Méta-modélisation du Comportement d'un Modèle de Processus : Une Démarche de Construction d'un Moteur d'Exécution

Sana Damak Mallouli

THESE DE DOCTORAT

DE L'UNIVERSITE PARIS 1 PANTHEON – SORBONNE

Réalisée par :

Sana Damak Mallouli

Pour l'obtention du titre de :

DOCTEUR DE L'UNIVERSITE PARIS 1 PANTHEON – SORBONNE

Spécialité : INFORMATIQUE

Méta-modélisation du Comportement d'un Modèle de Processus : Une Démarche de Construction d'un Moteur d'Exécution

---

Soutenue le 25/07/2014 devant le jury composé de :

Slimane HAMMOUDI	Membre du jury
Corine CAUVET	Rapporteur
Samira SI SAID	Rapporteur
Carine SOUVEYET	Directeur de thèse
Saïd ASSAR	Encadrant de thèse

Remerciements

Comme toute thèse, cette recherche a été ponctuée de nombreux moments d'enthousiasme et de joie, mais également de nombreuses périodes de doute et de découragement. L'achèvement de ce travail n'aurait pas été possible sans la précieuse contribution de nombreuses personnes que je veux remercier ici.

Je souhaiterais tout d'abord adresser mes sincères remerciements à la directrice de cette thèse Madame Carine Souveyet, Professeur à l'université Paris1 Panthéon-Sorbonne, pour la confiance qu'elle m'a accordée en acceptant de diriger cette thèse, puis pour m'avoir guidée, encouragée et conseillée, tout au long de ce travail de recherche. Je la remercie infiniment pour m'avoir fait bénéficier de sa grande compétence, de sa rigueur intellectuelle et de ces précieux conseils.

Je tiens également à exprimer ma plus vive reconnaissance à mon encadrant de recherche Mr Saïd Assar, Maitre de conférences à Institut Mines Telecom -Telecom Ecole de Management. Il m'a encouragée par ses orientations sans cesser d'être une grande source de motivation et d'enthousiasme. Je le remercie sincèrement pour sa disponibilité, sa persévérance, et ses encouragements continus. Je tiens à le remercier également pour ses qualités humaines d'écoute et son soutien moral pendant les moments difficiles.

Je remercie Madame Corine Cauvet, Professeur à l'université Aix-Marseille 3, et Madame Samira Si-Saïd Cherfi, Maître de conférences, HDR à la Conservatoire National des Arts et Métiers (CNAM) Paris, qui ont eu la gentillesse d'accepter le rôle de rapporteur de cette thèse. Elles ont contribué par leurs nombreuses remarques et suggestions à améliorer la qualité de ce mémoire, et je leur en suis très reconnaissante.

Je remercie également Mr Slimane Hammoudi, Professeur ESEO, HDR à l'Université d'Angers d'avoir accepté de présider le jury de cette thèse.

Je remercie chaleureusement, Madame Colette Rolland, Professeur Emérite à l'Université de Paris 1 Panthéon – Sorbonne, pour la confiance qu'elle m'a témoignée en m'accueillant dans son équipe. Mes remerciements vont également à Mr Camille Salinesi, Professeur - Directeur du Centre de Recherche en Informatique à l'Université de Paris 1 Panthéon – Sorbonne.

Je tiens aussi à mentionner le plaisir que j'ai eu à travailler au sein du CRI (Centre de Recherche Informatique) et j'en remercie ici tous les membres pour leurs amitiés et leurs encouragements. Une pensée particulière va à Amina, Assia, Adrian, Ali, Bénédicte, Camille, Carine, Charlotte, Daniel, Elena, Ghazalleh, Hela, Ines, Islem, Kahina, Manuele, Saïd, Salma, Selmine, Raul, Rawia et Rebbeca.

Ces années de thèses resteront ancrées dans ma mémoire comme une période très enrichissante de ma vie. Cet aboutissement ne pouvait se réaliser sans être associé au soutien

des personnes les plus proches de moi. Merci à mes parents et mes grands parents pour l'affection qu'ils m'ont toujours manifestée et pour leur soutien lors des moments les plus difficiles de cette thèse même quand je suis loin d'eux. Merci à mes adorables sœurs Olfa et Wafa pour leur amour inconditionnel et leur bonne humeur. Merci infiniment à ma belle-famille, à mes proches et à mes amis notamment ma tante Souad et Faten pour leurs encouragements continus. L'affection qu'ils m'ont apportée durant ces années m'a aidée à garder le moral haut et à achever ma thèse.

Je garde pour la fin un remerciement particulier pour mon mari, Wissam. Je le remercie du fond du cœur de m'avoir aidée, soutenue, encouragée et surtout aimée. Ces dernières années n'ont pas été très simples, il m'a supportée dans les moments les plus difficiles et m'a aidée à retrouver le sourire pendant les périodes de doute.

Pour finir, je souhaiterais dédier ce travail à tous ceux que j'aime. Qu'ils y trouvent ici l'expression de ma profonde affection et de mes plus sincères remerciements.

Sana

Résumé

De nos jours, le nombre de langages de modélisation ne cesse d'augmenter en raison de différentes exigences et contextes (par exemple les langages spécifiques au domaine). Pour être utilisés, ces langages ont besoin d'outils pour réaliser différentes fonctionnalités comme l'édition, la transformation, la validation et l'exécution de modèles conformes à ces langages. La construction de ces outils est un enjeu et un objectif important aussi bien dans la communauté du génie logiciel que celle des Systèmes d'Information. C'est une tâche non-triviale qui fait appel à des approches différentes parmi lesquelles l'utilisation des environnements méta-CASE et des langages de méta-programmation.

Par rapport à une approche ad-hoc, les méta-CASE définissent un support outillé et une démarche basée sur la méta-modélisation. Ils apportent des améliorations significatives à la problématique de construction d'outils. Néanmoins, des limitations majeures persistent, notamment pour les langages de modélisation des processus, à cause de la complexité de l'expression de la sémantique opérationnelle d'un modèle de processus et la capture de la logique d'exécution de celui-ci. La construction de ces outils selon une approche ad-hoc engendre un coût élevé, des risques d'erreurs et des problèmes de maintenabilité et de portabilité. En outre, un outil d'exécution de modèle doit satisfaire un critère d'interactivité avec son environnement d'exécution. Cet aspect n'est pas suffisamment pris en compte dans les travaux de recherche actuels sur la spécification des langages de modélisation.

Pour répondre à cette problématique, nous proposons dans cette thèse une démarche dirigée par les modèles qui permet de dériver l'architecture d'un moteur d'exécution à partir de la spécification conceptuelle d'un langage de modélisation de processus. Cette spécification repose sur une méta-modélisation élargie qui intègre l'expression de la sémantique d'exécution d'un méta-modèle de processus. Elle est composée, d'une part, d'une structure à deux niveaux d'abstraction qui permet de représenter de manière générique les modèles à exécuter et les instances générées lors de leur exécution. D'autre part, cette spécification est complétée par une représentation déclarative et graphique du comportement du méta-modèle de processus. Pour cette représentation, nous avons choisi un formalisme orienté événement qui reflète la vision systémique et les différentes interactions du modèle de processus avec son environnement. Finalement, afin d'exploiter la sémantique d'exécution, nous proposons des règles de transformation permettant de dériver l'architecture technique d'un outil d'exécution sous une forme standard pour pouvoir l'implémenter dans un environnement de génération de code existant, le code généré correspondra à l'outil d'exécution souhaité.

La démarche proposée a été appliquée dans le cas d'un modèle de processus intentionnel appelé Map. Cette application a permis d'explorer la faisabilité de la proposition et d'évaluer la qualité de la spécification de l'outil d'exécution obtenue par rapport aux exigences fixées. La pertinence de notre proposition est qu'elle permet de guider l'ingénieur dans le processus de spécification et de construction d'un outil d'exécution tout en minimisant l'effort de programmation. De plus, en appliquant les étapes de la démarche proposée, nous sommes en mesure de fournir un outil d'exécution d'une certaine qualité ; à savoir un outil interagissant avec son environnement, facilement maintenable et à moindre coût.

Abstract

Nowadays, the number of modeling languages is increasing due to different requirements and contexts (*e.g.*, domain-specific languages). These languages require appropriate tools to support them by achieving various functionalities such as editing, transforming, validating and implementing models that conform to these languages. The construction of such tools constitutes a challenging goal for both software engineering and information systems communities. This non obvious task is involves different approaches such as meta-CASE and meta-programming languages.

Unlike ad-hoc approach, meta-CASE environments define software tools and meta-modeling based approaches to improve tools construction. However, major limitations still exist, in particular for process modeling languages, due to the complexity of expressing process model operational semantics and capturing its execution logics. These limitations commonly lead to a manual construction of tools which is labor-intensive, error-prone and poses maintainability and portability issues. Besides, a process model execution tool should satisfy the criterion of interactivity with its environment. This aspect is not sufficiently taken into account in actual research works on the specification of modeling languages.

To address this problem, we propose in this thesis a model driven approach to derive the software architecture of an enactment engine from the conceptual specification of the process modeling language. This specification relies on an enlarged meta-modeling approach that includes the expression of the execution semantics of a process meta-model. It consists of a structural specification incorporating both concepts structures and instances that are generated during execution. This specificaiton is completed by a declarative and graphical representation of the process behavioural meta-model. For this representation, an event-oriented paradigm is adopted to reflect the dynamic vision and the different interactions of the process model with its environment. Finally, transformation rules are proposed in order to obtain an enactment engine architecture in an object-oriented form. The enactment engine will be obtained by implementing the object-oriented architecture in an existing code-generation environment.

The proposed approach has been applied to the case of an intentional process model called “Map”. This application case aims at evaluating the feasibility of the proposed approach and assessing the quality of the derived enactment engine in regard with initial requirements. The proposed approach is relevant, since on one hand, it allows better guidance for engineers during the specification of the modeling language and the implementation of an enactment tool, and on the other hand, it minimizes programming efforts. In addition, by applying the different steps of the proposed approach, we are able to guarantee a certain quality of the enactment tool ensuring interactivity, maintainability and development low cost.

Table des matières

Table des figures

Table des tableaux

CHAPITRE 1 : INTRODUCTION

1.1. Les Outils CASE

La complexité croissante des systèmes d'information (SI) rend le développement des SI lourd et coûteux (Giraudin, 2007). Afin de maitriser cette complexité, différentes techniques et approches sont proposées par l'ingénierie des SI (ISI). Le but des outils connus sous le nom d'Ateliers de Génie Logiciel (CASE en anglais pour Computer-Aided Software Engineering) est d'assister et d'aider l'ingénieur à concevoir et développer des SI (Fuggetta, 1993) grâce à un ensemble de fonctionnalités permettant d'alléger le travail de l'ingénieur et aussi d'introduire l'automatisation dans la mise en œuvre d'une méthode d'ingénierie (Figure 1).

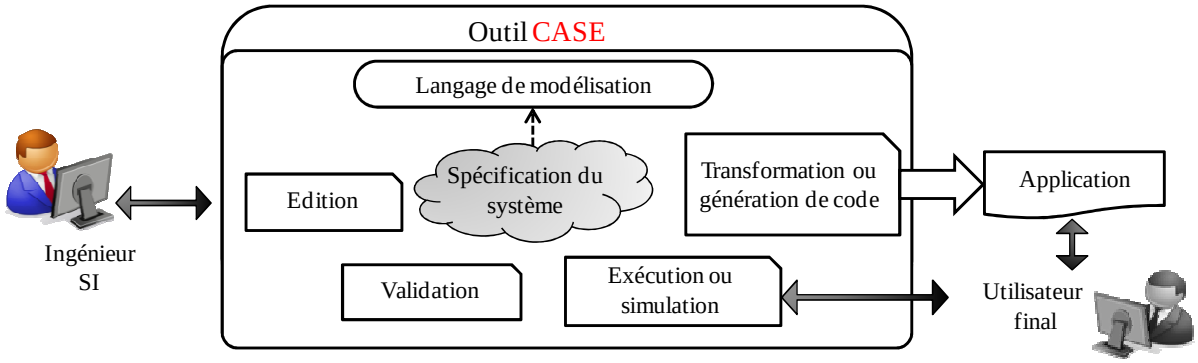


Figure 1. Les fonctionnalités d'un outil CASE dans l'ingénierie des SI

Comme exemples d'outil CASE, on peut citer StarUML[1], Power AMC[2], WebRatio[3], NetBeans[4], Rational Rose[5], Eclipse[6]. Ces outils proposent diverses fonctionnalités avec des niveaux de sophistication variables, et ne couvrent pas forcément les mêmes phases du cycle de vie d'un système. Une distinction courante est entre *upper-* et *lower-CASE* (Fuggetta, 1993). Un *upper-CASE* est dédié aux phases d'analyse et de conception et il est généralement associé à une méthode d'ingénierie de SI. Il inclut des interfaces graphiques pour éditer les spécifications d'un système et des dictionnaires de données pour les stocker, des fonctions pour gérer la documentation, des moyens de contrôle de qualité et de correction de schémas. Un *lower-CASE* est destiné aux phases d'implémentation avec des éditeurs syntaxiques, des librairies de programmes, des compilateurs, des générateur d'interfaces homme/machine, ou encore des outils pour la mise au point de programmes et la configuration des codes sources. La distinction entre *lower-* et *upper-CASE* s'estompe lorsque l'outil intègre des fonctionnalités de génération de code et de transformation de modèle, c'est ce que préconise l'ingénierie dirigée par les modèles.

[1] http://staruml.sourceforge.net/en/
[2] http://wikipedia.org/wiki/PowerAMC
[3] http://www.webratio.com/
[4] https://netbeans.org/
[5] http://www.ibm.com/developerworks/rational/products/rose/
[6] Eclipse : http://www.eclipse.org/

1.2. L'IDM et l'ingénierie des langages

Les outils CASE jouent un rôle important dans l'ingénierie dirigée par les modèles (IDM ou MDE[7]). En effet, l'IDM place les modèles au centre du processus d'ingénierie des systèmes, et grâce à des outils logiciels adéquats, attribue aux modèles une fonction productive par opposition au rôle contemplatif[8] habituel (Favre, 2006). Cette production s'effectue à travers des transformations, pour aller vers une génération automatique – ou semi automatique – de logiciels pour des plateformes d'implémentation cibles. Avec le développement de l'IDM, la construction des outils CASE est devenue un enjeu et un objectif important. C'est une tâche non-triviale qui fait appel à la définition de langages de modélisation et à l'implémentation de fonctionnalités non seulement de transformation et de génération de code mais aussi de test, vérification et exécution. Elle soulève des questionnements multiples : Quelle architecture pour l'outil? Quelles fonctionnalités à fournir? Est ce qu'on cherche à développer un générateur de code, un moteur d'exécution ou simplement un éditeur de modèles? Comment implémenter chacune de ces fonctionnalités? Comment garantir certaines propriétés de l'outil telles que la portabilité (en cas d'évolution de la plateforme sur laquelle s'exécute l'outil), la maintenabilité (en cas d'évolution des langages de modélisation), ou la réutilisabilité (en cas de développement de nouveaux langages)? Enfin, quelles méthodes et quels (méta-) outils pour concevoir, spécifier et implémenter ces outils?

Pour traiter ces problématiques, en plus de l'IDM, un domaine de recherche complémentaire a émergé : *l'ingénierie des langages* pour le génie logiciel[9] (Kleppe, 2009), (Favre et al., 2009). Il est à l'intersection de multiples thématiques de l'ingénierie des SI et du génie logiciel telles que la modélisation conceptuelle et la transformation de modèles, l'ingénierie des méthodes, la méta-modélisation, les outils méta-CASE, les langages spécifiques au domaine (ou DSM[10]) ou encore la compilation et les langages de programmation. La problématique des langages pour le génie logiciel dépasse le cadre restreint des langages de programmation (tels que C ou Java). Les langages de génie logiciel sont des artéfacts au cœur du développement de différents domaines d'ingénierie informatique. Ils sont conçus et développés pour résoudre une problématique générique de représentation de systèmes (UML, XML), ou des problématiques plus spécifiques telles que l'exécution des processus à base de services web (BPEL[11]), l'ingénierie des besoins (i*[12], Tropos[13]) ou la modélisation des processus d'ingénierie (SPEM[14]). La spécification de tels langages, et surtout la construction d'outils pour les accompagner, sont au cœur du sujet de cette thèse.

[7] MDE: Model-Driven Engineering
[8] (un modèle contemplatif est utilisé pour représenter graphiquement une spécification conceptuelle ou bien pour la documentation et la compréhension d'un projet
[9] SLE: Software Language Engineering
[10] DSM: Domain Specific Modeling
[11] BPEL: Business Process Execution Language
[12] i*: http://www.cs.toronto.edu/km/istar/
[13] Tropos: http://www.troposproject.org/
[14] SPEM: Software Process Engineering Metamodel

1.3. Méta-CASE, ingénierie des méthodes et CAME

Apparue dans les années 1990, l'objectif de la technologie méta-CASE est de faciliter et accélérer la conception et le développement d'outils CASE dédiés à une méthode d'ingénierie particulière (Alderson, 1991) (Smolander et al., 1991). Alors que la méthode est prédéfinie dans un outil CASE, les outils méta-CASE permettent la personnalisation et la construction de méthodes d'ingénierie afin de les adapter aux besoins spécifiques des ingénieurs (Kelly et Smolander, 1996) (Assar et al., 2012).

La technologie méta-CASE introduit un niveau d'architecture supplémentaire pour la spécification de la méthode que l'ingénieur souhaite outiller. Alors qu'une méthode comporte des « produits » (tout livrable ou artéfact produit par l'application de la méthode) et des « processus » (des cheminements suivis pour aboutir aux produits) (Olle, 1988), il est fréquent dans la littérature des méta-CASE qu'une méthode soit restreinte à sa composante « produit ». C'est le cas notamment dans les premières publications de l'outil MetaEdit+ par exemple (Smolander et al., 1991) (Kelly et al., 1996), où par la suite, c'est la spécificité du « produit » pour un domaine qui est mise en avant (Kelly et Tolvanen, 2008) (Kelly et al., 2013). L'ingénierie des méthodes a grandement contribué à expliciter l'importance du processus dans la description d'une méthode (Brinkkemper et al., 1996). Et c'est dans le domaine des environnements de génie logiciel centrés processus (PCSE[15] ou PSEE[16]) que plusieurs langages ont été proposés pour modéliser et spécifier les processus d'ingénierie logiciel (Fuggetta, 1996) (Zamli and Lee, 2001)(Arbaoui et al., 2002). Le terme CAME[17] a été introduit pour désigner des outils logiciels semblables aux méta-CASE, la différence se situant surtout au niveau de la finalité : pour un méta-CASE, c'est la génération d'un outil spécifique (pour un langage de modélisation de produit) ; pour un CAME, c'est la spécification d'une méthode avec les deux perspectives « produit » et « processus » (Heym et Österle, 1993) (Saeki, 2003). Ces deux termes sont souvent semblables sinon identiques étant donné que la spécification d'une méthode nécessite la construction d'un outil correspondant. L'outil MetaEdit+ est ainsi mentionné dans la littérature en tant que méta-CASE pour l'outillage des DSM, mais aussi en tant qu'environnement CAME pour outiller des méthodes (Kelly et Tolvanen, 2008) (Niknafs and Ramsin, 2008) (Kelly et al., 2013).

Le niveau d'architecture supplémentaire qu'introduit les méta-CASE et les CAME inclut, d'une part, des langages de modélisation et de méta-modélisation pour la méta-spécification d'une méthode, et d'autre part, des fonctionnalités pour la construction (ou génération semi-automatique) d'un outil CASE à partir de cette méta-spécification. Le langage de méta-modélisation sert à définir les méta-modèles de la méthode, c'est-à-dire les méta-modèles de produit et de processus. Un méta-modèle de produit contient les concepts, les règles et les notations graphiques du langage de modélisation des produits dans une méthode donnée. Par exemple, un méta-modèle d'UML (ou de BPMN[18] respectivement) spécifie des concepts tels que 'classe' et 'héritage' (ou 'tâche' et 'événement' respect.) et des liens entre ces concepts.

[15] PCSE: Process Centred Software Engineering
[16] PSEE: Process-centred Software Engineering Environments
[17] CAME: Computer Aided Method Engineering
[18] Business Process Modeling Notation

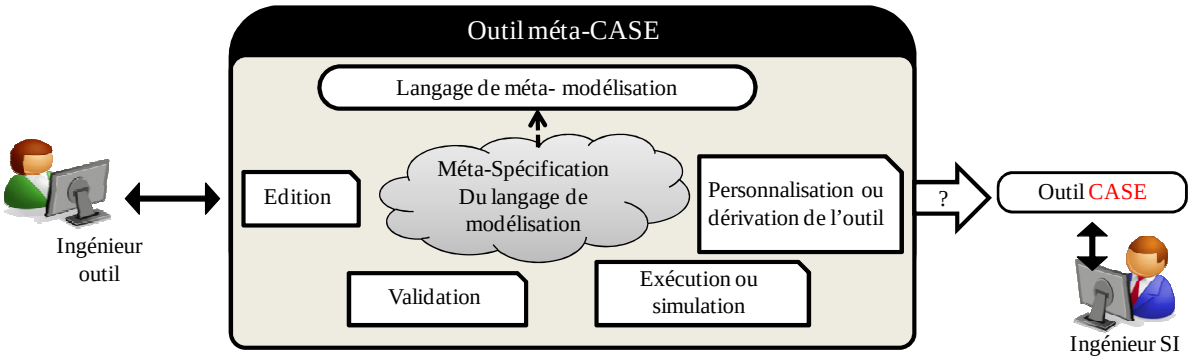


Figure 2. Les fonctionnalités d'un outil méta-CASE

L'ingénieur outil utilise le langage de méta-modélisation pour définir les méta-modèles de la méthode pour laquelle il souhaite construire un outil CASE (Figure 2). Pour la perspective « produit », cette méta-définition doit couvrir les trois facettes fondamentales d'un langage de modélisation : **la syntaxe abstraite**[19] (les concepts et les liens entre ces concepts), **la syntaxe concrète** (la forme extérieure – graphique ou textuelle – des concepts et des liens) et **la sémantique** (le sens attribué à ces concepts).

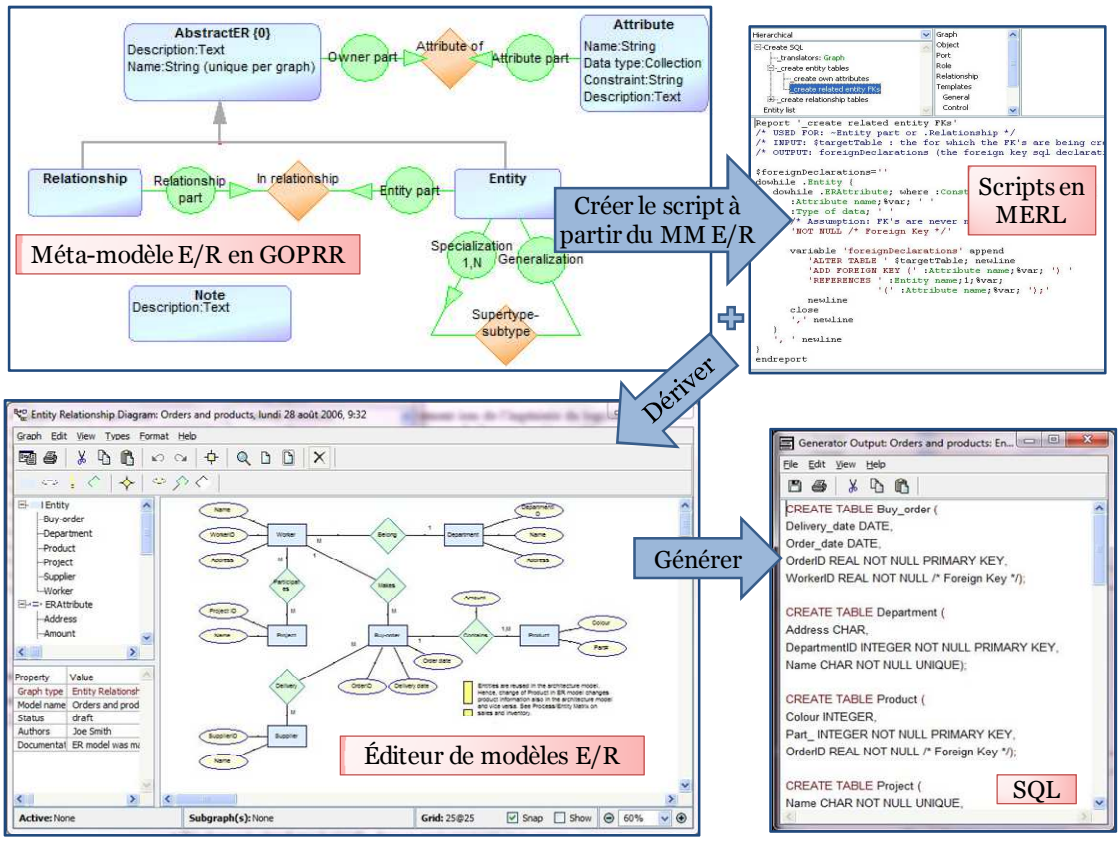


Figure 3. Exemple de conception d'un CASE pour le modèle E/R avec l'outil MetaEdit+

Dans l'outil MetaEdit+ (Kelly and Smolander, 1996) par exemple, la méta-spécification s'exprime avec le langage de méta-modélisation GOPRR[20]. Un outil CASE cible dérivé à partir de cette méta-spécification comporte par défaut des fonctionnalités d'édition de modèle.

[19] La syntaxe abstraite est généralement définie, en IDM, sous la forme d'un ensemble de classes et un modèle issu du langage est un ensemble d'objets, instances de ces classes.
[20] GOPRR: Graph, Object, Property, Relationship, Role

Le méta-modèle peut être complété par des scripts écrits dans un méta-langage spécifique (MERL). Ces scripts, qui incluent des commandes de navigation dans les instances des méta-modèles, peuvent décrire un comportement permettant à l'outil CASE cible d'avoir une fonctionnalité de génération de code.

La figure 3 illustre cette démarche pour construire un outil de modélisation à base de la notation Entité/Relation. Le générateur de code implémente les règles de transformation de structure du type « une entité devient une table » ou « une association de cardinalité 1-N devient une clé étrangère ». Alors que la phase de méta-modélisation se fait d'une manière déclarative et avec des outils graphiques, le processus de construction du générateur de code est de nature algorithmique à l'aide de simple éditeur de texte. Dans ces scripts séquentiels organisés en modules non paramétrés, l'ingénieur méthode utilise sa connaissance implicite de la sémantique des méta-modèles manipulés pour définir le générateur de code.

1.4. Méta-modèles et sémantique d'exécution

Généralement, la pratique de la méta-modélisation consiste à spécifier des méta-modèles contemplatifs qui reflètent *l'aspect statique des modèles*. **L'aspect statique d'un modèle** représente l'ensemble des concepts qui ont servi à décrire ce modèle ainsi que les liens entre ces concepts (Jeusfeld et al., 2009). On utilise aussi le terme *aspect structurel*. **L'aspect dynamique** complète l'aspect statique et renseigne sur la manière avec laquelle les instances des concepts vont intéragir lors de l'exécution du méta-modèle. Cela revient à décrire le **comportement dynamique** des instances des concepts, c'est-à-dire la **sémantique d'exécution** du modèle.

Selon le langage de méta-modélisation utilisé, un méta-modèle contemplatif peut être complété par la spécification de certaines contraintes. En utilisant UML par exemple pour construire un (méta-) diagramme de classes, il est possible de compléter cette spécification avec des contraintes exprimées avec le langage OCL. Pour un langage de modélisation de processus (tels que XPDL[21], BPMN ou SPEM), la méta-modélisation telle qu'elle vient d'être illustrée avec l'outil MetaEdit+, se restreint à décrire la syntaxe abstraite des concepts qui correspond à l'aspect statique (ou structurel) de ces concepts, et ne prend pas en compte la sémantique des concepts. Par exemple, un méta-modèle de BPMN représente un « produit » avec des objets et des propriétés (les concepts), des relations et des rôles (les liens entre ces concepts). Tandis que la sémantique d'un modèle de processus BPMN correspond à un certain comportement dans un environnement d'exécution donné. Ce comportement peut être assuré par un moteur d'exécution qui interprète les instances du modèle de processus, interagit avec l'environnement et fournit un résultat qui varie en fonction de cette interaction (Figure 4).

[21] XML Process Definition Language

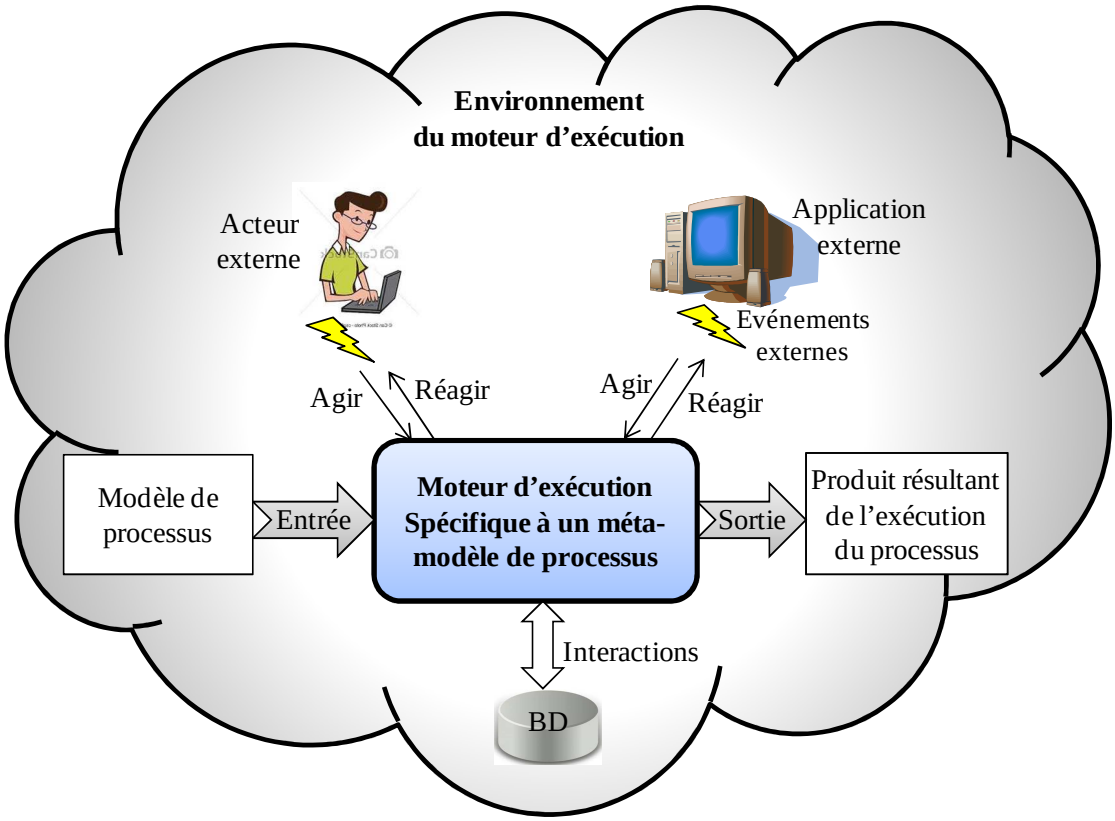


Figure 4. Schéma générique pour un moteur d'exécution

Pour exécuter un processus, il est en effet souvent nécessaire de communiquer et d'interagir avec d'autres acteurs humains et/ou agents logiciels afin de déléguer des tâches ou de recevoir des éléments indispensables pour continuer l'exécution (un fragment de produit, une décision, un choix, etc.). Par ailleurs, l'évocation de l'exécution d'un modèle de processus est généralement faite à l'aide du terme anglo-saxon « ***enactment*** » pour souligner la différence qu'il y a entre « *exécuter un programme* » et « *énacter un processus* » et mettre en œuvre l'aspect distribué, coopératif et parfois asynchrone d'un processus.

L'expression de la sémantique des langages est une problématique connue et bien développée depuis plusieurs décennies dans le domaine de la compilation et des langages de programmation. Les techniques les plus probantes s'appuient sur la grammaire des attributs (Knuth, 1968), (Paakki, 1995). Pour les langages de modélisation, l'intérêt pour l'expression explicite de la sémantique est assez récent (Harel and Rumpe, 2004). L'intérêt pour l'expression de la sémantique des modèles va de pair avec la question de comment rendre exécutable un modèle, à priori contemplatif, décrit à l'aide d'un méta-langage (Muller et al., 2005).

La direction suivie pour résoudre ces problèmes cherche à enrichir les langages et environnements de méta-modélisation pour permettre l'expression explicite et adéquate de la sémantique d'un langage de modélisation. Dans (Paige et al., 2006) par exemple, il est proposé d'enrichir le méta-méta-modèle MOF[22] avec un langage d'action qui permettra de programmer des « méta-méthodes » rattachées aux méta-classes d'un méta-diagramme MOF.

[22] MOF : Meta Object Facility, formalism de méta-modélisation proposé par l'OMG (http://www.omg.org/mof/)

1.5. Constats et problématique

Nous constatons qu'aujourd'hui, par rapport à l'approche ad-hoc, les environnements méta-CASE définissent des démarches basées sur la méta-modélisation, et qui apportent des améliorations significatives aux problématiques de spécification d'un langage de modélisation et de construction d'outils correspondants. Néanmoins, trois limitations majeures persistent:

- D'une part, la méta-modélisation est fortement orientée « produit » ; d'un point de vue d'ingénierie de méthode, la spécification du processus méthodologique est complexe, difficile, sinon impossible, à capturer et à énacter. Les tentatives de l'environnement MetaEdit+ dans ce sens par exemple (Koskinen et Marttiin, 1998) n'ont finalement pas été retenues dans l'outil commercialisé par la suite par l'entreprise MetaCASE[23].
- D'autre part, cette spécification des « produits » n'en capture que la syntaxe abstraite et une partie réduite de la sémantique sous forme de contraintes. Si le produit est un modèle de processus, et si l'ingénieur méthode souhaite exécuter ces processus, et cherche à obtenir un moteur d'exécution de ces modèles, la construction d'un tel moteur se fait manuellement. Elle nécessite une connaissance en programmation et un effort supplémentaire de la part de l'ingénieur. De plus, cette implémentation artisanale entraine des difficultés de maintenance du moteur d'exécution car toute évolution des modèles oblige l'ingénieur outil à propager les changements dans le code (les scripts Merl dans le cas de MetaEdit+).
- Enfin, la spécification du modèle de processus n'integre pas l'aspect interactif avec des acteurs externes ce qui implque que l'outil d'exécution associé ne va pas satisfaire des contraintes en termes d'interactivité avec l'environnement d'exécution.

1.6. Objectif de la thèse

Notre objectif est de proposer une approche de spécification de langage de modélisation de processus et de construction d'outils d'exécution de modèles permettant de satisfaire les contraintes suivantes :

- Cette approche doit offrir un support de méta-modélisation avec une prise en compte claire et explicite de la sémantique d'exécution du modèle et un formalisme graphique pour une représentation visuelle de cette spécification.
- L'approche devra comporter une démarche et des techniques pour exploiter la sémantique d'exécution du méta-modèle de processus afin d'en dériver de manière systématique une architecture d'un moteur d'exécution qui satisfait des contraintes de maintenabilité et de portabilité.
- La spécification du modèle de processus doit interagir avec son environnement pour que le moteur d'exécution associé à ce modèle puisse avoir ce critère d'interactivité.
- Cette approche doit être suffisamment générique pour pouvoir être appliquée sur n'importe quel modèle de processus.

[23] http://www.metacase.com

1.7. Démarche de recherche suivie dans cette thèse

La démarche de recherche de la thèse est schématisée à la Figure 5.

La première étape est un projet exploratoire que nous avons mené pour acquérir une expérience pratique dans le développement d'un outil par méta-modélisation. Ce projet à été réalisé dans le cadre d'un co-encadrement d'un Master Recherche pour explorer une technologie méta-CASE pour le développement d'outil d'exécution pour un modèle de processus intentionnel (Harrous, 2011) et dont une synthèse des résultats a été publiée dans (Mallouli, 2013). A l'issue de ce projet exploratoire, nous avons déterminé un ensemble de constats qui ont permis de fixer des exigences précises pour répondre à notre problématique.

La seconde étape de notre recherche est une synthèse de l'état de l'art concernant l'exécutabilité des modèles, la construction d'outils ainsi que les langages et outils de méta-modélisation. Cette étape a été réalisée en parallèle avec un co-encadrement de deux mémoires de Master Recherche pour étudier et comparer les langages et les outils de méta-modélisation (Souag, 2010) (Zaidi, 2010). A la fin de l'étude de l'état de l'art, nous avons réalisé une comparaison entre différentes approches de construction d'outils en se basant sur des critères définis à la lumière des exigences fixées lors de la première étape.

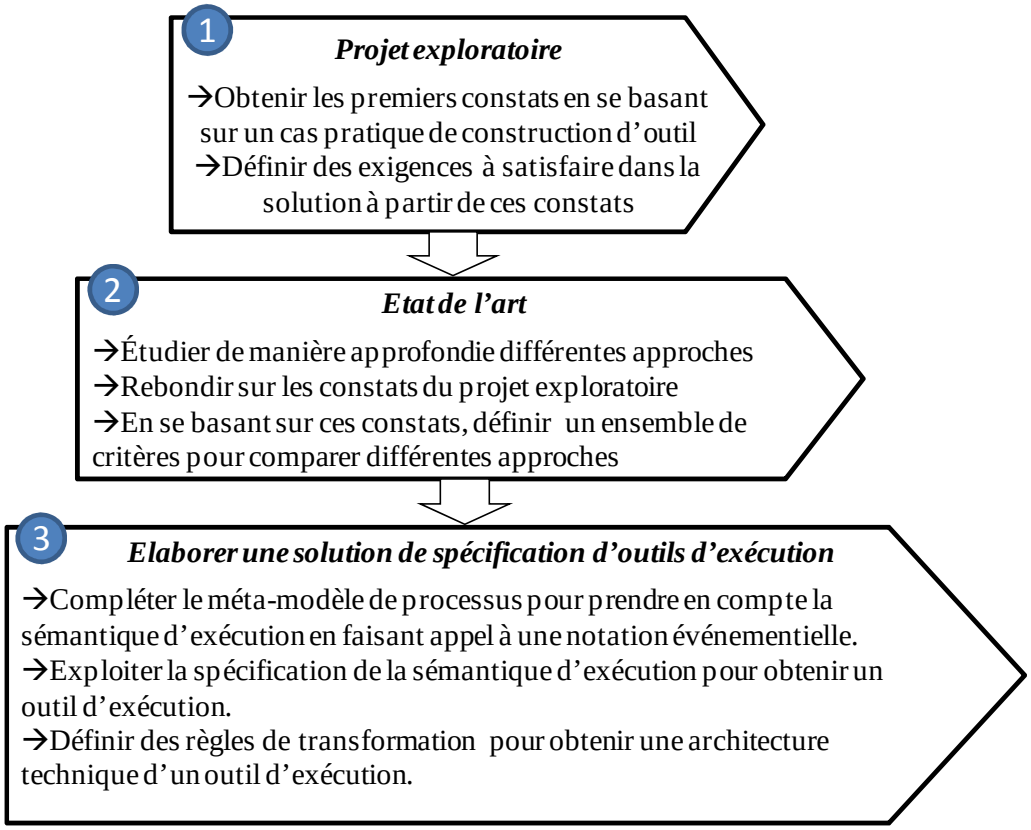


Figure 5. Les étapes de la démarche de recherche suivie

La troisième étape correspond à l'élaboration de la solution que nous proposons. Les principales idées de notre démarche consistent à élargir le périmètre d'expression d'un méta-modèle en y ajoutant la spécification de sa sémantique d'exécution, et à introduire plus d'automatisation dans le processus de construction d'outils grâce à l'exploitation des spécifications du niveau « méta ».

1.8. Aperçu de la solution proposée

Contrairement aux approches classiques, qui se basent sur la structure d'un méta-modèle puis définissent la sémantique d'exécution directement dans le code ou sous forme de scripts attachés aux éléments statiques, notre démarche utilise un formalisme soigneusement choisi pour la spécification de la sémantique d'exécution, ce qui facilite la génération de l'architecture d'un outil d'exécution. Notre solution comporte trois étapes schématisées à la Figure 6 :

- La première étape définit le méta-modèle structurel du langage de modélisation.
- La seconde étape définit la sémantique d'exécution du modèle à travers un méta-modèle de comportement. Pour ce faire, il faut au préalable compléter le méta-modèle structurel en introduisant un second niveau de représentation qui correspond aux instances du méta-modèle. Ce second niveau est nécessaire pour l'expression de la sémantique d'exécution ; en effet, celle-ci s'exprime en manipulant les instances du méta-modèle au moment de l'exécution d'un modèle afin d'extraire l'instance intéressante à un instant t précis.
- La troisième étape applique des règles de transformation pour dériver une architecture d'outil d'exécution à partir de la spécification conceptuelle.

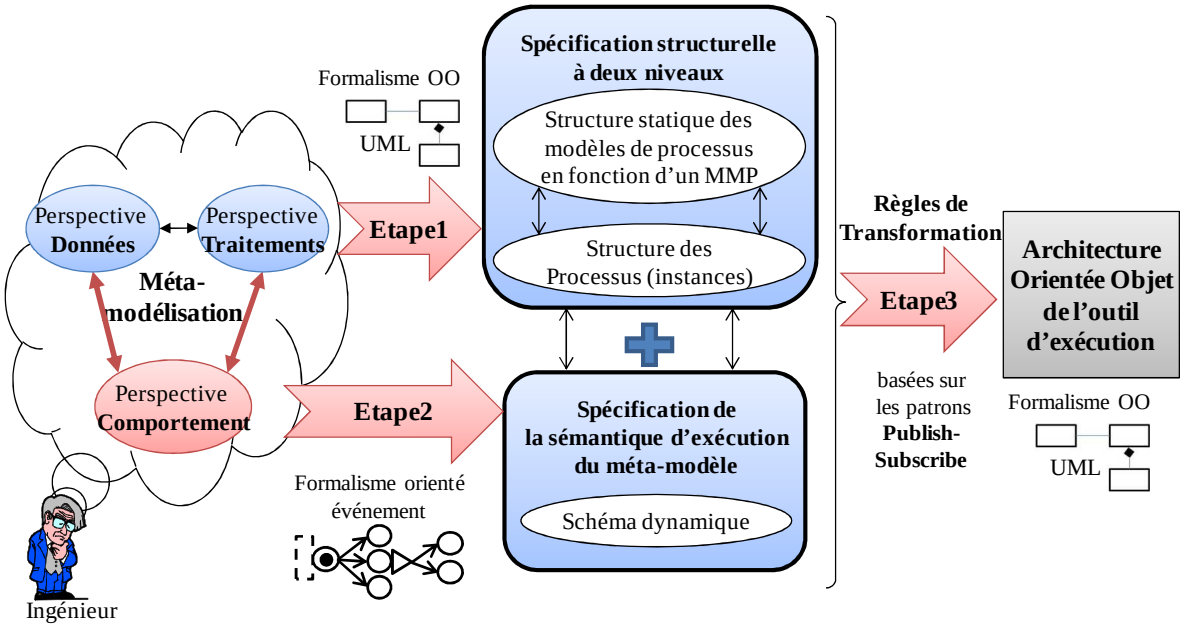


Figure 6. Aperçu de l'approche

Pour spécifier le comportement du modèle et sa sémantique d'exécution, nous adoptons une persepctive «systémique» du modèle de comportement et introduisons une notation basée sur les événements (Assar et al., 2011). La vision systémique préconise de voir le comportement d'un système selon un schéma <action-réaction> où l'action est un stimulus externe de l'environnement du système et la réaction est l'activité que le système déclenche en réponse. Concrètement, notre approche consiste à introduire de nouveaux concepts (Evénement, Acteur, Transition, Trigger) dans le langage de méta-modélisation. Ces concepts serviront à spécifier le comportement dynamique des structures définies au niveau des méta-modèles. De cette manière, la spécification de la sémantique d'exécution du modèle intègre

une représentation des éléments de l'environnement du processus dans lequel le modèle s'exécutera ainsi que les éléments échangés avec lui. La sémantique d'exécution est ainsi exprimée à travers un schéma graphique.

Cette expression de la sémantique d'exécution est de nature opérationnelle, elle montre l'interprétation du modèle par un outil d'exécution qui gère les événements, assure les interactions avec l'environnement et contrôle l'interprétation du modèle. Cette sémantique opérationnelle peut de ce fait être traduite en une architecture d'outil logiciel à l'aide d'une technique de type « par traduction » (Combemale et al., 2006). Nous élaborons donc un ensemble de règles pour transformer la spécification conceptuelle en une spécification exécutable. En raison de la nature dynamique et interactive du modèle de comportement, nous choisissons le patron publication/abonnement (publish/subscribe) comme modèle cible d'implémentation du concept d'événement au niveau conceptuel. Dans ce modèle, les abonnés (des agents ou composants logiciels) manifestent leur intérêt pour un événement, et sont automatiquement notifiés de toute occurrence de cet événement (Eugster et al., 2003). Une occurrence d'événement est donc propagée de manière synchrone ou asynchrone à tous les abonnés qui sont inscrits à cet événement. Les abonnés réagissent àla notification d'un événement par le déclenchement d'un traitement spécifié dans une méthode particulière.

En appliquant un ensemble de règles de transformation, la spécification conceptuelle permet de dériver l'architecture d'un outil logiciel directement implémentable.

Notre solution présente deux contributions majeures:

- *Un apport conceptuel* qui consiste à capturer la sémantique opérationnelle du méta-modèle avec une vision systémique et l'exprimer à travers des concepts événementiels.
- *Une démarche par méta-modélisation* qui exploite les méta-modèles conceptuels pour arriver à la phase d'implémentation avec une architecture exécutable en appliquant de manière systématique des règles de transformation.

L'originalité de ce travail se résume dans les points suivants :

- L'approche suivie est de type « méta-démarche » et permet de guider l'ingénieur dans le processus de spécification et de construction d'un outil d'exécution.
- Dans la partie spécification conceptuelle, le comportement de l'outil d'exécution à construire est représenté par un modèle orienté événement,
    - *Le modèle de comportement choisi est fondé sur le paradigme systémique,*
    - *Ce modèle est largement utilisé en SI et il est bien adapté aux outils d'exécution qui sont en complète interaction avec leur environnement.*
- Dans la partie implémentation de cette démarche, une approche par transformation, du niveau conceptuel vers le niveau technique, des concepts liés au comportement, est appliquée afin de générer une architecture d'un outil d'exécution de manière méthodique à partir de la spécification conceptuelle.
    - *Cette approche favorise les critères de maintenabilité, de portabilité et de minimisation de coût.*

1.9. Plan de la thèse

Cette thèse est organisée comme suit :

Le chapitre 2 : Projet exploratoire

Dans ce chapitre, nous présentons projet que nous avons réalisé avec un outil méta-CASE pour explorer la problématique de construction d'un outil d'exécution pour un méta-modèle de processus orienté but. Ce méta-outil a été soigneusement choisi et il est caractérisé par la possibilité de génération de code à partir de modèles spécifiques au domaine. A travers ce projet, nous présentons un premier bilan (**expérimental**) concernant le domaine de construction d'outil et d'expression de la sémantique d'exécution, avec des constats concrets servant à mieux définir la problématique. Aussi, en se basant sur les résultats de ce projet exploratoire, nous définissions un ensemble d'exigences à prendre en compte par la suite.

Le chapitre 3 : Etat de l'art

Ce chapitre fait une synthèse des différentes approches existantes de construction d'outils, en l'occurrence les approches par utilisation des environnements méta-CASE. Dans ce chapitre nous présentons une étude comparative réalisée en se basant sur un ensemble de critères qui se rapportent à (1) la démarche suivie tout au long du processus de construction d'outils d'exécution, (2) au langage de méta-modélisation utilisé, et (3) aux fonctionnalités offertes par ces méta-outils pour spécifier et construire des outils. Nous nous focalisons aussi sur la question de l'expression de la sémantique d'exécution des langages de modélisation de processus ainsi que sur l'exploitation de cette sémantique au niveau des méta-outils. Cet état de l'art permet de constituer un deuxième bilan (**théorique**) permettant de renforcer ce qui a été constaté dans le chapitre précédent.

Le chapitre 4 : Proposition d'une démarche IDM pour la construction d'outils d'exécution

Ce chapitre présente, dans un premier temps, une vue globale de l'approche proposée en introduisant ses composants, et dans un deuxième temps, une vue détaillée de la solution structurée comme suit:

- La spécification structurelle du méta-modèle de processus
- L'élargissement du périmètre du méta-modèle pour prendre en compte la sémantique d'exécution grâce à l'introduction d'une vision événementielle et d'une notation graphique adéquate.
- La transformation de cette spécification étendue afin d'obtenir une architecture orientée objet standard prête à être interprétée par un outil existant de génération de code à partir d'UML.
- L'adaptation de la spécification orientée objet de l'architecture du moteur d'exécution à une plateforme cible.

Le chapitre 5 : Application de la démarche au cas du Map et validation

Dans ce chapitre, notre propos est de chercher à valider notre approche en l'appliquant dans le cas du modèle de processus intentionnel Map. L'objectif est de construire un moteur d'exécution pour ce modèle Map. Ce chapitre met en évidence la faisabilité de l'approche et tente d'évaluer la démarche proposée à l'aide d'une comparaison avec des travaux similaires basée sur les critères et les exigences fixées dans le deuxième chapitre.

Le chapitre 6 : Conclusion et perspectives

Ce chapitre récapitule le travail réalisé en mettant l'accent sur les points forts et ouvre de nouvelles perspectives permettant d'enrichir la contribution de cette thèse.

CHAPITRE 2 : PROJET EXPLORATOIRE

2.1. Introduction

Dans le cadre de notre étude de la problématique de l'expression de la sémantique d'exécution pendant le processus complexe de construction d'outils, nous avons mené un projet exploratoire en utilisant le méta-outil de référence MetaEdit+ (Kelly et al., 1996). La finalité de ce projet est de se servir de la technologie méta-CASE afin de spécifier et de générer un prototype correspondant à un moteur d'exécution pour le modèle de processus orienté but « Map » (ou modèle de la carte).

L'objectif principal de ce chapitre est d'évaluer, en se basant sur ce projet exploratoire, la démarche de construction du moteur d'exécution pour le modèle Map ainsi que la qualité de ce moteur d'exécution. Plus précisément, nous cherchons à répondre à la question suivante : *dans quelle mesure l'outil MetaEdit+ est-il une solution viable et adéquate pour construire un outil d'exécution pour un langage de modélisation de processus orienté but?*

Il est important de noter que les résultats de ce projet permettent d'approfondir notre analyse et notre recherche concernant l'expression de l'exécutabilité des modèles à travers un cas pratique se rapportant au modèle Map. C'est une démarche d'investigation qui exploite la construction d'un système pour mieux comprendre un problème et mieux appréhender la manière de le résoudre. Autrement dit, « *la construction d'un système en soi ne constitue pas la recherche mais elle contribue à faire de la recherche fondamentale puisqu'elle donne la possibilité de faire une vraie synthèse en observant les nouveaux concepts dans un cadre réel et avec un produit tangible* » (Nunamaker, J.F. and Chen, 1990).

2.2. Motivation pour le projet exploratoire

Le projet exploratoire a pour but d'obtenir un premier bilan expérimental et un ensemble de constats concrets concernant la manière d'exprimer la sémantique d'exécution et la manière de l'exploiter afin d'en dériver un outil d'exécution. Deux raisons fondamentales ont motivé ce travail que nous présentons brièvement dans les deux sous-sections suivantes.

2.2.1. Un outil d'exécution pour le Map

Notre première motivation pour ce projet exploratoire était la volonté de développer un outil d'exécution pour un modèle de processus intentionnel particulier qui est le modèle de la carte (ou Map). Le formalisme de la carte est un langage de modélisation orienté intention qui a été développé dans notre laboratoire (Rolland et al., 1999). Il est particulièrement bien adapté pour représenter des processus à haut niveau d'abstraction et à forte variabilité, notamment des processus d'ingénierie. Depuis son introduction dans (Rolland et al., 1999), il a été utilisé avec succès pour décrire la construction de méthode (Ralyté and Roll, 2001) ou

pour la personnalisation d'ERP[24] (Rolland and Prakash, 2000). Pour construire un outil logiciel qui accompagne ce langage, nous capitalisons sur nos expériences antérieures. En effet, notre équipe de recherche a conçu plusieurs outils pour des langages de modélisation orientés événement (Rolland et al., 1988), orientés contexte (Si-Said et al., 1996), et orientés buts (Souveyet et Tawbi, 1998). Etant construits de manière ad-hoc, c.à.d. sans démarche méthodologique précise, nous avons constaté des problèmes récurrents pour maintenir ces outils en cas d'évolution du langage et/ou pour les porter vers de nouveaux environnements en cas d'évolution des plates-formes techniques. Le fait de mieux spécifier la méthodologie de construction de tels outils permet éventuellement de mieux remédier à ce genre de problèmes. Il s'agit de l'un des objectifs du projet exploratoire que nous avons mené.

2.2.2. Exploration de la nouvelle technologie méta-CASE

La deuxième motivation de ce projet est liée à notre intérêt à la technologie méta-CASE et notre curiosité de pouvoir évaluer ses capacités et ses limites dans la construction d'outils d'exécution. Depuis son introduction dans l'ingénierie des systèmes d'information, la technologie méta-CASE a apporté un profond changement à la construction des outils CASE. Ceci n'a pas été véritablement le cas, au paravent, en raison du coût très élevé de la personnalisation de tels outils. Ces méta-outils offrent plusieurs fonctionnalités dont notamment celle de génération de code. Cette fonctionnalité peut être décrite comme une sorte d'interpréteur qui comprend un modèle et notamment sa sémantique d'exécution pour générer le code exécutable qui lui correspond.

L'objectif d'enquêter la technologie méta-CASE est d'expérimenter une approche de développement orientée modèles, et d'évaluer les possibilités de l'exécutabilité du modèle de la carte à travers un exemple concret. Le défi ne consiste pas uniquement à démontrer la faisabilité d'une telle approche, mais à montrer aussi le niveau de qualité et de satisfaction des outils obtenus ainsi que la démarche suivie pour les obtenir et notamment la manière d'exprimer la sémantique d'exécution. De plus, le choix de l'outil méta-CASE (MetaEdit+) doit nous permettre également de vérifier et d'analyser les capacités de cette approche dite « de génération de code » à contourner les problèmes relatifs à la construction des outils de manière ad hoc (i.e., la maintenabilité et la portabilité). A cela s'ajoute le fait qu'aucune approche similaire à la notre n'a été expérimentée.

2.3. Méthodologie de l'expérimentation

La démarche suivie dans l'expérimentation peut se résumer par les points suivants :

- Choix du cas d'application,
- Développement du prototype tout en récoltant des données pertinentes,
- Evaluation du prototype et de la démarche en se basant sur des critères,
- Observation du prototype et le tester,
- Analyse des données recueillies,

[24] *Enterprise Resource Planning*

- Evaluation en se basant sur des critères de qualité,
- Définition d'un ensemble d'exigences,
- Confrontation des exigences par rapport à l'état de l'art (les méta-outils),
- Proposition d'une méta-démarche.

Afin de profiter le plus possible de cette expérimentation concernant la technologie méta-CASE, le choix du méta-outil est essentiel. Pour cela, nous avons réalisé une étude comparative d'un ensemble de méta-outils (tels que Mentor, MetaEdit+, Kogge, Maestro II/Decamerone, etc) tout en s'inspirant d'autres études comparatives nottamment les travaux de (Marttiin et al., 1996), (Marttiin et al., 1993) , (Niknafs and Ramsin, 2008), (VAN and al., 2007), (Isazadeh and Lamb, 1997a) et (Ebert et al., 1997). Cette étude nous a permis de sélectionner l'outil le plus performant, le plus complet et le mieux adapté à notre projet.

Le cas d'étude, CREWS l'Ecritoire, que nous avons appliqué dans l'expérimentation est une méthode d'élicitation de buts et de scénario qui a déjà été outillée dans notre labo (Tawbi, 2001). Aujourd'hui, cet outil ne fonctionne plus et nous essayons via cette expérimentation de suivre une approche plus appropriée pour garantir un outil maintenable et portable. Le choix du même cas d'étude permet de faire plus facilement des comparaisons et d'en tirer des conclusions.

Ce projet a été mené par un groupe de trois personnes : un chercheur expérimenté, un étudiant de niveau Master et moi même. Tous les trois ont une expertise de débutant de MetaEdit+. Le chercheur principal ayant une connaissance d'expert en ingénierie des méthodes s'est chargé de contrôler le déroulement du projet, de recueillir les données et de fournir un soutien tout au long du projet en cas de problèmes spécifiques. Quant à moi, j'ai spécifié les méta-modèles de produit et de processus de la méthode en utilisant l'interface graphique de MetaEdit+, et j'ai fourni une assistance dans l'expression et la validation de la sémantique d'exécution du modèle Map. L'étudiant a conçu le générateur de code en exploitant les méta-modèles et en utilisant le langage de script Merl de MetaEdit+ pour la plate-forme cible en MS Access et VB.net.

Les données ont été recueillies selon un protocole simple : des réunions hebdomadaires durant lesquelles les participants du projet discutaient des difficultés conceptuelles et les problèmes techniques rencontrés puis notifiaient l'état d'avancement ainsi que la durée de chaque tâche. Le projet a duré 6 mois, ce qui équivaut à environ 10 mois/hommes comme charge de travail.

Afin d'évaluer les résultats de cette expérimentation, on propose de s'appuyer sur l'inspection du prototype obtenu et sur l'analyse des données recueillies tout au long du projet. Cette évaluation utilise des critères de qualité élaborés à partir de l'analyse de travaux connexes. A partir de ces premiers résultats, nous pouvons déduire un ensemble d'exigences que doit satisfaire un atelier de construction d'outils d'exécution de modèles de processus.

Ensuite, et pour valider notre analyse, nous allons confronter ces besoins à l'état de l'art actuel des principaux outils de méta-modélisation. Enfin, ceci nous amène à proposer une

méta-démarche de construction d'outils d'exécution. Cette méta-démarche vise à intégrer dans le méta-modèle d'un langage de processus une représentation déclarative et graphique de sa sémantique d'exécution, et de définir ensuite une transformation du méta-modèle conceptuel vers une spécification exécutable selon une approche de type IDM.

2.4. Description du projet exploratoire

2.4.1. L'approche d'évaluation du projet exploratoire

Avant de commencer tout projet, il est important de se fixer la démarche de travail et de choisir l'approche adéquate qu'il faut adopter afin de faciliter l'évaluation. D'après l'étude de la littérature, nous pouvons affirmer qu'il existe différentes approches pour les évaluations empiriques des outils et artefacts logiciels: expérience de laboratoire, étude de cas, enquête, etc. (Wohlin et al., 2003). Le choix de la technique appropriée dépend du contexte de la recherche et des objectifs, et sur les ressources disponibles pour mener de telles recherches. Dans notre cas, en raison de la complexité technique de l'expérience (à savoir l'ingénierie des méthodes et les méta-outils), l'approche d'évaluation la plus appropriée était de faire une expérience unique avec un méta-outil spécifique. Cette approche est de nature empirique car on se concentre sur le système en cours d'utilisation (**system-in-use**), plutôt que le système-en tant-que-tel (**system-as-such**) qui est souvent rencontré dans les études d'évaluation similaires (Cronholm and Goldkuhl, 2003).

En plus d'être une expérience empirique dans laquelle certains résultats seront évalués, notre projet est également considéré comme du développement de systèmes et donc de type prototypage (Nunamaker, J.F. and Chen, 1990). Cette technique d'évaluation nous permet d'acquérir de nouvelles connaissances sur le domaine du problème et la complexité des solutions disponibles, et en apprendre davantage sur les concepts et les modèles à travers le processus de construction du système. Dans notre cas, l'évaluation s'est appuyée sur l'inspection du système obtenu à la fin de l'expérience et sur l'analyse des données recueillies tout au long du processus d'ingénierie. Cette évaluation est basée sur un cadre de qualité qui est synthétisée à partir de l'analyse des travaux connexes.

2.4.2. L'outil MetaEdit+

L'outil que nous avons choisi pour le projet exploratoire est MetaEdit+. Ce choix a été fait parce que cet outil est reconnu par le milieu universitaire et industriel comme étant un outil avec un haut niveau de maturité technique, et en tant que leader dans la technologie de méta-CASE (Niknafs and Ramsin, 2008) (Kelly and Tolvanen, 2008). Cet outil est un environnement puissant et polyvalent proposant simultanément la conception de systèmes et le développement de méthodes. En tant qu'outil CASE, il forme un environnement convivial qui permet, avec une grande flexibilité, la création, la maintenance, la manipulation, la recherche et la représentation d'éléments de conception pour un groupe de développeurs (MetaCASE, 2004). En tant qu'outil méta-CASE, MetaEdit+ propose aussi un environnement convivial pour la spécification, la gestion et la réutilisation de méthodes (MetaCASE, 2004) où il est donc possible de définir des méta-modèles et de les instancier.

Avec MetaEdit+, l'ingénieur méthode définit un langage de modélisation spécifique au domaine utilisant le méta-langage GOPRR et les interfaces de modélisation du Workbench de MetaEdit+. La représentation des modèles est stockée dans un référentiel, et peut être complétée par une spécification dynamique à travers des fonctions de génération de code exprimées sous forme de scripts de procédure à l'aide de Merl (un langage de script fourni par l'outil). Ces spécifications sont ensuite utilisées pour construire un outil cible (une version personnalisée de MetaEdit+), selon le langage de modélisation choisi par l'utilisateur. Cet outil cible pourra contenir des fonctionnalités de génération de code pour transformer n'importe quel modèle en code pour la plate-forme cible qui a été choisi par l'ingénieur de la méthode. Une description plus détaillée de l'outil MetaEdit+ ainsi que du langage de méta-modélisation associé (GOPPR) seront présentées à la section 3.5.6 du chapitre 3.

La Figure 7 présente l'interface graphique initiale de l'outil MetaEdit+ sur laquelle on voit bien les différentes fonctionnalités possibles pour la conception d'un projet donné.

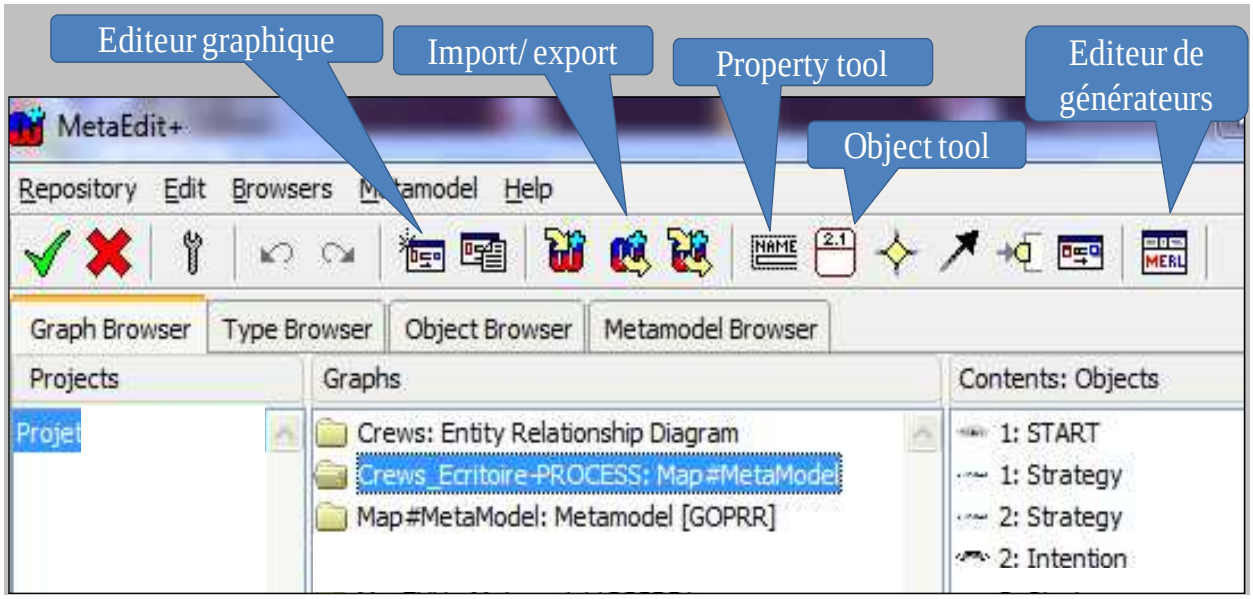


Figure 7. Interface graphique principale de l'outil MetaEdit+

Dans la suite, nous présentons ces fonctionnalités qui sont offertes par MetaEdit+ sous forme d'un ensemble intégré d'outils :

Éditeur graphique : L'éditeur de graphique prend en charge la modélisation de schémas graphiques. Il est possible de réutiliser ces schémas graphiques (les concepts et les liens entre ces concepts) dans d'autres projets. Cet éditeur permet également l'exportation à d'autres outils, comme les navigateurs web et les éditeurs de texte. Il existe différentes formes pour représenter les données dans l'éditeur (tableau, matrice, etc.).

API et import / export de fichiers XML : L'API est une interface qui permet de renforcer l'intégration de l'outil avec d'autres outils. L'API fournit une interface pour lire, créer et mettre à jour les éléments de modèle, ainsi que des contrôles pour le manipuler ou simuler le script. MetaEdit+ utilise les standard SOAP / Web Services /.NET pour l'intégration d'applications, et les fonctions sont accessibles à partir de presque tout les langages de

programmation, plateformes ou environnements. Outre ce lien dynamique en temps réel, les modèles peuvent également être importés ou exportés au format XML.

Navigateur (Browser) : MetaEdit+ offre plusieurs vues de navigateur pour la recherche, le montage et la gestion du contenu du référentiel. Il est possible d'afficher les dessins par projet, type, contenu, représentation, utilisation, hiérarchie de modèle, etc. En plus, il est possible de faire la recherche et le filtrage avec caractères génériques pour trouver des objets.

Outils de développement du langage : En plus des outils conceptuels de méta-modélisation et de l'éditeur graphique, MetaEdit+ fournit également des outils pour gérer les langages de modélisation et leurs composants. Ces outils sont le navigateur de méta-modèle (Metamodel Browser) et le gestionnaire de Type (Type Manager).

Rapport et générateur de code : A part la génération de code, nous pouvons générer pour notre langage de modélisation, des rapports préconstruits pour l'analyse de modèles, la vérification et la documentation dans les formats Word, RTF et HTML.

Éditeur d'objets : Lorsqu'on souhaite créer un nouveau modèle, on a besoin tout d'abord de créer les objets en utilisant la fonctionnalité adéquate à savoir 'Object tool'. Cet outil donne la possibilité de définir un objet par son nom, ses différents propriétés, ses contraintes, de l'affecter à un projet spécifique et de le stocker dans un référentiel d'objet pour pouvoir le réutiliser plus tard dans d'autres projets.

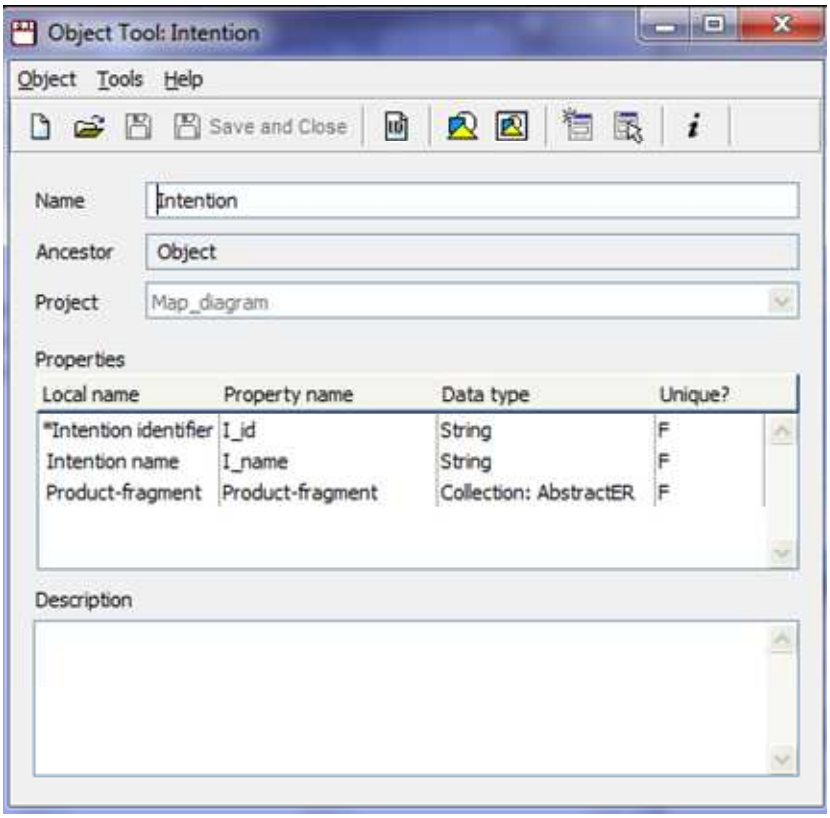


Figure 8. L'interface de l'éditeur d'objet dans MetaEdit+

La Figure 8 présente un exemple de la création d'un objet *Intention* grâce à cet éditeur.

2.4.3. Le modèle de processus : Map

Le *Map* est un formalisme qui permet de représenter des modèles de processus sous forme de carte. Il est construit sur deux concepts de base: *l'Intention* et la *Stratégie*. Une intention est un but à atteindre, elle peut être atteinte par plusieurs stratégies. Une stratégie est une manière d'atteindre une intention. Un autre concept dérivé de la carte est celui de la section qui est un triplet composé d'une intention source, une intention cible et une stratégie (par exemple <*Spécifier une Classe*, *Manuellement*, *Spécifier une Association*> de la Figure 9). Deux catégories de sections sont définies: une *section atomique*, dans laquelle la stratégie est opérationnelle car elle est associée à un code exécutable (une procédure, un composant, un service, etc) ; une *section non-atomique* correspond à une stratégie complexe de réalisation de l'intention nécessitant d'être affinée en une autre carte. Grâce à ce mécanisme d'affinement, le concepteur peut exprimer un processus à différents niveaux d'abstraction comme une hiérarchie de cartes.

Le Map a été présenté dans (Rolland et al., 1999) comme étant un ordonnancement non figé d'intentions et de stratégies. Toute carte comporte en particulier une intention « *Démarrer* » et une intention « Arrêter » (parfois nommée « *Terminer* »), et représente un ensemble de chemins allant de « *Démarrer* » à « *Arrêter* ». Chacun de ces chemins est à lui seul un modèle de processus. C'est pourquoi la carte est un multi-modèle. La représentation graphique d'une instance du modèle Map est composée de nœuds (les intentions) et d'arcs orientés (les stratégies) reliant les nœuds. Chaque instance du modèle Map est appelée une carte (ou un Map).

Pour illustrer un processus sous forme d'une carte, la Figure 9 représente un extrait d'une carte spécifiant une démarche de conception d'un schéma de classes (Edme, 2005). Cet extrait de carte est constitué de quatre intentions, et de huit stratégies permettant de réaliser chacune d'entre elles.

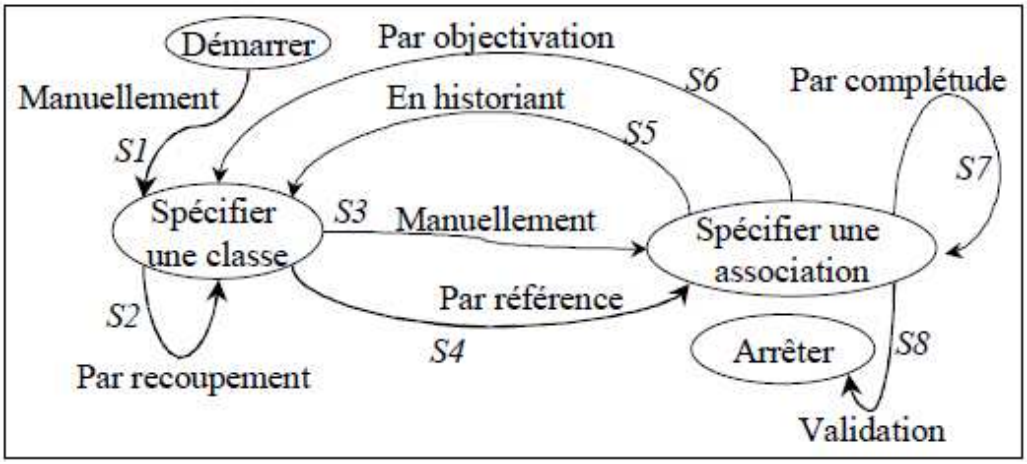


Figure 9. Extrait d'une carte de conception d'un schéma de classes

L'utilisation du modèle Map est fortement recommandée aussi bien en ingénierie des systèmes qu'en ingénierie des méthodes pour représenter des processus intentionnels sous forme de graphe d'intentions et de stratégies. En effet, les processus d'ingénierie sont généralement orientés *but*, ce qui fait qu'à chaque instant, l'ingénieur SI ou l'ingénieur

méthode est confronté à des objectifs qu'il doit atteindre sans connaitre pour autant la façon de procéder pour les réaliser. Les différentes stratégies donnent une grande flexibilité pour guider l'ingénieur et le faire avancer dans le processus d'ingénierie.

La Figure 10 représente le schéma classique du méta-modèle Map comme présenté dans (Rolland and Prakash, 2000). Ce méta-modèle représente uniquement l'aspect statique du modèle Map et ne donne aucune information sur son comportement. Mais la question qu'on se pose est : **pourquoi a-t-on besoin de méta-modéliser le comportement du Map?**

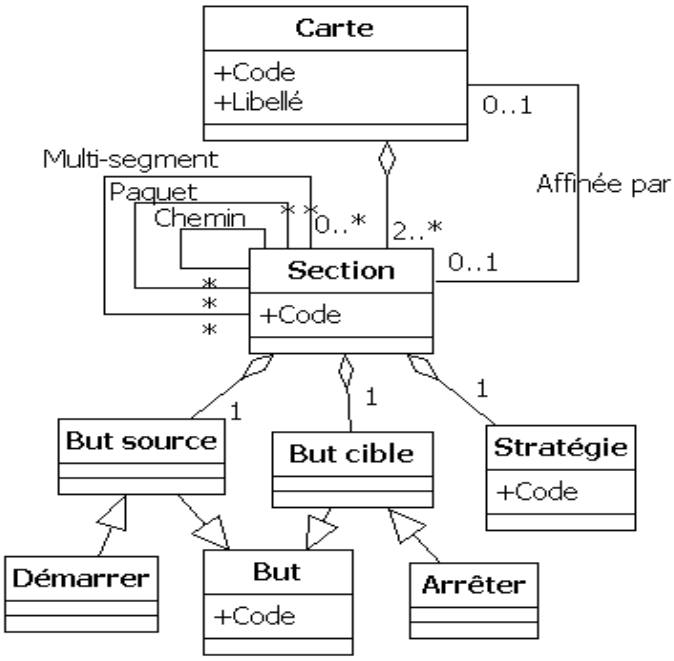


Figure 10. Le méta-modèle du Map

La réponse à cette question est fortement liée à l'usage que l'on fait de ce méta-modèle. En effet, cette question ne se pose pas pour un usage contemplatif du modèle Map. En revanche, lorsqu'on souhaite exécuter ou simuler un processus conforme au modèle Map et que cette exécution/ simulation soit réalisée à travers un outil, cela va nécessiter un outil qui interprète le comportement du Map. Pour cela, on a besoin de représenter ce comportement du Map au niveau du méta modèle afin qu'il soit implémenté dans l'outil d'exécution/simulation. Un tel comportement précise la manière opératoire du modèle lors de son exécution et c'est ce qu'on appelle la sémantique opérationnelle du modèle que nous allons exprimer dans la section suivante.

2.4.4. La sémantique opérationnelle du modèle Map

La sémantique opérationnelle du modèle intentionnel Map ne correspond pas à un processus séquentiel dans lequel des tâches se succèdent (comme c'est le cas d'une machine à états finis : FSM) ; c'est plutôt une navigation dans un graphe suivant les intentions de l'utilisateur et selon le contexte situationnel par rapport à l'état du produit. Ceci autorise une grande variabilité dans l'ordonnancement de réalisation des intentions et dans le choix des stratégies à appliquer. Afin de comprendre la sémantique opérationnelle du modèle Map, nous nous basons sur l'exemple de carte suivante :

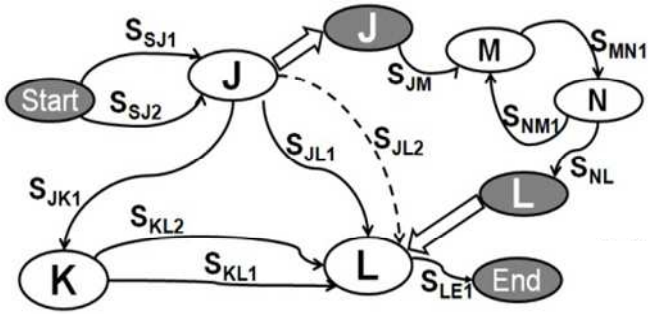


Figure 11. Un exemple d'une carte Map

Par exemple, la Figure 11 montre qu'à partir de l'intention J (une fois l'intention J a été réalisée), il y a deux chemins possibles pour progresser: ou bien atteindre l'intention K avec la stratégie S_{JK1}, ou bien réaliser l'intention L soit par la stratégie S_{JL1} ou encore la stratégie S_{JL2}. Dans certain cas, il est nécessaire de définir des contraintes de précédence et/ou d'exclusion entre les sections afin de restreindre l'ordre dans lequel les sections sont exécutées, ou pour exprimer des relations d'exclusivité entre les alternatives. Toutes ces alternatives et sections raffinées offrent de multiples possibilités pour la réalisation des intentions, et offrent par la suite plus de flexibilité dans la représentation et la performance des processus.

Pour l'exprimer plus précisément, la sémantique d'exécution d'une carte a besoin d'avoir des références vers le produit. En effet, la réalisation d'une intention correspond à l'exécution d'une action sur une instance d'un fragment d'un produit dans le cadre d'une stratégie spécifique (Nurcan and Edme, 2005) (Assar et al., 2000). En exécutant une section <J_i, S, L>, nous entendons atteindre l'intention L, en utilisant la stratégie S, et à partir de J_i qui est une réalisation antérieure de l'intention J (identifiée à l'aide d'un index i). Cela signifie que la même section dans une carte peut être exécutée pour différentes instances d'un même fragment de produit.

Si J_{i1}, J_{i2} et J_{i3} sont les réalisations passées de l'intention J avec différentes instances d'un même fragment de produit (comme le montre la Figure 12), alors les sections <J, S_{JK1}, K>, <J,S_{JL1}, L> ou <J, S_{JL2}, L> peuvent être exécutées pour chacune de ces réalisations passées, et le résultat sera les réalisations d'intention K_{i1}, K_{i2} et K_{i3} pour l'intention K (ou L_{i1}, L_{i2} et L_{i3} pour l'intention L).

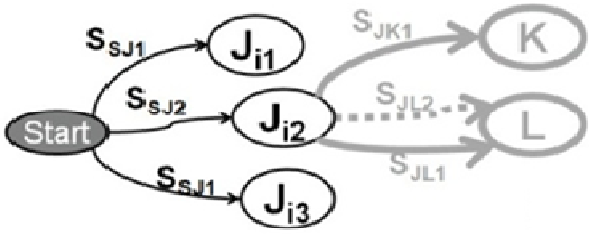


Figure 12. Une illustration des sections candidates

La combinaison d'une réalisation d'intention, une stratégie et une intention (exemple le triplet <J_{i1}, S, L>) est appelée une ***section candidate*** et c'est un concept fondamental pour exprimer la sémantique opérationnelle. A chaque exécution d'une section, une nouvelle collection de sections candidates (c'est à dire des sections qui pourraient être exécutées dans l'étape suivante) doit être calculée. Compte tenu d'un certain état du produit (c'est à dire

l'ensemble des instances du produit) et de l'historique des réalisations d'intentions, le calcul de l'ensemble des sections candidates se fait en comptant les sections qui correspondent aux chemins possibles et en vérifiant les contraintes. Comme cet ensemble se développe rapidement, différentes stratégies de progression peuvent être définies pour le réduire (Edme, 2005).

La sémantique opérationnelle peut être facile à formaliser et à exprimer pour un modèle simple comme le modèle SimplePDL ou encore le modèle de machine à états finis qui peuvent se présenter sous forme d'un réseaux de Pétri (Combemale, 2008). En revanche, ce n'est toujours pas le cas. En effet, la sémantique des modèles orientés *buts* est ambigüe et difficile à exprimer formellement. Dans ce contexte, nous sommes amenés à réfléchir sur la possibilité de pouvoir dériver de manière systématique un outil d'exécution à partir d'un modèle particulier parmi ces modèles.

2.4.5. Le cas d'application : La méthode CREWS l'Ecritoire

CREWS L'Ecritoire est une approche semi-automatique de découverte et de documentation des besoins (CREWS, 1999). Le processus de l'approche exploite la relation entre les deux concepts du *but* et de *scénario* selon un mouvement bidirectionnel consistant en deux activités principales: la découverte des buts et l'écriture des scénarios. Les scénarios sont écrits en langage naturel, l'écriture d'un scénario est une activité complexe où l'on peut distinguer trois sous-activités: Ecrire le scénario, vérifier et compléter les actions du scénario à l'aide de patrons sémantiques, et conceptualiser le scénario. Une fois le scénario est conceptualisé, l'approche propose d'exploiter les actions de celui-ci afin de découvrir de nouveaux buts. Ces deux activités sont répétées itérativement pour remplir progressivement la hiérarchie des Fragments de Besoin (ou Requirement Chunk) qui représente le modèle de produit.

Nous avons choisi le processus d'ingénierie des exigences de CREWS L'Ecritoire pour deux raisons: (i) le processus est moyennement complexe (Rolland et al., 1998), et (ii) un prototype avait déjà été mis au point pour ce processus (Souveyet and Tawbi, 1998) et utilisé pour expérimenter les directives fournies à susciter exigences (Si-Said and Rolland, 1998).

2.4.5.1. Le Map pour modéliser le processus de CREWS L'Ecritoire

Le processus de découverte des Fragments de Besoin peut être présenté sous forme d'une carte Map comme le montre la Figure 13 (Hug et al., 2012). Cette carte a été légèrement simplifiée (en éliminant trois stratégies) afin de faciliter son implémentation dans notre projet.

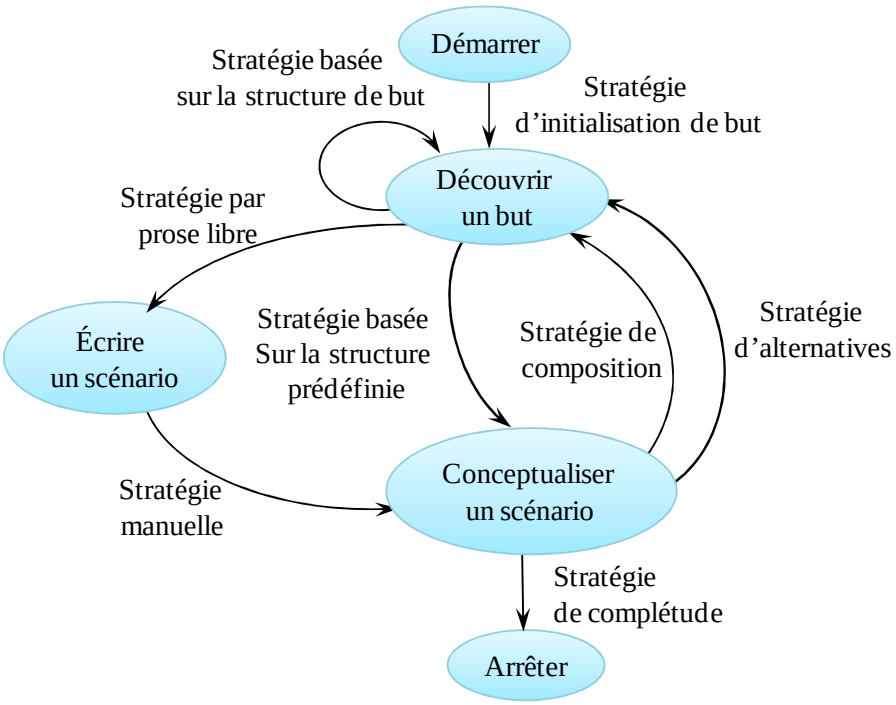


Figure 13. Map de CREWS l'Ecritoire

La carte de CREWS L'Ecritoire de la Figure 13 présente huit sections composées à partir de cinq intentions et huit stratégies. Les cinq intentions de la carte de CREWS L'Ecritoire sont présentées dans le Tableau 1 :

Les intentions de la carte de L'Ecritoire
Démarrer
Découvrir un but
Écrire un scénario
Conceptualiser un scénario
Arrêter

Tableau 1. Les intentions de la carte de CREWS L'Ecritoire

Les huit stratégies sont énumérées dans le Tableau 2:

Les stratégies de la carte de L'Ecritoire
Stratégie d'initialisation de but
Stratégie basée sur la structure de but
Stratégie par prose libre
Stratégie basée sur la structure prédéfinie
Stratégie manuelle
Stratégie de composition
Stratégie d'alternatives
Stratégie de complétude

Tableau 2. Les stratégies de la carte de CREWS L'Ecritoire

Les huit sections sont définies dans le Tableau 3:

Les sections de la carte de L'Ecritoire
< *Démarrer, Stratégie d'initialisation de but, Découvrir un but*> *:* Elle permet de découvrir un but initial.
< *Découvrir un but, Stratégie basée sur la structure de but, Découvrir un but* > *:* Elle permet de découvrir un but basée sur la structure de but (verbe, cible, manière …)
< *Découvrir un but, Stratégie par prose libre, Écrire un scénario* > *:* Elle permet d'écrire un scenario pour un but donnée de manière libre.
< *Découvrir un but, Stratégie basée sur la structure prédéfinie, Conceptualiser un scénario* > *:* Elle permet de conceptualiser un scenario pour un but donnée en utilisant une structure prédéfinie.
< *Écrire un scénario, Stratégie manuelle, Conceptualiser un scénario* > *:* Elle permet de conceptualiser un scenario pour un but donnée de manière manuelle.
< *Conceptualiser un scénario, Stratégie de composition, Découvrir un but* > *:* Elle permet de découvrir un nouveau but en utilisant le paramètre « ET » sur les fragments de produits.
< *Conceptualiser un scénario, Stratégie d'alternatives, Découvrir un but* > : Elle permet de découvrir un nouveau but en utilisant le paramètre « OU » sur les fragments de produits.
< *Conceptualiser un scénario, Stratégie de complétude, Arrêter* > *:* Elle permet de mettre fin au processus en complétant le produit.

Tableau 3. Les sections de la carte de CREWS L'Ecritoire

Comme les travaux antérieurs qui ont enquêté sur l'exécutabilité du modèle de la carte et sur la formalisation de sa sémantique opérationnelle (Edme, 2005) (Nurcan and Edme, 2005) et (Souveyet and Tawbi, 1998) ont utilisé le langage de programmation Visual Basic et la base de données relationnel MS Access, nous avons fait un choix similaire pour la plate-forme cible pour la génération de code. Ce choix permet de mieux comparer les deux travaux qui sont développés dans des conditions similaires.

2.4.5.2. Le concept de Fragment de Besoin

Pour documenter les besoins d'un système sous forme de buts et de scénarios, le concept du Fragment de Besoin (FB) est introduit. Un FB est un couple <*but, scénario*>. Étant donné qu'un but est intentionnel et qu'un scénario est opérationnel, on définit un FB comme étant « *une manière de réaliser un but au moyen d'un scénario* ».

Un *but* est défini comme « un objectif à réaliser en utilisant le futur système » (Plihon et al., 1998). Un *scénario* est défini comme « un comportement possible limité à un ensemble d'interactions entre plusieurs agents ».

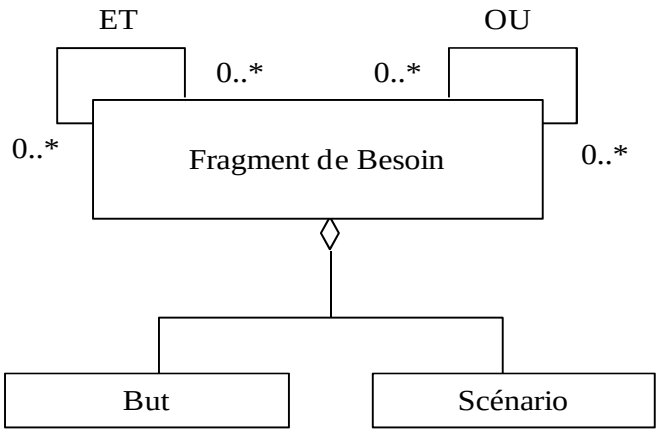


Figure 14. La structure d'un fragment de besoin

Les FB sont assemblés selon deux types de relations nommés c*omposition et alternative*. Les deux premières relations sont représentées par une structure horizontale reliant les FBs par des liens appelés *'ET'* et *'OU'*.

2.4.5.3. L'exécution d'un processus selon une carte Map

L'exécution d'un processus est une démarche d'instanciation simultanée d'un modèle structurel de produit et d'une carte Map. La Figure 15 rappelle les deux niveaux inférieurs d'abstraction de la structure d'un produit et d'un processus. On y distingue les deux univers de la modélisation et de l'exécution. La partie droite de la figure décrit l'instanciation d'une carte Map en un processus. Il y a une double relation entre une carte Map et un processus exécuté selon cette carte. D'une part, le processus est une instance d'une carte Map représentant la chronologie des sections qui ont été exécutées pour élaborer le produit. D'autre part, ce processus est guidé selon cette même carte car le principe d'exécution de la carte consiste à calculer et à adapter les sections qui peuvent être exécutées à la prochaine étape de ce processus.

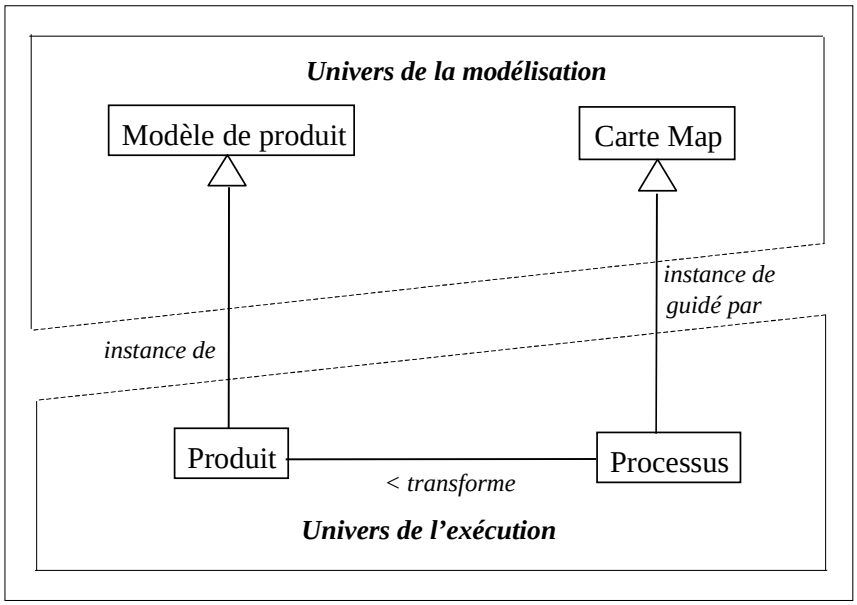


Figure 15. Les deux niveaux d'abstraction de la structure du produit et d'un processus (Edme, 2005)

Au cours de l'exécution du processus de la carte, chaque intention peut y être instanciée plusieurs fois. A chacune de ces instanciations d'intention, une nouvelle *instance d'intention* est créée au sein du processus. Chacune de ces instances manifeste que l'objectif représenté par l'intention a été atteint. Pour ce faire, une des stratégies proposée par la carte a été utilisée. Une *réalisation d'intention* est une instance d'intention qui marque une transformation du produit. Chaque exécution d'une stratégie marque le procédé par lequel une réalisation d'intention a été atteinte. Le processus obtenu par l'instanciation d'une carte Map est *l'historique* des intentions atteintes et des stratégies utilisées pour réaliser chacune des intentions. Il exprime le chemin parcouru pour aboutir à l'état actuel du produit.

La *section candidate* est un concept clé dans le processus d'exécution d'une Carte Map. A chaque étape du processus, les sections candidates offrent un guidage en déterminant toutes les possibilités de progresser dans exécution. Comme son nom l'indique, l'exécution d'une section candidate n'est pas imposée puisqu'elle ne représente qu'une proposition que l'ingénieur d'application a le choix d'exécuter ou non. C'est par les candidates que chaque carte Map assure un guidage des processus qui sont des instances de cette carte. A chaque nouvelle réalisation d'intention, un algorithme de calcul de sections candidates est mis à jour pour donner la nouvelle liste qui est généralement plus grande. On comprend alors que la quantité des sections-candidates s'accroît considérablement plus vite que celle des réalisations d'intentions. La multiplicité des réalisations d'intentions d'un processus entraîne celle des candidates proposées. C'est ainsi qu'apparaît l'apport du formalisme de Map.

2.4.5.4. La relation dynamique entre le produit construit et le processus exécuté

Parallèlement à l'instanciation d'une carte Map en un processus, le modèle de produit est instancié en un produit. Un produit construit est le fruit d'un processus exécuté. Les différentes parties du produit construit sont identifiées au sein du processus par les réalisations d'intentions. Cette connaissance est acquise dynamiquement au cours de l'exécution de la carte. La relation dynamique entre produit et processus correspond à celle qui associe les parties du produit construit et les réalisations d'intentions.

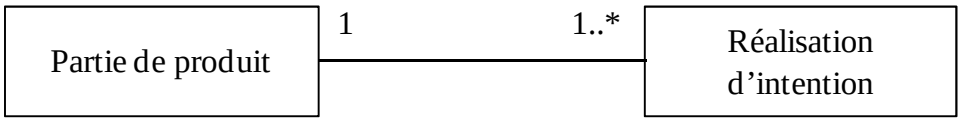


Figure 16. Relation entre parties du produit construit et réalisations d'intentions.

La Figure 16 schématise l'existence de cette relation dynamique entre les parties du produit construit et les réalisations d'intentions. On observe à travers cette relation qu'une réalisation d'intention ne représente, au sein du processus exécuté, qu'une unique partie du produit construit. Réciproquement, toute partie de produit doit être représentée au sein du processus pour toutes les intentions qu'elle a réalisées. Par exemple, chaque réalisation d'une intention RI signale que l'unique partie du produit qui lui correspond, $P_{i,}$ a réalisé I.

Les parties du produit et les réalisations d'intentions du processus sont obtenues au cours de l'exécution du processus. La relation décrite ici est dynamique car l'association entre le produit et le processus est définie au cours de l'exécution du processus. Il est important de noter que l'historique du produit ainsi que les réalisations d'intention forment la *trace* qui est

une connaissance indispensable pour calculer les sections candidates et avancer dans le processus d'exécution de la carte.

2.4.6. Le développement du prototype

Nous rappelons que l'objectif principal de ce projet est l'exploration de la technologie Méta-CASE à travers l'outil MetaEdit+ afin d'évaluer ses possibilités à exprimer la sémantique opérationnelle d'un modèle de processus intentionnel et à construire un outil d'exécution correspondant. Les motivations pour ce projet ont été mentionnées en début de ce chapitre, mais nous pouvons ajouter une autre : c'est le fait qu'aucune approche similaire à la notre n'a été expérimentée. En effet, le choix de l'outil MetaEdit+ doit nous permettre de vérifier et d'analyser les capacités de l'approche dite « par génération de code ». Jusqu'à présent, seul « Nokia », géant de la téléphonie mobile, affirme avoir modélisé son système d'information sur MetaEdit+. Malheureusement pour le monde informatique, ces résultats ne sont pas accessibles ce qui nous offre aucune possibilité d'évaluation.

2.4.6.1. L'architecture globale du projet exploratoire

Avant d'entrer plus dans les détails de la mise en œuvre du projet, nous proposons une vue d'ensemble de la démarche suivie dans cette expérimentation schématisée à la Figure 17.

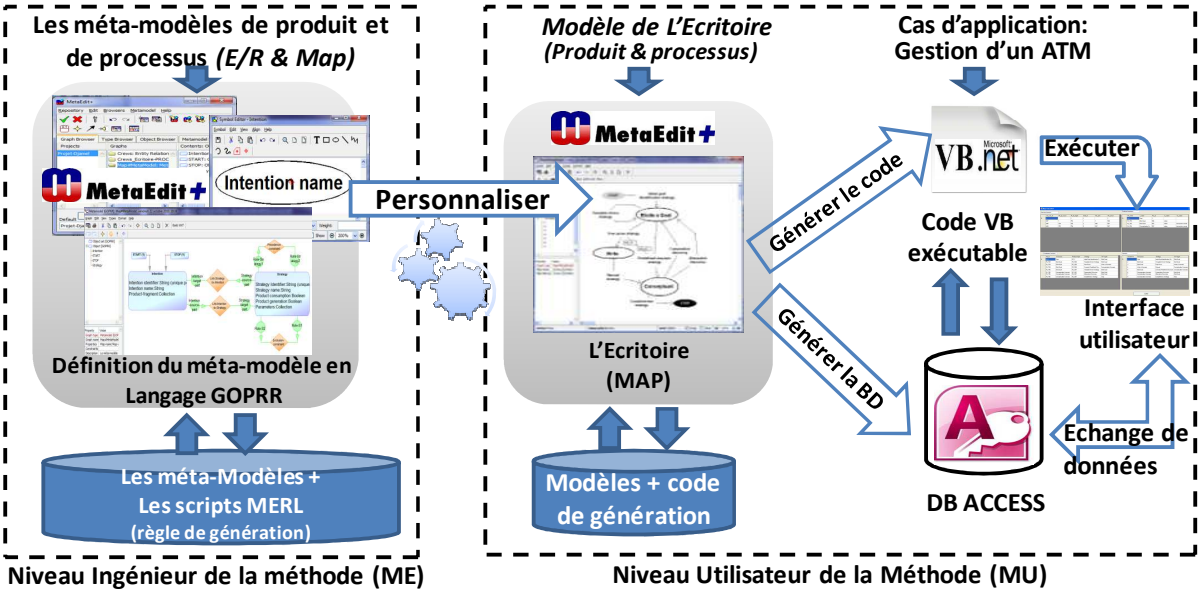


Figure 17. Aperçu de la méta-démarche suivie dans le projet avec MetaEdit+

Le point de départ de la démarche est effectué par l'ingénieur méthode qui définit les méta- modèles (de produit et de processus) à l'aide de ce qu'on appelle MetaEdit+ Workbench ainsi que les concepts de méta-modélisation fournis par le langage GOPRR. Aussi, lors de cette étape, l'ingénieur précise les règles de génération avec le langage de script Merl. A la fin de cette première étape, une version personnalisée de MetaEdit+ est automatiquement construite et elle représente l'outil CASE qui va nous aider à construire l'outil d'exécution pour le modèle Map. Cet outil CASE comprend des fonctionnalités de génération de code. L'utilisateur de la méthode peut alors utiliser cet outil pour définir un modèle de produit et un

modèle de processus, et génère le moteur d'exécution qui sera implémenté sur la plate-forme cible conformément au code script initialement écrit par l'ingénieur méthode.

2.4.6.2. L'implémentation de l'outil d'exécution du Map

Dans cette partie nous allons expliquer comment fonctionne notre outil d'exécution du Map. La première étape de la réalisation de cet outil est la mise en place de la base de données. Afin de respecter l'objectif principal de ce travail, à savoir mettre en place une architecture générique pour l'exécution d'un modèle d'ingénierie orienté *but*, la base de données a été crée à partir de MetaEdit+. Le langage de script MERL nous a permis de reprendre la structure du méta-modèle Map, et de la compléter afin de générer automatiquement une base de données qui soit relatives au modèle de la carte. Nous présentons, dans la section suivante, les scripts de création de cette base de données (en langage textuelle) utilisée par la suite dans l'outil d'exécution.

La construction de l'outil d'exécution nécessite la connaissance de l'historique du processus exécuté que nous appelons *Trace*. En effet, afin de calculer les sections candidates puis les proposer à l'utilisateur, l'outil d'exécution va devoir analyser d'abord l'ensemble des exécutions réalisées. La construction de l'outil d'exécution nécessite aussi l'existence d'un *référentiel* de carte Map qui apporte la connaissance sur la carte Map dont l'outil doit assurer l'exécution. Ces informations sont nécessaires durant toute la durée d'exécution d'une carte. Ainsi, le référentiel des cartes Map et la Trace du processus sont exploités systématiquement à chaque étape du processus pour mettre à jour l'ensemble des sections candidates.

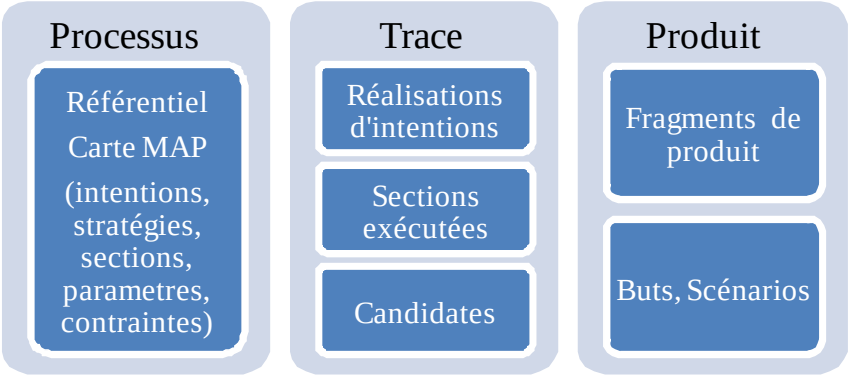


Figure 18. Représentation de l'architecture de l'outil d'exécution

La Figure 18 présente les composants de l'architecture nécessaire dans un environnement de développement tels qu'un outil CASE (personnalisé) pour pouvoir construire un outil d'exécution pour le Map.

2.4.6.3. Les Scripts de création de la Base de données Map

Le script de création de la base de données a été réalisé en plusieurs parties. Chaque partie générée est ajoutée à la précédente pour former l'ensemble du script. La génération du script en plusieurs parties était, à notre avis, la meilleure manière pour garantir un script qui respecte notre architecture en trois composants. Cela apporte également plus de lisibilité et une meilleure clarté du code et de l'architecture qu'une génération en une seule partie.

Au total, nous avons définit cinq scripts qui sont :

- Script1_Map_Process
- Script2_Map_Product
- Script3_Map_Trace
- Script4_Map_Links
- Script5_Map_Init

Dans ce qui suit, nous détaillons ces scripts afin de comprendre comment la base de données est créée, mais également afin de montrer le coté générique de l'outil d'exécution.

Script1_Map_Process

Le premier script a pour fonction de créer la base de données, les tables relatives au processus ainsi que des variables et constantes nécessaires pour les fonctions de créations. (L'ensemble des fonctions de ce script sont détaillés à l'**Annexe 1**). Parmi les fonctions de ce script, on cite à titre d'exemple la fonction *Sub Create().* Cette dernière est une procédure de création de la base de données au sein de laquelle nous définissons les tables de la base de données à savoir : Intentions, Strategies, Sections, Parameters et Map_constraints. Ci-après, un extrait de la table *Intentions* qui est définie par un identifiant "I_id" et un nom "I_name":

```
''' ========================================================= ' newline
'''   Table : INTENTIONS ' newline
''' ========================================================= ' newline
'    If AccCanCreateTab("Intentions") = IDYES Then ' newline
'       AccCreateTable "Intentions" ' newline
'          AccAddColumn "I_id", "counter", 0, 0, "YES", "", "", "", 1 ' newline
'          AccAddColumn "I_name", "Text(30)", 30, 0, "YES", "", "", "", 2 ' newline
'       AccEndTable ' newline
'    End If ' newline
'    AccAddColProp "Intentions", "I_id", "Caption", DB_TEXT, "I_id" ' newline
'    AccAddColProp "Intentions", "I_name", "Caption", DB_TEXT, "I_name" ' newline
 ''' ========================================================= ' newline
'''   Index : INTENTIONS_PK ' newline
''' ========================================================= ' newline
'    AccCreateIndex "primarykey", "unique", "", "Intentions_PK", "Intentions", "I_id" '
```

Script2_Map_Product

Le second script a pour fonction de créer les tables du composant produit de l'architecture ainsi que les indexs. Il s'agit des tables suivantes:

- *RCs* : la table des Fragments de produit
- *RC_Hierarchies* : la table de l'hiérarchie des Fragments de produit
- *Goals* : la table des buts
- *Scenarios* : la table des scenarios

Script3_Map_Trace

Le troisième script a pour fonction de créer les tables du niveau de la Trace (2ème composant de l'architecture de l'outil de la Figure 18), ainsi que leur index. Ces tables sont:

- Candidates_Sections

- Executed_Sections
- Realized_Intention

Script4_Map_Links

Le quatrième script a pour fonction de créer les jointures entre toutes les tables. L'ensemble des jointures est présenté à l'**Annexe 2**. Prenons-en juste un exemple:

```
AccCreateReference "I_CANDI_SOURCE", "Intentions", "Candidates_Sections",  "",  ""
```

Cet exemple permet la création d'une relation qui s'appelle "I_CANDI_SOURCE", entre les tables "Intentions" et "Candidates_Sections". Au sein de cette fonction nous appelons la fonction : « *AccAddRefrCol "Strat_id_candi", "Strat_id", "Strat_id"* » qui nous permet de créer une jointure qui s'appelle "Strat_id_candi" entre les colonnes " Strat_id" de la table "Intentions" et "Strat_id" de la tables "Candidates_Sections".

Script5_Map_Init

Le cinquième script a pour fonction d'initialiser notre base de données de manière générique en se basant sur le modèle défini. Ainsi, il est toujours possible d'obtenir une base de données conforme au modèle implémenté sans avoir à changer le code du script. Par exemple, si nous décidons d'ajouter une stratégie entre l'intention « Découvrir un but » et « Écrire un scénario » ; lors de la génération du code, cette stratégie sera parfaitement intégrée à la base. Dans ce cas, l'ajout des données se fait comme suit (extrait de l'**Annexe 3**) :

```
'Dim sql As String ' newline
foreach .START
{
'sql = "insert into [Intentions] (I_name) VALUES ('':Intention name;'')" ' newline
'dtbs.Execute (sql) ' newline
}
foreach .Intention
{
'sql = "insert into [Intentions] (I_name) VALUES ('':Intention name;'')" ' newline
'dtbs.Execute (sql) ' newline
}
foreach .STOP
{
'sql = "insert into [Intentions] (I_name) VALUES ('':Intention name;'')" ' newline
'dtbs.Execute (sql) ' newline
}
```

Le mot clé **Foreach .Start**{} permet de parcourir l'ensemble des valeurs de l'objet *Start*. Pour chaque objet *Start*, nous exécutons une requête SQL (en langage MERL) qui permet d'insérer dans la table « Intentions » les données de l'attribut « I_name ». La valeur de cette donnée est contenue dans la variable « : Intention name; », qui sera automatiquement chargée depuis l'interprétation graphique du modèle implémenté. Nous procédons selon le même principe pour la table « Strategies ». Il est important de vérifier la cohérence des types de valeurs entre ce que génère MetaEdit+, et ce qu'accepte notre SGBD Access. C'est pourquoi

avant de procéder à l'exécution de la requête *insert*, nous remplaçons les valeurs booléennes « T » par « -1 » et « F » par « 0 ». Pour terminer, nous ajoutons les sections et les contraintes du modèle. (**Annexe 4**, procédure foreach .Intention)

Une fois le code script est complet, nous pouvons désormais l'exécuter. Pour cela, nous utilisons un module VBa à partir d'Access. Une fois exécuté, nous obtenons la base de données avec toutes les tables nécessaires à la construction de l'outil d'exécution. La Figure 19 présente les différentes tables de notre base de données ainsi que les liens entre eux:

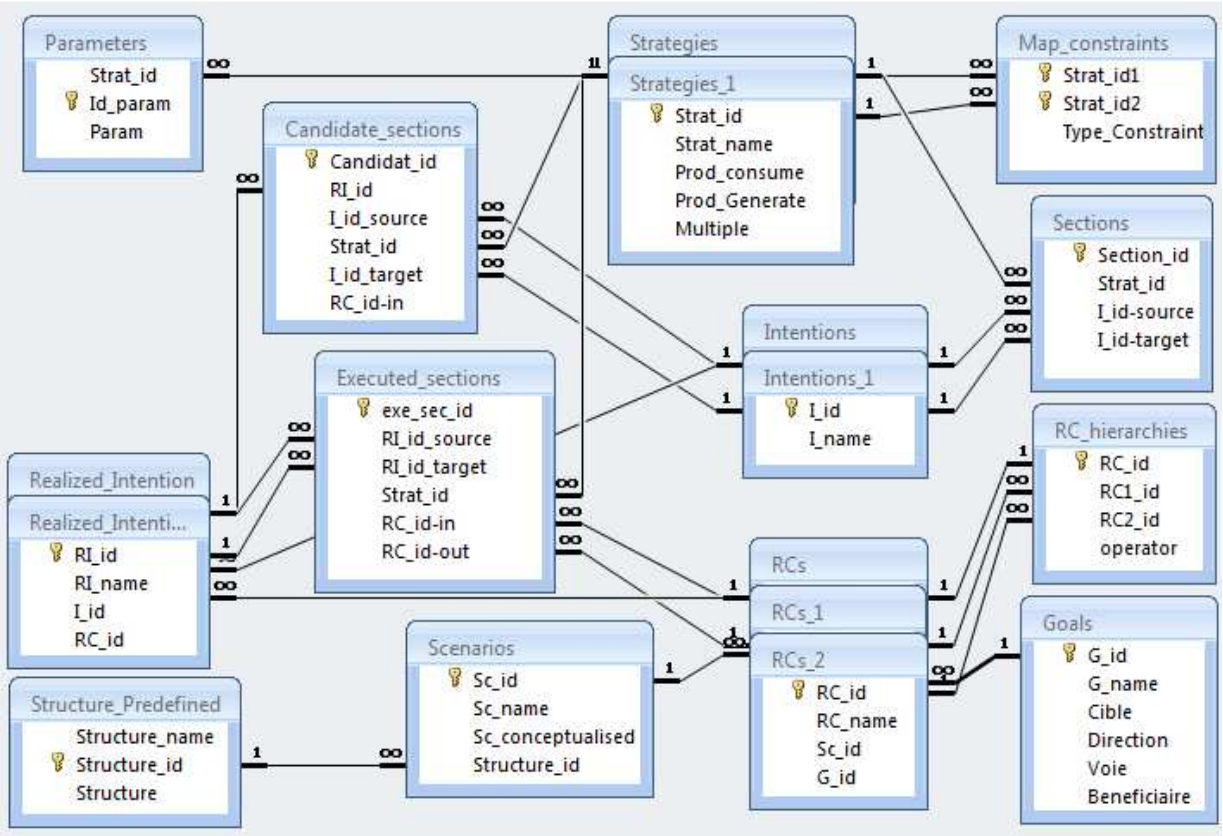


Figure 19. Base de données générée pour notre moteur d'exécution

Le tableau suivant (Tableau 4) récapitule la structure de la base de données en termes de tables, et pour chaque table, nous présentons l'ensemble de ses attributs ainsi que leurs types.

Nom de la table	Attribut	Type
Intentions	I_id I_name	Auto incrément Texte
Sections	Section_id I_id_source, I_id_target, Strat_id	Auto incrément Numérique
Strategies	Strat_id Strat_name Prod_consume, Prod_generate, Multiple	Auto incrément Texte Booléen
Map_constraints	Strat_id1, Strat_id2 Type_Constraint	Numérique Texte

Parameters	Id_param Param Strat_id	Auto incrément Texte Numérique
Candidate_sections	Candidat_id RI_id, I_id_source, Strat_id, I_id_target, RC_id-in	Auto incrément Numérique Numérique
Realized_Intention	RI_id RI_name I_id, RC_id	Auto incrément Texte Numérique
Executed_sections	Exe_sec_id RI_id_source, RI_id_target, Strat_id, RC_id-in, RC_id-out	Auto incrément Numérique Numérique
RCs	RC_id RC_name Sc_id, G_id	Auto incrément Texte Numérique
RC_hierarchies	RC_id RC1_id, RC2_id operator	Numérique Numérique Texte
Scenarios	Sc_id Sc_name, Sc_conceptualised Structure_id	Auto incrément Texte Numérique
Goals	G_id G_name, Cible, Direction, Voie, Beneficiaire	Auto incrément Texte

Tableau 4. Récapitulatif des tables de la Base de données

2.4.6.4. L'algorithme d'exécution du moteur Map

L'outil d'exécution est l'objectif final de notre projet. Nous exigeons que cet outil soit le plus générique possible pour ne pas refaire un moteur à chaque modification du modèle Map. Cette généricité doit être prise en compte au moment de la définition de l'algorithme d'exécution du moteur relatif au modèle de processus Map. Cette exigence contribue ainsi à la complexité de notre mission sachant que le modèle de processus est fortement lié au modèle de produit et qu'à chaque étape de l'exécution, le processus a besoin du produit pour effectuer l'exécution. D'ailleurs, il est important de rappeler qu'une section candidate est un couple <section, fragment de produit> c'est à dire que la section s'exécute sur ce fragment de produit.

Dans ce qui suit, nous présentons l'algorithme d'exécution du moteur de manière synthétique en langage naturel. Cet algorithme est composé de 3 parties : l'initialisation de la base de données au moment du chargement du moteur, l'algorithme de calcul des sections-candidates et enfin, l'algorithme d'exécution d'une section candidate.

Au chargement du moteur :

Mise à jour de la base de données => cette étape de mise à jour de la base de données consiste à supprimer les enregistrements existants de toutes les tables sauf la 1ère table qui correspond à l'initialisation.

1. On charge les tableaux avec les données de la base => Execute Sections, Realized Intentions, Sections, Candidates Sections.
2. On calcul les sections candidates :

a. On supprime toutes les candidates précédentes.

b. On vérifie si la section choisie est celle qui met fin au programme

Si oui, alors programme terminé

Si non, alors on calcul les sections candidates

Algorithme du calcul des candidates (détaillé à l'Annexe 5):

On récupère l'ensemble des « RI_name » et « RI_id » de la table Realized_Intentions

Pour chaque « RI_id »

On récupère l'ensemble des « I_id » et « I_name » de la table Intentions

Pour chaque « I_id-source » de l'ensemble des sections.

On récupère les stratégies « Strat_id »

Pour chaque couple « I_id-source || Strat_id » de sections,

On récupère l'ensemble des intentions cibles « I_id-target »

Ensuite on récupère le produit « RC_id » et « RC_name » correspondant à la Réalisation d'Intention actuelle.

A ce niveau, on récupère les stratégies, les intentions et les produits, il faut alors vérifier si ces stratégies ne sont pas exclues suite à une « constraints_exclusion ».

Pour la Stratégie récupérée, on cherche la 2ème stratégie formant le couple d'une contrainte.

Si une contrainte d'exclusion existe, alors :

| On vérifie si l'une des 2 stratégies a déjà été exécutée ou non. (On rappelle qu'une |
contrainte d'exclusion entre A et B rend impossible l'exécution d'une stratégie B si A
| a déjà été exécutée, et inversement.).

| Si oui, il faut alors savoir sur quel produit a été exécutée cette stratégie.

| | Si le produit est différent du produit actuel, alors on lui propose cette
| | | stratégie en ajoutant cette section candidate

| | Sinon rien.

| Sinon alors on ajoute cette section candidate

|Sinon, on ajoute cette section candidate

Enfin, on peut mettre à jour le tableau Candidate Section, ainsi que la base de données.

Algorithme d'exécution d'une section candidate :

1. On récupère l'identifiant de la section candidate « *Strat_id* »
2. On demande confirmation à l'utilisateur en lui donnant le nom de la stratégie qu'il souhaite exécuter.

3. Si l'utilisateur confirme, on continue l'exécution de la stratégie, sinon, on lui propose de nouveau l'écran des sections candidates pour qu'il puisse renouveler son choix.
4. Ensuite, selon la stratégie exécutée, on affiche à l'utilisateur la fenêtre correspondante.
5. On appelle alors la fonction « NewRCs_enreg(int_strat_id) » qui consiste à créer le produit en passant en paramètre l'identifiant de la stratégie.
6. Selon la stratégie, on crée un nouveau Fragment de produit (Requieremnt Chunck) ou on modifie l'actuel.
7. Ensuite on appelle la fonction « NewRealizedIntention_enreg() » qui consiste à créer une nouvelle Réalisation d'intention.
8. On appelle la fonction « Recupere_Last_RC_id() » qui nous permet de connaitre le dernier produit crée.
9. Ensuite on appelle la fonction « NewExecutedSections_enreg(int_strat_id)» qui consiste à créer une nouvelle Exécution de Section, c'est-à-dire alimenter la Trace.
10. On rafraichit les tableaux pour les données
11. Enfin, on recalcule les sections candidates
12. Fin de l'algorithme

Concernant l'interaction entre les sections et les fragments de produits, nous avons procédé de la même manière que pour la création de la base de données, et nous avons utilisé pour cela le langage de script MERL.

Lorsque cet algorithme est implémenté, il est chargé et exécuté sur VB.Net. L'intérêt de l'utilisation de l'environnement VN.Net est la possibilité de créer des interfaces graphiques qui permettent de communiquer avec l'utilisateur. Les différentes fenêtres qui sont présentées à l'utilisateur facilite le choix de l'utilisateur puisque les données lui sont présentées de manière structurée et explicite.

2.5. Les résultats de l'expérimentation

Les résultats de notre projet exploratoire se présentent sous forme :

- de méta-modèles de produit et de processus : décrivant la méthode à implémenter, et
- d'un moteur d'exécution (outil d'exécution du Map) présenté à travers son interface utilisateur et la structure de sa base de données.

2.5.1. Le méta-modèle de produit

Pour la partie produit, comme le formalisme Map ne précise pas les modèles de produit, nous avons défini un méta-modèle de produit avec le formalisme standard *Entité/Relation* comme c'est le cas dans le travail similaire (Edme, 2005). Dans ce formalisme, un fragment d'un produit peut être une *Entité* ou une *Relation* et peut avoir un ensemble d'attributs tels que le nom, le type de données, les contraintes et une description.

La Figure 20 présente le méta-modèle de produit exprimé en GOPRR. L'association reliant les deux concepts de base de ce méta-modèle est schématisée dans GOPRR par une relation (*In relationship*) sous forme de losange avec deux rôles correspondants à un connecteur avec la partie *Entité* et un connecteur avec la partie *Relation*.

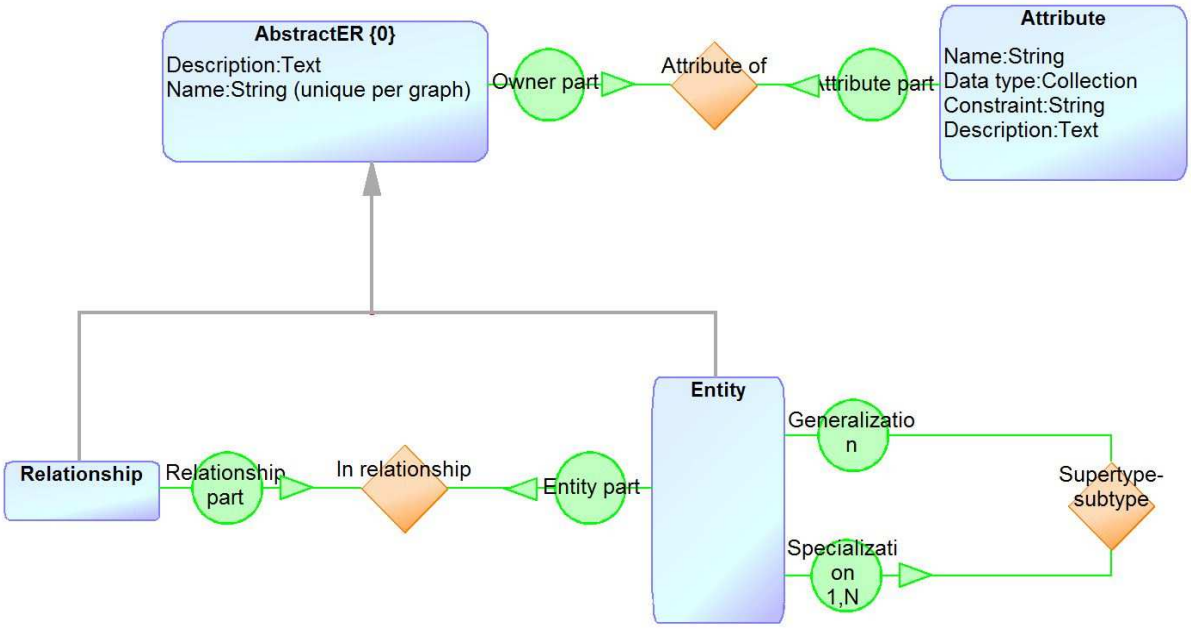


Figure 20. Le méta-modèle de produit : formalisme Entité/Relation exprimé en GOPRR

Ce méta-modèle a été choisi pour sa simplicité, il est décrit en détail dans (Rolland et al., 1999). Les éléments du produit seront référencés tout au long de l'exécution du processus d'exécution du Map de CREWS l'Ecritoire.

2.5.2. Le méta-modèle de processus

Le méta-modèle de processus est présenté à la Figure 21 avec les notations du formalisme GOPRR.

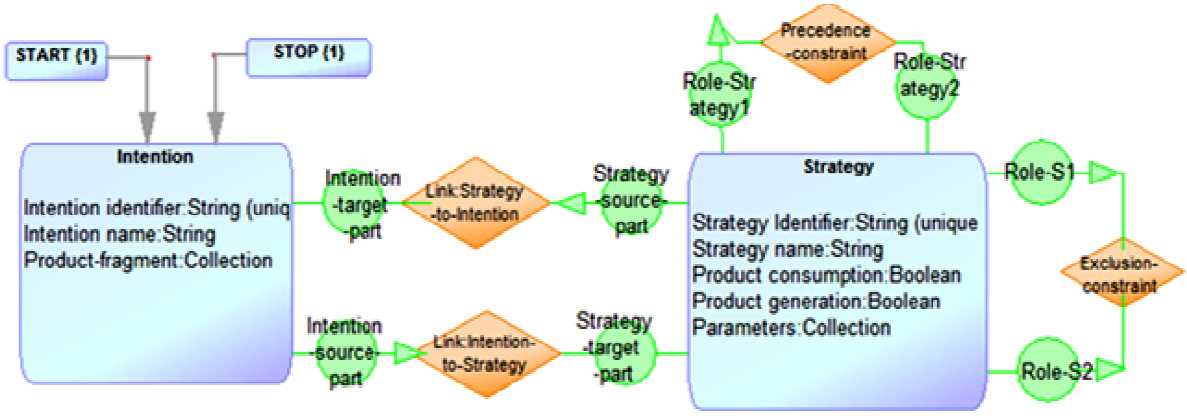


Figure 21. Le méta-modèle statique du formalisme de carte Map défini avec GOPRR

A travers les outils d'édition et de génération de graphes de MetaEdit+, nous avons la possibilité d'instancier notre méta-modèle de processus de Map de la Figure 21 et de définir la carte spécifique de CREWS l'Ecritoire. Le schéma de la Figure 22 correspond au résultat obtenu par cette instanciation.

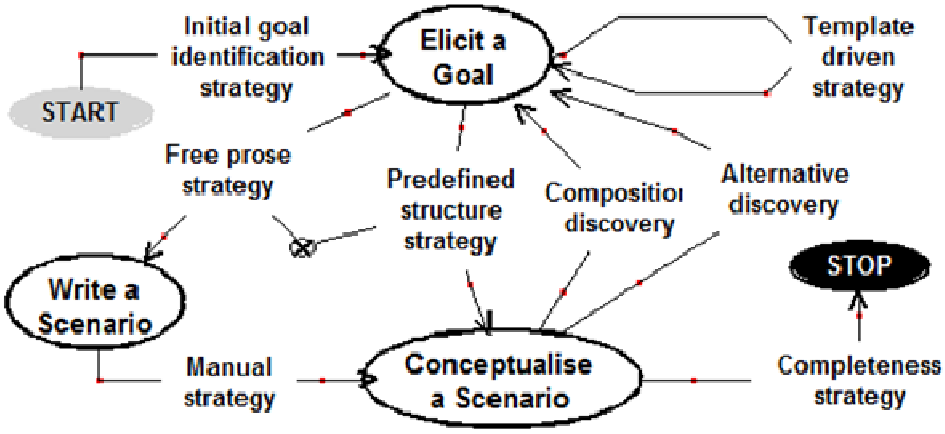


Figure 22. Le modèle du processus CREWS L'Ecritoire défini dans l'outil CASE cible

Il s'agit du modèle de processus de CREWS l'Ecritoire qui est le même que celui présenté à la Figure 13 sauf que cette fois-ci il est obtenue en utilisant notre propre langage de modélisation spécifique ainsi que nos propres symboles que nous avons définis grâce au générateur graphique de MetaEdit+.

2.5.3. Le moteur d'exécution de la carte de CREWS L'Ecritoire

L'exécution d'une carte Map spécifique est assurée par le moteur de cartes généré par les scripts Merl. Ce moteur est un programme VB.net qui manipule une base de données MS Access. Le code exécutable du moteur ainsi que la structure de la base de données dépendent du modèle de produit, et sont construits par le générateur de code selon les termes du modèle de processus (qui dans notre exemple la carte de CREWS L'Ecritoire). Le moteur de carte est spécifique au modèle de processus de l'Ecritoire alors que le générateur de code est générique. Ceci facilite la génération de moteur d'exécution spécifique à d'autre processus conforme au modèle Map.

2.5.3.1. La structure de la base de données

La structure de la base de données manipulée par le moteur est présentée à la Figure 23. On distingue trois groupes dans cette structure :

- Les tables dans le groupe «Process» sont génériques et représentent la définition d'une carte Map,
- Les tables du groupe "Trace" seront remplies au fur et à mesure de l'exécution du processus pour contenir les résultats de l'exécution des sections et des réalisations des intentions (elles sont alors dépendantes du produit), et
- Les tables du groupe «Produit» contiennent les instances du produit (à savoir les morceaux de fragments de besoin).

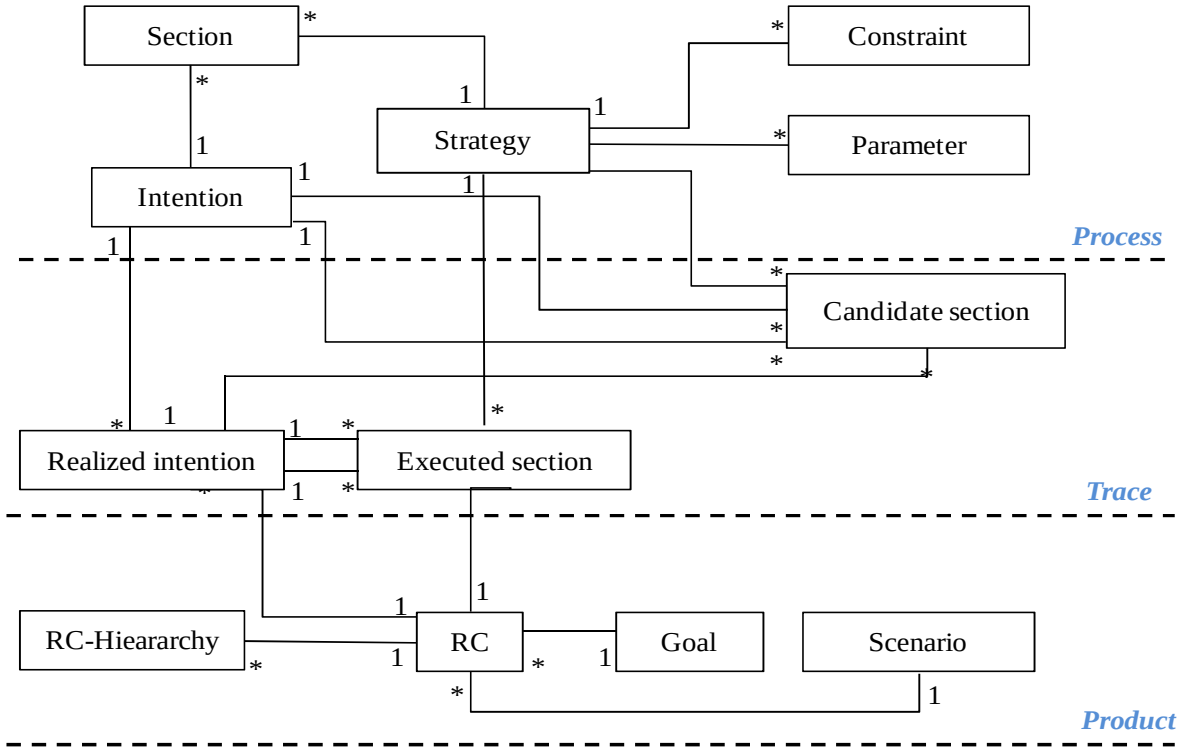


Figure 23. La structure de la DB du moteur du Map

La dépendance du moteur de la carte vis-à-vis de la structure du produit a un impact sur l'algorithme de calcul des sections candidates qui est un élément essentiel dans le processus d'exécution. Ce calcul est basé sur le contenu de la base de données du moteur.

2.5.3.2. L'interface du moteur d'exécution

La Figure 24 présente une capture d'écran de l'interface du moteur d'exécution de la carte Map dans le cas de CREWS l'Ecritoire.

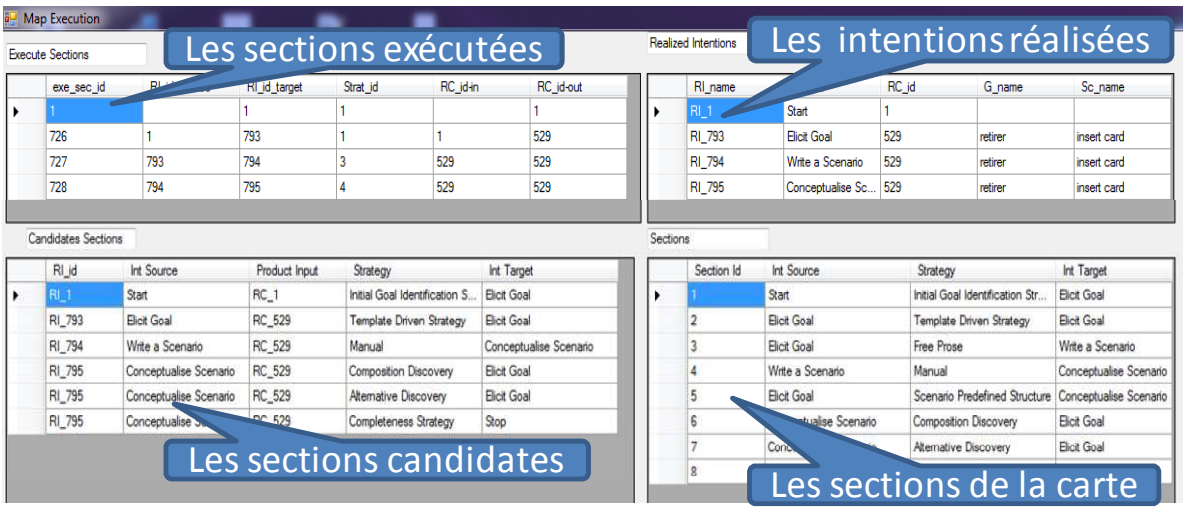


Figure 24. Interface Graphique du Moteur d'exécution

Cette interface est divisée en quatre fenêtres : la 1ère contient la liste de sections exécutées ; la deuxième contient la pile des intentions obtenues ; le contenu de la 3ème fenêtre est statique, il s'agit des sections du processus Map en cours d'exécution; enfin, la 4ème fenêtre contient la liste dynamique des sections candidates (c.à.d. les sections

susceptibles d'être exécutées à la prochaine étape). La structure de cette interface reste la même du démarrage de l'application jusqu'à la fin de l'exécution.

Les informations qu'on peut trouver dans l'interface sont présentées dans les quatre fenêtres comme suit :

- *Les sections exécutées* :
  - Exe_sec_id : Identifiant d'exécution.
  - RI_id_source : Identifiant de la Réalisation d'intention source
  - RI_id_target : Identifiant de la Réalisation d'intention cible
  - Strat_id : Identifiant de la stratégie exécutée
  - RC_id-in : Identifiant du Produit en entrée
  - RC_id-out : Identifiant du Produit (obtenue en sortie)
- *Les intentions réalisées* :
  - RI_name : Nom de la Réalisation d'intention
  - I_name : Nom de l'intention correspondante
  - Rc_id : Identifiant du produit obtenu
  - G_name : Nom du Goal
  - Sc_name : Nom du Scenario
- *Les sections candidates :*
  - RI_id : Identifiant de la Réalisation d'intention
  - Int Source : Nom de l'intention source
  - Product Input : Identifiant du produit en entré
  - Strategy : Nom de la stratégie
  - Int Target : Nom de l'intention cible
- *Les sections de la carte :*
  - Sections_id
  - Int Source : Nom de l'intention source
  - Strategy : Nom de la stratégie
  - Int Target : Nom de l'intention cible

L'interaction avec l'interface consiste à sélectionner une section candidate. Cet événement déclenche l'exécution du code VB.net associé à la stratégie sélectionnée. A la fin de cette exécution, qui comprend généralement une saisie de données par l'utilisateur (par exemple, la saisie de l'intitulé d'un but ou la description d'un scénario), la section exécutée et l'intention réalisée apparaissent respectivement dans la 1ère et 2ème fenêtre. Les sections candidates sont recalculées à chaque itération en fonction de la nouvelle situation.

La première fenêtre correspond à l'historique d'exécution du processus. On y retrouve les données relatives aux réalisations d'intentions, aux stratégies et au produit. Cette fenêtre a une double importance. Tout d'abord, elle permet à l'utilisateur de savoir à tout moment son état d'avancement dans le processus d'exécution. En effet, à chaque exécution, l'utilisateur peut se référer à cette table afin de s'assurer qu'il a bien exécuté telle où telle stratégie avant d'exécuter une autre par exemple. Ensuite, le code qui est derrière cette fenêtre permet au moteur de calculer les sections candidates susceptibles d'être exécutée à l'étape prochaine. En

effet, le calcul des sections candidates est possible grâce à l'exploitation de la trace d'exécution.

La seconde fenêtre correspond au résultat de l'exécution. Tout d'abord, elle nous donne les informations concernant l'exécution de la carte *(RI_name, I_name*). Cela permet de savoir pour une exécution particulière, où est ce qu'on se situe dans le processus ? En début, au milieu ou en fin d'exécution. Aussi, cette fenêtre nous donne les informations sur le produit (*Rc_id, G_id, Sc_id*). Cela permet de connaitre quel fragment de produit a été généré pour chacune des réalisations d'intentions.

La troisième fenêtre est la seule table « interactive ». En effet, c'est grâce à cette table que l'utilisateur va choisir une section candidate pour demander son exécution. Elle informe sur l'état avant l'exécution (*RI_id, Int Source, Product Input*), ainsi que sur l'intention cible (*Int Target*) correspondant à telle stratégie (*Strategy*). C'est en double-cliquant sur une ligne du tableau que l'exécution démarre. Une fenêtre de confirmation s'ouvre alors, elle reprécise le nom de la stratégie qui est sur le point d'être exécuté.

Enfin la quatrième fenêtre contient un tableau d'information qui indique à l'utilisateur toutes les sections existantes de la carte. Cela est pratique lorsque l'utilisateur souhaite préparer une exécution programmée, ou anticiper les stratégies qu'il va exécuter.

Quatre autres écrans s'affichent à l'utilisateur, *Free Prose, Scenario Predefined Structure, Manual.* Ces écrans sont propres aux stratégies du même nom.

- Initial Goal Identification Strategy : Elle permet d'éliciter un nouveau But (**Annexe 6**)
- Free prose : Elle permet à l'utilisateur de saisir son scénario (**Annexe 7**)
- Scenario Predefined Structure : Elle permet de choisir un scénario prédéfinie (**Annexe 8**)
- Manual : L'utilisateur est invité à conceptualiser son scénario de manière manuelle (**Annexe 9**).

L'exécution du moteur guide l'utilisateur à travers l'affichage d'écrans successifs. Ces écrans proposent à l'utilisateur des choix, et des données pour suivre l'historique d'exécution du processus ainsi que l'évolution du produit. Ils permettent également de recueillir les choix et les saisies de l'utilisateur.

2.6. Evaluation de l'expérimentation

2.6.1. Les critères d'évaluation

A travers ce projet exploratoire, nous souhaitons évaluer les trois perspectives suivantes :

- les notations et les formalismes proposés par le méta-outil,
- les fonctionnalités et les interfaces fournies par le méta-outil, et
- l'outil cible obtenu par rapport aux questions clés de maintenance, de portabilité et d'interactivité.

Le Tableau 5 présente nos critères de qualité pour cette évaluation. C'est un ensemble minimal de critères adaptés à notre objectif de recherche.

Perspective	Critères
Méta-formalisme	▪ Niveau d'expressivité pour la spécification d'un processus ▪ Effort cognitif pour spécifier le processus
Meta-outil	▪ Adéquation pour la spécification du processus ▪ Facilité d'usage pour la spécification du processus
Moteur d'exécution	▪ Niveau de gains en termes de maintenabilité de l'outil cible ▪ Niveau de gains en termes de portabilité de l'outil cible ▪ Prise en compte de l'interactivité du moteur par rapport à son environnement

Tableau 5. Les critères d'évaluation

Ces critères ont été inspirés à partir des critères de qualité de l'ISO/IEC 9126 et de ceux d'un travail d'évaluation des méta-outils présenté dans (Niknafs and Ramsin, 2008).

2.6.2. Les résultats de l'évaluation

Le Tableau 6 résume les résultats de l'évaluation de notre projet selon les critères que nous avons choisis.

Perspective	Critère	Résultat
Méta-formalisme	Expressivité	Insuffisante : la sémantique d'exécution s'exprime avec un grand nombre de scripts procéduraux en Merl
	Effort cognitif	Très élevé : le concepteur doit mentalement corréler les scripts de génération en Merl avec les instructions du code généré (en VB.net dans notre cas)
Meta-outil	Adéquation	Faible : un simple éditeur de texte pour écrire les scripts, pas de modèle ni d'interface graphique
	Facilité d'usage	Faible : la validation d'un script en Merl est une tâche ardue, les fonctionnalités de débogage sont limitées
Moteur d'exécution	Maintenabilité	Satisfaisante si les modifications n'altèrent pas la sémantique d'exécution ; sinon elle est insatisfaisante car cela induit des changements complexes dans les scripts de génération de code
	Portabilité	Insatisfaisante : un changement de plateforme cible induit un grand nombre de modifications dans les scripts de génération de code
	Interactivité	L'aspect interaction de l'outil avec l'utilisateur n'est pas pris en compte dès la phase de modélisation, tout cet aspect est codé en dur

Tableau 6. Les résultats de l'évaluation du projet

Concernant la méta-modélisation de la syntaxe abstraite, le formalisme GOPRR de l'outil MetaEdit+ est parfaitement adapté : le méta-modèle est défini directement et reproduit fidèlement la structure statique d'une carte Map. La spécification de la sémantique d'exécution nécessite par contre un effort cognitif très élevé. Elle se fait à travers une collection de scripts écrits en Merl. Ces scripts définissent une logique de navigation à travers les méta-modèles. Cette étape est très complexe, et a du être traitée en deux temps: d'abord, écrire, exécuter et déboguer une première version du code cible du moteur d'exécution, et quand ce code est satisfaisant, l'adapter et l'intégrer dans les scripts de génération Merl. Les erreurs dans le code généré sont difficiles à corriger directement dans les scripts Merl générés. Bien que le méta-outil fournisse une fonction de débogage, elle est largement insuffisante lorsque le code généré devient complexe. Enfin, l'effort cognitif pour corréler le code généré avec les procédures de génération de ce même code était si élevé que toute l'équipe du projet devait coopérer fréquemment sur cette tâche. Les interfaces du méta-outil sont par ailleurs inadéquates et difficile à utiliser lorsqu'il s'agit d'exprimer la sémantique d'exécution dans les scripts de génération de code.

En ce qui concerne la maintenabilité de l'outil obtenu, le score est variable. Toute modification du formalisme carte qui n'a aucune incidence sur la sémantique opérationnelle est facilement manipulable avec l'interface graphique de l'atelier MetaEdit+ et des mises à jour mineures au code généré. Toutefois, si la modification altère la sémantique du langage, l'impact sur le code généré peut alors être beaucoup plus grand, et peut nécessiter des modifications complexes dans ces scripts.

La question de la portabilité est stratégiquement plus importante, car elle détermine la survie des outils à moyen et à long terme. Le score ici n'est pas très satisfaisant. En effet, bien qu'il soit tout à fait possible de mettre à jour le code généré et les scripts de génération pour cibler une nouvelle version de la plate-forme cible (par exemple, une nouvelle version de VB.net) ou une plate-forme différente (par exemple, le SGBD MySQL et le langage PHP), le coût de cette migration est très grand. Il nécessite en fait une réécriture complète du générateur de code.

2.6.3. Exigences fondamentales pour une méta-modélisation des processus dans les outils méta-CASE

A travers ce projet avec MetaEdit+, nous avons concrètement exploré les possibilités qu'offrent les ateliers actuels de méta-modélisation et de construction d'outils. Par rapport à d'autres études similaires, telles que celle de (Marttiin et al., 1993) ou de (Niknafs and Ramsin, 2008), nous avons mis l'accent, d'une part sur une évaluation basée sur une expérimentation de l'outil dans un projet concret, et d'autre part, nous nous sommes particulièrement intéressés à la représentation de la sémantique d'exécution d'un langage de modélisation de processus, et comment exploiter cette représentation pour dériver un moteur d'exécution opérationnel. Il apparaît clairement que la méta-modélisation couvre la dimension produit d'un langage de modélisation, c.à.d. sa syntaxe abstraite. La sémantique d'exécution s'exprime sous forme impérative avec des langages procéduraux qui, certes, permettent de profiter des possibilités de l'atelier de méta-conception pour construire un outil CASE pour un langage de modélisation de processus, mais la méta-modélisation ne couvre en réalité

qu'une petite partie de ce processus. De plus, la maintenabilité et la portabilité des outils CASE obtenus, quoique plus facile à gérer que dans le cas d'un développement ad-hoc, elle nécessite des efforts non négligeables pour mettre à jour les générateurs de code.

Le projet exploratoire nous a permis de mieux comprendre un autre critère qui nous semble très important dans la spécification de la sémantique d'exécution d'un langage de méta-modélisation, il s'agit de la spécification du comportement qui exprime l'interactivité de l'outil d'exécution avec son environnement. Ce critère concerne particulièrement la méta-modélisation des langages de modélisation de processus, il s'agit de la forte interaction du processus avec l'environnement d'exécution. L'absence de cet aspect implique un effort de programmation important. Pour cela, nous pensons que l'expression de cet aspect dans la méta-spécification de l'outil ainsi que sa formalisation pourrait contribuer considérablement à l'automatisation de la génération de code de l'outil d'exécution.

A partir de ce constat, nous considérons que la spécification de la sémantique d'exécution d'un langage dans un atelier de type méta-CASE doit satisfaire les exigences suivantes :

(1) La spécification doit prendre une forme **déclarative** pour faciliter son élaboration, sa compréhension et sa validation.

(2) Elle doit posséder une représentation **graphique** qui accroît sa facilité de compréhension et simplifie sa communication.

(3) Cette notation déclarative et graphique doit avoir un **fondement formel** qui garantit son sens et élimine les ambiguïtés d'interprétation.

(4) Pour qu'elle puisse exprimer efficacement la sémantique d'exécution, cette notation doit être **corrélée** avec le langage de méta-modélisation pour pouvoir manipuler les éléments de la syntaxe abstraite (c.à.d. le méta-modèle de produit).

(5) Enfin, cette notation doit être suffisamment **riche** pour qu'il soit possible d'en déduire par génération automatique un outil CASE cible qui intègre un moteur d'exécution de processus.

2.7. Conclusion

Dans ce chapitre nous avons présenté un projet exploratoire qui consiste à développer un moteur d'exécution pour le modèle de processus intentionnel Map en utilisant la technologie méta-CASE. Ce travail nous a amené à explorer la problématique de l'expression de la sémantique d'un méta-modèle de processus avec ces techniques et de montrer, par l'expérimentation, les avantages et les vraies limitations qu'on peut rencontrer lors d'une démarche de construction d'outil d'exécution.

Notre projet est en soi une première contribution à la problématique de construction d'outils d'exécution pour un modèle de processus orienté but. Il se distingue par rapport aux travaux similaires (tels que (Tawbi, 2001), (Moreno and Fernando, 2003) et (Edme, 2005)) par le fait qu'il propose une solution générique de l'exécution de ce modèle intentionnel de processus. En effet, grâce à notre moteur d'exécution générique, il est possible d'exécuter n'importe quel processus décrit par le modèle Map et non pas seulement celui du cas d'étude Crews l'Ecritoire comme c'est le cas dans (Tawbi, 2001). En outre notre solution qui est

basée sur la technologie méta-CASE est plus simple et offre plus de facilité par rapport à la génération de code en la comparant avec la proposition de (Edme, 2005) dans laquelle l'exécution du modèle Map revient à une manipulation d'une base de données relationnelle très compliquée.

Cependant, d'après ce projet exploratoire, nous avons constaté quelques limites notamment un faible pouvoir d'expression du langage de méta-modélisation. Ce qui fait que la spécification de l'outil est insuffisante pour générer automatiquement tout le code et que l'ingénieur est amené à implémenter une couche supplémentaire de code pour mettre en œuvre l'aspect manquant de la spécification. A partir de ce constat, nous avons définit une liste de cinq exigences que les environnements de type metaCASE devraient satisfaire. Touts ces exigences portent sur le langage et l'environnement de méta-modélisation, car les possibilités d'automatisation de la construction de l'outil dépendent directement de la richesse d'expression du méta-modèle et de son l'exploitabilité.

Afin d'élargir notre analyse de ces questions, nous présentons dans le chapitre suivant un état de l'art concernant les outils et les langages de méta-modélisation utilisés dans le cadre de construction d'outils d'exécution de processus. Cet état de l'art a pour objectif d'étudier comment les travaux similaires au notre abordent ces problèmes et quelles sont les limitations qui caractérisent les ateliers de méta-modélisation actuellement disponibles.

CHAPITRE 3 : ETAT DE L'ART

Ce chapitre est consacré à l'état de l'art concernant l'ingénierie des langages, la méta-modélisation et la construction d'outils d'exécution de modèles. Nous nous intéressons particulièrement à l'approche assistée par des méta-outils et nous étudions ses avantages et ses limites à travers quelques exemples provenant de la communauté génie logiciel et celle de SI. A travers cet état de l'art, nous cherchons aussi à comparer les travaux existants, en se basant sur un ensemble de critères qui se rapportent :

- à la démarche suivie tout au long du processus d'ingénierie de langages et d'outils,
- au langage de méta-modélisation qui a servi à spécifier les modèles sous jacents,
- aux fonctionnalités offertes par les méta-outils pour spécifier et construire des outils.

Le but de cette comparaison est de mieux comprendre les technologies actuellement disponibles et pouvoir ainsi mettre en évidence leurs forces et faiblesses.

Le présent chapitre est organisé comme suit : nous présentons d'abord deux exemples introductifs pour expliciter et clarifier le problème du manque d'expression dans les langages de méta-modélisation, notamment par rapport à la sémantique des modèles. La section 2 introduit les principaux concepts relatifs à la méta-modélisation, l'ingénierie des méthodes et la construction d'outils. La section 3 présente une sélection de différentes approches de construction d'outils d'exécution de modèles provenant de domaines différents. La section 4 est consacrée à comparer ces approches et à préciser le positionnement de notre recherche par rapport à eux. Dans la section 5, nous proposons notre grille de comparaison qui est ensuite appliquée dans l'étude comparative. Et enfin, la section 6 clôture ce chapitre par un résumé de la comparaison.

3.1. Exemples introductifs

Dans l'ingénierie des SI, lorsqu'on définit des méta-modèles, ils sont généralement contemplatifs car ils reflètent uniquement la structure statique des modèles, c.à.d. les concepts et les liens entre ces concepts (Clark et al., 2008), (Sprinkle et al., 2010), (Kern et al., 2011), (Fidalgo et al., 2013). Ils sont utilisés en priorité pour faciliter la communication entre les différents acteurs du projet avec des perspectives limitées d'usage ultérieur dans l'ingénierie du modèle. Le concept de méta-modèle productif est apparue avec l'ingénierie dirigée par les modèles (IDM) (Bezivin et al., 2005) et s'est sont généralisé avec l'adoption des travaux de l'OMG sur l'architecture MDA (Soley, 2000). Un méta-modèle producitf est utilisé non seulement pour définir une structure mais aussi pour produire un résultat, un artéfact tel qu'un éditeur de modèle, un outil de validation, de génération de code ou d'exécution de modèles. La communauté de l'ingénierie du logiciel s'est ainsi sensiblement recentrée sur le concept de modèle afin d'exploiter son caractère productif. Elle préconise une vision des outils d'ingénierie qui cherche à mieux exploiter les modèles tout le long du cycle de vie et s'en

servir non seulement pour communiquer mais aussi pour contribuer à la spécification, la construction, la maintenant et l'évolution des systèmes (Bezivin et al., 2005).

Dans le contexte de l'IDM, les modèles sont manipulés via l'utilisation des langages. Un langage consiste en une notation ou syntaxe associée à une sémantique. Un tel langage est soit graphique, soit textuel, soit composé de texte et de symboles graphiques (Kleppe, 2009). Mais les symboles graphiques ou les mots d'un langage textuel ne peuvent pas être exploités si on ne leur donne pas une interprétation par rapport aux éléments du système modélisé. Ainsi, pour comprendre et manipuler un élément du modèle, il faut qu'il puisse être interprété. L'interprétation fournit « le sens » ou la sémantique du modèle par rapport au système qu'il décrit. L'expression de cette sémantique peut être implicite ou explicite, formelle ou non (décrite en langage naturelle par exemple). Cependant, la définition explicite, éventuellement formelle, d'une sémantique est indispensable pour qu'elle puisse être exploitée automatiquement.

A titre d'exemple, on considère ici le méta-modèle suivant qui représente – en utilisant la notation MOF – un extrait du méta-modèle SPEM (Figure 25). Ce méta-modèle décrit la dimension structurelle de ce modèle de processus (les concepts et les relations entre eux). La manière avec laquelle ces éléments interagissent lors de l'exécution n'est pas explicitement exprimée dans le méta-modèle.

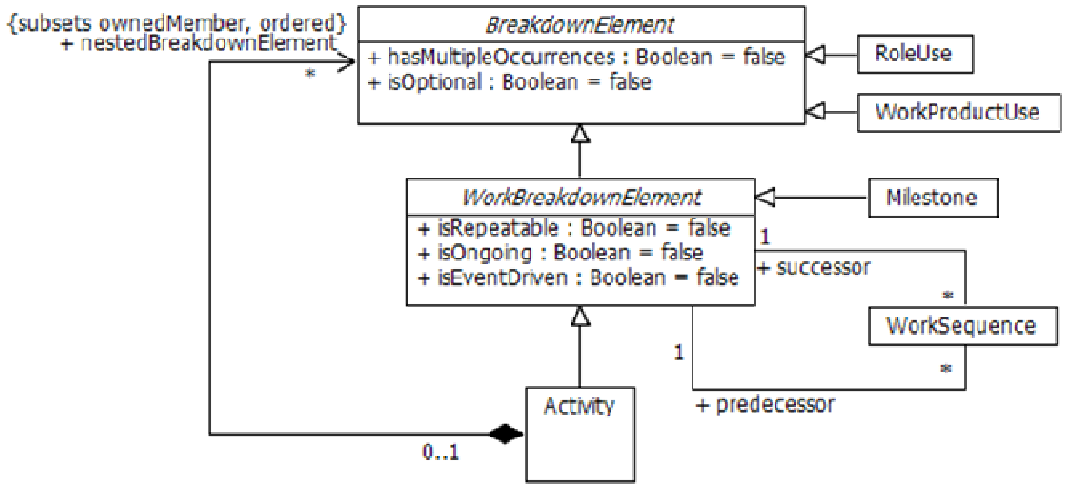


Figure 25. Un extrait du méta-modèle SPEM (SPEM, 2008)

Un deuxième exemple est celui du méta-modèle de workflow tel qu'il est proposé par l'organisation Workflow Management Coalition (Hollingsworth, 1996). Les concepts du langage avec lequel ce méta-modèle de processus est présenté à la Figure 25 ne permettent de représenter que la vue structurelle et n'exprime pas la sémantique du workflow.

Ce n'est que récemment, avec l'accroissement du besoin de construire des outils d'exécution de manière dirigée par les modèles, que la question s'est posée de l'expression explicite de la sémantique d'exécution des langages de modélisation de processus (Paige et al., 2006), (Bendraou et al., 2007a), (Object Management Group, 2011). En effet, les modèles de processus définis par ces langages ont vu leur statut évoluer de contemplatif (aide à la communication et le raisonnement) vers productif (contribution à la réalisation du système

final). Ils représentent désormais des artefacts de base manipulés comme des entrées-sorties par des outils logiciels pour l'exécution, la transformation ou la validation.

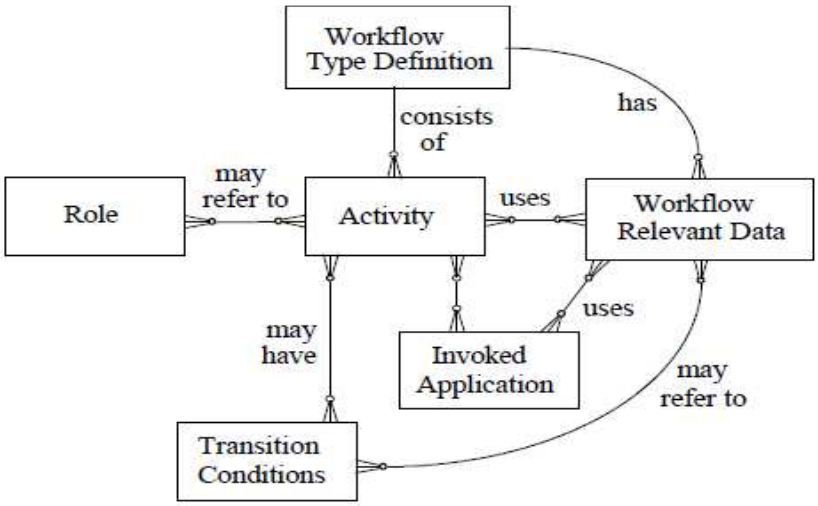


Figure 26. Méta-modèle de workflow (Hollingsworth, 1996)

Dans la suite de ce chapitre, nous proposons un ensemble de définitions reliées aux domaines « spécification des langages » et «construction des outils d'ingénierie ».

3.2. Présentation des concepts

3.2.1. La construction des outils

La construction des outils (ou ingénierie des outils) est une problématique commune à l'Ingénierie des SI (communautés SI) et à l'ingénierie des logiciels (communauté GL) comme montre la Figure 27. Son objectif est de spécifier des outils et les développer en se basant sur de nouveaux méta-langages. L'étude de modèles exécutables et la construction d'outils d'exécution manipulant ces modèles est également un objectif de l'ingénierie des outils.

Récemment, avec l'émergence de l'IDM, la communauté GL cherche à proposer des solutions plus flexibles et moins techniques (c.à.d. plus proche des concepteurs). Pour cela, elle se base de plus en plus sur des modèles à un niveau d'abstraction plus haut et propose des transformations pour passer vers des niveaux plus techniques. Ainsi, elle permet aux ingénieurs de se concentrer davantage sur des aspects plus abstraits que le code. Autrement dit, les ingénieurs du GL veulent s'affranchir des problèmes techniques et améliorer la qualité de leurs logiciels tout en générant le plus de code possible.

De l'autre coté, les gens de la communauté SI souhaitent, en appliquant l'approche dirigée par les modèles, passer par des transformations pour pouvoir générer de manière assez systématique le code à partir d'une spécification d'un niveau d'abstraction élevé. Même si les objectifs sont différents, la construction des outils représente une préoccupation commune des deux communautés.

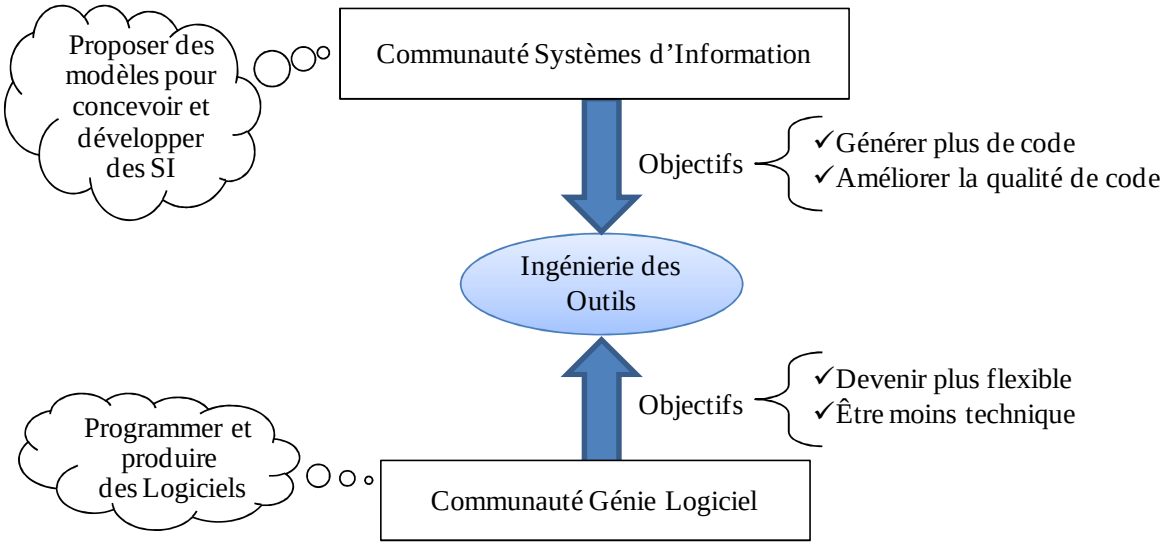


Figure 27. Le croisement des préoccupations des deux communautés SI et GL

La construction d'outils d'ingénierie est un domaine important parce que ces outils permettent d'automatiser le processus de développement d'un SI/ logiciel, d'augmenter la productivité et d'améliorer la qualité des systèmes et des logiciels. Ces outils sont aussi connus sous l'appellation générique CASE pour *Computer Aided Software Engineering* (Loucopoulos and Zicari, 1992). Le besoin de construire des outils d'ingénierie s'est accentué avec l'émergence de l'IDM et le besoin d'automatiser les transformations de modèles. Pour cela, la catégorie de « méta » outils est apparu dans le but de supporter le processus d'ingénierie d'outils et d'améliorer la qualité des outils qu'elle génère. Il s'agit des outils méta-CASE et CAME que nous présentons dans la suite de cette section. Nous introduisons aussi, dans ce qui suit, des définitions de quelques concepts clé de ce mémoire notamment l'ingénierie des méthodes, la méta-modélisation, l'exécutabilité des modèles, la sémantique opérationnelle.

3.2.2. L'ingénierie des méthodes

L'IM se définit comme « une discipline de conceptualisation, de construction et d'adaptation de méthodes, de techniques et d'outils pour le développent des systèmes d'information » (Brinkkemper et al., 1996). Elle est avant tout une tentative de réponse aux difficultés rencontrées dans la mise en œuvre des méthodes. En IM, une méthode traite les deux aspects de l'ingénierie, le produit et le processus. Alors que le produit représente le résultat d'application d'une méthode, le processus décrit une démarche à suivre, généralement sous forme d'un ensemble d'activités reliées entre elles, dans le but d'obtenir ce produit.

Devant la croissance importante du nombre de méthodes d'ingénierie de systèmes, le développement de supports outillés pour la construction et la mise en œuvre de telles méthodes est devenu une préoccupation majeure de la communauté IM et a fait plus tard l'objet de toute une ingénierie. C'est ainsi que l'IM a fait émerger une nouvelle génération d'ateliers logiciels appelée Atelier d'Ingénierie des Méthodes (AIM) (ou **CAME** en anglais pour Computer Aided Method Engineering) (Saeki, 2003).

Une méthode se compose de deux parties: la partie produit, qui capte les connaissances de liées au produit, et la partie processus englobant les aspects liés à l'activité de la méthode. Un outil CASE fournit un support pour une méthode prédéfinie, soit ses modèles de produit et de processus. Les fonctionnalités typiques pour la partie du produit sont : éditer, stocker, valider et transformer les schémas de produit (par exemple en code généré). Si aucun support pour la partie processus n'est disponible, il convient à l'utilisateur final d'utiliser correctement ces fonctionnalités et au bon moment, et c'est au chef de projet d'intégrer l'utilisation du CASE dans le processus d'ingénierie. Les ingénieurs méthodes définissent de nouvelles méthodes et implémentent des outils pour supporter ces méthodes.

Il est à noter que la problématique concernant la construction des outils se pose aussi bien pour les outils CASE que celle des outils CAME (Si dans les CASE on s'intéresse à l'expression de la sémantique des modèles de processus, dans les CAME, il s'agit de celle des méta-modèles). Dans cette thèse, nous nous occupons uniquement de la construction des outils CASE pour la fonctionnalité d'exécution ou de simulation. (Voir Figure 28)

3.2.3. Les méta-outils et les CAME

Par analogie avec les ateliers de génie logiciel (ou CASE) dont le propos est d'apporter une aide à la conduite du développement des SI, un CAME vise à guider la construction et l'adaptation des méthodes. C'est en fait le CASE de l'ingénierie des méthodes. La technologie CAME est une approche de développement de méthodes dans laquelle les ordinateurs sont utilisés pour supporter ou automatiser certaines tâches. Parmi les fonctionnalités offertes par un CAME on cite des éditeurs de spécification des modèles de la méthode, des moyens pour la vérification et la validation, des générateurs de code permettant la simulation ou l'automatisation (partielle ou totale) du modèle de processus de construction de la méthode.

Le CAME est un méta-outil qui permet la personnalisation d'outils et qui n'est pas limité à l'application d'une unique méthode de développement. Cette personnalisation permet de contourner la limite des outils CASE qui sont spécifiques à une méthode particulière. En plus, l'approche assistée par des méta-outils pour le développement et la mise en œuvre de méthodes présente un double avantage par rapport à **l'approche ad-hoc** qui est une approche de construction d'outils en se basant sur l'intuition et n'ayant pas une démarche précise. D'une part, le fait de *réutiliser* le même méta-outil et le personnaliser pour différentes méthodes permet de *réduire l'effort d'apprentissage* et, par conséquent, de *diminuer le coût* de développement en termes de temps et d'effort cognitif. D'une autre part, cette approche *facilite la tâche de maintenance* de l'outil CASE résultat puisqu'il est obtenu en s'appuyant sur une démarche bien définie.

La démarche classique suivie dans un environnement CAME est décrite dans le schéma de la Figure 28. Dans un environnement CAME, l'ingénieur méthode se charge dans une première étape de spécifier les composants génériques du méta-outil en se référant à une méta-méthode. L'étape suivante consiste à personnaliser le méta-outil selon les termes d'une méthode particulière afin d'obtenir un outil spécifique. Enfin, l'ingénieur SI se sert de l'outil résultat pour générer le produit final (qui est le SI).

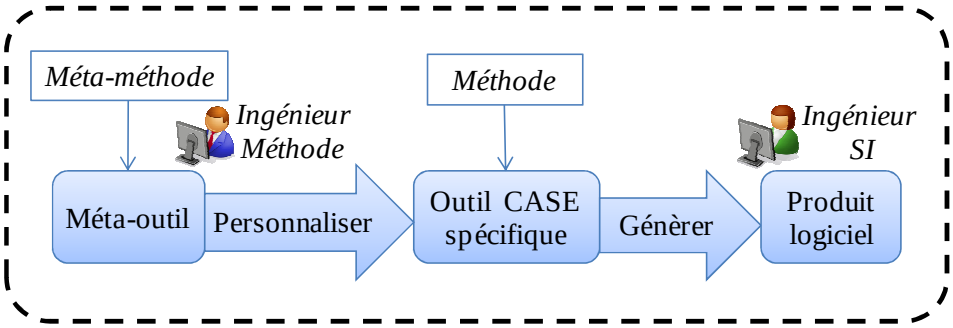


Figure 28. Environnement CAME

Ainsi, on peut distinguer entre trois composants dans un environnement CAME:

- Le composant spécification est à la charge de l'ingénieur méthode: qui définit les méta-modèles de l'outil en utilisant des langages de méta-modélisation et des fonctionnalités de construction d'éditeur du méta-outil.
- Le composant personnalisation : pour instancier les méta-modèles et les adapter à une méthode particulière afin d'obtenir une spécification conceptuelle de l'outil, ensuite transformer cette spécification en une spécification exécutable correspondante à un outil CASE personnalisé. On utilise généralement dans cette phase des facilités de générateur de code.
- Le composant d'exécution de l'outil CASE lui-même et production du résultat (qui peut être le logiciel ou l'outil d'exécution).

Plusieurs produits CAME et méta-CASE ont été développés en réalisant partiellement certaines fonctionnalités. Harmsen, Brinkkemper et Oei (Harmsen et al., 1994) ont été les premiers à définir les exigences d'un environnement CAME dont certaines ont été implémentées dans le prototype Decamerone (Harmsen and Brinkkemper, 1995). Un certain nombre de prototypes ont vu le jour tels que MetaEdit+, Meet (Heym and Österle, 1993), Meru (Prakash, 1999), (Gupta and Prakash, 2001) et Mentor (Si-Said et al., 1996).

Dans notre étude, nous allons explorer des environnements CAME afin de savoir les facilités et les limites des méta-outils dans la construction de l'outil CASE. Notre objectif est de savoir comment se servir des fonctionnalités offertes par un environnement CAME pour répondre aux exigences de l'ingénieur méthode qui souhaite produire un outil avec une action de production spécifique (outil d'édition, outil de validation, outil de simulation…).

D'après (Harmsen et al., 1994), les fonctionnalités qu'un outil CAME devrait fournir sont à ce jour bien définies (comme le montre la figure 2 dans le chapitre introductif). Par contre, des travaux considérables doivent encore être réalisés pour implémenter ces fonctionnalités. Dans cette thèse, nous nous intéressons à la fonctionnalité d'exécution/ simulation qui nous aide à construire des outils d'exécution.

3.2.4. La méta-modélisation

La méta-modélisation est l'activité de construire des méta-modèles, elle représente un concept fondamental sur lequel se base l'ingénierie des outils. En effet, dans les outils CAME, la technique de méta-modélisation est utilisée intensivement pour définir les méta-modèles de

produit et de processus qui vont servir à obtenir l'outil CASE. L'utilisation de cette technique s'est accentuée avec l'arrivée de l'IDM qui vise à fournir des langages de modélisation, plus abstraits et plus facilement maîtrisables que des langages de programmation tel que Java ou C, tout en ayant les qualités d'un langage directement interprétable par une machine.

Dans le domaine de l'informatique, la méta-modélisation se définit comme la mise en évidence d'un ensemble de concepts pour un domaine particulier (Barais, 2007). Parmi ces concepts, il y a le *modèle* qui représente un phénomène du monde réel, le méta-modèle qui est une abstraction mettant en évidence les concepts utilisés pour définir le modèle et le méta-méta-modèle qui est forme d'abstraction d'ordre supérieur composée de concepts génériques permettant de définir des méta-modèles (voir Figure 29). Un méta-modèle est une « définition formelle » d'un modèle qui aide à le comprendre et qui facilite le raisonnement sur sa structure, sa sémantique et son usage. De la même manière qu'il est nécessaire d'avoir un méta-modèle pour interpréter un modèle, pour pouvoir interpréter un méta-modèle il faut disposer d'une description du langage dans lequel il est écrit : un méta-modèle pour les méta-modèles. C'est naturellement que l'on désigne ce méta-modèle particulier par le terme de méta-méta-modèle (ou langage de méta-modélisation). En pratique, un méta-méta-modèle détermine le paradigme utilisé dans les méta-modèles (les concepts, les liens entre eux et leurs sémantiques). Un langage de modélisation conceptuel peut servir, dans la plus part des cas comme un langage de méta-modélisation. Par exemple, le langage UML peut être considéré comme un langage de modélisation ou encore un langage de méta-modélisation.

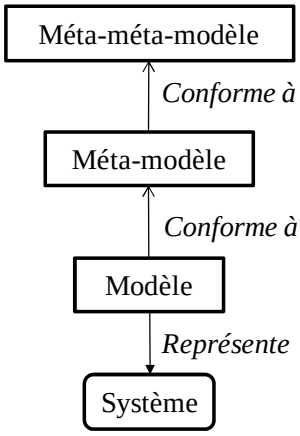


Figure 29. Notions de base de la méta-modélisation

La méta-modélisation, est très utilisée dans le domaine de l'ingénierie des systèmes d'information et particulièrement dans l'ingénierie des modèles et des méthodes (Rolland, 2005a, 2007) (Souveyet, 1991). Cette technique bénéficie des avantages de la modélisation, notamment la séparation des préoccupations pour définir un système qui est une propriété de plus en plus utilisée dans le développement de systèmes informatiques afin d'en maîtriser la complexité.

Dans l'ingénierie des méthodes, la méta-modélisation représente un outil conceptuel indispensable pour définir et raisonner sur de nouveaux modèles. En effet, une méthode dispose habituellement d'un ou de plusieurs « langages de modélisation », permettant de représenter différents aspects (statiques, dynamiques, etc.), de différentes phases (analyse, conception, etc.) du développement d'un système d'information. Il est donc intéressant de

disposer d'un méta-langage définit à un niveau d'abstraction plus La manière avec laquelle ces éléments interagissent lors de l'exécution haut pour pouvoir lier les concepts des différents langages de modélisation mis en œuvre dans la méthode.

Dans l'ingénierie des outils, les méta-modèles sont une définition formelle nécessaire pour concevoir et construire les outils pour éditer, vérifier, transformer et éventuellement exécuter des instances de ces modèles. Dans ce qui suit, nous présentons certain langages de méta-modélisation dans le contexte de l'ingénierie des outils.

3.2.5. Les langages de méta-modélisation

Les premiers travaux de la communauté IDM se sont portés sur la définition du bon niveau d'abstraction des concepts pour définir des langages de méta-modélisation. Parmi ces travaux, on cite à titre d'exemple le standard de l'OMG MOF, Ecore (Budinsky et al., 2003) implanté dans le projet EMF d'Eclipse, KM3 (Jouault et al., 2006) défini par le projet ATLAS de l'INRIA et Kermeta (Muller et al., 2005) défini par le projet TRISKELL de l'INRIA.

D'après (Sunyé, 1999), les travaux sur la méta-modélisation peuvent être classées en deux principales approches :

1. La modélisation des outils de modélisation: les langages issus de cette approche offrent une **vision « données »** tel qu'elle est définie dans le cadre de référence des modèles d'ingénierie des SI (Olle, 1988). Il s'agit d'un point de vue qui fournit la structure des données nécessaires au déroulement d'un processus. Les langages de cette catégorie sont basés principalement sur le formalisme Entité/Association étendu par des contraintes d'intégrité. C'est le cas par exemple de GOPRR et de MOF. D'autres travaux utilisent des langages plus formel comme le langage Z (Saeki and Wenyin, 1994) ou des réseaux de Petri (Bézivin, 1995) ou encore d'autres concepts orienté objet comme Telos (Mylopoulos, 1990).
2. La modélisation du raisonnement de conception : les langages utilisés dans cette catégorie offrent une vision appelée **vision « processus »** permettant de modéliser le processus de développement logiciel. Task Structured Diagrams (Wijers, 1991) est un exemple de langage de méta-modélisation orienté processus. Plus récemment, le langage SPEM (Bendraou et al., 2007a) est proposé par l'OMG pour modéliser des processus.

D'autres langages de méta-modélisation échappent à cette classification comme le langage MEL (Harmsen and Brinkkemper, 1995). Il s'agit d'un langage hybride qui permet de modéliser à la fois les deux aspects « données » et « proceccus ». Dans la suite de ce document, pour éviter la confusion avec les méta-modèles ciblés par ce travail, l'aspect « processus » est nommée aspect « traitements ».

Plus tard dans cet état de l'art (section 3.4), nous allons présenter quelques exemples de langages de méta-modélisation (les plus connus) en citant leurs caractéristiques principales, leurs concepts et notations, ainsi que leur rôle dans la construction d'outils.

3.3. De l'exécution des programmes vers l'exécutabilité des modèles

L'exécution des programmes était, depuis l'apparition de l'informatique, une préoccupation fondamentale pour les programmeurs. En informatique, ***l'exécution*** représente le processus par lequel un ordinateur ou une machine virtuelle met en œuvre les instructions d'un programme informatique qui est écrit dans les termes d'un langage de programmation (tels que le C ou le java). Aujourd'hui, on parle d'exécutabilité des modèles, de sémantique d'exécution de langage de modélisation et d'outils pour supporter ces langages. On constate ainsi que les programmes ne sont plus les seuls artefacts qui ont la caractéristique d'être exécutable. Dans ce qui suit nous définitions quelques concepts clés de notre problématique.

3.3.1. Quelques définitions

3.3.1.1. Syntaxe et sémantique d'un langage

Un langage de programmation est décrit par une syntaxe qui se présente concrètement sous forme de chaines de caractères formant le texte d'un programme. Pour spécifier quels sont les programmes syntaxiquement corrects, on utilise des grammaires qui définissent les règles de bonne formation des programmes. Une fois leurs syntaxes est correcte, les programmes pourront être transformés en instruction machines à l'aide des algorithmes contenus dans un compilateur ou un interpréteur. Les instructions du programme entraînent des séquences d'actions élémentaires sur la machine d'exécution conformément à ce qu'on appelle *la **sémantique***.

Exemple de syntaxe et sémantique d'un langage de programmation:

- *Syntaxe*: z:=x; x:=y; y:=z; => 3 instructions d'affectation séparées par des ';'
- *Sémantique*: échanger les valeurs des variables x et y et donner à z la dernière valeur de y.

C'est quoi la sémantique ? La sémantique est simplement un synonyme du mot sens. Pour un langage de programmation, la sémantique est une donnée d'un système qui permet de donner un sens à chaque programme de ce langage. Elle décrit ce qui se passe dans l'ordinateur lorsqu'un programme de ce langage s'exécute. La sémantique d'un langage définit de manière précise et non ambiguë la signification des constructions de ce langage. Cette connaissance est exprimée généralement dans les algorithmes qui sont codés en dur dans des outils (compilateurs ou interpréteurs) ce qui permet de manipuler le programme et d'en dériver un résultat (ou un comportement). Lorsqu'un programme s'exécute, l'ordinateur traite les données. Cela signifie que la description de la sémantique doit expliquer : (1) les données traitées, (2) les processus de traitement des données et (3) la relation des données et des traitements avec le programme du langage. Les parties 1 et 2 représentent le système d'exécution (runtime system) et la partie 3 correspond à la sémantique.

3.3.1.2. L'expression de la sémantique d'un langage

D'après K. Anneke « la description de la sémantique *d'un langage L est un moyen de communiquer une compréhension subjective des énoncés linguistiques de L à une ou plusieurs personne(s)* » (Kleppe, 2008) . Ceci dit que cette description est bien destinée à des être humains et non pas à des machines, cependant, elle est indispensable pour les personnes

qui sont censées construire les outils qui supportent ces langages (notamment les ingénieurs d'outils).

L'objectif de décrire la sémantique du langage de manière formelle et non ambigüe est de pouvoir raisonner rigoureusement sur ce langage, de le tester, et idéalement de générer différents outils pour ce langage (par exemple des outils d'analyse, de vérification ou de compilation). Mais même si la sémantique des langages de programmation est précise, son expression pose toujours un problème. Ce problème d'expression de la sémantique d'exécution se pose de plus en plus avec l'émergence de l'ingénierie dirigée par des modèles, là où le code et les programmes cèdent de plus en plus de l'espace aux modèles et par conséquences, la question sur ***l'expression de la sémantique*** se pose désormais pour les langages de modélisation au lieu des langages de programmation. On parle ainsi de l'exécutabilité des modèles.

Pour trouver une réponse à la question de l'exécutabilité de la sémantique des langages de modélisation, les premiers travaux se sont inspirés des langages de programmation. La méta-modélisation et la programmation sont deux actions traditionnellement considérées comme différentes. De même, les caractéristiques des langages utilisés pour la méta-modélisation ou bien pour la programmation sont perçues comme différentes. Cependant, les similitudes entre ces langages sont plus grandes que les différences. En effet, le méta-modèle et le programme permettent de décrire un logiciel ou un SI, et surtout, une fois transformé et/ou compilé, ils ont besoin d'être exécutables.

Le Tableau 7 montre quelques différences entre un langage de méta-modélisation et un langage de programmation.

Langage de modélisation	**Langage de programmation**
Imprécis	Précis
Non exécutable	exécutable
Niveau élevé	Niveau bas
Visuel	Textuel
Sémantique informelle	Sémantique exécutable
Produit dérivé	Produit de base

Tableau 7. Comparaison entre langage de modélisation et langages de programmation (Kleppe, 2008)

Les langages de programmation ont une sémantique exécutable mais son expression a toujours été un problème. Cette sémantique est formelle mais elle n'est pas explicitement exprimée, elle est définit directement dans les compilateurs. Par contre dans les langages de modélisation, la sémantique n'est ni précise ni formelle ce qui rend son expression un vrai problème pour les chercheurs visant à développer des outils par dérivation automatique à partir de modèles. La sémantique des langages de modélisation est à ce jour rarement défini et fait actuellement l'objet d'intenses travaux de recherche.

Pour de nombreux langages de modélisation, la sémantique est traditionnellement décrite de manière informelle, en langage naturel, nous citons notamment la norme UML2. Dans le langage UML, on peut trouver parfois un même concept qui peut être interprété de différentes manières. Ceci peut donner lieu à un code qui ne correspond pas exactement à ce qu'on a

modélisé. Actuellement, il n'existe pas de document de référence établissant les bases d'une mise en œuvre de la vérification de la sémantique dans UML. Certes, il y a des pistes dans les ouvrages et dans les manuels de documentation, mais il reste encore un travail énorme à faire.

D'une manière générale, une sémantique informelle (ou encore le manque d'expression de manière explicite) pose plusieurs problèmes, parmi lesquels nous citons :

L'ambigüité et les mauvaises interprétations des concepts du langage de modélisation par les utilisateurs. Les normes des langages exigent une sémantique précise.

Une sémantique informelle ne peut être interprétée par les outils, ce qui incite les développeurs d'outils à implémenter leur propre compréhension de la sémantique. Le même langage pourrait être implémenté différemment. Ainsi, deux outils différents peuvent offrir des implémentations contradictoires de la même sémantique.

Ainsi, la sémantique joue un rôle capital pour déterminer les capacités essentielles d'un langage de modélisation dans une démarche de développement d'outils tel que la capacité d'exécution, de transformation ou de vérification. Et désormais, les différences entre ces deux langages (de modélisation et de programmation) qui semblaient traditionnellement très importantes sont de moins en moins distinctives. En effet, l'IDM a exigé que les méta-modèles soient de plus en plus précis et que les modèles deviennent exécutables. De nouveaux langages et travaux ont ainsi vu le jour. Parmi ces travaux, notre proposition représente un exemple de contribution à l'expression de la sémantique d'exécution de langage de modélisation.

3.3.1.3. Plusieurs manières d'exprimer la sémantique

L'expression de la sémantique d'un langage de modélisation dépend du type de la sémantique. Cette sémantique peut être divisée en deux types: la sémantique statique (ou structurelle) et la sémantique dynamique (ou comportementale). La sémantique statique traite des propriétés à vérifier avant l'exécution et concerne la signification du modèle en termes de structure et de règles de bonne formation. Quant à la sémantique dynamique, elle permet de donner une description du comportement du modèle pendant l'exécution. Dans ce dernier type, on distingue différentes catégories :

- **La sémantique dénotationnelle** : Cette sémantique trop abstraite a été introduite par Scott et Strachey (Stoy, 1977). Son principe est d'associer chaque concept du langage d'origine à un objet mathématique appelé dénotation d'un autre formalisme rigoureusement défini. On parle aussi de modèle formel du langage. Ces objets forment le domaine sémantique du langage d'origine alors que l'association définit sa sémantique. Par ailleurs, la difficulté que nous pouvons rencontrer pour décrire ce type de sémantique est d'identifier les objets mathématiques qui correspondent pertinemment aux concepts du langage et de créer des correspondances entre eux. Dans le contexte de l'IDM, la sémantique d'un langage de modélisation est définie par sa transformation ou traduction vers un autre langage formellement définit tel que les réseaux de Pétri. Sachant que les réseaux de Pétri ont une sémantique formelle, le modèle obtenu prend donc le sens du réseau de Pétri généré. En pratique, cela revient à créer une passerelle entre l'espace technique source et cible permettant en conséquence de profiter des outils disponibles

dans l'espace technique cible pour des fins de simulation ou de vérification par exemple (on l'appelle aussi de sémantique transactionnelle).

- **La sémantique axiomatique :** L'approche axiomatique (Coust, 1990) est basée sur la logique mathématique. La sémantique dans ce cas est spécifiée par des assertions (ou axiomes) sur des propriétés des éléments syntaxiques du langage. Dans le cadre d'un langage de modélisation, la sémantique axiomatique peut être exprimée sur la syntaxe abstraite soit par le langage de méta-modélisation lui-même, soit par un autre langage tel que le langage OCL. L'approche axiomatique est la plus abstraite des 3 méthodes. Elle offre une vision déclarative bien adaptée aux preuves pour les systèmes logiques. Néanmoins, il reste difficile de décrire du comportement avec ce type de sémantique.
- **La sémantique opérationnelle :** (Mosses, 2001) La sémantique opérationnelle décrit comment les programmes ou les modèles sont directement exécutés sur une machine réelle ou abstraite, ce qui revient à décrire une forme d'interpréteur pour le langage. Contrairement à la sémantique dénotationnelle qui s'exprime sur une représentation abstraite différente de celle définie pour le langage considéré, la sémantique opérationnelle s'exprime directement sur la même représentation abstraite du langage. Elle est plus facile à comprendre et à écrire. Cela permet d'implémenter facilement des interpréteurs de langages et de mettre et mettre en œuvre les outils support pour l'exécution.

C'est ce dernier type de sémantique au quel on s'intéresse dans cette thèse.

3.3.1.4. La sémantique opérationnelle

La sémantique opérationnelle sert à donner une signification concernant l'implémentation des systèmes informatiques. On l'appelle aussi une sémantique comportementale ou d'exécution. elle est considérée comme l'une des manières les plus précises de décrire la sémantique comportementale d'un langage (Koudri et al., 2007). Elle est définie comme étant la signification rigoureuse du comportement des instances de ce langage. Par exemple, la sémantique opérationnelle d'un langage de programmation (tels que java ou C) décrit comment chaque programme valide de ce langage doit être interprété. Par analogie, la sémantique opérationnelle d'un langage de modélisation décrit le comportement d'un modèle conforme à ce langage.

Les techniques IDM permettent de définir la structure d'un langage de méta-modélisation, c'est-à-dire les constructions syntaxiques qui composent ce langage et les relations entre ces constructions, mais offrent peu de moyens généraux pour associer une interprétation sémantique à ces constructions. Généralement, la sémantique comportementale des modèles est décrite de manière ad-hoc (à travers des exemples dans les manuels ou expliquée en langage naturel). Et même si divers moyens ont été proposés pour l'exprimer (Utilisation d'un langage d'action comme pour AS-MOF (Paige et al., 2006) ou d'un langage de méta-programmation comme pour avec Kermeta (Muller et al. 2005), ou alors des règles de réécriture QVT), la formalisation de cette sémantique reste toujours un défi à relever.

3.3.2. L'exécutabilité des modèles

Jean Bézivin est l'un des premier chercheurs qui se sont intéressés à la question de l'exécutabilité des modèles (Breton and Bézivin, 2001b). Il définit l'exécutabilité par « *l'étude des comportements opérationnels des modèles* ». L'exécutabilité signifie la capacité à être exécutable, c.à.d. à être interprétable par une machine. Alors que les modèles contemplatifs sont interprétés par l'homme, la préoccupation première des informaticiens est de produire des artéfacts interprétables par la machine. Et c'est pour cette raison qu'ils s'intéressent à étudier l'exécutabilité des modèles.

Pour qu'une machine puisse comprendre un modèle, il faut que les concepts de celui-ci atteignent un certain niveau de précision dans leurs définitions. Cette précision dépend du **degré de formalité** dans les langages de modélisation. On peut trouver un concept qui n'est pas précis par ce qu'il n'y a aucune opérationnalisation derrière ce concept (Il n'est présenté finalement que par un dessin). Tant qu'on ne définit pas les concepts des modèles avec un degré de formalité précis, on aura de l'ambiguïté et de l'imprécision comme c'est le cas dans le monde réel, où l'on peut utiliser différent modèles pour présenter la même chose. Par exemple un planning peut être représenté comme une classe ou encore une association.

Avec des langages de modélisation, on peut interpréter le même concept de différentes manières cependant ce n'est pas le cas avec les langages de programmation. Par exemple, si on prend le modèle E/R, on peut représenter par des concepts différents un même objet. Cela est dû à une imprécision au niveau des concepts. Si on avait une définition très précise de ce que c'est qu'une entité et une définition très précise de l'association, on pourrait rejeter l'une par rapport à l'autre. C'est la réponse à la question « Comment je définis le concept entité ? » qui déterminer le degré de formalité du concept et par la suite du modèle au quel il appartient.

Lorsqu'il s'agit d'un événement en tant qu'un concept dynamique, généralement, on ne décrit jamais la sémantique opérationnelle parce qu'on a l'impression que c'est induit, le problème est qu'on peut induire des sémantiques différentes. Ce genre de concept est traditionnellement décrit de manière incomplète. Il y a pleins de facettes qui restent implicites. Or dans notre travail, on n'a pas besoin que la spécification des concepts soit implicite mais plutôt qu'elle soit complète et formelle pour pouvoir être interprétée par un outil. C'est pour cette raison qu'on s'intéresse à la spécification formelle.

La spécification formelle selon un langage peut être interprétée de plusieurs manières et on peut ainsi dériver des comportements différents. A fin d'obtenir un seul comportement, il faut avoir une description précise et formelle des concepts avec une spécification opérationnelle et il faut aller à un niveau de détail de cette spécification qui fait qu'on peut exécuter le modèle formé par ces concepts. D'ailleurs, l'imprécision dans les langages de méta-modélisation est due au fait que ces langages s'arrêtent à un niveau de détail de l'expression de la sémantique qui n'est pas tout à fait précise et cela nécessite plus d'effort de programmation. Toutefois, le fait d'arriver à un niveau de détail proche du code est une tâche très couteuse en termes de temps.

Tant que la sémantique d'exécution d'un méta-modèle n'est pas précise ni formellement définie, ce dernier demeure un méta-modèle contemplatif et empêche par conséquence la

construction systématique de l'outil qui le supporte, ce qui engendre quelques problèmes dont les principaux sont : le coût élevé de développement et les difficultés associées à la tâche de maintenabilité de l'outil (nous allons détailler ces problèmes par la suite).

L'intérêt à l'étude de l'exécutabilité des modèles ne cesse de grandir. Pour preuve, nous soulignons par exemple la dynamique de la communauté du génie logiciel sur cette problématique. Les conférences, ateliers et projets de recherche se multiplient sur ce thème. On cite à titre d'exemple Exécutable UML (Breton and Bézivin, 2001b), (Mellor and Balcer, 2002), le profile UML pour java (OMG-java Profile, 2004), le projet Triskell qui propose des générateurs automatique d'éditeurs au sein de l'IRISA (unité mixte de recherche regroupant le CNRS, l'Université Rennes 1, l'INRIA et l'INSA à Rennes).

3.3.3. L'expression de l'exécutabilité des modèles

C'est dans le domaine du génie logiciel que la problématique de l'expression de l'exécutabilité dans les méta-modèles a été étudiée en profondeur, grâce notamment à l'essor de l'approche IDM. En effet, étant donné que l'IDM repose fondamentalement sur l'usage extensif de modèles dans toutes les phases de développement d'un logiciel, la question s'est vite posée de comment exécuter un modèle et de comment exprimer la sémantique opérationnelle de celui-ci.

Dans (Breton and Bézivin, 2001b), une première réflexion sur le lien entre méta-modèle et l'expression de l'exécutabilité des modèles a été faite sur les réseaux de Petri. Les auteurs complètent le méta-modèle statique qui décrit les structures fixes dans un modèle (arcs et transitions dans un réseau de Petri), par un méta-modèle dynamique qui décrit les structures de données nécessaires pendant l'exécution d'une instance de ce modèle (les marquages et les mouvements de jeton). Les auteurs reconnaissent cependant que cet ajout n'est pas suffisant pour exprimer toute l'exécutabilité, car le formalisme utilisé (diagramme de classe UML) n'a lui-même pas de sémantique exécutable, et ils appellent ainsi à la création d'un UML exécutable (UML Foundation ou FUML) (Object Management Group, 2011). Et c'est probablement suite à ces réflexions préliminaires sur l'expression de l'exécutabilité que le langage Kermeta a été développé (Muller et al., 2005).

D'autres travaux dans la communauté génie logiciel et IDM se sont penchés sur les modèles de processus d'ingénierie (appelés modèles de procédés logiciels), étant donnée l'importance de décrire, contrôler, et automatiser les procédures avec lesquels les systèmes logiciels sont construits. Un travail important dans ce registre est celui autour d'UML4SPM, qui définit un langage de modélisation de processus d'ingénierie qui est basé sur UML et proche du modèle SPEM de l'OMG (Bendraou et al., 2007a) (Bendraou et al., 2006, 2007b). Plusieurs expérimentations sont faites pour spécifier la sémantique et exprimer l'exécutabilité de ce langage : d'abord, en utilisant le langage d'exécution de processus métier BPEL (Bendraou et al., 2007b), et ensuite en utilisant le langage de méta-spécification Kermeta (Combemale et al., 2008). Pour ces deux approches, la problématique de l'interaction avec l'extérieur (l'utilisateur ou d'autres systèmes) est soulignée comme celle qui distingue les deux approches. Malgré plusieurs avantages (notamment l'existence de systèmes techniques fiables pour l'exécution de modèles BPEL), l'usage de BPEL n'est pas jugé satisfaisant à

cause de l'absence de concepts pour prendre en compte l'interaction avec l'utilisateur. D'autres recherches, plus orientées vers l'étude comparatives d'approches, complètent ces travaux autour de l'exécutabilité d'un modèle de processus (Combemale et al., 2008).

Dans le domaine de l'ingénierie des SI et celui de l'ingénierie des méthodes en particulier, peu de travaux à notre connaissance se sont penchés sur la question de l'expression explicite de l'exécutabilité dans un méta-modèle de processus. La définition d'une méthode étant la combinaison d'un méta-modèle de produit et d'un méta-modèle de processus, la spécification des produits est historiquement la plus ancienne (Harmsen and Saeki, 1996). Concernant le processus, c'est l'approche par assemblage de composants méthodologiques qui est la plus utilisée. Dans (Brinkkemper et al., 2001), le langage MEL (Method Engineering Language), un langage formel pour la spécification des méthodes, est proposé. En dehors de la spécification structurelle des composants, l'aspect processus est décrit dans MEL sous forme d'opérateurs formels dont la sémantique est garantie par la notation mathématique sous-jacente. Cette approche par assemblage de composants méthodologique reste actuellement prédominante (Henderson-sellers and Ralyté), on s'interroge cependant sur les modèles utilisés pour formaliser l'approche, pour spécifier le contenu d'un composant méthodologique, et pour exprimer le processus d'assemblage (Seidita et al., 2007).

Enfin, d'autres travaux sont en cours d'élaboration sur la sémantique des actions pour UML (Action Semantics) visant à normaliser, indépendamment d'un langage de programmation particulier, les différentes formes d'exécutabilité associables à des modèles UML. Pour terminer ce tour d'horizon, il faut mentionner les formalismes et langages de méta-modélisation proposés les outils et environnement de type méta-CASE (tels que MetaEdit+ et ConceptBase). Dans la suite de cette section, nous présentons plus en détail quelques langages de méta-modélisation tout en mettant l'accent sur leur pouvoir d'expression de la sémantique d'exécution.

3.4. Exemples de langages de méta-modélisation

3.4.1. Meta Object Facility (MOF)

Le MOF est un standard de l'OMG (Object Management Group) depuis novembre 1997. C'est un formalisme, pour établir des langages de modélisation (méta-modèles) permettant eux-mêmes de définir des modèles. Le but du MOF est de définir un langage de méta-modélisation (un méta-méta-modèle) unique et utilisé par tous pour représenter des méta-modèles et les modèles qui en découlent. Il est constitué d'un ensemble relativement petit de concepts "objet" permettant de modéliser ce type d'information. Le MOF peut être étendu par héritage ou par composition de manière à représenter des modèles plus évolués.

D'après la Figure 30, on peut constater qu'avec MOF il est possible de décrire un méta-modèle sous forme de diagramme de classes (une description statique), on peut exprimer partiellement la dynamique d'un objet grâce aux opérations liées à une classe (mais cela s'arrête à une signature de l'opération). Avec des contraintes OCL, il est aussi possible d'exprimer partiellement l'aspect dynamique d'un modèle, mais cette expression reste limitée car avec des conditions OCL on ne peut pas invoquer ou déclencher des opérations et on ne

peut pas non plus capturer le détail de l'algorithme d'exécution ou de simulation d'un modèle. La liaison entre MOF et OCL n'existe pas, n'est pas représenté à la Figure 30, mais même si on l'ajoute, ce lien n'est pas formalisé. L'absence de cette relation au niveau conceptuel nécessite un effort important de programmation et une expertise de la part du développeur de l'outil d'exécution du modèle (qui doit savoir exactement comment enchainer les contraintes par exemple).

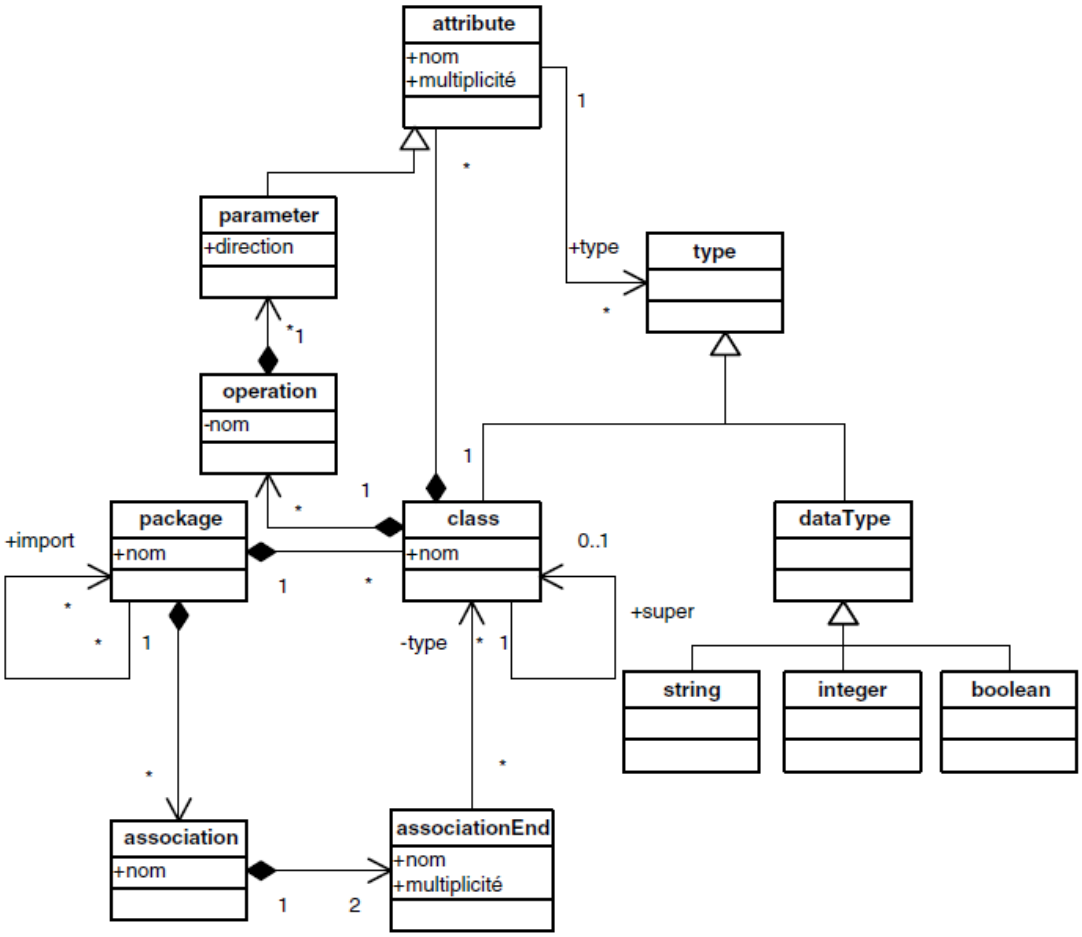


Figure 30. Les concepts principaux de MOF 1.4 (Blanc, 2005)

En tant que méta-méta-modèle, MOF permet de définir la structure d'un ensemble de modèles et de méta-modèles mais pas leurs sémantiques. Le manque de la connaissance sur l'opérationnalisation des modèles conformes aux méta-modèles de MOF explique le fait qu'il n'y a pas d'outillage qui va avec ce langage. (Ce langage ne sera pas pris en compte dans notre étude comparative de la section 3.6 puisqu'on s'intéresse à comparer des couples : <<Langage de méta-modélisation, outil correspondant>>.Le faible pouvoir d'expression du langage MOF justifie le manque d'outillage associé à ce langage et ce manque a créé une sorte de motivation dans certaines équipes pour étudier la question de la sémantique des modèles (Atelier SéMo'07 sur la sémantique des modèles).

3.4.2. Ecore

Ecore (Core Moel) est un méta-méta-modèle très proche du MOF. Il est en fait un sous ensemble du MOF. La Figure 31 illustre les méta-classes d'Ecore. Il est important de savoir

que les méta-modèles conformes à ce méta-méta-modèle sont composés d'EClass contenant des EAttribute et des EReference.

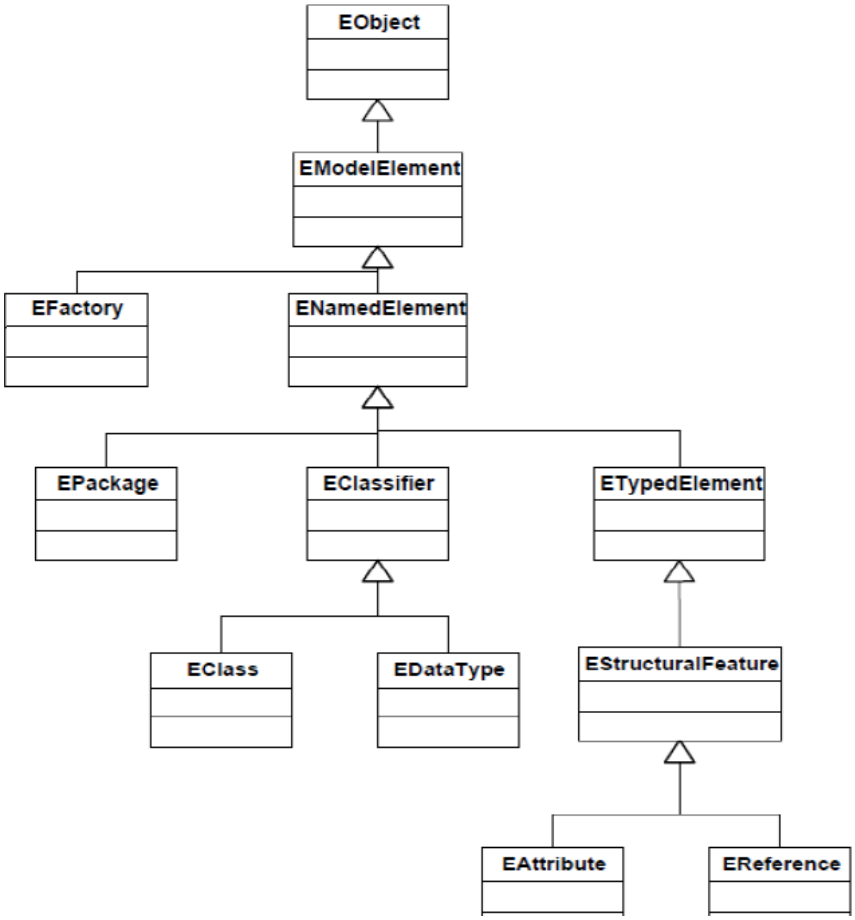


Figure 31. Les concepts du méta-modèle Ecore [25]

Eclass : représente une classe modelée, et contient : un nom, zéro ou plusieurs attributs, zéro ou plusieurs références.

EAttribute : représente un attribut modelé. Un attribut a un nom et un type.

EReference : représente une extrémité d'une association entre classe. Elle a un nom, un flag booléen pour indiquer si elle a du contenu ou non et une référence type.

EDataType: représente le type d'un attribut. Un type de données peut être primitif ou de type objet. Les EDataType en Ecore sont (EBoolean, Echar, EFloat, EString, EByteArray, EBooleanObject, EFloatObject, EJavaObject).

EPackage : Cette classe apporte des facilités pour accéder aux méta-données Ecore du modèle. Elle contient des accesseurs aux EClasses, EAttributes et EReferences implémentées dans le modèle.

EFactory : Comprend une méthode « Create » pour chacune des classes du modèle d'entrée. Cela va permettre de créer des instances (des objets) des classes de l'application.

[25] http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

Avec ces concepts, exprimés à un bon niveau d'abstraction, Ecore permet de définir des modèles et méta-modèles (uniquement la structure). Malgré le manque d'expressivité, de ce langage de méta-modélisation plusieurs outils ont été développés en se basant sur ecore. On cite à titre d'exemples : l'outil TopCased (Farail et al., 2006), EMF, GMF (Graphical Modeling Framework), Visual Editor ou encore Tiger (Ehrig et al., 2005). La plupart de ces outils se limitent à des fonctionnalités d'édition textuelle ou graphique et offrent parfois la possibilité de faire de la vérification. Et ils sont tous développé de manière ad-hoc.

Par exemple, EMF (Eclipse Modeling Framework) est proposé comme étant un outil bon marché tirant parti des avantages de la modélisation formelle et la génération de code Java. Il permet de créer un méta-modèle et des modèles issus de ce méta-modèle ensuite de les manipuler avec un outillage adapté. Cependant si on souhaite changer ou évoluer le méta-méta-modèle, il va falloir changer cet outillage qui a été développé spécifiquement à Ecore de manière ad-hoc et complètement inconnus pour ses utilisateurs. Cette tâche aurait pu être simplifiée si le langage Ecore était suffisamment riche sémantiquement c.à.d. si chaque concept de ce langage avait une sémantique précisant son comportement au moment de l'exécution des modèles qui lui sont conformes. Cette limite d'expression du langage est détectée dans le cas où l'on souhaite développer un outil d'exécution à partir d'un méta-modèle Ecore. Lorsque l'objectif est uniquement de définir des modèles cohérents, une des solutions proposée dans (Garcia and Shidqie, 2007) est l'intégration d' OCL[26] qui est un langage formel avec une sémantique précise pour exprimer des contraintes afin de permettre la vérification. Cette intégration permet de définir des modèles plus précis et de produire des référentiels dotés de comportements de vérification des contraintes structurelles. Le couplage OCL-EMF a été réalisé dans un but de vérification et non pas d'exécution/ simulation de modèles. D'ailleurs, d'après une étude sur la formalisation des méta-modèles dans (Koudri et al., 2007), l'utilisation des couples MOF/OCL et ECore/OCL n'est pas suffisante pour préciser la sémantique d'exécution des méta-modèles d'où l'intérêt d'enrichir d'avantage le niveau méta-modèle ou de trouver d'autres solutions pour garantir la conformité des modèles au regard de leur définition et diminuer la part d'interprétation des outilleurs.

3.4.3. GOPRR

GOPRR (Graph, Objects, Property, Role and Relationship) est un langage de méta-modélisation développé à partir d'OPRR (Object, Property, Relationship, Rôle), une forme de modification du modèle de données Entité-Relation (ER) (Smolander, 1992). C'est un langage qui n'est pas indépendant comme c'est le cas de MOF ou Ecore, mais il est plutôt proposé avec son outil MetaEdit+ qui le supporte. Comme le montre la Figure 32 (récupérée à partir de la thèse de Steven Kelly (Kelly, 1997), ce méta-méta-modèle est structuré avec les concepts suivants: *Graphes, Objets, Propriétés, Relations n-aires et des Rôles*). Les propriétés décrivent les objets, les relations, les graphiques et les rôles. Les rôles peuvent avoir des cardinalités et peuvent être joués par plusieurs objets. Un graphe est composé d'objets et de relations (avec des rôles spécifiques). En outre, les graphes peuvent partager des composants. Les objets peuvent avoir des graphes de décomposition et plusieurs graphs d'explosion. La

[26] http://www.omg.org/spec/OCL/

décomposition permet de décrire plusieurs objets à la fois, alors qu'une explosion est un mécanisme qui est associée à un seul objet. La spécialisation et la généralisation entre les objets sont possibles via l'héritage simple. Les propriétés peuvent être contraintes par des règles qui peuvent être redéfinies dans le sous-type de propriétés.

Une représentation graphique des spécifications est associée au langage GOPRR à travers un éditeur graphique qui permet à l'ingénieur méthode de décrire chaque objet et ses attributs avec une forme. Cette forme est automatiquement utilisée lorsque les objets sont affichés. Afin d'interpréter les spécifications graphiques d'un produit, il est possible, à travers l'outil de GOPRR (MetaEdit+) de définir des rapports personnalisables avec un langage de script (Merl). Des travaux ont ajouté des installations hypertextes (Kaipala, 1997) et de contrôle de processus (Marttiin, 1998) sur l'outil.

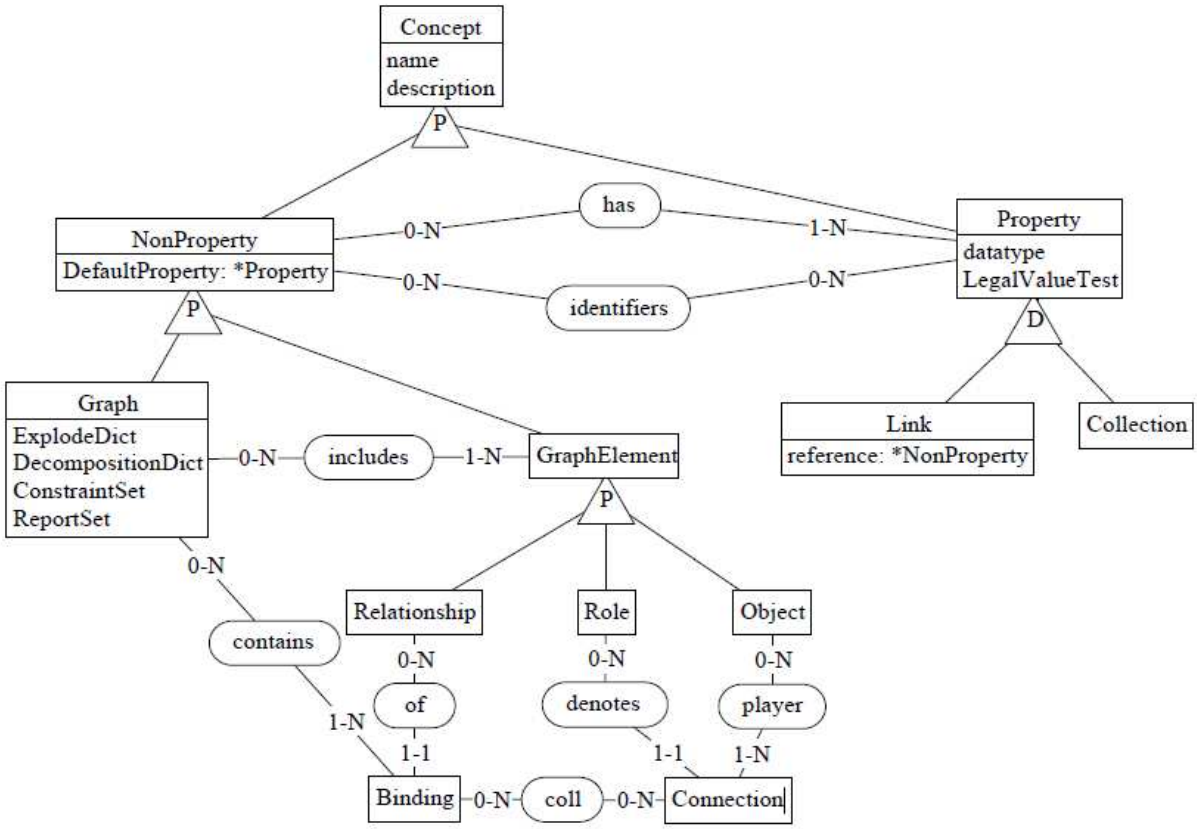


Figure 32. Les concepts du langage GOPRR

GOPRR est un langage simple, facile à utiliser et à mettre en œuvre puisqu'il est soutenu par un environnement méta-CASE. La manipulation de GOPRR à travers MetaEdit+ est partiellement guidée et offre ainsi de nombreux avantages. Tout d'abord, elle donne la possibilité au concepteur d'envisager le méta-méta-modèle. Deuxièmement, elle permet de valider et tester les méta-modèles avec des méthodes réelles, une procédure qui prend énormément de temps avec papier et crayon. Troisièmement, elle permet à un public beaucoup plus large (que scientifique) et plus représentatif pour évaluer les méta-modèles.

Concernant l'expression de la sémantique d'exécution, la sémantique des méta-modèles qui sont conformes à GOPRR est implicite et elle est directement exprimer dans les scripts en

Merl. Des constats et des remarques sur cette question seront présentés en détails dans le chapitre 3 qui concerne une expérimentation effectuée avec l'outil MetaEdit+.

3.4.4. Synthèse sur les langages de méta-modélisation

D'après ces exemples, nous constatons des différences et des similitudes entre les langages de méta-modélisation, mais nous remarquons aussi que, de plus en plus, ces langages évoluent au fil du temps afin de répondre à des exigences particulières. Parmi ces exigences nous nous intéressions dans cette thèse à l'exécutabilité des modèles. Pour satisfaire le besoin de définir des modèles exécutables, les langages de méta-modélisation doivent offrir, comme c'est le cas des langages de programmation, une spécification suffisamment complète pour permettre de construire des outils et de définir des méta-modèles interprétables par ces outils. On peut dire que de Kermeta est le résultat d'une réflexion pour répondre aux limitations des langages de méta-modélisation existants de point de vue exécutabilité. En effet, les langages de méta-modélisation que nous avons vus jusqu'ici (MOF, Ecore, GOPRR) se focalisent sur la spécification structurelle, ils ne peuvent pas être précisément utilisés pour décrire le comportement des méta-modèles. Lorsque une telle définition est nécessaire, l'utilisateur doit se référer à des langages externes, soit impérative comme Java, ou bien déclarative comme OCL. Avec Kermeta, il est devenu possible d'intégrer la spécification comportementale au niveau des méta-modèles grâce à un langage d'action et d'en instancier des modèles qui seront exécutés par un moteur dérivé à partir de langage Kermeta et du langage d'action. Cependant, d'après une étude ,que nous avons effectué dans (Mallouli et al., 2011), nous avons montré que cette solution reste limitée à certain type de méta-modèles : ceux qui ne représentent pas des comportements à forte interactivité (c'est d'ailleurs le cas de leur exemple « machine à état finis »), d'où l'objectif de ce travail.

3.5. Les approches de construction d'outils d'exécution de modèles

Différentes approches de construction d'outils d'exécution de modèles existent. Ces approches se distinguent selon plusieurs critères parmi lesquels : le choix du langage de méta-modélisation qui sera la base de la spécification du produit, la nature de ce langage (déclarative ou impérative), ou la démarche (ou processus) adoptée pour spécifier et générer l'outil d'exécution (démarche par méta-modélisation ou par méta-programmation). Ainsi, on trouve des approches de nature ad-hoc, d'autres plus structurées suivant une démarche dirigée par les modèles ou d'autres approches inspirées par les outils CAME et l'ingénierie des méthodes. Dans ce qui suit nous allons présenter les approches les plus connues.

3.5.1. Approche ad-hoc

La technique ad-hoc de construction d'outils est une technique traditionnelle qui est basée essentiellement sur l'expérience personnelle des ingénieurs acquise lors des travaux antérieurs de développement d'outils. Tant que cette expérience n'est pas formalisée et ne constitue pas une connaissance de base disponible pour les différents ingénieurs d'application, on peut dire que cette connaissance est le résultat d'une technique de construction ad-hoc.

L'inconvénient de l'approche ad-hoc est qu'elle se concentre sur la fabrication manuelle de l'outil et ne suit aucune démarche méthodologique. Par conséquence, on fournit beaucoup d'énergie pour développer l'outil d'exécution d'un modèle, mais surtout, on est obligé de fournir le même effort dans le cas d'évolution du modèle puisqu'on est amené à redévelopper l'outil de nouveau.

3.5.2. Eclipse Modeling Framework (EMF)

EMF, comme nous l'avons déjà signalé dans la section « Ecore », est une plate-forme de modélisation et de génération de code qui facilite la construction d'outils. Il s'agit d'un ensemble d'outils de développement intégré à l'environnement Eclipse sous forme de plugins parmi lesquels on cite : le méta-modèle *Ecore*, l'éditeur *EMF.Edit*, le modèle de génération *GenModel*, etc. EMF a été conçu pour ouvrir Eclipse au développement dirigé par les modèles, c'est une approche basée sur une simplification du MOF. Il permet de définir des méta-modèles puis d'en dériver une implantation en Java pour construire des modèles instances.

La Figure 33 présente l'architecture globale du Framework EMF. Le rôle principal de cet outillage est d'accepter en entrée des modèles ou des fichiers et de générer en sortie du code correspondant à des outils (plug-in) manipulant les données fournies en entrée.

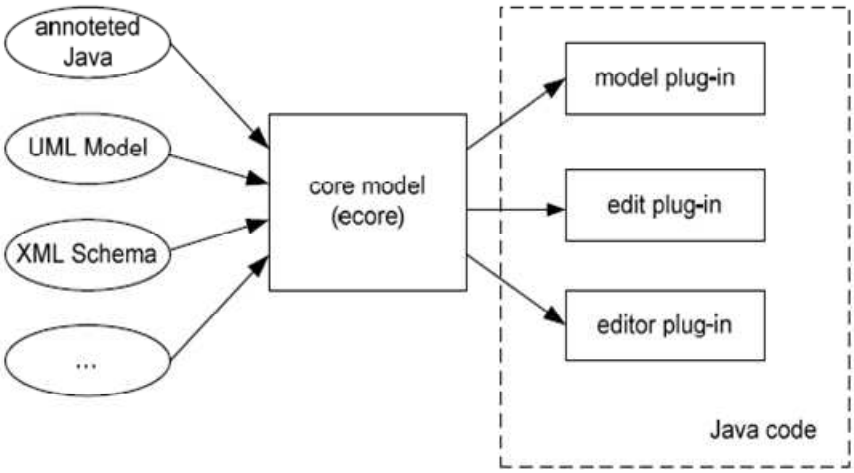


Figure 33. Architecture d'EMF

Parmi les fonctionnalités d'EMF, on peut citer en premier la génération automatique d'un simple éditeur graphique permettant l'édition des modèles sous forme arborescente, c'est-à-dire générer automatiquement, à partir d'un méta-modèle, un éditeur graphique offrant une vue arborescente d'un modèle. Chacun des nœuds de l'éditeur représentera une instance d'une méta-classe. Ensuite, la génération des interfaces de manipulation des modèles consiste à fournir des interfaces graphiques génériques pour manipuler des modèles. Enfin, la génération de code, c'est bien là son objectif premier, améliore la productivité de développement d'application par l'automatisation de la génération de code à partir du modèle. Effectivement, une fois le modèle créé « quelques clics » suffisent à cette génération.

Cette génération de code est paramétrée à l'aide d'un méta-modèle appelé *GenModel*. Le modèle de génération est construit automatiquement à partir du modèle Ecore en associant un élément de paramétrage correspondant à chaque élément du modèle. Les informations

détenues par ces éléments de paramétrage concernent différents aspects qui ont un impact sur la génération : règles de nommage, localisation des fichiers générés, mode de visualisation et d'édition, etc. Cette approche de génération paramétrée par un modèle présente comme avantages de ne pas polluer les éléments de modélisation avec ingrédients relatifs à la génération et de pouvoir appliquer plusieurs générations à un même modèle. Par contre, cette approche présente un inconvénient concernant l'expression de la sémantique d'exécution qui est dans ce cas totalement transparent par rapport aux utilisateurs.

3.5.3. Méta-programmation avec Kermeta

Dans (Jézéquel et al., 2011), les auteurs définissent Kermeta comme un méta-atelier pour l'Ingénierie des Langages Dirigée par les Modèles.

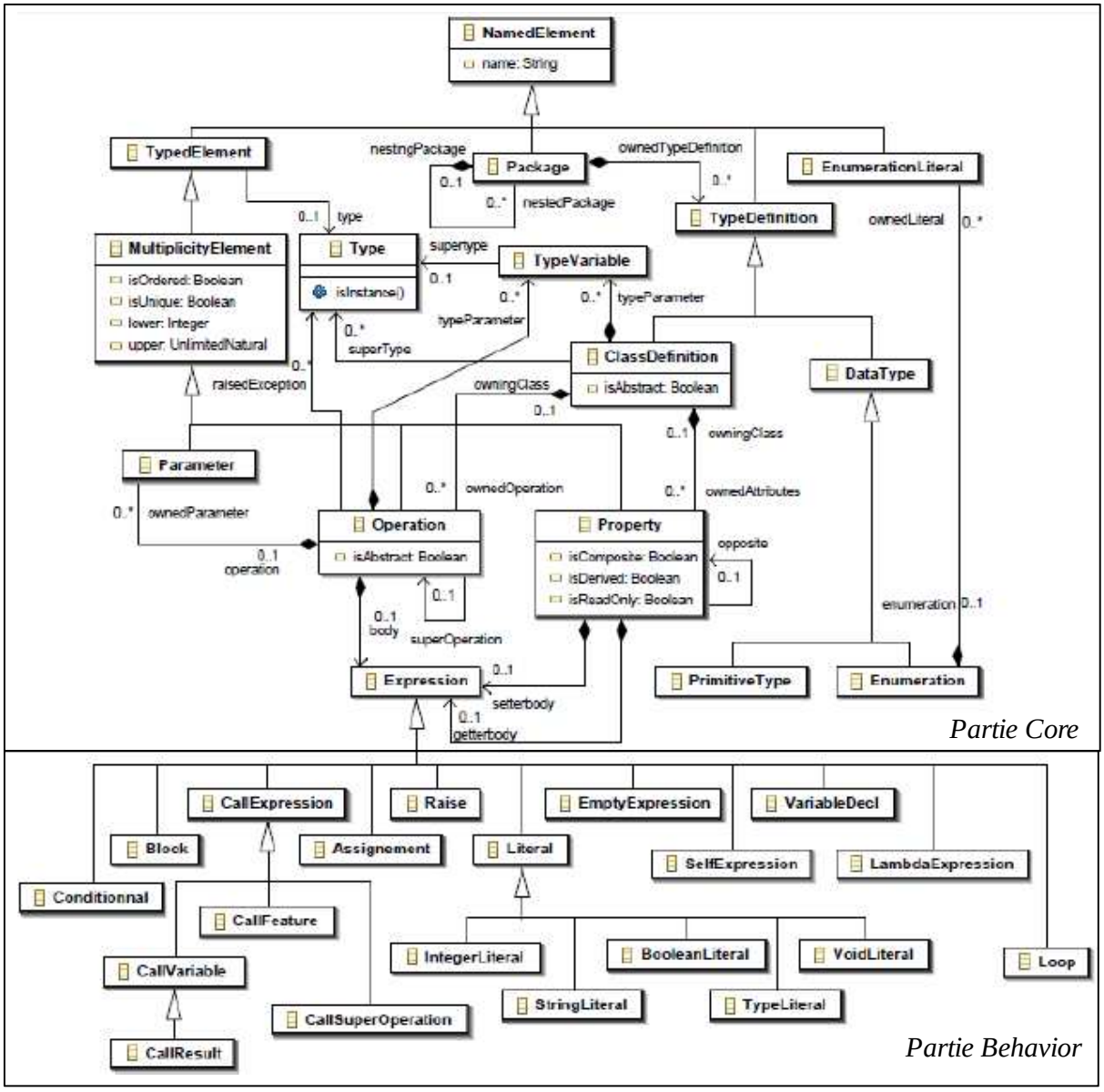


Figure 34. Méta-modèle de Kermeta (Zoé et al., 2009)

Kermeta est un langage de méta-modélisation associé à un environnement de développement de méta-modèles exécutables. Il est basé sur EMOF, une variante du méta-méta-modèle MOF intégrée avec l'environnement Eclipse (Kermeta, 2012). En plus de la spécification de la structure des méta-modèles, Kermeta permet de décrire leurs comportements. Kermeta a été lancée en 2005 (Muller et al., 2005) dans l'équipe Triskell de l'IRISA[27]. Il est utilisée par INRIA[28] et ses partenaires académiques ou industriel dans le monde entier (CNRS[29], INSA[30] de l'université de Rennes…) pour répondre à une variété de besoins dont :

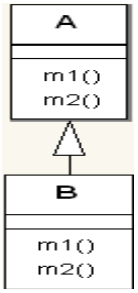
- l'édition de méta-modèles EMOF[31] décrits sous forme textuelle dans l'environnement Eclipse,
- l'ingénierie de langages et de DSL incluant une spécification statique et dynamique,
- l'ajout de contraintes de « bonne formation » (en OCL) aux méta-modèles EMOF, ces contraintes seront vérifiées par l'interpréteur Kermeta lors du chargement du modèle.
- la transformation et la simulation de modèles et de méta-modèles.

Kermeta permet ainsi de définir et d'outiller de nouveaux langages en améliorant la manière de spécifier, simuler et tester la sémantique opérationnelle des méta-modèles. Il définit le comportement de chaque concept de *manière impérative*. Kermeta apporte aussi un complément aux méta-diagrammes UML sous la forme d'une méta-spécification directement opérationnelle. En effet, une description en Kermeta est assimilable à un programme issu de la fusion d'un ensemble de métadonnées (EMOF) et du méta-modèle d'action (Action Semantics) qui est maintenant intégré dans UML 2.0 Superstructure.

Le méta-modèle de Kermeta est composé de deux packages *Core* et *Behavior* (voir Figure 34) correspondant respectivement à Ecore (la partie structurelle) et à une hiérarchie de méta-classes représentant des expressions impératives (partie comportementale). On peut associer chaque expression à une opération et une propriété. Les expressions Kermeta permettent de définir un comportement.

En ce qui concerne la sémantique statique, Kermeta utilise d'abord le langage OCL pour exprimer des contraintes sur le méta-modèle. Ensuite, pour la sémantique d'exécution, le langage utilise des méthodes directement rattachées aux méta-classes. Ce langage d'action capture de manière opérationnelle les règles de déclenchement des opérations du méta-modèle à exécuter. Dans cette approche par méta-programmation, la sémantique d'exécution est ainsi décrite dans le corps des opérations sous forme d'un programme informatique. Elle prend la forme de séquences d'instructions dans le corps des méthodes rattachées aux méta-classes à l'aide d'un langage de programmation spécifique inspiré de Java et qui intègre le paradigme orienté aspects (comme le montre la Figure 35).

[27] Institute for research in computer science and random systems
[28] Institut national de recherche en informatique et en automatique
[29] Centre national de la recherche scientifique
[30] Institut National des Sciences Appliquées
[31] EMOF: *Essential MOF:* est la partie de la spécification MOF qui est utilisée pour définir des méta-modèles simples en utilisant des concepts simples




```
class
A
{
operation m1() is do// Some behavior end
// method m2 is abstract operation m2() is abstract}
class B inherits A
{
method m1() is do// Behavior redefinition end
    method m2() is do// Implementation of the abstract method
```

Figure 35. Définition d'une opération en Kermeta (sémantique opérationnelle)

A partir de cette méta-spécification, Kermeta génère un éditeur, un vérificateur et un exécuteur pour le langage en cours de construction. Il est à noter que cette méta-spécification n'est ni déclarative ni graphique, aucune conceptualisation n'étant disponible pour décrire les séquences d'instructions. Elle n'est pas formelle dans le sens mathématique du terme, elle est cependant strictement encadrée par un langage de méta-programmation. Elle est directement corrélée avec le méta-modèle de produit (syntaxe abstraite) car les méthodes sont rattachées aux méta-classes et agissent sur leurs instances. Enfin, elle est suffisamment riche pour directement s'exécuter dans l'atelier Kermeta.

3.5.4. Spécification de systèmes industriels avec TOPCASED

Le projet TOPCASED a pour but de fournir un atelier basé sur l'IDM pour le développement des systèmes logiciels et matériels embarqués (Farail, 2012). L'outillage TOPCASED a pour principal objectif de simplifier la définition de nouveaux DSL (Domain Specific Languages) ou de langages de modélisation en fournissant des technologies de niveau *méta* telles que des générateurs d'éditeurs syntaxiques (textuels et graphiques), des outils de validation statique et d'exécution de modèles, ce qui lui confère les atouts d'un véritable outil de méta-programmation.

TOPCASED est le résultat d'un projet collaboratif industrie-académie qui cible le développement de systèmes embarqués ayant des contraintes fortes de sûreté de fonctionnement (Farail, 2012). C'est une solution orienté IDM qui s'intègre dans la plateforme de développment Eclipse. Elle utilise les langages UML et SysML et met particulièrement l'accent sur la vérification, la validation et la traçabilité de modèles ainsi que la génération de code à 100%. Comme dans Kermeta, la syntaxe abstraite s'exprime avec Ecore complétée avec une syntaxe concrète définie avec un éditeur graphique. La sémantique du méta-modèle s'exprime à l'aide de formalismes à base d'état (SDMM[32]) et d'événements (EDMM[33]). Cette sémantique permet la simulation d'exécution à l'aide d'une sorte de moteur générique composé de trois éléments : *l'Agenda*, qui gère les instances d'événements décrit dans l'EDMM ; *le Contrôleur* (ou Driver), il supervise l'exécution et interagit avec l'extérieur à l'aide d'interfaces spécifiques (des API : Application Program Interface) ; et *l'Interpréteur* qui déclenche l'exécution des méthodes, met à jour les états des méta-objets et traite les nouvelles occurrences d'événements à traiter.

[32] SDMM: State Definition Meta Model
[33] EDMM: Event Definition Meta Model

La spécification de la sémantique d'exécution est opérationnelle mais 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 doit être réécrit pour chaque nouveau langage. Cette écriture n'est ni déclarative ni graphique, elle n'est pas non plus explicitement corrélée avec le méta-modèle de produit. Les deux méta-modèles (états et événements) sont en effet insuffisamment riches pour permettre une spécification complète de l'exécution, et la sémantique d'exécution se retrouve en grande partie dans l'interpréteur (codée en dur en langage Java). Dans (Crégut et al., 2010), les auteurs reconnaissent cet état de fait et proposent l'identification de 'patterns' génériques pour simplifier sa construction.

3.5.5. Exécution des processus d'ingénierie avec UML4SPM

Cette approche concerne le langage de modélisation des processus d'ingénierie SPEM (*Software Process Engineering Metamodel*) de l'OMG (*Open Management Group*). Dans (Bendraou et al., 2008), les auteurs ont montré que le standard SPEM ne satisfait pas le critère d'exécutabilité. Ils ont proposé UML4SPM (*UML2.0 -Based Language For Software Process Modeling*), 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 (Bendraou et al., 2007b).

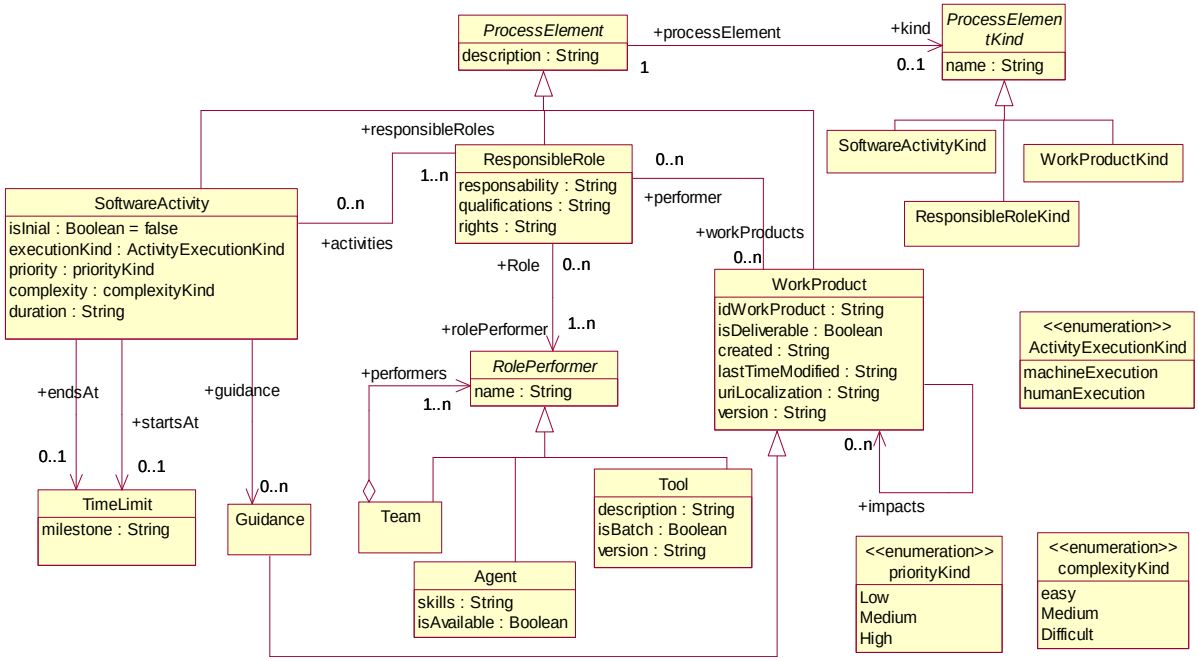


Figure 36. Vue globale du méta-modèle UML4SPM (Bendraou, 2007)

Le langage UML4SPM (Figure 36) vient sous la forme d'un méta-modèle MOF qui étend un sous ensemble du standard UML2.0. UML4SPM profite de l'expressivité d'UML 2.0 en étendant un sous ensemble de leurs éléments convenables à la modélisation du processus (comme *Activités* et *Actions*). En adoptant UML 2.0 comme le base de l'UML4SPM, le langage UML4SPM a profité des avantages d'UML 2.0 qui est actuellement le langage de modélisation le plus connu dans l'industrie, et tout le monde est familiarisé avec ce langage ainsi qu'avec plusieurs outils qui supportent ce langage. De plus, les diagrammes UML sont intuitifs et faciles à comprendre, et les modeleurs d'UML et de SPEM peuvent adopter

facilement avec UML4SPM. Notamment, UML4SPM satisfait le critère d'exécutabilité, on utilise des modèles de processus exécutables au lieu des modèles de processus contemplatifs.

Pour l'exécution des modèles de procédés UML4SPM, Bendraou propose deux approches :

- La première, consiste à définir des règles de correspondances d'UML4SPM vers WS-BPEL (Web Services Business Process Execution Language). Cette approche suit *une technique de traduction* vers BPEL et permet ainsi de réutiliser le moteur d'exécution BPEL qui existe déjà.
- La seconde consiste à *définir un modèle de comportement décrivant la sémantique d'exécution des concepts* UML4SPM, et les modèles de procédés UML4SPM pourront être exécutés directement. Cette dernière approche construit ainsi un moteur d'exécution afin d'exécuter directement des modèles UML4SPM. Dans cette approche, la sémantique opérationnelle des concepts de UML4SPM est décrite par un modèle de comportement, nommé modèle d'exécution. On se base sur la spécification de Executable UML Foundation (Mellor and Balcer, 2002), on y ajoute certaines fonctions (propriétés, opérations, classes) ainsi que la sémantique opérationnelle des concepts sous forme d'opérations décrivant son comportement à l'exécution du modèle. Aussi, pour chaque concept du méta-modèle UML4SPM ayant une sémantique opérationnelle, on définit une Classe d'Exécution dans le modèle d'exécution.

Par contre, aucun prototype n'a été mis en œuvre et aucun outil n'est fourni.

3.5.6. Méta-modélisation et génération de code avec MetaEdit+

MetaEdit+ est un outil méta-CASE développé à l'Université de Jyväskylä, au début des années 1990, dans le cadre du projet 'MetaPHOR' (Kelly et al., 1996). Il est depuis commercialisé par l'entreprise MetaCASE. La fonction principale d'un outil méta-CASE est la construction d'outils CASE et la génération d'outils ayant divers fonctionnalités tels que l'édition, la validation, la transformation, la simulation ou encore l'exécution. MetaEdit+ a été construit autour du langage de méta-modélisation GOPRR présenté dans la section 3.4.3. Grace à ce langage de méta-modélisation, il est possible de spécifier un méta-modèle statique et de dériver un éditeur graphique pour le modèle (MetaCase, 2011). MetaEdit+ est destiné à la création et l'outillage de nouveaux langages spécifique aux domaines (Kelly and Tolvanen, 2008).

MetaEdit+ est un environnement multi-utilisateurs qui peut fonctionner sur de nombreux postes de travail (c'est à-dire des clients connectés à un serveur par un réseau) grâce à un « object repository ». Il offre comme le montre la Figure 37 un ensemble d'outils intégrés :

- Editeurs graphiques de diagrammes, de tableaux et de matrices,
- Navigateurs de modèles (*Browsers*),
- Outils de développement de méthodes,
- Générateur de code et de rapports,
- API et import / export XML.

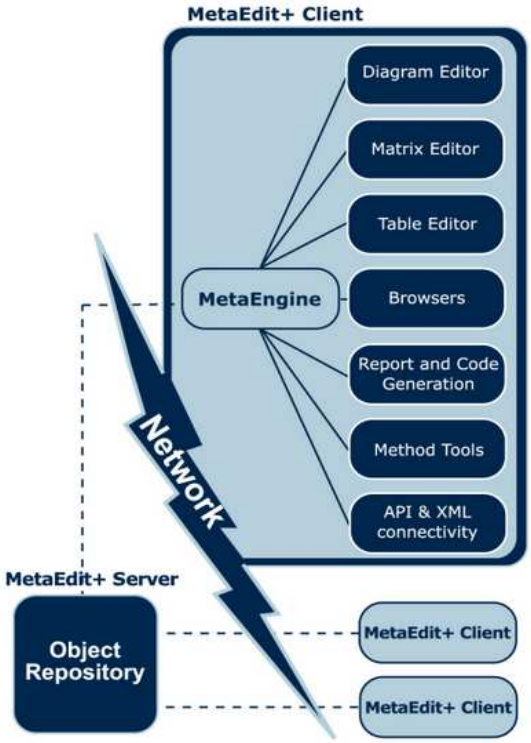


Figure 37. Architecture générale de MetaEdit+ (Kelly et al., 1996)

La spécification d'une fonctionnalité de génération de code s'effectue avec un langage de script (appelé Merl) qui permet de parcourir et manipuler les instances du modèle et de générer des instructions dans n'importe quel langage cible (HTML, XML, C++, Java, etc.).

La sémantique opérationnelle d'un langage s'exprime ainsi par la transformation des structures statiques et non productives du modèle en un ensemble d'instructions (ou de composants) logiciels dans un autre langage ou un autre modèle dont l'exécutabilité est connue et supportée par une plate-forme cible. Alors que la définition du méta-modèle se fait d'une manière déclarative avec une interface graphique, l'exécutabilité s'exprime d'une manière opérationnelle avec une interface de type programmation. C'est le principal inconvénient de MetaEdit+.

3.5.7. Méta-modélisation déclarative avec Concept Base

ConceptBase est définit comme un outil de méta-modélisation pour l'ingénierie des méthodes. Il repose sur un système de gestion de base de données déductive et orientée objet (Jarke et al., 2010). Il peut aussi être considéré comme un environnement de type méta-CASE (Isazadeh and Lamb, 1997b).

ConceptBase est développé à l'Université de Aachen et l'Université de Tilburg (Jarke et al., 2010). Il est principalement utilisé pour la modélisation et méta-modélisation conceptuelle dans le domaine du génie logiciel. Construit avec le langage de logique déclarative Datalog, ConceptBase est un environnement de méta-modélisation extrêmement puissant qui permet de spécifier un nombre quelconque de niveaux d'abstractions et d'exprimer des contraintes et des requêtes sur plusieurs de ces niveaux (et non pas sur un seul niveau comme dans OCL). La saisie d'une spécification se fait textuellement, mais le contenu du référentiel s'affiche graphiquement à la demande de l'utilisateur.

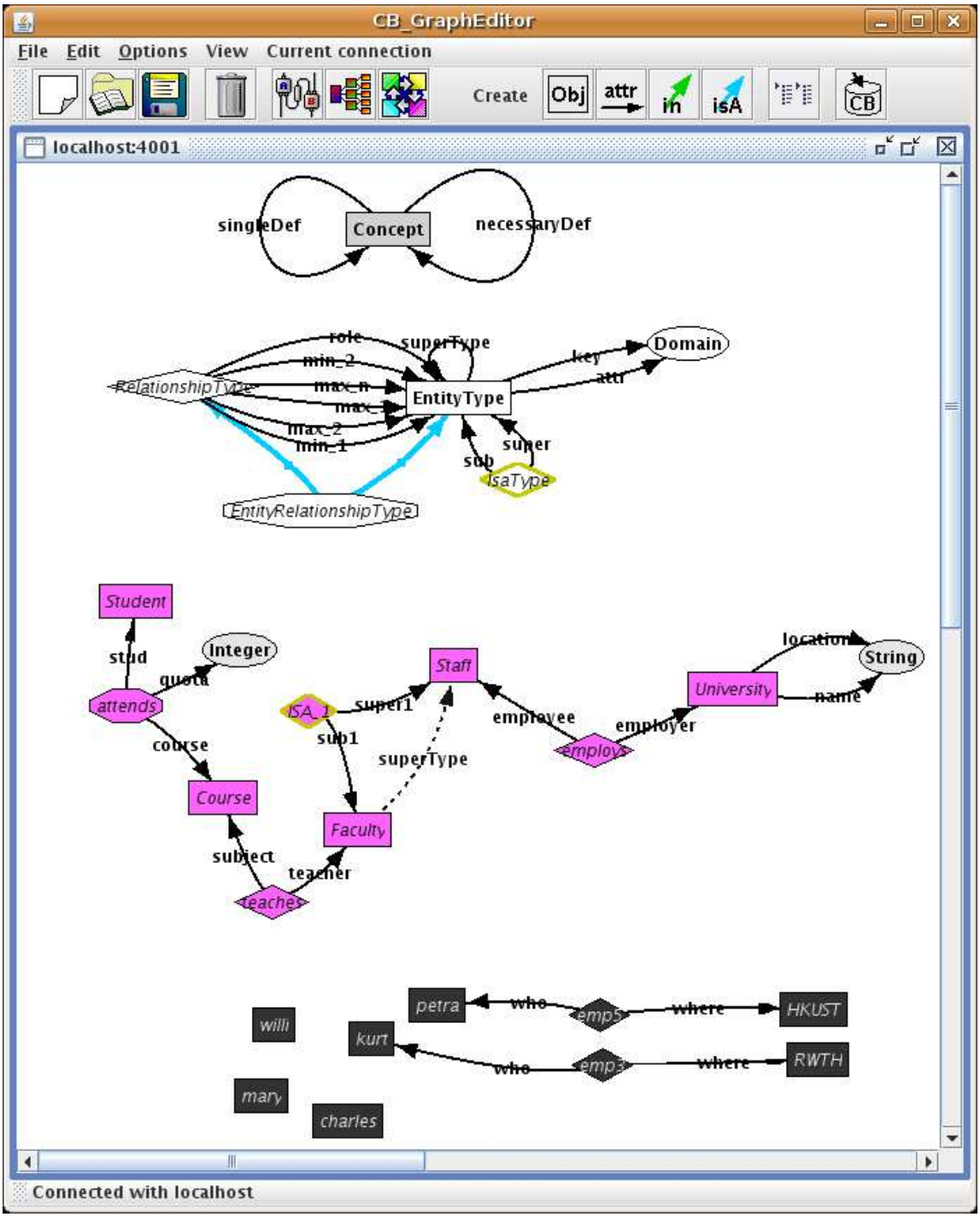


Figure 38. Éditeur graphique de l'outil ConceptBase : un exemple d'instanciation du modèle 'entité-association' en 4 niveaux d'abstraction

L'environnement ConceptBase utilise le langage Telos qui est un langage de modélisation de données et de connaissance issu du courant de modélisation conceptuel en ingénierie des SI (Mylopoulos, 1990). Dans Telos, il n'y pas de niveaux strictes et prédéfinis d'instanciations et d'abstraction, un objet peut potentiellement être instancié à l'infini tant qu'on n'a pas définit de règle pour l'interdire. C'est à l'utilisateur qui organise ses modèles de sorte à avoir un sens clair en termes de données de base, de structures et de méta-structures. L'attribut d'un objet X est un lien de celui-ci vers un autre objet qui peut être une valeur atomique (une date de création), une variable (âge d'un client) ou une structure (adresse). Un

individu X peut ainsi être un objet terminal ou une classe, ou les deux à la fois ; et cela sur deux ou trois niveaux (ou plus) d'abstraction et d'instanciation. Combiné à l'héritage simple et multiple, Telos est un puissant langage formel et graphique pour modéliser et méta-modéliser.

La sémantique statique d'un modèle ou d'un méta-modèle défini dans ConceptBase s'exprime sous forme de contraintes avec un langage de règles. On peut aussi définir des règles déductives pour inférer une donnée ou un lien à partir de données ou de liens existants. Enfin, les règles actives de type Evénement – Condition – Action (ECA) permettent de rendre un modèle dynamique et de lui conférer la possibilité de réagir à l'occurrence de certains événements. Une illustration est proposée dans (Jarke et al., 2010) avec les règles d'exécution d'un réseau de Pétri. La sémantique d'exécution n'est cependant pas exprimable avec ces langages de règles. En effet, les possibilités en termes de simulation et de prototypage s'appuient sur le modèle des bases de données actives fondée sur la réponse à des évènements et les exceptions. Pour le prototypage plus spécifique, il est nécessaire de mettre en œuvre des générateurs de code.

3.5.8. Méta-modélisation et grammaire des attributs avec JastAdd et JastEMF

La grammaire des attributs (GA) est une technique introduite par D. Knuth pour exprimer la sémantique d'un langage de programmation (Knuth, 1968). A chaque nœud de la syntaxe abstraite est associé un (ou plusieurs) attribut(s), ainsi que des fonctions pour calculer la valeur de ces attributs à partir des attributs des nœuds ascendants (héritage) ou descendants (synthèse). Cette technique a été largement développée et appliquée en compilation pour construire les analyseurs sémantiques et les générateurs de code. Le système JastAdd[34] est un méta-compilateur expérimental qui se distingue des applications antérieures par deux innovations majeures (Hedin, 2011). D'une part, la syntaxe abstraite[35] se définit à l'aide d'un langage orienté objet basé sur Java où un nœud est une classe dont la structure (champs et méthodes) sont des attributs (dans le sens de GA). D'autre part, des règles déclaratives sont énoncées dans un langage orienté aspect (basé aussi sur Java) et qui sont interprétées par le moteur du système JastAdd. Ces règles ne sont pas directement rattachées à la syntaxe abstraite, elles peuvent être énoncées dans n'importe quel ordre, et peuvent être regroupées en modules pour faciliter la maintenance et l'extension du langage ainsi que leurs réutilisations futures.

Le système JastAdd étant construit pour la méta-compilation de langages, la proposition du prototype de recherche JastEMF[36] est de le combiner avec l'environnement de méta-modélisation EMF d'Eclipse (Bürger et al., 2011). JastEMF est de ce fait un environnement de méta-modélisation qui intègre la puissance de la grammaire des attributs pour la spécification de la sémantique d'un méta-modèle. Celle-ci s'exprime donc d'une manière tout à fait déclarative et formelle, et fait référence au contenu de la syntaxe abstraite. Cette

[34] JastAdd – http://jastadd.org/web/

[35] Dans JastAdd, il n'y a pas de méta-modèle explicite d'un langage, celui-ci est directement défini par sa syntaxe concrète sous forme de règle BNF à partir de laquelle la syntaxe abstraite est dérivée grâce à un analyseur syntaxique classique.

[36] JastEMF – http://code.google.com/a/eclipselabs.org/p/jastemf/

spécification déclarative est exécutable dans le code généré par JastEMF. Elle n'a cependant pas de représentation graphique, elle s'exprime uniquement sous forme textuelle.

3.5.9. Ingénierie des méthodes basée sur l'IDM avec Moskitt4ME

Moskitt4ME est un environnement CAME qui est destiné à l'ingénierie des méthodes (IM) (Cervera et al., 2012). Cet environnement a été conçu pour surmonter les limitations des approches traditionnelles de l'IM qui sont liés principalement au fait que la plupart d'entre eux se concentrent sur la partie du produit des méthodes et ignorent la partie du processus (Niknafs and Ramsin, 2008). La partie *produit* représente à la fois les objets devant être produites au cours de l'exécution de la méthode et les formalismes qui permettent à ces objets d'être créés et manipulés. La partie *processus* établit la démarche de production des objets de la partie produit. L'objectif de Moskitt4ME est de fournir une solution à la fois méthodologique (théorique) et pratique. Cette solution offre une infrastructure qui (1) permet aux ingénieurs de définir des méthodes complètes qui peuvent être appliquées dans des projets logiciels réels, et aussi, (2) automatise partiellement le processus de construction d'outils en fournissant un support adéquat à la spécification de méthodes. Pour définir avec succès cette infrastructure, l'approche du développement dirigé par les modèles a été utilisée ainsi que des techniques de méta-modélisation et de transformation de modèles.

Afin de guider l'ingénieur méthode pour construire des outils CASE, l'architecture de l'environnement Moskitt4ME adopte une approche de développement dirigée par les modèles. Selon les principes de cette approche, les méthodes sont d'abord définies comme des modèles (method Design), ces modèles sont ensuite utilisés par les transformations de modèles pour générer les environnements CASE (Method Implementation) (comme le montre la Figure 39).

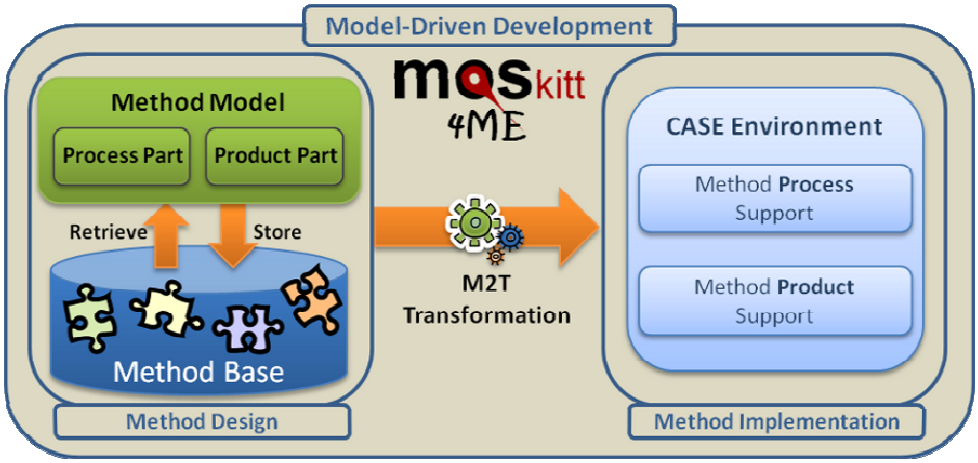


Figure 39. Aperçu de l'approche moskitt4ME (Cervera et al., 2012)

Lors de la conception du processus, les ingénieurs de la méthode définissent à la fois le produit et le processus de la méthode. Ceci est réalisé par l'assemblage de fragments de méthodes qui sont disponibles dans un référentiel de base de la méthode. La norme SPEM 2.0 est utilisée pour la construction des modèles de la méthode. Une fois la spécification conceptuelle est terminée, une série de transformation M2T permet d'obtenir un environnement CASE pour la mise en œuvre de la méthode. Cet environnement CASE offre un support d'assistance pour les deux parties produit et processus de la méthode. En effet, le

support pour le produit consiste en un ensemble d'outils qui permettent la création et la manipulation des modèles de produit alors que le support pour le processus consiste en un moteur d'exécution qui permet la simulation du processus de la méthode.

La sémantique opérationnelle dans ce cas se trouve dans le code de ce moteur d'exécution et elle n'est ni explicite ni déclarative.

3.6. Notre grille de comparaison

Dans le but de comparer les méta-outils étudiés dans ce chapitre, nous proposons dans cette section une grille de comparaison basée sur un ensemble de critères. Pour bien choisir nos critères, nous nous sommes inspirés d'autres travaux comparatifs de méta-outils (Marttiin et al., 1993), (Harmsen and Saeki, 1996), (Kelly, 2004), (Niknafs and Ramsin), (Kelly et al., 2013) et (Erdweg et al., 2013). Ces études ont été la base de l'étude comparative réalisée dans le mémoire de Master Recherche de (Zaidi, 2010). Dans ce mémoire, des outils de méta-modélisation existants on été analysés et comparés à l'aide d'une grille de comparaison et de classification. Cette grille considère l'outil de méta-modélisation selon quatre aspects à savoir: l'aspect *général* (nom date de création, objectifs…), le *formalisme de méta-modélisation* (concepts, relations entre concepts…), les *fonctionnalités* (modélisation, vérification, exécutabilité…) et la *qualité de l'outil généré* (documentation, maturité, complexité de génération de code, complexité du processus général...).

3.6.1. Les critères de comparaison

En s'appuyant sur les connaissances regroupées à partir de ces différents travaux, nous avons constaté que les critères de comparaison changent selon l'objectif de l'étude comparative. Dans notre cas, nous proposons notre propre grille de comparaison d'environnement de méta-outils afin d'en tirer une conclusion concernant la problématique de construction d'outils d'exécution et d'exécutabilité des modèles (Figure 40).

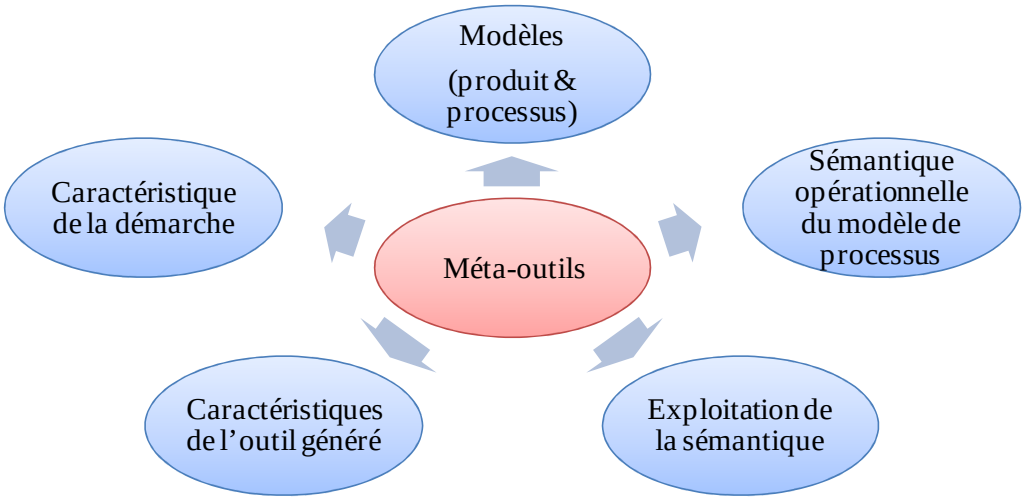


Figure 40. La grille de comparaison des méta-outils

Cette grille se base sur une liste de critères qui ont été d'abord empiriquement validés par analyse de la littérature concernant la comparaison d'outils, ensuite, cette liste a été raffinée par des tests successifs sur les approches étudiés pour aboutir aux critères suivants :

- Langages et formalismes de méta-modélisation
- Expression de la sémantique d'exécution
- Exploitation de la sémantique pour en générer l'outil
- Caractéristiques de la démarche
- Caractéristiques de l'outil généré

Dans la suite, nous allons détailler chacun des critères de comparaison et présenter juste après le tableau comparatif dans lequel sont positionnés les outils étudiés.

3.6.1.1. Langages et formalismes de méta-modélsiation

Ce critère mesure le niveau d'expressivité du langage de méta-modélisation et l'effort cognitif pour spécifier les modèles de produit et de processus. La construction d'un outil se réfère généralement à une méthode d'ingénierie, et comme nous l'avons mentionné précédemment, une méthode se compose d'un produit et d'un processus. Les outils de méta-modélisation qui sont destiné à mettre en œuvre une méthode sont donc amenés à manipuler des modèles de produit et des modèles de processus. On trouve cependant des environnements qui sont orientés produit et se concentrent sur les modèles de produit pour atteindre leurs objectifs. D'autres environnements sont orientés processus et donnent plus d'importance à la formalisation de démarche de mise en œuvre de la méthode.

Ce critère s'intéresse aussi au pouvoir d'expression du langage de méta-modélisation utilisé dans la démarche de construction d'outils (son expressivité). L'expressivité de chaque formalisme (modèle, langage) est connue et mesurée à travers l'information exprimée dans ces concepts, et les relations entre ces concepts (classes, héritages, agrégations …). Si le formalisme ne fournit pas des concepts assez suffisants, l'utilisateur devra encoder ses informations de manière ad-hoc, souvent avec du texte informel attaché (ou séparé) à son méta-modèle, ceci aboutit à des méta-modèles inconsistants, complexes et difficiles à comprendre.

3.6.1.2. Expressoin de la sémantique d'exécution

L'un des enjeux majeurs de l'ingénierie dirigée par les modèles est d'utiliser les modèles non seulement pour des fins de compréhension et de description mais aussi de production c'est-à-dire d'assurer leur exécution. C'est de là que la notion de sémantique d'exécution de modèles apparait. L'exécutabilité d'un modèle est une propriété qui caractérise un modèle qui a une sémantique d'exécution, appelée aussi sémantique opérationnelle car exprimable généralement avec des suites d'opérations sur les éléments du langage.

La sémantique d'exécution est un critère important à prendre en compte dans une démarche de construction d'outils d'exécution. A travers ce critère, on cherche à analyser l'intégration de l'expression de la sémantique opérationnelle dans la spécification conceptuelle. L'expression de la sémantique peut être semi-formelle, ou informelle. Cette

information permettra de donner une idée sur l'effort nécessaire dans le processus de développement de l'outil ainsi qu'à la qualité de cet outil.

3.6.1.3. Exploitation de la sémantique

Ce critère définit la manière avec la quelle la sémantique d'exécution est utilisé par l'outil de méta-modélisation. L'objectif de ce critère est de préciser pour chaque méta-outil s'il assure ou non l'exécutabilité du modèle, et si oui, comment l'exécution est elle obtenue à partir de l'expression de la sémantique. En effet, dans les approches étudiées dans cet état de l'art, des solutions fondamentalement très différentes ont été apportées : projection sur des procédés métiers dans UML4SPEM, génération de code à l'aide de scripts dans MetaEdit+, méta-programmation directe dans les méthodes des méta-classes dans Kermeta, etc.

3.6.1.4. Caractéristiques de la démarche

Par ce critère, on souhaite évaluer la démarche globale proposée par le méta-outil pour obtenir l'outil d'exécution souhaité. Pour qualifier ce critère, on s'intéresse à mesurer le niveau de complexité du processus global et le niveau de complexité de génération de code. Pour cela, on peut se baser par exemple sur le nombre des étapes de développement de méthode et sur les compétences nécessaires pour accomplir ce processus (effort cognitif et connaissance de langages de programmation ou de langage de script, etc.).

Ce critère est difficile à mesurer vu son aspect informel et intangible, il est par exemple difficile de mesurer l'effort cognitif fourni par un être humain pour faire une tâche d'ingénierie. Nous nous sommes basés dans notre évaluation sur des projets exploratoires concrets pour les outils EMF, Kermeta, MetaEdit+ et Concept Base; pour les autres, nous nous sommes basés sur la documentation de l'outil et sur les articles de recherche publiés.

3.6.1.5. Caractéristiques de l'outil généré

Le produit qu'on cherche à générer en utilisant un environnement de méta-modélisation est un outil d'exécution de modèles de processus. Cet outil généré doit satisfaire un ensemble de caractéristiques telles que :

- L'interactivité de l'outil : cette caractéristique concerne l'interaction de l'outil avec son environnement (applications externes et utilisateurs finaux).
- La maintenabilité : cette caractéristique mesure le niveau de gains en termes de maintenabilité de l'outil cible. Elle est satisfaisante si les modifications n'altèrent pas la sémantique d'exécution ; sinon elle est insatisfaisante car cela induit des changements complexes dans les scripts de génération de code.
- La portabilité : cette caractéristique mesure le niveau de gains en termes de portabilité de l'outil cible. Elle est insatisfaisante dans le cas où le changement de plateforme cible induit à un grand nombre de modifications dans les scripts de génération de code.
- La facilité d'usage : de l'outil résultat de point de vue de l'utilisateur final. Cette caractéristique inclut la qualité de l'interface homme-machine de l'outil.

Ce critère (avec ses sous-critères) a été souvent ignoré dans les études comparatives mentionnées auparavant, il s'agit pourtant d'un aspect important pour qualifier et évaluer l'environnement qui a été utilisé pour générer l'outil d'exécution.

3.6.2. Le tableau de comparaison

Pour réaliser la grille de comparaison de méta-outils et d'environnements de construction d'outils, nous nous sommes restreints aux outils que nous avons pu manipuler concrètement, il s'agit de: EMF, Kermeta, TopCased, MetaEdit+, ConceptBase et Moskitt4ME. La première partie du tableau contiendra des critères généraux relatifs à chaque outil : le nom de l'outil, l'origine ou le créateur, l'année de création, les objectifs et les grandes fonctionnalités générales de l'outil (d'après ses créateurs). La partie suivante du tableau contient les 5 critères de comparaison introduits et explicités à la section 3.6.1.

Critères		EMF	Kermeta	TopCased	MetaEdit+	ConceptBase	MosKitt4ME
Généralités	Nom	ConceptBase.cc Eclipse Modeling Framework	Kermeta (Kernel Metamodeling)	Toolkit in Open Source for Critical Applications & Systems Development	MetaEdit+	ConceptBase.cc	Modeling Software KIT (MOSKitt) for Method engineering
	Origine/ créateur	Projet Eclipse GMT, IBM	Projet Triskell, IRISA, France	Consortium / pôle de compétitivité à Toulouse-France	Projet 'MetaPHOR' Université Jyväskylä Finlande	Projet ESPRIT à l'Université d'Aachen et l'Université de Tilburg	Ministère régional de l'Infrastructure et des Transports à Valence, Projet EVERYWARE
	Date de création	2002	2005	2004	1990	1987	2007
	Objectifs de l'outil	Edition de méta-modèles- Génération d'éditeurs arborescents de modèles- génération de code Java	Spécification de méta-modèles- Exécution de modèles	Développement d'applications critiques et de systèmes	Modélisation, Définition de nouveaux DSM- génération d'éditeurs graphiques- Edition de générateurs de code.	Modélisation, méta-modélisation et coordination dans les environnements de conception.	Une extension de l'outil MosKitt pour supporter l'ingénierie des méthodes dans le cadre IDM
	Fonctionnalités de l'outil	Modélisation, vérification syntaxique, génération de code pour la vérification de modèles	Edition textuel, interprétation, débogage, conversion (depuis/vers Ecore), vérification de contraintes OCL, génération de code pour l'exécution de modèles	Mise en œuvre de la première branche du cycle en V pour l'ingénierie du logiciel et/ou du matériel	Modélisation, édition graphique de modèles, conception de méthodes, usage de méthodes, génération de code pour vérification, transformation ou exécution de modèles	Méta-modélisation avec de nombreux niveaux d'abstraction	Définition de méthodes, configuration de méthodes, définition du processus d'exécution de la méthode, dérivation d'outils CASE

Langages et formalismes de méta-modélisatrion	**Nom**	Ecore/UML	Kermeta+MOF	Ecore	GOPPRR	Telos	SPEM 2.0/MOF
	Type	Formalisme de méta-modélisation	Formalisme de méta-programmation	Formalisme de méta-modélisation	Formalisme de méta-modélisation	Langage de modélisation de données et de connaissance	Langage de modélisation de processus
	Concepts	Eclass, Eattribute, EDataType, Epackage, Efactory, Ereference	Class, operation, attribute, reference, Affectation, litteral,…	Eattribute, EDataType, Epackage, Efactory, Ereference	Graphe, Objet, Rôle, Propriété, Port, Relation	Objet, Attribut (ou lien)	Class, DataType, Association, Package, Référence,...
Expression de la sémantique d'exécution		Non exprimée	Exprimée par des opérations attachées à la spécification statique	Non exprimée	Sémantique implicite, quelque part dans le générateur de code Merl	Sémantique statique exprimée par des règles actives conformes aux paradigmes : action, condition et événement mais pas de représentation de la sémantique d'exécution	Implicite et exprimée par une suite de transformation du formalisme SPEM vers BPMN, elle n'est ni explicite ni déclarative
Exploitation de la sémantique		Ad-hoc	Utilisation de la programmation orientée aspects pour lier l'aspect dynamique au statique (niveau technique)	Ad-hoc	Exécution effectuée par un interpréteur externe correspondant au code généré suite à l'exécution du code Merl	Implicite, dans le moteur d'exécution de l'outil ConceptBase	Utilisation d'un moteur d'exécution appelé Activiti Engine
		EMF	**Kermeta**	**MetaEdit+**	**TopCased**	**ConceptBase**	**MosKitt4ME**

Caractéristique de la démarche de construction de l'outil d'exécution	**Niveau de la complexité de la génération de code**	Moyen, car il y a des transformations semi automatiques qui génère en partie le code d'exécution	Elevé, car nécessite une maitrise du langage Kermeta et de la programmation orientée aspect	Moyen, grâce aux transformations entre le niveau conceptuel et le code	Elevé, car l'implémentation nécessite des compétences de programmation à part la maitrise du langage de Script Merl	Elevée, car la sémantique d'exécution n'est pas exprimable avec les règles déductives	Moyen, car il existe un moteur qui interprète les modèles en BPMN et génère du code
	Niveau de complexité du processus global	Moyen, (-) Plusieurs diagrammes difficiles à personnaliser. (+) intégration automatisée OCL-EMF	Elevé, pas de guidage dans le processus et chaque étape nécessite un effort cognitif important pour la réaliser	Elevé, car outil Moyen, spécifique à un domaine particulier, peu utilisé en d'autres	Moyen, nécessite la maitriser d'au moins un langage de programmation pour la génération de code. Guidage faible au niveau de la création du code	Elevée, car ad-hoc	Moyen, car partiellement guidé
Caractéristiques de l'outil généré		Moyennement facile à maintenir et portabilité dépend de celle de la plateforme Eclipse	Maintenabilité et portabilité insuffisantes	Maintenabilité insuffisante: l'interpréteur doit être réécrit pour chaque nouveau langage.	Moyennement facile à maintenir et à porter	Difficilement maintenable et peu portable	Outil généré de meilleure qualité car méthode guidée mais spécifique à un processus procédural et donc peu interactif
		EMF	**Kermeta**	**TopCased**	**MetaEdit+**	**ConceptBase**	**MosKitt4ME**

3.7. Analyse de l'étude comparative

L'analyse du tableau comparatif nous permet de constater que globalement, les méta-outils offrent un support sigificatif pour la spécification d'un langage de modélisation. Ces outils fournissent des fonctionnalités avancées pour la définition de la syntaxe abstraite et la gestion et l'exploitation des méta-modèles structurels pour dériver des éditeurs de modèles (EMF, Kermeta, MetaEdit+) et des outils de vérification ou de simulation (TOPCASED). D'autres, tel que Concept Base, se focalisent sur la richesse d'expression du langage de méta-modélisation et la construction de méta-modèles complexes et très variés.

Concernant l'exécutabilité des modèles et l'obtention d'outils d'exécution, on constate que plusieurs outils n'offrent aucun support pour l'expression explicite de la sémantique. C'est le cas par exemple des outils EMF, TOPCASED et MetaEdit+. Dans ces outils, l'ingénieur devra compléter la spécification des méta-modèles avec d'autres composants logiciels de niveau technique et non plus conceptuel. Ce qui d'une part nécessite une expertise détaillée dans des langages et outils spécifiques de programmation, et d'autre part, soulève des problèmes de maintenabilité, d'évolutivité et de portabilité des outils générés.

Deux outils cependant apportent des réponses originales à ces questions, il s'agit de Kermeta et MosKit4ME. Dans Kermeta, la spécification d'exécution reste de niveau technique même si elle s'exprime dans des méthodes rattachées aux méta-classes du méta-modèle. Cette approche est issue de l'univers du génie logiciel, elle est très complexe à mettre en œuvre, et la spécification obtenue n'est pas du niveau conceptuel et aucune représentation (autre que le code) n'est prévue. Concernant le critère de portabilité des outils générés, un certain progrès est réalisé car l'usage du langage Java garantit en lui-même une grande portabilité de l'implémentation. Pour ce qui est de la maintenabilité de l'outil et de l'évolutivité des méta-modèles, la situation reste problématique car il s'agira pour l'ingénieur des langages de travailler directement sur le code des méthodes.

Etant donné que l'outil MosKit4ME est issu d'un univers assez différent, celui de l'ingénierie des méthodes, les réponses apportées concernent donc une problématique légèrement différente, celle de l'expression d'un processus dans une méthode d'ingénierie. Le choix fait par MosKit4ME est d'utiliser le langage de modélisation SPEM pour décrire ce processus méthodologique. Cette spécificaiton est par la suite directement exécutable sur un moteur d'exécution existant (Activiti Engine) et qui met en œuvre des transforamtions de SPEM vers BPMN selon une démarche de type IDM.

3.8. Conclusion

Dans ce chapitre, nous avons présenté l'état de l'art relatif à la méta-modélisation et construction d'outils d'exécution de processus. Cette étude fait partie d'une recherche autour de la conception et la spécification d'un outil d'exécution pour un modèle de processus intentionnel (MAP). Pour cela, nous avons d'abord présenté un large éventail d'outils et environnements de méta-modélisation. Nous avons ensuite analysé plus en détails et comparé un sous ensemble d'outils que nous avons eu la possibilité d'exéprimenter et de manipuler concrètement (EMF, Kermeta, TopCASED, MetaEdit+, ConceptBase, Moskitt4ME,). Notre

objectif est d'évaluer la pertinence des approches de construction d'outils logiciels pour un modèle de processus. Nous avons pu conclure à travers cette étude comparative que les approches de méta-modélisation sont multiples et complémentaires, mais aussi limitées lorsqu'il s'agit de prendre en compte tous les besoins du concepteur en termes de représentation graphique de la sémantique d'exécution, de la spécification de l'exécutabilité des modèles de processus, et de l'interactivité de l'outil d'exécution vis-à-vis de son environnement. Ce qui limite fortement les possibilités d'exploiter cette spécification pour dériver un moteur d'exécution.

La diversité des approches analysées dans cet état de l'art a été une source d'inspiration pour la suite de notre travail. Dans les approches étudiées, deux mécanismes ont retenu notre attention : l'exploitation des principes de l'IDM dans l'outil d'ingénierie de méthodes MosKit4ME pour exécuter un processus d'ingénierie, et la méta-programmation pour spécifier la sémantique exécutable d'un méta-modèle. En se basant aussi sur l'expérimentation présentée dans le deuxième chapitre, nous sommes amenés à proposer une démarche d'ingénierie qui automatise le processus de construction d'outils d'exécution de modèles, tout en garantissant un résultat final d'une certaine qualité, à savoir un outil d'exécution interactif qui est maintenable et portable. Cette démarche devra intégrer dans le langage de méta-modélisation une représentation explicite et graphique de la sémantique d'exécution du méta-modèle.

Pour cela, nous élaborons dans le chapitre suivant notre proposition d'une spécification de la sémantique d'exécution grâce à la méta-modélisation du comportement qui capture la sémantique d'exécution d'un modèle de processus et qui exprime l'interaction entre les différents éléments de l'architecture de l'outil, ainsi que l'interaction avec l'environnement du modèle de processus (applications externes et utilisateurs finaux).

CHAPITRE 4 : PROPOSITION D'UNE DEMARCHE IDM POUR LA CONSTRUCTION D'OUTILS D'EXECUTION

4.1. Introduction

La génération de code dans les environnements de développement (tels que Eclipse ou NetBeans) permet l'implémentation automatique d'un diagramme de classes orienté objet en UML. Cependant, un diagramme de classes n'offre qu'une vision statique de l'outil ou du système qu'il représente. Cette vision statique s'apparente à la perspective « données » du cadre de référence des modèles d'ingénierie des SI (Olle et al., 1988). Et même s'il est possible de représenter la vision dynamique qui reflète la perspective « traitements », en utilisant d'autres diagrammes orientés objet (par exemple : le diagramme de séquence, ou diagramme d'activité), l'intégration de cet aspect dans le code s'effectue généralement de manière manuelle, ce qui nécessite beaucoup d'effort de programmation.

Afin de générer le plus de code possible à partir d'une spécification d'un modèle de processus ou d'un système, il est très important que cette spécification soit complète c.à.d. qu'elle couvre les trois perspectives de méta-modélisation : « données », « traitements » et « comportement ». En effet, une telle représentation permettrait de modéliser toutes les facettes du méta-modèle de processus et aurait le potentiel de pouvoir dériver un code exécutable correspondant à l'architecture d'un outil d'exécution de modèles de processus en appliquant une série de transformation aux modèles. Cette démarche aurait l'avantage de minimiser l'effort de programmation et d'échapper à l'implémentation manuelle (ad-hoc) de l'aspect dynamique.

Notre objectif est donc de contribuer au développement d'un outil d'exécution (qu'on appelle aussi moteur d'exécution) en proposant une démarche basée sur la méta-modélisation pour décrire son comportement interactif et fournir une spécification exécutable. Le problème lié à la réalisation de cet objectif réside dans la difficulté d'exprimer explicitement et d'exploiter la sémantique comportementale d'un modèle de processus. En effet, la spécification du méta-modèle de processus doit être assez détaillée et assez claire pour décrire la dynamique du modèle de processus mais en même temps elle ne doit pas être d'un niveau trop abstrait pour garantir l'exécutabilité du modèle.

Pour cela, il faut que :

- Cette spécification soit complète, c.à.d. elle doit couvrir les trois facettes de méta-modélisation des langages (« données », « traitements », et « comportement ») et elle doit

prendre en compte aussi bien l'aspect statique que l'aspect dynamique qui traduit la sémantique d'exécution du modèle de processus.

- Cette spécification se présente sous une forme orientée objet standard pour qu'elle puisse être interprétée facilement dans un environnement de développement qui supporte UML et qui génère du code à partir de diagrammes de classes.

Pour atteindre ces objectifs, nous proposons une démarche dirigée par les modèles qui consiste, comme l'indique la Figure 41, à :

- compléter le méta-modèle orienté objet utilisé pour représenter les perspectives « données » et « traitements » de l'outil par une spécification qui offre une vision dynamique afin d'obtenir une spécification conceptuelle qui couvre les trois perspectives de méta-modélisation, et
- définir des règles de transformation et les appliquer sur cette spécification afin d'obtenir de nouveau un schéma orienté objet prêt à être exécuté par l'un des outils de génération de code d'UML existants.

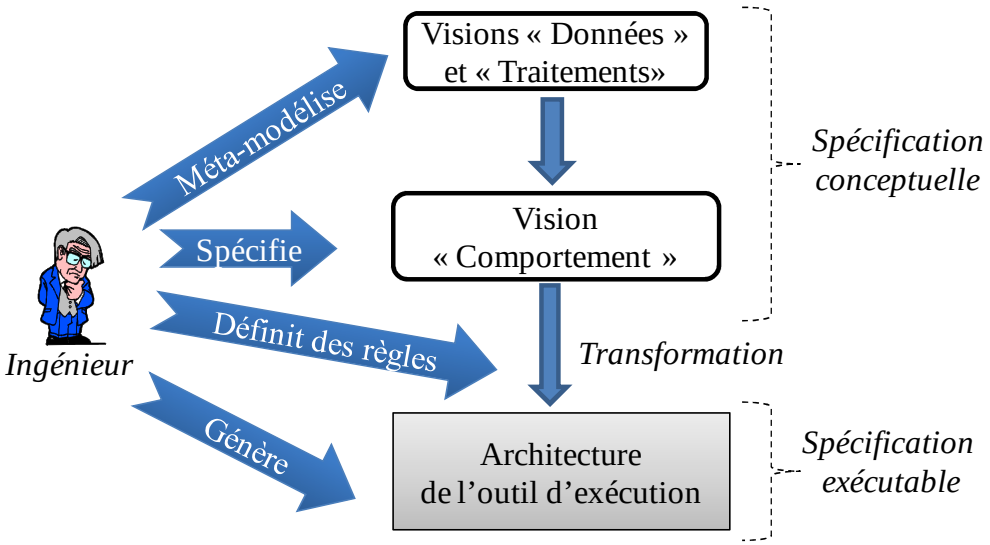


Figure 41. Un aperçu de la démarche proposée

Cette démarche s'inscrit dans une approche générale de construction d'outils. Cette approche est présentée dans la section suivante (Figure 42).

Ce chapitre est composé de 7 sections. La section 2 introduit l'approche globale dans laquelle s'inscrit notre proposition. La section 3 détaille notre proposition qui consiste en une démarche dirigée par les modèles pour la construction d'outils d'exécution. Cette démarche comporte trois parties qui feront respectivement l'objet des sections 4, 5 et 6. La section 7 précise la manière d'adapter la spécification orientée objet de l'architecture à une plateforme cible. Avant de conclure ce chapitre à la section 9, la section 8 présente les points forts de notre solution ainsi que son originalité.

4.2. Une approche IDM pour l'ingénierie d'outils d'exécution

Notre proposition a pour objectif d'intégrer plus d'automatisation dans le processus de construction d'outils d'exécution tout en assurant à la fin un produit d'une certaine qualité, à savoir un outil d'exécution interactif, facilement maintenable et portable et à moindre coût. Ce processus représente l'approche globale dans laquelle s'inscrit notre contribution.

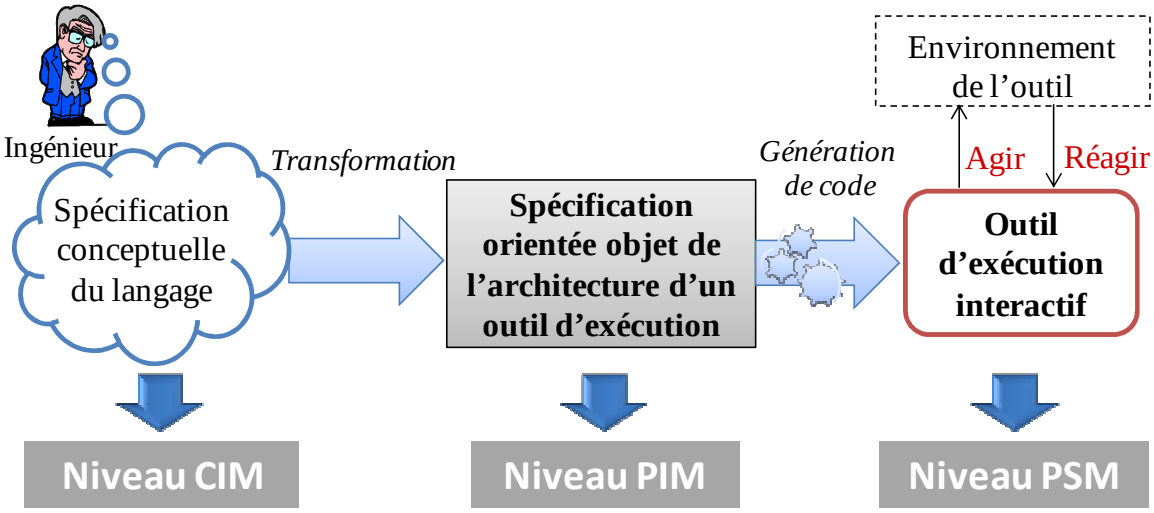


Figure 42. Le synopsis de notre proposition pour la construction d'outils d'exécution

La Figure 42 présente une vue d'ensemble de cette approche d'ingénierie d'outil d'exécution et son positionnement dans le cadre de l'architecture MDA. Cette approche consiste, tout d'abord, à définir une spécification conceptuelle complète du langage de modélisation pour lequel on veut construire un outil d'exécution. Cette spécification intègre la vision statique et dynamique des modèles, c.à.d. la structure et la sémantique du langage. Dans l'architecture MDA, cette étape correspond au niveau CIM ***Computational Independant Model*** et son rôle est de décrire l'application indépendamment des détails liés à son implémentation. Cette description n'est pas exécutable parce qu'elle est exprimée, dans notre cas, par deux formalismes différents exprimant chacun une perspective spécifique (le diagramme de classes pour la perspective « données » (statique) et « traitements » et le modèle événementiel pour la perspective « comportement » (dynamique interactive). Pour devenir exécutable, une transformation est appliquée sur la spécification conceptuelle permettant d'obtenir un modèle orienté objet qui correspond à l'architecture d'un outil d'exécution. Le résultat de cette étape correspond au niveau PIM (***Platform Independant Model)*** de l'architecture MDA. Une fois la spécification de l'architecture d'un outil d'exécution obtenue, on peut passer à la phase d'implémentation de cet outil sur une plateforme particulière. Cette étape correspond au niveau ***PSM (Platform Specific Model)*** du MDA et à ce niveau on se rapproche le plus du code final de l'application puisqu'on décrit l'implémentation de l'outil d'exécution sur une plateforme cible. Il est important de noter que dans cette approche d'ingénierie d'outils, la sémantique d'exécution est exprimée à l'aide du modèle dynamique (événementiel) et que grâce à cette méta-modélisation événementielle du comportement, il est possible de dériver un outil d'exécution qui aura la possibilité d'agir et de réagir avec son environnement.

Notre solution s'arrête à la phase de spécification conceptuelle et à la transformation de cette spécification pour la rendre exécutable sous forme d'un modèle UML, c.à.d. nous nous arrêtons au niveau ***PIM***. En effet, notre but est de proposer une spécification conceptuelle exécutable de l'outil d'exécution tout en étant indépendant d'une plateforme cible. Une fois cette spécification est obtenue, son implémentation peut se faire dans un outil UML existant de génération de code (tels que Eclipse, Netbeans, etc.).

4.3. Une vue détaillée de la démarche proposée

Notre proposition est schématisée à la Figure 43 selon trois points de vue différents: point de vue utilisation (par l'ingénieur méthode), point de vue conception et finalement, point de vue implémentation.

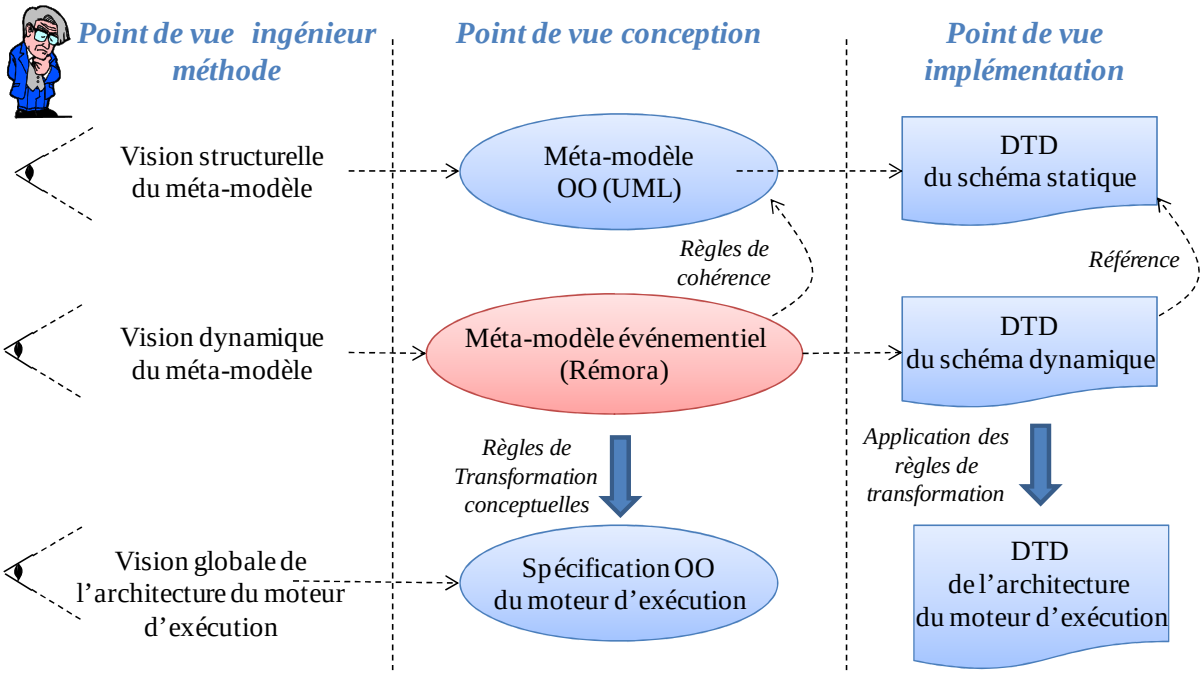


Figure 43. La vue détaillée de notre proposition

D'un point de vue ingénieur de méthode et de langage, la première étape de la démarche consiste à définir la spécification conceptuelle sous forme d'un méta-modèle de processus. Cette vision structurelle est completée avec les méthodes qui permettent d'accéder et de modifer les données du processus. A ce niveau, l'ingénieur méthode définit une spécification qui traduit les visions « données » et « traitements » du méta-modèle de processus. Cette spécification est représentée, au niveau conceptuel, par un méta-modèle orienté objet sous forme d'un diagramme de classes UML auquel va correspondre, au niveau implémentation, un schéma XML (ayant l'extension .xsd ou .dtd) appelé « XML-schema ». Nous ajoutons à cette étape également l'application de deux principes :

- la définition d'une structure à deux niveaux d'abstraction pour manipuler le niveau type (modèle de processus) et le niveau instances (processus ou/et traces d'exécution) en conformité avec le méta-modèle de processus choisi, et

- l'identification des attributs de type « état » pour gérer les états des instances des concepts du modèle de processus.

La deuxième étape de la démarche offre une vision dynamique de la spécification obtenue à la fin de la première étape. Cette étape consiste à représenter la sémantique d'exécution du méta-modèle de processus. Pour cela, on complète la spécification orientée objet par un schéma événementiel qui permet d'exprimer la sémantique d'exécution du méta-modèle de processus sous une forme déclarative et en mettant l'accent sur le fait que le modèle de processus doit interagir très fortement avec les éléments de son environnement tels que des humains, des systèmes, des applications, etc. A ce niveau de la démarche, l'ingénieur méthode doit s'assurer de la cohérence entre les deux méta-modèles aussi bien au niveau conceptuel qu'au niveau implémentation. Par exemple, il est important de s'assurer que les éléments du schéma dynamique référencent les élements de la spécification structurelle.

La troisième étape de la démarche vise à déduire de la spécification de la sémantique opérationnelle du méta-modèle de processus une description orientée objet de l'architecture d'un moteur d'exécution. Du point de vue conception, le résultat de cette étape est concrétisée par l'élaboration d'une spécification homogène de l'architecture du moteur d'exécution sous forme d'un diagramme de classes UML. Cette architecture est obtenue en appliquant des règles de transformation sur les spécifications résultant de la deuxième étape, et elle est représentée au niveau technique par un document XML-schéma.

Ainsi, la démarche de la proposition présente trois contributions principales : (i) la définition d'une spécification statique du modèle de processus à exécuter et la compléter par une structure d'instances ainsi que par des élements décrivant l'aspect « traitements » du méta-modèle de processus ; (ii) l'expression de la sémantique d'exécution du méta-modèle de processus par l'ajout d'une spécification comportementale avec une notation événementielle qui complète la spécification initiale en lui apportant une vision dynamique, et (iii) l'exploitation de la sémantique d'exécution par une transformation de la spécification obtenue vers une forme standard orientée objet afin d'obtenir une architecture de l'outil d'exécution qu'on peut implémenter dans un environnement existant de génération de code.

4.3.1. La 1ère étape : La spécification structurelle du méta-modèle de processus

L'expérimentation présentée au chapitre 2 a permis de mettre en évidence le besoin de compléter la syntaxe abstraite par une autre spécification pour prendre en compte l'aspect exécutable d'un modèle de processus. En effet, pour spécifier l'exécution un modèle de processus, il est nécessaire d'ajouter à sa spécification statique les informations dynamiques qui permettent de capturer les comportements possibles de ce modèle au cours de son exécution. Celles-ci doivent être définies à un niveau d'abstraction adéquat afin d'obtenir une spécification conceptuelle complète du méta-modèle.

a. La structure à deux niveaux

Le point de départ de cette première étape est la définition du méta-modèle statique du méta-modèle de processus que nous souhaitons opérationnaliser. Nous proposons de représenter ce méta-modèle sous forme d'un diagramme de classes UML que nous trouvons

bien adapté pour satisfaire l'objectif de cette étape de la démarche. En effet, un diagramme de classes est une collection d'éléments de modélisation statique correspondant à la structure d'un modèle. Une fois que la structure statique des concepts du méta-modèle est définie, nous la complétons par une structure de classes pour le niveau « instances », selon le proncipe présenté à la Figure 44, afin d'obtenir en tout une spécification à deux niveaux qui permet de représenter statiquement de manière générique les modèles à exécuter et les instances générées lors de leur exécution.

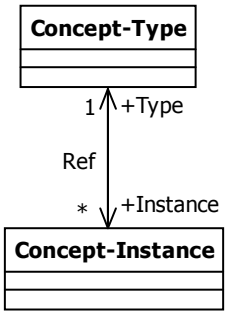


Figure 44. Le principe de dérivation de la structure d'instances à partir de la structure des concepts

Comme le montre la Figure 45, la sémantique opérationnelle est relative à un méta-modèle de processsus (M2) mais elle s'applique à n'importe quel modèle de processus (M1) conforme au M2 et le déroulement ou la simulation de l'exécution dirigée par la sémantique permet de contrôler l'exécution d'éléments du niveau processus (M0) en conformité avec le niveau M1 et M2. L'expression de la sémantique opérationnelle nécessite l'accès au modèle de processus (M1) et au niveau processus (M0) selon les termes du méta-modèle de processus (M2). Par conséquent, la hiérarchie d'instanciation à deux niveaux (M2-M1-M0) peut être plus communément retranscrite par une structure à deux niveaux de classes « type » et « instances » et une relation bidirectionnelle afin de relier :

- chaque objet du niveau « instance » à l'objet qui représente son type au niveau « type », et
- chaque objet du niveau « type » aux objets du niveau « instance » qui représentent ses instances.

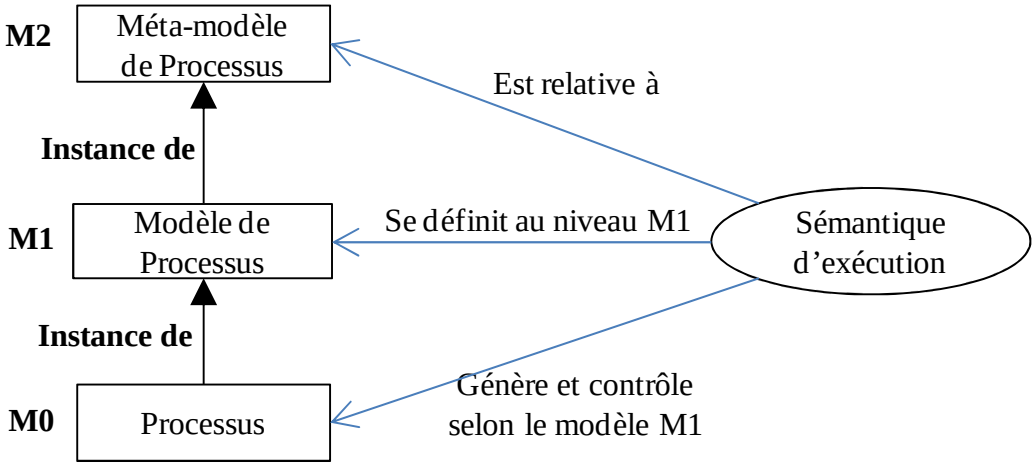


Figure 45. La sémantique d'exécution par rapport aux niveaux d'abstraction d'un modèle de processus

La structure de classes aux deux niveaux est, dans un premier temps, presque similaire puisque la structure de représentation des types (M1) et des instances (M0) est celle du méta-modèle de processsus (M2). Il faut donc dupliquer la structure de concepts du M2 : une pour représenter le niveau « type » et l'autre pour représenter le niveau « instance » en reliant les classes concernées par le lien bidirectionnelle « Ref ». Le niveau « type » doit inclure dans ses classes des opérations pour accéder aux informations (de type « get » et « set ») mais n'a pas besoin d'avoir d'opérations de modification car les objets de ce niveau ne seront jamais modifiés. Le niveau « instance », par contre, doit contenir des instances du méta-modèle qui traduit sa sémantique opérationnelle. C'est pour cette raison que nous complétons ce niveau « instance » avec les méthodes d'instances nécessaires qui constituent une partie de la sémantique opérationnelle du modèle de processus. Cependant, même si ces méthodes peuvent donner une idée sur le comportement d'un objet et par la suite sur le comportement du modèle de processus, leur présence dans ce modèle est restreinte à une expression syntaxique de la forme : « *Visibilité nomMéthode (nomParamètre : typeParamètre) : typeRetour* » où la visibilité indique qui peut avoir accès à la méthode.

b. Identification d'attribut de type « état »

Il est également important d'introduire un attribut particulier « **état** » sur certains concepts au niveau « instance ». L'importance de ce type d'attribut en SI est observée sur deux points :

- Cet attribut permet de déduire l'état d'avancement du processus d'exécution et ainsi déterminer à travers différentes valeurs qu'il peut prendre, le cycle de vie de ces concepts. Par exemple, si le cycle de vie d'une commande est composé des états suivants : créée, en attente, confirmée, annulée, expédiée, remboursée et terminée, alors, on peut savoir à un instant t quel est son état d'avancement dans son cycle de vie et savoir déduire quelles sont les activités futures autorisées.
- Dans le cadre d'une approche événementielle d'expression de la sémantique opérationnelle, les changements remarquables d'état d'objets vont permettre de déclencher des réactions nécessaires au contrôle de l'exécution du modèle de processus. En effet, dans cette approche, le changement d'état peut être vu comme un événement à partir duquel une réaction ou un traitement est attendu. Ceci permet de spécifier de manière plus déclarative les règles de comportement associées à un méta-modèle de processus.

Il est à noter que l'identification des méthodes et des attributs peut également avoir lieu durant l'étape suivante d'élaboration du schéma événementiel. En effet, la découverte d'événements va pouvoir permettre l'identficiation d'opérations ou d'attributs manquants dans le diagramme de classes à deux niveaux. L'idée est tout simplement de mettre en cohérence le schéma événementiel et le schéma structurel à deux niveaux avant de passer à l'étape suivante.

4.3.2. La 2ème étape : Compléter la spécification structurelle par une spécification du comportement du méta-modèle

A ce niveau de la démarche, nous nous trouvons avec un diagramme de classes à deux niveaux qui traduit **partiellement** la sémantique d'exécution du méta-modèle de processus considéré. Afin d'exprimer clairement et **explicitement** cette sémantique, et comme un diagramme de classes fait abstraction des aspects dynamiques, nous proposons de compléter cette spécification par une spécification du comportement du méta-modèle (Figure 46).

Il est important que la démarche conceptuelle se déroule de sorte à satisfaire certains critères. Dans notre cas, nous cherchons à exprimer la sémantique d'exécution de modèles de processus à forte interaction avec l'environnement. Pour satisfaire ce critère, nous avons choisi un formalisme orienté événement pour décrire le comportement du méta-modèle (sa sémantique opérationnelle). Ce formalisme capture la dynamique du processus d'exécution grâce à des notations basées sur les événements (Assar et al., 2011) et exprime ainsi l'interaction entre les différents éléments du processus, ainsi que l'interaction de l'exécution du processus avec des éléments de son environnement (par exemple : avec les applications extérieurs et les utilisateurs finaux). Traditionnellement, l'aspect interactif d'un modèle de processus n'est pas pris en compte dans la spécification conceptuelle et encore moins de manière explicite.

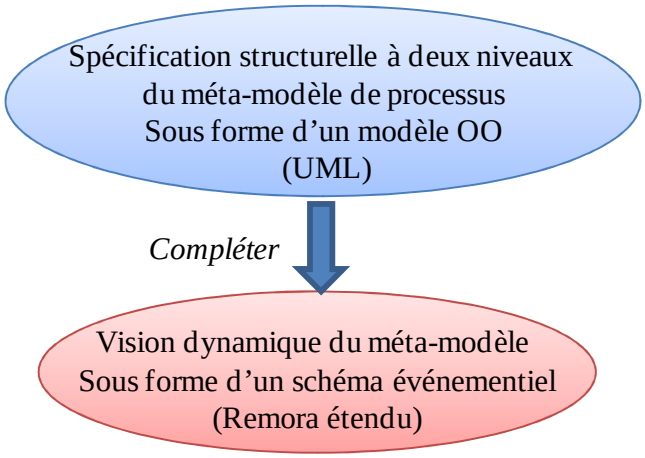


Figure 46. L'expression explicite de la sémantique d'exécution grâce au modèle événementiel

La construction du schéma dynamique qui vient compléter la spécification à deux niveaux du méta-modèle de processus se fait en plusieurs itérations. Il s'agit d'un processus itératif qui exige plusieurs allers-retours afin de s'assurer de la complétude et de la cohérence des modèles spécifiant l'aspect statique et dynamique du méta-modèle de processus à opérationnaliser.

En partant de la spécification structurelle du méta-modèle de processus, la construction du schéma dynamique correspondant nécessite plusieurs confrontations entre le schéma dynamique et la structure à deux niveaux tout en examinant un à un les objets et les opérations pour s'assurer de la cohérence des deux spécifications (pas d'objet/opération en plus ni en moins). Pour mieux guider le concepteur dans ce processus, nous proposons quelques guidelines :

G1 : Partant du diagramme de classes de la spécification structurelle à deux niveaux, associer à chaque classe pertinente un objet portant le même nom dans le schéma Remora étendu.

G2 : Chaque opération d'une classe est représentée par une flèche qui pointe sur l'objet correspondant à cette même classe.

G3 : Pour chaque objet de la structure à deux niveaux, définir l'ensemble des événements et des messages qui l'accompagnent. Cette étape se base sur la compréhension du comportement du modèle de processus à exécuter.

G4 : L'acteur est une entité qui intéragit avec le modèle de processus en émettant des évenements et recevant des messges, elle doit donc être représentée dans le shéma dynamique.

L'ajout du schéma dynamique à la spécification structurelle à deux niveaux permet d'obtenir une spécification complète du méta-modèle (c-à-d qui couvre les trois perspectives de méta-modélisation). Cet ajout combine les avantages de la modélisation conceptuelle événementielle aux avancées de l'approche orientée objet. De cette façon, nous avons pu enrichir sémantiquement le diagramme de classes UML et profiter des atouts supplémentaires à la modélisation orientée objet qui ne permet pas une modélisation complète et performante des aspects dynamiques et événementiels.

Dans notre solution, nous sommes partis de l'hypothèse que la combinaison de la modélisation conceptuelle événementielle, qui a fait ses preuves depuis des années, avec l'approche orientée objet est intéressante. Cette hypothèse a été vérifiée d'abord, parce que cette combinaison permet d'abstraire une description conceptuelle du modèle de processus dans laquelle il est vu comme un système qui interagit avec son environnement au travers des événements. Ensuite, parce que cette description sert de support non ambigu à l'équipe de conception technique en évitant les retours en arrière toujours coûteux et sources d'erreurs.

4.3.3. La 3ème étape : Transformation pour obtenir une architecture orientée objet de l'outil d'exécution

La spécification obtenue jusque là est formée d'un diagramme de classes complété par un schéma événementiel de type Remora étendu. Cette spécification représente une expression de la sémantique opérationnelle du modèle de processus mais elle ne représente pas encore l'architecture de l'outil d'exécution. Il manque dans cette spécification une spécification de la gestion globale des objets du niveau « type » et les objets du niveau « instance » et des éléments de l'environnement qui doivent être configurés physiquement pour être capable d'interopérer avec eux durant l'exécution du processus en fonction du modèle de processus.

De plus, la sémantique opérationnelle est représentée par un diagramme de classes et un schéma événementiel ; celui-ci est très puissant pour décrire de manière synthétique le comportement mais il n'est pas standard et ne pourra pas être facilement représentable dans une architecture orientée objet standard de l'outil d'exécution. Pour qu'elle revienne à une forme standard et interprétable par les outils existants de génération de code, nous proposons de la transformer en une spécification homogène ayant une forme orientée objet standard

(diagrammes de classes et diagrammes de séquence UML). L'étape de transformation consiste à appliquer un ensemble de règles permettant le passage de la spécification conceptuelle spécifique vers une spécification technique sous une forme orientée objet standard et en introduisant également la classe globale de l'outil avec les fonctionalités nécessaires pour que cette formalisation orientée objet puisse être considérée comme l'architecture objet de l'outil d'exécution.

Les règles de transformation pour traduire les concepts de Remora étendu en classes d'objets que nous avons définies sont basées sur les patrons conceptuels de messagerie publish/subscribe qui permettent de transcrire la sémantique d'exécution du méta-modèle de processus décrite dans le schéma dynamique. Le résultat de l'application de ces règles de transformation est une architecture orientée objet de l'outil d'exécution de modèles de processus. Cette architecture est indépendante d'une plateforme technique mais sa forme standard facilite son implémentation dans les environnements de développement basés sur UML.

Dans la suite de cette section, nous allons détailler les différentes étapes de la démarche de notre proposition à savoir :

- Etape1 : La spécification structurelle du méta-modèle
- Etape2 : La spécification graphique de la sémantique d'ecécution du méta-modèle
- Etape3 : La transformation des spécifications obtenues vers une architecture orientée objet du moteur d'exécution.

4.4. Etape 1 : La spécification structurelle du méta-modèle de processus

Le modèle statique est réalisé avec le méta-modèle de classes UML. L'objectif de cette étape est de faire de la méta-modélisation de l'outil d'exécution à construire et de ressortir sa structure en termes de classes d'objets et de liens entre ces classes. La méta-modélisation est une clé de la mise en œuvre réutilisable, elle est particulièrement intéressante pour les outils complexes qui sont destinés à des utilisations variées.

Nous pensons que le diagramme de classes est bien placé pour accomplir cet objectif. D'abord, parce que ce diagramme permet de donner une représentation statique des éléments qui composent un système (modèle de processus dans notre cas) et de leurs relations. Ensuite, parce que l'opérationnalisation de l'aspect statique décrit par un diagramme de classes ne pose généralement pas de gros problèmes; il s'agit de transposer des représentations plus ou moins graphiques en leur équivalent textuel dans un langage de programmation (exemple : des définitions de classes dans un langage à objets). Chaque langage de programmation orienté objets donne un moyen spécifique d'implémenter le paradigme objet (pointeurs, héritage multiple, etc.), mais le diagramme de classes permet de modéliser les classes du système et leurs relations indépendamment d'un langage de programmation particulier. Afin que les classes et les relations utilisées dans le diagramme de classes puissent servir de base à la génération du code d'une application, il est essentiel de respecter au maximum les chartes de nommage ainsi que les bases de syntaxe des langages informatiques (Roques, 2009).

4.4.1. Le méta-modèle de diagramme de classes

La Figure 47 présente le méta-modèle de diagramme de classes sous forme d'un diagramme de classes UML. Le concept central de ce méta-modèle est celui de classe (class).

La classe est un concept abstrait qui permet de représenter toutes les entités d'un système. Elle peut être le type d'un attribut qui peut aussi être de type entier (integer), chaine de caractères (string) ou booléen (boolean). Le concept classe est défini par son nom, un ensemble d'attributs et un ensemble d'opérations.

L'association représente une relation entre deux classes. Elle spécifie tout simplement qu'une classe peut en utiliser une autre. L'association est habituellement traduite dans le code par un pointeur. Une association est définie par ses deux terminaisons (*associationEnd*) qui ont chacune:

- Un rôle : C'est le nom que prendra l'attribut de la classe.
- Une multiplicité : Pour définir des tableaux, des listes de valeurs
- Une navigabilité : Pour rendre l'association bidirectionnelle ou monodirectionnelle.
- Une visibilité : + public / # protected / - private.

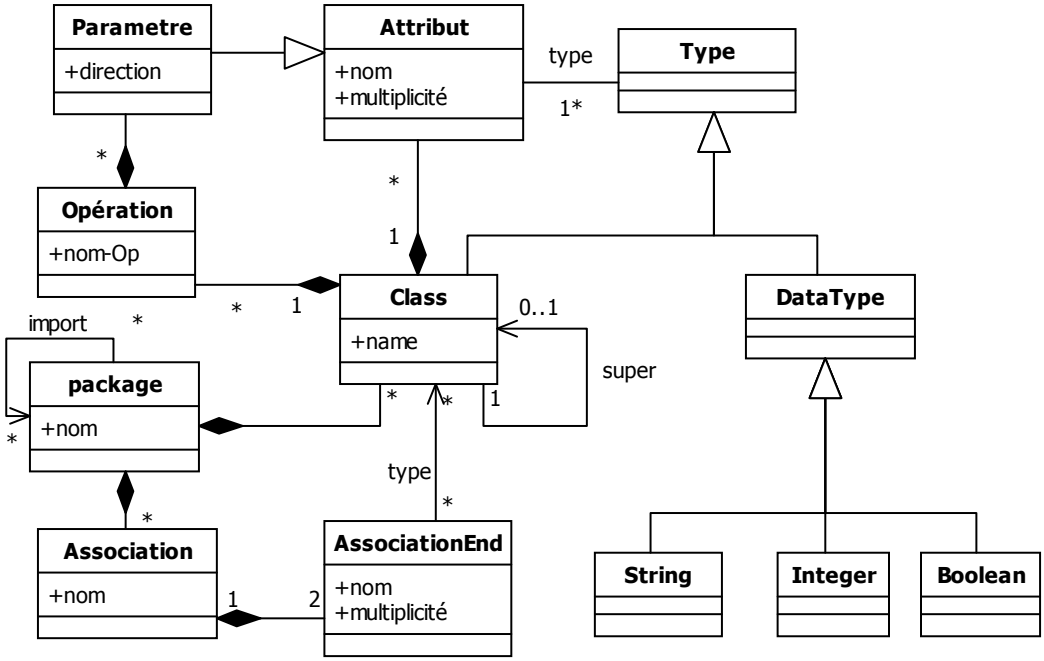


Figure 47. Le méta-modèle de diagramme de classes UML (Blanc, 2005)

Ce méta-modèle est largement suffisant pour décrire l'aspect statique de l'outil d'exécution que nous souhaitons construire.

4.4.2. Exemple d'un diagramme de classes UML

Afin d'illustrer la spécification structurelle (« données » et « traitements »), nous présentons, à la Figure 48, le méta-modèle de workflow sous forme de diagramme de classes UML. Un workflow est défini comme la représentation sous forme de flux d'opérations à

réaliser par des agents (WF_Participant) pour accomplir un ensemble de tâches ou d'activités (WF_Activité) regroupées en un même processus métier (WF_Processus).

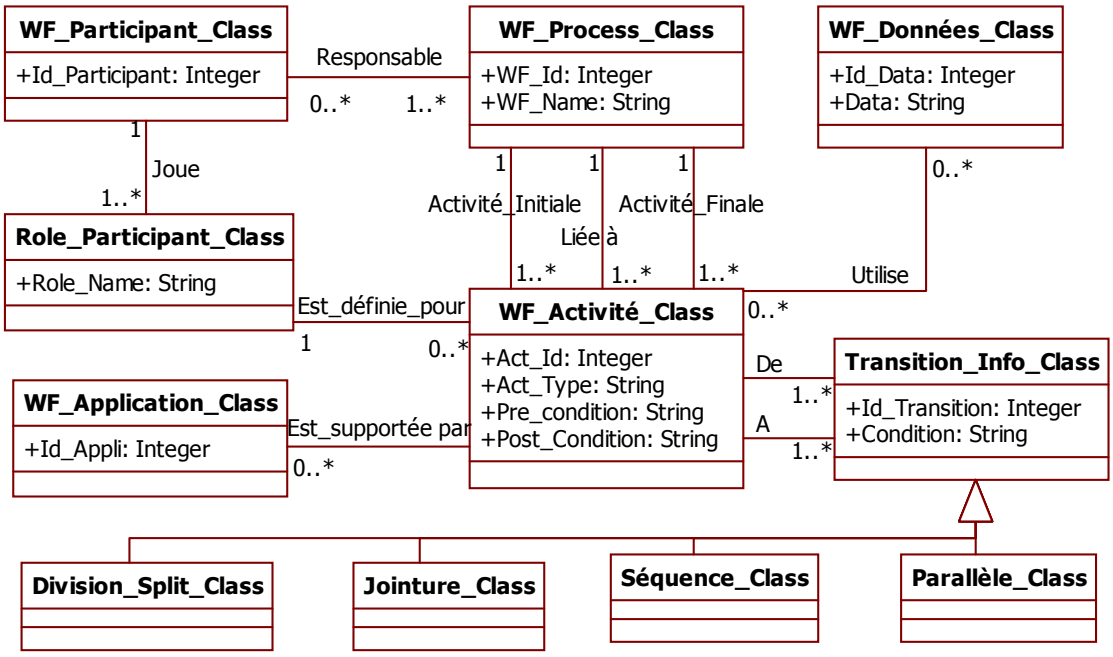


Figure 48. Le diagramme de classes simplifié du méta-modèle de Workflow

La Figure 49 présente l'architecture à deux niveaux du modèle de workflow sous forme d'un diagramme de classe UML qui décrit la structure du modèle en termes de classes. Pour des raisons de lisibilité du schéma, nous n'avons pas représenté, au niveau des classes-types, les classes spécifiques qui héritent de la classe Transition_Info_Class. Il est à noter que les associations nommées « Ref » à la Figure 49 entre une classe du niveau « instance » et une classe du niveau « type » correspondent à l'association « type-instance » présentée à la Figure 44. Pour des raisons de lisibilité de la figure, nous n'avons pas mentionné l'ensemble des rôles constituant l'association.

Dans le niveau en haut, on représente les classes du niveau « type » du modèle de processus étudié (workflow). Chacune de ces classes est définie par un nom, des attributs, et des opérations privées de type « get » et de type « set » que nous ne représentons pas dans notre exemple par crainte d'encombrer la figure. Dans le niveau d'en bas, on définit la structure des instances formée à l'aide des classes objets.

Comme le montre la Figure 49, chaque classe du niveau « instance » est reliée à la méta-classe correspondante dans le niveau « type » par une association bidirectionnelle nommée « *Ref* » ayant deux rôles : « type » et « instance ». Cette association se traduit au niveau implémentation, d'une part, par un lien de référence qui est manipulée à travers une méthode appelée getClass() qui permet aux classes instances de récupérer la description de leur classe du niveau « type ». D'autre part, chaque classe des concept-type inclut la définition d'une collection de références sur ses instances avec un ensemble de méthodes pour manipuler la collection d'instances. Les associations entre les classes du niveau « type » sont dupliquées entre les classes instances pour permettre la navigation entre les objets du niveau « instance ».

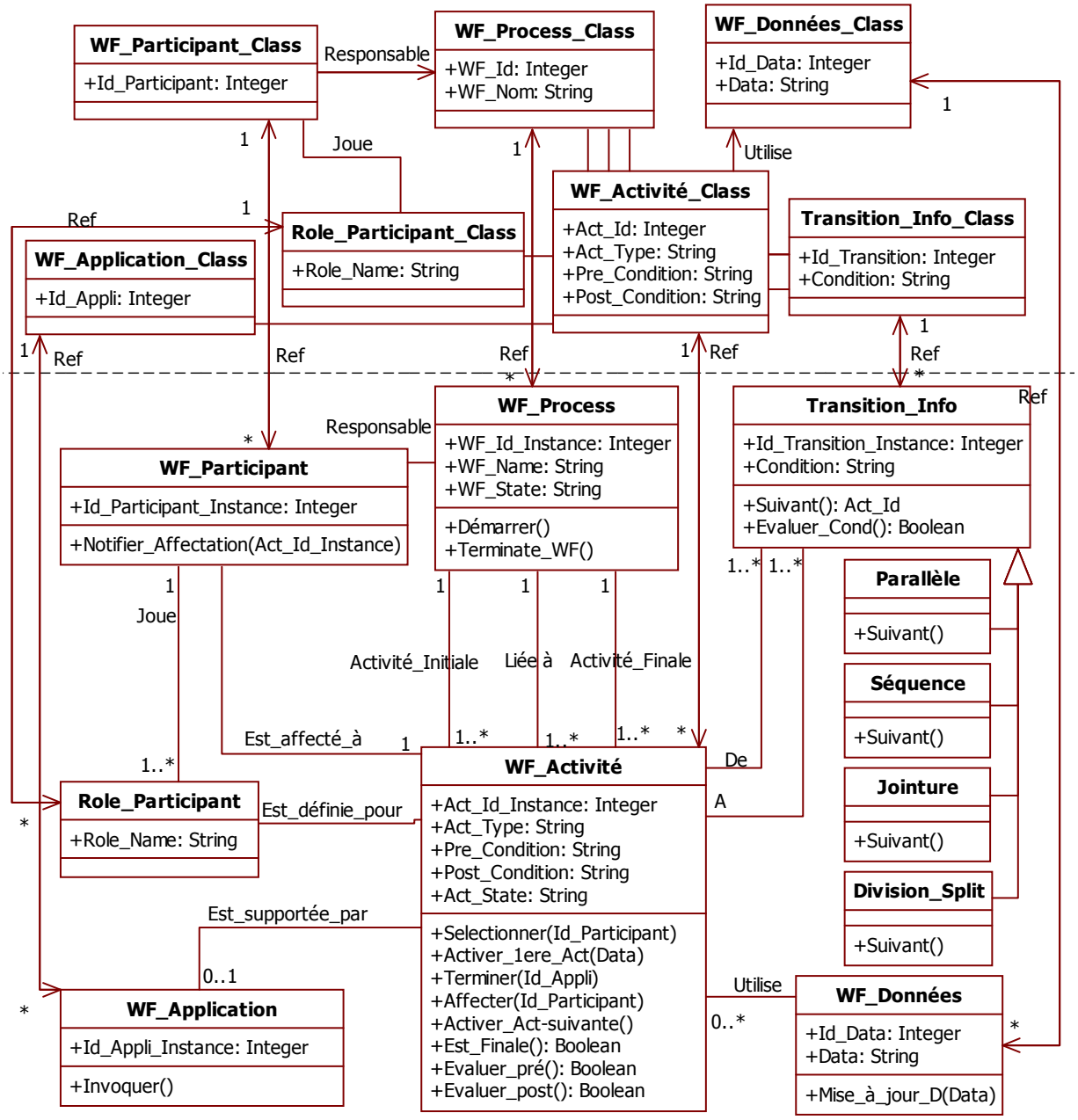


Figure 49. L'architecture à deux niveaux du méta-modèle de Workflow

La définition de la structure des instances ne se fait pas d'emblée, elle nécessite, dans un premier temps, l'ajout d'éléments de méta-information nécessaires à l'exécution (tel que l'attribut « Act_State » dans la classe WF_Activité). Dans un deuxième temps, des méthodes sont ajoutées aux classes instances pour identifier les différentes opérations nécessaires modifiant les objets du niveau « instance ». Par rapport à l'orientation objet, il est à noter que dans le modèle événementiel Remora étendu, une opération ne doit changer qu'une seule instance d'objet, par contre les méthodes de type accesseur doivent être également identifiées ; elles peuvent accéder à plusieurs objets mais n'en modifient aucun. Plus tard, les opérations externes (qui sont destinées à des acteurs externes) sont ajoutées au moment de la définition du schéma dynamique de manière à prendre en compte les opérations qui envoient des messages aux acteurs constituant l'environnement du modèle de processus. Dans certains cas, on peut même ajouter de nouvelles classes de type « Concept-instance » si le cas l'exige,

ces ajouts se font par le concepteur au fur et à mesure du processus de la spécification du comportement du modèle de processus c.à.d. au moment de la définition du schéma dynamique correspondant.

4.5. Etape 2 : La spécification graphique de la sémantique d'exécution

Durant l'étape de la spécification dynamique d'un modèle de processus, le concepteur est amené à traduire la sémantique d'exécution du langage qui a servi à modéliser ce processus. Pour cela, nous avons choisi d'exprimer cette sémantique à l'aide d'un schéma dynamique décrivant graphiquement le comportement du modèle de processus en se basant sur un formalisme orienté événement appelé Remora (Rolland et al., 1988). L'idée principale de cette étape est d'élargir le modèle orienté objet UML par les concepts événementiels de Remora pour formaliser la sémantique opérationnelle du méta-modèle de processus. L'avantage de cet élargissement du champ d'expression du méta-modèle est qu'il permet de représenter l'aspect dynamique du modèle et d'en capturer sa sémantique afin d'obtenir une spécification plus complète de langage de modélisation du processus incluant le comportement du modèle et ses interactions avec des éléments externes. Le modèle Remora est décrit plus en détail dans la section suivante.

4.5.1. Pourquoi le modèle Remora ?

Lorsqu'on s'intéresse à l'exécution d'un méta-modèle en IDM, on est amené à penser au langage de méta-modélisation qui sera utilisé pour représenter le comportement du méta-modèle. Il existe plusieurs formalismes et notations candidats à capturer cette représentation, à savoir les réseaux de Petri, les diagrammes de séquence UML, les automates, les workflows, les bases de données relationnelles, etc. Certains modèles sont adaptés à un type de système plus que d'autres. Par exemple, un workflow est très utile pour représenter des systèmes à base d'activités. Les automates sont utilisés pour les systèmes à caractère automatique (exemple : fonctionnement d'une montre).

D'après nos recherches, nous avons remarqué que pour représenter l'aspect dynamique d'un système à forte interaction avec des acteurs extérieurs, il est très pertinent d'utiliser un modèle à base d'événement. Le paradigme événementiel est très utilisé pour décrire les systèmes réactifs et asynchrones, et nous semble très approprié pour capturer la vision systémique d'une exécution de processus en fonction d'un modèle de niveau M1 et de niveau M2 qui doit réagir et interagir avec son environnement. Il n'est cependant pas présenté comme un concept central des langages tels qu'UML ou réseaux de Pétri. Un tel paradigme événementiel permettrait de décrire quelles opérations sont déclenchées suite à quels événements. Ainsi le déroulement de l'exécution du modèle de processus ne se préoccupe pas de l'ordre dans lequel les différents acteurs interagissent avec lui. Cette flexibilité est exigée dans certains modèles, tel que celui de la carte Map. Ceci nous a amené à choisir le formalisme dynamique de la méthode Remora pour ces différentes raisons synthétisées :

- sa concision (peu de concepts) ce qui facilite l'étape suivante de l'exploitation de la sémantique,

- son expressivité pour la vision systémique : ceci permet d'avoir une représentation plus globale du système dans son environnement et de prendre en compte des interactions entre le modèle de processus, les acteurs et les applications externes avec qui il communique (événement et échange de messages),
- sa sémantique claire et bien définie ainsi que les possibilités d'implémentation : cette sémantique a déjà été exprimée dans des travaux antérieurs (Cauvet et al., 1989), (SAI PECK, 1994), et elle peut être implémentée dans différents langages de programmation.

4.5.2. Le modèle événementiel

Remora est un formalisme de modélisation orientée événement. Il représente la base d'une méthode événementielle pour la modélisation systémique. Déjà, cette dernière est de grand apport dans le domaine de l'ingénierie des SI puisque cette modélisation systémique permet de représenter les interactions d'un système qui agit et réagit avec d'autres systèmes. Dans le cadre de la modélisation systémique, l'approche événementielle permet de représenter à la fois l'aspect statique du système et l'aspect dynamique (événements, changement d'état, actions, etc.). On peut noter que Merise/2, ou les méthodes à objets comme OMT ou UML ou O* (Cauvet et al., 1989) (Colette Rolland, 1991), reconnaissent implicitement la pertinence de cette orientation et s'en sont inspirés. Les principaux atouts du formalisme Remora sont la simplicité (peu de concepts), l'extensibilité, la représentation graphique de modèles (voir la Figure 50) et la description formelle. En effet, ce formalisme a été implémenté par l'outil Rubis (Lingat, 1988), ce qui montre qu'il a une sémantique claire et exécutable, ce qui est un critère nécessaire pour la spécification de modèles exécutables.

Le langage de modélisation événementielle Remora connu dans l'ingénierie des SI est utilisé ici pour spécifier la sémantique opérationnelle d'un méta-modèle de processus. Il offre, d'une part, une représentation graphique lisible et précise, et d'autre part, dispose d'une sémantique claire et bien définie qui prend en compte la nature indéterministe de l'exécution d'un processus orienté intention tel que le modèle de la carte Map.

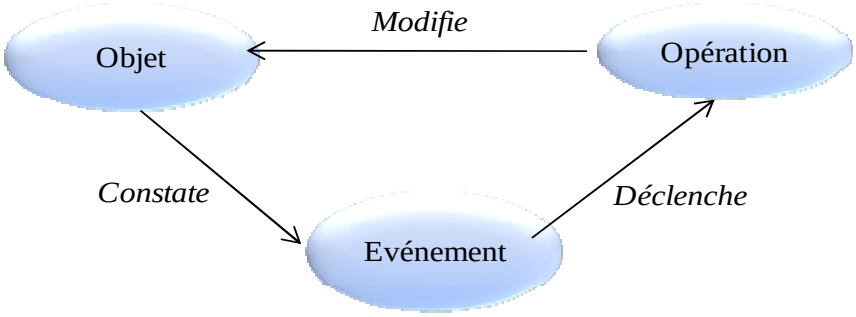


Figure 50. Le modèle d'événement Remora

Remora est un formalisme événementiel simple qui se base sur trois concepts comme le montre la Figure 50.

L'objet : c'est une représentation informationnelle d'un constituant abstrait ou concret du monde réel. Il existe indépendamment des transformations qu'il peut subir ou du rôle qu'il

peut jouer. Un objet est définit par un nom et possède un état. L'état d'un objet est l'ensemble des valeurs qui le caractérisent et qui forment son cycle de vie.

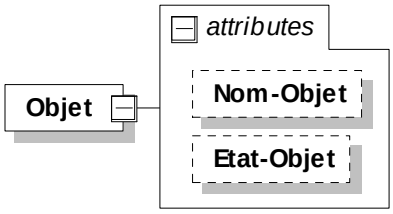


Figure 51. La structure du concept Objet

L'opération : c'est une action qui peut être exécutée dans un système en réaction à un événement et dont l'effet sur le système est décrit par une collection d'actions élementaires modifiant les objets du système. Une opération est exécutée par un acteur (un homme ou une machine) dans un lieu donnée, à un instant donné, et dure un certain laps de temps. Cette exécution se traduit parfois par la création de nouveaux objets et parfois par la modification de l'état d'un objet.

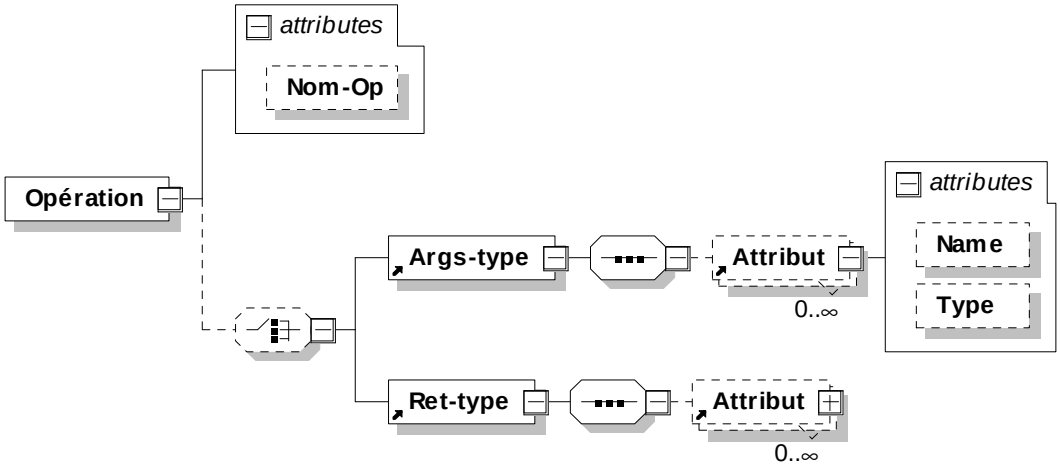


Figure 52. La structure du concept Opération

Il est à noter que, dans le schéma dynamique Remora, seules les opérations modifiant l'état des objets sont identifiées ; Les accesseurs (les getters) ne sont pas représentées dans la dynamique. Aussi, une opération modifie l'état que d'un seul objet d'une même classe.

L'événement : c'est une projection d'un phénomène du monde réel qui survient dans le système, à un instant donné, provoquant un changement d'état d'objets à travers l'exécution d'opérations. La durée de vie d'un événement est limitée à la période comprise entre l'instant où il survient et l'instant où ses conséquences immédiates sont effectives. Les occurrences d'un événement sont représentées par le concept *Evénement* qui est caractérisé par un nom, un type (interne ou extérieur), un prédicat exprimant quand il se produit et un corps décrivant un flux d'opérations à exécuter lorsque l'événement se déclenche. Un *événement interne* constate le changement d'état remarquable d'un objet du système alors qu'un *événement externe* a pour origine un stimulus en provenance de l'environnement du système, il est associé à un message.

Extension de Remora avec le concept Acteur : Le modèle Remora de la Figure 50 a été étendue dans le cadre de la méthode O* de conception orientée objet et événement (Cauvet et

al., 1986) (Roland, 2005b) par le concept Acteur. En effet, dans cette proposition, la description d'un événement est encapsulée dans une classe qui dépend du type de cet événement :

- un *événement interne* est défini dans la classe de l'objet dont il constate un changement d'état,
- un *événement externe* est défini dans une classe de type Acteur. Il peut y avoir plusieurs classes de ce type dans un schéma, une pour chaque agent qui interagit avec le système. Plusieurs événements externes peuvent être attachés à la même classe acteur, tous ceux qui sont sous le contrôle d'un même acteur.

Dans notre travail, nous avons adopté cette version étendue du modèle événementiel parce que la présence du concept acteur est nécessaire pour la spécification du comportement interactif d'un modèle. La structure détaillée du concept Evénement sera présentée dans la sous-section 4.5.2.3.

Dans la suite de cette section, nous allons présenter le modèle conceptuel pour la spécification de la sémantique opérationnelle d'un méta-modèle de processus. Mais juste avant, nous allons introduire les concepts de ce modèle en l'illustrant par des exemples.

4.5.2.1. La représentation graphique associée au modèle de comportement

Le modèle événementiel a été choisi pour sa simplicité des concepts permettant d'obtenir des schémas dynamiques facilement compréhensibles. A travers un nombre limité de concepts, il est possible de décrire un comportement complexe d'un système soumis à de nombreux stimuli extérieurs et de nombreuses rétroactions. La description orientée événements permet d'exprimer à la fois les raisons pour lesquelles les objets évoluent, comment ils évoluent (en réponse à un stimulus de l'environnement ou bien à un changement d'état d'un objet) et les dépendances existantes entre les cycles de vie des objets. Le choix de ce modèle événementiel revient aussi à sa représentation graphique car, par rapport à une spécification de type méta-programmation (tel que Kermeta) ou purement déclarative (tel qu'avec ConceptBase), la représentation graphique qui est associée à ce modèle de comportement apporte un complément indéniable dans la mise en évidence des points d'interaction entre le processus qui s'exécute et son environnement.

La notation graphique du modèle Remora est présentée à la Figure 53. D'après cette figure, la spécification du processus se modélise avec le concept «*objet*» représenté graphiquement par une forme ovale contenant le nom de l'objet, le concept « *événement* » représenté par un triangle, et le concept « *opération* » représenté par une flèche qui relie l'événement déclenchant et l'objet qui sera modifié suite à cet événement. Ces concepts sont utiles pour représenter le comportement dynamique interne du modèle de processus. En effet, un événement provoque un changement d'état d'un ou plusieurs objets à travers l'exécution d'opérations. On représente cela par la relation « *déclenche*» entre le concept événement et le concept opération. La relation « *constate* » est définie à partir d'un message d'un agent ou une classe d'objet à un événement.

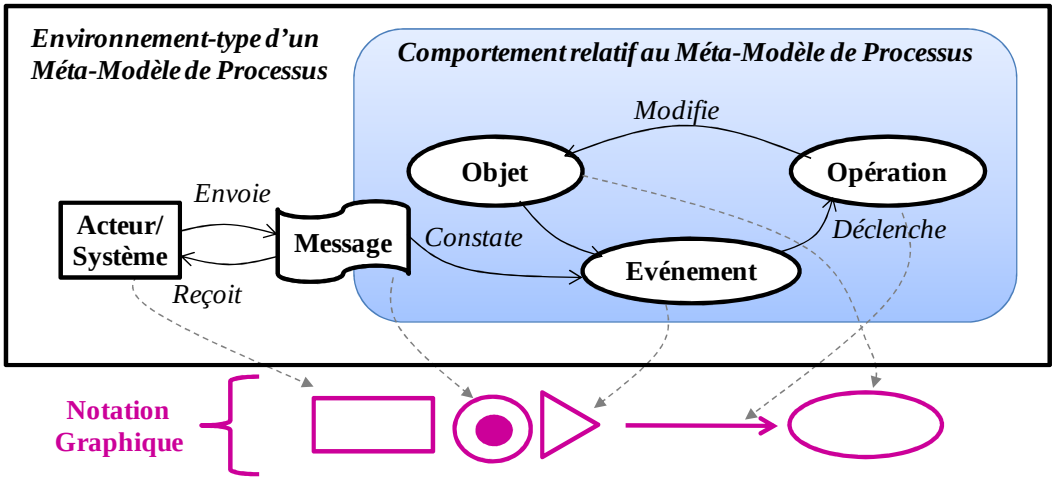


Figure 53. Le modèle événementiel et sa représentation graphique

A part ces concepts qui sont les fondements d'un modèle événementiel, on note la présence du concept « *acteur* » qui peut être un agent humain ou un système/application et qui est représenté graphiquement par un rectangle. Ce concept représente les entités qui se trouvent dans l'environnement du processus et qui communiquent à l'aide de messages représentés graphiquement par un rond noir dans un petit cercle. La réception des messages produits ou envoyés par un acteur de l'environnement du processus représente l'événement externe qui agit en déclechant des traitements pour continuer le contrôle de l'exécution du processus. Nous verrons par la suite que certains traitements déclenchés en réaction à un événement peuvent produire des messages qui sont envoyés à des acteurs consituant l'environnement du processus.

4.5.2.2. Définitions des concepts de la spécification dynamique

Dans la suite de cette section, nous allons détailler les concepts particuliers qui vont servir à la gestion du comportement événementiel et montrer l'intérêt de les intégrer dans un cadre de méta-modélisation. Pour illustrer chacun de ces concepts, nous nous basons sur le méta-modèle de workflow (présenté à la Figure 48).

4.5.2.3. Le concept Evénement

L'événement est le concept clé de l'approche événementielle que nous adoptons dans notre solution. Cette approche combine le concept d'événement à celui d'objet, afin de permettre une modélisation des aspects dynamiques liés à la causalité (inférence d'événements) et à la concurrence (synchronisation d'événements ou d'opérations) qui ne sont en général pas complètement et/ou correctement pris en compte dans les approches orientées objet classiques (Rolland et al., 1988).

Nous avons choisi d'introduire ce concept au niveau du langage de méta-modélisation parce que nous sommes convaincus de l'importance de spécifier les aspects dynamiques, explicitement, durant l'étape de méta-modélisation conceptuelle. L'aspect dynamique ne concerne pas uniquement le comportement des éléments de la spécification du modèle de processus, mais aussi l'interaction de ces éléments avec l'environnement du processus. Pour cela, deux types d'événements existent: externe et interne.

Représentation d'un événement :

Qu'il soit interne ou externe, le concept d'événement est décrit par son identifiant (Id-Event), son nom (Nom-Event), la description de son prédicat (Prédicat-Event), et la description du trigger qu'il déclenche et qui informe sur l'ensemble des opérations déclenchées ainsi que les facteurs et les conditions de déclenchement.

Afin de représenter en XML les structures de la spécification événementielle, nous avons utilisé un outil particulier appelé ALTOVA XMLSPY[37]. Cet outil permet de définir, avec une excellente ergonomie, des fichiers XML schema qui représentent la version améliorée des DTD (pour Document Type Definition), et d'en dériver des documents XML. Aussi, cet outil donne la possibilité de présenter les modèles dans différentes formes (graphique, textuelle, grille, WSDL, Browser, etc.). Ainsi, la forme graphique de la structure d'un événement est présentée comme suit à la Figure 54.

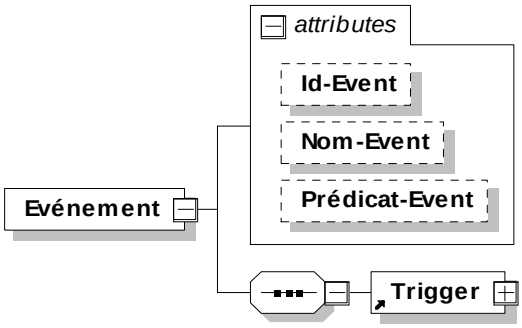


Figure 54. La forme graphique de la structure du concept Evénements

La représentation textuelle du concept événement interne est présentée à la Figure 55.

```
<xs:element name="Evénement">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="Trigger"/>
                </xs:sequence>
                <xs:attribute name="Id-Event"/>
                <xs:attribute name="Nom-Event"/>
        <xs:attribute name="Prédicat-Event"/>
        </xs:complexType>
</xs:element>
```

Figure 55. La forme textuelle de la structure du concept d'Evénement

Cette structure de base d'un événement est encapsulée dans la description des événements de type interne et externe.

a. L'événement interne

Un événement interne se produit à la suite du changement d'un état particulier ou d'une valeur d'une propriété d'un objet. Quand un événement interne se produit, un ensemble d'opérations doit être déclenché et ces opérations induisent des changements d'état d'objets qui peuvent donner lieu quelque fois à la constatation de nouvelles occurrences d'événement

[37] http://www.altova.com/xml-editor/

interne. La définition d'un événement interne inclut, par rapport à la structure de base d'un événement, une référence vers l'objet sur lequel est constaté l'événement.

Représentation d'un événement interne:

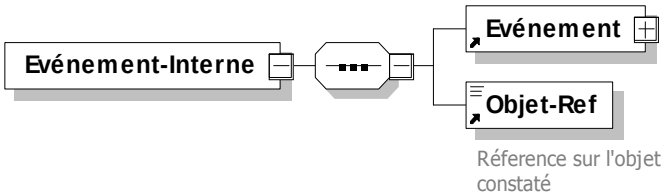


Figure 56. La structure d'un événement interne.

Le prédicat d'un événement interne est une expression booléenne spécifiant formellement le changement d'état de la propriété sur laquelle est défini l'événement interne. Un événement interne survient quand le résultat de l'évaluation du prédicat est vrai. Dans l'expression textuelle du prédicat, les mots clés "OLD" et "NEW" servent à qualifier respectivement l'état 'précédent' et l'état 'nouveau' de l'objet sur lequel l'événement est constaté. Graphiquement, cette expression peut figurer dans le schéma dynamique en tant que note à coté du triangle qui représente l'événement interne.

Exemple d'événement interne:

L'exemple suivant, extrait de la spécification dynamique du moteur de workflow, montre l'événement interne *EV5* relatif à un changement d'état remarquable d'un objet de type *WF_Activité.*

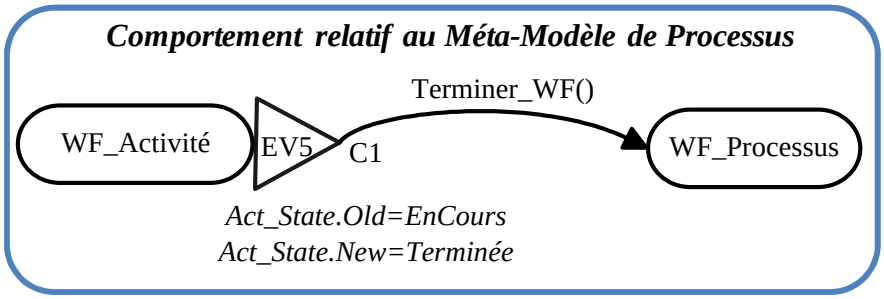


Figure 57. Un exemple d'événement interne

Dans cet exemple, la valeur de la propriété *Act_State* de cet objet, passe de « En cours» d'exécution à « Terminée ». Dans le cas où l'activité est finale (condition C1 est vraie), la fin d'exécution du processus est détectée d'où l'exécution de l'opération « *Terminer_WF()* » sur l'objet *WF_Processus*. Il s'agit d'un événement interne car l'objet constaté est un objet interne de la spécification du modèle de processus et en l'occurrence *WF_Activité*.

La représentation textuelle de l'événement interne EV5 est présentée à la Figure 58.

```
<Evénement-Interne>
        <Objet-Ref> WF_Activité</Objet-Ref>
        <Evénement
          Id-Event="EV5"
          Nom-Event="Fin d'exécution d'une activité"
          Prédicat-Event = " Act_State.Old='EnCours' et  Act_State.New= 'Terminée' ">
                <Trigger >
                        <!—Déclenchement des opérations relatives à événement EV5-->
```

```
                </Trigger>
        </Evénement>
</Evénement-Interne>
```

Figure 58. La représentation textuelle de l'exemple d'un événement interne

Les spécifications de la condition et du trigger seront présentées à la section 4.5.2.5.

b. L'événement externe

L'événement externe est le concept qui permet de représenter l'action de l'environnement sur le déroulement et le contrôle du processus selon une vision systémique. L'interaction du processus avec son environnement extérieur est ainsi représentée grâce aux événements externes. Un événement externe survient à l'arrivée d'un stimulus envoyé par un acteur de l'environnement. L'acteur décrit le stimulus sous la forme d'un message informationnel.

Le prédicat, dans le cas d'un événement externe, ne se constate pas sur les attributs d'un objet qui changent de valeurs comme c'est le cas d'un événement interne, mais il se constate sur le message qui accompagne l'événement. Ce prédicat contient les différentes conditions de validité du message reçu. Par exemple, dans le cas du workflow, le message associé à l'événement 'Démarrer le processus' est composé d'un ensemble d'attributs précisant quel processus et quel participant sont concernés par cet événement, à savoir 'Id_Participant' et 'WF_Id'. Ce message est présenté sous forme d'un prédicat qui exprime les conditions de son acceptation.

<u>Représentation d'un événement externe:</u>

L'élément **Evénement-Externe** est un type particulier du concept Evénement. Il se différencie par rapport au concept Evénement-Interne par la nature de l'objet constaté qui est dans ce cas un message et non pas un objet. Ainsi, la structure de l'événement externe est composée par la structure de base de l'événement à laquelle on ajoute une référence vers le message constaté (voir Figure 59).

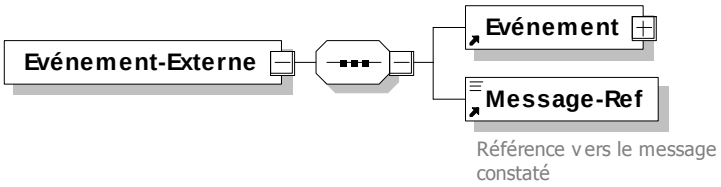


Figure 59. La structure d'un événement externe

L'élément **Message-Ref** représente une référence sur l'élément **Message** dont la structure est présentée à la Figure 60. Comme le montre cette figure, un message est défini par un identifiant, un nom et une structure de données formée par un ensemble d'attributs. Cette structure de données doit contenir les informations nécessaires aux opérations à déclencher dans la partie trigger de l'événement. Les informations du message sont utilisées pour déterminer l'objet sur lequel l'opération s'applique ou/et les valeurs des paramètres d'entrée de l'opération.

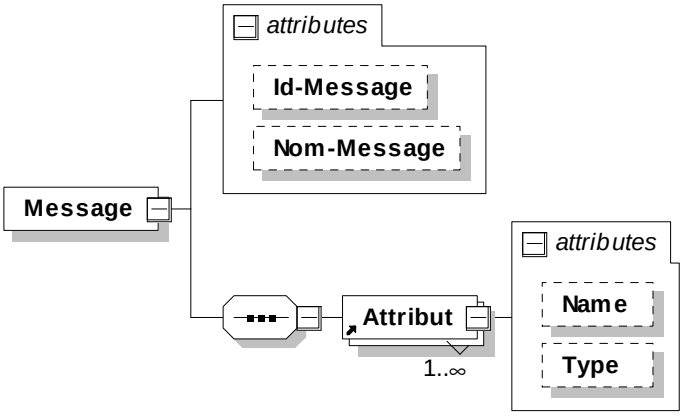


Figure 60. La structure du concept Message

Exemple d'événement externe :

L'événement *EV1* de la Figure 61 illustre le cas d'un événement externe.

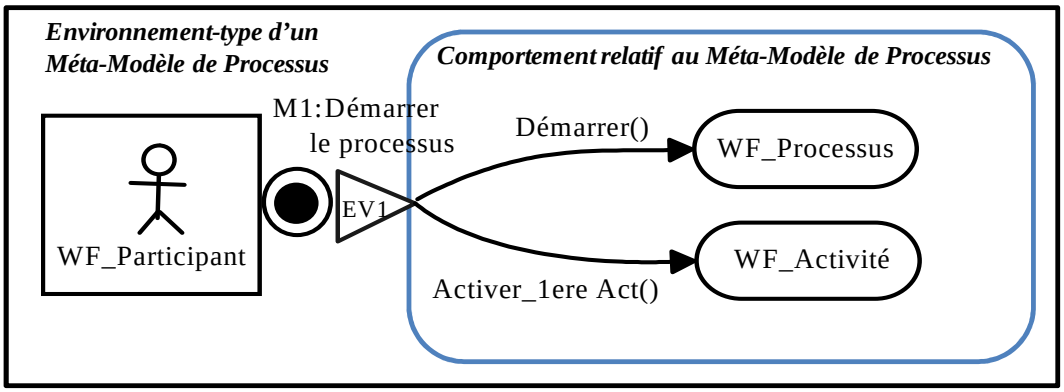


Figure 61. Un exemple d'événement externe

L'acteur *WF_Participant* est le responsable du démarrage du processus d'exécution du workflow et le comportement du méta-modèle de processus doit réagir en conséquence en enregistrant le démarrage du processus choisi et en initialisant la première activité du processus et mettre son état à « en_cours » (*WF_Activité)*. La représentation textuelle de cet exemple illustrant la spécification d'un événement externe est présentée à Figure 62.

```
<Evénement-Externe>
        <Evénement Id-Event="EV1" Nom-Event="Démarrage d'un processus"
                    Prédicat- Event =" Existe (WF_Participant,WF_Processus)">
                <Trigger >
                    <!--Déclenchement de l'opération Démarrer () sur l'objet WF_Processus
                     et de l'opération Activer_1ere_Act () sur l'objet WF_Activité-->
                </Trigger>
        </Evénement>
        <Message-Ref>M1</Message-Ref>
</Evénement-Externe>
```

Figure 62. La représentation textuelle de l'exemple d'un événement externe

Dans cet exemple, M1 représente la référence du message dont la structure est la suivante :

```
<Message Id-Message="M1" Nom-Message="Démarrer le processus ">
        <Attribut Name="Id_Participant" Type="Integer"/>
        <Attribut Name="WF_Id" Type="Integer"/>
</Message>
```

Ce message détermine quel processus va être exécuté à travers l'information (WF_Id qui est l'identifiant du processus WF_Process_Class) et qui est l'acteur qui demande cette exécution (Id_Participant).

4.5.2.4. Le concept Acteur

Un acteur représente un ensemble de rôles joués par les entités externes qui interagissent avec le modèle de processus. Ces entités peuvent être des utilisateurs humains ou des applications. Un acteur produit des événements externes qui en réaction, déclenchent des traitements affectant les objets constituant le niveau « instance ». Le concept Acteur est intéressant à représenter au niveau du méta-modèle parce qu'il permet de modéliser le comportement interactif entre des éléments de l'environnement du processus modélisé.

Représentation d'un acteur :

Un Acteur est décrit par son nom, son type (humain ou système), un ensemble de d'opérations externes et un ensemble d'événements externes.

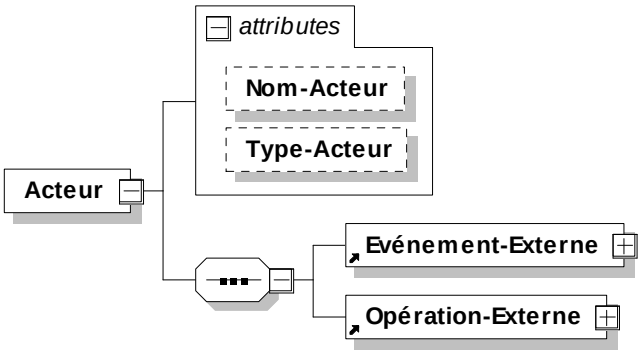


Figure 63. La structure du concept Acteur

La forme graphique de la structure du concept Acteur est présentée comme le montre la Figure 63.

La forme textuelle associée à la structure du concept Acteur est présentée à la Figure 64. L'élément **Evénement-Externe** a déjà été introduit à la section 4.5.2.3.

```
<xs:element name="Acteur">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="Evénement-Externe"/>
                                <xs:element ref="Opération-Externe"/>
                        </xs:sequence>
                        <xs:attribute name="Nom-Acteur" type="xs:string"/>
                        <xs:attribute name="Type-Acteur" type="xs:string"/>
        </xs:complexType>
</xs:element>
```

Figure 64. La forme textuelle de la structure du concept Acteur

L'élément **Opération-Externe** est un concept qui, comme l'élément Opération du modèle de classes orienté objet (présenté dans la section 4.5.2), a un nom (Nom-Op) et des paramètres d'entrées, par contre elle n'a pas de paramètres de sortie et elle fait référence à un des messages reçus par l'acteur externe. La structure du concept Opération-Externe est présentée à la Figure 65 :

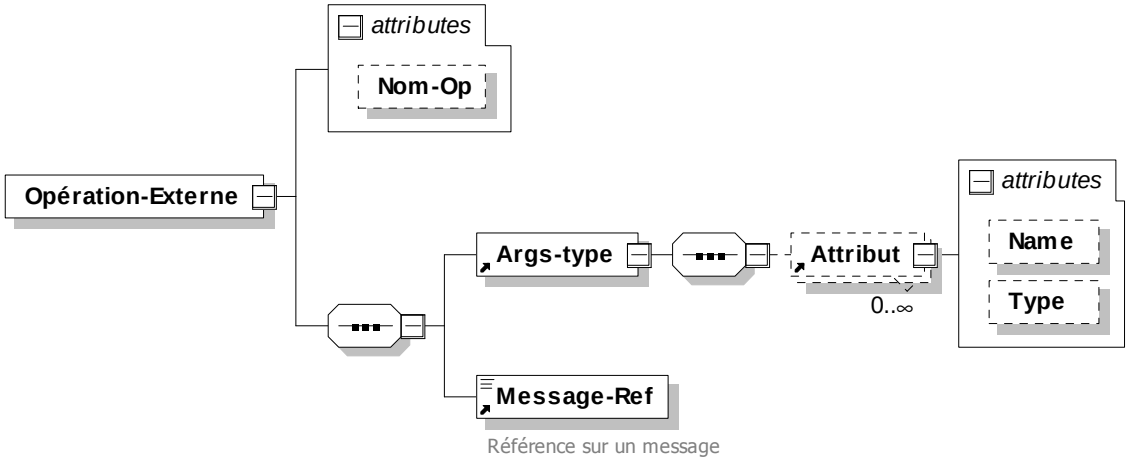


Figure 65. La structure du concept Opération-Externe

Les paramètres d'entrée **Args-type** sont représentés par une structure de données formée par un ensemble d'attributs définis chacun par un nom et un type. La forme textuelle associée à la structure du concept Opération-Externe est présentée à la Figure 66.

```
<xs:element name="Opération-Externe">

        <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="Args-type"/>
                                <xs:element ref="Message-Ref"/>
                        </xs:sequence>
                        <xs:attribute name="Nom-Op" type="xs:string"/>
                </xs:complexType>


</xs:element>
```

Figure 66. La forme textuelle de la structure du concept Opération-Externe

<u>Exemple illustrant le concept Acteur :</u>

Le méta-modèle de workflow tel qu'il est présenté à la Figure 48 ne décrit pas l'interaction du méta-modèle de processus avec les acteurs externes. Ce méta-modèle ne montre pas le coté systémique de l'exécution du processus, c'est pour cette raison que sa spécification dynamique est effectuée à l'aide des concepts du modèle Remora étendu et leur représentation graphique associée. La Figure 67 présente un extrait de cette représentation illustrant une partie de la spécification dynamique du méta-modèle de workflow avec la présence des deux acteurs *WF_Participant* et *WF_Application*.

D'après cette représentation, on peut comprendre que l'acteur *WF_Participant* est à l'origine de deux événements externes (*EV1* et *EV3*) qui vont agir sur des objets de la spécification dynamique du méta-modèle de workflow (tels que *WF_Processus* et *WF_Activité*). L'événement *EV1* déclenche le processus de démarrage de l'exécution du workflow, il est constaté par l'arrivée du message *M1 : Démarrer le processus* de la part de l'acteur *WF_Participant*. Le prédicat de cet événement doit vérifier la validité des données fournies par le message M1, à savoir l'existence du processus-type et du participant (*WF_Participant*). L'événement EV3 est produit par le même acteur *WF_Participant* et déclenche l'opération de sélection d'une activité une fois que celle ci constate l'arrivée du

message *M3 : sélectionner une activité*. Le prédicat de cet événement doit vérifier l'existence de l'activité sélectionnée ainsi que le fait que le participant est bien celui qui est affecté à cette activité. De même, l'acteur *WF_Application* permet de supporter l'exécution d'une activité sélectionnée par l'acteur de type WF_Participant affecté à cette activité.

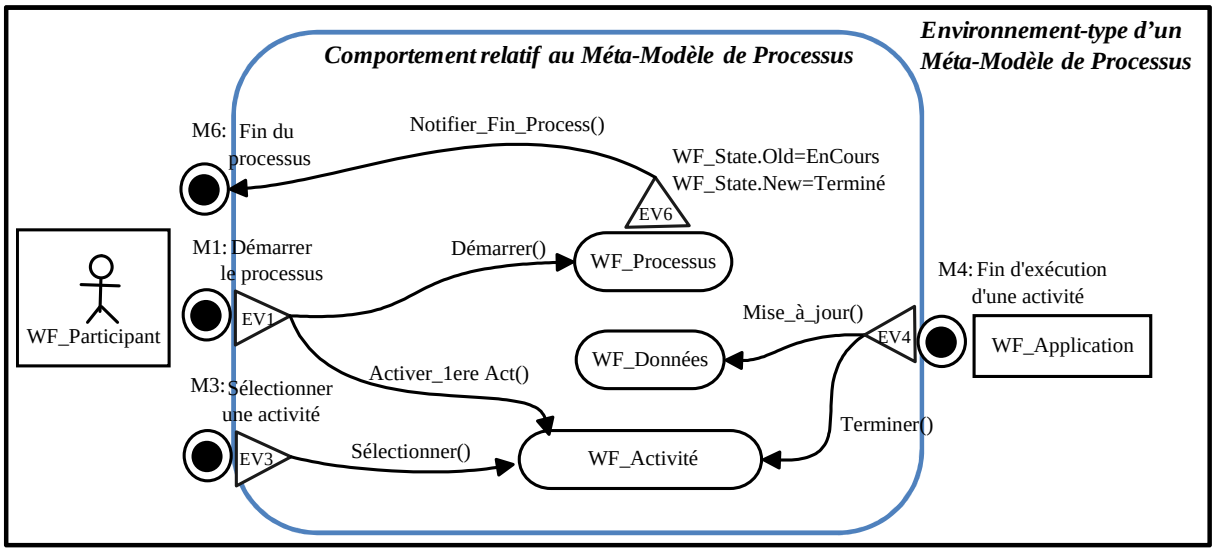


Figure 67. Un exemple illustrant les acteurs du workflow et leurs interactions avec l'outil

L'événement EV4 permet de recevoir le message de fin d'exécution d'activité de l'application. Cet événement est important puisqu'il faut calculer les activités suivantes et les affecter aux acteurs de type WF_Participant. Le message M4 associé à l'événement EV4 contient l'identifiant de l'activité terminée, l'identifiant des données produites par l'exécution de cette activité.

La représentation textuelle relative à l'exemple de la Figure 67 est présentée à la Figure 68.

```
<!--Sample XML file generated by XMLSpy v2014 sp1 (x64) (http://www.altova.com)-->
<Acteur  Nom-Acteur="WF_Participant" Type-Acteur="Humain" ">
        <Evénement-Externe>
                <Evénement Id-Event="EV1" Nom-Event="Démarrage d'un processus"
                                Prédicat- Event =" Existe (WF_Participant, WF_Id) ">
                        <Trigger >
                                <!--Déclenchement de l'opération Démarrer () sur l'objet WF_Processus
                                    et de l'opération Activer_1ere_Act () sur l'objet WF_Activité-->
                        </Trigger>
                </Evénement>
                <Message-Ref>M1</Message-Ref>
        </Evénement-Externe>
        <Evénement-Externe>
                <Evénement Id-Event="EV3" Nom-Event="Sélection d'une activité"
                  Prédicat-Event="Activité existe & Participant est celui affecté à l'activité ">
                        <Trigger>
                            <!--Déclenchement de l'opération Sélectionner () sur l'objet WF_Activité-->
                        </Trigger>
                </Evénement>
                <Message-Ref>M3</Message-Ref>
        </Evénement-Externe>
        <Opération-Externe Nom-Op="Notifier_Fin_Process">
                <Args-type Name=" WF_Id" Type="Integer" />
                <Message-Ref>M5</Message-Ref>
        </Opération-Externe>
```

```
</Acteur>
<Acteur  Nom-Acteur="WF_Application" Type-Acteur="Système" ">
        <Evénement-Externe>
                <Evénement Id-Event="EV4" Nom-Event="Fin d'exécution"
                        Prédicat-Event=  "Activité existe & Données existent  ">
                        <Trigger>
                          <!--Déclenchement de l'opération Mise_à_jour () sur l'objet WF_Données
                             et de l'opération Terminer () sur l'objet WF_Activité -->
                        </Trigger>
                </Evénement>
                <Message-Ref>M4</Message-Ref>
        </Evénement-Externe>
</Acteur>
  <!—La liste des messages référencés -->
<Message Id-Message="M1" Nom-Message="Démarrer le processus ">
        <Attribut Name="Id_Participant" Type="Integer"/>
        <Attribut Name="WF_Id" Type="Integer"/>
</Message>
<Message Id-Message="M3" Nom-Message="Sélectionner une activité ">
        <Attribut Name="Id_Participant" Type="Integer"/>
        <Attribut Name="Act_Id" Type="Integer"/>
</Message>
<Message Id-Message="M4" Nom-Message="Fin d'exécution d'une activité ">
        <Attribut Name="Act_Id" Type="Integer"/>
        <Attribut Name="Id_Data" Type="Integer"/>
</Message>
<Message Id-Message="M5" Nom-Message="Fin du processus ">
        <Attribut Name="WF_Id" Type="Integer"/>
        <Attribut Name="Id_Participant" Type="Integer"/>
</Message>
```

Figure 68. Un exemple illustratif du concept Acteur

Cet extrait de la spécification dynamique du modèle de workflow illustre l'interaction de l'exécution du processus de workflow avec l'acteur WF_Participant à travers les deux événements externes : « Démarrage d'un processus » et « Sélection d'une activité » ainsi qu'à l'opération externe de notification de la fin du processus, d'une part et d'autre part, avec l'acteur WF_Application à travers l'événement externe « Fin d'exécution ».

4.5.2.5. Le concept Trigger

Le trigger représente l'impact d'une occurrence d'événement sur les objets du niveau « instance ». Il correspond à l'ensemble des opérations à déclencher lors d'une occurrence d'un événement fixé. Les opérations n'ont pas forcément d'ordre d'exécution: elles s'exécutent en parallèle. Le déclenchement de chacune de ces opérations peut être :

- conditionnel : il se produit si une *condition* est vraie, et - ou
- itératif: il se produit plusieurs fois selon un *facteur* d'itération

L'exécution des opérations se fait selon l'algorithme suivant :

```
Si (Condition et Facteur)
Alors si (Condition est vraie)
        {Pour chaque Facteur {Exécuter Opération}}
Sinon
    Si (Condition et pas de Facteur)
    Alors {si (Condition est vraie)
```

```
        {Exécuter Opération}}
    Si (pas de Condition et Facteur)
    Alors {Pour chaque facteur
        {Exécuter Opération}}
    Sinon // (pas de Condition et pas de Facteur)
        {Exécuter Opération}
```

Représentation d'un Trigger :

Comme le montre la structure graphique de la Figure 69, un Trigger est défini par un identifiant (Id-Trigger) et est composé par un ou plusieurs éléments de type 'Trigger-Elément'.

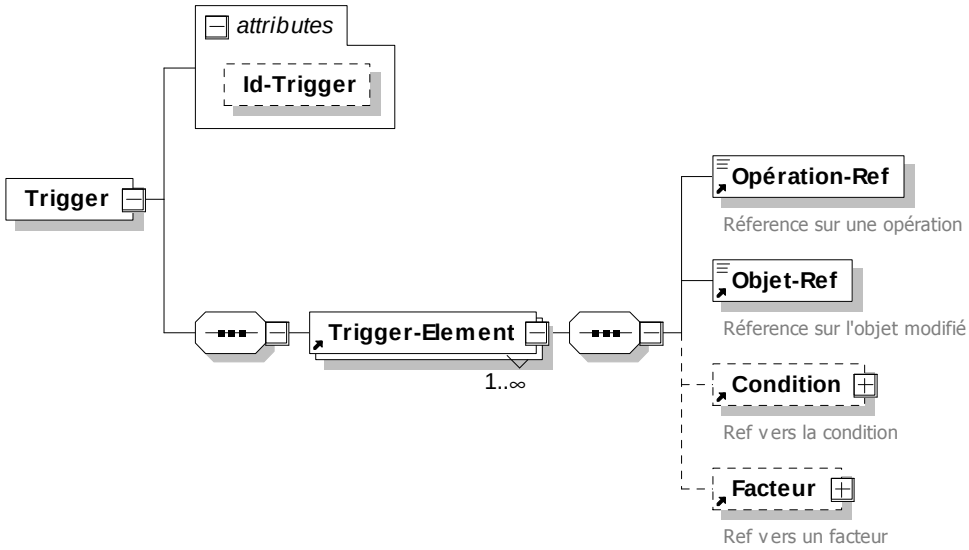


Figure 69. La structure d'un Trigger

Etant donné qu'un Trigger peut contenir plusieurs opérations et peut modifier différents objets à la fois, nous avons ajouté l'élément '**Trigger-Elément'** qui permet d'associer la définition d'une opération avec l'objet qu'elle modifie et les conditions et les facteurs qui accompagnent son déclenchement. Pour cela, nous avons défini pour chaque Trigger-Element une seule opération (Opération-Ref), un seul objet modifié (Objet-Ref), zéro ou une condition (Condition) et zéro ou un facteur (Facteur).

La forme textuelle de la structure d'un trigger en XML est présentée à la Figure 70.

```
<xs:element name="Trigger">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="Trigger-Element" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="Id-Trigger"/>
        </xs:complexType>
</xs:element>

<xs:element name="Trigger-Element">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref=" Opération-Ref "/>
                        <xs:element ref="Objet-Ref"/>
```

```
                    <xs:element ref="Condition" minOccurs="0" maxOccurs="1"/>
                    <xs:element ref="Facteur" minOccurs="0" maxOccurs="1"/>
                </xs:sequence>
            </xs:complexType>

        </xs:element>
```

Figure 70. Forme textuelle de la structure d'un Trigger

Les éléments Condition et Facteur sont en fait deux références vers les concepts Condition et Facteur qui sont représentés à la Figure 71.

L'élément **Condition** est caractérisé par un identifiant (Id-Condition), un nom (Nom-Condition) et un texte (Texte-Condition) qui comprend une structure de données qu'il faut vérifier avant le déclenchement d'une opération. La condition d'occurrence d'un événement interne décrit le changement remarquable d'état d'un objet tandis que la condition d'occurrence d'un événement externe exprime l'arrivée d'un message valide définissant les paramètres transmis par l'environnement ainsi que leur type. Graphiquement, une condition est représentée par son nom (Nom-Condition) qui accompagne la flèche de l'opération.

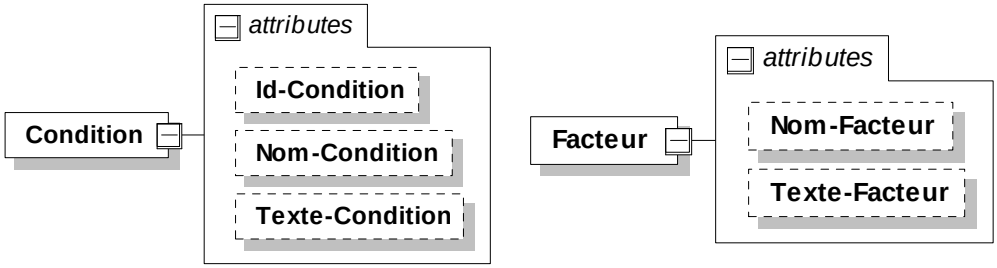


Figure 71. Les deux concepts Condition et Facteur

L'illustration d'une opération soumise à une condition est présentée avec l'exemple du Trigger à la Figure 74.

L'élément **Facteur** est défini par un nom (exemple : F1) et un texte qui indique les objets sur lesquels l'itération se produit. Il est visualisé par une double flèche à laquelle le nom du facteur est associé, comme le montre l'exemple de la Figure 72.

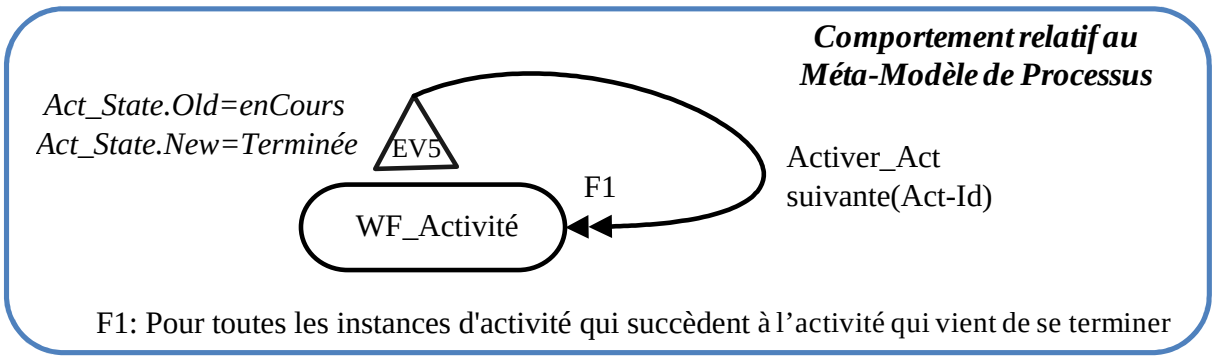


Figure 72. Un exemple d'un Facteur

Dans cet exemple, le facteur F1 inclut les objets de la classe WF_Activité_Class qui succèdent à l'activité qui vient de se terminer. L'opération *Activer_Act_suivante()* doit créer un objet relié à l'objet « WF_Activité_Class » qui lui correspond et avec un état initial « Activée» .

La spécification XML correspondante à cet exemple de facteur est présentée comme suit :

```
<Facteur Nom-Facteur="F1" Texte-Facteur="Pour toutes les objets de WF_ActivitéClass qui
succèdent à l'activité courante"/>
```

Exemple de trigger:

La Figure 73 présente un exemple de trigger (la partie encerclé en trait pointillé) déclenché par l'événement interne EV5.

Cet événement, qui signale la fin d'exécution d'une activité, déclenche le trigger composé par les opérations *Activer_Act_Suivante(), mise_à_jour_D()* et *Terminer_WF()*. Ce déclenchement des opérations est conditionné. Si la condition C1 est vérifiée alors il s'agit d'une activité finale et dans ce cas l'opération *Terminer_WF()* est déclenchée. Dans le cas contraire, c'est l'opération Activer_Act_suivante() qui est déclenchée pour chaque activité qui suit celle qui vient de se terminer.

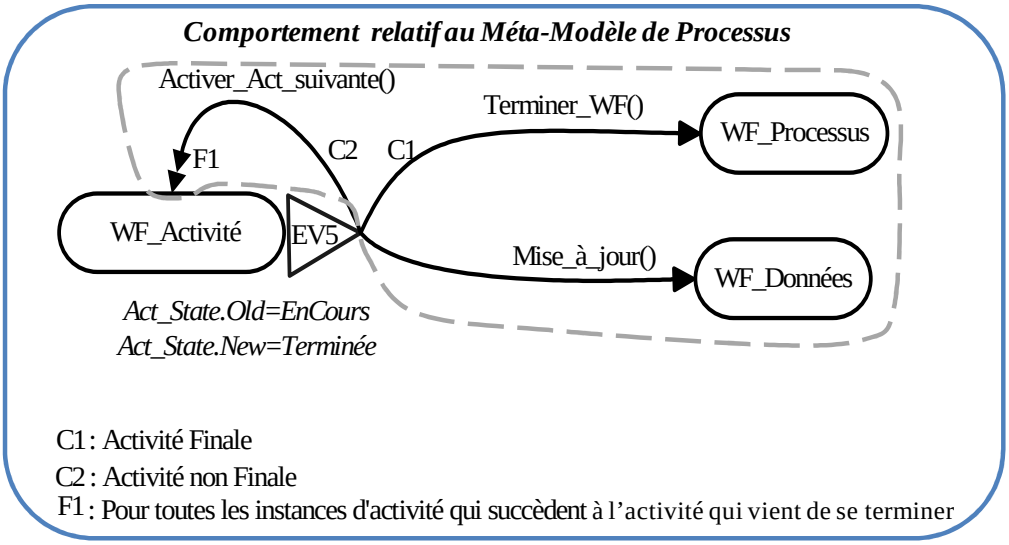


Figure 73. Un exemple d'un trigger extrait du modèle dynamique de workflow

La représentation du trigger relatif à l'exemple précedent est présentée à la Figure 74 comme suit :

```
<Trigger  Id-Trigger="TR5""">
        <Trigger-Element>
                <Opération-Ref> Terminer_WF </Opération-Ref>
                <Objet-Ref>WF_Processus</Objet-Ref>
                <Condition> C1 <Condition/>
        </Trigger-Element>
        <Trigger-Element>
                < Opération-Ref Nom-Op=" Mise_à_jour ">
                        <Args-type >
                                <Attribut Name="Id_Data_Instance" Type="Integer" />
                        <Args-type/>
                <// Opération-Ref >
                <Objet-Ref>WF_Données</Objet-Ref>
        </Trigger-Element>
        <Trigger-Element>
                < Opération-Ref Nom-Op="Activer_Act_Suivante ">
                        <Args-type >
                                <Attribut Name="Id_Data_Instance" Type="Integer" />
                        <Args-type/>
```

```
                    </Opération-Ref >
                     <Objet-Ref>WF_Activité</Objet-Ref>
                     <Condition> C2 <Condition/>
                     <Facteur> F1 <Facteur/>
          </Trigger-Element>
</Trigger>
<!—La liste des Conditions -->
<Condition Id-Condition="C1" Nom-Condition="Activité finale" Texte-Condition=" Vérifier que l'activité
courante est l'activité finale du processus"/>
<Condition Id-Condition="C2" Nom-Condition="Activité non finale" Texte-Condition= " Vérifier que
l'activité courante n'est pas l'activité finale du processus "/>
<!—La liste des Facteurs -->
<Facteur Nom-Facteur="F1" Texte-Facteur="Pour toutes les objets d'activité_Class qui succèdent
l'activité courante"/>
```

Figure 74. Un exemple illustratif d'un Trigger

A titre d'illustration, l'implémentation de la condition C1 va se faire avec la méthode *Est_Finale (IdActivité)* et elle devra faire les actions suivantes :

- Récupérer l'identifiant du processus instance (WF_Id_Instance) à laquelle appartient l'activité courante (à travers le lien liée à)
- Récupérer l'identifiant du processus-type (WF_Id) à travers le lien « Ref »
- Chercher la classe X qui est rattachée au processus WF_Id par l'association Activité_Finale.
- Vérifier que l'attribut Act_Id_Instance appartient à cette classe X.

La structure complète du modèle événementiel est présentée comme suit à la Figure 75.

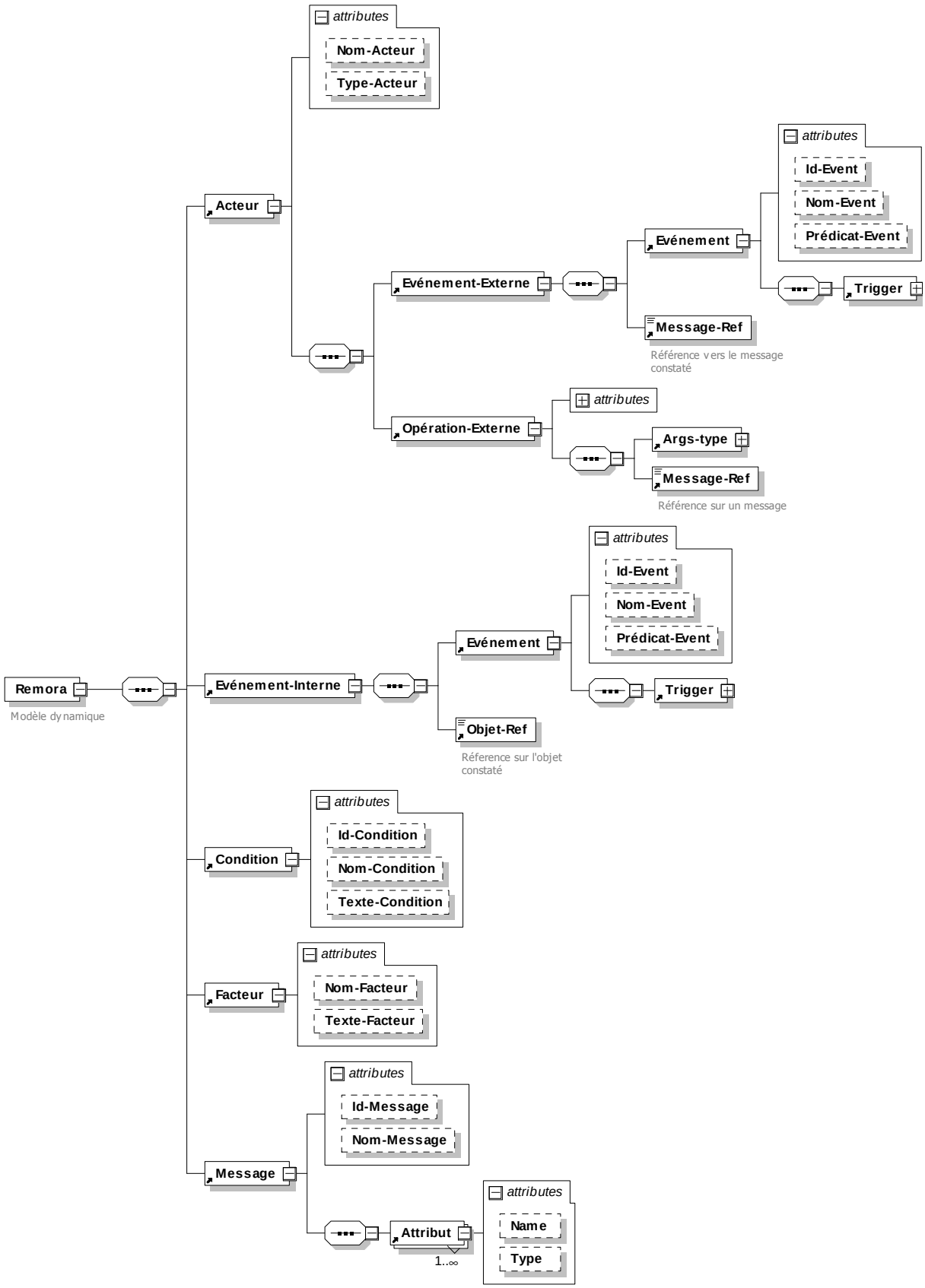


Figure 75. La structure du modèle Remora étendu en format XML graphique

4.5.3. Le schéma dynamique de la sémantique opérationnelle du Méta-Modèle de Processus

Le rôle du schéma dynamique est de présenter de manière globale et synthétique le comportement attendu d'un modèle de processus.

Ce schéma est présenté sous forme graphique et sera accompagné par un document explicatif qui décrit pour chaque transition dynamique :

- Nom de l'évènement et son type
- Description du message
- Nom de l'acteur ou l'objet émetteur
- Enumération des opérations du trigger
- Description des facteurs et des conditions de déclenchement
- Description de chacune des opérations

Une fois le schéma dynamique élaboré, un ensemble de règles de conformité, de cohérence et de complétude doit être appliqué. Ci-après, nous présentons des exemples de ces règles :

Règles de conformité du modèle événementiel :

- Une opération agit sur un seul objet
- Un évènement constate le changement d'état d'un seul objet
- Une opération est déclenchée par un seul évènement

Règles de Cohérence

- Deux opérations agissant sur le même objet et déclenchées par le même évènement doivent s'exclure mutuellement.
- Deux chemins partant du même évènement et aboutissant au même objet doivent s'exclure
- Une transition dynamique doit laisser le système d'objets dans un état cohérent

Règles de Complétude

- Toute opération est déclenchée par un évènement
- Tout évènement déclenche au moins une opération
- Tout objet est modifié par au moins une opération

<u>Exemple de schéma dynamique :</u>

Afin d'avoir une idée plus claire sur la spécification de l'aspect dynamique avec le modèle événementiel, nous présentons à la Figure 76 un schéma dynamique illustrant cette représentation dans le cas du workflow.

Il s'agit d'une représentation simple qui met en évidence l'ensemble des événements externes et internes possibles lors de l'exécution d'un processus de Workflow ainsi que l'ensemble des opérations qu'ils déclenchent. Il existe deux types d'acteur qui interagissent avec le modèle de Workflow: *WF_Participant* et *WF_Application*. Il est nécessaire de concevoir les interactions entre ce modèle du workflow et ces acteurs.

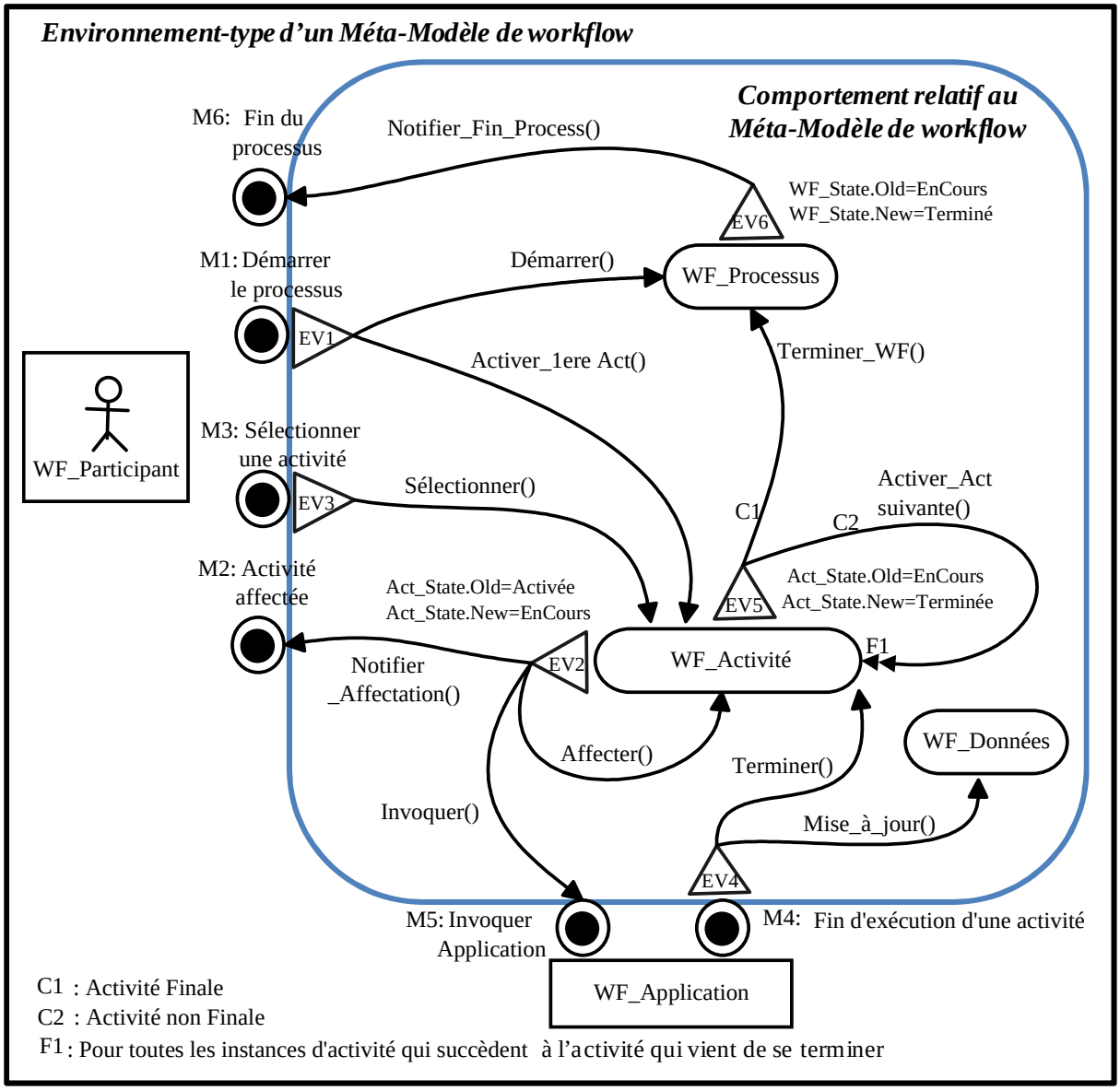


Figure 76. Le schéma dynamique du méta-modèle de Workflow (exemple simplifié)

Cette représentation montre que WF_participant est vu comme une entité externe au processus et non pas juste comme une classe d'objet. Les événements présentés dans l'exemple ci-dessus sont :

Les événements internes :

- *EV2: Affectation d'une activité* : constate le changement de l'état d'une activité (this.Act_State. old= « Activée » et this.Act_State. new= « EnCours »)
- *EV5: Fin d'exécution d'une activité* : constate le changement de l'état d'une activité. Son prédicat est : (this.Act_State.old= « EnCours » et this.Act_State. new= « Terminée»)
- *EV6: Fin d'exécution d'un processus* : constate le changement de l'état d'un processus workflow. Son prédicat est : (this.WF_State.old= « EnCours » et this.WF_State.new= « Terminé »)

Les événements externes :

- *EV1: Démarrer Processus*: se déclenche si le message M1 reçu vérifie le prédicat suivant : (M1.WF_Id).WF_State = « créé », c.à.d. si le processus (dont l'identifiant figure dans le message) existe.
- *EV3: Sélectionner activité*: se déclenche si le message M3 reçu vérifie le prédicat suivant : (M3.Act_Id).Act_State = « Activée » and M3.Id_Participant existe.
- *EV4: Fin_activité:* se déclenche lorsque le message M4 reçu vérifie le prédicat suivant : (M4.Act_Id.Act_State= « EnCours »).

L'avantage de la spécification présentée à la Figure 76 est qu'elle regroupe toutes les transitions dynamiques dans un même graphe. Elle permet de visualiser les événements externes, leurs impacts sur les objets ainsi que les événements internes résultants. Ce graphe est un instrument puissant de représentation synthétique du comportement attendu associé aux concepts du méta-modèle de processus.

La totalité de la spécification en XML qui correspond au schéma dynamique du workflow est présentée à l'**Annexe 10**. A la Figure 77, nous présentons un extrait qui illustre les principaux éléments de la structure de cette spécification.

```
<Acteur  Nom-Acteur="WF_Participant" Type-Acteur="Humain" ">
        <Evénement-Externe>
                <Evénement Id-Event="EV1" Nom-Event="Démarrage d'un processus"
                  Prédicat- Event =" Existe (WF_Participant,WF_Processus) ">
                        <Trigger Id-Trigger="TR1"> …   </Trigger>
                </Evénement>
                <Message-Ref>M1</Message-Ref>
        </Evénement-Externe>
        <Opération-Externe Nom-Op="Notifier_Fin_Process">
                <Args-type Name=" WF_Id" Type="Integer" />
                <Message-Ref>M6</Message-Ref>
        </Opération-Externe>
</Acteur>
<Evénement-Interne>
        <Objet-Ref> WF_Activité</Objet-Ref>
               <Evénement … ">
                <Trigger  Id-Trigger="TR5">…   </Trigger>
        </Evénement>
</Evénement-Interne>
<!—La liste des facteurs -->
<!—La liste des messages référencés -->
<!—La liste des opérations référencés -->
```

Figure 77. Un extrait de la spécification en XML du schéma dynamique de workflow

La spécification dynamique présentée à l'**Annexe 10** doit être ajoutée à la spécification statique du méta-modèle de workflow tout en respectant la cohérence des objets et des opérations afin d'obtenir une spécification complète relative au méta-modèle de workflow. En revanche, cette spécification n'est pas exécutable puisqu'elle est formée de deux formalismes de nature différente.

Dans l'objectif de définir une spécification homogène sous une forme orientée objet, nous proposons, dans la suite de ce chapitre, des règles de transformation permettant d'obtenir une spécification standard à partir de la description dynamique du modèle de processus.

Après avoir défini les concepts particuliers qu'on aura besoin pour construire une spécification dynamique du méta-modèle de processus, nous allons présenter, à la section suivante, les règles de transformation pour dériver l'arichtecture de l'outil associé à ce méta-modèle de processus.

4.6. Etape 3: Transformation du modèle comportemental vers un modèle objet technique standard

La spécification de la sémantique opérationnelle obtenue à l'issue de l'étape 2 permet d'exprimer le contrôle de l'exécution du processus conforme au méta-modèle de processus sous forme de schéma de comportement exprimant quand et pourquoi les traitements à exécuter dans la partie trigger des événements (internes ou externes) sont déclenchés. Ces traitements sont représentés par un ensemble d'opérations permettant de mettre à jour l'état des objets du processus et envoyer des messages aux éléments de l'environnement du processus si nécessaire.

L'objectif de la troisième étape est double :

- Dériver l'architecture de l'outil d'exécution à partir des modèles de processus conformes au méta-modèle de processus en exploitant la spécification de la sémantique opérationnelle,
- Transformer la sémantique opérationnelle des concepts de Remora étendu en UML standard (diagramme de classes et diagramme de séquences) pour que l'architecture de l'outil d'exécution prenne une forme orientée objet standard (UML) et peut ainsi être interprétée par n'importe quel outil permettant de générer le code en fonction de la plateforme technique cible.

Ainsi, notre solution au problème de construction d'outil d'exécution ne s'arrête pas à proposer une spécification conceptuelle spécifique, mais elle vise aussi à aboutir à une opérationnalisation cette spécification. Pour que la spécification de la sémantique opérationnelle puisse constituer une architecture d'outil d'exécution de processus, il faut lui adjoindre une classe de haut niveau permettant de charger le modèle de processus à exécuter et configurer l'environnement de celui-ci au niveau technique et notamment les différents acteurs et leurs configurations techniques afin de pouvoir interagir avec eux à partir de l'outil.

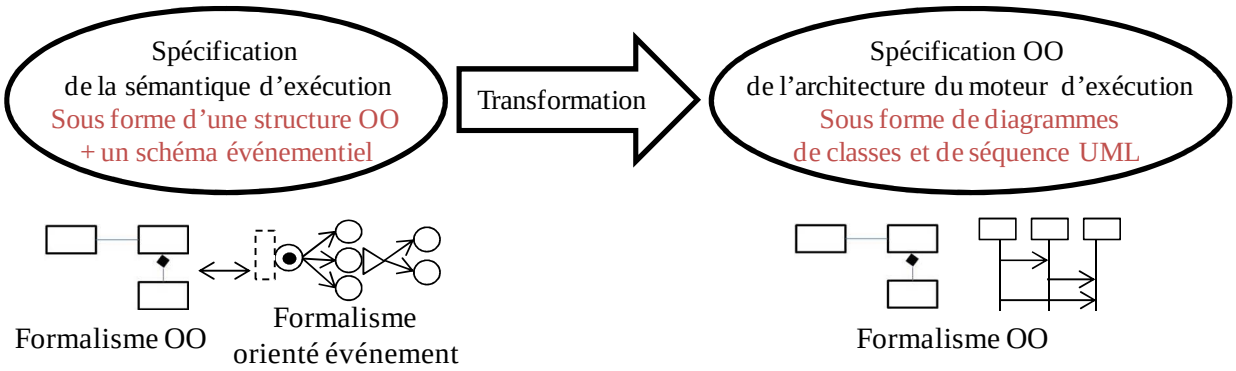


Figure 78. La 3ème étape de la proposition: La transformation

Comme le montre la Figure 78, la transformation, dans notre cas, a pour entrée une spécification de la sémantique d'exécution sous une forme orientée objet complétée par un schéma événementiel et a comme sortie une spécification orientée objet (diagrammes UML) qui correspond à l'architecture du moteur d'exécution. Il est à préciser que la transformation s'effectue sur des modèles instances qui sont conformes à des méta-modèles. Pour la spécification d'entrée, la relation entre les le méta-modèle OO et le méta-modèle événementiel est schématisée à la Figure 79.

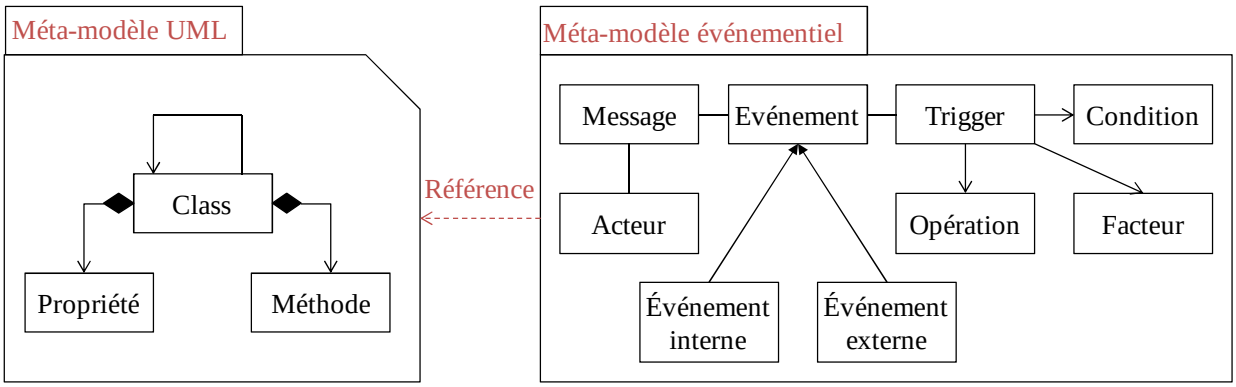


Figure 79. La relation entre le méta-modèle UML et le méta-modèle événementiel

Cette relation montre que les objets du modèle événementiel pointent sur des éléments du méta-modèle statique afin d'assurer la cohérence de la spécification. Ceci exige la vérification du fait que tous les élements du modèle dynamique correspondent à des éléments dans le méta-modèle en UML. Cette relation montre aussi que le modèle source n'est pas uniquement le modèle événementiel, mais plutôt c'est un modèle qui combine des concepts événementiels et des concepts orientés objet.

4.6.1. Formalisation des schémas d'entrée et de sortie pour la transformation

La réalisation de l'étape de transformation dans un environnement de développement exige une formalisation des schémas d'entrée et de sortie de cette transformation. Ainsi, pour préparer le terrain à une future implémentation, nous proposons une représentation en XML des méta-modèles source et cible.

4.6.1.1. Le méta-modèle source

Le document en entrée de la transformation correspond au modèle événementiel. Le fichier XML correspondant à ce modèle est nommé « RemoraMM.xsd » et présenté à la Figure 80. D'après ce modèle, les différents concepts du modèle événementiel sont présentés sous une forme graphique réalisée avec le logiciel XMLSPY dans sa dernière version (2014).

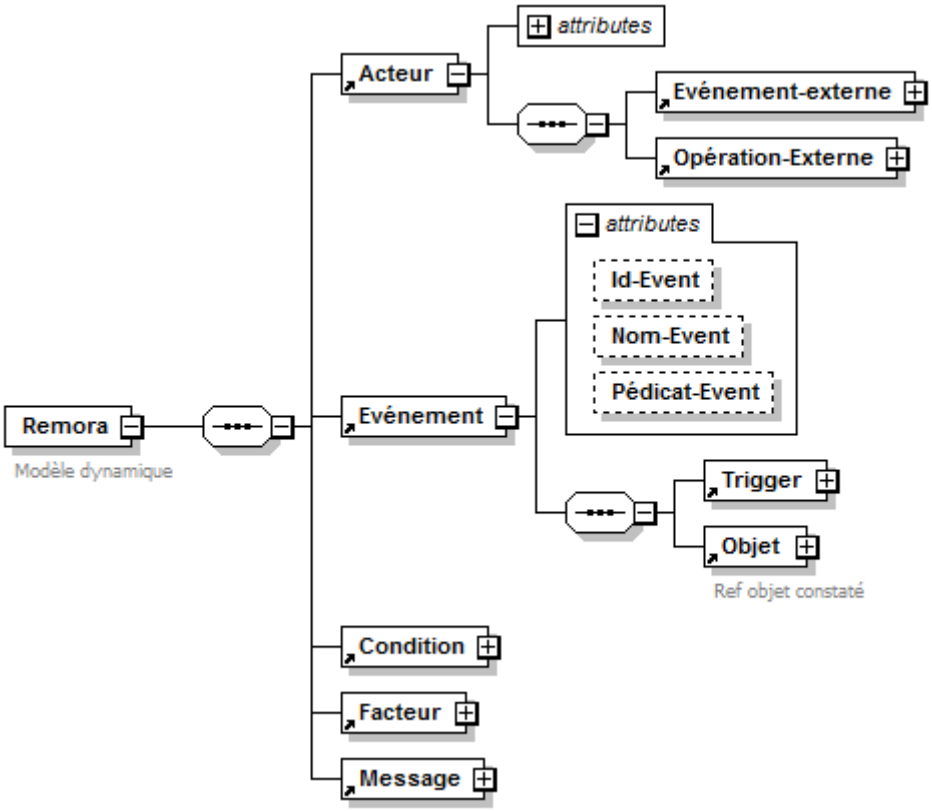


Figure 80. Le méta-modèle source de la transformation

La spécification de l'aspect dynamique du méta-modèle de processus est décrite dans le fichier XML nommé « Remora.xml » qui est conforme au document « RemoraMM.xsd ». Ce fichier est en liaison avec le modèle statique (le diagramme de classes). La liaison entre ces deux documents doit se faire au niveau des DTD comme le montre la Figure 81. Pour cela, les objets du modèle événementiel pointent sur des éléments du méta-modèle statique afin d'assurer la cohérence de la spécification. D'ailleurs, il est nécessaire de vérifier que tous les objets du modèle dynamique correspondent à des classes dans le méta-modèle en UML.

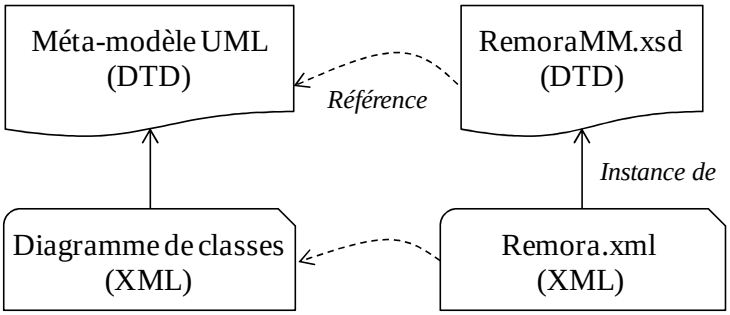


Figure 81. Les documents XML qui représentent les entrées de l'étape de transformation

Un exemple de cette liaison est présenté dans l'extrait suivant de « RemoraMM.xsd » :

```
<xs:element name="Paramètre">
        <xs:complexType>
                <xs:attribute name="Ref-Objet" type="Uml.class"/>
        </xs:complexType>
</xs:element>
```

En effet, le paramètre d'une opération peut être de type objet. Si c'est le cas, il faut s'assurer que cet objet existe dans le modèle statique. Pour cela, on définit l'attribut du paramètre par le nom *Ref-Objet* et on précise que son type est bien *Uml.class*.

La liaison entre des documents XML est assurée grâce au mécanisme d'importation permettant aux éléments des schémas issus de différents 'espaces de noms' d'être utilisés ensemble. Dans ce cas, l''élément <import> apparaît en tant qu'enfant de l'élément <schema> racine. Il permet d'importer des déclarations et des définitions de schéma dans un espace de noms séparé spécifié via l'attribut 'namespace' et l'attribut 'schemaLocation'.

4.6.1.2. Le méta-modèle cible

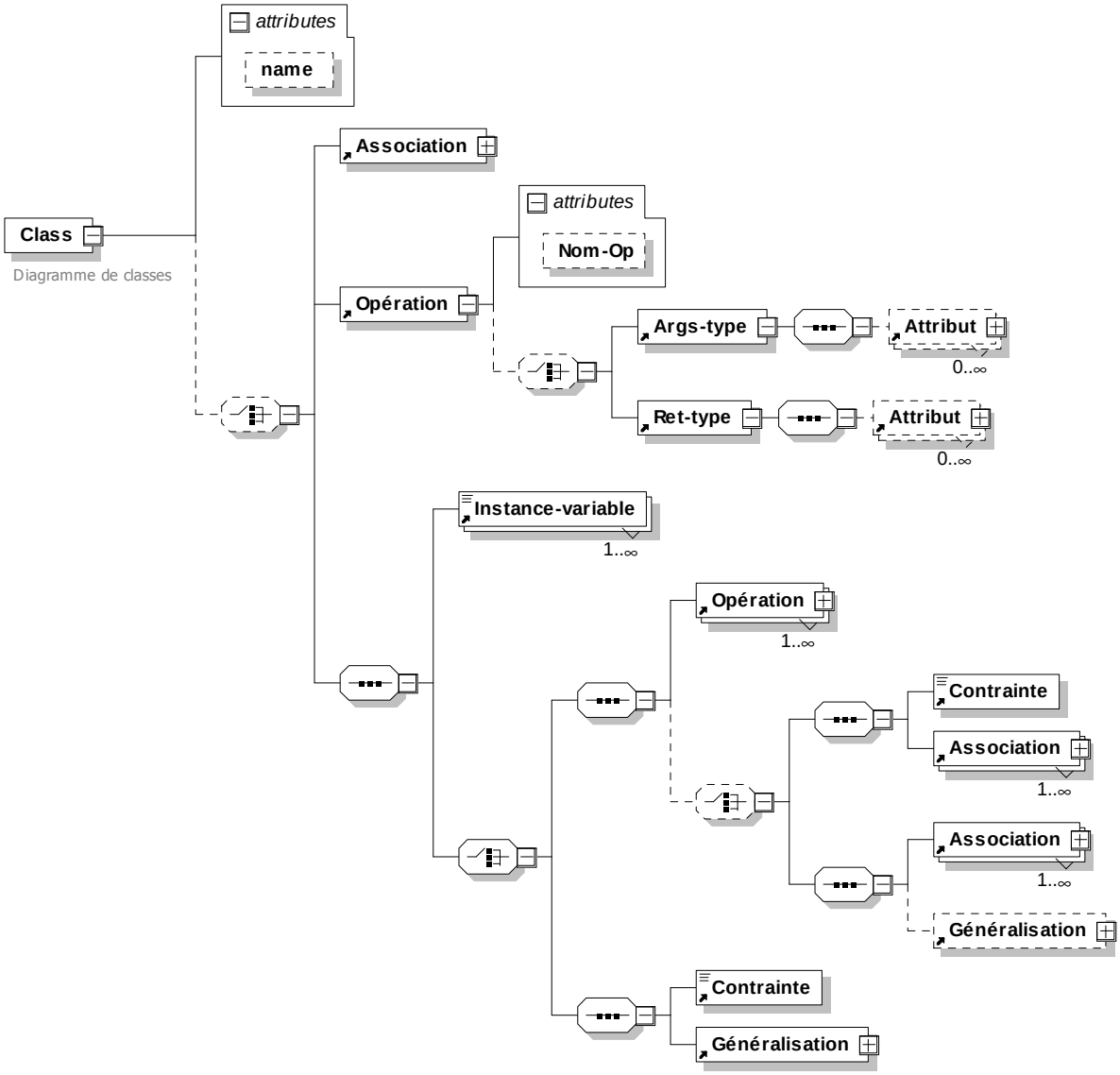


Figure 82. Le méta-modèle cible de la transformation

Le résultat souhaité de la transformation est un modèle orienté objet conforme au méta-modèle de classes UML. La Figure 82 représente la DTD de ce méta-modèle. La forme textuelle de cette DTD est un fichier de type XSD qu'on appelle «*UmlClassMM.xsd* ».

Une fois les méta-modèles source et cible ainsi que le modèle source, obtenue par instanciation du fichier RemoraMM.xsd, ont pris la forme de fichiers XML, on peut passer à l'implémentation des règles.

L'objectif de l'étape de transformation est de représenter la sémantique opérationnelle des concepts d'événement internes, d'événement externe et d'opération externes sous forme d'une spécification orientée objet standard UML. Pour atteindre cet objectif, nous exploitons les principes de publication/souscription et de messagerie point à point. Ces principes vont nous aider à associer à chacun des concepts du modèle événementiel ce qui lui correspond en termes de classes, méthodes, attributs et liens entre classes conformément à un méta-modèle UML. En se basant sur le modèle source, nous pouvant déjà détecter quelques transformations simples et évidentes tels que les règles suivantes :

- L'élément *Object* se transforme en une classe portant son nom,
- L'élément *Operation* se transforme en une méthode,
- Le lien *modifie* entre *Object* et *Operation* se transforme en un lien d'appartenance de la méthode à la classe,
- L'ensemble des liens *déclenche* partant d'un *Evénement* représente l'ensemble des opérations à déclencher lorsque l'événement survient. Ce traitement est matérialisé par une classe particulière appelée classe *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).

D'autres règles sont beaucoup plus compliquées à définir car elles traduisent la sémantique opérationnelle du modèle événementiel. L'application de l'ensemble des règles de transformation au méta-modèle événementiel devra générer des concepts conformes au méta-modèle UML (Figure 47) et définis à l'aide des principes de publication/souscription et de messagerie point à point. Ces concepts permettront d'augmenter le digramme de classes initial qui représente la spécification structurelle de l'outil à construire.

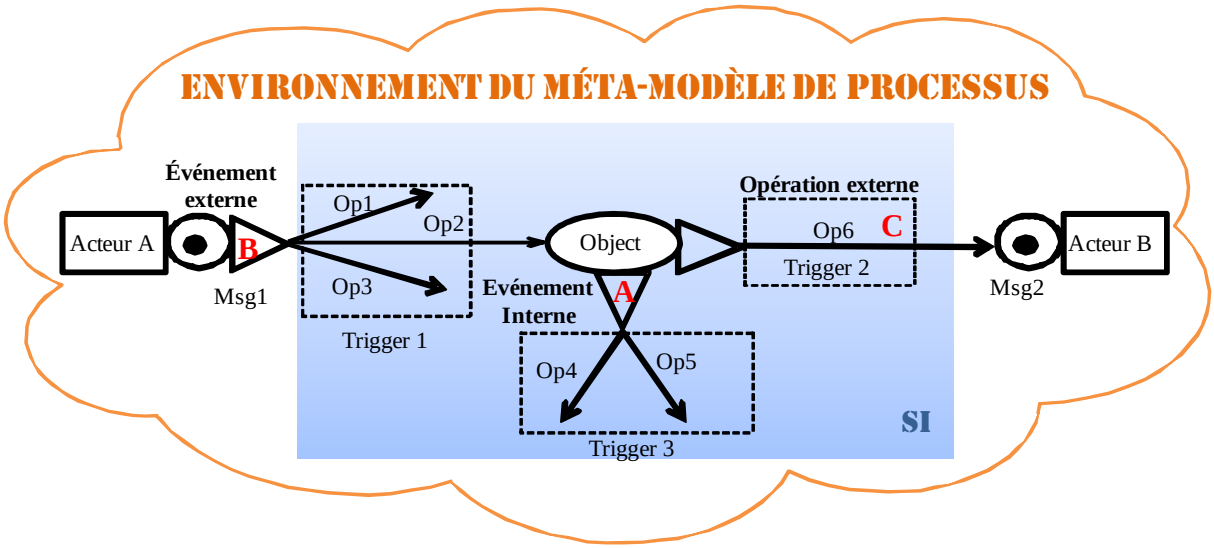


Figure 83. Les trois situations de comportement possibles dans un modèle événementiel

Nous allons détailler dans la suite de cette section les transformations par type de comportement. Nous rappelons que dans un outil interactif, on distingue trois cas différents de comportements comme le montre la Figure 83 :

- Cas d'un événement interne (A), ce comportement correspond à un changement d'état d'un objet du système.
- Cas d'un événement externe qui est provoqué par une action provenant de l'extérieur (B),
- Cas d'une opération vers l'extérieure qui est une invocation d'un acteur externe (C).

Mais avant cela, nous allons expliquer, dans un premier temps, comment dériver l'architecture d'un outil à partir de l'expression de la sémantique d'exécution. Dans un deuxième temps, nous allons présenter les principes de la gestion événementielle sur lesquels nous nous sommes basés pour définir les règles de transformation.

4.6.2. La dérivation de l'architecture objet de l'outil d'exécution à partir de la sémantique opérationnelle

Pour être opérationnel selon les principes du publication/souscription et de messagerie point à point, le schéma de classes à deux niveaux « type » et « instance » doit inclure une classe de plus haut niveau appelée « ***EngineContext*** » qui perme:

- de charger un méta-modèle de processus et instancier les classes du niveau type avant l'exécution d'un processus,
- de configurer techniquement les acteurs constituant l'environnement du processus avant de démarrer son exécution afin de pouvoir réaliser les envois et les réceptions de messages,
- de donner accès aux objets qui constituent le niveau « instance ».

Les règles de transformation proposées à l'étape 3 vont introduire la classe « EngineContext » et aider à transformer la sémantique opérationnelle du modèle Remora étendu en diagrammes de classes et de séquences UML.

4.6.3. Les principes de la gestion événementielle

Les principes de la gestion événementielle sont des modèles de conception qu'on peut utiliser pour implémenter un modèle événementiel. Ces modèles conceptuels vont servir à obtenir une forme exécutable quelque soit le langage d'implémentation utilisé (java, php, c, c#, etc.). Dans ce qui suit, nous allons présenter les principes à appliquer à notre modèle événementiel que nous avons défini précédemment. On distingue trois situations :

- Réalisation de la sémantique d'un événement interne à l'aide du modèle publication-souscription (publish/subscribe),
- Réalisation de la sémantique d'un evénement externe à l'aide du modèle de messagerie point à point
- Réalisation de la sémantique d'une opération externe à l'aide du modèle de messagerie point à point (point to point).

4.6.3.1. Réalisation de la sémantique opérationnelle d'un événement interne

Le premier cas de comportement événementiel est l'interaction interne qui se déroule entre des objets qui appartiennent au même système. Ce comportement peut s'implémenter en appliquant le modèle de publication/souscription. Le principe du modèle publication souscription est simple et ressemble à ce qui se passe réellement avec les éditeurs de journaux où il y a plusieurs particuliers et entreprises qui s'abonnent à un éditeur de journaux, et par suite à chaque nouvelle publication du journal, tous les abonnés reçoivent le journal. Les abonnés représentent des *objets dynamiques* qui peuvent s'abonner ou résilier leurs abonnements, alors que l'éditeur représente *l'objet observé*. La publication du journal est représentée par un événement qui est généré par l'objet observé lorsqu'un changement d'état de l'objet est constaté.

Ainsi, dans le modèle publication/souscription, on distingue deux types d'objet particulier : l'objet observé et l'objet dynamique. L'objet observé diffuse un événement et l'objet dynamique s'abonne à l'objet observé pour constater cet événement (Figure 84). Après constatation d'un changement d'état d'une propriété, l'objet observé génère un événement qu'il diffuse aux objets dynamiques abonnés à cet événement.

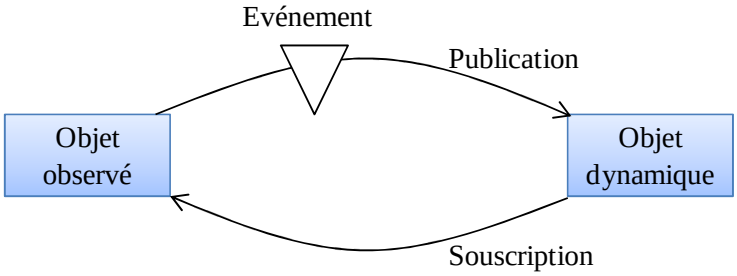


Figure 84. Le principe de gestion d'une interaction interne

Dans le modèle publication/souscription, l'événement peut être reçu par plusieurs objets, ce sont tous les objets dynamiques qui se sont abonnés à l'objet observé pour ce type d'événement. Ce modèle fait partie des modèles de conception qui visent à limiter la dépendance entre les objets. 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. En effet durant l'exécution du programme, les objets dynamiques abonnés à l'événement peuvent changer en fonction de la situation.

Les deux autres principes suivants correspondent à la gestion de messagerie.

4.6.3.2. Réalisation de la sémantique opérationnelle d'un événement externe

Dans le cas d'une interaction externe entre objets appartenant à deux systèmes différents, on applique un principe de messagerie connu sous le nom de Point à Point. Lorsque le message (l'événement) vient de l'extérieur, on parle d'une interaction entrante et dans ce cas l'objet observé est à l'extérieur du système considéré (il appartient à une application externe) quant à l'objet dynamique il appartient au système étudié (Figure 85).

Le modèle de messagerie Point à Point représente un système de production-consommation dans lequel un message est envoyé par un producteur (qui correspond à l'objet observé) et est reçu par un seul consommateur (qui est l'objet dynamique). Les messages sont

d'abord acheminés à une file d'attente appelée *Queue* et l'objet dynamique (le consommateur) viendra le chercher ensuite. Pour cela, il faut que le consommateur s'abonne à cette *Queue* qui va lui diffuser les messages.

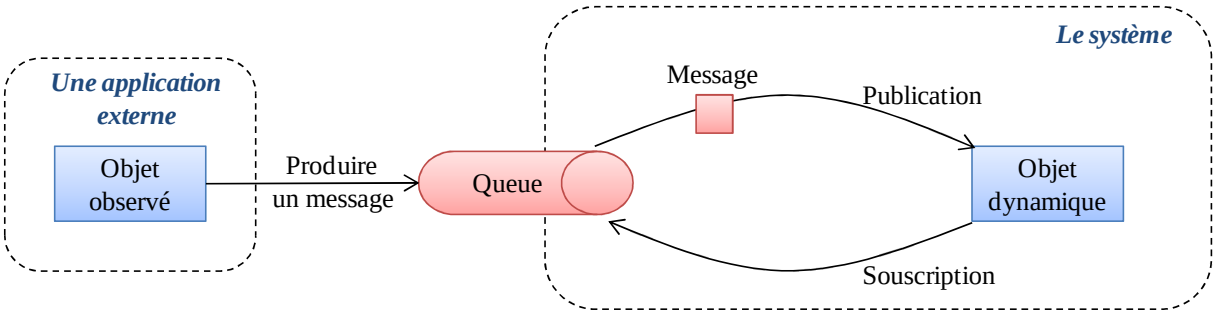


Figure 85. Le principe de gestion d'une interaction formalisée dans un événement externe

Dans la gestion d'une interaction externe avec un message entrant, ce qui nous intéresse c'est la manière avec laquelle notre système va réagir à l'arrivée de ce message. Dans ce cas, c'est la consommation du message qui est événementielle et qui peut s'implémenter par un principe de Publication-souscription. Pour cela, il est inutile de s'abonner directement à l'application externe qui émet le message, il suffit de représenter l'interaction entre l'objet consommateur (comme objet dynamique) et la file d'attente (comme objet observé) qui diffuse les messages provenant de l'application externe.

4.6.3.3. Réalisation de la sémantique opérationnelle d'une opération externe

L'interaction sortante désigne une interaction externe dans laquelle le message est produit par un objet du système pour être consommé par une application externe. C'est le cas par exemple d'une invocation d'une application externe pour exécuter une partie du processus ou le cas d'une notification d'un acteur externe par l'état actuel du système. Dans cette situation, on applique le principe de messagerie publication/souscription précédemment présenté mais à l'envers, c.à.d. l'objet observé appartient au système et l'objet dynamique fait partie d'une application externe.

Comme le montre la Figure 86, l'objet observé produit le message qui sera stocké dans une file d'attente puis consommé par l'objet dynamique. Mais dans cette situation, les opérations d'abonnement et de publication vont être gérées par l'application externe et non par le système.

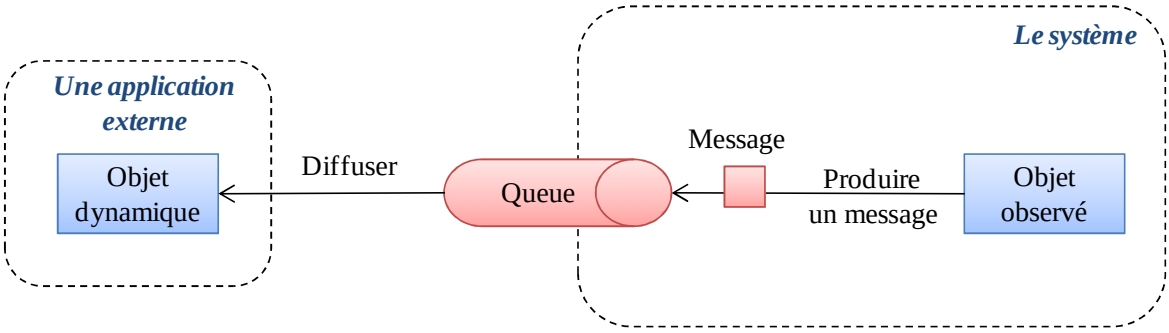


Figure 86. Le principe de gestion d'une interaction externe avec message sortant

La différence principale entre les deux modèles de messagerie, c'est que dans le modèle publication/souscription tous les consommateurs qui souscrivent à un objet observé peuvent

recevoir tous les messages publiés par cet objet, alors qu'avec le modèle Point-to-Point, seulement un consommateur reçoit un message sur une file d'attente. La réception d'un message se fait par défaut de manière **asynchrone** car l'application qui produit le message n'attend pas que l'application qui le consomme ne traite le message.

4.6.4. Règle n°1 : Cas d'un événement interne

Objectif : la règle n°1 vise à traduire la sémantique d'une interaction interne en terme de concepts d'un modèle orienté objet standard. Ceci revient à associer à chaque élément de l'interaction événementielle, qui se déroule entre des objets du même système, ce qui lui correspond dans le modèle de classes UML.

Structure d'entrée : La structure d'entrée de cette règle est la description de l'ensemble des éléments qui participent à cette interaction interne.

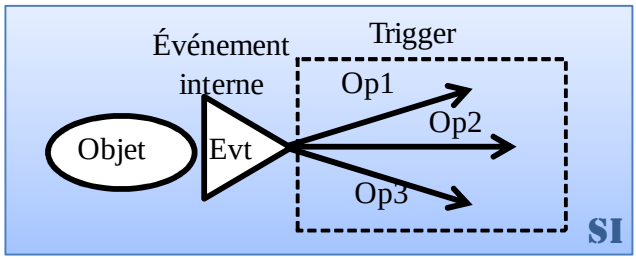


Figure 87. Schéma d'un événement interne

Ces éléments sont : l'objet qui constate l'événement interne, la structure de l'événement, et le trigger avec toutes ses opérations, ses condition et facteurs de déclenchement ainsi que les objets sur lesquels il pointe.

Principe : Pour la première règle, nous appliquons le principe de gestion du comportement d'un événement interne que nous avons présenté dans la section 4.6.3.1.

Règle de transformation : Les sous-règles sont présentées dans l'ordre de leur application comme suit:

R1. L'objet sur lequel est constaté l'événement doit être représenté par une classe (***ObservedObject***) du diagramme de classe. Cette classe doit avoir une propriété de type *State* avec une méthode de type « ***get*** » et « ***set*** » pour y accéder. Cette propriété et ces méthodes doivent être ajoutées si elles n'existent pas déjà. Un commentaire est utile également pour mentionner la liste des valeurs possibles que peut prendre cette propriété et vérifier que les valeurs utilisées dans le prédicat de l'événement sont bien mentionnées, si elles ne le sont pas, il faut ajouter les valeurs dans le commentaire.

R2. Cette classe (***ObservedObject***) doit hériter d'une classe prédéfinie ***ObjectChangeSupport*** afin d'avoir les capacités de gérer des objets dynamiques de type « écouteur » qui s'abonnement à des événements de type ***PropertyChangeEvent***. Les méthodes ***addPropertyChangeListener*** et ***removePropertyChangeListener*** permettent de gérer la liste des objets qui s'abonnent aux événements de type ***PropertyChangeListener*** pour une propriété particulière de la classe ***ObservedObject***. La méthode ***firePropertyChange*** permet de notifier la constatation du changement

d'état d'une propriété (événement), de générer un objet de type ***PropertyChangeEvent*** et de diffuser cet événement aux écouteurs abonnés à cette propriété en invoquant la méthode ***propertyChange***. Pour que ce mécanisme soit générique, les objets type « écouteur » qui s'abonnent aux événements doivent être issus d'une classe qui implémente l'interface ***PropertyChangeListener.***

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

R4. a. -TRIGGER- Le trigger de l'événement interne doit être implémenté dans une classe de type «écouteur», qu'on nomme ***DynamicObject***, qui va s'abonner aux événements associés à la propriété référencée par cet événement (On définit un seul ***DynamicObject*** par propriété modifiée). Cette classe doit implémenter l'interface <<***PropertyChangeListener***>> et notamment la méthode ***propertyChange***. Il y doit y avoir dans cette classe une méthode privée ***triggerEvt()*** pour réaliser l'exécution du traitement spécifié dans la partie trigger de l'événement interne et une méthode privée ***predicatEvt()*** pour l'évaluation du prédicat de l'événement interne. La méthode publique ***propertyChange()*** doit évaluer le prédicat de l'événement et déclencher le trigger si le prédicat est vrai. -- complétée par la règle R4.d et R4.e.

S'il existe plusieurs événements internes sur le même objet et la même propriété, il y a une seule classe de type « écouteur », une méthode privée ***triggerEvt()*** et ***predicatEvt()*** par événement et la reconnaissance de l'événement interne à exécuter en fonction de l'objet ***PropertyChangeEvent*** reçu est réalisée dans la méthode ***propertyChange***.

Les opérations Op1, Op2, et Op3 du trigger sont des méthodes des objets modifiés.

R4.b. -CONDITION- Chaque condition utilisée dans le trigger d'un événement devient une méthode ***Condition()*** retournant une valeur booléenne et prenant comme paramètres d'entrée les données issues de l'événement (***PropertyChangeEvent*** dans le cas d'un événement interne et ***MessageEvent*** dans le cadre d'un événement externe). Cette méthode sera une méthode encapsulée dans la classe ***EngineContext***. Cette méthode n'est pas une méthode locale car une condition doit pouvoir avoir accès à toutes les données du processus (objets des niveaux « type » et « instance »).

R4.c. -FACTEUR- Chaque facteur utilisé dans le trigger d'un événement devient une méthode ***Factor()*** retournant une collection d'objets et prenant comme paramètres d'entrée des données nécessaires issues de l'événement (***PropertyChangeEvent*** dans le cas d'un événement interne et ***MessageEvent*** dans le cadre d'un événement externe). Cette méthode sera une méthode de classe encapsulée dans la classe ***EngineContext***. Cette méthode n'est pas une méthode locale car un facteur doit pouvoir avoir accès à toutes les données du processus (objets des niveaux « type » et « instance »).

R4.d. Une opération associée à une condition de déclenchement doit être intégrée dans une structure algorithmique :

SI <appel de la méthode associée à la condition>

ALORS <appel de la méthode associée à l'opération>.

Ce bloc d'instructions est situé dans la méthode privée ***triggerEvt()*** de la classe <écouteur>, et complète la règle **R4.a.**

R4.e. Une opération associée à un facteur de déclenchement doit être intégrée dans une structure algorithmique

POUR CHAQUE X <appel de la méthode associée au facteur>

FAIRE <appel de la méthode associée à l'opération sur l'objet X>.

Ce bloc d'instructions est situé dans la méthode privée ***triggerEvt()*** de la classe <écouteur> et complète la règle **R4.a.**

Si l'opération est associée à un facteur et à une condition, il faut d'abord appliquer **R4.d** et ensuite **R4.e**.

R1.a. Cette règle concerne le **diagramme de séquence** et consiste à ajouter l'invocation la méthode ***firePropertyChange*** dans toutes les opérations de la classe ***ObservedObject*** qui modifient l'état d'une propriété qui est la source d'un événement interne. Elle complète la méthode de la spécification des méthodes de type ***set ()*** associées à la propriété ciblée par l'evénement interne.

R1.b. Comme le règle **R1.a**., cette règle concerne aussi le diagramme de séquence. Il faut ajouter des constructeurs à la classe ***ObservedObject*** en fonction de paramètres d'entrée ou pas. La construction d'un objet de la classe ***ObservedObject*** doit intégrer obligatoirement la création des classes de type « écouteur » définies à la règle **R4** et leur abonnement aux événements en fonction de la propriété ciblée – complète la spécification du constructeur de la classe.

R0. Une classe ***EngineContext*** est créée pour pouvoir accéder à l'ensemble des données des processus-type et l'ensemble des données d'exécution des processus. Chaque concept du modèle est à la fois une collection d'élément-type et une collection d'instance. Cette classe contiendra toutes les informations que le code aura besoin lorsque le processus va s'exécuter, en l'occurrence il contiendra l'ensemble des méthodes utilitaires tels que les conditions et les facteurs et elle est gérée en **singleton**.

Structure de sortie : Les classes obtenues après l'application de la première règle de transformation sont présentées à la Figure 88. Les classes, qui sont en gris foncé dans le schéma, représentent des classes qui sont prédéfinies et réutilisables. Elles constituent une bibliothèque de classes liées à la sémantique opérationnelle des événements internes.

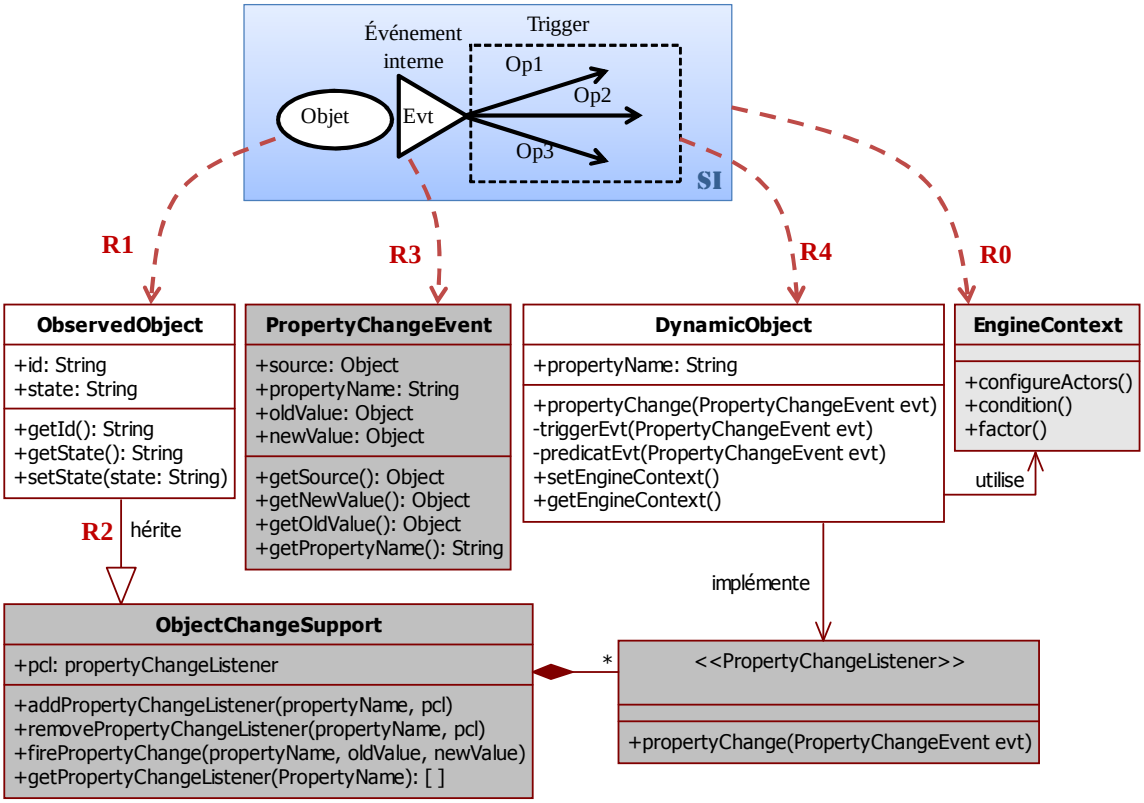


Figure 88. La règle de transformation n°1 : cas d'un événement interne

Le diagramme de classes obtenu à la Figure 45 est complété par des diagrammes de séquence qui résultent de l'application des règles R1.a et R1.b.

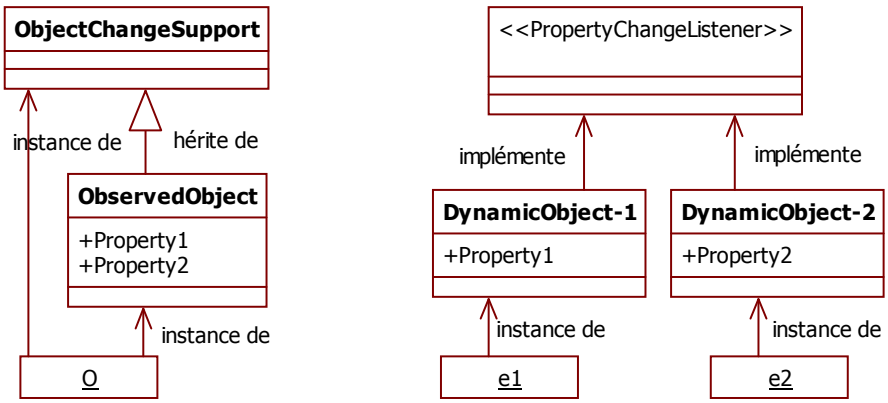


Figure 89. Un exemple générique pour la construction des objets observés et objets dynamiques

La Figure 89 présente un exemple de cas générique pour la construction d'un objet observé O et deux objets dynamiques e1 et e2 associés respectivement au changement de valeur de deux propriétés différentes (Property1, Property2) de l'objet O. Cet objet O est une instance de la classe (***ObservedObject***) qui hérite de la classe ***ObjectChangeSupport*** des opérations (telle que *addPropertyChangeListener()*) pour pouvoir gérer les objets dynamiques.

Pour cet exemple, la Figure 90 présente le diagramme de séquence correspondant à la création des objets relatifs à ce cas générique d'événement interne.

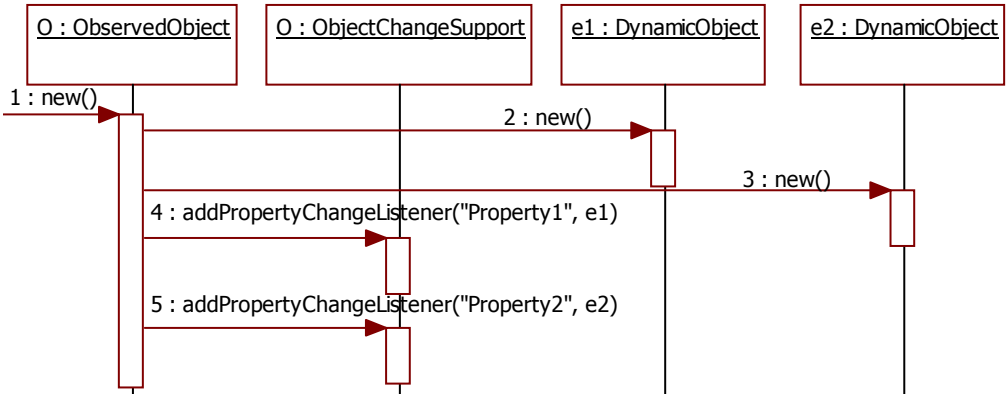


Figure 90. Le diagramme de séquence pour la création des objets relatifs à un événement interne

Une fois que les objets sont crées, le déclenchement automatique d'un événement interne se produit par l'exécution de la méthode *setState()*. Cette méthode change la valeur de la propriété de l'objet observé O par la nouvelle valeur « *newValue* » passée en paramètre car celle-ci inclut l'appel de la méthode *firePropertyChange()*.

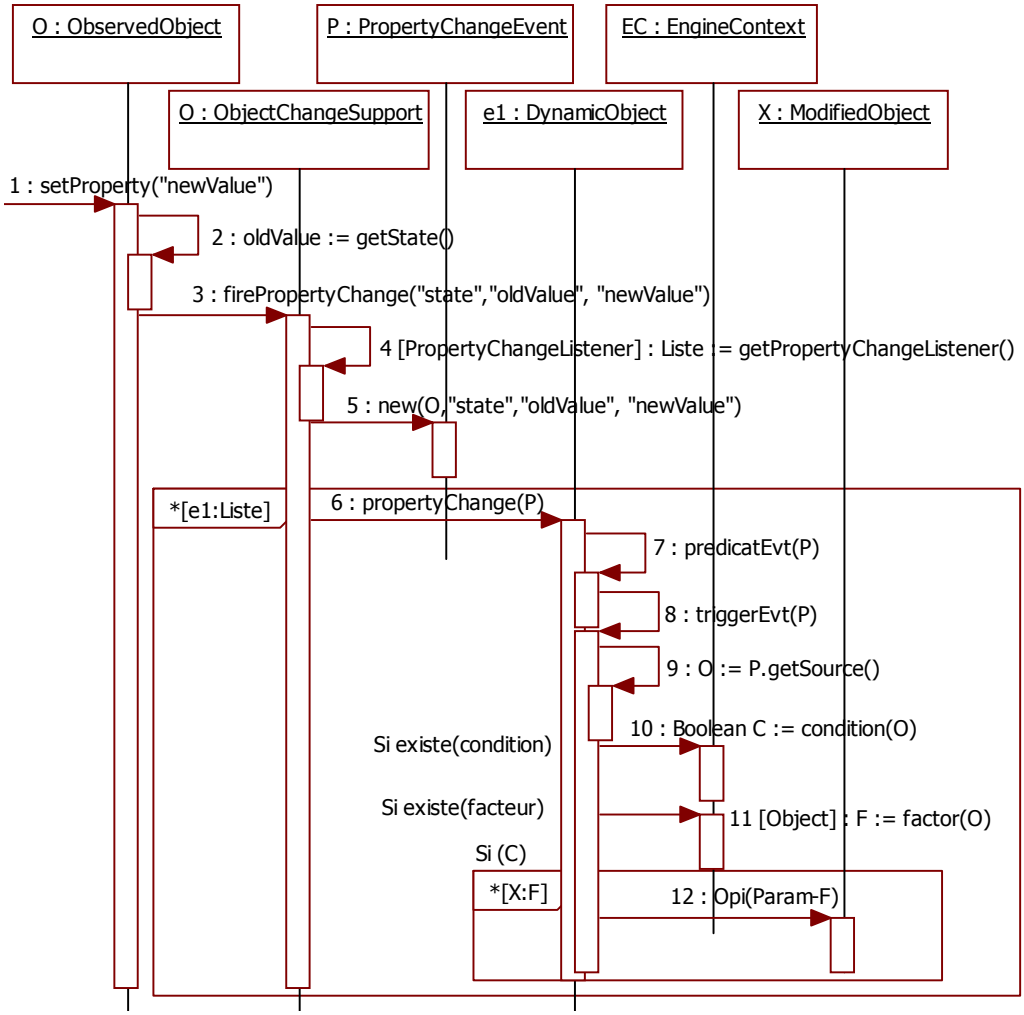


Figure 91. Le diagramme de séquence pour l'exécution d'un événement interne

La Figure 91 présente le diagramme de séquence correspondant à l'exécution d'un événement interne. Le cas où le trigger d'un événement interne contient une opération externe n'est pas dans cette partie mais traité plus tard à la section 4.6.6.

La Figure 91 montre qu'avant d'exécuter le trigger déclenché par un événement interne, la classe ***ObjectChangeSupport*** doit créer une nouvelle instance d'événement de type ***PropertyChangeEvent***, ensuite, récupérer la liste des objets de type ***PropertyChangeListener*** en appelant la méthode *getPropertyChangeListener()*. Le traitement relatif au trigger sera effectué pour chque élément de cette liste

Il est à noter que, dans notre cas, les opérations invoquées dans les diagrammes de séquence sont appelées de manière synchrone. Une intéraction synchrone représente sémantiquement un appel et un retour d'une réponse. Graphiquement, une flèche <<retour>> en pointillé apparaît à la fin de l'occurrence d'exécution (le rectangle qui est sur la ligne de vie) pour indiquer un élement de retour. Pour des raisons de simplification et de clarité des figures nous avons choisi de ne pas faire figurer les réponses (les flèches pointillées).

Exemple d'application de la règle n°1: extrait du modèle workflow

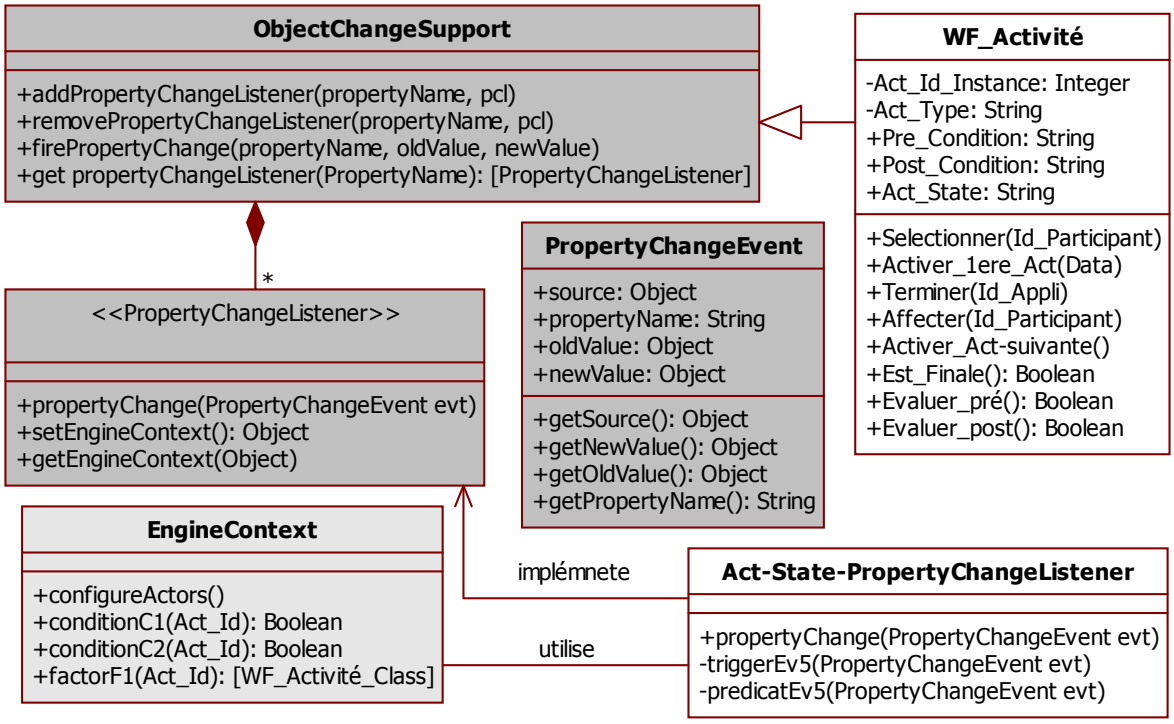


Figure 92. Un extrait du diagramme de classes de workflow après application de la règle n°1

En se basant sur le schéma dynamique du modèle de workflow de la Figure 76, nous présentons dans cet exemple le modèle UML qui résulte de l'application de la règle de transformation n°1 au comportement événementiel interne de l'objet *WF_Activité*, notamment pour l'événement *EV5*. La Figure 92 présente les classes en UML qui permettent de gérer le comportement interne de l'objet *WF_Activité*. Ces classes complètent le modèle statique du workflow présenté à la Figure 48 facilitant ainsi l'implémentation du modèle. Cette spécification peut encore être complétée avec les diagrammes de séquence générés aussi à partir de l'application des règles de transformation (voir Figure 93 et Figure 94).

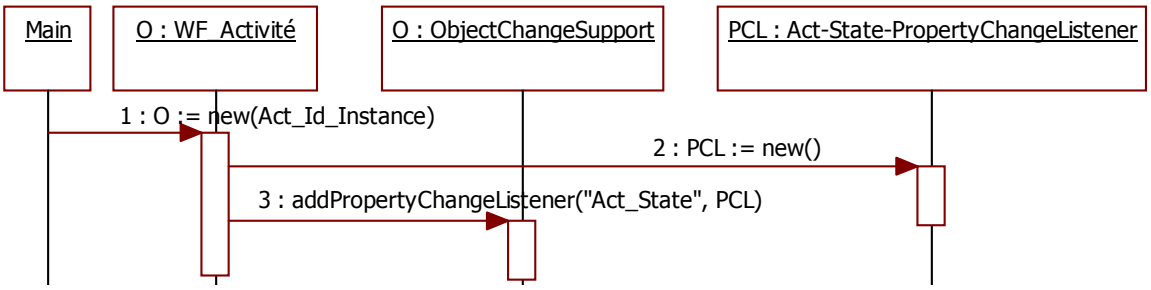


Figure 93. Le diagramme de séquence pour la création des objets correspondant à l'événement interne EV5 du workflow

Lors de la gestion de l'événement interne **EV5**, la création d'une instance O de la classe de l'objet observé ***WF_Activité*** entraine la création d'une instance PCL de l'objet dynamique ***Act-State-PropertyChangeListener***. Cette création intègre la classe ObjectChangeSupport qui permet de gérer les objets dynamiques.

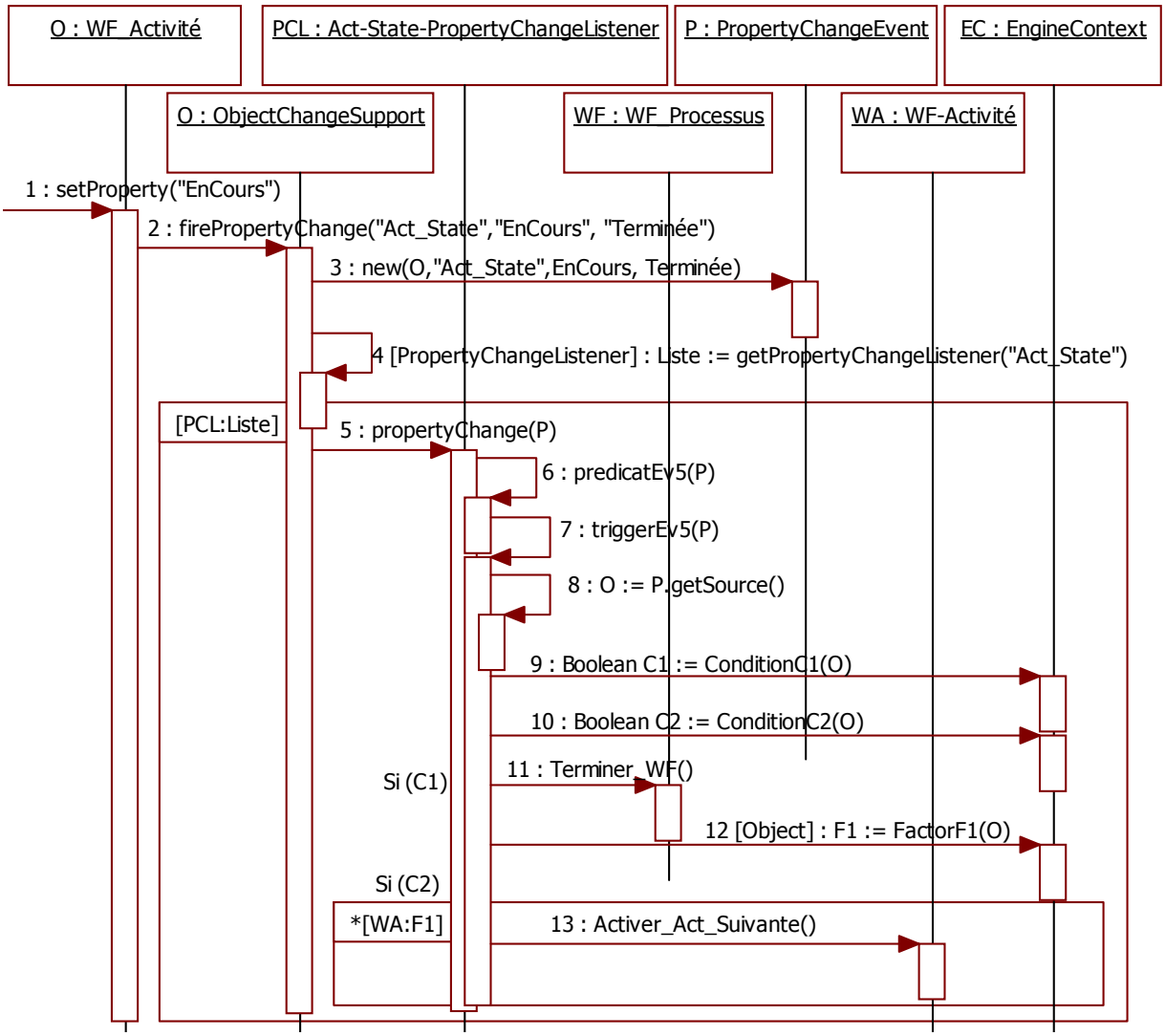


Figure 94. Le diagramme de séquence pour l'exécution de l'événement interne EV5 du workflow

Dans cet exemple, on remarque qu'il y a un seul objet dynamique (*Listener*) qui correspond à la classe *Act-State-PropertyChangeListener*. Ceci est dû au fait que nous avons qu'une seule propriété relevante dont les changements d'état correspondent à des événements

remarquables (qui est *act-State*). La règle de transformation définit un objet dynamique par propriété modifiée dans l'objet observé c.à.d. on aura autant de classes d'objets dynamiques qui implémentent l'interface <<*PropertyChangeListener*>> que de propriétés sources d'événements internes. Chacune de ces classes contient un méthode *propertyChange()* permettant d'appeler les opérations du trigger. Dans le cas de l'événement EV5, une collection (Liste) d'objets dynamiques de type *PropertyChangeListener* est retournée par la méthode *getPropertychangeListener()*, le traitement du trigger sera itéré pour chaque objet de cette Liste.

La spécification orientée objet obtenue par l'application de la règle 1, sera complétée par le résultat de l'application des règles de transformation correspondants aux interactions avec des élements externes. On distingue entre deux cas :

- Cas d'un événement externe avec réception d'un message
- Cas d'une opération externe avec envoie de message à un acteur externe

4.6.5. Règle n°2: Cas d'un événement externe avec réception d'un message

Objectif : cette règle permet de représenter sous forme de diagrammes OO standard un comportement événementiel du système dans le cadre d'une interaction avec son environnement. Il s'agit du cas d'une action provenant d'un acteur externe et qui déclenche des opérations agissant sur les objets du système comme le schématise la Figure 95.

Structure d'entrée : cette règle sera appliquée sur la spécification dynamique telle que celle que nous avons présentée dans la section 4.5.1.2.2

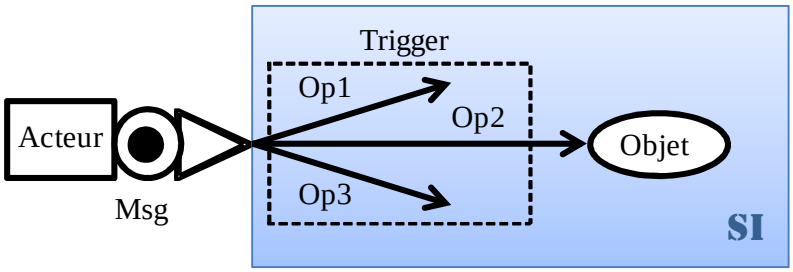


Figure 95. Le schéma d'un événement externe : action provenant d'un acteur

Principe : Le principe de messagerie *Point à Point*, introduit dans la section 4.6.3.2, est choisi pour ce cas. Ce modèle de messagerie s'adapte le mieux avec ce type d'événement. En effet, dans la réalité, le système peut subir à la fois plusieurs actions provenant d'acteurs différents. Pour cela, on considère chaque acteur comme un producteur de messages et on représente les objets du système en tant que consommateurs de ces messages.

Règle de transformation :

R5. La réception d'un message envoyé par un acteur est représentée par un objet instance de la classe globale ***MessageConsumer*** qui permet d'avoir une file d'attente par type de message à recevoir. Par conséquent, il y aura autant d'instances de la classe ***MessageConsumer*** qu'il y aura d'événements externes et d'instances d'acteur. La réception de ce message est réalisée de manière événementielle avec l'abonnement de la

classe de type « écouteur » aux événements de type ***MessageEvent***. La propriété source de ce dernier permet de spécifier le nom de la file d'attente, utilisé pour synchroniser sur la même file d'attente le consommateur et le producteur de message.

R6. A chaque arrivée de message détectée automatiquement par le système, un événement est créé et est diffusé à tous les objets abonnés. Cet événement est implémenté dans une classe ***MessageEvent.***

R6.a. La classe ***MessageEvent*** représente de manière générique tous les événements externes du schéma dynamique. Un événement de type ***MessageEvent*** est défini par l'objet ***message*** et l'objet ***source*** qui représente l'objet de type ***MessageConsumer*** reçevant le message. Les méthodes : ***setMessage (Object o)*** et ***getMessage():Object*** permettent de gérer dans cette classe n'importe quelle structure de message.

R6.b. Le message reçu par l'événement externe doit être représenté par une classe ***Message*** permettant de stocker l'ensemble des propriétés du message avec les méthodes de type « ***get*** » et « ***set*** » correspondantes.

R7. Associer au groupe d'opérations déclenchées par un événement externe (c.à.d. au trigger) une classe ***DynamicObject*** qui implémente l'interface <<***MessageListener***>>. Cette classe va s'abonner aux événements générés par ***MessageConsumer*** (qui représente la file d'attente correspondante à l'événement externe) et invoquera la méthode ***onMessage***() qui activera les méthodes correspondantes aux opérations.

R7.a. Il doit y avoir dans la classe ***DynamicObject*** une méthode privée ***triggerEvt()*** pour réaliser l'exécution du traitement spécifié dans la partie trigger de l'événement externe. La méthode publique ***onMessage()*** doit évaluer le prédicat de l'événement externe et déclencher le trigger si le prédicat est vrai.

R7.b. Une méthode publique ***predicatEvt()*** définie dans ***EngineContext*** permettra d'évaluer la validité du message reçu avant de déclencher le trigger associé à l'événement externe. La méthode retourne une valeur booléenne et prend un objet de la classe ***Message*** en paramètre d'entrée. Elle est encapsulée dans la classe ***EngineContext*** car elle peut avoir besoin d'accéder aux données du processus comme les conditions et les facteurs.

R7.c. Appliquer les règles **R4.a**, **R4.b**, **R4.c** et **R4.d** pour traiter les conditions et les facteurs du trigger de l'événement externe.

R8. La création de l'objet de type ***MessageConsumer*** et d'un objet de la classe ***DynamicObject*** générée à la règle **R7.a** ainsi que l'abonnement de celui-ci au ***MessageConsumer*** doivent être intégrés à la méthode *configureActors()* de la classe ***EngineContext*** pour pouvoir mettre en place le dispositif de réception de message et de déclenchement automatique du traitement associé. La liste des ***MessageConsumer*** relatifs à chaque acteur et à chaque événement sera stockée au niveau de la classe ***EngineConexte.*** Le lien entre la classe ***EngineContext*** et celle de ***MessageConsumer*** à la Figure 96 traduit cette opération de stockage qui est réalisée à travers la méthode *addMessageConsumer (« nom », MessageConsumer)* permettant justement d'ajouter

ces objets de type ***MessageConsumer*** dans un tableau ordonné par un nom de la forme « Acteur/Event » précisant l'événement et l'acteur qui l'envoie.

Structure de sortie :

L'application de la règle de transformation n°2 génère les classes schématisées à la Figure 96 ainsi que les diagrammes de séquence représentés à la Figure 97 et à la Figure 98.

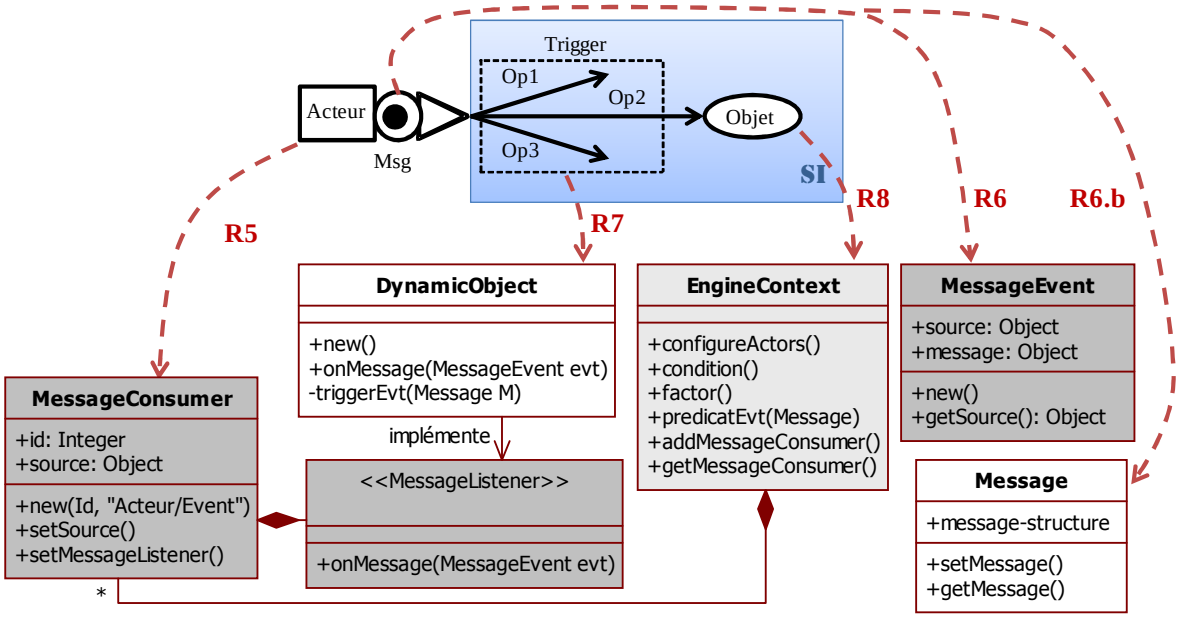


Figure 96. Les règles de transformation dans le cas d'un événement externe

A la Figure 97, nous présentons le diagramme de séquence qui correspond à la création des objets à l'arrivée d'un événement externe.

La création de l'objet ***MessageConsumer*** par l'opération *configureActors()* nécessite d'avoir en paramètre un identifiant « *id* » et une source « *Acteur/Event* » qui renseigne sur l'acteur qui a crée l'événement. Cette source change en fonction de l'acteur et de l'événement. L'abonnement de l'objet dynamique (DO) à la réception du message de la file d'attente du ***MessageConsumer*** (MC) se fait au niveau de la classe globale ***EngineContext*** à travers l'opération *addMessageConsumer()*. Les objets ***MessageConsumer*** seront stockés dans un tableau ordonné par le nom de la source au niveau de la classe ***EngineContext***. Ce tableau est représenté au niveau du diagramme de classes de la Figure 96 par le lien de composition entre la classe ***EngineContext*** et la classe ***MessageConsumer***.

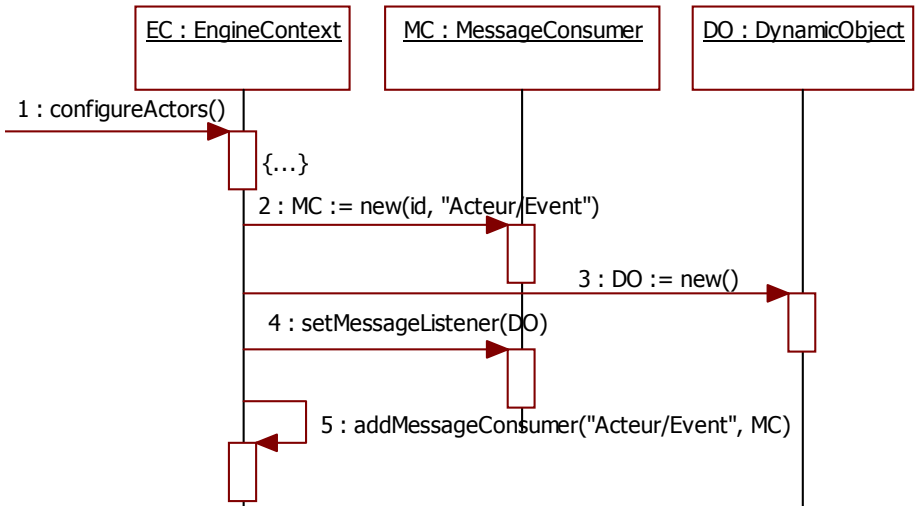


Figure 97. Extrait du diagramme de séquence pour la création des objets suite à l'arrivée d'un événement externe

A la Figure 98, le diagramme de séquence précise l'interaction entre les objets créés à fin d'exécuter le traitement déclenché suite à l'arrivée d'un événement externe.

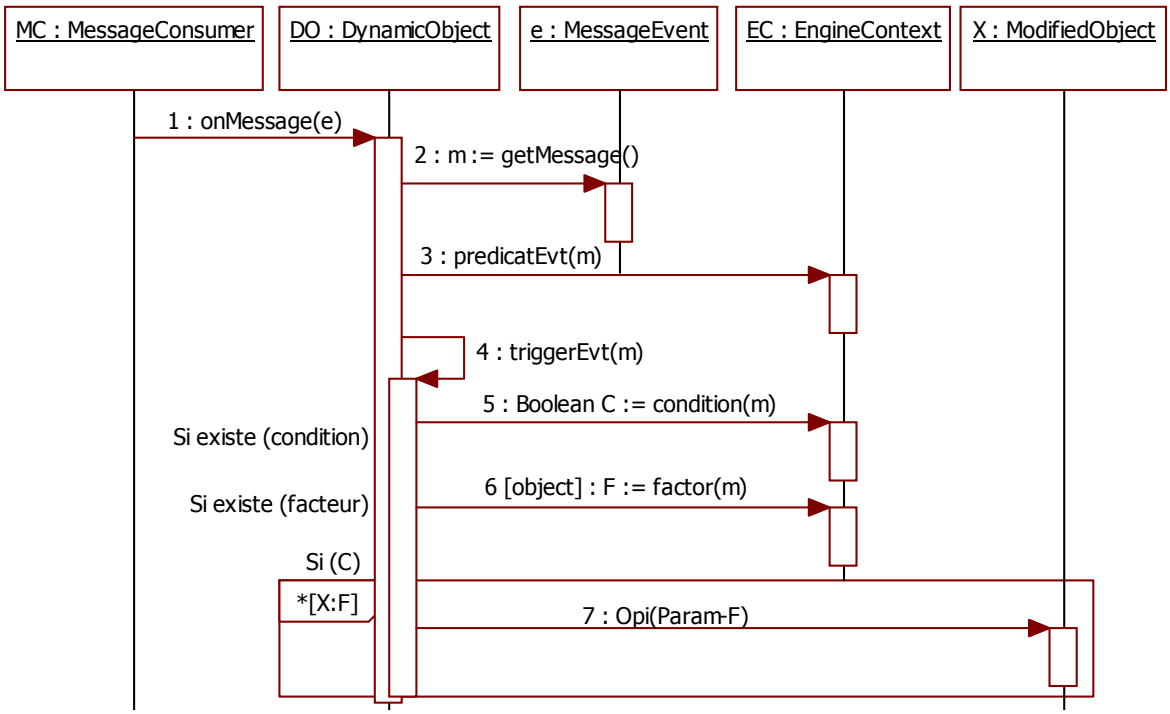


Figure 98. Le diagramme de séquence pour l'exécution d'un événement externe

L'application de la règle de transformation d'un événement externe se place du coté du consommateur et n'intègre pas la partie 'producteur' du patron Publication/Souscription. En effet, la partie 'producteur' est jouée par l'acteur externe qui se situe dans l'environnement du moteur et non dans le moteur lui-même.

Exemple d'application de la règle n°2: extrait du modèle workflow

L'événement EV1 (Figure 76) correspond à une action de démarrage de l'exécution d'un processus workflow par un acteur externe. Voici la description de cet événement :

EV1 : Le démarrage d'un processus <WF_Processus> consiste à :

- activer la première <WF_Activité> du <WF_Processus>,
- initialiser l'état de cette < WF_Activité > à « activée »,

L'application des règles (R5->R8) dans le cas de l'événement EV1 donne le résultat présenté à la Figure 99. Dans cet exemple, on a associé à l'événement EV1 une classe ***M1*** et au trigger correspondant à cet événement la classe dynamique ***Ev1-DynamicObject***. Cette classe contient la méthode *onMessage()* qui déclenche les opérations *Démarrer()* et *Activer_1ere_Act()* pour modifier les états des deux objets: *WF-Process et WF_Activity.*

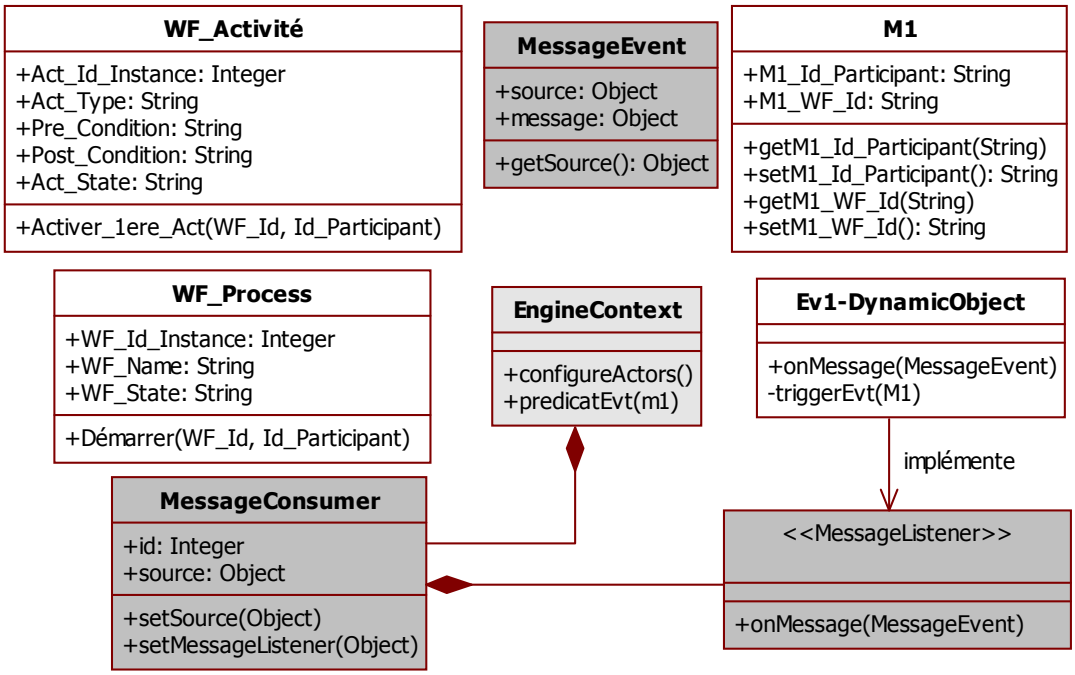


Figure 99. Un extrait du diagramme de classes de workflow obtenu par l'application de la règle n°2

L'application de la règle de transformation n°2 permet également de générer des diagrammes de séquence qui permettent de représenter l'aspect dynamique d'une interaction interne concernant l'initialisation et la création des objets (Figure 100) et concernant l'exécution de l'événement (Figure 101).

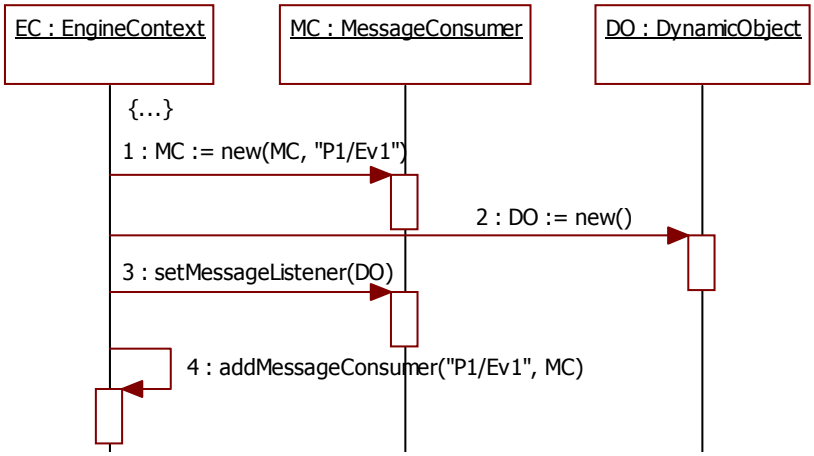


Figure 100. Extrait du diagramme de séquence obtenu par l'application de la règle n°2 pour l'initialisation de l'événement externe EV1 du modèle de workflow

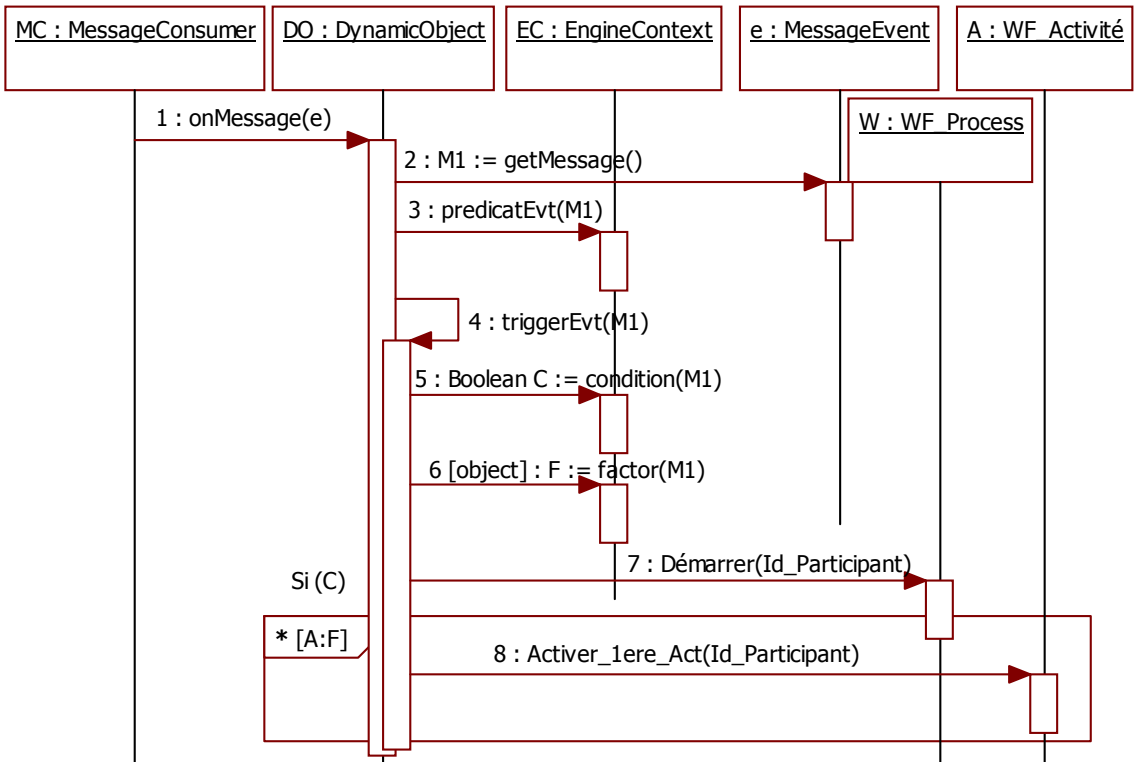


Figure 101. Le diagramme de séquence pour l'exécution de l'événement externe EV1 du modèle de workflow

4.6.6. Règle n°3 : Cas d'une opération externe vers un acteur externe

Objectif : Cette règle n°3 a pour objectif de transformer l'opération déclenchée dans une interaction externe sortante en son équivalent en termes de concepts UML. Cette interaction vise à implémenter un comportement dans lequel le système invoque des éléments de son environnement pour effectuer quelques tâches ou simplement pour leur informer de l'état actuel du système.

Structure d'entrée : La structure d'entrée de cette troisième règle correspond à la description de l'opération qui est déclenchée dans le cadre d'un trigger par un objet interne à destination d'un objet externe.

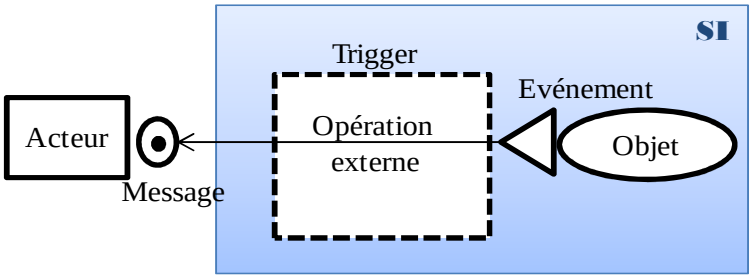


Figure 102. Le schéma d'une opération externe

Principe : Le cas d'une opération externe avec message invoquant/notifiant un acteur externe peut être implémenté par le principe présenté dans la section 4.6.3.3 et qui traite la gestion d'une interaction sortante. Il s'agit d'une interaction entre le système et son environnement dans laquelle un objet du système envoie un message à un acteur externe. La

transformation dans ce cas est schématisée à la Figure 103 et le principe de base de messagerie reste le même que dans le cas d'un événement externe sauf que dans cette situation les rôles sont inversés par rapport au cas précédent, c.à.d. l'objet représente le producteur du message et l'acteur externe représente le consommateur de ce message.

Règle de transformation :

R9. Dans le cas d'une opération externe qui est liée à un acteur externe on distingue entre deux cas qui dépendent de la représentation de l'acteur consommateur du message:

- le cas où l'acteur est représenté par un concept qui fait partie du méta-modèle, et
- le cas où l'acteur n'est pas représenté par un concept au niveau du méta-modèle

L'envoie d'un message à travers une opération externe sera traité de manière différente dans les deux cas.

R9.a. ***Cas1 : L'acteur externe est représenté par un concept du modèle nommé X.***

- Ajouter dans la classe ***X*** au niveau instance une ***collection*** d'objets de type ***MessageProducer*** indexée sur le nom du message que l'acteur doit être capable de recevoir.
- Ajouter une méthode à la classe ***X*** pour pouvoir insérer un couple <nom, objet de type ***MessageProducer***> dans cette collection.
- Ajouter aussi une méthode ***findMessageProducer*** (String nom) qui retourne l'objet ***MessageProducer*** associé au nom.
- Le constructeur de la classe ***X*** doit initialiser la collection de ***MessageProducer*** en prenant les informations nécessaires en paramètre d'entrée.
- La création des instances de la classe ***X*** et leur configuration (***MessageProducer***) doivent être réalisées par la méthode *configureActors()* de la classe ***EngineContext.***
- Il y aura autant d'instances de la classe ***MessageProducer*** qu'il y aura de type de messages à envoyer. L'envoi de ce message est réalisé par la méthode ***sendMessage(Message).*** La propriété *destination* permet de spécifier la boite aux lettres dédiée à l'envoi d'un message spécifique. Elle est utilisée pour synchroniser sur la même boite aux lettres le consommateur et le producteur de message.
- La classe Message est une classe abstraite qui sert de classe générique pour toutes les classes qui vont implémenter les message-types à envoyer.

R9.b. ***Cas 2 : L'acteur externe n'est pas représenté par un concept du modèle.***

- Ajouter un objet de type ***MessageProducer*** comme une variable d'instance globale de ***EngineContext*** avec comme nom le nom de l'acteur externe.
- Ajouter une méthode de type «***get »*** et ***« set »*** sur cette variable d'instance.
- Initialiser la variable d'instance dans la méthode *configureActors()* de la classe ***EngineContext***.

R10. Associer à l'opération externe une classe ***DynamicObject*** avec une méthode *triggerEvt()* et une autre correspondante à l'opération externe même.

R11. Le message envoyé par l'opération externe est défini dans une classe héritant de la classe ***Messsage*** et il est accessible grâce aux méthodes *getMessage()* et *setMessage().*

R11.a. Il est possible de compléter ces règles relatives à l'opération externe par une règle qui génère un diagramme de séquence comme c'est le cas pour les règles **R1.a** et **R1.b.**

Le but du diagramme de séquence et de préciser comment les objets interagissent par les méthodes entre eux et de fournir un squelette plus complet du programme.

Structure de sortie :

Cas1 : Dans le cas où l'acteur est un concept du modèle de processus, le résultat obtenu par l'application de la règle n°3 correspond aux classes schématisées à la Figure 103 et aux diagrammes de séquence présentés dans les figures qui la suivent.

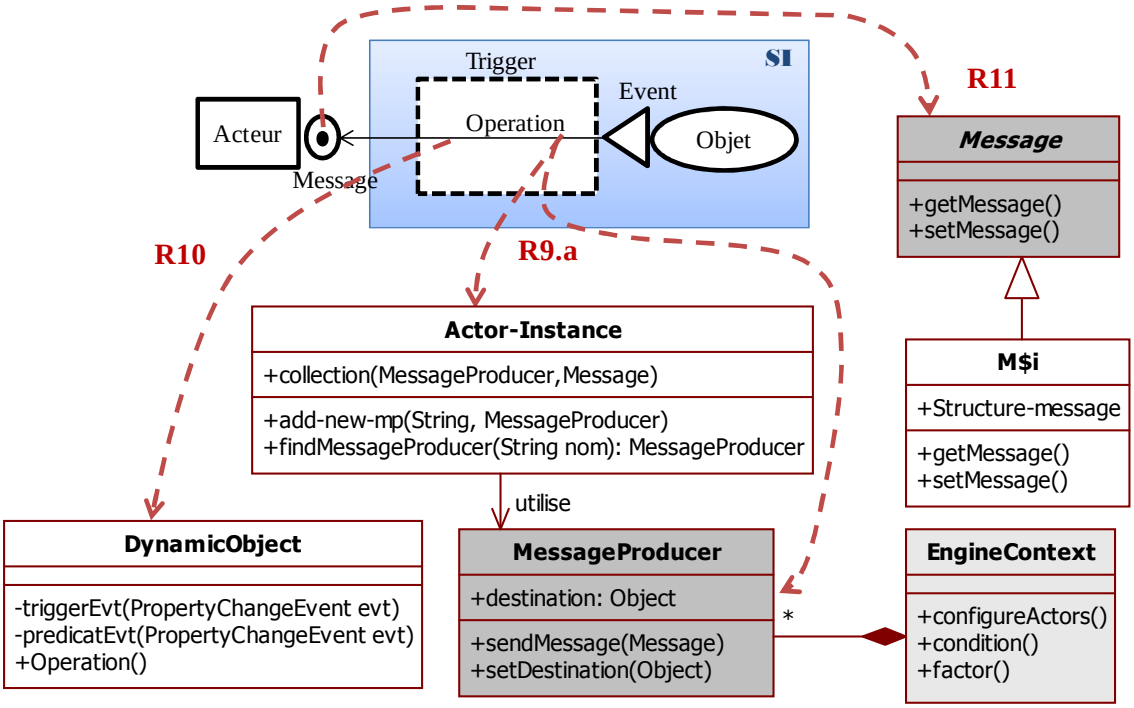


Figure 103. La règle n°3 pour la transformation d'une opération externe (Cas1)

Le diagramme de séquence correspondant à la création et à l'initialisation des objets dans le premier cas de la 3ème règle est présenté à la Figure 104.

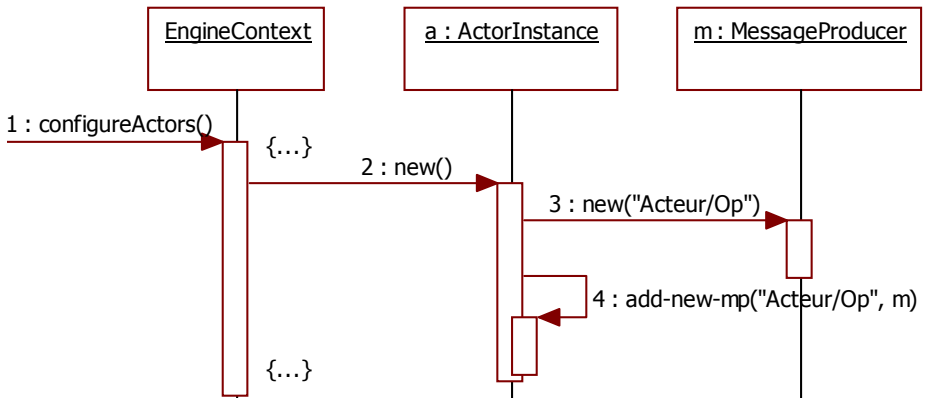


Figure 104. Extrait du diagramme de séquence pour la création des objets lors d'une opération externe (Cas1)

Comme le montre le diagramme à la Figure 104, c'est la classe ***EngineContext*** qui se charge de créer une collection d'instances de type d'acteur « *ActorInstance* » en passant en paramètre de la méthode *new()* l'ensemble des données nécessaires à cette création. Ensuite, pour chaque instance d'acteur, un ensemble d'objets de type ***MessageProducer*** sera créé tout en mentionnant pour chacun le nom « Acteur/Op ». Le lien « utilise » qui figure dans le diagramme de classes de la Figure 103 correspond à la méthode *add-new-mp()* de ce diagramme de séquence, celle-ci a pour rôle d'associer le ***MessageProducer*** créé à l'acteur qui va recevoir le message émis par l'opération externe.

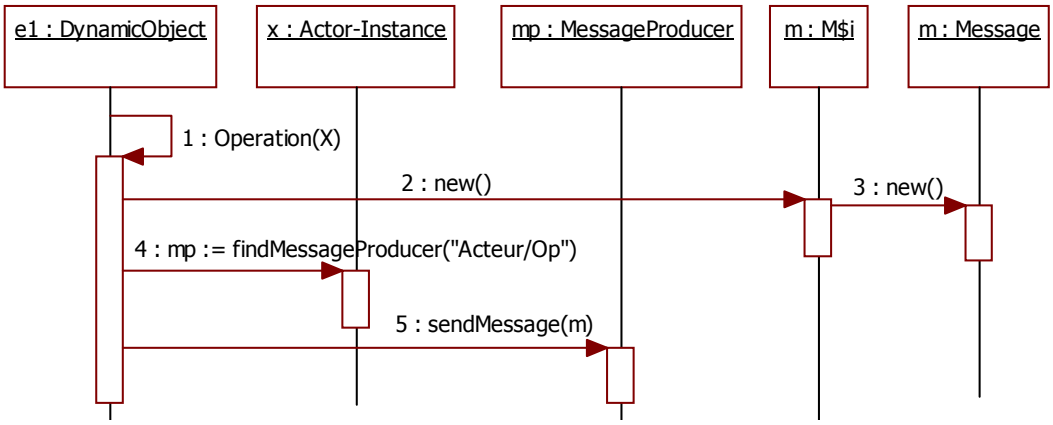


Figure 105. Le diagramme de séquence relatif à l'exécution d'une opération externe (Cas 1)

Le deuxième diagramme de séquence présenté à la Figure 105 complète aussi la structure de classes faisant partie de l'architecture de l'outil cible. Ce diagramme correspond au traitement exécuté à la suite du déclenchement d'une opération externe détectée par la méthode *Operation()*. A l'arrivée de cette opération, on crée le message, ensuite, on cherche la destination (qui est *MessageProducer*) auprès de la classe *ActorInstance* à travers la méthode *FindMessageProducer()*, enfin, on envoie le message à cette destination en invoquant la méthode *sendMessage(m)* de la classe *MessageProducer*.

Les figures (Figure 103, Figure 104, et Figure 105) permettent de traiter le cas d'une opération externe concernant un acteur représenté par un concept du méta-modèle de processus. Nous allons maintenant traiter le cas où l'acteur n'est pas représenté au niveau de ce méta-modèle.

Cas2 : Le diagramme de classes obtenu par l'application de la 3ème règle de transformation, dans le cas où l'acteur ne fait pas partie du modèle structurel de processus, est légèrement différent par rapport au premier cas et il est présenté à la Figure 106.

Dans cette situation, la gestion de la construction des ***MessageProducer*** ne se fait pas de la même manière puisque c'est dans la méthode *configureActors*() de la classe ***EngineContext*** qui les crées et non pas dans le constructeur de la classe *ActorInstance*.

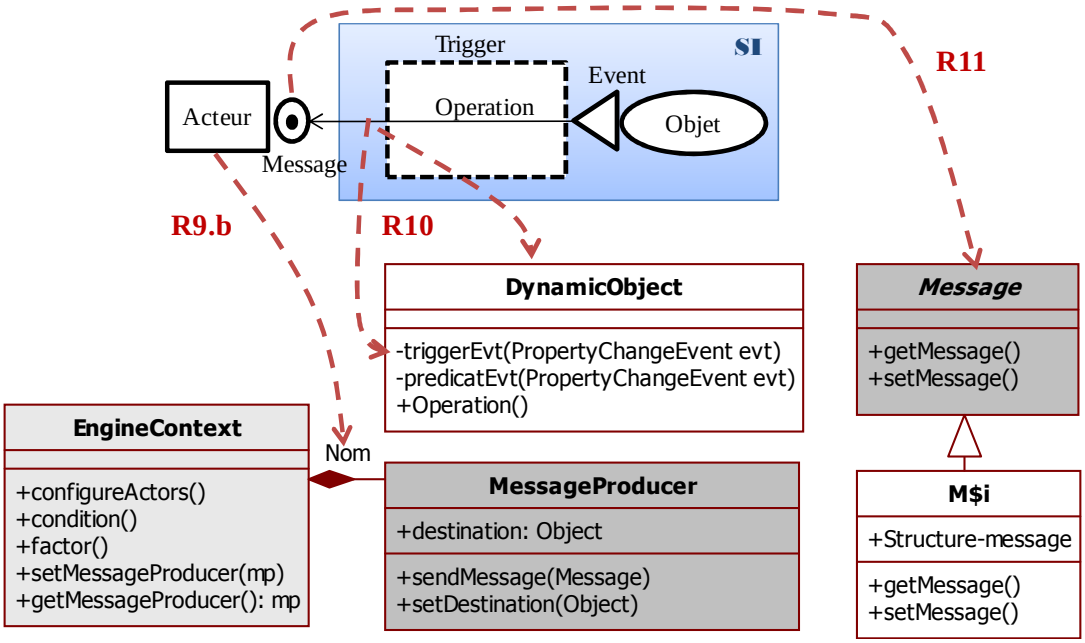


Figure 106. La règle n°3 pour la transformation d'une opération externe (Cas2)

La Figure 107 montre un extrait du diagramme de séquence qui correspond à la création des objets dans le deuxième cas de la règle de transformation n°3. Cet extrait est à intégrer au constructeur de la classe ***EngineContext.***

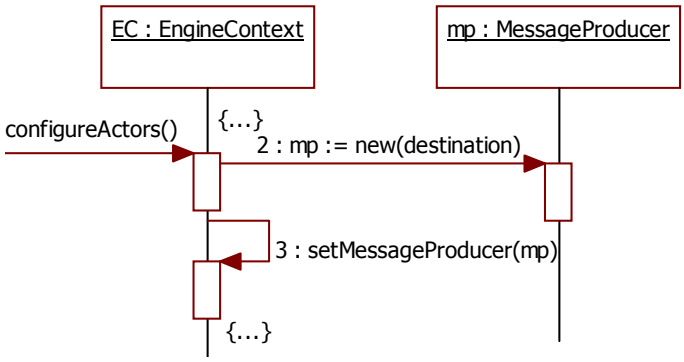


Figure 107. Extrait du diagramme de séquence pour la création des objets lors d'une opération externe (Cas2)

En ce qui concerne l'exécution de l'opération externe, l'application de la règle de transformation génère le diagramme de séquence présenté à la Figure 108.

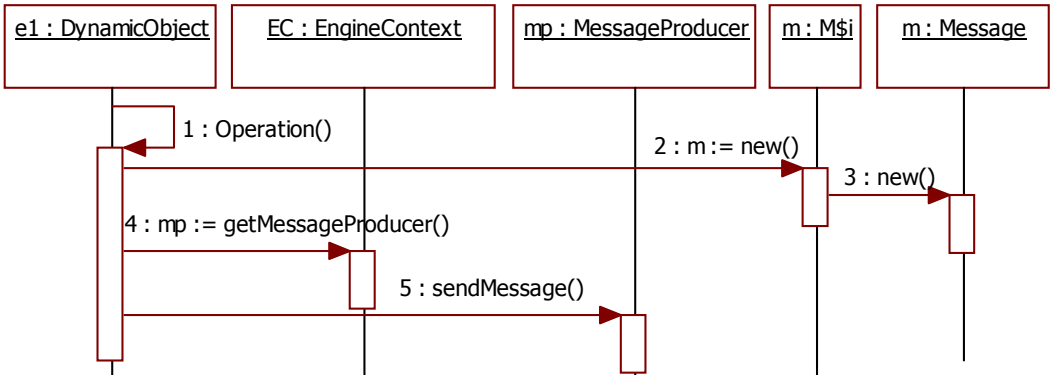


Figure 108. Le diagramme de séquence relatif à l'exécution d'une opération externe (Cas 2)

La différence par rapport au premier cas réside dans l'invocation de la méthode *getMessageProducer()* qui permet de récupérer l'objet *MessageProducer* afin de pouvoir lui envoyer le message de l'opération externe à travers la méthode *sendMessage(m).*

Exemple d'application de la règle n°3: extrait du modèle workflow

Pour illustrer la règle de transformation n°3, nous revenons sur un exemple d'opération externe extrait du schéma dynamique du modèle de workflow et présenté à la Figure 76. La classe ***MessageProducer*** représente l'objet qui va envoyer le message M2 dont la structure est définie dans l'objet ***M2***. L'acteur externe *WF_Participant* fait partie du modèle statique ce qui signifie que nous allons appliquer le 1er cas de cette troisième règle de transformation. Cet acteur est référencé par l'attribut destination de la classe ***MessageProducer.***

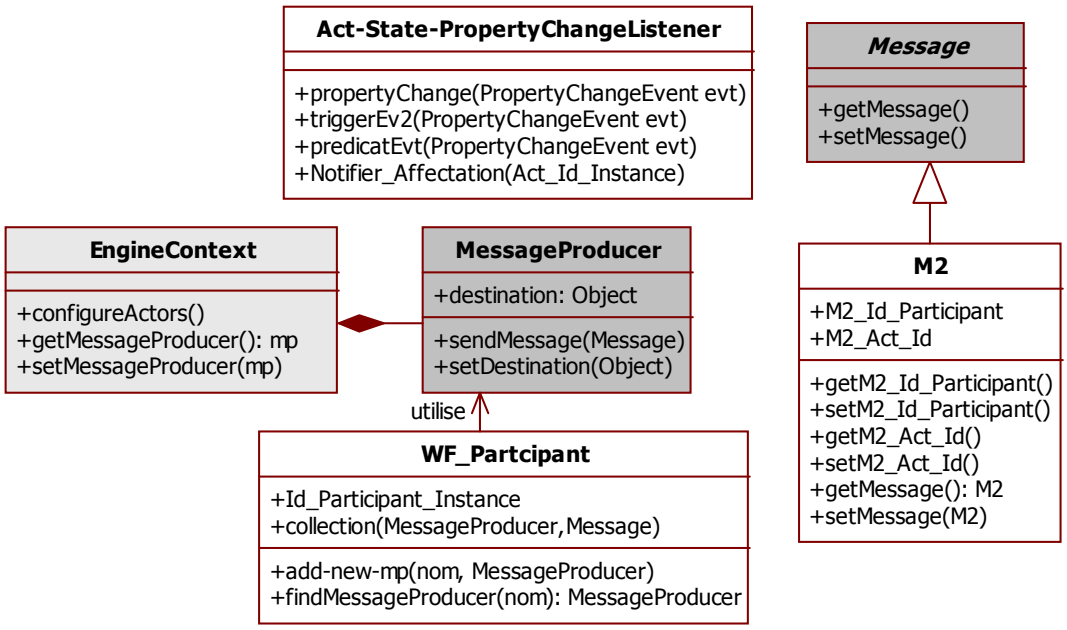


Figure 109. Un extrait du diagramme de classes de workflow après application de la règle n°3 : Exemple de l'opération externe déclenchée par l'événement EV2

Ce diagramme de classes est complété par le diagramme de séquence de la Figure 110 qui permet de préciser l'étape de la création des objets nécessaires à cette opération externe.

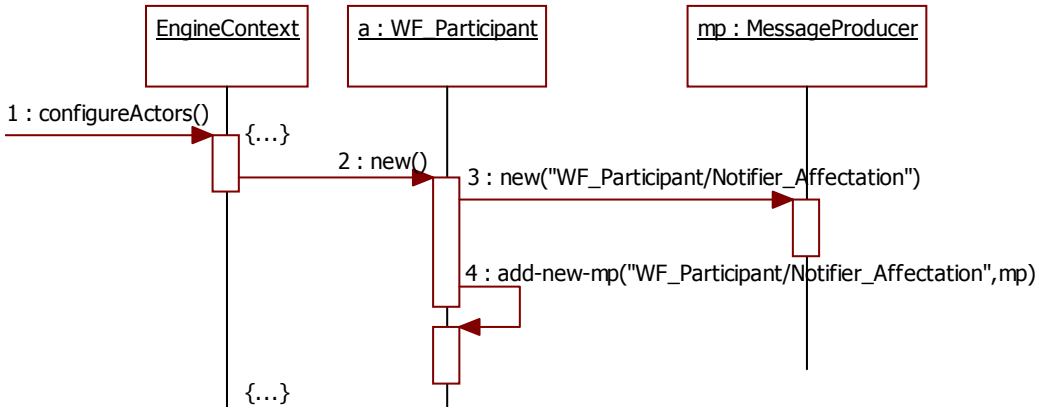


Figure 110. Un extrait du diagramme de séquence pour l'opération externe Notifier-Affectation dans l'exemple du workflow

L'opération externe *Notifier_Affectation(Object)* a comme paramètre un objet de type ***WF-participant*** ; elle entraine d'abord le déclenchement de la création du message ***M2,*** et ensuite de l'opération *findMessageProducer(MessageM2)* qui aura le message ***M2*** comme paramètre et qui retourne l'objet ***MessageProducer*** associé à ce message à partir de la collection qui se trouve dans la classe acteur (***WF_Participant***).

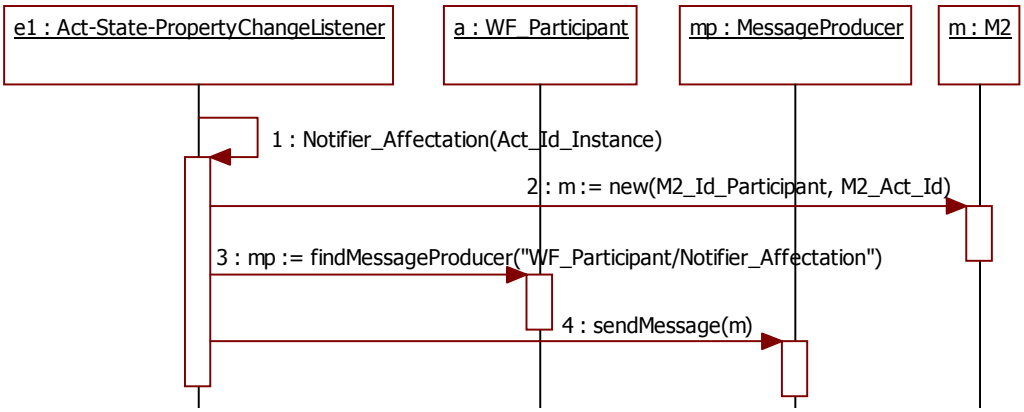


Figure 111. Un exemple de diagramme de séquence relatif àl'exécution d'une opération externe (Cas1)

Ce deuxième diagramme de séquence doit être intégré au diagramme de l'événement EV2 qui déclenche l'opération externe *Notifier_Affectation().*

4.6.7. Règles spécifiques à EngineContext

Les règles de transformation précédentes sont liées à la sémantique opérationnelle des concepts du modèle Remora étendu et sa transcription dans une structure objet standard. Nous introduisons dans cette section, des règles annexes qui permettent de mettre en œuvre la structure à deux niveaux et l'accès aux données relatives au méta-modèle de processus et processus afin d'en dériver une architecture objet de l'outil d'exécution.

R12. *Gestion des liens de référence entre les classes des deux niveaux :* Ces liens de référence doivent être gérés de manière bidirectionnelle. Le lien entre la classe de niveau « instance » et la classe de niveau « type » est implémenté par un pointeur sur un objet (un seul objet) (R12.a) alors que le lien entre classe type et la classe instance doit être implémenté par une collection d'objets des classes du niveau « type » (R12.b).

R12.a : Une classe générique et racine des classes du niveau « type » appelée ***ClasseType*** est prédéfinie et permet de gérer la collection d'instances pour un objet d'une classe de niveau « type ». Cette collection s'appelle ***instances*** et la méthode pour accéder aux instances s'appelle ***getInstances()***. De plus, le constructeur associé à la classe ***ClasseType*** doit initialiser la collection ***instances*** à vide mais prête à accueillir des instances de ce nouvel objet. Enfin, il faut au minimum deux méthodes pour ajouter et supprimer une instance dans la collection (***addInstance()*** et ***removeInstance()***)

R12.b : Une classe générique et racine des classes du niveau « instance» appelée ***ClasseInstance*** est prédéfinie et permet la référence sur l'objet correspondant à sa classe au niveau « type ». Cette référence s'appelle ***class*** et la méthode d'accès s'appelle

getClass(). Le constructeur de la ***ClasseInstance*** a en paramètre d'entrée un objet de ***ClasseType*** et il initialise la référence ***class*** de l'objet ***this*** avec ce paramètre et ajoute ensuite l'objet ***this*** dans la collection ***instances*** de l'objet représentant sa classe par la méthode ***addInstance ().***

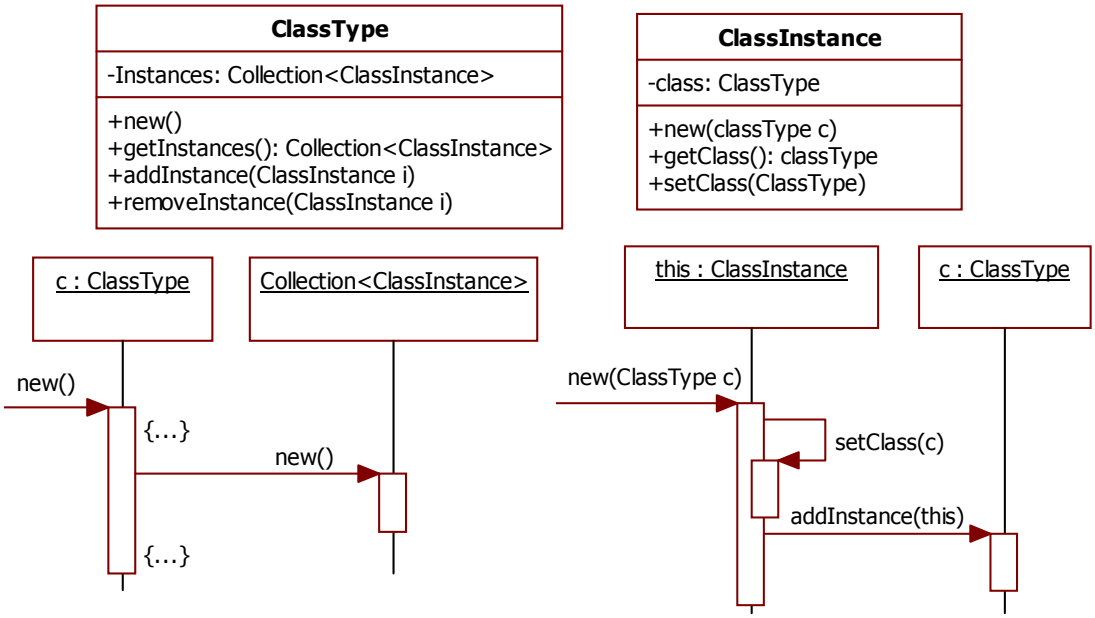


Figure 112. Les classes et les diagrammes générés par l'application de la règle R12 pour la gestion des liens de référence entre les classes de la structure à deux niveaux

La Figure 112 permet de décrire les variables d'instances et méthodes d'instances des deux classes ***ClasseType*** et ***ClasseInstance*** et les diagrammes de séquence relatifs à leur constructeur. Il permet aussi de gérer la cohérence de la relation bidirectionnelle dans le cas de la construction d'une instance du niveau « instance ». La suppression doit également être prise en compte en utilisant la méthode ***removeInstance().***

R13. *Définition par extension des classes du niveau « type » et du niveau « instance »*. Chaque classe du niveau « type » et « instance » doit permettre d'accéder à la collection d'objets construits à partir de cette classe. En effet, nous avons besoin d'accéder à l'ensemble des données du modèle de processus (niveau « type ») ou du processus (niveau « instance »). Pour réaliser cela, nous ajoutons à chaque classe **X** du schéma : une variable de classe privée ***collection*** permettant de regrouper les instances et les méthodes de classe : ***getCollection(), addInstance(), removeInstance()*** et le constructeur qui doit intégrer l'ajout de l'objet nouvellement construit (***this***) dans la variable de classe ***collection***.

La Figure 113 permet de décrire la variable de classe et les méthodes de classe qu'il faut ajouter à chaque classe **X** du schéma afin d'accéder à toutes les données relatives au modèle de processus mais aussi au processus.

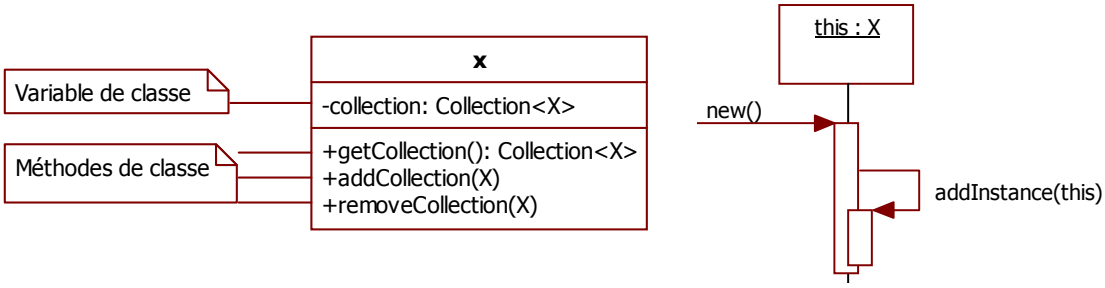


Figure 113. ***Règle de transformation pour la gestion des classes du niveau « type » et du niveau « instance »***

Il est à noter que le constructeur de la classe X doit appeler la méthode de classe *addInstance()* afin d'ajouter l'objet qui vient d'être crée dans variable de classe collection<X>.

R14. *Chargement du modèle à exécuter et sauvegarde du processus:* Une méthode de chargement du modèle de processus à exécuter doit être implémentée dans la classe ***EngineContext*** et intégrée dans son constructeur. Cette méthode s'appelle ***loadProcessModel (String fileName)*** et permet de charger les objets du niveau « type » à partir du nom du fichier contenant le document XML du modèle de processus à exécuter. Des méthodes pour le chargement et la sauvegarde du niveau Processus doivent être également implémentés dans Engine Context : ***loadProcessTrace (String fileName) et saveProcessTrace(String fileName)***

R15. La classe ***EngineContext*** doit être gérée comme un singleton. Par conséquent, une méthode de classe ***getInstance()*** permet d'obtenir un objet de la classe ***EngineContext*** qui est soit construit à la volée et stockée dans la variable de classe ***instance*** si celle-ci est nulle ou retourne la valeur de la variable de classe ***instance*** dans le cas contraire.

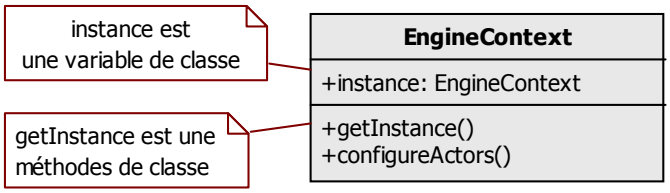


Figure 114. Variable de classe et méthode de classe de la classe globale *EngineContext*

La classe ***EngineContext*** doit avoir aussi une méthode de classe appelée *configureActors()* qui permet d'effectuer les configurations techniques de tous les acteurs constituant l'environnement de l'outil

4.7. Adaptation de la spécification OO de l'architecture de l'outil à une plateforme cible

L'architecture obtenue par application des règles de transformation prend la forme d'une spécification orientée objet. Cette forme standard lui permet d'être implémentée dans n'importe quelle plateforme[38] cible qui interprète des spécifications orientées objet (eg. Java,

[38] Une plate-forme est le matériel (hardware) et/ou l'environnement logiciel dans lequel un programme s'exécute

.Net, C++, etc.). Tous les éléments de l'architecture pourront être directement interprétés par l'une de ces plateformes cibles.

Un ensemble de classes prédéfinies sont proposées et sont réutilisables : ***ObjectChangeSupport, PropertyChangeListener, PropertyChangeEvent, ClasseInstance, ClasseType***. Ces classes peuvent être implémentées une seule fois en fonction du langage de programmation choisi.

Par contre, les classes qu'il faut adapter selon la plateforme cible choisie et notamment le système de messagerie sont ***MessageConsumer***, ***MessageProducer et MessageListener***. Ces trois éléments représentent en effet la vision conceptuelle des files d'attente (une sorte de boites aux lettres) qui permettent de synchroniser, sur une même file d'attente, le consommateur et le producteur d'un message. Par conséquent, il convient de les adapter selon la bibliothèque de classes proposée par le système de messagerie choisi (point à point ou publication/souscription). Ces adaptations entrainent d'autres adaptations de la classe ***EngineContext*** et plus particulièrement de sa méthode *configureActors()* car elle réunit les configurations techniques de tous les acteurs constituant l'environnement de l'outil.

Cette dernière étape d'adaptation permet d'obtenir une spécification objet de niveau PSM qui peut s'implémenter directement sur la plateforme choisie.

4.8. L'originalité de notre proposition

L'originalité de notre proposition se situt dans les points suivants :

- *La vision systémique* : L'un des avantages du modèle événementiel utilisé pour la spécification de l'aspect dynamique de l'outil est qu'il s'inscrit dans une perspective d'analyse du réel de type systémique. Dans cette perspective, la dynamique du système réel est abordée de manière causale, par l'analyse des causes et des conséquences des transitions d'états du système.
- *La complétude et la richesse de la spécification* : nous avons montré à travers notre proposition que la perspective « comportement » avec sa vision événementielle est indispensable à la complétude de la spécification dans une perspective d'ingénierie dirigée par le modèle. En effet, cette perspective permet de connaître les éléments-types constituant l'environnement de l'outil d'exécution et comment ces éléments agissent ou/et interagissent sur le flux d'exécution. La stratégie adoptée par la proposition combine les avantages des approches déclaratives et impératives. En effet, la perspective « traitements » est spécifiée selon une approche impérative et facilement exécutable alors que la perspective « comportement » avec sa vision événementielle permet de décrire le flux d'exécution selon une approche déclarative qui est néanmoins facilement exécutable sur des systèmes événementiels interactifs.
- *L'expression de l'exécutabilité* : la sémantique d'exécution du modèle est exprimée tout d'abord, en utilisant un modèle de représentation de la dynamique qui allie une sémantique comportementale rigoureuse et une représentation graphique adéquate. Ce modèle de comportement capture la dynamique d'un processus et exprime les interactions avec l'environnement d'exécution. Ensuite, la sémantique d'exécution du modèle est

exploitée à travers des règles de transformation qui traduisent la spécification conceptuelle du modèle en une architecture technique d'outil d'exécution.

- *La démarche dirigée par les modèles et la transformation :* l'adoption de la démarche MDA dans notre solution a permis de tirer profis des avantages de cette architecture basé sur la méta-modélisation dans le contexte de l'ingénierie des langages. En effet, la dérivation d'une architecture d'outil indépendante d'une plateforme cible (niveau PIM) à partir de la spécification conceptuelle du méta-modèle de processus (niveau CIM) permet de garantir des qualités importants dans l'outil cible notamment en termes de maintenabilité et de portabilité. Aussi, la transformation est une technique clé de l'ingénierie des modèles qui consiste à pouvoir rendre opérationnels les modèles conceptuels. L'intérêt de l'usage de la transformation des modèles dans notre solution est de les faire changer de niveau d'abstraction pour obtenir des formes destinées à un usage plus technique tel que des traductions entre langages ou la génération de code.

4.9. Conclusion

Dans ce chapitre, nous avons présenté notre approche dirigée par les modèles pour la dérivation d'une architecture d'un outil d'exécution de modèles de processus à partir de l'expression de la sémantique opérationnelle de ce modèle de processus. Cette approche s'inscrit dans une démarche globale de construction d'outils d'exécution par méta-modélisation. Notre solution se compose de deux parties principales, la première partie consiste à définir une spécification conceptuelle du méta-modèle en se basant sur un formalisme orienté objet complété par un formalisme événementiel pour prendre en compte de la sémantique d'exécution du méta-modèle. La seconde partie consiste à dériver une spécification exécutable en appliquant des règles de transformation sur la spécification conceptuelle. L'étape de transformation aboutit à une architecture de l'outil à construire qui traduit la sémantique opérationnelle du modèle. Cette architecture est sous forme orientée objet standard et par suite elle peut être exécutée dans n'importe quel outil UML de génération de code existant sur le marché.

Notre contribution a consisté à intégrer des concepts événementiels (essentiellement le concept acteur, le concept événement externe et le concept trigger) dans le langage de méta-modélisation. Cette intégration nous a permis de définir une spécification de la sémantique d'exécution relative à un méta-modèle selon une vision systémique. Notre proposition prend ainsi en compte aussi bien l'aspect statique que dynamique du langage de méta-modélisation du produit. En plus, cette spécification répond aux exigences que nous avons définies dans le chapitre 2 concernant la forme graphique, déclarative, corrélée et suffisamment riche de l'expression de sémantique pour que celle-ci allie richesse d'expression et lisibilité.

Le chapitre suivant présente un cas d'application de notre approche qui permet de la valider et de l'évaluer.

CHAPITRE 5 : APPLICATION DE LA DEMARCHE AU CAS DU MAP ET VALIDATION

5.1. Introduction

Dans ce chapitre, on se propose d'appliquer la démarche définie dans le chapitre précédent pour le cas du méta-modèle de processus Map dans l'objectif de construire un moteur d'exécution relatif à ce modèle. Le choix de ce cas d'application est justifié par le besoin récurrent de développer un moteur d'exécution relatif au modèle Map (Vélez, 2001), (Edme, 2005). En effet, ce modèle intentionnel s'adapte bien à la définition de divers sortes de processus (Souveyet and Tawbi, 1998), et en particulier des processus d'ingénierie dans les domaines des systèmes d'information, de l'ingénierie des méthodes ou encore dans l'ingénierie des outils. Le fait d'outiller ce modèle de processus, en lui construisant un moteur d'exécution, permettrait d'automatiser un processus d'ingénierie décrit sous forme d'un Map. Par ailleurs, ce moteur d'exécution pourrait s'intégrer dans un environnement de type méta-CASE ou CAME pour assister une démarche d'ingénierie de méthodes visant la construction d'applications et d'outils de manière plus systématique et plus structurée.

Nous rappelons que notre proposition suit une approche dirigée par les modèles et que son application revient à se baser sur le méta-modèle du Map, qui représente uniquement la structure des concepts et les liens entre ces concepts pour arriver à une spécification exécutable du moteur d'exécution sous forme d'un modèle de classes UML. Nous rappelons aussi que notre démarche s'inscrit dans une approche globale d'ingénierie d'outils qui consiste à développer un moteur d'exécution à partir d'un méta-modèle de processus.

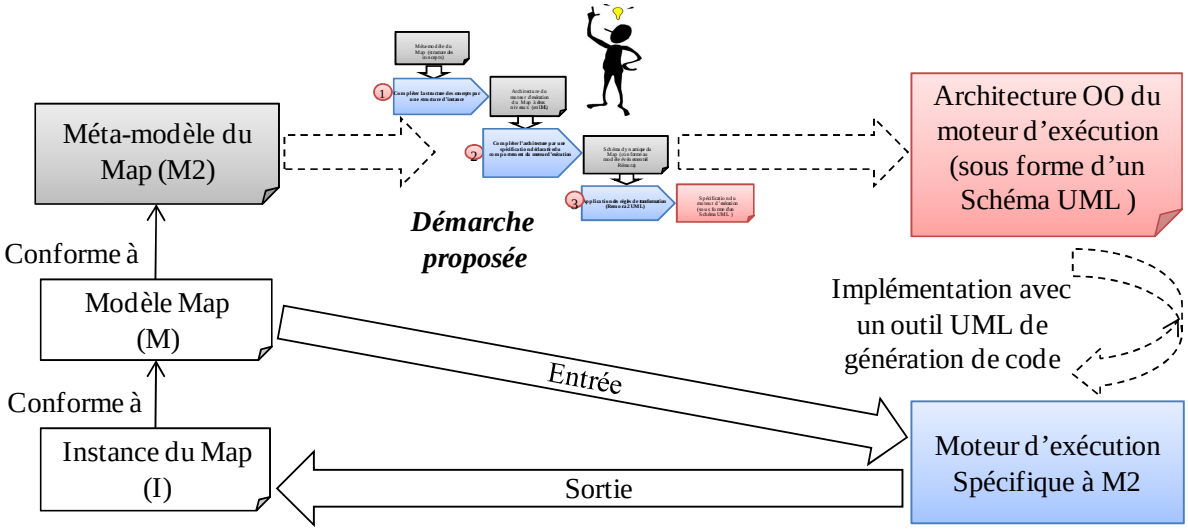


Figure 115. Une vue globale de l'application de la démarche proposée dans le cas du Map

La Figure 115 présente une vue globale de l'approche de dérivation d'un moteur d'exécution pour le Map. Cette vue permet de situer notre démarche au sein de l'approche

globale d'ingénierie et de préciser les entrées et sorties de cette démarche dans un cas d'application particulier. Le moteur d'exécution souhaité est construit sur la base d'une architecture dérivée à partir du méta-modèle du Map (M2). Ce moteur permet d'exécuter n'importe quel processus (I) conforme au modèle Map (M) qui lui même est conforme au méta-modèle Map (M2). Ceci implique que lorsqu'on change le méta-modèle, il faut mettre à jour sa sémantique opérationnelle en conséquence et redérouler le processus de transformation de l'architecture objet pour obtenir l'outil correspondant, pour cela, on l'appelle un moteur d'exécution spécifique au méta-modèle Map.

Dans notre cas d'application, la construction du moteur d'exécution du Map représente l'objectif final de l'approche d'ingénierie dans laquelle s'inscrit notre proposition. Ce moteur est le résultat d'une génération de code dans un environnement existant à partir de diagrammes UML. Quant à l'objectif de l'application de notre démarche même, c'est plutôt d'obtenir la spécification du moteur d'exécution sous une forme bien définie, tout en suivant un ensemble d'étapes en partant du méta-modèle du Map.

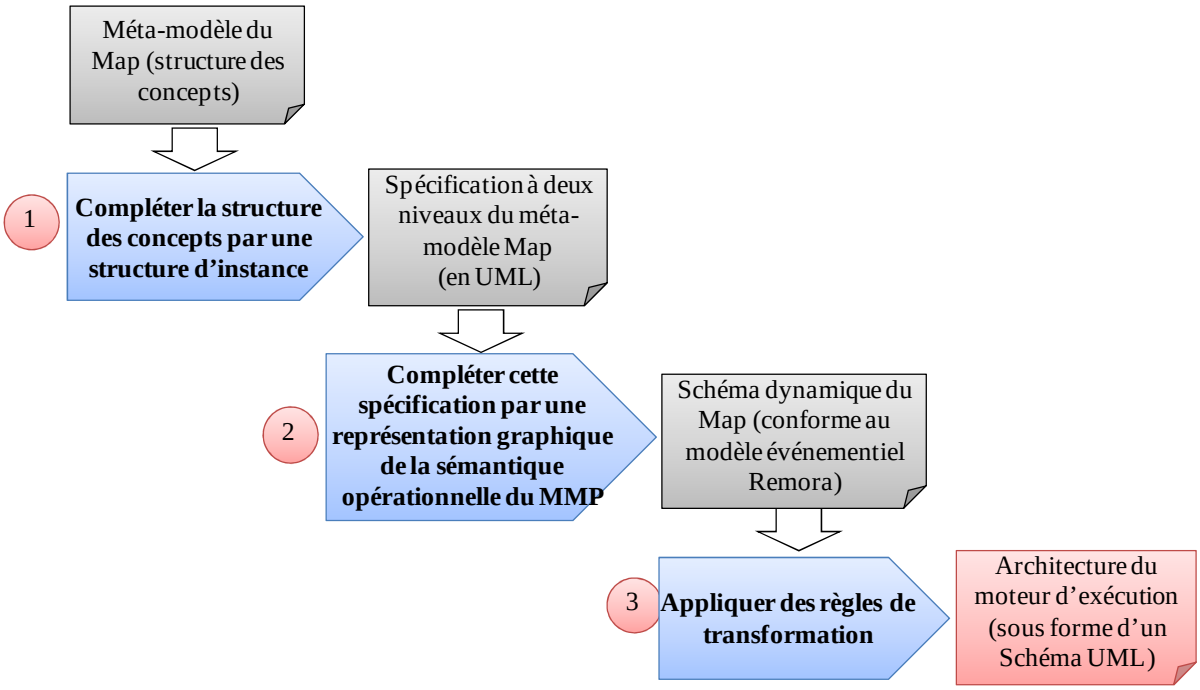


Figure 116. Les étapes de la démarche proposée appliquées dans le cas du Map

Notre démarche comporte 3 étapes :

1. Etant donné le méta-modèle du Map qui représente la structure des concepts, la première étape consiste à compléter cette structure des concepts par une structure d'instances pour enrichir la spécification statique avec une vision « traitements » à travers de nouvelles classes, des attributs particuliers et des méthodes. Ces nouveaux concepts seront utiles voir indispensables au moment de l'exécution. Le résultat obtenu à la fin de cette étape est une spécification à deux niveaux qui intègre partiellement l'expression de la sémantique d'exécution du Map.

2. La deuxième étape consiste à compléter la spécification à deux niveaux par une spécification explicite et graphique de la sémantique opérationnelle du méta-modèle

du Map. Il s'agit d'élargir le périmètre de la spécification du Map par la vision dynamique à travers le schéma événementiel qui décrit le comportement des éléments du Map pendant son exécution ainsi que l'interaction de celui-ci avec son environnement. A la fin de cette étape, nous obtenons une spécification qui couvre les trois perspectives de méta-modélisation et qui intègre aussi bien la syntaxe abstraite du méta-modèle du Map (à travers un modèle de classes) que sa sémantique d'exécution (à travers un schéma événementiel).

3. La troisième étape est consacrée à l'exploitation de la sémantique d'exécution du Map par l'application des règles de transformation, définies dans le chapitre précédent, pour dériver une architecture orientée objet du moteur d'exécution représentée par des diagrammes de classes et de séquence UML.

Les sections de ce chapitre vont correspondre aux trois étapes principales d'application de l'approche proposée dans le cas du modèle Map.

5.2. Etape 1 : Compléter le méta-modèle du Map par la structure des instances

5.2.1. La structure des concepts du méta-modèle Map

Le point de départ de l'application de l'approche est la spécification à deux niveaux qui correspond à la vision structurelle du méta-modèle de Map. La Figure 117 présente ce méta-modèle statique du Map tel que nous l'avons utilisé et spécifié par les notations du langage de méta-modélisation Ecore dans l'éditeur du Framework Eclipse (version Indigo 2012).

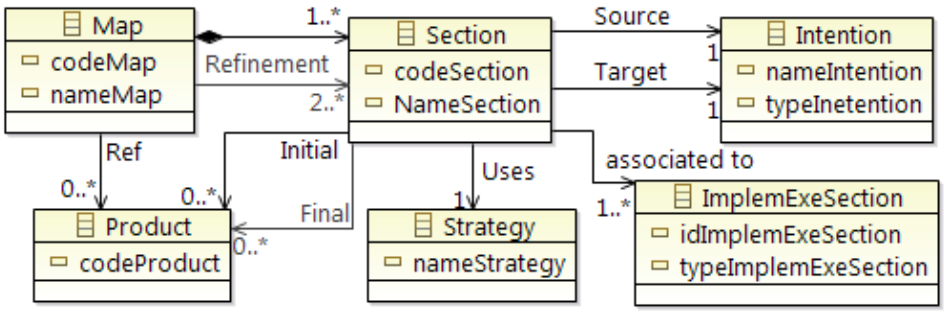


Figure 117. Le diagramme de classes du méta-modèle du Map (M2)

Comme nous l'avons déjà mentionné dans le chapitre 2 (sections 2.4.3), et comme le montre la Figure 117, un *Map* est composé d'un ensemble de sections. Chaque *Section* représente un triplet constitué d'une *Intention* source, une *Intention* cible et une *Stratégie*. La *Section* à laquelle appartient la stratégie est associée à un concept abstrait appelé *ImplemExeSection* qui renseigne sur la manière d'implémentation d'une exécution d'une section. Autrement dit, il représente un moyen d'exprimer, dans une situation donnée, la manière de satisfaire l'intention cible d'une section. On peut distinguer entre différents types d'implémentation, par exemple, ce concept peut être un service (Rim Samia Kaabi, 2007), un ensemble de directives (Rolland, 2005b) ou une application externe. La *Section* est aussi reliée à un produit et cette relation nous renseigne sur l'impact d'une section sur le produit. Par exemple, nous pouvons savoir quelle partie du produit sera nécessaire pour la réalisation

d'une *Intention* cible d'une *Section* et que sera la situation finale du produit à la fin d'exécution de cette *Section* (produit modifié, nouvelle partie du produit créée, etc.). Il est à noter que dans cette définition des concepts du méta-modèle de Map, deux modèles complémentaires sont considérées comme une boite noire au niveau du méta-modèle: le modèle permettant de décrire les parties de produit concernées par le concept de section et le modèle permettant de décrire comment est opérationnalisée la section.

5.2.2. La spécification structurelle à deux niveaux relative au méta-modèle Map

L'application de la première étape de la démarche proposée consiste à compléter la structure des concepts du méta-modèle Map par une structure d'instance pour former une spécification à deux niveaux du méta-modèle de Map: le niveau des « concepts-type » et le niveau des « concept-instance ». C'est à travers cette spécification que nous allons exprimer la sémantique d'exécution du modèle Map.

Chaque concept du méta-modèle du Map est une classe-type qui est une définition conforme au méta-méta-modèle Ecore. L'ensemble des ces classe-type constitue la structure statique du Map. Par exemple, si on considère un modèle Map avec 5 sections, la classe-type *Section* comportera 5 objets pour représenter chacune de ces sections. Pour opérationnaliser cette structure statique, nous avons besoin de définir une structure d'instance composée de classe-instances qui seront à leur tour instanciés. Chaque classe-instance du Map est une abstraction d'une collection d'objets qui sont rattachés à la classe-type et qui seront exploités pour diriger l'exécution selon le principe présenté à la Figure 118. Par exemple, à un objet « section A » qui est une instance particulière de la classe-type *Section*, on aura plusieurs objets qui sont conformes à la classe-instance « exécution de la section A ». La structure des concepts-instances est un modèle de classes qui est rattaché au modèle correspondant à la structure des concepts-types, *ces deux modèles ne sont pas du même niveau d'abstraction.*

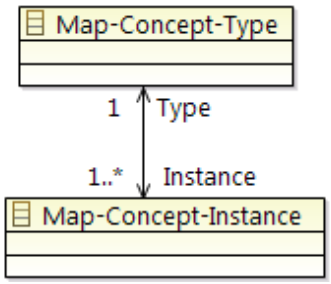


Figure 118. Le principe de dérivation de la structure des instances pour le cas du Map

Pour obtenir la structure des instances, nous avons, d'abord, attribué à chaque concept-type appartenant à la structure conceptuelle du Map une classe concept-instance correspondante conformément au principe de dérivation présenté à la Figure 118. Ensuite, en se basant sur la sémantique d'exécution du modèle Map, nous avons été amené à ajouter des méthodes et des attributs d'états ainsi que d'autres concepts que nous avons jugé indispensables pour l'exécution du modèle de processus Map. Le résultat obtenu à la fin de cette étape est une spécification structurelle à deux niveaux du méta-modèle du Map (Figure 119).

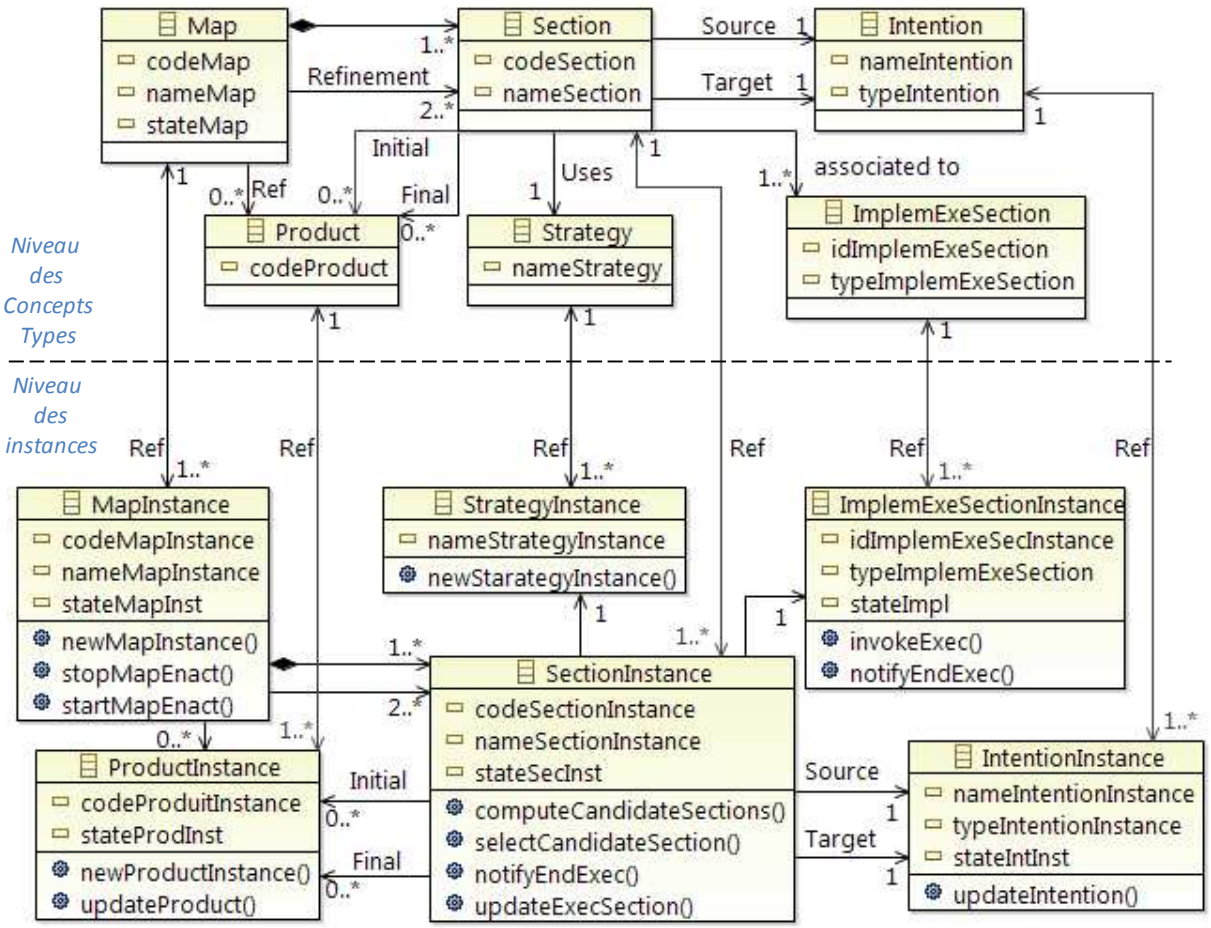


Figure 119. La spécification à deux niveaux (concept-type et instances) duméta-modèle Map

Comme le montre la Figure 119, chaque concept-instance hérite des attributs de son concept-type correspondant. Une fois les classes instances sont définies, nous avons ajouté à chacune d'elles un ensemble de méthodes et d'attributs supplémentaires correspondant principalement aux états des objets manipulés pendant l'exécution. Dans certains cas de modèles de processus, il est possible d'ajouter dans le niveau des instances de nouvelles classes qui n'ont pas de correspondant dans la structure des concepts-types ainsi que des associations les reliant aux autres classes de la spécification. Dans le cas du Map, pour des raisons de simplification, nous avons choisi de ne pas ajouter de classes. Par exemple, on aurait pu ajouter, à part la classe « *SectionInstance* », la classe « *SectionCandidate* » comme étant une instance de la classe-type « *Section* » puisqu'une section candidate représente un concept-instance qui est sémantiquement différent par rapport à une section exécutée. Il s'agit d'un concept intermédiaire qui est créée après le démarrage de l'exécution et il est recalculé à chaque étape du processus. Cependant, nous nous sommes contentés de gérer cette situation propre au Map par l'ajout d'un attribut « *stateSecInst* » dans la classe « *SectionInstance* » pour nous renseigner si l'instance de section est une section candidate, exécutée ou bien en cours d'exécution. De même, nous avons ajouté l'attribut « *stateMapInst* » pour la classe « *MapInstance* », l'attribut « *stateImpl* » pour « *ImplemExeSection* », l'attribut « *stateIntInst* » pour la classe « *IntentionInstance* » et l'attribut « *stateProdInst* » pour la classe « *ProduitInstance* ».

Un attribut d'état est une connaissance importante qui enrichit la définition des objets instances du modèle Map avec une vision dynamique. En effet, un état représente une période dans la vie d'un objet pendant laquelle ce dernier attend un événement ou accomplit une activité. Cette information renseigne sur l'évolution d'un objet lors de l'exécution du processus Map et joue ainsi un rôle important pour raisonner sur l'avancement de cette exécution. Les états d'un objet composent le domaine de valeurs de cet attribut. Nous avons défini pour chaque attribut son domaine de valeurs que nous présentons dans le Tableau 8.

Etat	Domaine de valeurs
stateMapInst	**Selected** : état d'une instance de Map lorsqu'elle est sélectionnée **Running** : état du Map en cours d'exécution **Enacted** : état du Map dont l'exécution est terminée
stateSecInst	**Created** : état d'initialisation d'une section **Candidate** : état d'une section candidate **Selected** : état d'une section en cours d'exécution **Executed** : état d'une section dont l'exécution est terminée
stateIntInst	**Created** : état d'initialisation d'une intention **Realised** : état d'une intention réalisée
EtatImpl	**Running** : état du module d'implémentation d'une section en cours d'exécution **Finishing** : état du module d'implémentation d'une section lorsqu'il termine son exécution

Tableau 8. Le domaine des valeurs des attributs d'état dans le modèle Map

L'ajout de l'ensemble de ces attributs ainsi que des opérations à chaque classe de l'architecture du moteur Map nécessite une connaissance parfaite de la sémantique d'exécution du modèle Map et leur définition s'est faite en réfléchissant au cycle de vie des concepts durant l'exécution du processus Map et en simulant les flux d'informations échangées entre les objets et les différents scénarios possibles.

Nom de la classe-instance	**Nom de l'opération**
MapInstance	*startMapEnact (codeMapInstance)*
	newMapInstance(codeMap)->MapInstance
	stopMapEnact(codeMapInstance)
SectionInstance	*computeCandidateSections(codeSectionInstance)*
	selectCandidateSection(codeSectionInstance)
	notifyEndExec()
	updateExecSection()
IntentionInstance	*updateIntention(nameIntentionInstance)*
ImplemExeSectionInstance	*invokeExec(idImpleExeSection)*
	notifyEndExec(codeSectionInstance)
ProductInstance	*newProductInstance(codeProduct)*
	updateProduct(codeSectionInstance)

Tableau 9 . La liste des opérations de la structure des instances

Le Tableau 9 illustre les opérations des classes instances. Afin de mieux comprendre le mode de fonctionnement d'un processus Map ainsi que l'interaction entre les différents objets du modèle, la spécification présentée avec le diagramme de classes peut être complétée par un diagramme de séquence comme le montre la Figure 120.

Il est à noter que la stratégie est juste une information qui n'intervient pas dans la dynamique du Map. Elle peut être représentée par un attribut dans la classe Section, mais pour respecter le méta-modèle d'origine du Map, nous avons gardé la classe (Stratégie) dans le méta-modèle, même si celle-ci ne joue aucun rôle dans la dynamique d'exécution.

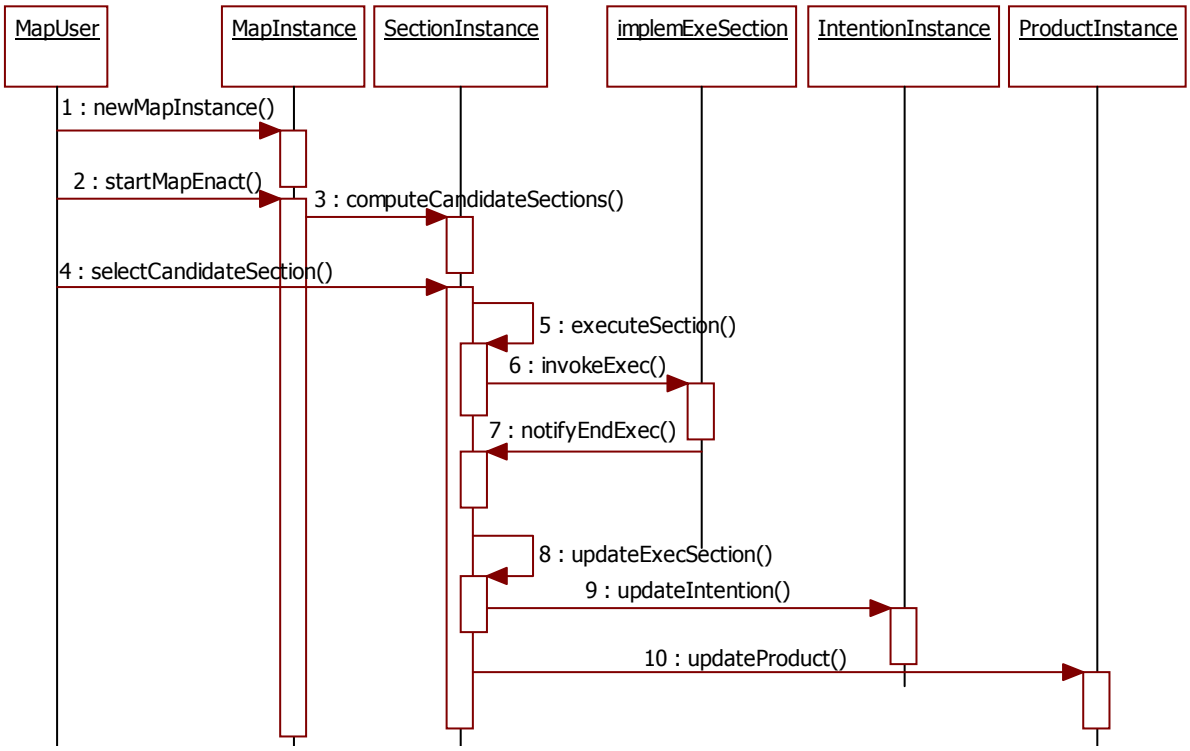


Figure 120. Le diagramme de séquence du méta-modèle Map

Comme son nom l'indique, ce diagramme permet de représenter une séquence d'opérations qui sont exécutées dans l'ordre chronologique. Cette vision procédurale (algorithmique) est insuffisante pour décrire les modèles d'applications interactives –tel que le modèle de la carte Map– qui doit réagir différemment suivant les événements externes qui agissent sur le méta-modèle Map. En effet le code qu'il est possible de générer à partir de ce diagramme de séquence (et du diagramme de classes correspondant) ne tient pas en compte du comportement interactif qu'a le méta-modèle Map avec des acteurs externes, d'où l'intérêt de notre proposition.

La spécification obtenue jusqu'à ce niveau de la démarche représente partiellement l'aspect dynamique modèle Map. Pour compléter cet aspect, nous proposons dans l'étape suivante de définir une spécification événementielle de cet aspect dynamique complétant la spécification du comportement du méta-modèle Map.

5.3. Etape 2 : Compléter la spécification à 2 niveaux par l'expression de la sémantique d'exécution

La spécification obtenue jusqu'ici résulte d'une méta-modélisation statique à l'aide d'un diagramme de classes UML. Cette spécification a été complétée d'une part avec des méta-structures nécessaires pour mettre en œuvre l'exécution, et d'autre part, avec la spécification exécutable (dans le sens orienté objet) des méthodes rattachées aux méta-classes. Ainsi, on peut conclure que *cette spécification couvre bien les perspectives « données » et « traitements » mais pas la perspective « comportementale »* (perspectives définis dans le Framework des méthodologies de SI (Olle et al., 1988)). La vision « traitements » représente une partie de la sémantique opérationnelle d'un modèle. On peut à ce niveau utiliser un langage tel que Kermeta pour spécifier en détail la sémantique opérationnelle au moyen d'expressions rattachées au corps des opérations des classes UML. Cependant, cette vision procédurale est insuffisante pour les modèles interactifs qui nécessitent la prise en compte de l'interaction avec des acteurs externes. Pour compléter l'aspect manquant qui décrit le comportement interactif du modèle Map, nous allons compléter la spécification obtenue sous forme d'une architecture à deux niveaux par une spécification orientée événement qui décrit la dynamique d'interaction de l'outil d'exécution du modèle Map. Ce complément va mettre en évidence la vision comportementale et donner lieu à une spécification qui couvre les trois perspectives de méta-modélisation.

5.3.1. La démarche suivie pour l'élaboration du schéma dynamique

Le processus d'obtention du schéma est loin d'être linéaire. En effet, il s'agit d'un processus itératif qui exige plusieurs allers retours afin de s'assurer de la complétude et de la cohérence des modèles spécifiant l'aspect statique et dynamique du méta-modèle de processus étudié. La Figure 121 présente le processus d'élaboration du schéma dynamique.

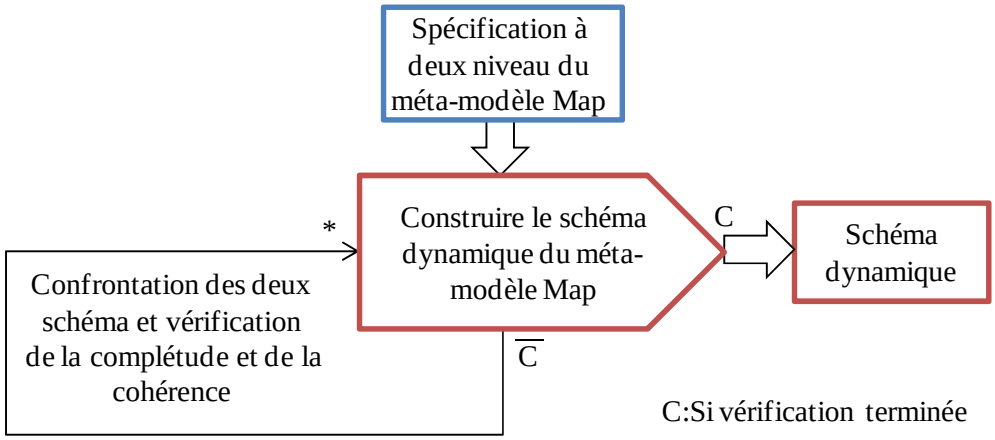


Figure 121. La démarche suivie pour l'élaboration du schéma dynamique

En partant de la spécification à deux niveaux du méta-modèle Map (qui est présenté à la Figure 119), la construction du schéma dynamique correspondant nécessite plusieurs confrontations entre les deux spécifications structurelle et comportementale. Pour cela, nous nous sommes basés, d'une part, sur les règles méthodologiques de construction d'un schéma événementiel qu'on peut trouver dans (Rolland et al., 1988). Ces règles permettent d'effectuer

des contrôles de conformité, de cohérence et de complétude et peuvent s'exécuter manuellement ou automatiquement. D'autre part, nous nous sommes basés sur d'autres règles que nous avons définies, parmi lesquelles :

1. Partant du diagramme de classes à deux niveaux relatif au méta-modèle Map (Figure 119), nous associons à chaque classe pertinente un objet portant le même nom.
2. Chaque opération d'une classe est représentée par une flèche qui pointe sur l'objet correspondant à cette classe.
3. Pour chaque objet de la structure à deux niveaux, nous définissons l'ensemble des événements et des messages qui l'accompagnent.
4. L'acteur est une entité qui intéragit avec le modèle de processus en émettant des évenements et recevant des messges, elle doit donc figurer dans le shéma dynamique.
5. Finalement, une confrontation entre le schéma dynamique et le modèle structurel est nécessaire en examinant un à un les objets et les opérations pour s'assurer de la cohérence des deux spécifications (pas d'objet et/ou opération en plus ni en moins).

5.3.2. Le schéma dynamique relatif au méta-modèle du Map

Pour obtenir le schéma dynamique, il faut suivre les étapes suivantes qui correspondent aux cinq règles précédemmnent citées :

Identification des objets : Les objets qui feront partie du schéma dynamique sont : *MapInstance, SectionInstance, ImplemExeSectionInstance, IntentionInstance* et *ProductInstance*.

Identification des événements : L'ensemble des événements du schéma dynamique est présenté au le Tableau 10:

Code événement	Description des événements
EV1	Arrivée d'un message informant la sélection d'un Map à exécuter par un acteur
EV2	Arrivée d'un message informant de début d'exécution d'une instance de Map
EV3	Lorsqu'une instance de Map devient « en cours d'exécution » (Running) le début de l'exécution
EV4	Lorsqu'une instance de section devient candidate (Candidate)
EV5	Arrivée d'un message de sélection d'une section candidate par un acteur (pour l'exécuter)
EV6	Lorsqu'une instance de section devient sélectionnée (Selected).
EV7	Lorsqu'une instance de module d'implémentation de section devient en cours d'exécution (Running)
EV8	Arrivée d'un message de notification de fin d'exécution d'une instance d'une section
EV9	Lorsqu'une instance de section devient exécutée (Executed)
EV10	Lorsque l'instance d'intention devient réalisée (Realised)
EV11	Message de notification d'un acteur externe pour mettre à jour le produit
EV12	Lorsqu'une instance de Map devient exécutée (Enacted)

Tableau 10. La liste des événements du schéma dynamique du méta-modèle Map

Identification des messages :

D'après la définition des opérations et l'étude du comportement du Map, il est possible d'identifier les messages échangés entre les objets de la spécification du méta-modèle Map et leur environnement externe. Les messages sont présentés dans le tableau Tableau 11. La liste des messages du schéma dynamique du méta-modèle Map Tableau 11:

Code message	Description des messages
M1	Demande de début d'exécution d'un Map
M1bis	Début d'exécution d'une instance de Map
M2	Liste des sections candidates
M3	Sélection d'une section candidate
M4	Notification de fin d'exécution d'une section
M5	Créer ou mettre à jour des instances de produit
M6	Notification de fin d'exécution d'un Map
M7	Demande d'exécution du module implémentant une section
M8	Notification de fin d'exécution d'une section
M9	Notification d'une nouvelle situation du produit

Tableau 11. La liste des messages du schéma dynamique du méta-modèle Map

Identification des acteurs :

Un acteur est un concept qui peut être ajouté au modèle de processus à opérationnaliser pour représenter la vision statique d'un élément de l'environnement. Il peut s'agir de l'utilisateur qui n'est pas présent dans le modèle ou un élément qui prend en charge une partie abstraite. Dans notre exemple, nous avons trois acteurs externes : *MapActor*, *ProductManager* et *SectionApplicationExecutor*.

MapActor est un utilisateur du modèle de processus Map, son rôle est de lancer l'exécution d'une carte en particulier et d'avancer dans ce processus, tout en étant guidé, en choisissant à chaque étape les sections à exécuter.

SectionApplicationExecutor est un agent de type système que le modèle Map invoque pour implémenter les sections (quand le cas l'exige). L'implémentation se fait en se basant sur un modèle de module d'implémentation des sections (nommé *guideline*) associé à cet agent et qui est relié également au modèle de produit. A la fin de l'exécution, cet agent se charge de notifier au modèle Map que le module implémentant la section en cours est exécutée. L'arrivée de ce message provoque le changement d'état de la section via l'opération *notifyEndExec()*.

ProductManager est aussi un agent qui interagit avec notre modèle de Map. Il est également relié au modèle de produit et gère ainsi le contenu du produit et de la trace d'exécution. Aussi, il connaît pour chaque intention (élément du modèle de processus) quelle est le fragment de produit qu'elle consomme et/ou qu'elle produit (éventuellement). Ainsi, cet agent peut évaluer les conditions sur les intentions et calculer les nouvelles situations du produit qui vont être associées aux intentions.

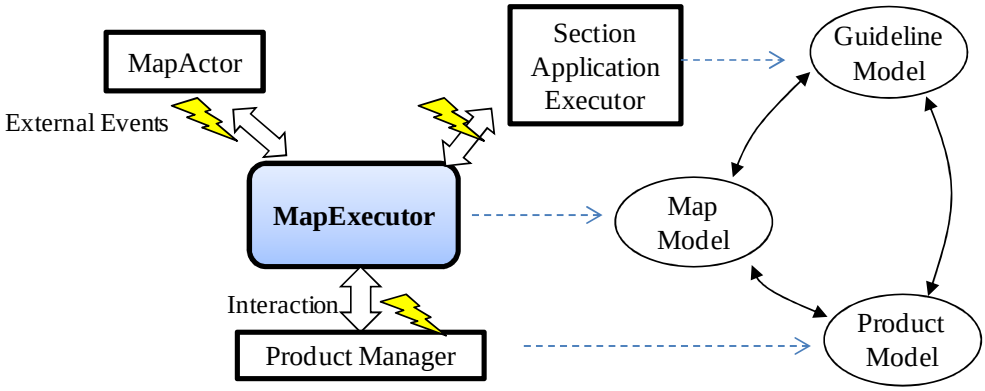


Figure 122. Les modules qui interagissent avec le moteur d'exécution

Contrairement au modèle de workflow présenté dans le chapitre précédent, les acteurs externes dans le cas du Map ne font pas partie du modèle statique de Map. Tous ces agents représentent des boites noires pour la spécification du modèle de processus (MapExecutor) qui va devoir détecter les événements externes et recevoir les messages provenant de ces agents. La Figure 122 schématise le comportement du modèle de processus Map avec les acteurs externes.

Elaboration du schéma dynamique :

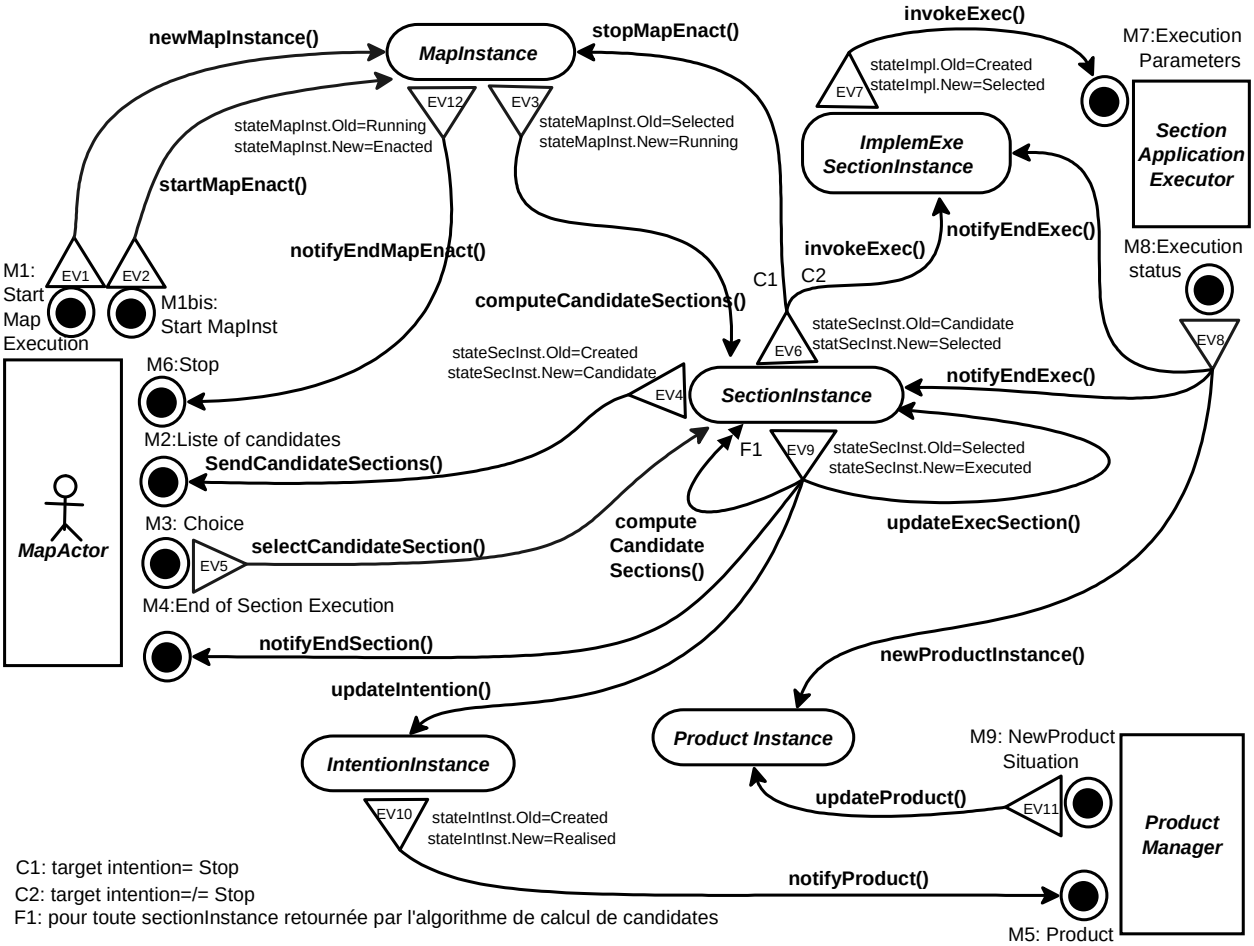


Figure 123. Le schéma dynamique du Map en utilisant le formalisme Remora étendu

Le schéma dynamique présenté à la Figure 123 vient compléter la spécification structurelle à deux niveaux du méta-modèle Map. Cette représentation orientée événement

améliore la connaissance sur le comportement d'un processus Map et facilite ainsi la compréhension de la sémantique opérationnelle des concepts du méta-modèle de Map.

Description du comportement dynamique du Map :

Le schéma dynamique présenté à la Figure 123 peut être lu de la manière suivante :

- Le démarrage du processus est lancé par l'utilisateur final ***MapActor*** qui sélectionne une carte Map (Exemple une carte de réservation). Ceci est représenté par l'événement externe **EV1** qui constate l'arrivée du message de démarrage **M1** contenant l'identifiant du ***Map*** (codeMap) qu'a choisi l'utilisateur (on suppose que dans l'interface utilisateur, il y a plusieurs cartes à exécuter). Cet événement permet également de configurer une nouvelle exécution du Map et de créer un nouvel objet de type ***MapInstance*** avec un état initial égal à la valeur *« Selected »*.

- L'arrivée d'un message **M1bis** (Start MapInst) correspond au deuxième événement externe **EV2** qui représente le fait que l'utilisateur final MapActor démarre l'exécution du Map après sa configuration. Cet événement déclenche l'opération *startMapEnact()* permettant le démarrage d'exécution d'une instance de Map.

- Lorsque l'état de l'objet ***MapInstance*** change de la valeur « *Selected* » à la valeur « *Running* » cela implique que cet objet constate l'événement **EV3** qui est responsable du déclenchement de l'algorithme de calcul des sections de candidates. Ainsi, les objets qui sont conformes à la classe « SectionInstance » et qui sont retournés par l'algorithme vont changer d'état pour devenir « Candidate ». Ce changement est à l'origine du déclenchement de l'opération *sendCandidateSections (nameSectionInstance*), par l'événement **EV4**, qui transmet le message **M2** à l'utilisateur et lui transmet la liste des sections candidates pour qu'il puisse choisir l'une des sections et avancer dans le processus d'exécution.

- Le choix d'une section candidate par l'utilisateur correspond à l'événement externe **EV5** qui est constaté sur le message **M3** et qui déclenche l'opération *selectCandidateSection(codeSectionInstance)* pour changer l'état de l'objet ***SectionInstance***, dont la référence est passée en paramètre de l'opération, de la valeur «*Candidate*» à la valeur «*Selected*». C'est à ce moment que l'événement **EV6** déclenche l'exécution du module implémentant la section ***ImplemExeSectionInstance***. Dans notre cas d'application, nous avons choisi que l'exécution des stratégies se fait par un module externe indépendant qu'on désigne par l'acteur « *SectionApplicationExecutor* » et qu'on invoque en lui transmettant les informations nécessaires à l'exécution d'une section via l'opération *invokeExec (idImplemExeSecInstance)*. A son tour, l'acteur renvoie un message (**M8**) lorsque le module d'implémentation associé à la section est terminé et l'arrivée de ce message déclenche la méthode *notifyEndExec()* ainsi que la méthode *newProductInstance()* qui permet la création d'une nouvelle instance de produit correspondant à la situation finale.

- Une fois la fin de l'exécution de la section est détectée suite à l'événement **EV8**, l'état de l'instance de section concernée change de la valeur « *Selected* » à « *Executed* ». Ce changement, qui correspond à l'événement **EV9,** est à l'origine d'un déclenchement de plusieurs opérations :
  - Mise à jour de l'état de l'objet *intentionInstance* qui devient « Realised » et qui induit à son tour la constatation de l'événement **EV10** déclenchant l'opération externe *notifyProduct().*
  - Mise à jour de l'instance de section ***SectionInstance*** puisqu'elle change d'état de « *Selected* » à « *Executed* ».
  - Lancement du processus de calcul de candidates via le déclenchement de l'opération *ComputeCandidateSections()* et qui fait dérouler l'algorithme permettant de rechercher les sections qui sont susceptibles d'être exécutées à la prochaine étape. Cette opération change l'état de toutes les occurrences de ***SectionInstance*** qui sont retournées par l'algorithme de calcul de candidates (elles sont mises à « *candidate* »).
  - Notification à l'utilisateur final de la fin d'exécution pour qu'il fasse un autre choix de section parmi les candidates à exécuter.

La représentation graphique traduit la vision systémique du méta-modèle Map au moment de son exécution. Pour des raisons de simplification et de clarté du schéma, nous n'avons pas représenté les paramètres des méthodes à la Figure 123.

La modélisation du comportement du modèle Map contribue à l'expression de son exécutabilité, mais cela reste toujours que du théorique, et donc pour prouver effectivement l'intérêt de cette solution rien n'est mieux que de la mettre en œuvre et la tester. Pour cela, une fois le schéma événementiel est valide, on peut passer à la seconde partie du processus de construction de l'outil d'exécution: l'application de règles de transformation de la spécification conceptuelle du modèle Map (statique et dynamique) vers une spécification exécutable. Dans ce qui suit, nous présentons les éléments du schéma dynamique sous forme XML. Ces éléments sont: les événements internes, les événements externes, les trigger, les messages.

5.4. Etape 3 : Transformation des spécifications vers une architecture orienté objet du moteur d'exécution

Dans cette partie, nous allons appliquer selon une approche dirigée par les modèles les règles de transformation, que nous avons définies dans le chapitre précédent, dans le cas du modèle Map. Le résultat attendu est un diagramme de classes complété par des diagrammes de séquences conformes au langage UML standard. L'ensemble de ces diagrammes forment l'architecture du moteur d'exécution du méta-modèle Map. Nous allons présenter, pour l'application de chacune des règles de transformation, un exemple illustratif (sous forme de diagrammes de classes et séquences). La totalité des diagrammes formant l'architecture orientée objet du moteur d'exécution du Map est présenté à l'**Annexe 11** (sous une forme UML graphique).

5.4.1. Exemple d'application de la règle n°1

D'après le schéma dynamique du Map (Figure 123), il y a 7 événements internes : EV3, EV4, EV6, Ev7, EV9, EV10 et EV12. L'application de la règle n°1 **(R1, R2, R3 et R4)** sur chacune des ces situations d'événements internes donne lieu à une spécification sous forme d'une structure de classes et deux diagrammes de séquence que nous présentons respectivement à la Figure 124, la Figure 125 et la Figure 126.

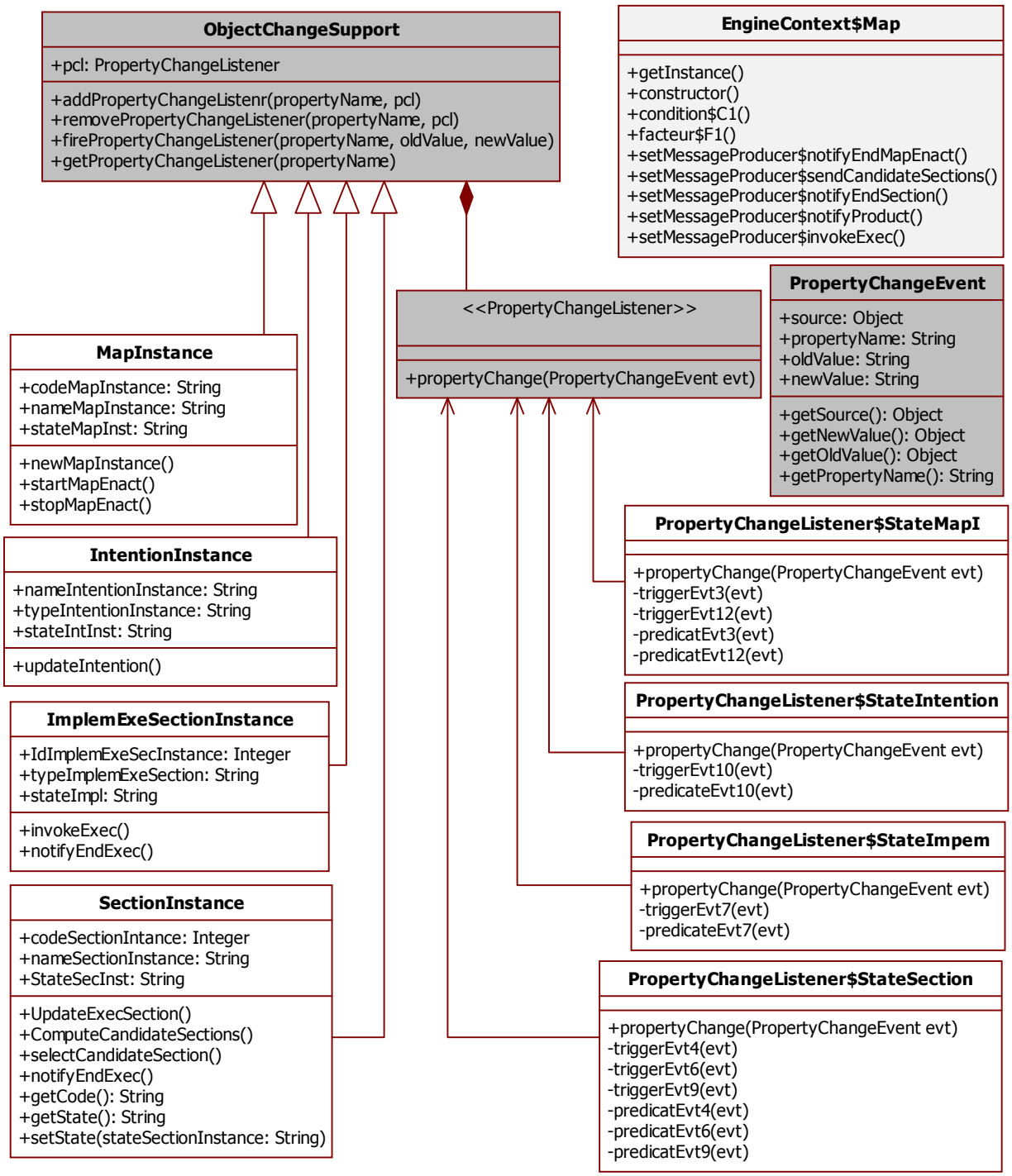


Figure 124. Le résultat de l'application de la règle de transformation n°1 au modèle Map

Le diagramme présenté à la Figure 125 correspond à un exemple de résultat obtenu par l'application de la règle n°1 pour l'événement interne EV6 du Map. Ce diagramme décrit la création d'objets ainsi que l'enchainement d'opérations nécessaires pour gérer une nouvelle occurrence d'un événement interne.

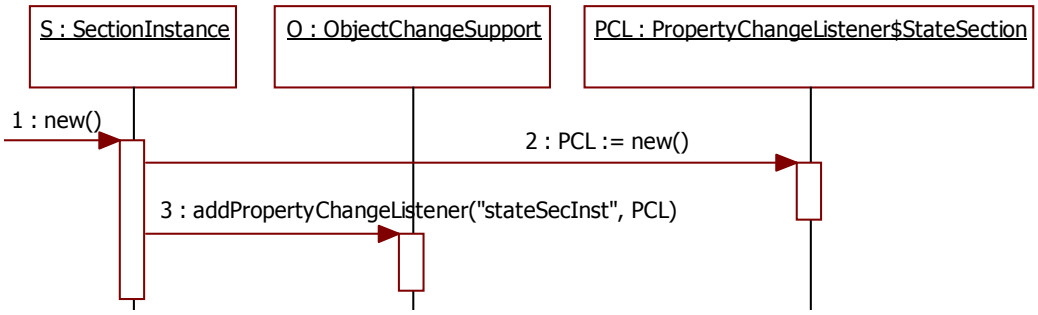


Figure 125. Le diagramme de séquence résultant de l'application de la règle n°1 : Exemple d'initialisation des objets pour l'événement interne EV6 du modèle Map

Le diagramme de séquence de la Figure 126 illustre l'application de la règle de transformation n°1 pour l'exécution de l'événement interne EV6.

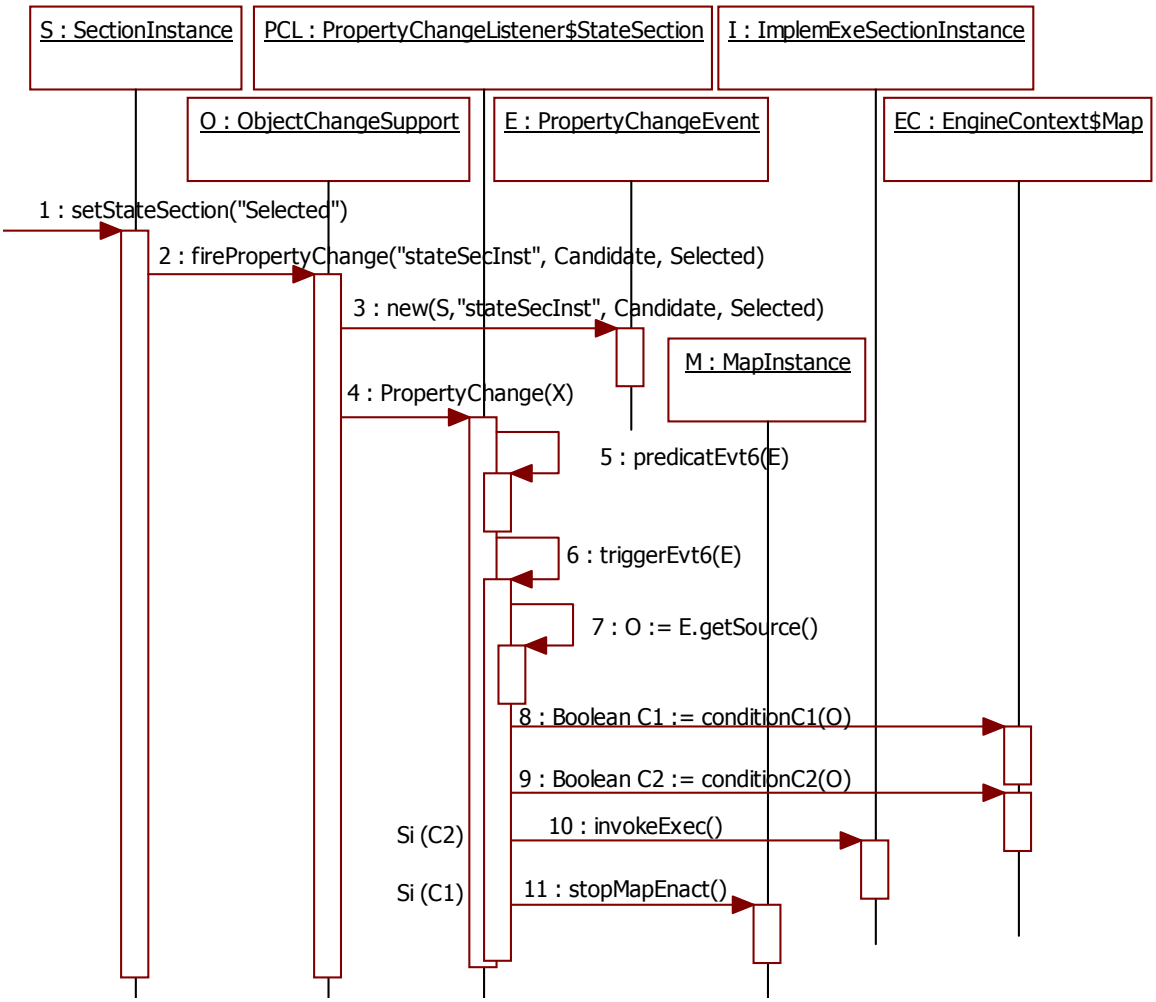


Figure 126. Le diagramme de séquence résultant de l'application de la règle n°1 : Exemple d'exécution de l'événement interne EV6 du modèle Map

5.4.2. Exemple d'application de la règle n°2

Le schéma dynamique du Map montre 5 événements externes qui sont EV1, EV2, EV5, EV8 et EV11. L'application de la règle n°2 (composée de **R5, R6, R7, et R8**) dans le cas du Map donne les classes présentées à la Figure 127.

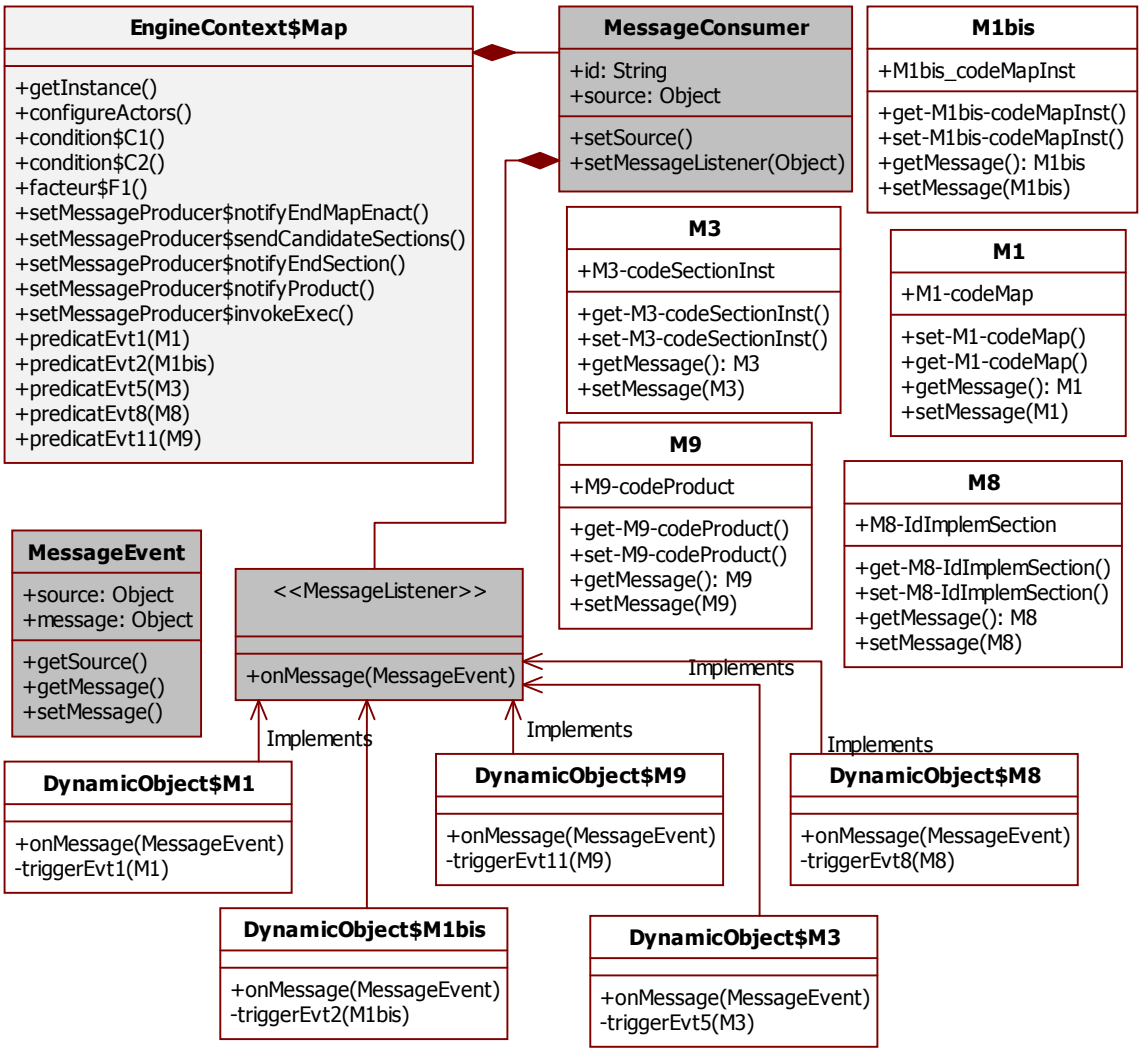


Figure 127. La structure des classes obtenue par l'application de la règle de transformation n°2 au Map

Les diagrammes de séquence générés par l'application de la règle de transformation n°2 contiennent les différentes opérations échangées dans le cadre des interactions externes avec message entrant. Nous présentons dans les figures suivantes (Figure 128 et Figure 129) un exemple de ces diagrammes pour le cas de l'événement externe EV1.

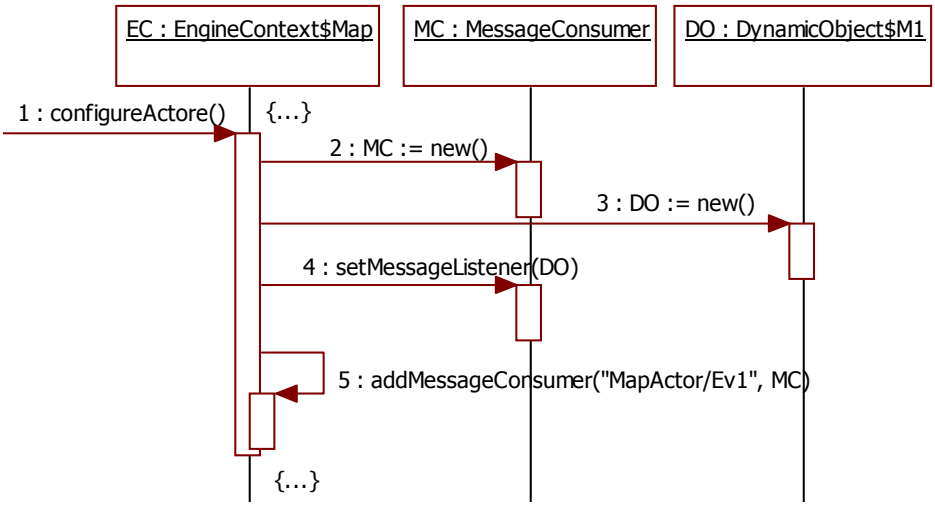


Figure 128. Un extrait du diagramme de séquence résultant de l'application de la règle n°2 : Exemple d'initialisation pour l'événement externe EV1 du Map

Le diagramme de la Figure 128 correspond à l'initialisation des objets qui seront manipulés au moment d'une interaction entrante (c-à-d à l'arrivée d'un message de la part d'un acteur externe).

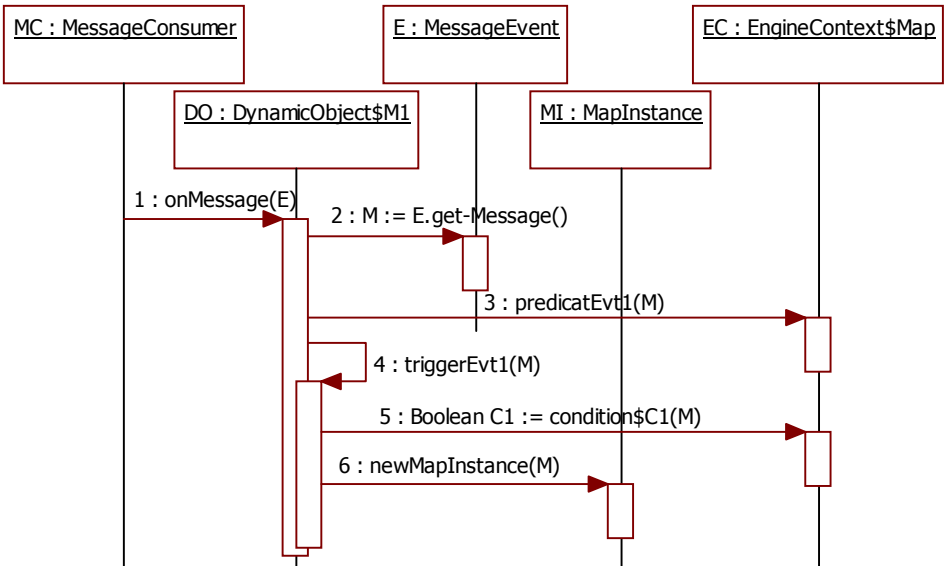


Figure 129. Le diagramme de séquence résultant de l'application de la règle n°2 : Exemple d'exécution de l'événement externe EV1 du Map

Le diagramme de la Figure 129 décrit ce qui se passe au niveau du système, en termes d'échange de méthodes entre les objets, au moment de l'exécution d'un événement externe qui est déclenché par la méthode *onMessage (MessageEvent).*

5.4.3. Exemple d'application de la règle n°3

Contrairement au cas du workflow, les acteurs externes du Map ne sont pas représentés par un concept du méta-modèle. Pour cela, nous appliquons la règle **R9.b** qui consiste à :

- Ajouter pour chaque acteur un objet de type ***MessageProducer*** comme une variable d'instance globale de ***EngineContext$Map*** avec comme nom le nom de l'acteur externe.
- Ajouter une méthode de type « «***get*** » et « s***et*** » sur cette variable d'instance.

- Initialiser la variable d'instance dans la méthode *configureActors()* de la classe ***EngineContext$Map***.

L'application de la règle n°3 (R9, R10 et R11) donne la spécification UML présentée graphiquement à la Figure 130.

Afin de détailler la manière avec laquelle les classes générées coopèrent entre elles, nous définissons les diagrammes de séquences relatifs à l'application de la règle n°3 concernant l'opération externe. A la Figure 131, nous présentons un exemple illustratif de l'application de cette règle pour l'initialisation de l'opération externe *notifyEndMapEncat()* déclenchée par **EV12**.

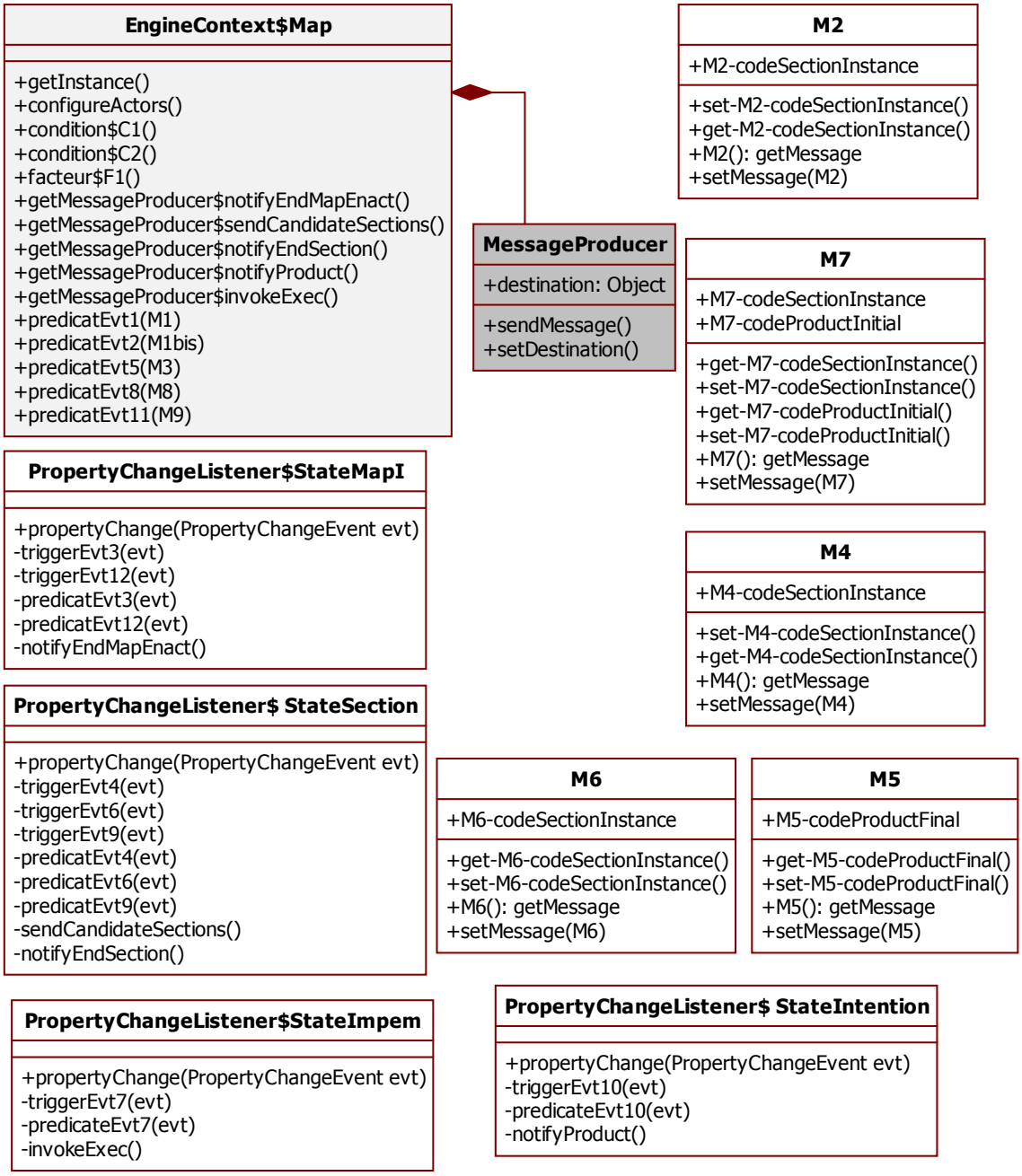


Figure 130. Les classes résultant de l'application de la règle de transformation n°3 au modèle Map

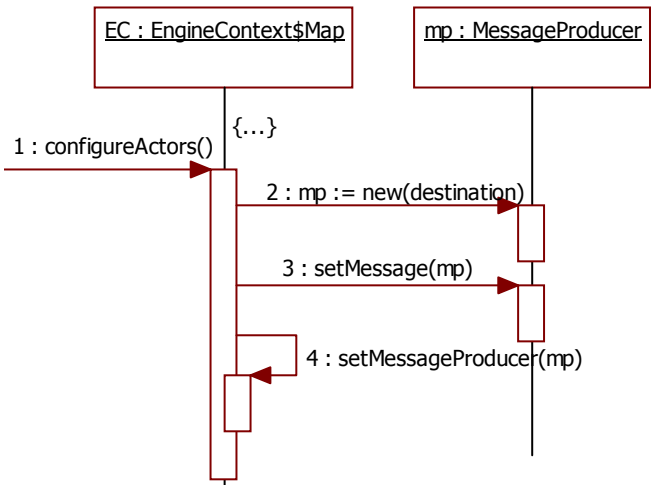


Figure 131. Un extrait du diagramme de séquence relatif à l'application de la règle n°3 pour l'initialisation d'une opération externe : Exemple EV12 du Map

Le deuxième diagramme de séquence présenté à la Figure 132 détaille l'intéraction des objets crées au moment de l'exécution d'une opération externe. Comme cette opération est déclenchée par un événement interne, ce diagramme sera intégré avec le diagramme de séquence correspondant à cet événement interne (**EV12** dans ce cas).

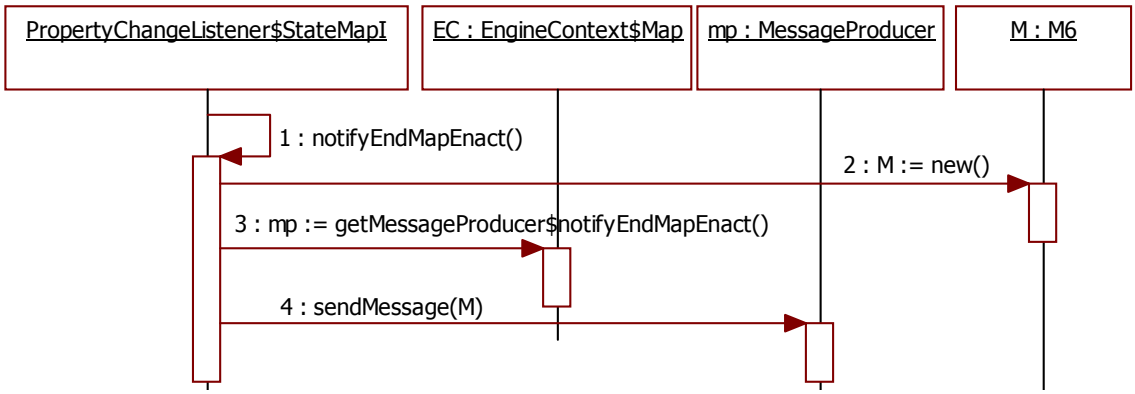


Figure 132. Un extrait du diagramme de séquence relatif à l'application de la règle n°3 pour l'exécution d'une opération externe : Exemple EV12 du Map

L'obtention de l'ensemble des diagrammes de séquence de toutes les opérations externes du modèle du Map se fait de la même manière que celle pour l'opération de notification de l'acteur ***MapActor*** par le message M6. La seule différence réside dans l'objet dynamique et le message qui change d'un cas d'une opération externe à une autre.

5.4.4. Exemple d'applications des transformations relatives à la structure à 2 niveaux et à l'acces aux données

L'application des règles de transformation relatives à la structure à deux niveaux et à l'accès aux données relatives au méta-modèle Map contribuent à la dérivation de l'architecture technique du moteur d'exécution du Map.

En effet, l'application de la règle R12 consiste à faire hériter chaque classe de niveau « type » d'une classe générique appelée ***ClassType***, et chaque classe de niveau « instance » d'une classe appelée ***ClassInstance***. Dans le cas, où un objet de la classe instance constate un événement interne, il faut que cette classe hérite de la classe ***objectChangeSuport*** qui va à son tour hériter de la classe ***ClassInstance*** (Figure 133). Cette solution permet de contourner le problème de l'héritage multiple qui n'existe pas en java par exemple. Chaque classe de la structure à deux niveaux du Map, quelque soit de niveau « type » ou de niveau « instance », doit permettre d'acceder à une collection d'objets construits à partir de cette classe. Pour cela, à chacune de ces classes nous ajoutons, conformément à la règle R13, une variable de classe privée ***collection*** permettant de regrouper les instances et les méthodes de classe : *getCollection(), addInstance(), removeInstance().*

Un exemple d'application des règles R12 et R13 pour la classe ***Map*** du niveau « type » et la classe ***MapInstance*** du niveau « instance » est présenté à la Figure 133.

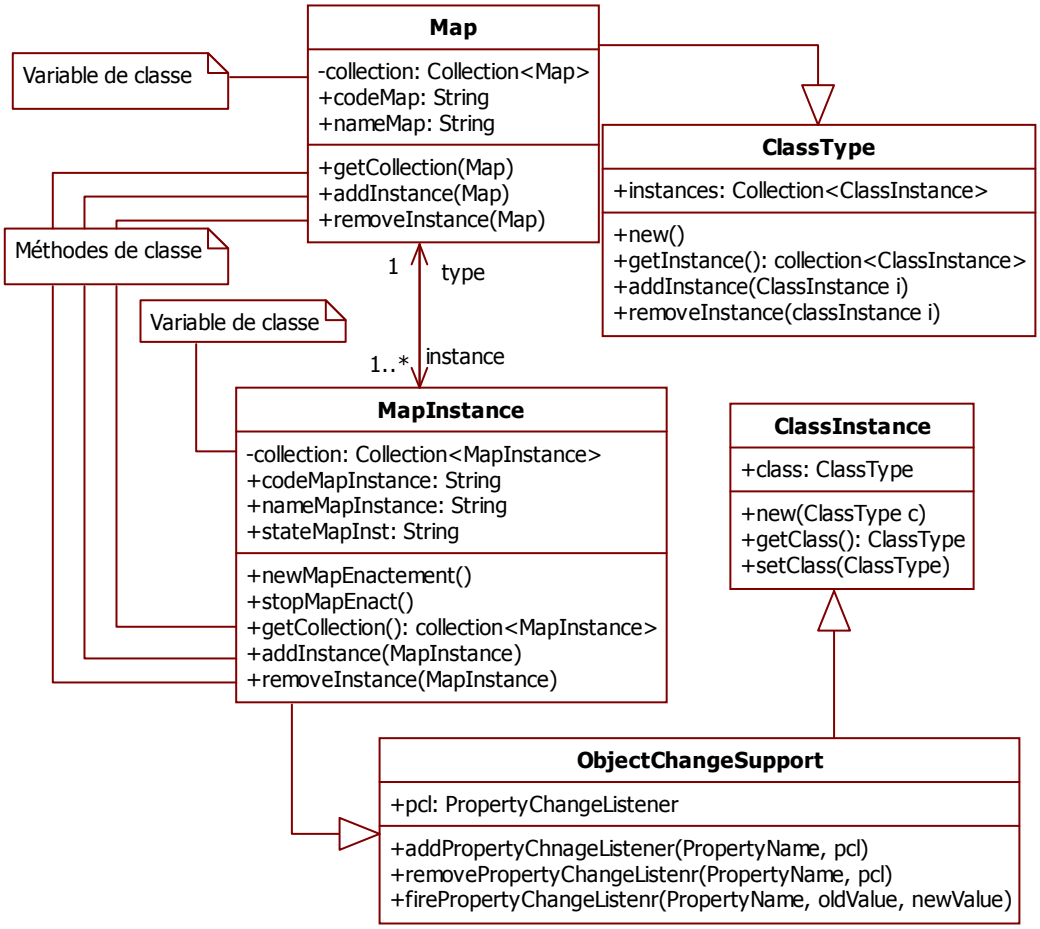


Figure 133. Exemple d'application des règles relatives à la gestion de la structure à deux niveaux

L'application de la règle R15 concerne la classe globale ***EngineContext.*** Dans le cas du Map, on a nommé cette classe ***EngineContext$Map*** (Figure 134). Elle contient la méthode de classe *getInstance()* permet d'obtenir un objet de cette classe qui est stocké dans la variable de classe *Instance*.

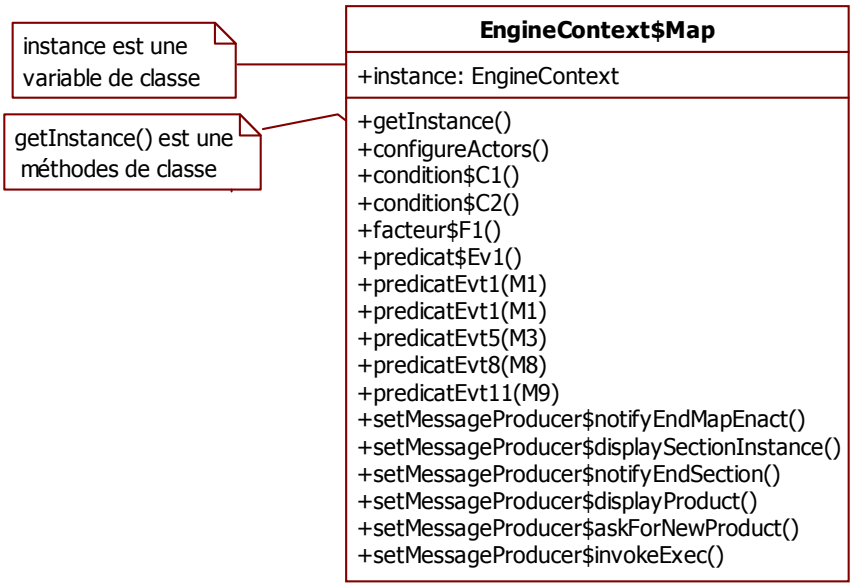


Figure 134. La Classe globale EngineContext relative au Map

La classe ***EngineContext$Map*** contient également les méthodes utilitaires tels que *condition$C1(), condition$C2* et *facteur$F1()* ainsi que d'autres méthodes que le code aura besoin lorsque le processus du map va s'exécuter.

Une vue d'ensemble des classes UML ainsi que du diagramme de séquence générés par l'étape de transformation sont présents à l'**Annexe11**. Comparés au diagramme de séquence du Map de la Figure 120, les modèles de séquences obtenus suite à l'application des règles de transformation offrent une connaissance supplémentaire concernant le comportement interactif du modèle Map avec les acteurs constituant son environnement. L'ensemble de ces diagrammes ainsi que les diagrammes de classes permettent de générer l'architecture objet du moteur d'exécution du Map permettant à celui-ci de réagir en fonction (1) des consommations de messages en provenance d'acteurs de son environnement et (2) des productions de messages à destination de ces mêmes acteurs.

5.5. Evaluation de la démarche

Dans cette section, nous nous proposons d'évaluer notre démarche en la comparant à d'autres travaux similaires. Pour cela, nous présentons un tableau comparatif qui positionne notre démarche par rapport à ces travaux ayant le même objectif que nous : construire un moteur d'exécution relatif au Map. Ces travaux sont présentés dans la première ligne du tableau selon l'ordre de leur réalisation et correspondent au prototype MapExecutor réalisé par Fernando VÉLEZ dans le cadre de sa thèse de doctorat (Moreno and Fernando, 2003), ensuite, le prototype de Marc-Henri EDME (EDME, 2005), puis notre projet exploratoire que nous avons présenté dans le chapitre 2 et qui a était publié dans (Damak Mallouli and Assar, 2013), enfin notre proposition réalisée dans le cadre de cette thèse.

Critère\Démarche		Velez	Edme	MetaEdit+	Notre proposition
Démarche	Type	Ad-hoc	Ad-hoc	Méta-CASE	IDM
	Générique/ spécifique	Spécifique	Spécifique	Générique	Générique
Langage de méta-modélisation	Formalisme statique	E/R	E/R	GOPRR	MOF (UML)
	Formalisme dynamique	-	-	Méta-Map	Evénementiel (Remora)
	Expression de la sémantique d'exécution	Code orienté objet	Requête + code+ tables relationnelles	Script Merl+ code généré	Schéma dynamique de l'architecture du moteur
	Effort cognitif pour exprimer la sémantique	Très élevé	Très élevé	Elevé	Elevé
Produit final (Moteur d'exécution)	Interactivité	Non	Non	Non	Oui
	Maintenabilité	-	-	+	++
	Portabilité	-	-	+	+

Tableau 12. Le tableau comparatif pour évaluation de la démarche proposée

Les critères de comparaison que nous avons choisis sont presque les mêmes critères que nous avons utilisés dans le chapitre 3 et qui ont été inspirés des autres études comparatives d'approches de construction d'outils permettant de récapituler les différents points de vue concernant la problématique générale de méta-modélisation et construction d'un outil d'exécution.

La démarche :

- La démarche de construction d'outils que nous proposons ici est, d'un point de vue méthodologique, plus structurée que les autres approches. En effet, les travaux de Velez et de Edme adoptent une démarche spécifique, ad-hoc et non structurée, directement orientée vers la production d'un outil logiciel d'exécution de carte Map. Une démarche ad-hoc signifie qu'elle ne suit pas des étapes structurées, bien définies et réutilisables. La démarche suivie dans le projet exploratoire avec l'outil MetaEdit+ est dirigée par l'outil méta-CASE. De manière générale, un méta-CASE offre un guidage partiel et implicite du processus d'ingénierie à travers ses interfaces utilisateurs qui se suivent tout en donnant la main au concepteur de choisir l'ordre de réalisation des différentes étapes aboutissant à la génération du produit final. La liberté de navigation entre les différentes fonctionnalités du méta-CASE peut être vue comme un atout de flexibilité de la démarche ou encore comme un manque de guidage et d'assistance. Pour cela, nous avons essayé dans notre proposition de définir explicitement les étapes d'une démarche de construction d'outil. Cette démarche est dirigée par les modèles et est composée de trois étapes dont la réalisation est détaillée par des guidelines et/ou des règles de transformation.
- Une démarche peut être générique ou spécifique. La démarche spécifique est relative uniquement au modèle Map et permet de construire un moteur d'exécution pour un

modèle en particulier (le Map) sans que cette démarche soit réutilisable ou applicable sur un autre modèle. La démarche générique est une démarche que nous pouvons appliquer sur n'importe quel modèle de processus (Map ou autre). Ce critère représente un des principaux avantages de notre proposition par rapport aux autres travaux.

Le langage de méta-modélisation

- Chaque démarche se base sur un formalisme pour la spécification des modèles qu'elle manipule. Ces formalismes varient en termes d'expressivité, de nombre et de nature des concepts et d'effort cognitif nécessaire pour exprimer la sémantique d'exécution. L'expression de la sémantique d'exécution est un critère important qui caractérise une démarche de construction d'outil d'exécution. Mieux cette sémantique est représentée, plus nous obtenons un outil de qualité. Par exemple, contrairement à l'expression implicite, l'expression explicite et déclarative de la sémantique d'exécution contribue à la dérivation d'un outil facilement maintenable. Par rapport aux travaux connexes, notre démarche propose une expression déclarative et graphique de la sémantique ce qui lui accorde un avantage supplémentaire.

Le moteur d'exécution :

- du point de vue de l'outil logiciel obtenu, les démarches de programmation directes de Velez et Edme aboutissent à des prototypes très difficiles à maintenir et à porter sans redéveloppement complet. D'ailleurs, à l'heure actuelle, ces outils ne fonctionnent plus faute de maintenance et de mise à jour. La démarche utilisant la technologie méta-CASE aboutit à une solution plus intéressante, En effet, la maintenabilité est satisfaisante si les modifications n'altèrent pas la sémantique d'exécution; sinon elle est insatisfaisante car en cas d'évolution du langage, elle nécessite une mise à jour du code généré et des changements complexes dans les scripts de génération de code. Concernant la portabilité, on peut dire qu'elle est insatisfaisante car un changement de plateforme cible induit un grand nombre de modifications dans les scripts de génération de code.
- 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+. Aussi, la structuration des étapes de notre démarche est plutôt un avantage qui joue en faveur de l'obtention d'un produit final facilement maintenable et portable car il suffit, dans le cas d'un changement du modèle de processus de re-dérouler les étapes de la démarche pour générer le nouveau moteur d'exécution.

5.6. Conclusion

Dans ce chapitre, nous avons présenté un exemple d'application de l'approche proposée afin d'en tester la faisabilité et de l'évaluer par rapport à un cas réel et par rapport à d'autres travaux existants ayant le même objectif. Nous avons suivi les étapes de la démarche

méthodologique permettant de dériver une spécification exécutable à partir d'une spécification structurelle statique d'un méta-modèle de processus Map. L'implémentation de cette spécification exécutable génère un moteur d'exécution qui satisfait un ensemble de critères grâce à la qualité de la démarche d'ingénierie dirigée par les modèles. En effet, le formalisme événementiel choisi pour spécifier l'aspect comportemental a permis de représenter la sémantique du langage de modélisation selon une vision systémique, ce qui permet de dériver un moteur de Map qui interagit avec les acteurs extérieurs de son environnement. Aussi, la démarche dirigée par les méta-modèles ainsi que la formalisation des règles de transformation permettent de rendre la génération du code plus systématique et plus organisée, et par conséquent, le moteur d'exécution obtenu sera facilement maintenable et portable.

CHAPITRE 6 : CONCLUSION ET PERSPECTIVES

6.1. Conclusion

Le travail réalisé dans cette thèse s'inscrit dans le cadre de l'ingénierie des langages de modélisation et vise la construction d'outils d'exécution relatifs à des méta-modèles de processus en se basant sur la spécification explicite de leur sémantique d'exécution. La construction des outils est un domaine de recherche intéressant aussi bien pour la communauté SI que pour celle du génie logiciel. Ce domaine est à l'intersection de multiples thématiques telles que l'ingénierie dirigée par les modèles, l'ingénierie des méthodes et la technologie méta-CASE. Dans ce cadre, nous nous sommes concentrés sur la problématique de l'exécutabilité des modèles et de l'expression de la sémantique d'exécution des langages de modélisation, qui est un champ d'investigation très large, et nous nous somme intéressés particulièrement à un aspect précis de cette problématique qui, d'après nos constats, n'a pas été pris en compte par les approches existantes. Il s'agit, d'une part, de la prise en compte de de l'aspect interactif du comportement d'un outil d'exécution de modèles de processus et de son expression explicite et graphique au niveau du méta-modèle. D'autre part, il s'agit d'exploiter cette richesse au niveau de l'expression conceptuelle du méta-modèle et de sa sémantique pour en dériver une architecture d'outil d'exécution.

6.1.1. Rappel de la problématique

Ce travail de thèse part du constat qu'aujourd'hui, les environnements méta-CASE définissent une méta-démarche basée sur la méta-modélisation, et qui apportent, par rapport à l'approche ad-hoc, des améliorations considérables à la problématique de construction d'outils. Néanmoins, nous constatons que la spécification d'un langage de modélisation de processus se restreint à décrire la structure statique (syntaxe abstraite) des concepts, et ne prend pas en compte la dynamique de ces concepts (la sémantique d'exécution) puisque la spécification de cette sémantique est complexe et doit faire appel à des concepts autres que ceux utilisés pour la méta-modélisation statique. Cela a un impact direct sur la construction de l'outil d'exécution des processus définit avec ce langage. En effet, la construction d'un tel outil se fait manuellement nécessitant une connaissance en programmation et un effort supplémentaire de la part de l'ingénieur. Par conséquent, cette manière ad-hoc d'implémentation entraine des difficultés de maintenance de l'outil d'exécution car toute évolution des méta-modèles de processus oblige l'ingénieur outil à propager manuellement des changements dans le code.

En se basant sur les limitations constatées d'après un projet exploratoire (présenté dans le *chapitre 2*) et d'après un état de l'art des travaux existants (qui a fait l'objet du *chapitre 3*), nous nous sommes fixés de répondre à la question suivante : *comment construire un outil*

d'exécution pour un modèle de processus d'ingénierie de manière à satisfaire un ensemble de critères ? Parmi ces critères :

- L'approche suivie doit offrir un support de méta-modélisation minimisant l'effort de développement, avec une spécification *explicite* de la sémantique d'exécution et une *notation graphique*.
- La spécification conceptuelle de l'outil exécutant un méta-modèle de processus doit garantir un résultat final d'une certaine *qualité*, à savoir un outil d'exécution *interactif* qui est *maintenable* et *portable*.

Notre objectif est d'aboutir à l'architecture de l'outil d'exécution et de la présenter sous *une forme UML standard* afin qu'elle puisse être exécutée dans n'importe quels environnements de de génération de code existants basés sur UML pour dériver l'outil d'exécution souhaité.

6.1.2. Bilan du travail réalisé

Afin de répondre à cette problématique, résumée ci-dessus, nous avons proposé, dans le cadre de cette thèse, une démarche dirigée par les modèles pour la construction d'une spécification exécutable de l'architecture d'un outil d'exécution.

Notre proposition se compose de trois étapes : *(1) La première étape* consiste à définir une spécification conceptuelle à deux niveaux du méta-modèle de processus sous forme d'un méta-modèle orienté objet. Cette spécification comporte une description de la structure du méta-modèle de processus à exécuter ainsi que la structure des instances qui traduit une partie de la sémantique d'exécution de ce méta-modèle. *(2) La seconde partie complète cette spécification structurelle par une description explicite, graphique et en partie déclarative déclarative de la sémantique d'exécution du méta-modèle de processus.* Cette description ajoute une vision dynamique à la spécification de ce méta-modèle de processus et traduit le comportement des modèles instances. Le choix du formalisme événementiel pour représenter ce comportement donne la possibilité de dériver un outil interactif qui communique avec son environnement. *(3) La troisième partie de notre démarche consiste à générer une architecture du moteur d'exécution par l'application de règles de transformation* sur la spécification conceptuelle composée par le méta-modèle à deux niveaux ainsi que par le schéma dynamique associé. Le résultat de cette étape prend la forme orientée objet standard ce qui lui permet d'être exécuté dans n'importe quel outil UML de génération de code existant sur le marché. Dans notre proposition (présentée au chapitre 4), nous avons complété la spécification du méta-modèle de processus, qui est conforme à un langage orienté objet, par des concepts événementiels (essentiellement les concepts acteur, événement externe et trigger). Ce complément a permis de définir une spécification de la sémantique d'exécution du mdèle de processus selon une *vision systémique* et de donner lieu, après application des règles de transformation, à une architecture logicielle de l'outil d'exécution.

L'application de notre démarche générique sur l'exemple du modèle de processus Map (dans le chapitre 5) a permis, d'une part, d'apporter des éléments concrets pour valider notre proposition, et d'autre part, de fournir une nouvelle solution au problème de construction d'un

moteur d'exécution pour ce modèle intentionnel. Ce moteur d'exécution représente sans aucun doute un outillage utile et intéressant pour assister la mise en œuvre de méthodes en ingénierie des SI ou des méthodes.

Nos contributions, pour mettre en place la démarche de construction d'un outil d'exécution d'un modèle de processus d'ingénierie tout en répondant aux exigences précédemment définies dans ce cadre, peuvent se résumer autour des points suivants :

- ***Une vision systémique :***

Dans notre proposition, nous nous sommes particulièrement intéressés à un formalisme orienté événement afin de répondre au besoin de prendre en compte la sémantique d'exécution de processus en relation avec l'environnement. L'un des avantages du modèle événementiel utilisé est qu'il s'inscrit dans une perspective d'analyse du réel de type systémique. La vision systémique qui est utilisée dans l'ISI préconise de voir le comportement d'un système selon un schéma <action-réaction> où l'action est un stimulus externe de l'environnement du système et la réaction est l'activité que le système déclenche en réponse. Dans cette perspective, la dynamique du système réel est abordée de manière causale, par l'analyse des causes et des conséquences des transitions d'états du système. Une transition d'état du système résulte de l'exécution des opérations déclenchées par un événement.

- ***Une spécification complète et riche de l'outil d'exécution :***

L'un des objectifs de notre proposition est de montrer que la perspective comportementale avec sa vision événementielle est complémentaire de la spécification d'un langage de modélisation de processus. Nous voulons mettre en évidence qu'une telle description du comportement à un niveau d'abstraction élevé permettrait de faciliter la dérivation d'un outil d'exécution et d'améliorer la qualité du résultat obtenu. Nous avons ainsi appliqué, dans un premier temps, une méta-modélisation statique à l'aide d'un diagramme de classe UML qui est complétée d'une part avec des méta-structures nécessaires pour mettre en œuvre l'exécution, et d'autre part, avec la spécification exécutable des méthodes (dans le sens orienté objet) rattachées aux méta-classes. Dans un second temps, un schéma dynamique est ajouté à cette spécification pour décrire le comportement interactif du méta-modèle de processus. Ce schéma correspond à la perspective « comportement ».

La stratégie adoptée par la proposition combine les avantages des approches déclaratives et impératives. En effet, la perspective « traitements » est spécifiée selon une approche impérative et facilement exécutable alors que la perspective « comportement » avec sa vision événementielle permet de décrire le flux d'exécution selon une approche déclarative qui est néanmoins facilement exécutable sur des systèmes événementiels interactifs.

- ***Une expression explicite de l'exécutabilité***

L'expression de l'exécutabilité est une question principale dans la problématique de cette thèse. Pour répondre à cette question, nous avons proposé une spécification de la sémantique d'exécution à travers un modèle de comportement qui capture la dynamique d'un méta-modèle de processus et qui exprime l'interaction entre les différents éléments de l'architecture de l'outil d'exécution, ainsi que l'interaction avec l'environnement de l'outil (applications

externes et utilisateurs finaux). Cette spécification se présente sous forme de l'architecture d'un outil et elle s'exprime grâce à une représentation graphique, claire et lisible, comment cet outil sera exécuté (c.à.d. sa sémantique d'exécution). Ainsi, nous pouvons dire que dans notre cas, *l'exécutabilité du méta-modèle est exprimée à travers la spécification à deux niveaux et le schéma dynamique associé.* Contrairement aux approches qui exploitent l'exécutabilité des modèles directement dans le code même en utilisant des formalismes dynamiques contemplatifs pour l'exprimer (par exemple les réseaux de Pétri, le diagramme d'états-transitions ou le diagramme de séquences, etc.), notre approche présente l'avantage de proposer une transformation de la spécification dynamique afin d'exploiter l'expression de la sémantique d'exécution et de générer une architecture OO à partir de laquelle une génération automatique de code est possible.

- ***La démarche dirigée par les modèles***

Comme nous l'avons montré dans le chapitre 4, notre solution suit une architecture MDA qui est une variante particulière de l'ingénierie dirigée par les modèles. Dans cette solution nous avons utilisé les techniques de méta-modélisation pour exprimer la sémantique d'exécution d'un méta-modèle de processus. Ensuite, nous avons réalisé des transformations des modèles afin d'exploiter cette sémantique d'exécution et de pouvoir passer d'un niveau conceptuel vers un niveau plus technique. Le principal avantage de l'approche MDA suivie est l'indépendance du niveau conceptuel vis à vis de la plate-forme technologique. De cet avantage découlent de nombreux autres. En effet, le fait de séparer la spécification conceptuelle du code, permet d'une part de résoudre des problèmes de portabilité de l'outil d'exécution et, d'autre part, elle permet de découvrir des possibilités d'erreur avant qu'elles ne soient commises. Les ingénieurs passent d'une démarche corrective à une démarche préventive, voire prédictive. Cela engendre, des gains de temps pendant le développement de l'outil d'exécution mais aussi des gains de qualité de cet outil. Aussi, grâce à la transformation de la spécification conceptuelle vers une architecture orientée objet, l'outil d'exécution est développé plus rapidement, il est plus performant et présente une meilleure maintenabilité.

6.2. Perspectives

Le présent travail soulève de nouvelles questions de recherche. Il ouvre la voie à de nouvelles perspectives que nous considérons intéressant de poursuivre leur analyse et développement. Nous soulignons certaines de ces perspectives qui vont contribuer à l'évolution de notre proposition.

- Améliorer le projet exploratoire avec MetaEdit+: réfléchir sur des moyens d'intégrer plus de structuration au niveau des scripts Merl (c.à.d au niveau de l'expression de la sémantique d'exécution).
- Evaluation plus approfondie de la démarche proposée. Nous envisageons de compléter le cas d'application du Map en générant le moteur d'exécution à partir de la spécification exécutable définie dans la thèse. Le fait de tester ce moteur d'exécution permettrait de mieux évaluer notre démarche méthodologique par rapport aux critères de maintenabilité et de portabilité.

- Explorer la démarche proposée dans d'autres domaines d'application (des langages spécifiques au domaine DSL) ce qui permettrait de raffiner cette démarche.
- Il serait intéressant de réfléchir sur une manière d'intégrer notre proposition d'expression explicite et graphique de la sémantique d'exécution dans des méta-outils existants tels que l'outil Kermeta ou l'environnement JustAdd.
- Formalisation des règles de transformation dans un langage de programmation pour former une sorte de moteur d'exécution relatif au formalisme événementiel Remora étendue. Ce moteur acceptera toute spécification conforme à ce formalisme pour en dériver une spécification orientée objet correspondante.

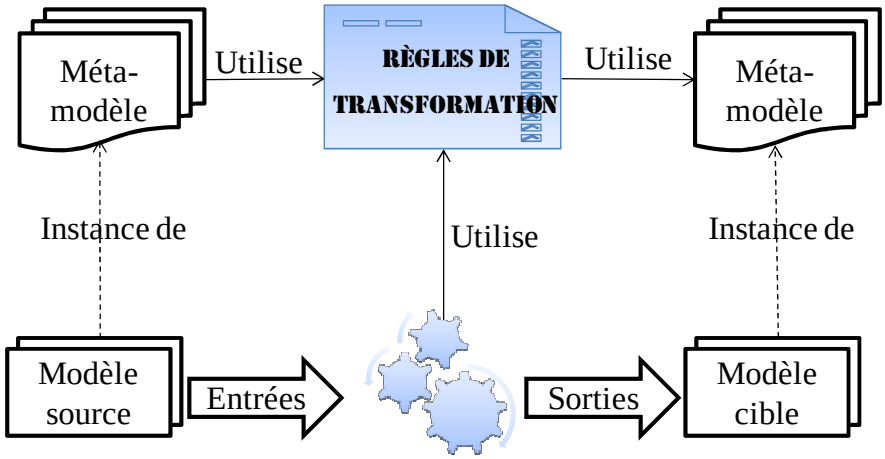


Figure 135. Le principe de transformation de modèle en IDM

La Figure 135 décrit ce principe qui considère la transformation comme un système ayant un modèle en entrée et un modèle en sortie. Ces modèles sont conformes aux méta-modèles source et cible. Les règles de transformation décrivent la correspondance entre les concepts et constructions du méta-modèle source et les concepts et constructions du méta-modèle cible et leur application s'effectue sur le modèle source pour générer le modèle cible.

Plusieurs outils et langages sont conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standard. Parmi ces outils nous pouvons citer Mia-Transformation de Mia-Software[39] et le plug-in ADT qui implémente le langage ATL du groupe ATLAS de l'INRIA-LINA (Jouault et al., 2008). Dans nos travaux futur à court termes, nous envisageons de formaliser les règles de transformation définies dans cette thèse en utilisant ce langage ATL et de les implémenter dans l'environnement Eclipse.

[39] http://www.mia-software.com/

Bibliographie

Alderson, A. (1991). Meta-case technology. In Software Development Environments and CASE Technology, A. Endres, and H. Weber, eds. (Springer Berlin Heidelberg), pp. 81–91.

Arbaoui, S., Derniame, J.-C., Oquendo, F., and Verjus, H. (2002). A Comparative Review of Process-Centered Software Engineering Environments. Ann. Softw. Eng. *14*, 311–340.

Assar, S., Achour, C.B., and Si-Said, S. (2000). Un Modèle pour la spécification des processus d'analyse des Systèmes d'Information. In INFORSID, pp. 287–301.

Assar, S., Damak Mallouli, S., and Souveyet, C. (2011). A Behavioral Perspective in Meta-modeling. In International Conference on Software Engineering and Data Technologies, (Seville, Espagne), pp. 238–243.

Assar, S., Cauvet, C., Hug, C., Front, A., and Ralyté, J. (2012). Les méthodes adaptables. In L'adaptation dans tous ses états, P. Lopisteguy, D. Rieu, and P. Roose, eds. (Cépaduès Éditions), pp. 203–225.

Barais, O. (2007). Séparation des préoccupations en phase de méta-modélisation.

Bendraou, R. (2007). UML4SPM : Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle.

Bendraou, R., Gervais, M., and Blanc, X. (2006). UML4SPM: An Executable Software Process Modeling Language Providing High-Level Abstractions. In 2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), (Hong Kong, China), pp. 297–306.

Bendraou, R., Combemale, B., Cregut, X., and Gervais, M.-P. (2007a). Definition of an Executable SPEM 2.0. In Proceedings of the 14th Asia-Pacific Software Engineering Conference, (Washington, DC, USA: IEEE Computer Society), pp. 390–397.

Bendraou, R., Sadovykh, A., Gervais, M.-P., and Blanc, X. (2007b). Software Process Modeling and Execution: The UML4SPM to WS-BPEL Approach. In Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications, (Washington, DC, USA: IEEE Computer Society), pp. 314–321.

Bendraou, R., Gervais, M.-P., Blanc, X., and Jézéquel, J.-M. (2008). Vers l'Exécutabilité des Modèles de Procédés Logiciels. In Langage Modèles et Objets LMO'08.

Bezivin, J., Jouault, F., and Touzet, D. (2005). Principles, standards and tools for model engineering. In 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings, pp. 28–29.

Bézivin, J. (1995). Technologie objet et ingéniérie des besoins : une réconciliation nécessaire. L'Objet, *1*, 21–26.

Blanc, X. (2005). MDA en action: ingénierie logicielle guidée par les modèles (Paris: Eyrolles).

Breton, E., and Bézivin, J. (2001). Towards an understanding of model executability. In Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001, (New York, NY, USA: ACM), pp. 70–80.

Brinkkemper, S., Lyytinen, K., and Welke, R. (1996). Method Engineering: Principles of method construction and tool support (Springer).

Brinkkemper, S., Saeki, M., and Harmsen, F. (2001). A Method Engineering Language for the Description of Systems Development Methods. In Advanced Information Systems Engineering, K.R. Dittrich, A. Geppert, and M.C. Norrie, eds. (Springer Berlin Heidelberg), pp. 473–476.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T.J. (2003). Eclipse Modeling Framework (Addison-Wesley Professional).

Bürger, C., Karol, S., Wende, C., and Aßmann, U. (2011). Reference Attribute Grammars for Metamodel Semantics. In Software Language Engineering, B. Malloy, S. Staab, and M. van den Brand, eds. (Springer Berlin Heidelberg), pp. 22–41.

Cauvet, C., Lingat, J.-Y., Nobecourt, P., and Rolland, C. (1986). RUBIS: Une interface d'aide à la gestion des aspects dynamiques d'une base de données relationelle. In BDA, pp. 171–188.

Cauvet, C., Rolland, C., and Lingat, J.-Y. (1989). Information System Engineering: The RUBIS System.

Cervera, M., Albert, M., Torres, V., and Pelechano, V. (2012). The MOSKitt4ME Approach: Providing Process Support in a Method Engineering Context. In Conceptual Modeling, P. Atzeni, D. Cheung, and S. Ram, eds. (Springer Berlin Heidelberg), pp. 228–241.

Clark, T., Sammut, P., Willans, J., Clark, T., Sammut, P., and Willans, J. (2008). Applied metamodelling: a foundation for language driven development. (Sheffield: Ceteva).

Rolland, C., and Cauvet, C. (1991). Modélisation conceptuelle orientée objet. In Proceedings of 7èmes Journées Bases de Données Avancées, (Lyon, France).

Combemale, B. (2008). Approche de métamodélisation pour la simulation et la vérification de modèle. Application à l'ingénierie des procédés (Institut National Polytechnique, Université de Toulouse).

Combemale, B., Crégut, X., Michel, P., and Pantel, M. (2008). SéMo'07 : premier atelier sur la Sémantique des Modèles. L'Objet *13*.

Combemale et al., S.R. (2006). Towards Rigorous Metamodeling. 5–14.

Coust, P. (1990). Handbook of Theoretical Computer Science (Vol. B). J. van Leeuwen, ed. (Cambridge, MA, USA: MIT Press), pp. 841–993.

Crégut, X., Combemale, B., Pantel, M., Faudoux, R., and Pavei, J. (2010). Generative technologies for model animation in the topcased platform. In Proceedings of the 6th European Conference on Modelling Foundations and Applications, (Berlin, Heidelberg: Springer-Verlag), pp. 90–103.

CREWS (1999). Cooperative Requirements Engineering With Scenarios.

Cronholm, S., and Goldkuhl, G. (2003). Strategies for Information Systems Evaluation – Six Generic Types. In Electronic Journal of Information Systems Evaluation, Volume 6, Issue 2, pp 65 – 74.

Damak Mallouli, S., and Assar, S. (2013). Enacting a Requirement Engineering Process with Meta-Tools: an Exploratory Project. ICCGI 2013, pp. 208–213.

Damak Mallouli, S., Assar, S., and Souveyet, C. (2011). Pour une perspective comportementale dans les méta-modèles de processus. In Actes de la XXIX édition d'INFORSID, pp. 351–366.

Ebert, J., Süttenbach, R., and Uhe, I. (1997). Meta-CASE in Practice: a Case for KOGGE. In IN, (Springer), pp. 203–216.

Edme, M.H. (2005). Proposition pour la modélisation intentionnelle et le guidage de l'usage des systèmes d'information.

Ehrig, K., Ermel, C., Hänsgen, S., and Taentzer, G. (2005). Generation of visual editors as eclipse plug-ins. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, (New York, NY, USA: ACM), pp. 134–143.

Erdweg, S., Storm, T. van der, Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., et al. (2013). The State of the Art in Language Workbenches. In Software Language Engineering, M. Erwig, R.F. Paige, and E.V. Wyk, eds. (Springer International Publishing), pp. 197–217.

Eugster, P.T., Felber, P.A., Guerraoui, R., and Kermarrec, A.-M. (2003). The Many Faces of Publish/Subscribe. ACM Comput. Surv. *35*, 114–131.

Farail, P. (2012). Toolkit in OPen-source for Critical Applications & SystEms Development.

Farail, P., Gaufillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Cr{é}gut, X., and Pantel, M. (2006). The TOPCASED project: a toolkit in open source for critical aeronautic systems design. Ing. Automob. *781*, 54–59.

Favre, J.-M. (2006). L'ingénierie dirigée par les modèles : au-delà du MDA (Hermes Science Publications).

Favre, J.-M., Gaevi, D., Lämmel, R., and Winter, A. (2009). Guest Editors' Introduction to the Special Section on Software Language Engineering. IEEE Trans. Softw. Eng. 737–741.

Fidalgo, R.N., Alves, E., España, S., Castro, J., and Pastor, O. (2013). Metamodeling the Enhanced Entity-Relationship Model. J. Inf. Data Manag. *4*, 406.

Fuggetta, A. (1993). A classification of CASE technology. Computer *26*, 25–38.

Fuggetta, A. (1996). Functionality and architecture of PSEEs. Inf. Softw. Technol. *38*, 289–293.

Garcia, M., and Shidqie, A.J. (2007). OCL Compiler for EMF.

Giraudin, J.-P. (2007). Complexité des systèmes d'information et de leur ingénierie.

Gupta, D., and Prakash, N. (2001). Engineering Methods from Method Requirements Specifications. Requir. Eng. *6*, 135–160.

Harel, D., and Rumpe, B. (2004). Meaningful modeling: what's the semantics of "semantics"? Computer *37*, 64–72.

Harmsen, F., and Brinkkemper, S. (1995). Design and implementation of a method base management system for a situational CASE environment. In Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific, pp. 430–438.

Harmsen, F., and Saeki, M. (1996). Comparison of four Method Engineering languages. In Method Engineering, S. Brinkkemper, K. Lyytinen, and R.J. Welke, eds. (Springer US), pp. 209–231.

Harmsen, F., Brinkkemper, S., and Oei, J.L.H. (1994). Situational method engineering for informational system project approaches. In Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle, (New York, NY, USA: Elsevier Science Inc.), pp. 169–194.

Harrous, D. (2011). Étude d'une approche générique d'ingénierie des modèles.

Hedin, G. (2011). An Introductory Tutorial on JastAdd Attribute Grammars.

Henderson-sellers, B., and Ralyté, J. Situational Method Engineering: State-of-the-Art Review.

Heym, M., and Österle, H. (1993). Computer-aided methodology engineering. Inf. Softw. Technol. *35*, 345–354.

Hollingsworth, D. (1996). Workflow Management Coalition - The Workflow Reference Model.

Hug, C., Deneckere, R., and Salinesi, C. (2012). Map-TBS: Map process enactment traces and analysis. (IEEE), pp. 1–6.

Isazadeh, H., and Lamb, D.A. (1997b). CASE Environments and MetaCASE Tools.

Jarke, M., Jeusfeld, M.A., Nissen, H.W., Quix, C., and Staudt, M. (2010). Metamodelling with Datalog and Classes: ConceptBase at the Age of 21. In Object Databases, M.C. Norrie, and M. Grossniklaus, eds. (Springer Berlin Heidelberg), pp. 95–112.

Jeusfeld, M.., Jarke, M., and Mylopoulos (2009). Metamodeling for Method Engineering. Cambridge (USA): The MIT Press. 424 p. ISBN.

Jézéquel, J.-M., Barais, O., and Fleurey, F. (2011). Model Driven Language Engineering with Kermeta. In Generative and Transformational Techniques in Software Engineering III, J.M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, eds. (Springer Berlin Heidelberg), pp. 201–221.

Jouault, F., Bézivin, J., and Kurtev, I. (2006). TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In Proceedings of the 5th International Conference on Generative Programming and Component Engineering, (New York, NY, USA: ACM), pp. 249–254.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. Sci Comput Program *72*, 31–39.

Kaipala, J. (1997). Augmenting CASE tools with hypertext: Desired functionality and implementation issues. In Advanced Information Systems Engineering, A. Olivé, and J.A. Pastor, eds. (Springer Berlin Heidelberg), pp. 217–230.

Kelly, S. (1997). Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+. University of Jyvaskyla.

Kelly, S. (2004). Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development,.

Kelly, S., and Smolander, K. (1996). Evolution and issues in metaCASE. Inf. Softw. Technol. *38*, 261–266.

Kelly, S., and Tolvanen, J.-P. (2008). Domain-Specific Modeling: Enabling Full Code Generation (John Wiley & Sons).

Kelly, S., Lyytinen, K., and Rossi, M. (1996). MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In Advanced Information Systems Engineering (CAiSE), P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, eds. (Berlin/Heidelberg: Springer), pp. 1–21.

Kelly, S., Lyytinen, K., and Rossi, M. (2013). MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In Seminal Contributions to Information Systems Engineering, J.A.B. Jr., J. Krogstie, O. Pastor, B. Pernici, C. Rolland, and A. Sølvberg, eds. (Springer), pp. 109–129.

Kern, H., Hummel, A., and Kühne, S. (2011). Towards a Comparative Analysis of Meta-metamodels. In Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11, (New York, NY, USA: ACM), pp. 7–12.

Kleppe, A. (2008). Software Language Engineering: Creating Domain-Specific Languages Using Metamodels (Addison-Wesley Professional).

Kleppe, A. (2009). The Field of Software Language Engineering. In Software Language Engineering, D. Gašević, R. Lämmel, and E.V. Wyk, eds. (Springer Berlin Heidelberg), pp. 1–7.

Knuth, D.E. (1968). Semantics of context-free languages. Math. Syst. Theory *2*, 127–145.

Koskinen, M., and Marttiin, P. (1998). Developing a customizable process modelling environment: Lessons learnt and future prospects. In Software Process Technology, V. Gruhn, ed. (Berlin/Heidelberg: Springer), pp. 13–27.

Koudri, A., Champeau, J., and Aulagnier, D. (2007). Une sémantique opérationnelle pour une meilleure méta-modélisation. Atelier Sur Sémant. Modèles SéMo07 Toulouse 29–30.

Lingat, J.-Y. (1988). Rubis: un système pour la spécification et le prototypage d'applications bases de données (Grenoble, France: ANRT (réf.18656)).

Loucopoulos, P., and Zicari, R. (1992). Conceptual Modeling, Databases, and Case: An Integrated View of Information Systems Development (New York, NY, USA: John Wiley & Sons, Inc.).

Marttiin, P. (1998). Customizable process modeling support and tools for design environments (Department of Computer Science and Information Systems. University of Jyv¨askyl¨).

Marttiin, P., Rossi, M., Tahvanainen, V.-P., and Lyytinen, K. (1993). A comparative review of CASE shells: A preliminary framework and research outcomes. Inf. Manage. *25*, 11–31.

Marttiin, P., Harmsen, F., and Rossi, M. (1996). A functional framework for evaluating method engineering environments: the case of Maestro II/ Decamerone and MetaEdit+. In Proceedings of the IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering, (London, UK), pp. 63–86.

Mellor, S.J., and Balcer, M.J. (2002). Executable Uml: A Foundation for Model-Driven Architecture (Addison-Wesley Professional).

MetaCase (2004). La technologie metacase.

Moreno, V., and Fernando, J. (2003). Proposition d'un environnement logiciel centré processus pour l'ingénierie des systèmes d'information (Paris 1).

Mosses, P.D. (2001). The Varieties of Programming Language Semantics And Their Uses.

Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005). Weaving Executability into Object-Oriented Meta-languages. In Model Driven Engineering Languages and Systems, L. Briand, and C. Williams, eds. (Springer Berlin Heidelberg), pp. 264–278.

Mylopoulos, J. (1990). Conceptual Modelling and Telos.

Niknafs, A., and Ramsin, R. (2008). Computer-Aided Method Engineering: An Analysis of Existing Environments. In Advanced Information Systems Engineering, Z. Bellahsène, and M. Léonard, eds. (Springer Berlin Heidelberg), pp. 525–540.

Nunamaker, J.F., J., and Chen, M. (1990). Systems development in information systems research. In , Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, 1990, pp. 631–640 vol.3.

Nurcan, S., and Edme, M.-H. (2005). Intention-driven modeling for flexible workflow applications. Softw. Process Improv. Pract. *10*, 363–377.

Object Management Group (2011). Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0.

Olle, T.W., Hagelstein, J., MacDonald, I.G., Rolland, C., Sol, H.G., Van Assche, F., and Verrijn-Stuart, A. (1988). Information Systems Methodologies: a framework for understanding. Wokingham Al.

OMG-java Profile (2004). Metamodel and UML Profile for Java and EJB Specification.

Paakki, J. (1995). Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput Surv *27*, 196–255.

Paige, R.F., Kolovos, D.S., and Polack, F.A.C. (2006). An Action Semantics for Mof 2.0. (New York, NY, USA: ACM), pp. 1304–1305.

Plihon, V., Ralyte, J., Benjamen, A., Maiden, N.A.., Sutcliffe, A., Dubois, E., and Heymans, P. (1998). A reuse-Oriented Approach for the Construction of Scenario Bases Methods. In Proceedings of International Conference on Software Process, (États-Unis), pp. 1–16.

Prakash, N. (1999). On method statics and dynamics. Inf. Syst. *24*, 613–637.

Ralyté, J., and Roll, C. (2001). An Assembly Process Model for Method Engineering. In Eds.) Advanced Information Systems Engineering), LNCS2068, (Springer-Verlag), pp. 267–283.

Rim Samia Kaabi, C.S. (2007). Capturing Intentional Services with Business Process Maps. 309–318.

Rolland, C. (2005a). L'ingénierie des méthodes : une visite guidée Article invité.

Roland, C. (2005b). Modelisation of the O* process with MAP.

Rolland, C. (2007). Capturing System Intentionality with Maps. In Conceptual Modelling in Information Systems Engineering, J. Krogstie, A.L. Opdahl, and S. Brinkkemper, eds. (Springer Berlin Heidelberg), pp. 141–158.

Rolland, C., and Prakash, N. (2000). Bridging the Gap Between Organisational Needs and ERP Functionality. Requir. Eng. *5*, 180–193.

Rolland, C., Foucaut, O., and Benci, G. (1988). Conception des systèmes d'information : La méthode REMORA (Eyrolles).

Rolland, C., Souveyet, C., and Achour, C.B. (1998). Guiding Goal Modeling Using Scenarios. IEEE Trans. Softw. Eng. *24*, 1055–1071.

Rolland, C., Prakash, N., and Benjamen, A. (1999). A multi-model view of process modelling. Requir. Eng. *4*, 169–187.

Roques, P. (2009). UML 2 par la pratique: études de cas et exercices corrigés (Editions Eyrolles).

Saeki, M. (2003). CAME : The first step to automated method engineering. In OPSLA 2003: Workshop on Process Engineering for Object-Oriented and Component-Based Development, pp. 7–18.

Saeki, M., and Wenyin, K. (1994). Specifying software specification & design methods. In Advanced Information Systems Engineering, G. Wijers, S. Brinkkemper, and T. Wasserman, eds. (Springer Berlin Heidelberg), pp. 353–366.

SAI PECK, L. (1994). Formalisation et aide outillée à la modélisation conceptuelle. Université de Paris I.

Si-Said, S., and Rolland, C. (1998). Formalising guidance for the crews goal-scenario approach to requirements engineering.

Si-Said, S., Rolland, C., and Grosz, G. (1996). MENTOR: A Computer Aided Requirements Engineering Environment. In Advanced Information Systems Engineering, pp. 22–43.

Smolander, K. (1992). OPRR: a model for modelling systems development methods. In Next Generation CASE Tools, (K. Lyytinen, V.-P. Tahvanainen), pp. 224–239.

Smolander, K., Lyytinen, K., Tahvanainen, V.-P., and Marttiin, P. (1991). MetaEdit— A flexible graphical environment for methodology modelling. In Advanced Information Systems Engineering, R. Andersen, J.A.B. Jr, and A. Sølvberg, eds. (Springer Berlin Heidelberg), pp. 168–193.

Soley, R. (2000). Model driven architecture.

Souag, A. (2010). Etude comparative des formalismes de métamodélisation.

Souveyet, C. (1991). Validation des specifications conceptuelles d'un systeme d'information, thèse de doctora de Paris 6.

Souveyet, C., and Tawbi, M. (1998). Process centred approach for developing tool support of situated methods. In Database and Expert Systems Applications, G. Quirchmayr, E. Schweighofer, and T.J.M. Bench-Capon, eds. (Springer Berlin Heidelberg), pp. 206–215.

SPEM (2008). Software & Systems Process Engineering Meta-Model Specification, OMG document formal.

Sprinkle, J., Rumpe, B., Vangheluwe, H., and Karsai, G. (2010). 3 Metamodelling. In Model-Based Engineering of Embedded Real-Time Systems, H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz, eds. (Springer Berlin Heidelberg), pp. 57–76.

Stoy, J.E. (1977). Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory (Cambridge, MA, USA: MIT Press).

Sunyé, G. (1999). mise en oeuvre de patterns de conception: un outil. Université Paris 6.

Tawbi, M. (2001). CREWS-L'Ecritoire: Une approche guidant l'ingénierie des besoins. In Proceedings of Informatique des Organisations et Systèmes d'Information et de Décision, (Suisse), p. 1.

VAN, D.E.W., and al. (2007). An evaluation of computerized tools for method construction. *107*, 59–63.

Vélez, F. (2001). MapExecutor: A Dynamic Enactment Support to Specify and Execute Methods with Maps.

Wijers, G. (1991). Modelling support in information systems development. University of Nijmegen.

Wohlin, C., Höst, M., and Henningsson, K. (2003). Empirical Research Methods in Software Engineering. In Empirical Methods and Studies in Software Engineering, R. Conradi, and A.I. Wang, eds. (Springer Berlin Heidelberg), pp. 7–23.

Zaidi, H. (2010). Etude Comparative d'outils de méta-modélisation.

Zamli, K.Z., and Lee, P.A. (2001). Taxonomy of process modeling languages. In Computer Systems and Applications, ACS/IEEE International Conference On. 2001, pp. 435–437.

Zoé, D., Cyril, F., Franck, F., Vincent, M., and Didier, V. (2009). Kermeta language.

Liste de mes publications nationales et internationales:

- *« Process behavior meta-modeling for the derivation of execution engines"*, Sana Damak Mallouli, Saïd Assar and Carine Souveyet, in The IEEE eighth International Conference on Research Challenges in Information Science (RCIS), Marrakesh, Morroco, 2014.
- "*Proposition d'une démarche de type IDM pour la construction d'outils d'exécution de processus"*, Sana Damak Mallouli, Saïd Assar and Carine Souveyet aux actes du 31ème Congrès INFORSID (INFormatique des ORganisations et Systèmes d'Information et de Décision), Lyon, France, 2014.
- "*Enacting a Requirement Engineering Process with Meta-Tools: an Exploratory Project*", Sana Damak Mallouli et Saïd Assar, in the 8th Int. Multi-Conference on Computing in the Global Inf. Technology (ICCGI) Nice, France, 2013.
- "*A Model-driven Approach to Process Enactment*". Sana Damak Mallouli, Saïd Assar et Carine Souveyet, In the 7th Int. Conf. on Software and Data Technologies (ICSOFT), Roma, Italy, 2012 .
- "*A Behavioral Perspective in Meta-modeling*". Saïd Assar, Sana Damak Mallouli et Carine Souveyet, in the proceedings of the 6th International Conference on Software and Data Technologies (ICSOFT), Volume 2, Seville, Spain, 18-21 July, 2011.
- *"Pour une perspective comportementale dans les méta-modèles de processus"*. Sana Damak Mallouli, Saïd Assar, Carine Souveyet, aux actes du XXIXème Congrès INFORSID (INFormatique des ORganisations et Systèmes d'Information et de Décision), Lille, France, mai 2011.
- *« Comment améliorer l'exécutabilité des modèles en se basant sur l'ISDM »*. Sana Damak Mallouli, aux assises du GDR I3 du CNRS, Strasbourg, France, 30 juin au 2 juillet 2010.

- *« Méta-modélisation du comportement d'un modèle: concepts, modèle et outil »*. Sana Damak Mallouli aux FJC du XXVIIème Congrès INFORSID, Toulouse, France, 26-29 mai 2009.

Annexes

Annexe 1: Liste des fonctions du script Script1_Map_Process

- *Private Function AccCanCreateTab(tabName As String) As Integer*

Cette fonction teste si une table peut être créée.

- *Private Sub AccCreateTable(tabName As String)*

Cette fonction créer une table

- *Private Sub AccEndTable()*

Cette-ci ajoute la table courante à la base de données

- *Private Sub AccAddTabProp(tabName As String, propName As String, propType As Integer, propValue As Variant)*

Celle-ci ajoute une propriété à la table courante

- *Private Sub AccAddColumn(cnam As String, dttp As String, mlen As Long, prec As Integer, isMand As String, lval As String, hval As String, dval As String, colnNo As Integer)*

Celle-ci crée une colonne

- *Private Sub AccAddColProp(tabName As String, colName As String, propName As String, propType As Integer, propValue As Variant)*

Celle-ci ajoute une propriété à la colonne courante

- *Private Sub AccDeleteTable(ByVal tabName As String)*

Celle-ci supprime une table

- *Private Sub AccCreateIndex(ByVal prim As String, ByVal uniq As String, ByVal clus As String, ByVal idxName As String, ByVal tabName As String, ByVal colList As String)*

Celle-ci crée un index

- *Private Sub AccDeleteIndex(ByVal idxName As String, ByVal tabName As String)*

Celle-ci supprime un index

- *Private Sub AccCreateReference(refrName As String, primTab As String, fornTab As String, urul As String, drul As String)*

Celle-ci crée une relation entre deux tables

- *Private Sub AccAddRefrCol(refrName As String, primKey As String, fornKey As String)*

Celle-ci crée une jointure à la relation courante

- *Private Sub AccEndReference(refrName As String)*

Celle-ci ajoute la relation courante

- *Private Sub AccTestError(ret As Long)*

Celle-ci affiche un message d’erreur.

Annexe 2: Liste des jointures du script Script4_Map_Links

```
- AccCreateReference "RC1_HIERARCHIE", "RCs", "RC_Hierarchies", "", ""
- AccCreateReference "RC2_HIERARCHIE", "RCs", "RC_Hierarchies", "", ""
- AccCreateReference "Goal_RC_id", "Goals", "RCs", "", ""
- AccCreateReference "Scenarios_RC_id", "Scenarios", "RCs", "", ""
- AccCreateReference "RI_id", "Realized_Intention","Candidates_Sections", "","
- AccCreateReference "I_RI_SOURCE", "Realized_Intention", "Executed_Sections", "",
  ""
- AccCreateReference "I_RI_TARGET", "Realized_Intention", "Executed_Sections", "",
  ""
- AccCreateReference "RC_SOURCE", "RCs", "Executed_Sections", "", ""
- AccCreateReference "RC_TARGET", "RCs", "Executed_Sections", "", ""
- AccCreateReference "RC_ri_id", "RCs", "Realized_Intention", "", ""
- AccCreateReference "CORRESPONDRE", "Strategies", "Sections", "", ""
- AccCreateReference "I_SOURCE", "Intentions", "Sections", "", ""
- AccCreateReference "I_TARGET", "Intentions", "Sections", "", ""
- AccCreateReference "CONSTRAINTS_1","Strategies","Map_constraints","", ""
- AccCreateReference "CONSTRAINTS_2","Strategies","Map_constraints", "","
- AccCreateReference "PARAM_STRAT", "Strategies", "Parameters", "", ""
- AccCreateReference "I_CANDI_TARGET", "Intentions", "Candidates_Sections", "", ""
- AccCreateReference "I_id", "Intentions", "Realized_Intention", "", ""
- AccCreateReference "Strat_id_exe", "Strategies", "Executed_Sections", "", ""
- AccCreateReference "Strat_id_candi", "Strategies", "Candidates_Sections","","
```

Annexe 3: Procédure foreach .Strategy, script Script5_Map_Init

```
   foreach .Strategy{
   if :Product consommation?;='T' then
       $Prod_consommation='-1'
   else
       $Prod_consommation='0'
   endif
   if :Product generation;='T' then
       $Prod_generation='-1'
   else
       $Prod_generation='0'
   endif
   if :Multiple='T' then
       $Multiple='-1'
   else
       $Multiple='0'
   endif
   '    sql = "insert into [Strategies] (Strat_name, Prod_consume,
Prod_Generate,Multiple) VALUES (''':Strategy
name;''','$Prod_consommation','$Prod_generation','$Multiple')" ' newline
   '    dtbs.Execute (sql) ' newline
   }
```

Annexe 4: Procédure foreach .Intention, script Script5_Map_Init

```
foreach .Intention
/* On récupère le nom de l'intention source */
   $Intention_source=:Intention identifier;
   do ~Intention-source-part
   {
       do ~Strategy-target-part
       {
              do .Strategy
              {
```

```
              /* On récupère le nom de la stratégie */

              $Strategy=:Strategy Identifier;
                     do ~Strategy-source-part
                     {
                            do ~Intention-target-part
                            {
                                   do .Intention
                                   {
                            /* on recupere le nom de l’intention cible*/
                            $Intention_target=:Intention identifier;
                            /* on ajout le tout *
‘ sql = "insert into [Sections] (I_id_source, I_id_target,Strat_id) VALUES
   (‘$Intention_source’, ‘$Intention_target’, ‘$Strategy’)" ‘ newline
‘    dtbs.Execute (sql) ‘ newline
newline
                                   }
                            }
                     }
              }
       }
   }
}
```

Annexe 5: Processus du calcul des candidates

```
Public Sub MaJ_Candidate_section()
‘‘‘‘‘‘‘‘‘‘‘Vide la table candidate_section de notre base de données’’’’’’’’’’’’’’’’’’’
‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
        ‘créer un objet connexion ainsi qu’un objet command
        ‘ Dim myConnexion As OleDbConnection = New
System.Data.OleDb.OleDbConnection(My.Settings.Map_ConnectionString())
        Dim myCommand As OleDbCommand = New System.Data.OleDb.OleDbCommand()
        Dim myCommand7 As OleDbCommand = New System.Data.OleDb.OleDbCommand()
        myCommand7.Connection = myConnexion
        myCommand.Connection = myConnexion
        ‘on implement notre commande
        myCommand.CommandText = "delete * from Candidate_sections;"

‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘Vide la Datagrid candidate_section du Form’’’’’’’’’’’’’’’’’’’’’’’’’
‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
        DataGrid_Candidates.Rows.Clear()

‘‘‘‘‘‘‘‘‘‘‘‘avant de calculer les candidates on vérifie qu’on a pas exécuter la
stratégie qui met fin au process’’’’’’’’’’’’’
        ‘‘‘‘‘‘‘‘‘‘‘‘‘‘ si oui alors le processus est terminer et on indique à
l’utilisateur  que c’est terminé,
        ‘‘‘‘‘‘‘‘‘‘‘‘‘‘ sinon on continue en calculant les candidates

‘ici je récupère l’id de la stratégie qui met fin au processus
        Dim Strat_id_STOP As Integer
        Dim myCom_Stop_exe As OleDbCommand = New System.Data.OleDb.OleDbCommand()
        myCom_Stop_exe.Connection = myConnexion
        myCom_Stop_exe.CommandText = "SELECT strategies.Strat_id from intentions,
sections, strategies where sections.strat_id = strategies.strat_id" & _
                                     " and sections.[I_id-target]=intentions.I_id" & _
                                     " and intentions.I_name=‘Stop’;"
        myConnexion.Open()
        Strat_id_STOP = CInt(myCom_Stop_exe.ExecuteScalar())
        myConnexion.Close()
        ‘Je verifie si cette stratégie a déjà été exécute ou non
```

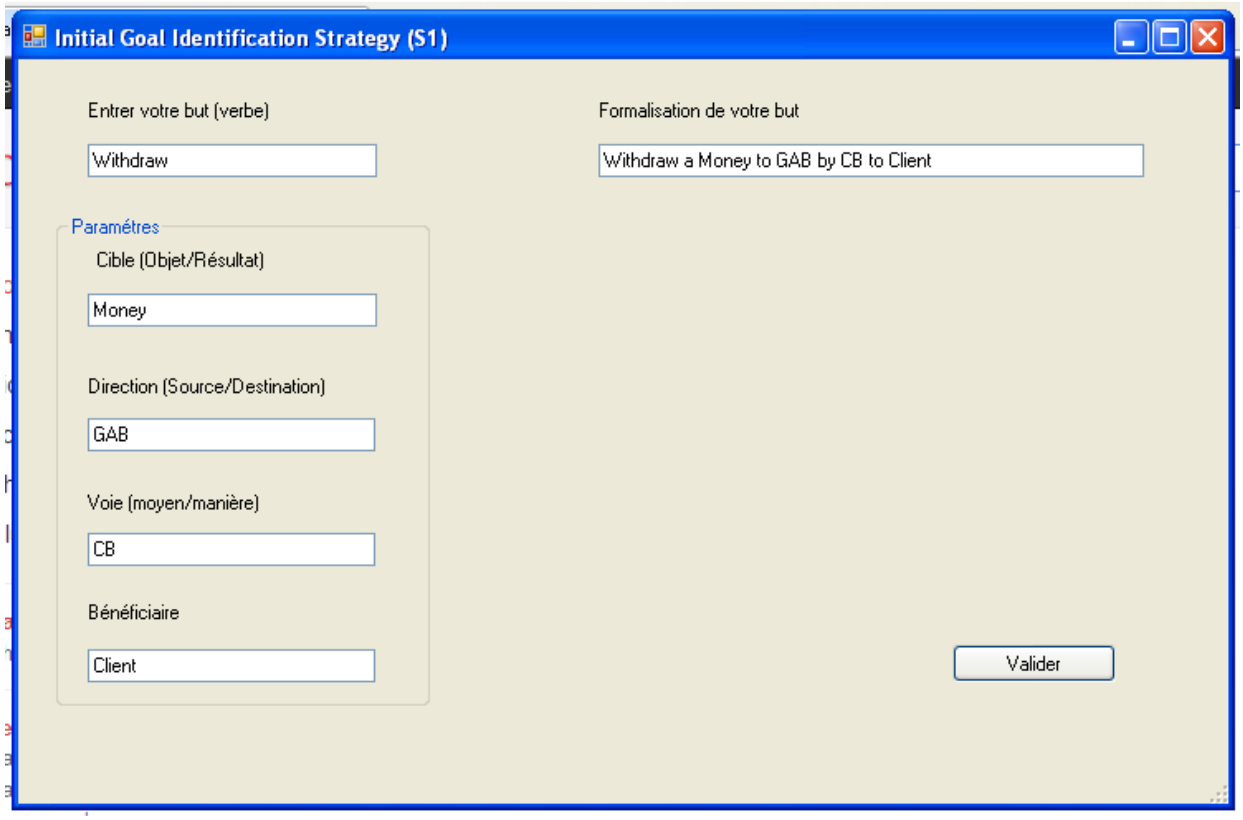
```
        Dim verif_executed As OleDbCommand = New System.Data.OleDb.OleDbCommand()
        verif_executed.Connection = myConnexion
        verif_executed.CommandText = " select strat_id from Executed_sections where 
strat_id=" & Strat_id_STOP & ";"
        myConnexion.Open()
        Dim strat_id_executed As Integer = CInt(verif_executed.ExecuteScalar())
        myConnexion.Close()
        If Strat_id_STOP = strat_id_executed Then
            MsgBox("Vous avez terminé le processus.")

        Else

            Try
                ‘ouvre la connexion a la bdd
                myConnexion.Open()
                ‘execute notre commande
                myCommand.ExecuteNonQuery()
‘‘‘‘‘‘‘On met à jour notre table Candidate_sections dans notre base de données’’’’’’’
‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
 ‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘calcul des candidates’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’’

   ‘ la variable x correspond a l’index de la ligne du tableau candidate_section 
actuel (currentRows)
                Dim x As Integer = -1
                Dim myCommand1 As OleDbCommand = New System.Data.OleDb.OleDbCommand()
                myCommand1.Connection = myConnexion
                myCommand1.CommandText = "Select DISTINCT 
RI_id,Realized_Intention.RI_name From Realized_Intention;"

                Dim RI_id_datareader As OleDbDataReader
                RI_id_datareader = myCommand1.ExecuteReader()
                Do While RI_id_datareader.Read()

                    Dim I_id_source_C_Datareader As OleDbDataReader
                    Dim myCommand3 As OleDbCommand = New 
System.Data.OleDb.OleDbCommand()
                    myCommand3.Connection = myConnexion
                    myCommand3.CommandText = "Select DISTINCT 
Realized_Intention.[I_id] , Intentions.[I_name] " & _
                        "From Realized_Intention, Intentions WHERE 
Realized_Intention.[I_id]=Intentions.[I_id]" & _
                        "and Realized_Intention.[RI_id]=" & 
CInt(RI_id_datareader.GetValue(0)) & ";"

                    I_id_source_C_Datareader = myCommand3.ExecuteReader()

                    Do While I_id_source_C_Datareader.Read()

                        Dim ID_strat_datareader As OleDbDataReader
                        Dim myCommand4 As OleDbCommand = New 
System.Data.OleDb.OleDbCommand()
                        myCommand4.Connection = myConnexion
                        myCommand4.CommandText = "SELECT Strategies.Strat_id, 
Strategies.Strat_name FROM Sections, Strategies" & _
                           " WHERE Sections.Strat_id = Strategies.Strat_id AND 
Sections.[I_id-source]=" & CInt(I_id_source_C_Datareader.GetValue(0)) & ";"

                        ID_strat_datareader = myCommand4.ExecuteReader()

                        Do While ID_strat_datareader.Read()

                            Dim I_id_target_datareader As OleDbDataReader
```

```
                            Dim myCommand5 As OleDbCommand = New
System.Data.OleDb.OleDbCommand()
                            myCommand5.Connection = myConnexion
                            myCommand5.CommandText = "Select DISTINCT Sections.[I_id-
target], Intentions.I_name From Sections , Intentions " & _
                                "WHERE Sections.[I_id-target]=Intentions.[I_id] AND
Sections.[I_id-source]=" & CInt(I_id_source_C_Datareader.GetValue(0)) & " " & _
                                "AND Sections.Strat_id=" &
CInt(ID_strat_datareader.GetValue(0)) & ";"
                            I_id_target_datareader = myCommand5.ExecuteReader()

                            Do While I_id_target_datareader.Read()

                                Dim RC_id_datareader As OleDbDataReader
                                Dim myCommand6 As OleDbCommand = New
System.Data.OleDb.OleDbCommand()
                                myCommand6.Connection = myConnexion
                                myCommand6.CommandText = "Select DISTINCT
Realized_Intention.RC_id, RCs.RC_name, Realized_Intention.RI_id " & _
                                    "From Realized_Intention , RCs Where
Realized_Intention.RC_id=RCs.RC_id " & _
                               "AND RI_id=" & CInt(RI_id_datareader.GetValue(0)) & ";"
                                RC_id_datareader = myCommand6.ExecuteReader()
                                RC_id_datareader.Read()


‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
      ‘ à ce niveau la, on a récupéré les stratégies, les intentions et les produits
      ‘ il faut alors vérifié si ces stratégies ne sont pas exclues suite a une
constraints_exclusion
Dim strat_id_constraint As OleDbDataReader
Dim myCommandConstraint As OleDbCommand = New System.Data.OleDb.OleDbCommand()
myCommandConstraint.Connection = myConnexion
myCommandConstraint.CommandText = "Select Strat_id1, Strat_id2 from Map_constraints" &
_ " where( Strat_id1=" & CInt(ID_strat_datareader.GetValue(0)) & " or Strat_id2=" &
CInt(ID_strat_datareader.GetValue(0)) & ");"
strat_id_constraint = myCommandConstraint.ExecuteReader()
Dim boolean_executed As Boolean = False
  ‘on viens de remplir note datareader des resultats de la requete, 2 cas se presente
a nous
       ‘1) il n’y a aucune contrainte sur cette strategie donc la requete ne retourne
rien et notre datareader est vide
      ‘2) il existe une seule contrainte sur cette strategie ~\ boucle sur le
datareader

Dim cas1 As Boolean = False
  Dim cas2 As Boolean = False
If strat_id_constraint.HasRows Then
  Do While strat_id_constraint.Read()

    ‘si on se trouve dans cette condition cela veut dire que l’une des 2 stratégies du
couple (constraint_exclusion) à déjà été exécuté, il faut alors savoir sur quel
produit, si le produit et different alors on lui propose ces strategies, sinon rien

Dim RC_id_constraint As OleDbDataReader
Dim myCommand8 As OleDbCommand = New System.Data.OleDb.OleDbCommand()
myCommand8.Connection = myConnexion
myCommand8.CommandText = "select distinct [RC_id-in], RI_id_source from
Executed_sections where (Strat_id=" & CInt(strat_id_constraint(0)) & " or Strat_id=" &
CInt(strat_id_constraint(1)) & ") and [RC_id-in] Is Not Null;"
```

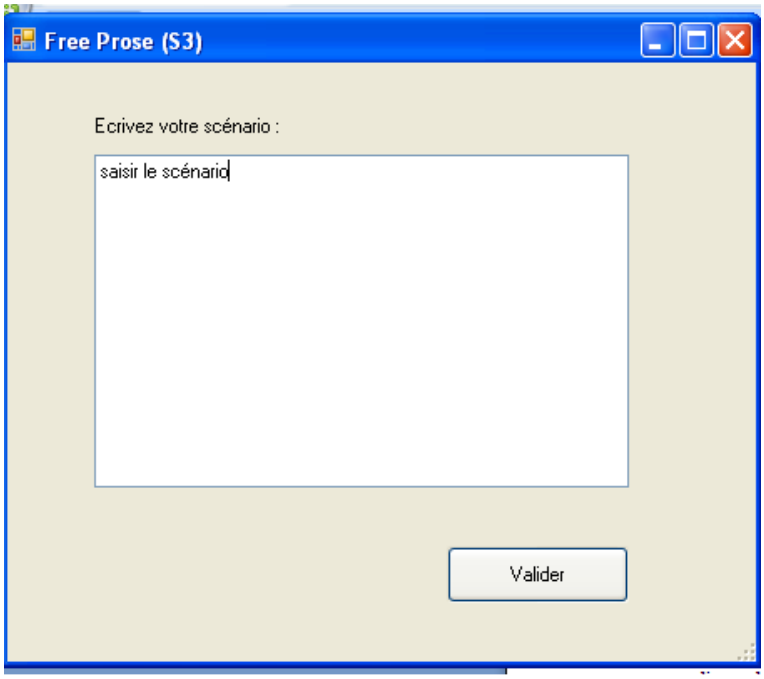
```
RC_id_constraint = myCommand8.ExecuteReader()
If RC_id_constraint.HasRows Then
‘Afin de connaitre le  nombre de ligne dans mon datareader je vais boucler dessus en
incrémenter un Integer
Dim counter As Integer = 0
While RC_id_constraint.Read()
   counter += 1
End While
RC_id_constraint.Close()
Dim maxi As Integer = counter
counter = 0
‘Ici je recharge mon datareader de ma requête pcq’un datareader fonctionne que par
lecture avant
RC_id_constraint = myCommand8.ExecuteReader()
Do While RC_id_constraint.Read()
counter += 1
‘si le produit sur lequel il y a une contrainte est égal au produit de notre pré
candidate alors on ne peut pas l’ajouter aux candidates
   If CInt(RC_id_datareader.GetValue(0)) = CInt(RC_id_constraint.GetValue(0)) _
And CInt(RI_id_datareader.GetValue(0)) = CInt(RC_id_constraint.GetValue(1)) Then

‘Ici il est égal donc plus la peine de tester le reste du datareader
cas2 = False
counter = 0
Exit Do
End If
  Loop
    If maxi = counter Then
      cas2 = True
  End If
Else : cas2 = True
End If
  RC_id_constraint.Close()
Loop
Else
     cas1 = True
End If

If cas1 = True Or cas2 = True Then
   ‘Donc on peut ajouter la requête dans candidates
   myCommand7.CommandText = "insert into Candidate_sections (RI_id, I_id_source,
Strat_id, [I_id_target], [RC_id-in]) values (" & CInt(RI_id_datareader.GetValue(0)) &
", " &  CInt(I_id_source_C_Datareader.GetValue(0)) & ", " &
CInt(ID_strat_datareader.GetValue(0)) & ", "& CInt(I_id_target_datareader.GetValue(0))
& ", " & CInt(RC_id_datareader.GetValue(0)) & ");"
myCommand7.ExecuteNonQuery()

‘‘‘‘‘On met à jour notre Dataset Candidate_sections et notre base de données’’’’’’’’’’
‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘‘
‘configure le datagrid
‘insere dans le datagrid
x = x + 1
  DataGrid_Candidates.Rows.Add()
DataGrid_Candidates.Item(0, x).Value = RI_id_datareader.GetValue(1)
DataGrid_Candidates.Item(1, x).Value = I_id_source_C_Datareader.GetValue(1)
DataGrid_Candidates.Item(2, x).Value = RC_id_datareader.GetValue(1)
DataGrid_Candidates.Item(3, x).Value = ID_strat_datareader.GetValue(1)
DataGrid_Candidates.Item(4, x).Value = I_id_target_datareader.GetValue(1)
End If
strat_id_constraint.Close()
                                ‘ferme le datareader
                                RC_id_datareader.Close()
```

```
                        Loop
                        'ferme le datareader
                        I_id_target_datareader.Close()
                    Loop
                    'ferme le datareader
                    ID_strat_datareader.Close()
                Loop
                'ferme le datareader
                I_id_source_C_Datareader.Close()
            Loop
            'ferme le datareader
            RI_id_datareader.Close()

            'fermeture de la connexion
            myConnexion.Close()

        Catch ex As OleDb.OleDbException
            MessageBox.Show(ex.ToString)
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        Finally
            If Not myConnexion Is Nothing Then
                If myConnexion.State = ConnectionState.Open Then
                    myConnexion.Close()
                End If
            End If
        End Try
    End If
End Sub
```

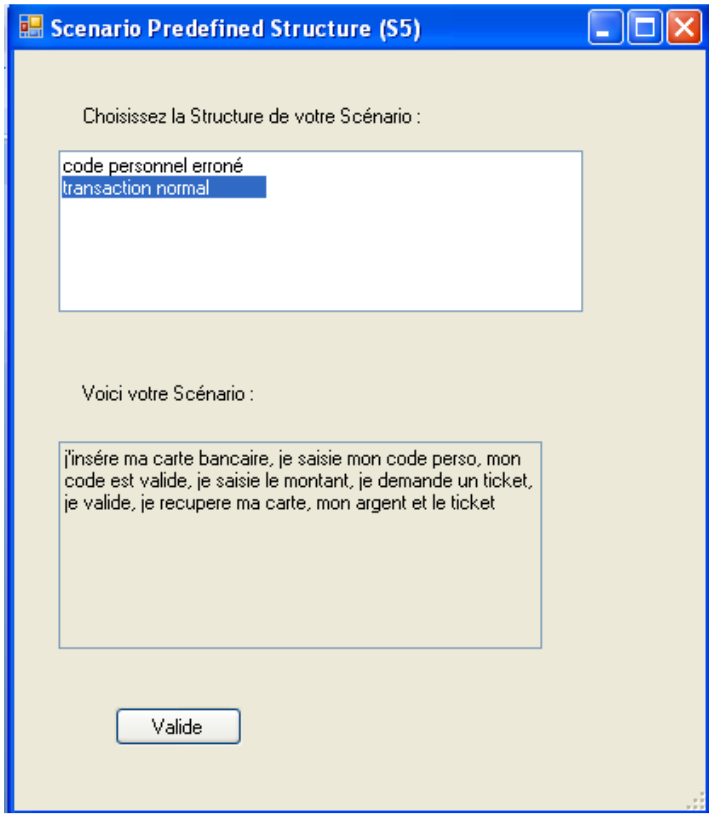
Annexe 6: Initial Goal Identification Strategy



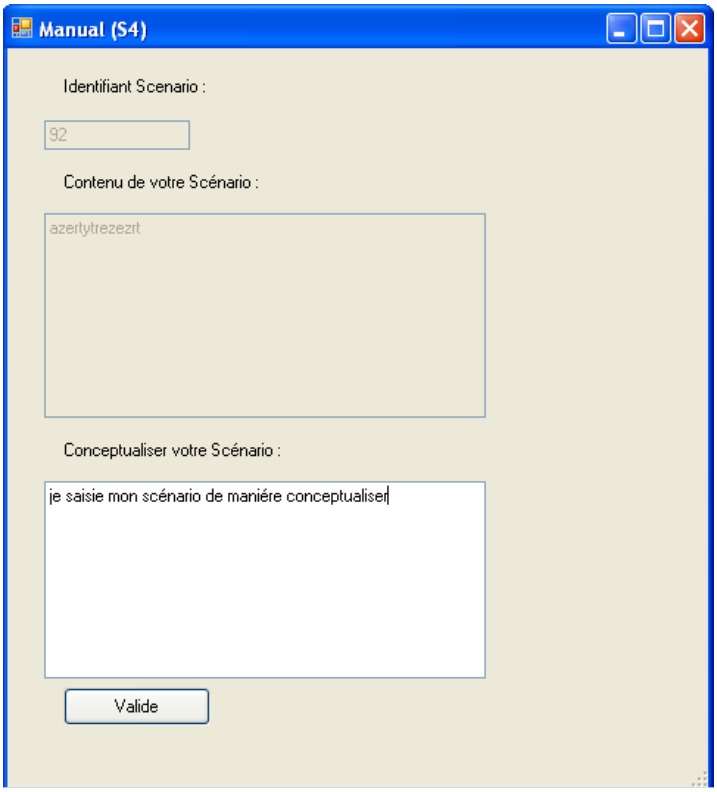
Annexe 7: L'interface de la stratégie « Free prose » permettant à l'utilisateur de saisir son scénario.



Annexe 8: L'interface de la stratégie « Scenario Predefined Structure » qui permet de choisir un scénario prédéfinie.



Annexe 9: L'interface de la stratégie « Manual » : qui permet à l'utilisateur de conceptualiser son scénario de manière manuelle.



Annexe 10: Spécification événementielle en XML de l'exemple du Workflow

```
<Acteur  Nom-Acteur="WF_Participant" Type-Acteur="Humain" ">
        <Evénement-Externe>
                <Evénement Id-Event="EV1" Nom-Event="Démarrage d'un processus"
                 Prédicat- Event =" Existe (WF_Participant,WF_Processus) ">
                        <Trigger Id-Trigger="TR1">
                                <Trigger-Element>
                                        <Opération-Ref> Démarrer </Opération-Ref>
                                        <Objet-Ref>WF_Processus</Objet-Ref>
                                </Trigger-Element>
                                <Trigger-Element>
                                        <Opération-Ref> Activer_1ere_Act </Opération-Ref>
                                        <Objet-Ref>WF_Activité</Objet-Ref>
                                </Trigger-Element>
                        </Trigger>
                </Evénement>
                <Message-Ref>M1</Message-Ref>
        </Evénement-Externe>
        <Evénement-Externe>
                <Evénement Id-Event="EV3" Nom-Event="Sélection d'une activité"
                 Prédicat-Event=" Activité existe & Participant est celui affecté à l'activité ">
                        <Trigger Id-Trigger="TR3">
                                <Trigger-Element>
                                        <Opération-Ref> Selectionner </Opération-Ref>
                                        <Objet-Ref>WF_Activité</Objet-Ref>
                                </Trigger-Element>
                        </Trigger>
                </Evénement>
```

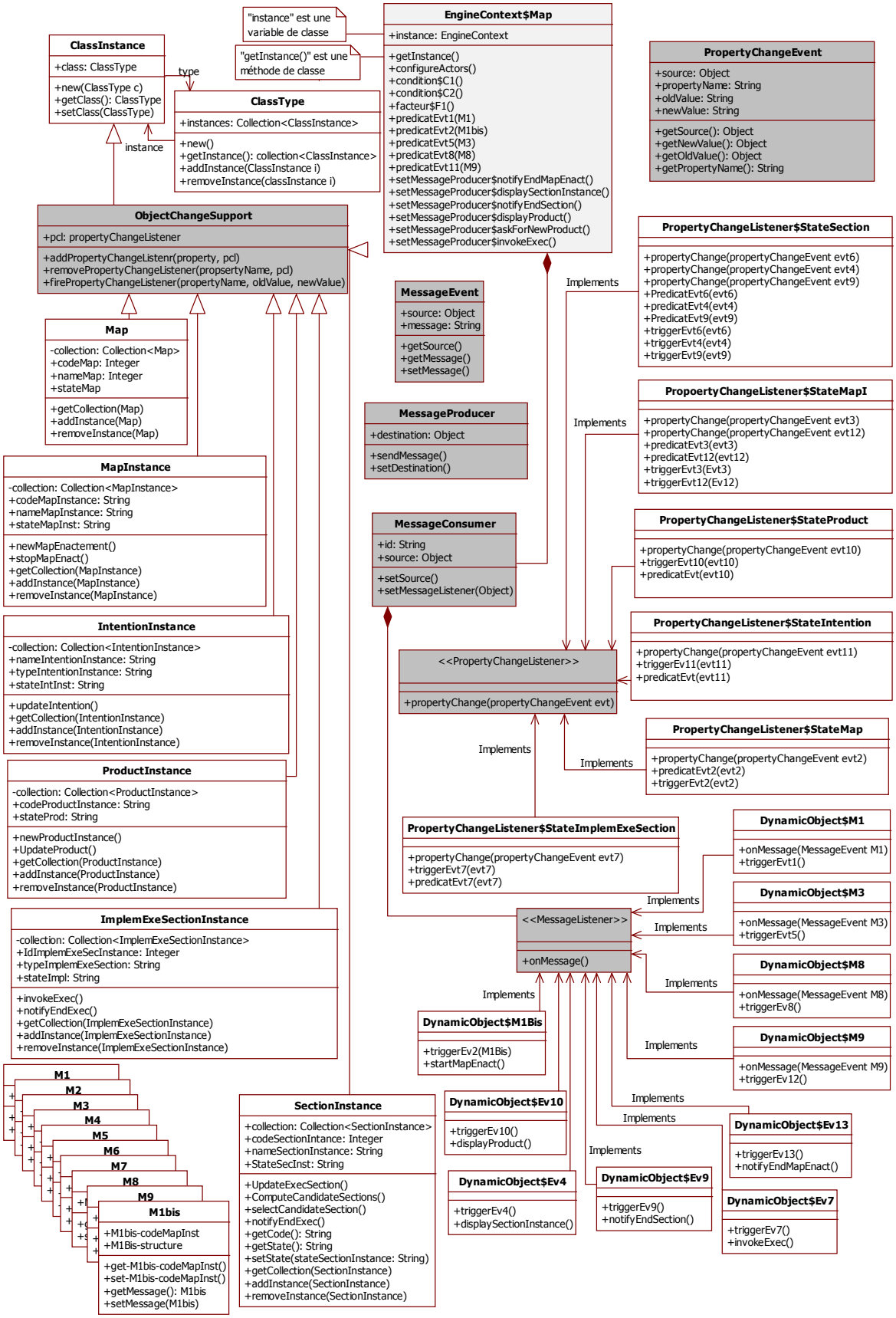
```
                <Message-Ref>M3</Message-Ref>
        </Événement-Externe>

        <Opération-Externe Nom-Op="Notifier_Fin_Process">
                <Args-type Name=" WF_Id" Type="Integer" />
                <Message-Ref>M6</Message-Ref>
        </Opération-Externe>
        <Opération-Externe Nom-Op=" Notifier_Affectation ">
                <Args-type Name=" Act-Id" Type="Integer" />
                <Message-Ref>M2</Message-Ref>
        </Opération-Externe>
</Acteur>
<Acteur  Nom-Acteur="WF_Applicationt" Type-Acteur="Système" ">
        <Evénement-Externe>
        <Evénement Id-Event="EV4" Nom-Event="Fin d'exécution" Prédicat-Event=
                        " Activité existe & Données existent  ">
                        <Trigger Id-Trigger="TR4">
                                <Trigger-Element>
                                        <Opération-Ref> Mise_à_jour </Opération-Ref>
                                        <Objet-Ref>WF_Données</Objet-Ref>
                                </Trigger-Element>
                                <Trigger-Element>
                                        <Opération-Ref> Terminer </Opération-Ref>
                                        <Objet-Ref>WF_Activité</Objet-Ref>
                                </Trigger-Element>
                        </Trigger>
                </Evénement>
                <Message-Ref>M4</Message-Ref>
        </Evénement-Externe>

        <Opération-Externe Nom-Op="Invoquer">
                <Args-type Name=" Id_Appli" Type="Integer" />
                <Message-Ref>M5</Message-Ref>
        </Opération-Externe>

</Acteur>
<Evénement-Interne>
        <Objet-Ref> WF_Activité</Objet-Ref>
                <Evénement
        Id-Event="EV5"
        Nom-Event="Fin d'exécution d'une activité"
        Prédicat-Event = " Act_State.Old='Encours d'exécution' et  Act_State.New= 'Terminée' ">
        <Trigger  Id-Trigger="TR5"">
        <Trigger-Element>
                <Opération-Ref> Terminer_WF </Opération-Ref>
                <Objet-Ref>WF_Processus</Objet-Ref>
                <Condition> C1 <Condition/>
        </Trigger-Element>
        <Trigger-Element>
                < Opération-Ref Nom-Op="Activer_Act_Suivante ">
                        <Args-type >
                                <Attribut Name="Id_Data_Instance" Type="Integer" />
                        <Args-type/>
                </Opération-Ref >
                <Objet-Ref>WF_Activité</Objet-Ref>
                <Condition> C2 <Condition/>
                <Facteur> F1 <Facteur/>
        </Trigger-Element>
</Trigger>
        </Événement>
```

```
</Evénement-Interne>
<Evénement-Interne>
        <Objet-Ref> WF_Activité</Objet-Ref>
        <Evénement
          Id-Event="EV2"
          Nom-Event="Affectation d'une activité"
          Prédicat-Event = " Act_State.Old='Activée' et  Act_State.New= 'EnCours' ">
                  <Trigger  Id-Trigger="TR2">
                          <Trigger-Element>
                                  <Opération-Ref Nom-Op="  Affecter ">
                                  <Args-type >
                                          <Attribut Name=" Id_Participant " Type="Integer" />
                                   <Args-type/>
                          </ Opération-Ref >
                          <Objet-Ref>WF_Activité</Objet-Ref>
                  </Trigger-Element>
                  </Trigger>
          </Evénement>
</Evénement-Interne>

<Evénement-Interne>
        <Objet-Ref> WF_Processus</Objet-Ref>
        <Evénement
          Id-Event="EV6"
          Nom-Event="Notification fin de processus"
          Prédicat-Event = " WF_State.Old='EnCours' et  WF_State.New= 'Terminé' ">
                  <Trigger  Id-Trigger="TR6">
                          <Trigger-Element>
                                  < Opération-Ref Nom-Op=" Notifier_Fin_Processus ">
                                    <Args-type >
                                          <Attribut Name=" WF_Id " Type="Integer" />
                                    <Args-type/>
                                  </ Opération-Ref >
                                  <Objet-Ref>WF_Participant</Objet-Ref>
                          </Trigger-Element>
                  </Trigger>
          </Evénement>
</Evénement-Interne>

<!—La liste des conditions -->

<Condition Id-Condition="C1" Nom-Condition="Activité finale" Texte-Condition=" Vérifier que l'activité
courante est l'activité finale du processus "/>
<Condition Id-Condition="C2" Nom-Condition="Activité non finale" Texte-Condition=" Vérifier que
l'activité courante n'est pas l'activité finale du processus "/>

 <!—La liste des facteurs -->

<Facteur Nom-Facteur="F1" Texte-Facteur="Pour toutes les instances d'activité qui succèdent
l'activité courante"/>

 <!—La liste des messages référencés -->
<Message Id-Message="M1" Nom-Message="Démarrer le processus ">
          <Attribut Name="Id_Participant" Type="Integer"/>
          <Attribut Name="WF_Id" Type="Integer"/>
</Message>

<Message Id-Message="M2" Nom-Message="Activité Affectée ">
          <Attribut Name="Id_Participant" Type="Integer"/>
          <Attribut Name="Act_Id" Type="Integer"/>
```

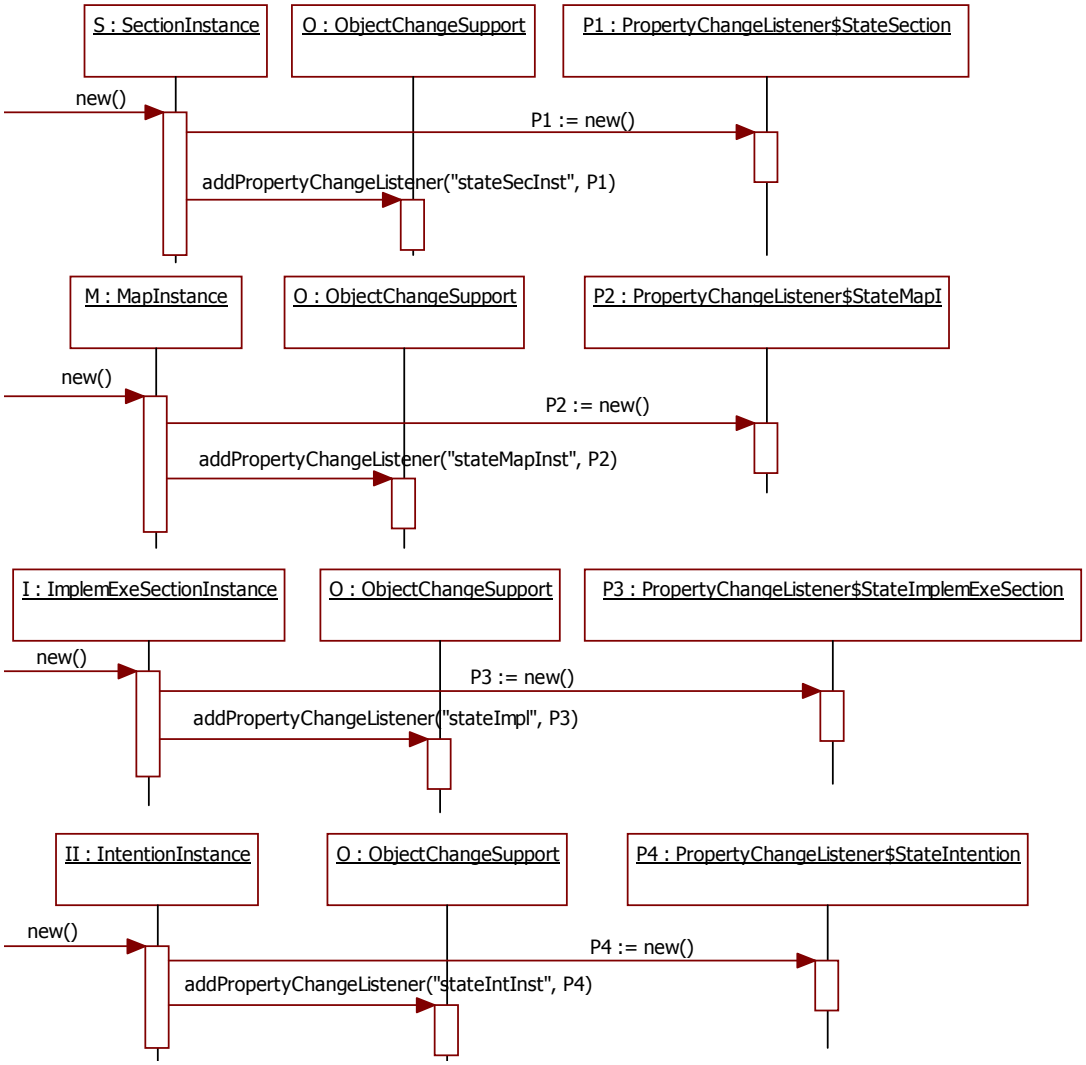
```
</Message>
<Message Id-Message="M3" Nom-Message="Sélectionner activité ">
        <Attribut Name="Id_Participant" Type="Integer"/>
        <Attribut Name="Act_Id" Type="Integer"/>
</Message>
<Message Id-Message="M4" Nom-Message="Fin_activité ">
        <Attribut Name="Act_Id" Type="Integer"/>
        <Attribut Name="Id_Data" Type="Integer"/>
</Message>
<Message Id-Message="M5" Nom-Message="Fin du processus ">
        <Attribut Name="WF_Id" Type="Integer"/>
        <Attribut Name="Id_Participant" Type="Integer"/>
</Message>

<!—La liste des opérations référencés -->
<Opération Nom-Op="Démarrer">
        <Args-type Name=" Id_Participant" Type="Integer" />
</Opération >
<Opération Nom-Op="Terminer"/>
<Opération Nom-Op="Sélectionner">
        <Args-type Name=" Id_Participant" Type="Integer" />
</Opération >
<Opération Nom-Op="Activer_Act_suivante">
        <Ret-type Name=" Act_Id_Instance" Type="Integer" />
</Opération >
<Opération Nom-Op="Mettre_à_jour">
        <Args-type Name=" Data" Type="String" />
</Opération >
```

Annexe 11 : L'architecture OO du moteur d'exécution du Map

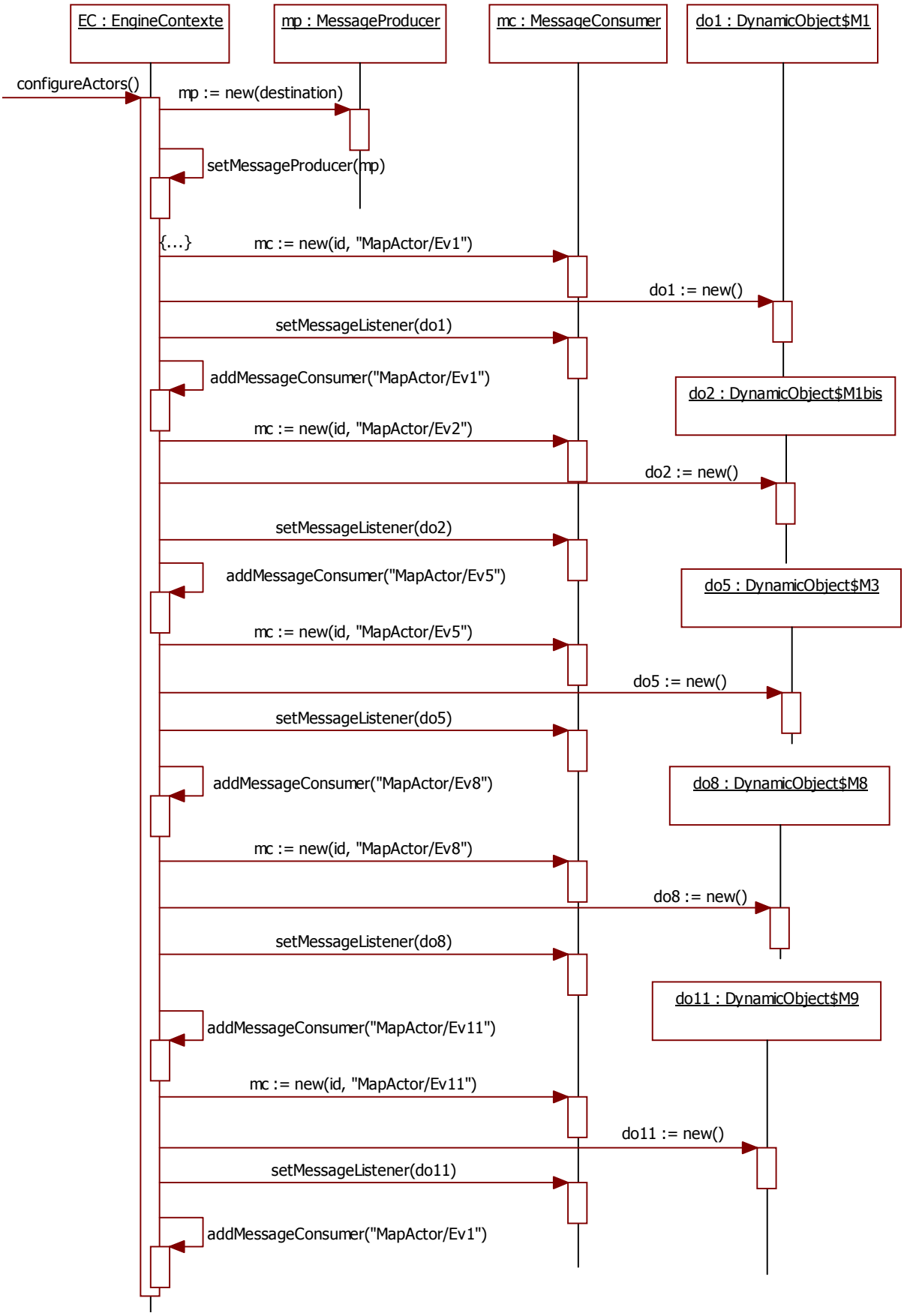
Le schéma suivant représente les classes de l'architecture orientée objet du moteur d'exécution du Map



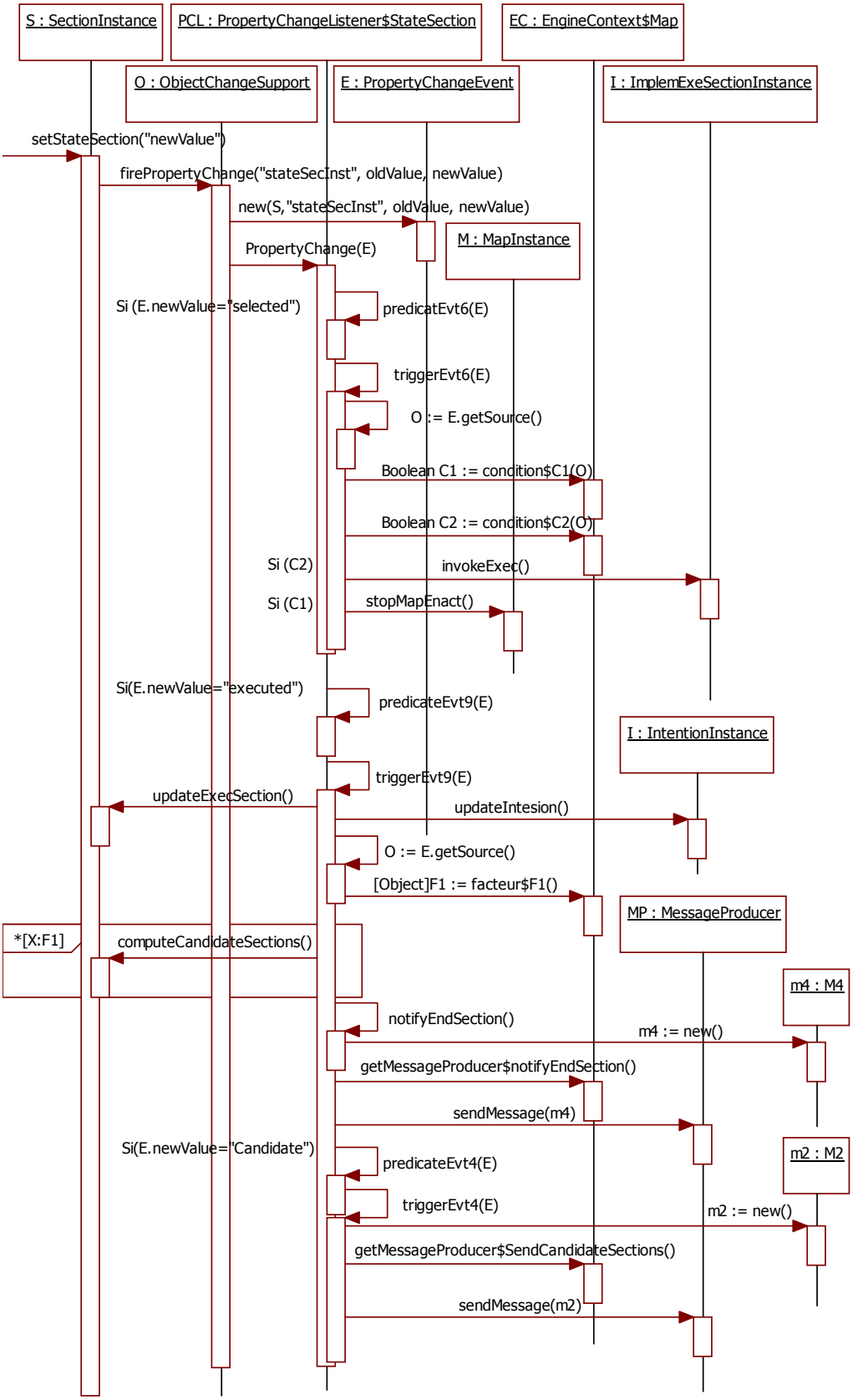
Le diagramme de séquence suivant concerne la **création** des objets dans le cas des événements internes constatés sur les objets ***SectionInstance*** et ***MapInstance.***



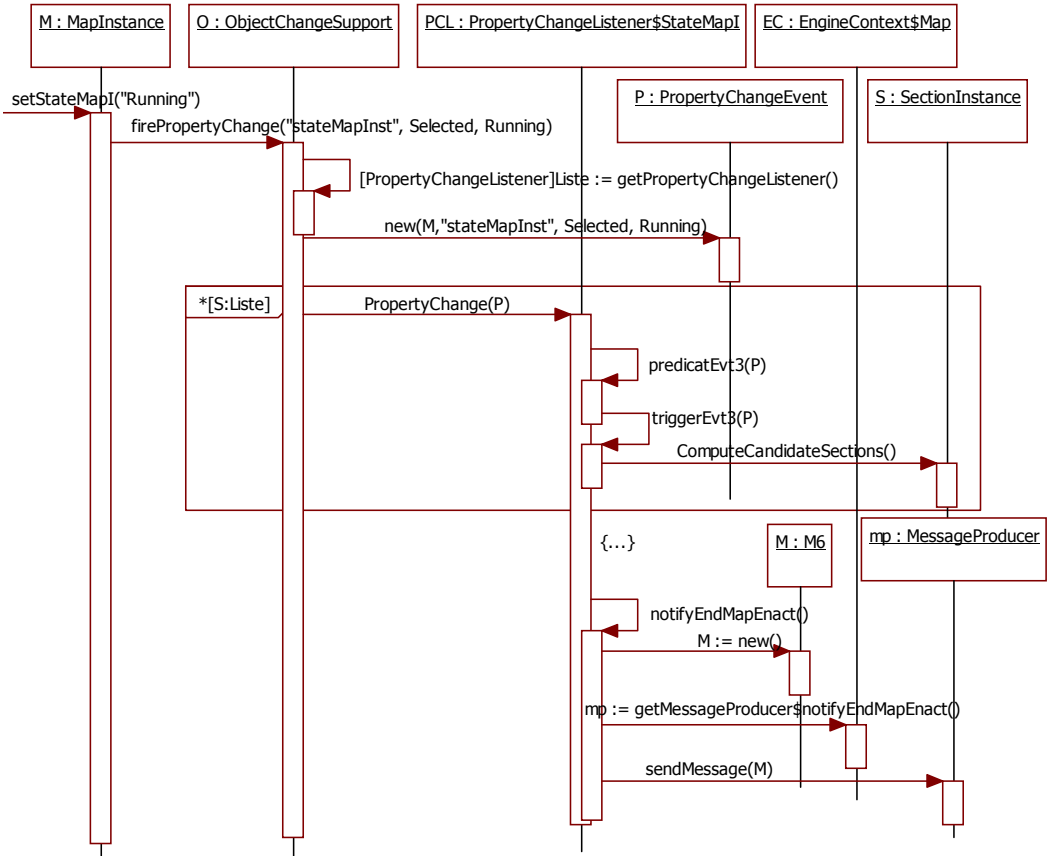
Le diagramme de séquence suivant correpond à la création des objets dans le cas des opérations externes déclenchées par EV4, EV7, EV9, EV10 et EV12, et dans le cas des événements externes EV1, EV2, EV5, EV8 et EV11.

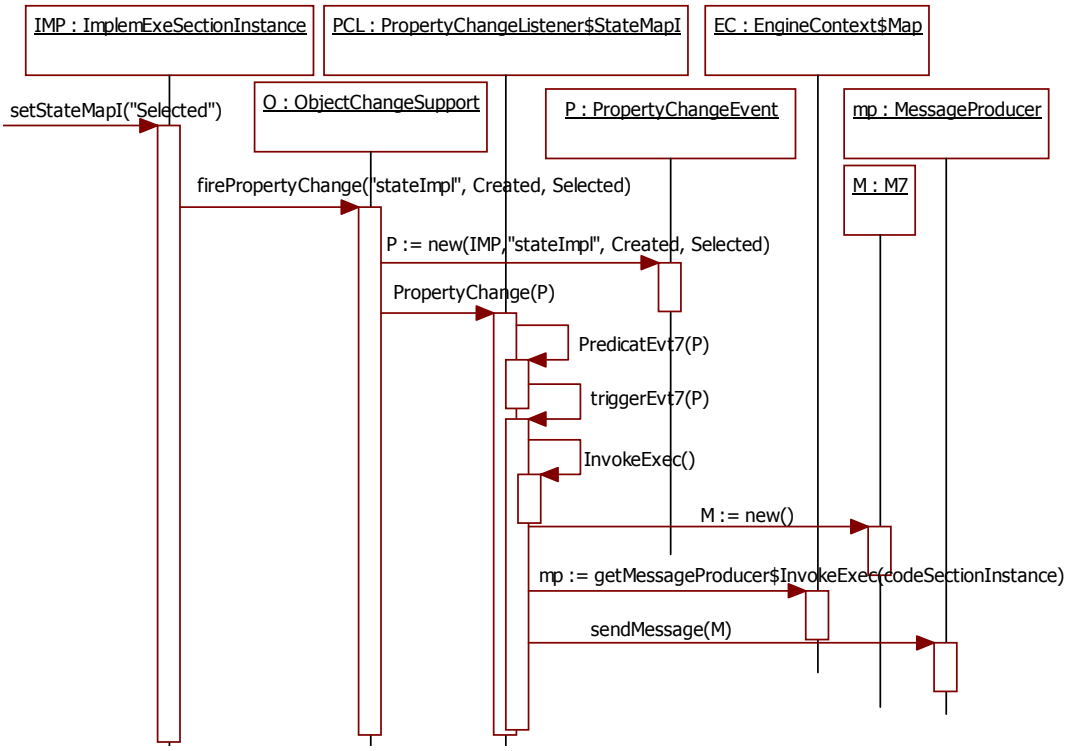

EC : EngineContexte
mp : MessageProducer
mc : MessageConsumer
do1 : DynamicObject$M1
configureActors()
mp := new(destination)
setMessageProducer(mp)
{...}
mc := new(id, "MapActor/Ev1")
do1 := new()
setMessageListener(do1)
addMessageConsumer("MapActor/Ev1")
do2 : DynamicObject$M1bis
mc := new(id, "MapActor/Ev2")
do2 := new()
setMessageListener(do2)
addMessageConsumer("MapActor/Ev5")
do5 : DynamicObject$M3
mc := new(id, "MapActor/Ev5")
do5 := new()
setMessageListener(do5)
addMessageConsumer("MapActor/Ev8")
do8 : DynamicObject$M8
mc := new(id, "MapActor/Ev8")
do8 := new()
setMessageListener(do8)
do11 : DynamicObject$M9
addMessageConsumer("MapActor/Ev11")
mc := new(id, "MapActor/Ev11")
do11 := new()
setMessageListener(do11)
addMessageConsumer("MapActor/Ev1")

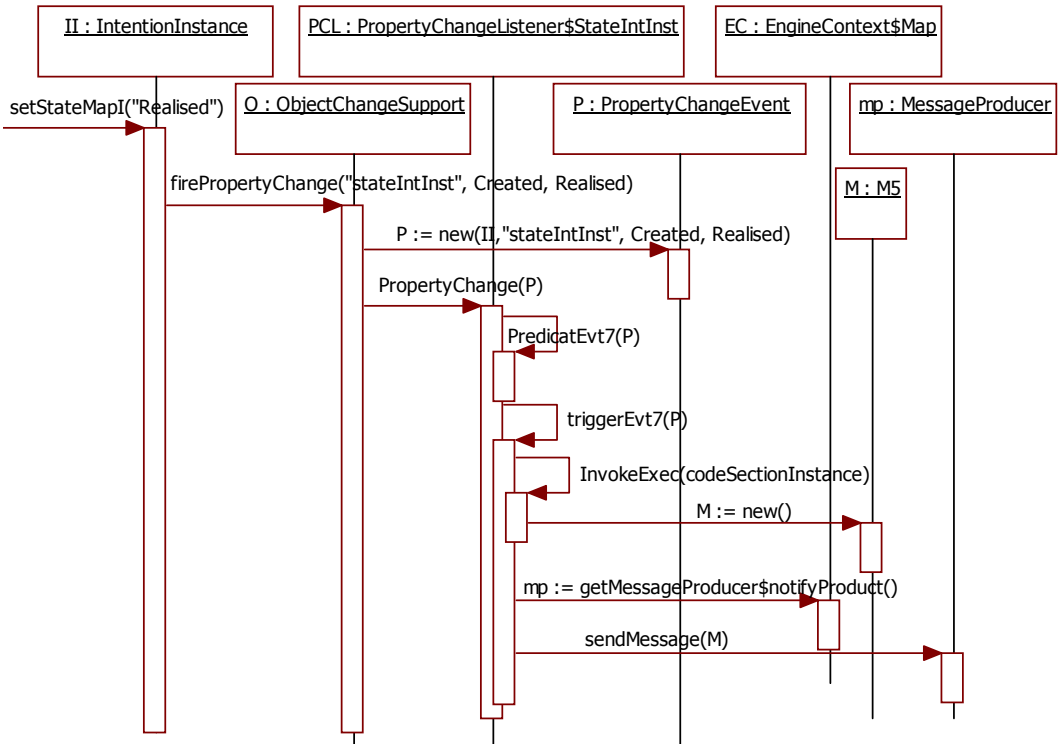
Le diagramme de séquence suivant décrit **l'exécution** des événements internes qui sont constatés sur l'objet ***SectionInstance*** ainsi que le traitement des opérations externes associées à cet évenement :



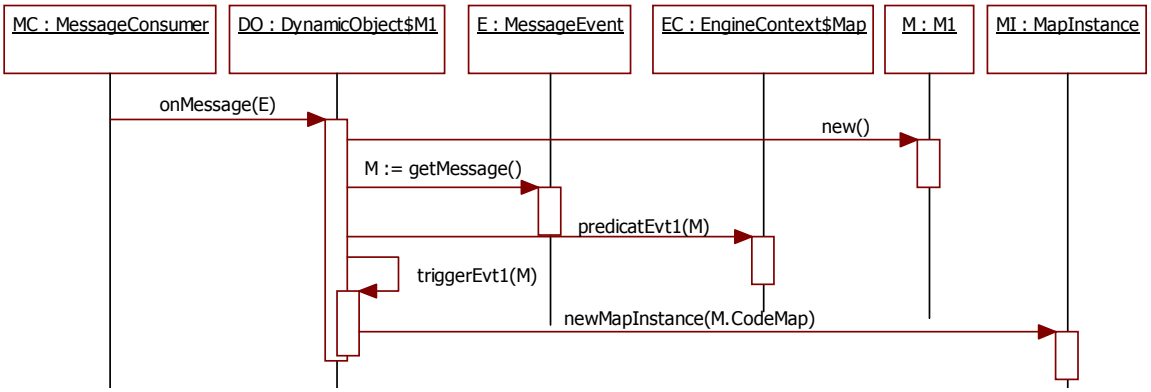
Les diagrammes de séquence suivants représentent **l'exécution** des événements internes (EV3, EV7 et EV10), constatés respectivement sur les objets ***MapInstance, ImplemExeSectionInstance et IntentionInstance,*** ainsi que l'exécution des opérations externes associées à ces évenements :

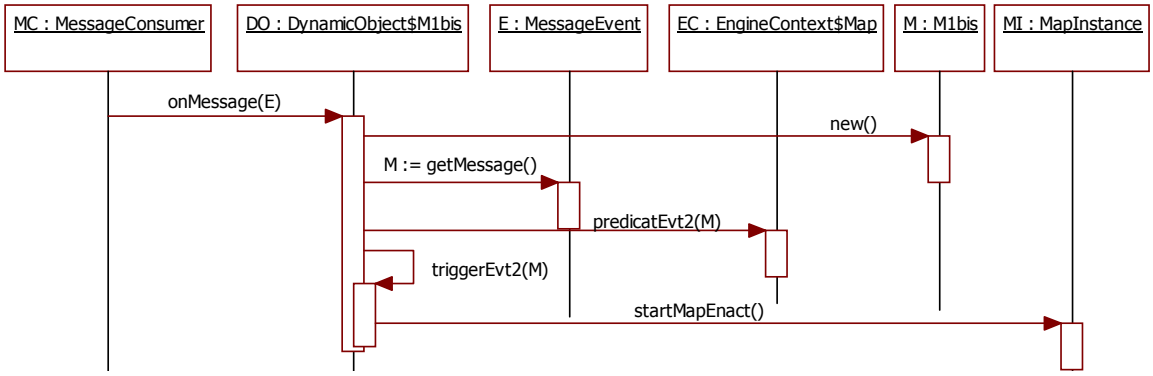


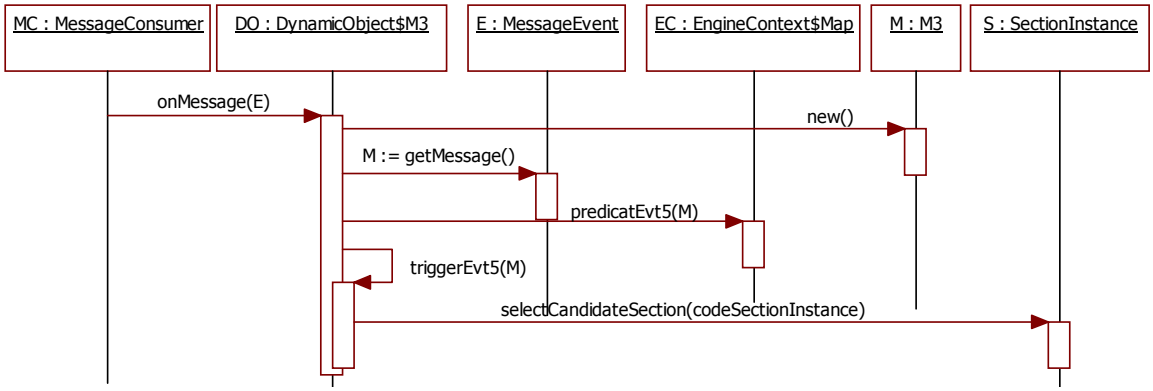


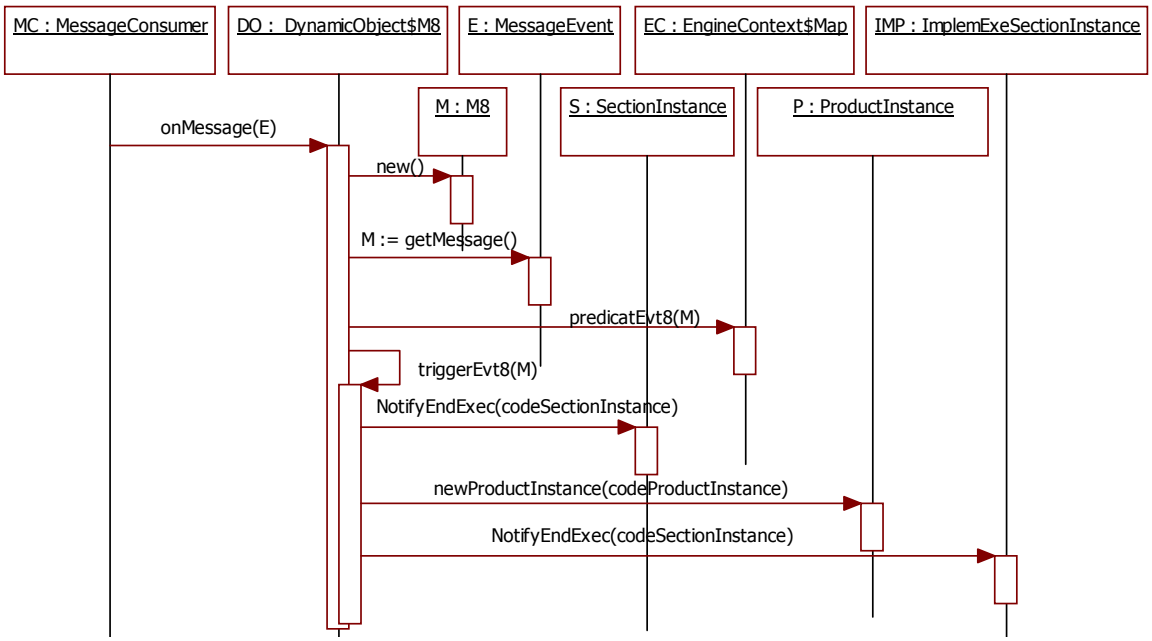


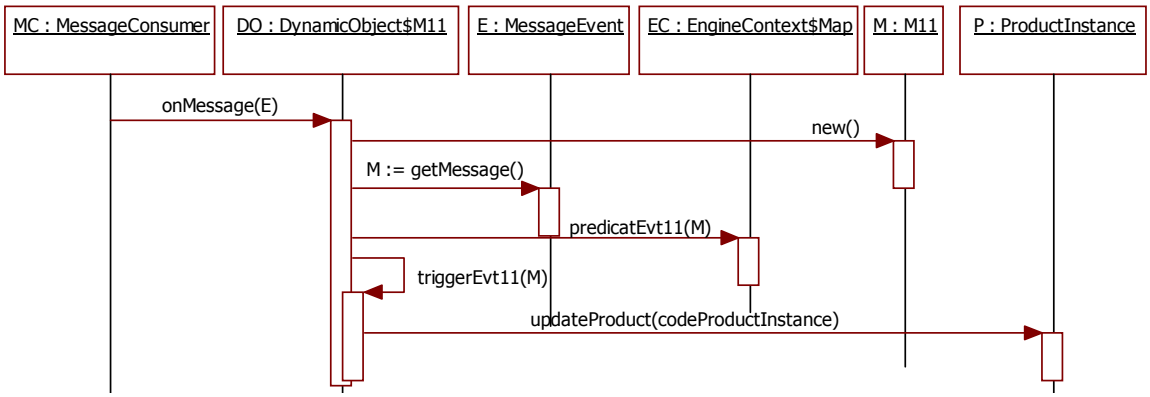
Les cinq diagrammes de séquence suivants sont relatifs à la gestion des événements externes EV1, EV2, EV5, EV8 et EV11.






MC : MessageConsumer
DO : DynamicObject$M3
E : MessageEvent
EC : EngineContext$Map
M : M3
S : SectionInstance
onMessage(E)
new()
M := getMessage()
predicatEvt5(M)
triggerEvt5(M)
selectCandidateSection(codeSectionInstance)


MC : MessageConsumer
DO : DynamicObject$M8
E : MessageEvent
EC : EngineContext$Map
IMP : ImplemExeSectionInstance
M : M8
S : SectionInstance
P : ProductInstance
onMessage(E)
new()
M := getMessage()
predicatEvt8(M)
triggerEvt8(M)
NotifyEndExec(codeSectionInstance)
newProductInstance(codeProductInstance)
NotifyEndExec(codeSectionInstance)


MC : MessageConsumer
DO : DynamicObject$M11
E : MessageEvent
EC : EngineContext$Map
M : M11
P : ProductInstance
onMessage(E)
new()
M := getMessage()
predicatEvt11(M)
triggerEvt11(M)
updateProduct(codeProductInstance)