Proposition d'une démarche de type IDM pour la construction d'outils d'exécution de processus

Sana Mallouli[1] — Saïd Assar [2,1]— Carine Souveyet[1]

[1] Université Paris 1 Panthéon Sorbonne, Centre de Recherche en Informatique,
90, Rue de Tolbiac F-75013 Paris
sana.mallouli@malix.univ-paris1.fr
carine.souveyet@univ-paris1.fr
[2] Institut Mines-Télécom, École de Management, Dépt. Systèmes d'Information,
9, Rue Charles Fourrier F-91011 Evry
said.assar@telecom-em.eu

RÉSUMÉ. *L'ingénierie des systèmes d'information fait appel à de multiples langages pour modéliser, programmer et manipuler divers artéfacts tout le long du cycle de développement. Ces langages sont généralement supportés par des outils logiciels. La construction de ces outils est une tâche complexe menée souvent de manière ad-hoc. L'exploitation adéquate des techniques de méta-modélisation a le potentiel faciliter cette tâche. Cependant, pour un langage de modélisation de processus, la prise en compte de sa sémantique d'exécution dans un méta-modèle n'est pas encore au point. Dans cet article, nous présentons une démarche basée sur l'usage de méta-modèles événementiels et de règles de transformation pour décrire la sémantique d'exécution d'un modèle de processus et en dériver une architecture d'outil d'exécution. Cette démarche est expérimentée avec un langage de modélisation orienté intention et le résultat est comparé avec des travaux antérieurs.*

ABSTRACT. *Information System engineering involves multiple languages for modeling, programming and handling various artifacts throughout the development cycle. These languages are usually supported by software tools. The construction of these tools is a complex task often done in an ad-hoc manner. Proper use of meta-modeling techniques has the potential to facilitate this task. However, for process modeling language, properly expressing and exploiting process execution semantics is still a challenging issue. In this paper, we present an approach in which process execution semantics are expressed using an event meta-model, and a set of transformation rules are defined and applied in order to derive the technical architecture of an execution tool. This approach is tested on the case of an intention-oriented modeling language, and the result is compared with previous work.*

MOTS-CLÉS : Ingénierie des langages logiciels, méta-modélisation, exécutabilité d'un modèle, sémantique d'un modèle de processus, exécution de processus.

KEYWORDS: Software languages engineering, meta-modeling, model executability, process model semantics, process execution.

1. Introduction

Dans l'essor du génie logiciel et de l'ingénierie des systèmes d'information, les langages jouent un rôle prépondérant. Que ce soit pour programmer, modéliser, transformer ou interroger, ces langages sont essentiels pour construire, manipuler et raisonner sur des artéfacts tout au long du cycle de vie des systèmes. Ceci nécessite la construction d'outils logiciels divers et variés pour supporter ces langages et rendre concrètement possible leur utilisation. Ce besoin s'est accentué avec le développement de l'ingénierie dirigée par les modèles (IDM) qui, justement, prône l'usage intensif de modèles tout le long du cycle de conception et de développement des systèmes (Favre *et al.*, 2006). L'édition et la vérification d'un modèle, la transformation d'un modèle vers un autre et la génération du code final deviennent ainsi des tâches essentielles qui doivent être spécifiées, formalisées et mises en œuvre dans des outils logiciels qu'il faut construire (Jouault *et al.*, 2009).

L'objectif de l'ingénierie des langages est de spécifier les langages informatiques et de construire des outils logiciels de support (Favre *et al.*, 2009). Pour spécifier un langage de modélisation, on fait appel à la méta-modélisation (Atkinson et Kühne, 2003)(Sprinkle *et al.*, 2011). Un méta-modèle est un modèle d'un langage de modélisation. C'est une représentation, généralement graphique (mais pas forcément), des concepts sous-jacents au langage, des liens entre ces concepts ainsi que d'éventuelles contraintes que doivent satisfaire ses instances. Dans la terminologie des langages de programmation, un méta-modèle est l'équivalent de la syntaxe abstraite d'un langage (Kleppe, 2009a). La sémantique d'un langage concerne le sens que peuvent prendre les constructions de celui-ci lorsqu'elles sont instanciées dans un modèle (Harel et Rumpe, 2004). Elle n'est que partiellement prise en compte par un méta-modèle (Sprinkle *et al.*, 2011). Dans l'univers des langages de programmation, le problème de l'expression de la sémantique s'y pose dans des termes similaires. En effet, les méta-compilateurs exploitent des langages basés sur la notation Backus-Naur Form (BNF) pour spécifier la syntaxe et s'en servir pour dériver des outils (Kleppe, 2009b). L'aspect sémantique est relégué à un traitement par programmation directe, même si la compilation a introduit des approches innovantes pour organiser cette tâche, telle que la grammaire des attributs (Paakki, 1995). Et depuis peu, une communauté de chercheurs appelle explicitement à rapprocher l'univers de l'IDM et celui de la compilation (Jézéquel *et al.*2012).

L'expression explicite, éventuellement formelle, de la sémantique d'un langage est une problématique importante dans l'ingénierie des langages. Elle répondrait à plusieurs besoins tels qu'une spécification plus rigoureuse d'un langage, l'analyse et la validation de cette spécification, ou encore la génération automatique d'outils support pour le langage (Bryant *et al.*, 2011). Lorsque cette sémantique se limite à des contraintes de validité que doivent respecter les instances d'un modèle, elle est appelée sémantique statique. Elle peut s'exprimer avec un langage de règles au niveau du méta-modèle, tel le langage OCL (Object Constraint Language). Mais

pour un langage de modélisation de processus, avec des concepts tels que flot de données, état et transition, la sémantique désigne alors un comportement dynamique qui correspond à la manière avec laquelle un processus s'exécute. Cette sémantique d'exécution est difficile à capturer dans un méta-modèle, elle est néanmoins indispensable pour construire des outils pour exécuter un modèle de processus.

Dans cet article, nous développons une démarche d'ingénierie des langages pour construire un outil d'exécution de modèles de processus. A l'aide d'une notation événementielle, nous complétons la spécification statique avec un schéma dynamique qui capture la sémantique d'exécution du modèle. Grâce à des règles de transformation, ce schéma est converti en une architecture logicielle d'un outil d'exécution dans un environnement cible. Cette architecture exploite des motifs génériques d'exécution (des « patterns ») de type publier/souscrire (Eugster *et al.*, 2003). Ces motifs architecturaux sont adaptés pour la programmation de systèmes asynchrones faiblement couplés (Hinze *et al.*, 2009), et permettent de transcrire correctement la sémantique d'exécution événementielle d'un modèle de processus.

Le reste de cet article est structuré comme suit. La section 2 est un bref état de l'art concernant l'expression de la sémantique, et les méta-outils d'ingénierie des langages. La section 3 présente notre méta-démarche et introduit la modélisation événementielle de sémantique d'exécution. Les règles de transformation sont ensuite présentées dans la section 4. Dans les sections 5 et 6, cette démarche est appliquée à un langage de modélisation intentionnelle des processus et en guise d'évaluation, le résultat obtenu est comparé avec des travaux antérieurs ayant le même objectif. L'article se termine par une conclusion qui évoque les limites et les travaux futurs.

2. État de l'art

La question de la sémantique d'un langage se retrouve dans deux courants de recherche complémentaires. Le premier est directement lié aux méthodes et techniques de spécification des langages informatiques. Ce domaine est pratiquement aussi ancien que l'informatique elle-même et se confond avec la création des premiers langages de programmation et des premiers compilateurs. Ce domaine a pris un essor important avec le développement des langages et modèles spécifiques aux domaines, connus sous les sigles DSL et DSM (Sprinkle *et al.*, 2009). Le second courant est celui des outils et environnements logiciels de méta-modélisation. Certains de ces outils ont pour objectif explicite la définition de nouveaux langages, notamment des DSL et DSM, tel que MetaEdit+ par exemple (Kelly et Tolvanen, 2008) ; d'autres font partie d'ateliers sophistiqués de génie logiciel tel que TOPCASED (Farail, 2012). Dans cette section, nous faisons une brève synthèse de ces deux courants de recherche en s'inspirant des présentations faites dans (Gargantini *et al*, 2009) et (Bryant *et al.*, 2011). Notre objectif est de faire le point sur la manière avec laquelle la spécification de la sémantique est faite, est-elle déclarative, est-il possible de la représenter graphiquement ; et si éventuellement, elle permet la génération automatique d'un outil d'exécution.

La technique la plus connue pour définir la sémantique d'un langage de programmation est la grammaire des attributs introduite par D. Knuth en 1968. A chaque nœud de la syntaxe abstraite d'un langage est associé un (ou plusieurs) attribut(s), ainsi que des fonctions pour calculer et propager la valeur de ces attributs à partir des attributs des nœuds adjacents. Cette technique a été largement appliquée en compilation pour construire des analyseurs sémantiques et des générateurs de code. Le concepteur décrit ainsi sous forme d'opérations et de calculs la sémantique des constructions du langage. Cette vision *opérationnelle* et *impérative* de la sémantique se distingue d'autres approches déclaratives et plus formelles, telles que les sémantiques axiomatique et dénotationnelle (Winskel, 1993).

Généralement associée avec le développement des langages à syntaxe textuelle, la grammaire des attributs a été adaptée à la spécification des langages de modélisation. Le prototype de recherche JastEMF (Bürger *et al.*, 2011) combine l'environnement de méta-modélisation EMF d'Eclipse avec le méta-compilateur expérimental JastAdd basé sur la grammaire des attributs (Hedin, 2011). JastEMF est de ce fait un environnement de méta-modélisation qui intègre la puissance de la grammaire des attributs. L'expression de la sémantique est en partie déclarative (fonctions rattachées aux noeuds) et en partie impérative (des calculs dans les fonctions). Cependant, elle est textuelle et n'a pas de représentation graphique.

Dans l'univers de l'IDM, le langage Kermeta est une des approches les plus innovantes. Dans (Jézéquel *et al.*, 2011), les auteurs définissent Kermeta comme un méta-atelier pour *l'Ingénierie des Langages Dirigée par les Modèles*. C'est une approche qui se situe dans l'univers du méta-méta-modèle MOF (Meta Object Facility). Kermeta utilise Ecore, une variante du MOF intégrée avec l'environnement Eclipse, pour définir la syntaxe abstraite d'un langage sous forme d'un méta-modèle statique. La sémantique statique (contraintes sur le méta-modèle) s'exprime avec le langage OCL, et la sémantique d'exécution avec des méthodes directement rattachées aux méta-classes à l'aide d'un langage de programmation spécifique orienté aspects inspiré de Java. A partir de cette méta-spécification, Kermeta génère un éditeur, un vérificateur et un exécuteur pour le langage en cours de construction. Cette spécification de la sémantique opérationnelle n'est pas déclarative et ne possède pas de représentation graphique, aucune conceptualisation n'étant disponible pour décrire les séquences d'instructions.

Un autre projet très abouti dans le monde de l'IDM est TOPCASED. TOPCASED est un atelier de génie logiciel destiné au développement de systèmes embarqués (Farail, 2012). C'est une solution basée sur la plateforme logicielle Eclipse et utilise les langages UML et SysML. TOPCASED est aussi un méta-environnement pour la définition de nouveaux langages (Crégut *et al.*, 2010). Comme dans Kermeta, la syntaxe abstraite s'exprime avec Ecore et elle est complétée par une syntaxe concrète définie avec un éditeur graphique. La sémantique du méta-modèle s'exprime à l'aide de formalismes à base d'état et d'événements. Cette spécification permet la simulation d'exécution à l'aide de trois éléments : *l'Agenda*, *le Contrôleur* (ou Driver), et *l'Interpréteur*. Elle est en partie déclarative, graphique et formelle

grâce aux méta-modèles d'états et d'événements. Deux éléments du moteur d'animation sont génériques, alors que l'interpréteur est réécrit pour chaque langage et cette écriture n'est ni déclarative ni graphique. Dans (Crégut *et al.*, 2010), les auteurs proposent l'identification de motifs génériques (*patterns*) pour y remédier.

Enfin, dans le monde DSL et DSM, un des ateliers les plus connus est MetaEdit+ (Kelly et Tolvanen, 2008)(MetaCASE, 2012). C'est un outil de méta-modélisation, de génération d'ateliers de génie logiciel et d'ingénierie des méthodes qui est régulièrement cité et évalué (Niknafs et Ramsin, 2008)(El Kouhen *et al.*, 2012). La spécification de la sémantique se fait au niveau des scripts de génération de code, elle n'est ni déclarative ni graphique. Et nous avons pu constater dans un projet exploratoire que nous avons mené (Mallouli et Assar, 2013), que les fonctionnalités de méta-modélisation sont effectivement puissantes et faciles à utiliser. Cependant, la programmation du générateur de code est rendue très complexe par l'absence de représentation graphique de l'architecture du générateur.

Ce bref tour d'horizon de quelques travaux relatifs à l'ingénierie des langages nous permet de dresser le constat suivant : la méta-modélisation est une technique qui est actuellement supportée par des outils sophistiqués et fiables, cependant, lorsqu'il s'agit de spécifier un langage de modélisation ayant une sémantique exécutable, l'intégration de la spécification de la sémantique dans le méta-modèle reste difficile. Plusieurs approches sont explorées dans la littérature, elles restent pour le moment difficiles à mettre en œuvre. Pour notre part, nous considérons qu'il est important que cette spécification de la sémantique puisse être représentée graphiquement et qu'elle soit exprimée dans une logique déclarative et non pas uniquement opérationnelle. C'est l'objectif de la démarche que nous présentons ici.

3. Méta-modélisation événementielle

L'étude de l'état de l'art illustre la diversité des approches pour spécifier le plus précisément possible la sémantique d'un langage, et surtout, pour exploiter cette spécification dans la construction des outils nécessaires à l'utilisation concrète du langage. Pour élaborer notre approche, nous avons cherché une spécification de la sémantique d'un langage qui satisfait plusieurs contraintes : (i) Elle doit être de nature **déclarative** pour faciliter son élaboration; (ii) elle doit avoir une représentation **graphique** qui facilite sa lecture et sa compréhension; (iii) elle doit posséder une **sémantique claire** pour éviter les ambiguïtés d'interprétation ; et enfin, (iv) cette spécification doit être suffisamment **riche** pour qu'il soit possible d'en déduire, par génération de code, un outil d'exécution de modèles.

Nous avons choisi d'élaborer notre approche selon une logique d'IDM. En effet, notre objectif étant de construire des outils logiciels (pour exécuter des modèles de processus), il nous parait naturel de favoriser l'usage de modèles et d'exploiter ainsi la démarche IDM pour d'une part, offrir au concepteur de langage des spécifications de niveau conceptuel et d'autre part, obtenir des spécifications de niveau physique pour réaliser l'outil logiciel. Le synopsis de cette approche est présenté à la figure 1.

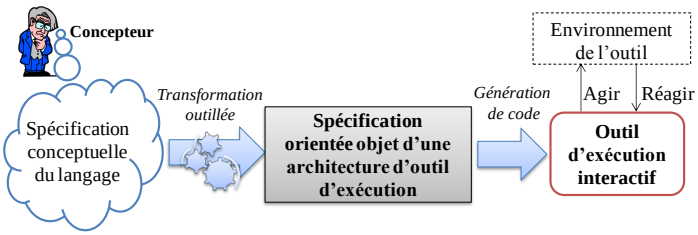

Figure 1. *Synopsis de notre approche*

La spécification conceptuelle du langage comporte deux parties selon deux visions spécifiques et complémentaires : un diagramme de classes UML pour une vision structurelle du méta-modèle, et un schéma événementiel pour une vision dynamique de la sémantique d'exécution du langage. Le paradigme événementiel est très utilisé pour décrire les systèmes réactifs et asynchrones, et nous semble très approprié pour capturer la vue systémique d'un outil d'exécution. Il n'est cependant pas présenté comme un concept central des langages tels qu'UML ou réseaux de Pétri. Ceci nous a amené à choisir le formalisme dynamique de la méthode Remora (Rolland *et al*, 1988) pour sa concision (peu de concepts), son expressivité pour la vue systémique (événement et échange de messages), sa sémantique claire et bien définie ainsi que les possibilités d'implémentation.

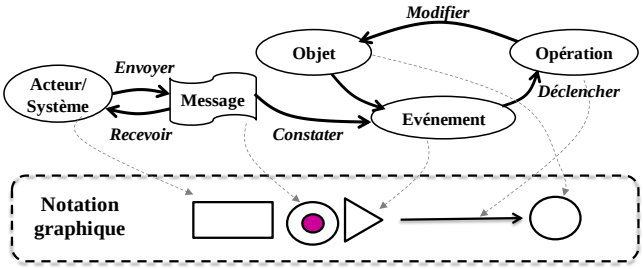

Figure 2. *Concepts et notation graphique du formalisme de modélisation Remora*

La figure 2 résume les concepts de Remora ainsi que la notation graphique. On note la présence du concept « acteur » qui peut être un agent humain ou un système (ou application) externe. Ce concept sert à décrire les entités qui se trouvent dans l'environnement du système et avec lesquels il échange des messages (entrants et sortants). L'envoi et la réception de ces messages constituent des événements externes qui agissent sur la dynamique du système en déclenchant des opérations.

La figure 3 décrit en détail notre approche IDM pour la spécification d'un langage de modélisation. Dans cette approche, la spécification de la sémantique d'exécution – capturée par le schéma événementiel – correspond en fait à la logique opérationnelle d'exécution d'un outil logiciel qui interprèterait les instances du modèle. C'est une vision opérationnelle de la sémantique, elle revient à spécifier le fonctionnement d'un interpréteur abstrait du modèle.

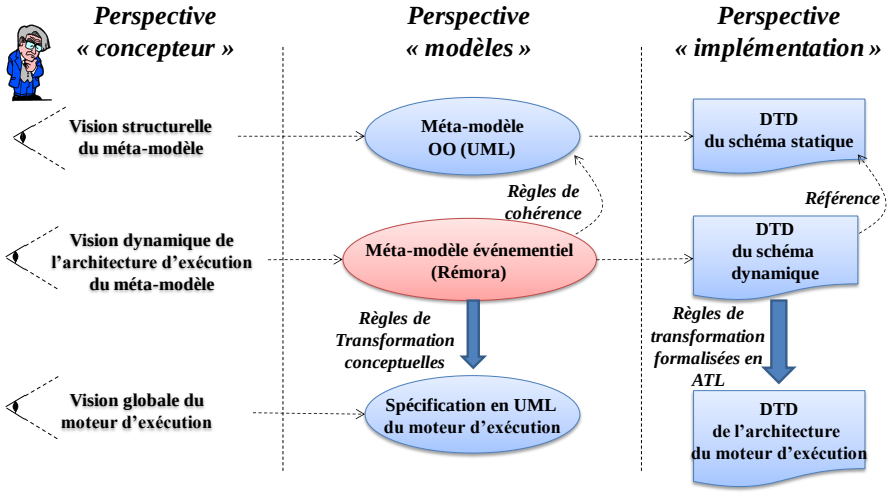

Figure 3. *Vue détaillée de notre approche selon trois perspectives*

Dans la première étape, le concepteur définit le méta-modèle structurel (i.e. la syntaxe abstraite) du langage sous forme d'un diagramme de classes UML. A l'étape suivante, le concepteur traduit la sémantique du langage à l'aide d'un schéma événementiel Remora. Comme ce schéma se confond avec la logique opérationnelle de fonctionnement d'un interpréteur, il faut au préalable rajouter aux méta-classes du diagramme UML un ensemble de classes pour représenter **les instances des concepts** du modèle. Ainsi, à chaque méta-classe est définie par extension une classe d'instances. Cet ensemble de classes et de méta-classes, organisé en deux niveaux d'abstractions, contient les objets sur lesquels va porter le schéma événementiel. Dans la perspective « implémentation », ces diagrammes de classes UML seront décrits dans des documents XML (fichiers avec extension .DTD).

4. Dérivation de l'architecture d'un outil d'exécution

La troisième étape de la démarche consiste à générer l'architecture technique de l'outil d'exécution de modèles. Il s'agit de traduire la sémantique d'exécution du modèle capturée dans le schéma événementiel en une spécification logicielle complète. Pour traduire la logique événementielle d'un schéma Remora, nous faisons appel à un cadre (ou « framework ») d'exécution connu, celui des motifs génériques « publier/souscrire » (Eugster *et al.*, 2003). Le résultat de cette transformation est une architecture logicielle décrite dans un diagramme de classes UML complet qui englobe les méta-classes, les classes d'extension ainsi que les classes qui implémentent la logique opérationnelle de l'outil d'exécution.

4.1 Le framework d'exécution asynchrone publier/souscrire

Les motifs génériques « publier/souscrire » sont issus de l'univers de la programmation distribuée, et sont considérés comme le paradigme de choix pour le développement d'applications asynchrones et réactives (Hinze *et al.*, 2009). Le principe fondamental est le suivant: un objet (appelé « objet dynamique ») qui doit réagir à l'occurrence d'un événement constaté sur un objet « observé » va souscrire à cet événement. A chaque occurrence de celui-ci, le système se chargera de le propager aux objets abonnés en les notifiant de l'occurrence de l'événement et en leur communiquant les données contextuelles associées. L'objectif est de permettre à un ou plusieurs objets de réagir aux messages d'autres objets, sans qu'ils ne soient connus à l'avance, sans devoir les lier « en dur » dans le code. Pour la transformation d'un schéma Remora, nous faisons appel à trois motifs d'exécution. Le premier correspond à une interaction interne entre objets appartenant à un même système. C'est une implémentation directe du paradigme « publier/souscrire », elle est schématisée dans la figure 4.

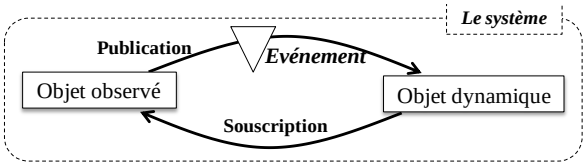

Figure 4. *Motif pour la gestion d'une interaction interne*

Le second motif correspond à une interaction externe, où un message entant venant de l'extérieur du système (une application externe) est attendu par des objets dynamiques. Ce motif est représenté par une interaction entre l'objet dynamique et une file d'attente dans laquelle sont déposés les messages de l'objet observé, et à laquelle va souscrire l'objet dynamique (figure 5).

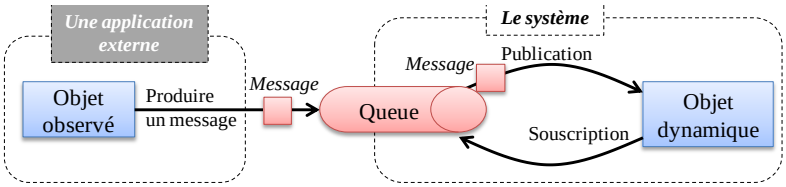

Figure 5. *Motif pour la gestion d'une interaction externe avec message entrant*

Le troisième motif correspond aussi à une interaction externe dans laquelle le message est produit par un objet dynamique du système pour être consommé par une application externe particulière. C'est le cas par exemple d'une invocation d'une application externe pour exécuter une tâche du processus, ou le cas d'une notification d'un état du système à destination d'un acteur externe. C'est une interaction de type « point à point » où le message produit par l'objet observé est directement consommé – après passage dans une file d'attente de type FIFO – par un seul objet consommateur (figure 6).

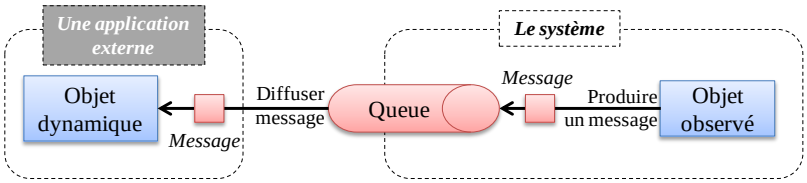

Figure 6. *Motif pour la gestion d'une interaction externe avec message sortant*

4.2 Les règles de transformation

Nous allons maintenant exploiter ces trois motifs d'exécution pour transformer le schéma dynamique Remora en une architecture logicielle. Une première analyse des concepts du formalisme source (figure 2) et celles du formalisme cible (diagramme de classes UML) permet d'identifier plusieurs transformations directes :

• Les éléments *Objet* n'ont pas à être transformés, elles correspondent aux classes-instances et méta-classes du diagramme UML.

• Lorsqu'elle modifie un objet, une *Opération* se transforme en une méthode ; sinon, elle est traitée par une règle

• Un lien *Modifie* entre un *Objet* et *Opération* se transforme en un lien d'appartenance de la méthode à la classe correspondant à l'*objet.*

• Un lien *Déclenche* entre *Événement* et *Objet* est transformé en une méthode d'une classe particulière appelée *Listener*. Cette méthode est invoquée après la constatation d'un événement (interne ou externe) qui lui est donné en paramètre. Le type du *Listener* dépend de l'événement (s'il est interne ou externe).

Au-delà de ces règles directes, nous introduisons trois groupes de règles pour la transformation des événements internes, des événements externes et des opérations qui communiquent avec les acteurs externes.

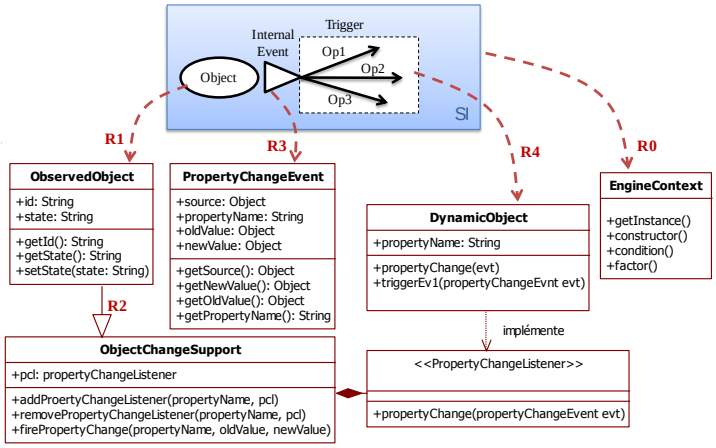

Figure 7. *Transformation d'un événement interne*

Le premier groupe des règles traduit un événement interne en une structure d'interaction interne de type « publier/souscrire » (cf. fig. 4). Ce qui correspond à cinq règles illustrées à la figure 7 :

R0. Une classe *EngineContext* est créée pour gérer les données génériques d'exécution des processus (niveau des concepts) et des données spécifiques du processus en cours d'exécution (niveau des instances).

R1. A la classe qui correspond à l'objet sur lequel est constaté l'événement, on crée une classe *ObservedObject* avec une propriété d'état et des méthodes prédéfinies *getState()* et *setState()* pour y accéder.

R2. La classe *ObservedObject* hérite de la classe prédéfinie *ObjectChangeSupport* afin d'avoir des capacités de gestion d'objets dynamiques qui s'abonnent à des événements. Cette super classe permet de créer ou de supprimer de nouveaux *Listener* correspondant à de nouveaux abonnements d'objets dynamiques. La méthode *firePropertyChange()* permet de notifier le changement d'état d'une propriété (événement), de générer une instance de l'objet *PropertyChangeEvent* et de diffuser cet événement aux abonnés à cette propriété en invoquant la méthode *propertyChange()*. Pour que ce mécanisme soit générique, les objets abonnés doivent être issus d'une classe qui implémente l'interface *PropertyChangeListener*.

R3. La classe prédéfinie *PropertyChangeEvent* représente de manière générique tous les événements internes du schéma dynamique. Un événement de type *PropertyChangeEvent* est défini par un nom de propriété (*propertyName*), l'ancienne et la nouvelle valeur (*oldValue, newValue*) et une référence de l'objet sur lequel l'événement est constaté (*source*).

R4. Associer au groupe d'opérations déclenchées par un événement interne une classe *DynamicObject* (observateur de l'événement). Cette classe implémente l'interface *<<PropertyChangeListener>>* et contient ainsi la méthode *propertyChange()* qui prend comme paramètre l'événement déclenchant. Dans cette méthode s'effectue l'acheminement des opérations du déclencheur en appliquant des tests sur la structure de l'événement qui est passé en paramètre.

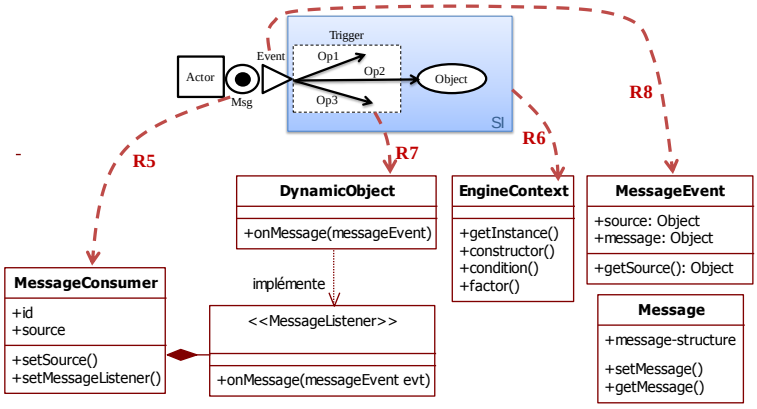

Figure 8. *Transformation d'un événement externe avec un message entrant*

Le second groupe de règles concerne les événements externes. En appliquant le motif d'interaction externe avec un message entrant (fig. 5), un acteur est considéré comme un producteur de messages. On souscrit alors les objets sur lesquels portent les opérations déclenchées par l'événement externe aux messages publiés par cet acteur. Ce qui se traduit par les règles illustrées à la fig.8.

R5. Pour l'événement externe, définir une file d'attente à travers la classe *MessageConsumer* qui comprend l'interface <<*MessageListener*>>. Cette interface souscrit à la file d'attente *MessageConsumer* et la méthode *onMessage()* est invoquée lorsqu'il y a un nouveau message à consommer dans la file d'attente. L'acteur lui-même apparaîtra dans la classe *MessageConsumer* via l'attribut *source*.

R6. Pour chaque groupe d'opérations déclenchées par un événement, pour chaque condition et chaque facteur, ajouter les méthodes correspondantes dans la classe globale *EngineContext*.

R7. Associer au groupe d'opérations déclenchées par un événement externe une classe *DynamicObject* qui implémente l'interface <<*MessageListener*>>. Cette classe souscrira à la file d'attente correspondante à l'événement externe. A l'arrivée d'un message, l'objet *MessageConsumer* génère une instance de *MessageEvent* avant d'invoquer la méthode *onMessage()* de la classe DynamicObject.

R8. Associer à un message une classe *MessageEvent*, ce message est consommé par la méthode *onMessage()* de l'objet dynamique.

Le 3ème groupe de règles transforme une opération qui invoque ou notifie un acteur externe en son équivalent en termes de concepts UML orientés objet. Ceci est réalisé avec le motif pour la gestion d'une interaction externe avec message sortant (cf. fig.6). Dans cette situation, l'objet est le producteur du message et l'acteur externe en est le consommateur. Ce qui se traduit par les règles illustrées à la fig.9.

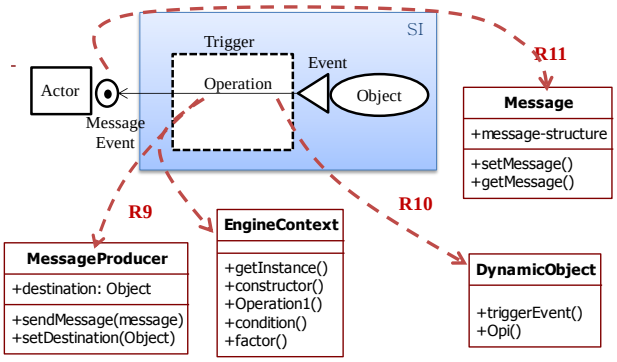

Figure 9. *Transformation d'une opération externe*

R9. Associer à l'opération externe une classe *MessageProducer*. Ajouter à cette classe une méthode *sendMessage()* pour diffuser le message. Référencer l'objet dynamique destinataire du message dans la classe *MessageProducer* avec l'attribut *destination*. La création d'une instance de *MessageProducer* est réalisée par la méthode *constructor*() de la classe globale *EngineContexte*.

R10. Pour l'opération déclenchée, ajouter l'opération externe *Opi* en tant que méthode dans l'objet *DynamicObject*.

R11. Le message envoyé par l'opération externe est définie dans une classe *Messsage* et il est géré grâce aux méthodes *getMessage()* et *setMessage().*

5. Illustration

Nous avons expérimenté notre approche sur le langage Map de modélisation intentionnel de processus (Rolland *et al.*, 1999). Ce langage est particulièrement adapté pour représenter des processus à haut niveau d'abstraction et à forte variabilité. La sémantique complexe d'exécution du Map correspond à une navigation dans un graphe selon les intentions de l'utilisateur et suivant le contexte situationnel relatif à l'état du produit. C'est un excellent exemple pour illustrer l'applicabilité de notre approche et son intérêt pour construire un outil d'exécution.

Brièvement, une carte Map est un ordonnancement non figé d'intentions reliées par des stratégies. Une stratégie est une manière de réaliser une intention. Elle peut être une action exécutée par une application externe et qui aboutit à des transformations du produit, ou être elle-même décrite récursivement par une sous-carte Map. Ce qui autorise une grande variabilité dans l'ordonnancement de réalisation des intentions et dans le choix des stratégies à appliquer.

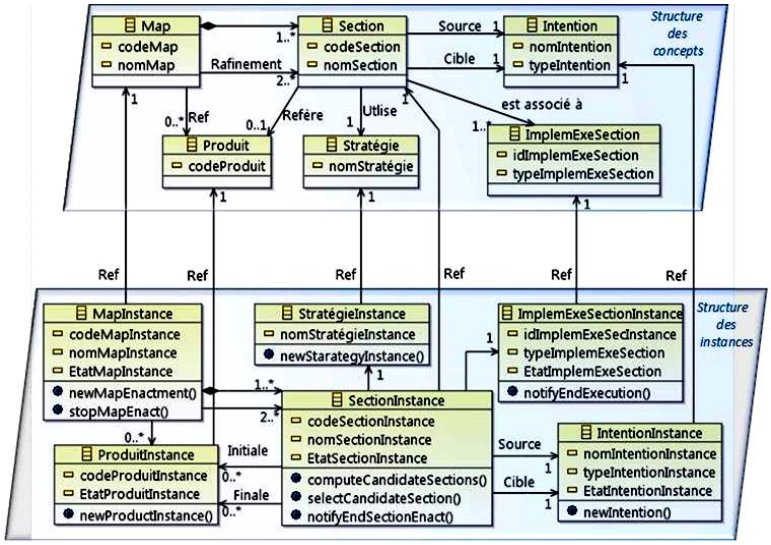

Figure 10. *Modèle structurel à deux niveaux (concept-type et instances) du Map*

La figure 10 présente le résultat de l'application de la 1ère étape. Le méta-modèle statique s'organise autour du concept de section. Une section représente un triplet constitué d'une Intention source, une Intention cible et une Stratégie. Cet élément est associé au concept abstrait *ImplemExeSection* qui renseigne sur la manière

d'exécuter une section (service web, ensemble de directive ou application externe). Une section est aussi reliée à un produit sur lequel porte l'exécution de la section.

Pour opérationnaliser cette structure statique, il faut rajouter des structures d'instances sous forme d'un ensemble de classes-instances (cf. section 4). Ensuite, en se basant sur la sémantique d'exécution du modèle Map, on complète par des méthodes et des attributs d'états nécessaires pour l'exécution du moteur du Map. Ces ajouts ne se font pas forcément dans cette étape, mais peuvent l'être lors de la spécification du schéma dynamique qui exprime la sémantique d'exécution du Map à l'aide d'un schéma événementiel Remora (fig.11). Dans ce schéma, on considère chaque classe du modèle structurel comme un objet susceptible d'avoir des changements d'états et sur lequel peuvent être constatés des événements.

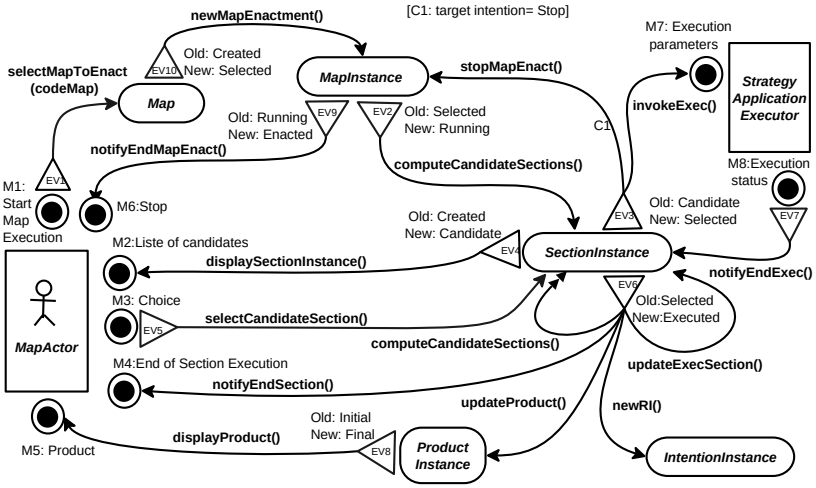

Figure 11. Spécification événementielle de la sémantique d'exécution du Map

Nous définissons par ailleurs deux acteurs externes : *MapActor* pour représenter l'utilisateur final qui exécute une carte, et *StrategyApplicationExecutor* pour un agent externe qui exécute les actions atomiques associées aux stratégies.

6. Évaluation

Le modèle des cartes de processus Map avec lequel nous avons illustré notre démarche a fait l'objet de plusieurs travaux de recherche visant à construire un outil d'exécution. Pour situer l'apport et la pertinence de cette démarche, nous la comparons avec ces travaux antérieurs selon trois catégories de critères (tableau 1) : la démarche, la méta-modélisation et les caractéristiques de l'outil d'exécution final.

La démarche que nous proposons ici est, d'un point de vue méthodologique, beaucoup plus structurée que les autres approches. Les travaux de (Velez, 2003) et (Edme, 2005) adoptent une démarche spécifique, ad-hoc et non structurée, directement orientée vers la production d'un outil logiciel d'exécution de carte Map.

Du point de vue de la méta-modélisation, ces deux travaux utilisent certes un méta-modèle des cartes Map dans leur raisonnement, mais la sémantique d'exécution est directement programmée dans l'outil.

		Approches			
Critères		**(Velez'03)**	**(Edme'05)**	**MétaEdit+** (Mallouli et al.' 13)	**Notre proposition**
Démarche	Nature	Ad-hoc	Ad-hoc	Meta-CASE	IDM
	Générique/ spécifique	Spécifique	Spécifique	Générique	Générique
Méta-modélisation	Formalisme statique	E/R	E/R	GOPRR	MOF (UML)
	Expression de la sémantique d'exécution	100% code orienté objet	Requête SQL + code VB + tables relationnelles	Script MERL + code généré	Schéma événementiel de l'architecture du moteur
Outil d'exéc-ution	Interactivité	Non	Non	Non	Oui
	Maintenabilité	-	-	+	++
	Portabilité	-	-	+	+

Tableau 1. *Évaluation de notre démarche par rapport à des approches similaires*

La démarche exploratoire expérimentée dans (Mallouli et Assar, 2013) est plus proche de celle développée ici. Elle repose certes sur l'usage de méta-modèles explicites grâce à l'utilisation du Meta-CASE (i.e. un outil de méta-modélisation) MetaEdit+ ; néanmoins, la sémantique d'exécution s'exprime dans un ensemble de scripts de génération de code, sans représentation graphique de cette sémantique. C'est suite à ce projet exploratoire que nous avons été convaincus de l'apport potentiel d'une approche IDM pour construire des outils d'exécution de modèles.

Enfin, du point de vue de l'outil logiciel obtenu, les démarches de programmation directes de (Velez, 2003) et (Edme, 2005) aboutissent à des prototypes très difficiles à maintenir et à porter sans redéveloppement complet. La démarche par méta-CASE de (Mallouli et al., 2013) aboutit à une solution plus intéressante, mais en cas d'évolution du langage, elle nécessite une mise à jour du code généré et des scripts de génération. Avec la démarche présentée ici, l'évolution du langage de modélisation nécessite une révision des méta-modèles statiques et dynamiques, et une nouvelle application des règles de transformations pour obtenir un nouvel outil d'exécution. La mise à jour d'un schéma graphique est généralement plus aisée que celle d'un ensemble de lignes de code, fut-ce des scripts de génération de code comme c'est le cas avec MetaEdit+.

7. Conclusion

Lorsqu'on développe un nouveau langage de programmation ou de modélisation, la construction d'outils logiciels pour le supporter se pose rapidement. Les approches de type méta-programmation et méta-modélisation se heurtent à la question ardue de l'expression de la sémantique du langage, surtout si cette

sémantique est exécutable comme c'est le cas avec les langages de modélisation de processus. Par rapport aux approches existantes, nous avons proposé dans cet article une démarche de type IDM qui introduit une spécification graphique et rigoureuse de la sémantique d'exécution du langage. Cette spécification est de nature opérationnelle, elle traduit le fonctionnement d'un outil qui interpréterait les instances de ces modèles, et nous avons défini des règles de transformation pour générer l'architecture logicielle d'un outil d'exécution. La validation de cette démarche est en cours, nous travaillons actuellement à l'implémentation des règles de transformation avec le langage ATL dans le cadre d'un prototype. Néanmoins, une première application de cette approche a permis de mettre en évidence son intérêt par rapport à des démarches plus classiques expérimentées dans des travaux antérieures. La validité externe est pour le moment un peu limitée, elle nécessiterait l'application de la méthode sur d'autres langages et une comparaison plus aboutie avec des approches concurrentes. C'est vers cet horizon que tendent nos travaux futurs.

8. Bibliographie

Atkinson, C., Kühne, T., "Model-Driven Development: A Metamodeling Foundation", *IEEE Software*, 20(5), p. 36–41, 2003.

Bryant, B., Gray, J., Mernik, M., Clarke, P., France, R., Karsai, G., "Challenges and Directions in Formalizing the Semantics of Modeling Languages", *Computer Science and Information Systems*, 8(2), p. 225‑253, 2011.

Bürger, C., Karol, S., Wende, C., Aßmann, U., "Reference Attribute Grammars for Metamodel Semantics", dans B. Malloy, S. Staab, M. van den Brand (éds), *Software Language Engineering*, Vol. 6563, p. 22‑41, Springer, 2011.

Crégut, X., Combemale, B., Pantel, M., Faudoux, R., Pavei, J., "Generative Technologies for Model Animation in the TopCased Platform" dans T. Kühne, B. Selic, M.-P. Gervais, F. Terrier (éds.), *Modelling Foundations and Applications*, p. 90‑103, Springer, 2010.

Edme, M., Proposition pour la modélisation intentionnelle et le guidage de l'usage des systèmes d'information (Thèse de doctorat). Univ. Paris 1 La Sorbonne, France, 2005.

El Kouhen, A., Dumoulin, C., Gerard, S., Boulet, P., "Evaluation of Modeling Tools Adaptation", 2012, consulté à http://hal.archives-ouvertes.fr/hal-00706701.

Eugster, P. T., Felber, P. A., Guerraoui, R., Kermarrec, A.-M., "The many faces of publish/subscribe", *ACM Comp. Surveys*, 35(2), p. 114–131, 2003.

Farail, P., "Toolkit in OPen-source for Critical Applications & SystEms Development", 2012, consulté à http://www.topcased.org/.

Favre, J.-M., Estublier, J., Blay-Fornarino, M., *L'ingénierie dirigée par les modèles au-delà du MDA*, Paris, France: Lavoisier–Hermès Sciences, 2006.

Favre, J.-M., Gasević, D., Lammel, R., Winter, A., "Guest Editors' Introduction to the Special Section on Software Language Engineering", *IEEE Transactions on Software Engineering*, 35(6), p. 737‑741, 2009.

Gargantini, A., Riccobene, E., Scandurra, P., "A semantic framework for metamodel-based languages", *Automated Software Engineering*, *16*(3-4), p. 415‑454, 2009.

Harel, D., Rumpe, B., “Meaningful modeling: what’s the semantics of « semantics »?”, *IEEE Computer*, 37(10), p. 64‑72, 2004.

Hedin, G., “An Introductory Tutorial on JastAdd Attribute Grammars”, dans J. Fernandes, R. Lämmel, J. Visser, J. Saraiva (éds.), *Generative and Transformational Techniques in Software Engineering III*, Vol. 6491, p. 166‑200, Springer, 2011.

Hinze, A., Sachs, K., Buchmann, A., “Event-based applications and enabling technologies, dans *Proc. 3rd ACM Int. Conf. on Distributed Event-Based Systems*, p. 1–15, ACM, 2009.

Jézéquel, J-M., Barais, O., Fleurey, F., “Model Driven Language Engineering with Kermeta”, dans J. M. Fernandes, R. Lämmel, J. Visser, J. Saraiva (éds.), *Generative and Transformational Techniques in Software Engineering III*, p. 201‑221, Springer, 2011.

Jézéquel, J.-M., Combemale, B., Derrien, S., et al. “Bridging the chasm between MDE and the world of compilation”, *Software & Systems Modeling*, 11(4), p. 581‑597, 2012.

Jouault, F., Bézivin, J., Barbero, M., “Towards an advanced model-driven engineering toolbox”, *Innovations in Systems and Software Engineering*, 5(1), p. 5‑12, 2009.

Kelly, S., Tolvanen, J-P., *Domain-specific modeling: enabling full code generation*. Hoboken, N.J.: Wiley-Interscience: IEEE Computer Society, 2008.

Kleppe, A., *Software language engineering: creating domain-specific languages using metamodels*, Addison-Wesley Professional, 2009a.

Kleppe, A., “The Field of Software Language Engineering”, dans D. Gasevic, R. Lämmel, E. V. Wyk (éds.), *Software Language Engineering (SLE’08)*, p. 1‑7, Springer, 2009b.

Knuth, D. E., “Semantics of context-free languages”, *Theory of Computing Systems*, 2(2), p. 127–145, 1968.

Mallouli, S., Assar, S., “Enacting a Requirement Engineering Process with Meta-Tools: an Exploratory Project” *Proceedings 8th Int. Multi-Conf. on Computing in the Global Information Technology (ICCGI 2013)*, p. 208‑213, 2013.

MetaCASE, consulté à http://www.metacase.com/, 2012.

Niknafs, A., Ramsin, R., “Computer-Aided Method Engineering: An Analysis of Existing Environments” dans Z. Bellahsène et al. (éds.), *CAiSE’08*, p. 525‑540, Springer, 2008.

Paakki, J., “Attribute grammar paradigms—a high-level methodology in language implementation“, *ACM Computing Surveys*, 27(2), p. 196–255, 1995.

Rolland, C., Foucault, O., Benci, G., *Conception des systèmes d’information: la méthode REMORA*, Paris, France: Eyrolles, 1988.

Rolland, C., Prakash, N., Benjamen, A., “A Multi-Model View of Process Modelling“, *Requirements Engineering*, 4(1), p. 169‑187, 1999.

Sprinkle, J., Mernik, M., Tolvanen, J., Spinellis, D., “Guest Editors’ Introduction: What Kinds of Nails Need a Domain-Specific Hammer?”, *IEEE Software*, 26(4), 15‑18, 2009.

Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G., “Metamodelling: State of the Art and Research Challenges”, dans H. Giese, G. Karsai, E. Lee, B. Rumpe, B. Schatz (éds.), *Model-Based Engineering of Embedded Real-Time Systems*, p. 57‑76, Springer, 2011.

Velez, F., *Proposition d’un environnement logiciel centré processus pour l’ingénierie des systèmes d’information* (Thèse de doctorat). Univ. Paris 1 La Sorbonne, France, 2003.

Winskel, G., *The Formal Semantics of Programming Languages*, MIT Press, 1993.