelée et peuvent être de type *in*, *out* or *inout*.

Un autre aspect important est l'utilisation des nœuds de contrôle tels que les *Decision Node* (branchement conditionnel) ou les *Merge Node* (acceptation d'un flux parmi plusieurs) afin de réguler le flux de données à travers les différentes actions de l'activité. Ces derniers peuvent être renforcés par l'utilisation de gardes sur les flux d'objets reliant deux actions (flèches entre différentes actions, ex. [Validation Ok]). Finalement, on peut noter l'utilisation de concepts *d'OpaqueActions* afin de modéliser des interactions humaines ou bien une connexion à une application métier (ex. Code Java, Service Web, Interface applicative, etc.).

Une fois ce modèle de procédé chargé par l'interpréteur d'UML4SPM, une première étape consiste à le parcourir et à instancier pour chaque concept (ex. *Initial Node*, *Object Flow*, *CallBehaviorAction*, etc), la classe exécutable lui correspondant dans le modèle d'exécution UML4SPM. Ensuite, la méthode « execute() » de l'activité initiale est appelée. Le flux de contrôle ainsi que celui des données sont réalisés en suivant la sémantique des diagrammes d'activités UML2.0. De même pour les actions, pour lesquelles nous avons définit une sémantique d'exécution parfaitement en accord avec UML2.0. Pour chaque type d'action, une implantation de sa méthode « doAction() » est réalisée et servira à activer le comportement l'action.

Dans l'exemple montré ci-dessus, la majorité des aspects sont automatisés, i.e., vérification de l'état des WorkProducts, leur chargement dans les inputs des actions, les appels vers les activités (synchrones ou asynchrone), vérification des types d'entrées et de sorties de chaque activité, l'évaluation des gardes, des pré et post conditions ainsi que des branchements conditionnels. Concernant les *OpaquesActions*, pour l'instant nous offrons une implantation qui permet d'exécuter du code Java à la volée sans interrompre l'exécution du procédé. Cela permet par exemple d'appeler une interface graphique pour recueillir les choix ainsi que les entrées des agents du procédé.

## 3.2 Réalisation du modèle d'exécution: l'approche Kermeta

Dans cette approche, nous utilisons Kermeta, un méta-langage exécutable afin de définir le comportement des éléments UML4SPM directement au niveau du méta-modèle. Kermeta est construit comme une extension du méta-modèle MOF lui rajoutant la partie description du corps des opérations. Ainsi, tout méta-modèle instance de MOF pour être utilisé par un programme Kermeta afin de lui définir un comportement exécutable. Le méta-modèle UML4SPM étant un méta-modèle MOF, c'est aussi dans un certain sens un « programme » Kermeta valide, mais qui ne contient aucune instruction (du point de vue Kermeta, il n'y a que des déclarations de classes sans comportement, i.e., juste la structure). Kermeta permet alors de tisser dans UML4SPM un aspect « interpretation » de manière à rendre un modèle UML4SPM directement exécutable. Sémantiquement, l'effet de ce tissage revient à ajouter une méthode « execute() » dans chacune des classes de UML4SPM, mais sans toucher physiquement au méta-modèle lui-même. Ce type de tissage, qui est une simple « introduction » dans la terminologie orientée aspects, est supportée en Kermeta par un mot clé, comme illustré ci-dessous :

```
@aspect "true"
class ActivityEdge {
  operation execute(context: InterpreterContext):Void is do   {
```

```
//…
  }
  operation sendOffer()  : Void is do {…}
  operation takeTokens()  : Void is do {…}
      // …
}
```

Dans les cas où une plus grande expressivité serait nécessaire (en particulier vis-à-vis du langage de point de coupe *(pointcut)*), Kermeta fournit un *framework* de tissage de modèle [Morin (2007)] permettant de customiser à volonté la composition du méta-modèle de base avec l'aspect souhaité. Ceci permet en particulier de gérer facilement et d'encapsuler de façon tout à fait modulaire la problématique des points de variation dans la sémantique d'UML4SPM : ceux-ci sont définis comme des aspects qui sont tissés à volonté dans le méta-modèle UML4SPM.

De façon similaire, Kermeta permet de tisser des contraintes OCL dans le méta-modèle UML4SPM : si l'on a décrit ces contraintes dans un fichier *StaticSemantics.ocl*, il suffira pour les tisser dans le méta-modèle d'écrire en Kermeta :

```
require "StaticSemantics.ocl"
```

Par exemple, considérons le tissage de la contrainte :

```
Context Team inv: self.performers->forAll (roleperformer|not role-
performer.isKindOf (Tool))
```

Du point de vue de la sémantique de Kermeta, après le tissage consécutif à l'instruction *require*, cette contrainte sera interprétée comme un invariant de la classe *Team*.

Pour revenir à l'exemple présenté en figure 3, un programme Kermeta aura pour tâche de charger le méta-modèle UML4SPM, de lui tisser les aspects interprétations définis eux aussi en Kermeta et en respect avec le modèle d'exécution UML4SPM ainsi qu'éventuellement les contraintes OCL. Ce que nous appelons aspects interprétations, c'est la définition en Kermeta du corps des opérations définies dans le méta-modèle UML4SPM. A la différence de l'approche *Visiteur*, ici aucune phase de parcours et d'instanciation des classes d'exécution n'est nécessaire. Le comportement est directement tissé dans les objets Kermeta instances du modèle de procédé. Il suffira alors juste d'appeler la méthode « execute() » sur l'activité initiale du procédé.

# 4 Discussion

Lors de nos expériences sur les différentes approches sur l'exécution des modèles de procédés UML4SPM, nous avons retenu certains points que nous reportons ci-dessous.

En ce qui concerne l'approche UML4SPM vers WS-BPEL (i.e., dite compilée), elle offre deux avantages qui ne sont pas négligeables. Le premier est la nette séparation entre la description de haut-niveau du procédé en utilisant UML4SPM (notation graphique) et son exécution en utilisant BPEL (dialecte XML). Ainsi la complexité du code BPEL est masquée par la phase de transformation. Le second avantage quant à lui se traduit par la possibilité dorénavant de réutiliser des outils performants dans le domaine des procédés métiers. Ces derniers offrent de nombreuses fonctionnalités avancées telles que le monitoring, la gestion et le versionning des artefacts, la régulation automatique des flux de données, etc.

Cependant plusieurs aspects pénalisent cette approche. Le premier concerne l'absence de tout support de l'aspect interaction humaine par BPEL. En effet, ce standard étant destiné à

l'automatisation des échanges de services entre plusieurs organisations, aucun concept n'est dédié à cet effet. Bien sûr, il existe des solutions à ce problème mais elles restent propriétaires aux outilleurs et diffèrent d'un outil à un autre. Un exemple d'initiative sur la dimension humaine dans BPEL est la proposition BPEL4People qui suggère d'implémenter les interactions humaines comme étant un service web comme un autre [Kloppmann (2005)]. Cependant jusqu'à présent, il n'existe pas d'interface standard pour ce type de service. Le second point est lié au fait qu'une fois le code BPEL généré, une étape intermédiaire est requise avant de pouvoir l'exécuter. Elle consiste à configurer les différents services (*Partner Link*) intervenant dans l'exécution du procédé. Ceci nécessite donc de manipuler du code BPEL qui est d'assez bas niveau (i.e., dialecte d'XML). De plus, si le code BPEL est amené à être modifié (par ex. ajout d'une variable) sans que cette modification soit manuellement reportée dans le modèle UML4SPM, cela conduirait à avoir deux définitions du procédé qui ne seront pas conformes. Cette approche conduit aussi à l'impossibilité de modifier les modèles de procédés UML4SPM pendant l'exécution vu qu'une étape de génération vers BPEL est requise après chaque modification. Cet obstacle n'est pas lié à BPEL mais à l'approche elle-même (i.e., compilée). La variabilité et l'évolution étant des caractéristiques fortes des procédés de développement logiciel, ce dernier point est incontestablement un frein pour cette approche.

L'approche interprétée vient justement pour répondre à ce problème d'évolution des modèles de procédés pendant l'exécution. Ceci est rendu possible grâce au couplage entre données et traitement offert par le modèle d'exécution d'UML4SPM. Pendant l'exécution, si le modèle de procédé UML4SPM était amené à changer, les classes d'exécution seront toujours à jour et pourront au besoin, récupérer les données du modèle de procédé chargé en mémoire. Les modèles UML4SPM seront directement exécutés et simulés sans passer par des étapes de raffinement ou de compilation. De plus, l'architecture de l'outil que nous proposons est assez flexible et permet l'extension du langage (ajout de nouvelles actions, concepts, propriétés) à moindre coût.

Pour le modèle d'exécution, nous avons proposé deux implantations, une en Java et l'autre en Kermeta. Notre choix s'est porté en premier lieu sur Java pour des raisons de performances. Cependant, un point qui pourrait être perçu comme un inconvénient dans cette solution est le fait que notre implantation de la sémantique d'exécution des éléments d'activités et d'actions UML2.0 ne tient pas compte des points de variations dans la sémantique liée à certains éléments. Cette implantation représente donc nos propres choix ce qui pourrait être en désaccord avec d'autres choix de conception propres aux exigences d'un projet particulier. La solution Kermeta vient résoudre ce problème en permettant de tisser des aspects selon la sémantique choisie. On pourra donc composer à volonté différents aspects qui seront chargés avant l'exécution du modèle de procédé et prendront en compte les choix de l'utilisateur (aspect interprétation, aspects liés aux points de variations, tissage des contraintes OCL, etc.).

Un autre intérêt de cette utilisation de Kermeta pour décrire la sémantique opérationnelle d'UML4SPM est qu'il est maintenant beaucoup plus facile d'envisager la description de la sémantique opérationnelle de reconfiguration dynamique du procédé. En effet, il suffit d'introduire un nouveau type de *Software Activity* pour lequel la méthode « execute() » n'a plus un effet sur le maniement des jetons liés à la sémantique des activity diagrams d'UML, mais bien sur la manière dont les autres instances de *Software Activity* (et des autres classes du méta-modèle UML4SPM) sont interconnectées. La solution Kermeta représente donc, un compromis idéal entre flexibilité et exécutabilité, le tout selon une démarche complètement orientée modèles.

# 5 Conclusion

L'objectif de nos travaux est de promouvoir l'expressivité, la compréhensibilité, la flexibilité et l'exécutabilité comme critères essentiels aux langages de modélisation de procédés. Ces critères répondent non seulement aux besoins des acteurs du procédé, si nombreux et avec des cultures si différentes mais aussi aux attentes du marché du logiciel, toujours plus exigeant en termes de fiabilité et de rapidité de production. Dans cet article, nous avons présenté UML4SPM, un langage exécutable et orienté modèle pour la modélisation des procédés logiciels. Pour la définition de notre langage, nous avons adopté une démarche normative en réutilisant le standard UML2.0 comme brique de base de notre langage. Pour l'exécution des modèles UML4SPM, deux approches ont été explorées. Une approche compilée visant à projeter les modèles UML4SPM vers BPEL et une autre interprétée en définissant un modèle d'exécution. Le modèle d'exécution d'UML4SPM a été implanté en premier lieu en Java puis en Kermeta.

Dans la littérature, d'autres travaux proposant des LMPs basés sur UML ont été déjà proposés. Les plus connus sont les travaux de Di Nitto [Di Nitto (2002)], de Chou [Chou (2002)] ainsi que l'approche Promenade [Franch (1998)]. Dans la première proposition, l'exécution du procédé est dérivée par génération de code à partir de diagrammes d'états transitions et de diagrammes d'activités. C'est donc une approche compilée et qui reprend les mêmes inconvénients que nous avons détaillés dans la section 4. De plus, du code est rajouté manuellement afin de lier les différents bouts de codes générés à partir des différents diagrammes. Dans [Chou (2002)], les auteurs proposent de décrire le modèle de procédé avec UML et de réécrire à la main le code exécutable lui correspondant dans un autre langage propriétaire. Aucune génération automatique n'est proposée. Quant à Promenade, aucune approche d'exécution n'a été proposée. Les auteurs se limitent à définir des diagrammes UML pour la documentation de procédés. Finalement, le standard SPEM de l'OMG, qui même dans sa dernière version [OMG (2007c)], ne propose aucune solution pour l'exécution des modèles de procédés logiciels.

UML4SPM est actuellement en cours d'évaluation dans le contexte du projet Modelplex dont ce travail fait partie. A long terme, notre intérêt porte particulièrement sur l'identification des conditions sous lesquelles les modèles de procédés UML4SPM peuvent être modifiés sans interrompre l'exécution du procédé. Cette particularité est en cours d'étude dans le contexte du framework *Kermeta*. Une autre perspective importante est quant à elle liée à l'apparition de nouvelles méthodes de travail telles que la sous-traitance et l'offshore. Cela implique une coopération entre plusieurs équipes souvent à distance l'une de l'autre et donc un besoin de répartir des bouts de modèles de procédés sur différents sites. Ceci entraine une préoccupation majeure: la coordination des différentes exécutions de ces modèles de procédés.

# Références

Bandinelli et al (1993). Software process model evolution in the SPADE environment. *IEEE Transaction in Software Engeneering.1128–1144*.

Bendraou R., Gervais M.P. and Blanc X. (2005). UML4SPM: A UML2.0-Based Metamodel for Software Process Modeling. *In Proceedings of the ACM/IEEE 8th Interna-*

*tional Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, LNCS, Vol. 3713, PP 17-38.*

Bendraou R., Gervais M.P. and Blanc X (2006). UML4SPM: An Executable Software Process Modelling Language Providing High-Level Abstractions. *In Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, pp. 297-306, Hong Kong, China.*

Bendraou R., Sadovykh A., Gervais M.P. and Blanc X (2007a). Software Process Modeling and Execution: The UML4SPM to WS-BPEL Approach. *In Proceedings of the 33rd EUROMICRO Conference of Software Engineering Advanced Application (SEAA), pp. 314-321, Lübeck, Germany, IEEE Computer Society Press.*

Bendraou R (2007b). Thèse de Doctorat de l'université de Pierre et Marie Curie (document en anglais): "UML4SPM: Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle".

Bendraou R., Combemale B., Crégut X. and Gervais M.P (2007c). Definition of an Executable SPEM2.0. *In Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society press, Nagoya, Japan.*

Curtis W., Kellner M. I. and Over J. (1992). Process Modelling. *Communication of ACM, 35 (9), pp. 75-90.*

Chou S.-C. (2002). A process modeling language consisting of high level UML diagrams and low level process language. *Journal of Object Technology 1, pp. 137–163*

Di Nitto E. et at. (2002). Deriving executable process descriptions from UML. *In Proceedings. of the 24th International Conference. on Software Engineering (ICSE'02), Florida,USA, ACM Press.*

Franch, X.; Ribó, J.M. (1998) A Structured Approach to Software Process Modelling. *In Proceedings 24th EUROMICRO Conference, IEEE Computer Society Press, Los Alamitos Washington Brussels Tokyo, pp. 753-762.*

Gamma E., Helm R., Johnson R., and Vlissides J. (1994). *Design Patterns: Elements of Reusable Object- Oriented Software*. Addison-Wesley.

Hausmann J.H., Störrle H. (2005). Towards a Formal Semantics of UML 2.0 Activities. *In Proceedings of the German Software Engineering Conference.*

Jaccheri M.L., Baldi M., Divitini M. (1999). Evaluating the Requirements for Software Process Modelling Languages and Systems. *In Proceedings of Process support for Distributed Team-based Software Development, Florida, USA.*

Junkermann, et al. (1994). MERLIN: Supporting cooperation in software development through a knowledge-based environment. *In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, Software Process Modelling and Technology, pp. 103 - 129. John Wiley & Sons*

Kaiser G.E., Barghouti N.S. and Sokolsky M.H. (1990). Preliminary experience with process modeling in the Marvel software development environment kernel. *In Proceedings of the 23d Annual Hawaii International. Conference on System Sciences, Vol.H Software Track. IEEE Computer Society, pp. 131-140, Washington, DC, USA.*

Kellner, M.I et al. (1991). ISPW-6 software process example. *In Proceedings of the first International. Conference on the Software Process. IEEE Computer Society, pp. 176-186, Washington, DC.*

Kloppmann, M. et al. (2005) "WS-BPEL Extension for People BPEL4People",

Modelplex, IST European Project (2007). Contract IST-3408.

Montangero C., Derniame J.C., and Kaba B.A., Warboys B. (1999). *The software process: Modelling and technology*. LNCS GmbH. Vol. 1500.

Brice M. et al. (2007). Towards a generic aspect-oriented modeling framework. In *Models and Aspects workshop, at ECOOP*.

Muller P.A, Fleurey F., and Jézéquel J.M (2005). Weaving executability into object-oriented meta-languages, *Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pages 264--278, Montego Bay, Jamaica, Springer*.

Sutton, Jr., S.M., Heimbigner D., and Osterweil L. J. (1995). APPL/A: A language for software-process programming. *ACM Transaction on Software Engineering and Methodology, 4(3):221–286.*

OMG SPEM1.0 (2002). *Software Process Engineering Metamodel*. OMG document formal/02-11/14, November 2002, at http://www.omg.org.

OMG xUML (2005), *Semantics of a Foundational Subset for Executable UML Models RFP*, OMG document ad/05-04-02 at: http://www.omg.org/docs/ad/05-04-02.pdf.

OMG UML (2007a). *Unified Modeling Language Infrastructure Specification, version 2.1.1*. OMG document formal/07-02-06 at http://www.omg.org

OMG UML (2007b). *Unified Modeling Language", Superstructure Specification, version 2.1.1*. OMG document formal/07-02-04 at http://www.omg.org

OMG SPEM2.0 (2007). *Software Process Engineering Metamodel*. OMG document, final adopted specification, ptc/07-03-03 at http://www.omg.org.

WS-BPEL (2007). *Web Services Business Process Execution Language Version 2.0*. Working Draft. WS-BPEL TC OASIS.

## Summary

One of the main objectives of the Model-Driven Engineering vision is to increase software productivity through the extensive use of models since earliest software development phases. The challenge targeted by this initiative is to use models not only for documentation purposes but also for production aims. In the area of software process modeling, software process modeling languages have not yet reached the level required for the specification of executable models. Executable software process models can help in improving coordination between development teams, in automating iterative and no-interactive tasks and in managing the different tools and artifacts used during the software construction. At this aim, we propose UML4SPM, a model-driven and executable language for software process modeling.