Année : 2008

THÈSE

soutenue en vue de l'obtention du titre de

DOCTEUR EN INFORMATIQUE
DE L'UNIVERSITÉ DE TOULOUSE

délivré par

l'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Approche de métamodélisation pour la simulation et la vérification de modèle

Application à l'ingénierie des procédés

Présentée et soutenue publiquement par

BENOÎT COMBEMALE

le 11 juillet 2008 à l'ENSEEIHT,
devant le jury composé des membres suivants :

Rapporteurs : Jean BÉZIVIN
Professeur, INRIA-ATLAS, Université de Nantes
Pierre-Alain MULLER
Professeur, Université de Haute-Alsace

Examinateurs : Antoine BEUGNARD (président)
Professeur, ENST de Bretagne
François VERNADAT
Professeur, LAAS CNRS, Université de Toulouse
Bernard COULETTE
Professeur, IRIT, Université de Toulouse
Xavier CRÉGUT (encadrant)
Maître de conférences, IRIT, Université de Toulouse

Invités : Patrick FARAIL
AIRBUS France

ÉCOLE DOCTORALE MITT
« Mathématiques Informatique et Télécommunications de Toulouse »

Directeurs : Patrick SALLÉ (IRIT, ENSEEIHT) et Bernard COULETTE (IRIT, UTM)
Laboratoire : Institut de Recherche en Informatique de Toulouse (IRIT, UMR 5505)

Remerciements

La réalisation d'une thèse est certes un long travail mais aussi une aventure humaine sans laquelle nos recherches auraient peu de sens. C'est celle-ci que je vais essayer de retracer dans ces remerciements. Cette étape importante n'est pas la plus simple, et je vous prie par avance de m'excuser pour les oublis éventuels.

Mes remerciements vont tout d'abord à JEAN BÉZIVIN et PIERRE-ALAIN MULLER pour avoir accepté de relire mon manuscrit et d'en être les rapporteurs. Je tiens aussi à remercier Jean pour son écoute tout au long de mes travaux et pour nos nombreux échanges à l'occasion de conférences, qui ont su à chaque fois m'éclairer dans mon travail. Un merci également à Pierre-Alain pour sa relecture minutieuse du manuscrit, ses remarques enrichissantes et sa vision éclairée permettant de mettre en perspectives mon travail.

Je remercie aussi les personnes qui ont acceptées d'évaluer mon travail et de participer à mon jury. ANTOINE BEUGNARD pour avoir présidé le jury et pour avoir partagé avec moi des discussions très intéressantes. FRANÇOIS VERNADAT pour nos collaborations tout au long des années, sa disponibilité et sa réactivité dans le travail. PATRICK FARAIL pour sa disponibilité et ses compétences qui ont fait de ma participation au projet TOPCASED un moment aussi enrichissant qu'agréable.

Merci à PATRICK SALLÉ et BERNARD COULETTE pour m'avoir fait confiance tout au long de ces années et pour avoir accepté de diriger cette thèse. Merci également à MARC PANTEL pour m'avoir permis de m'intégrer dans différents projets de recherche et de participer à de nombreux congrès.

Un éternel merci à XAVIER CRÉGUT. Merci d'abord pour ta confiance et pour avoir accepté d'encadrer mon stage de Master Recherche et mes travaux de thèse. Ton écoute, ton soutien sans limite et ton amitié ont fait de ces années des moments inoubliables et si agréables. Par ailleurs, ta disponibilité, ta rigueur et tes conseils ont fait de cette période une formation à la recherche très enrichissante et l'objet d'un travail intense. Pour tout cela je te suis infiniment redevable.

Mes travaux se sont nourris de nombreux échanges qui ont su baliser le chemin de ma thèse. Fruits de rencontres multiples, ils ont ainsi contribué à l'aboutissement de mon travail et à le rendre si agréable et passionnant. Pour cela, je remercie RÉDA BENDRAOU, MARIE-PIERRE GERVAIS, PIERRE MICHEL et FRANÇOIS VERNADAT, mais également tous ceux avec qui j'ai eu la chance d'échanger.

Ce travail a aussi et avant tout été réalisé au sein d'une équipe de recherche dont les relations entre ses membres sont le moteur au quotidien de mes travaux. Ainsi, merci à XAVIER THIRIOUX et PIERRE-LOÏC GAROCHE pour les nombreux moments et réunions de travail passés ensemble, à chaque fois très agréables. Un clin d'oeil à Pierre-Loïc pour son soutien constant et son amitié égayant au quotidien le travail. Merci aussi pour tous ces cafés préparés avec amour. Je remercie d'autre part mes *co*-bureau, MARCEL GANDRIAU et GÉRARD PADIOU, qui m'ont accueilli parmi eux comme un véritable collègue et m'ont toujours offert de leurs temps et leurs écoutes lorsque j'ai rencontré des difficultés. Merci enfin aux autres collègues pour les moments agréables passés ensemble : NASSIMA, NADÈGE, LES PHILIPPE(S), SANDRINE, TANGUY, ...

Si l'objet de ces remerciements est de retracer l'aventure humaine vécue au cours de ces trois dernières années, je ne peux oublier DANIEL HAGIMONT pour notre collaboration dynamique, amicale et plaisante.

En qualité de responsable pédagogique et pour son écoute constante au cours de mes trois années de monitorat, je tiens à remercier MARC BOYER. Je remercie également CHRISTIAN FRABOUL pour son accueil au sein de l'équipe pédagogique de son département de l'ENSEEIHT.

Je tiens aussi à souligner la disponibilité, le soutien permanent et la bonne humeur de notre grand et meilleur chef Michel ainsi que des secrétaires exceptionnelles, Les Sylvies et Sabyne.

Pour terminer cette aventure humaine mais néanmoins professionnelle, je ne peux oublier l'équipe de l'IUT de Blagnac, avec un merci particulier à LAURENCE, ALAIN et JEAN-MICHEL pour leur soutien constant et leur présence lors de ma soutenance.

Ces années de travail n'auraient pu être réalisées sans un soutien extérieur et infini de ma famille et de mes amis. Mes parents d'abord, MARTINE et CHRISTIAN, sans lesquels je n'aurai jamais pu en arriver là. Merci pour cela, merci aussi pour votre compréhension et vos sacrifices. Merci aussi à mes soeurs, SANDRINE et STÉPHANIE, ainsi qu'à leurs petites familles. je leur prie de voir aussi dans ces quelques lignes mes excuses pour l'éloignement toutes ces dernières années. Enfin, VÉRONIQUE, qui a partagé au quotidien avec moi les bons comme les mauvais moments avec une compréhension saisissante. Elle a su par son amour baliser le chemin de mon bonheur et égayer chaque jour passé pour préparer cette thèse.

Merci au KREUL, mon éternel compagnon de route, ainsi qu'a toute l'équipe de FASOLIDARITÉ. Merci pour la bonne humeur d'ELSA, de NANOU et de LUCAS. Merci pour les moments de détente avec toute *La bande* de Millau ainsi que pour leurs amitiés. DADOU, LUCAT, LUCHIO, PSYKO, PALPABLE, KEBAB, L'ITALIEN, BOUTCH, LE NAIN, KOALA, PIERROT,LES JO, FANNY et A-G-UEDE, recevez ici toute mon amitié. Et à ceux que je n'oublierai jamais, CARO et YANN.

Résumé

L'ingénierie dirigée par les modèles (IDM) a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles. Une des idées phares est d'utiliser autant de langages de modélisation différents (*Domain Specific Modeling Languages* – DSML) que les aspects chronologiques ou technologiques du développement le nécessitent. Le défi actuel de la communauté du génie logiciel est de simplifier la définition de nouveaux DSML en fournissant des technologies du *métaniveau* telles que des générateurs d'éditeurs syntaxiques (textuels ou graphiques), des outils d'exécution, de validation et de vérification (statique et dynamique). Ces outils de validation et de vérification nécessitent d'expliciter, en plus de la syntaxe abstraite, la sémantique d'exécution du DSML.

Au regard des travaux existants dans l'IDM et de l'expérience acquise avec les langages de programmation, nous proposons dans cette thèse une taxonomie précise des techniques permettant d'exprimer une sémantique d'exécution pour un DSML. Nous replaçons ensuite ces différentes techniques au sein d'une démarche complète permettant de décrire un DSML et les outils nécessaires à l'exécution, la vérification et la validation des modèles.

La démarche que nous proposons offre une architecture rigoureuse et générique de la syntaxe abstraite du DSML pour capturer les informations nécessaires à l'exécution d'un modèle et définir les propriétés temporelles qui doivent être vérifiées. Nous nous appuyons sur cette architecture générique pour expliciter la sémantique de référence (c.-à-d. issue de l'expérience des experts du domaine) et l'implanter. Plus particulièrement, nous étudions les moyens :

- d'exprimer et de valider la définition d'une traduction vers un domaine formel dans le but de réutiliser des outils de *model-checking* et permettre ainsi la vérification formelle et automatique des propriétés exprimées.
- de compléter la syntaxe abstraite par le comportement ; et profiter d'outils génériques pour pouvoir simuler les modèles construits.

Enfin, il est indispensable de valider les différentes sémantiques implantées vis-à-vis de la sémantique de référence. Aussi, nous proposons un cadre formel pour la métamodélisation, transparent pour le concepteur, permettant d'exprimer la sémantique de référence qui servira de base à cette validation.

La démarche est illustrée dans cette thèse à travers le domaine des procédés de développement. Après une étude approfondie de cette ingénierie, et plus particulièrement du langage SPEM proposé par l'OMG, nous définissons l'extension XSPEM permettant de construire des modèles exécutables. Nous décrivons également les outils permettant la vérification formelle et la simulation des modèles XSPEM. La démarche proposée est toutefois générique et a donné lieu à des transferts technologiques dans le projet TOPCASED en établissant des fonctionnalités de vérification et de simulation pour différents langages de l'atelier.

Abstract

The model-driven engineering (MDE) has allowed several significant improvements in the development of complex systems by putting the focus on a more abstract concern than the classical programming. It is a form of generative engineering in which (all or part of) an application is generated from models. One of the main ideas is to use many different *Domain Specific Modeling Languages* (DSML) as required over the time of the development or by technological aspect. The current challenge of the software engineering community is to simplify the definition of new DSML and thus providing technologies of *meta-level* such as syntactic editor generators (textual or graphical) or tools for model execution, validation and verification (static and dynamic). In addition to the abstract syntax, validation and verification tools need to define the execution semantics of the DSML.

In the light of existing work in the MDE community and experience with programming languages, we propose in this thesis a specific taxonomy of the mechanisms to express a DSML execution semantics. Then, we replace these different mechanisms within a comprehensive approach to describe a DSML and the tools required for model execution, verification and validation.

The proposed approach provides a rigorous and generic architecture for DSML abstract syntax to capture the information required for model execution. It also defines the temporal properties which must be checked. We rely on this generic architecture to explain the reference semantics (i.e. from the domain expert experiences) and to implement it. More specifically, we are studying the ways :

- to express and validate the definition of a translation into a formal domain in order to re-use model-checker and to allow the formal and automatic verification of the expressed properties.
- to complete the abstract syntax with the definition of the behaviour and to take advantage of generic tools to simulate the built models.

Finally, it is essential to validate the equivalence of the different semantics implemented according to the reference semantics. We also propose a formal metamodeling framework, transparent for the designer, to express the reference semantics. This framework will be the base for this validation.

The approach is illustrated in this thesis through the process engineering field. After a thorough study of this engineering, and especially SPEM the language proposed by OMG, we define an executable extension called xSPEM. We also describe the tools for formal verification and simulation of xSPEM models. Moreover, the approach proposed is generic and implemented in the TOPCASED project to provide verification and simulation facilities for different languages of the toolkit.

Table des matières

Table des figures

Listings

Chapitre 1

Introduction générale

Dans ce chapitre, nous commençons par décrire l'évolution des techniques et méthodes informatiques (section 1.1) qui, depuis 50 ans, ont pour objectif d'améliorer la qualité et la productivité du développement de logiciel. Nous détaillons ensuite les défis actuels (section 1.2) de la communauté du génie logiciel afin d'offrir les moyens de maîtriser des systèmes informatiques qui ne cessent de croître, tant par leur complexité que par leur caractère critique. L'ingénierie du logiciel s'oriente pour cela vers l'ingénierie dirigée par les modèles (IDM), au sein de laquelle un système est vu non pas comme une suite de lignes de code mais comme un ensemble de modèles plus abstraits et décrivant chacun une vue (c.-à-d. une préoccupation) particulière sur le système. Cette approche favorise l'utilisation de petits langages dédiés (*Domain Specific Modeling Language*, DSML), aux concepts plus abstraits et traduisant l'expérience des développeurs de plus en plus spécialisés en raison de la multiplication des domaines d'application de l'informatique. Il s'agit principalement d'augmenter le niveau d'abstraction des développements en permettant aux développeurs de se concentrer sur leurs préoccupations à l'aide de langages spécifiques à leur domaine. Cette ingénierie en émergence nécessite encore beaucoup d'études afin de fournir les mêmes outils sur les modèles dont disposent les développeurs pour les langages de programmation. Par exemple, il est important de fournir des outils de vérification et de validation pour s'assurer au plus tôt que les modèles construits sont corrects et répondent bien aux exigences. Nous étudions pour cela la construction même des langages de modélisation et proposons une approche tant pour définir les aspects dynamiques d'un DSML que les outils exploitant cette description pour vérifier et valider les modèles construits. Nous souhaitons également profiter de l'abstraction offerte par cette ingénierie pour définir des outils génériques réutilisables par plusieurs DSML. Nous détaillons les objectifs de cette thèse dans la section 1.3. Nous concluons ce chapitre en présentant le contexte de ces travaux (section 1.4) et en détaillant le plan de cette thèse (section 1.5).

1.1 Évolution du génie logiciel

L'arrivée de l'IDM est la suite d'une longue évolution en génie logiciel, dont l'objectif est de fournir les méthodes, les techniques et les outils concourant à la production d'un logiciel. L'IDM vise à fournir des langages de modélisation, plus abstraits et plus facilement maîtrisables que des langages de programmation tel que Java ou C, au même titre que ces derniers revendiqués une abstraction par rapport au langage directement interprétable par une machine tel que l'assembleur.

Un des aspects de l'abstraction apporté par l'IDM auquel nous nous intéressons dans cette thèse est l'utilisation de petits langages, dédiés à un domaine d'application particulier de l'informatique. Cet aspect entraîne la multiplication des langages de modélisation dont la définition, appelé *métamodélisation*, doit être maîtrisée. Dans le contexte de l'IDM, la définition d'un langage de modélisation a pris naturellement la forme d'un modèle (appelé *métamodèle*), et s'apparente à la modélisation, en considérant le langage de modélisation comme le système à définir. On peut alors définir un langage par différents modèles de manière à séparer les différents aspects d'un langage. Il est communément admis qu'au même titre que les langages de programmation ou la linguistique, un DSML peut être défini selon :

- sa *syntaxe abstraite* qui décrit les constructions du langage et leurs relations,
- sa *syntaxe concrète* qui décrit le formalisme permettant à l'utilisateur de manipuler les constructions du langage,
- sa *sémantique* qui décrit le sens des constructions du langage.

La métamodélisation bénéficie ainsi des avantages de la modélisation, comme d'intégrer intrinsèquement la séparation des préoccupations pour définir un système (ici, le langage). Cette propriété est de plus en plus usitée dans le développement de systèmes informatiques afin d'en maîtriser la complexité. Par exemple, dans la dernière version d'UML (*Unified Modeling Language*) [OMG07c, OMG07b], on trouve treize diagrammes pour définir les différents aspects d'un système. Bien entendu, ils ne sont pas tous obligatoires mais choisis en fonction du système à définir. Leur utilisation nécessite donc d'être organisée selon un procédé rigoureux. L'utilisation de ces diagrammes a donc été intégrée dans différentes méthodes, comme le RUP (*Ratinal Unified Process*) [BKK03], ou plus récemment le courant agile avec des méthodes comme XP (*eXtreme Programming*) [Bec02].

Dans ce contexte, les premiers travaux de la communauté se sont portés sur la définition du bon niveau d'abstraction des concepts nécessaires pour définir la syntaxe abstraite d'un DSML. Ces travaux ont donné lieu à différents langages, appelés *langages de métamodélisation*, décrits eux-mêmes par des modèles, appelés *méta-métamodèles*. Ils permettent ainsi d'unifier les concepts utilisés pour construire des métamodèles. Nous citerons par exemple le standard de l'OMG MOF [OMG06b], Ecore [BSE03] implanté dans le projet EMF [EMF07] d'Eclipse, KM3 [JB06] défini par le projet ATLAS de l'INRIA, Kermeta [MFJ05] défini par le projet TRISKELL de l'INRIA et Xcore [CSW08b] implanté dans XMF [XMF07].

Plus récemment, certains travaux proposent des langages pour définir des syn-

taxes concrètes à partir d'une syntaxe abstraite donnée. Celle-ci peut être textuelle ou graphique, et revient dans les deux cas à « décorer » les constructions de la syntaxe abstraite. Ces langages s'accompagnent généralement d'outils permettant à l'utilisateur du DSML de manipuler son langage. Il s'agit principalement de fournir des syntaxes concrètes textuelles (p. ex. TCS [JBK06] ou Sintaks [MFF+08]) ou graphiques (p. ex. Topcased [FGC+06], GMF [GMF07] ou Tiger [EEHT05b]) qui peuvent être exploitées par des éditeurs engendrés automatiquement ou génériques.

La seule volonté de définir des modèles contemplatifs utilisés uniquement pour communiquer avec des constructions plus abstraites sur le système, ne demande pas une définition plus complète du langage de modélisation. En effet, les syntaxes abstraite et concrète sont suffisantes pour manipuler textuellement ou graphiquement un langage et réfléchir de manière plus abstraite à la construction du système. Dans ce contexte, ce sont les noms choisis pour nommer les concepts qui sont support de la sémantique, et donc de la réflexion.

1.2 Les défis actuels

La complexité sans cesse croissante des systèmes actuels a déjà permis d'identifier certaines limites à la simple utilisation de modèles contemplatifs. En effet, une fois que l'on a réfléchi sur les modèles afin de concevoir un système, les méthodes actuelles visent à les implanter dans un langage de programmation compréhensible par un ordinateur. Il est alors indispensable de s'assurer que tous les outils et tous les acteurs du développement ont la même compréhension des modèles pour en fournir une implantation similaire. Il s'agit donc d'avoir une interprétation non ambiguë, commune et explicite des modèles. Une solution est d'obtenir automatiquement le code par transformation de modèle (c.-à-d. par compilation). Il reste néanmoins difficile de maintenir le logiciel tout au long de son cycle de vie directement sur les modèles. Cela nécessite d'avoir un lien de traçabilité fort entre les modèles et le code engendré et donc de formaliser la sémantique des constructions du langage de modélisation, en fonction de la traduction qui est définie. Cette étape permettrait ainsi de ne s'appuyer que sur les modèles pour décrire un système, et ainsi gagner en abstraction tout au long du développement.

Les premiers travaux pour préciser la sémantique des langages de modélisation ont fourni des langages permettant d'exprimer des contraintes sur la structure de la syntaxe abstraire de manière à restreindre l'ensemble des modèles valides. Ces contraintes viennent ainsi compléter celles capturées par le langage de métamodélisation lui-même (p. ex. les multiplicités, les contraintes d'ordre, etc.). OCL (*Object Constraint Language*) [OMG06c] est un exemple d'un tel langage. Il est proposé et utilisé par l'OMG pour préciser le sens des métamodèles qu'il défini. Ainsi, la plupart des concepts définis dans le métamodèle d'UML sont associés à un ensemble de contraintes OCL précisant leur sens. Ces langages permettent de capturer des contraintes invariantes, sur la structure des modèles. Ils ne permettent

cependant pas d'exprimer le comportement (à l'exécution) de ces concepts ni donc des modèles.

La prise en compte d'une sémantique d'exécution pour les langages de modélisation est une préoccupation récente. Elle permettrait ainsi de considérer les modèles comme un « artefact » terminal pour le développement d'un système, rendant ainsi transparent leurs compilations ou leurs interprétations. Ce gain d'abstraction est comparable et complémentaire à celui gagné grâce aux langages comme C ou Java vis-à-vis de l'assembleur.

Les premières solutions ont consisté à fournir des générateurs de code ou des machines virtuelles pour les langages de modélisation les plus courants (p. ex UML). Chacun de ces outils a fait l'objet d'un développement indépendant, définissant chacun la sémantique du langage considéré à l'aide d'un langage de programmation. Bien que ces propositions permettent toutes de rendre opérationnels les modèles (pour simuler, pour animer, pour exécuter, etc.), elles implantent chacune une sémantique d'exécution. Il est donc indispensable de pouvoir les valider par rapport à la sémantique implicite connue des experts du domaine. Cette sémantique du domaine doit être explicitée sous la forme d'une sémantique de référence directement exprimée sur les constructions du langage, de manière plus abstraite afin d'assurer une même compréhension du langage. C'est par exemple le cas pour les langages de programmation, où une sémantique de référence est définie (p. ex. à l'aide d'un système de transition) de manière à normaliser la sémantique du langage que les outils devront respecter.

Des travaux ont en partie répondu à cette problématique en proposant des langages d'action (chacun intégré dans un langage de métamodélisation) qui permettent de décrire le comportement d'un DSML directement sur les concepts de la syntaxe abstraite. Kermeta [MFJ05] et xOCL [CSW08b] en sont des exemples. Ces langages reposent sur des constructions impératives, proches des langages de programmation. Le gain d'abstraction est alors principalement gagné au niveau de la syntaxe abstraite sur laquelle s'appuie la définition de la sémantique. Leur utilisation reste encore peu fréquente (par exemple ils ne sont pas utilisés dans les standards) et ne s'intègre pas à une démarche méthodologique plus globale, couplée à la définition structurée de la syntaxe abstraite, permettant de capitaliser l'expérience de chaque DSML et les outils supports à l'exécution (simulateur, animateur, etc.). On retrouve ainsi au niveau métamodélisation le besoin d'expliciter le procédé de mise en œuvre, à l'instar de ce qui est fait pour UML et de ses différents diagrammes.

1.3 Objectifs de la thèse

Actuellement, de nombreux projets visent à élaborer des ateliers de développement dirigé par les modèles. Ces ateliers intègrent généralement différents DSML pour lesquels la validation et la vérification des modèles construits est une préoccupation majeure. C'est par exemple le cas dans l'atelier TOPCASED issu du projet

éponyme [1] au sein duquel s'intègrent les travaux de cette thèse.

TOPCASED est un projet de R&D du pôle de compétitivité mondial *Aerospace Valley* [2] dont le but est de fournir un atelier basé sur l'IDM pour le développement de systèmes logiciels et matériels embarqués. Les autorités de certification pour les domaines d'application de TOPCASED (aéronautique, espace, automobile, etc.) imposent des contraintes de qualification fortes pour lesquelles des approches formelles (analyse statique, vérification de modèle, preuve, etc.) sont envisagées.

L'outillage de TOPCASED a pour but de simplifier la définition de nouveaux DSML ou langages de modélisation en fournissant des technologies du *métaniveau* telles que des générateurs d'éditeurs syntaxiques (textuels ou graphiques), des outils de validation statique ainsi que des outils d'exécution, de vérification et de validation dynamique.

Pour atteindre ce dernier point (vérification et validation, *V & V*), il est nécessaire d'expliciter la sémantique d'exécution et de définir les outils de simulation et de vérification formelle pour chaque DSML à travers une approche outillée réduisant l'effort à fournir.

Au regard des travaux existants dans l'IDM et de l'expérience acquise avec les langages de programmation, nous proposons dans cette thèse une taxonomie précise des techniques permettant d'exprimer une sémantique d'exécution pour un DSML.

Nous replaçons ensuite ces différentes techniques au sein d'une démarche complète permettant de décrire un DSML et les outils nécessaires à l'exécution, la vérification et la validation des modèles. Cette démarche offre une architecture rigoureuse et générique de la syntaxe abstraite du DSML de manière à capturer les informations nécessaires à l'exécution d'un modèle et à définir les propriétés temporelles qui doivent être vérifiées.

Par la suite, nous nous appuyons sur cette architecture générique pour expliciter la sémantique de référence (issue de l'expérience des experts du domaine) et étendre la syntaxe abstraite par l'implantation d'une sémantique d'exécution. Nous étudions plus particulièrement les moyens :

- d'exprimer et de valider la définition d'une traduction vers un domaine formel pour réutiliser les outils de *model-checking* et permettre ainsi la vérification formelle et automatique des propriétés exprimées.
- de compléter facilement la syntaxe abstraite par le comportement de chaque concept ; et profiter d'outils génériques pour pouvoir simuler les modèles construits.

Enfin, il est indispensable de valider les différentes sémantiques implantées soit pour la vérification soit pour la simulation vis-à-vis de la sémantique de référence. Aussi, nous proposons un cadre formel pour la métamodélisation, transparent pour l'utilisateur, permettant d'exprimer la sémantique de référence qui servira de base

1. *Toolkit in OPen source for Critical Application & SystEms Development*, cf. http://www.topcased.org

2. Pôle AESE (Aéronautique, Espace, Systèmes Embarqués), cf. http://www.aerospace-valley.com

à cette validation.

L'approche que nous proposons dans cette thèse a été définie de manière générique pour s'appliquer à tout DSML. En plus de la prise en compte de certains DSML de l'atelier TOPCASED, nous proposons d'illustrer sa généricité à travers un autre champ d'application de l'IDM : l'ingénierie des procédés de développement. Au sein de celle-ci, une application est reconnue comme un produit manufacturé complexe dont la réalisation doit s'intégrer dans une démarche méthodologique : le *procédé de développement*. Pour améliorer la qualité du produit développé, il est important de maîtriser son procédé de développement dont la problématique s'apparente à celle du logiciel lui-même [Ost87, Est05]. En effet, au même titre que les logiciels, les procédés de développement deviennent de plus en plus complexes (offshore, sous-traitance, dispersion géographique des équipes, etc.). Dans ce contexte, l'IDM permet d'unifier la définition aussi bien d'un logiciel que du procédé qui a permis de le réaliser. De nombreux travaux proposent pour cela des langages de modélisation pour les procédés de développement. A ce titre, nous proposons dans cette thèse de poursuivre les travaux réalisés dans ce domaine afin de permettre de valider, vérifier et administrer les modèles de procédé. Nous étudions plus particulièrement SPEM (*Software & Systems Process Engineering Metamodel*) [OMG07a], le standard de l'OMG, et en proposons une extension outillée permettant d'éditer, de vérifier et d'animer les modèles de procédé.

Cette thèse participe donc à l'amélioration des techniques pour la définition des langages et outils pour le développement de systèmes. Elle applique ensuite ses premiers résultats à la maîtrise et la validation des procédé guidant la réalisation des systèmes et don l'utilisation de ces mêmes langages et outils. Cette thèse contribue ainsi de manière originale à la maîtrise de la complexité des systèmes informatiques actuels.

1.4 Contexte des travaux

Cette thèse s'inscrit dans la volonté d'une prise en compte plus forte de l'ingénierie de la sémantique dans le contexte de l'IDM. Elle se place dans un champ de recherche encore peu exploré mais dont l'intérêt 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. Par ailleurs, on ne compte plus les projets de R&D sur l'utilisation des modèles pour le développement de systèmes complexes et/ou critiques. Dans les deux cas, la vérification et la validation sont des préoccupations centrales.

D'autre part, cette thèse initie les travaux menés autour de l'ingénierie dirigée par les modèles dans l'équipe ACADIE où ont été réalisée les travaux présentés dans cette thèse. Elle s'appuie pour cela sur l'expérience acquise par les différents membres sur la sémantique des langages de programmation de manière à explorer les axes pertinents pour exprimer la sémantique des DSML. Ces travaux profitent également de l'expérience de Xavier Crégut sur l'ingénierie des procédés de dé-

veloppement [Cré97, CC97] afin de les appliquer tout le long à la définition d'un langage de modélisation de procédé et des outils permettant de construire, de simuler et de vérifier formellement les modèles.

Enfin, cette thèse s'inscrit dans le cadre du projet TOPCASED qui est actuellement un des projets majeurs de l'équipe. La définition d'une sémantique d'exécution pour un DSML est un besoin transversal au projet qui nous a amenés à participer à plusieurs lots qui le composent :

- Lot n°1 (procédé global) où nous avons participé à l'étude des langages de modélisation de procédé et à la définition d'un environnement de modélisation, de simulation et de vérification de modèle de procédé.
- Lot n°2 (moyens de développement des modèles) où nous avons appliqué l'approche proposée dans cette thèse de manière à pouvoir simuler les modèles construits à partir de certains DSML de l'atelier.
- Lot n°3 (vérification de modèle) où nous avons intégré nos travaux pour définir les traductions vers des outils de vérification formelle.
- Lot n°5 (transformation de modèle) où nous avons participé aux différentes études et évaluations des techniques de transformations de modèle que nous avons exploitées pour la définition des traductions vers des outils de vérification formelle.

1.5 Contenu du mémoire

Cette thèse comporte deux parties principales. La première introduit les principes généraux de l'IDM et de l'ingénierie des procédés, puis présente les outils pour exécuter des modèles. Ces expérimentations sont menées à travers un langage simplifié de modélisation de procédé que nous appelons SIMPLEPDL (*Simple Process Description Language*). Cette partie est composée de trois chapitres :

- Le chapitre 2 introduit la **métamodélisation** et l'illustre par la définition du DSML SIMPLEPDL.
- Le chapitre 3 fait un **état de l'art sur l'ingénierie des procédés**, et présente plus particulièrement le standard SPEM. Il permet de décrire informellement le sens des concepts principaux pour la définition des procédés. Il met également en évidence les limites actuelles du standard SPEM dans la description d'une sémantique d'exécution.
- Le chapitre 4 présente une **taxonomie des techniques pour décrire la sémantique d'exécution** d'un DSML et permettre ainsi d'exécuter les modèles construits. Nous illustrons ces différentes techniques en expérimentant chacune d'elles pour l'exécution de modèles SIMPLEPDL.

La deuxième partie comprend cinq chapitres qui présentent nos contributions pour la simulation et la vérification de modèle ainsi que leur transfert au sein de l'atelier TOPCASED.

- Le chapitre 5 décrit une **approche pour capturer et structurer au sein de la syntaxe abstraite d'un DSML les préoccupations** nécessaires à l'exécu-

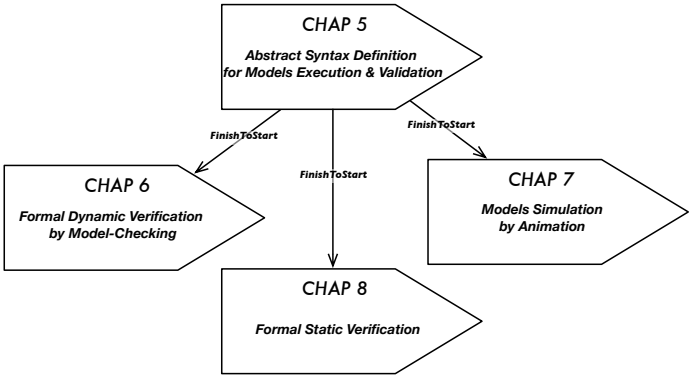


FIGURE 1.1: Principales étapes pour la vérification et la simulation de modèle

tion et la vérification des modèles. Ces travaux sont illustrés par l'extension de SPEM que nous proposons, XSPEM. Elle permet de capturer dans les modèles de procédé les informations liées à l'exécution d'un procédé.

- Le chapitre 6 complète l'**approche pour vérifier formellement les modèles** à travers un langage formel tel que les réseaux de Petri. Nous proposons également une implantation possible pour XSPEM.
- Le chapitre 7 introduit un **cadre générique pour la simulation de modèle**. Après une étude des besoins en terme de simulation, nous présentons une architecture générique d'un simulateur de modèle qui s'appuie sur la structure de la syntaxe abstraite du DSML. Nous définissons ensuite de manière générique les différents composants qui ne s'appuient pas sur le domaine considéré et étudions les moyens de générer les composants propres au domaine. Nous illustrons cette architecture par un premier prototype de simulateur de modèle XSPEM.
- Le chapitre 8 présente un **codage formel pour décrire les DSML et leurs modèles**. Celui ci est défini de manière à pouvoir être facilement réutilisable pour exprimer directement sur le DSML sa sémantique de référence et ainsi permettre de vérifier formellement les modèles, aussi bien statiquement que dynamiquement.
- Le chapitre 9 présente le **transfert de ces travaux au sein de l'atelier TOPCASED** et démontre l'aspect générique de l'approche proposée au sein de cette thèse en prenant en compte de nouveaux DSML.

Les apports de cette thèse sont décrits synthétiquement dans le modèle de procédé décrit dans la figure 1.1. Les dépendances entre activité correspondent ici aux chemins de lecture possibles des chapitres correspondants.

Enfin, nous concluons en faisant un bilan des contributions apportées dans cette thèse et en soulignant leurs faiblesses. Nous détaillons également les perspectives offertes par ces travaux et les voies de recherche que nous suivons actuellement.

Première partie

Vers une opérationalisation des modèles dans l'IDM

Chapitre 2

État de l'art sur l'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (IDM), ou *Model Driven Engineering* (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles. Un modèle est une abstraction, une simplification d'un système qui est suffisante pour comprendre le système modélisé et répondre aux questions que l'on se pose sur lui. Un système peut être décrit par différents modèles liés les uns aux autres. L'idée phare est d'utiliser autant de langages de modélisation différents (*Domain Specific Modeling Languages* – DSML) que les aspects chronologiques ou technologiques du développement du système le nécessitent. La définition de ces DSML, appelée *métamodélisation*, est donc une problématique clé de cette nouvelle ingénierie. Par ailleurs, afin de rendre opérationnels les modèles (pour la génération de code, de documentation et de test, la validation, la vérification, l'exécution, etc.), une autre problématique clé est celle de la *transformation de modèle*.

Nous proposons dans ce chapitre une présentation des principes clés de cette nouvelle ingénierie. Nous introduisons dans un premier temps la notion de modèle, les travaux de normalisation de l'OMG, et les principes de généralisation offerts à travers les DSML (section 2.1). Nous détaillons ensuite les deux axes principaux de l'IDM. La métamodélisation d'abord, dont le but est d'assurer une définition correcte des DSML (section 2.2). Nous illustrons cette partie par la définition de SIMPLEPDL, un langage simple de description de procédé de développement. Nous présentons ensuite les principes de la transformation de modèle et les outils actuellement disponibles (section 2.3). Nous concluons enfin par une discussion sur les limites actuelles de l'IDM (section 2.4).

> Cet état de l'art sur les techniques de métamodélisation et de transformation de modèle a donné lieu à la création d'un module d'enseigne-

ment disponible librement sur Internet [1].

2.1 Les modèles au coeur du développement de système

> « Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O » [Min68].

2.1.1 Les principes généraux de l'IDM

Suite à l'approche objet des années 80 et de son principe du « tout est objet », l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM) et le principe du « tout est modèle ». Cette nouvelle approche peut être considérée à la fois en *continuité* et en *rupture* avec les précédents travaux [Béz04b, Béz05]. Tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des systèmes informatiques sous la forme d'objets communicant entre eux, il s'est posé la question de les classifier en fonction de leurs différentes origines (objets métiers, techniques, etc.). L'IDM vise donc, de manière plus radicale que pouvaient l'être les approches des *patterns* [GHJ95] et des *aspects* [KLM+97], à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet.

Alors que l'approche objet est fondée sur deux relations essentielles, « InstanceDe » et « HériteDe », l'IDM est basée sur un autre jeu de concepts et de relations. Le concept central de l'IDM est la notion de *modèle* pour laquelle il n'existe pas à ce jour de définition universelle. Néanmoins, de nombreux travaux s'accordent à un relatif consensus d'une certaine compréhension. A partir des travaux de l'OMG [2], de Bézivin *et al.* [BG01] et de Seidewitz [Sei03], nous considérerons dans la suite de cette thèse la définition suivante d'un modèle.

Définition (*Modèle*) Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé.

On déduit de cette définition la première relation majeure de l'IDM, entre le modèle et le système qu'il représente, appelée *représentationDe* dans [AK03, Béz04a, Sei03], et nommée μ sur la figure 2.1.

Notons que même si la relation précédente a fait l'objet de nombreuses réflexions, il reste toutefois difficile de répondre à la question « qu'est ce qu'un bon modèle ? » et donc de formaliser précisément la relation μ. Néanmoins un modèle

1. L'ensemble du module d'enseignement dispensé à l'INSAT, à l'ENSEEIHT et à l'université de Yaounde I est disponible à l'adresse http://combemale.perso.enseeiht.fr/teaching/mde.

2. *The Object Management Group (OMG)*, cf. http://www.omg.org/.

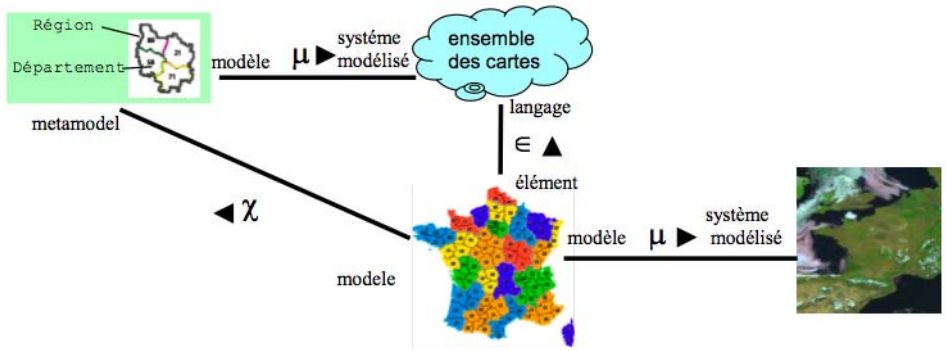


FIGURE 2.1: Relations entre système, modèle, métamodèle et langage [FEB06]

doit, par définition, être une abstraction pertinente du système qu'il modélise, c.-à-d. qu'il doit être suffisant et nécessaire pour permettre de répondre à certaines questions en lieu et place du système qu'il représente, exactement de la même façon que le système aurait répondu lui-même. Ce principe, dit de *substituabilité*, assure que le modèle peut se substituer au système pour permettre d'analyser de manière plus abstraite certaines de ses propriétés [Min68].

Définition (*Principe de substituabilité*) Un modèle doit être suffisant et nécessaire pour permettre de répondre à certaines questions en lieu et place du système qu'il est censé représenté, exactement de la même façon que le système aurait répondu lui-même.

Sur la figure 2.1, nous reprenons l'exemple utilisé dans [FEB06] qui s'appuie sur la cartographie pour illustrer l'IDM. Dans cet exemple, une carte est un modèle (une représentation) de la réalité, avec une intention particulière (carte routière, administrative, des reliefs, etc.).

La notion de modèle dans l'IDM fait explicitement référence à la notion de langage bien défini. En effet, pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être clairement défini. De manière naturelle, la définition d'un langage de modélisation a pris la forme d'un modèle, appelé *métamodèle*.

Définition (*Métamodèle*) Un métamodèle est un modèle qui définit le langage d'expression d'un modèle [OMG06b], c.-à-d. le langage de modélisation.

La notion de métamodèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée *conformeA* et nommée χ sur la figure 2.1.

En cartographie (cf. figure 2.1), il est effectivement indispensable d'associer à chaque carte la description du « langage » utilisé pour réaliser cette carte. Ceci

se fait notamment sous la forme d'une légende explicite. La carte doit, pour être utilisable, être conforme à cette légende. Plusieurs cartes peuvent être conformes à une même légende. La légende est alors considérée comme un modèle représentant cet ensemble de cartes (μ) et à laquelle chacune d'entre elles doit se conformer (χ).

Ces deux relations permettent ainsi de bien distinguer le langage qui joue le rôle de système, du (ou des) métamodèle(s) qui jouent le rôle de modèle(s) de ce langage.

C'est sur ces principes de base que s'appuie l'OMG pour définir l'ensemble de ses standards, en particulier UML (*Unified Modeling Language*) [OMG07b, OMG07c] dont le succès industriel est unanimement reconnu.

2.1.2 L'approche MDA

Le consensus sur UML fut décisif dans cette transition vers des techniques de production basées sur les modèles. Après l'acceptation du concept clé de métamodèle comme langage de description de modèle, de nombreux métamodèles ont émergés afin d'apporter chacun leurs spécificités dans un domaine particulier (développement logiciel, entrepôt de données, procédé de développement, etc.). Devant le danger de voir émerger indépendamment et de manière incompatible cette grande variété de métamodèles, il y avait un besoin urgent de donner un cadre général pour leur description. La réponse logique fut donc d'offrir un langage de définition de métamodèles qui prit lui-même la forme d'un modèle : ce fut le *métamétamodèle* MOF (*Meta-Object Facility*) [OMG06b]. En tant que modèle, il doit également être défini à partir d'un langage de modélisation. Pour limiter le nombre de niveaux d'abstraction, il doit alors avoir la propriété de *métacircularité*, c.-à-d. la capacité de se décrire lui-même.

Définition (*Métamétamodèle*) Un métamétamodèle est un modèle qui décrit un langage de métamodélisation, c.-à-d. les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même.

C'est sur ces principes que se base l'organisation de la modélisation de l'OMG généralement décrite sous une forme pyramidale (cf. figure 2.2). Le monde réel est représenté à la base de la pyramide (niveau $M0$). Les modèles représentant cette réalité constituent le niveau $M1$. Les métamodèles permettant la définition de ces modèles (p. ex. UML) constituent le niveau $M2$. Enfin, le métamétamodèle, unique et métacirculaire, est représenté au sommet de la pyramide (niveau $M3$).

L'approche consistant à considérer une hiérarchie de métamodèles n'est pas propre à l'OMG, ni même à l'IDM, puisqu'elle est utilisée depuis longtemps dans de nombreux domaines de l'informatique. Chaque hiérarchie définit un *espace technique* [KBA02, BJRV05, BK05]. Nous distinguons par exemple le *modelware* (espace technique des modèles), le *grammarware* (espace technique des gram-

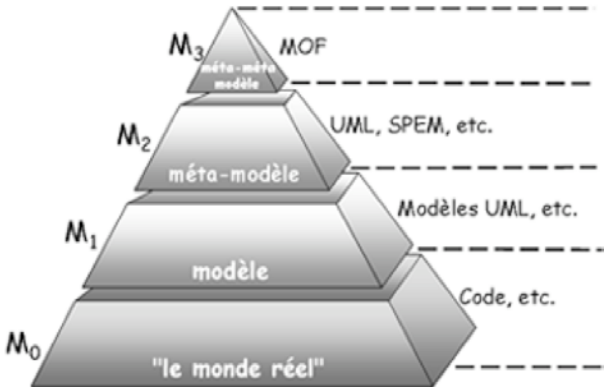


FIGURE 2.2: Pyramide de modélisation de l'OMG [Béz03]

maires définies par les langages tels que BNF[3] ou EBNF[4]), le *BDware* (espace technique des bases de données), etc.

Définition (*Espace technique*) Un espace technique est l'ensemble des outils et techniques issus d'une pyramide de métamodèles dont le sommet est occupé par une famille de (méta)métamodèles similaires [FEB06].

L'OMG a défini le MDA (*Model Driven Architecture*) en 2000 [Sol00] pour promulguer de bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles. En 2003, les membres ont adopté la dernière version de la spécification [MM03] donnant une définition détaillée de l'architecture. Cette approche vise à mettre en valeur les qualités intrinsèques des modèles, telles que pérennité, productivité et prise en compte des plateformes d'exécution. Le MDA inclut pour cela la définition de plusieurs standards, notamment UML, MOF et XMI[5].

Le principe clé et initial du MDA consiste à s'appuyer sur le standard UML pour décrire séparément des modèles pour les différentes phases du cycle de développement d'une application. Plus précisément, le MDA préconise l'élaboration de modèles (cf. figure 2.3) :

- d'exigence (*Computation Independent Model* – CIM) dans lesquels aucune considération informatique n'apparait,
- d'analyse et de conception (*Platform Independent Model* – PIM),
- de code (*Platform Specific Model* – PSM).

L'objectif majeur du MDA est l'élaboration de modèles pérennes (PIM), indépendants des détails techniques des plate-formes d'exécution (J2EE, .Net, PHP, etc.), afin de permettre la génération automatique de la totalité des modèles de code (PSM) et d'obtenir un gain significatif de productivité.

3. *Backus-Naur form*
4. *Extended BNF*
5. XMI, *XML Metadata Interchange*, est un format d'échange basé sur XML pour les modèles exprimés à partir d'un métamodèle MOF.

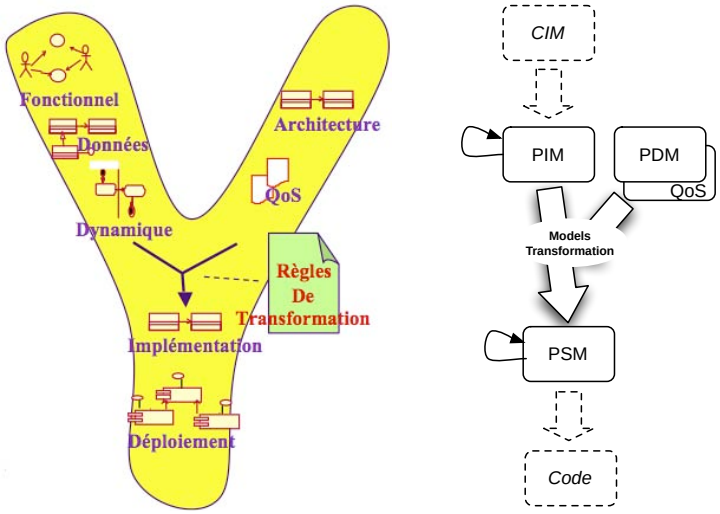


FIGURE 2.3: MDA : Un processus en Y dirigé par les modèles

Le passage de PIM à PSM fait intervenir des mécanismes de transformation de modèle (cf. section 2.3) et un modèle de description de la plateforme (*Platform Description Model* – PDM). Cette démarche s'organise donc selon un cycle de développement « en Y » propre au MDD (*Model Driven Development*) (cf. figure 2.3).

Le MDA a fait l'objet d'un grand intérêt dans la littérature spécialisée. Nous citons entre autre les ouvrages de X. Blanc [Bla05] et de A. Kleppe [KWB03] qui ont inspirés cette section.

2.1.3 Les langages dédiés de modélisation

De la même façon que l'arrivée de la programmation par objet n'a pas invalidé les apports de la programmation structurée, le développement dirigé par les modèles ne contredit pas les apports de la technologie objet. Il est donc important de ne pas considérer ces solutions comme antagonistes mais comme complémentaires.

Toutefois un point de divergence entre ces deux approches concerne l'intégration de paradigmes. Initialement, la technologie objet se voulait aussi une technologie d'intégration car il était théoriquement possible de représenter de façon uniforme les processus, les règles, les fonctions, etc. par des objets. Aujourd'hui, on revient à une vision moins hégémonique où les différents paradigmes de programmation coexistent sans donner plus d'importance à l'un ou à l'autre [Béz04b].

Un point important est alors de séparer clairement les approches IDM du formalisme UML, et de l'utilisation qui en est faite dans le MDA. En effet, non seulement la portée de l'IDM est plus large que celle d'UML mais la vision de l'IDM

est aussi très différente de celle d'UML, parfois même en contradiction. UML est un standard assez monolithique obtenu par consensus *a maxima*, dont on doit réduire ou étendre la portée à l'aide de mécanismes comme les profils [OMG07c, §18]. Ces mécanismes n'ont pas tous la précision souhaitable et mènent parfois à des contorsions dangereuses pour « rester » dans le monde UML.

Au contraire, l'IDM favorise la définition de langages de modélisation dédiés à un domaine particulier (*Domain Specific Modeling Languages* – DSML) offrant ainsi aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise. Ces langages sont généralement de petite taille et doivent être facilement manipulables, transformables, combinables, etc.

Selon ces principes, la définition d'un système complexe fait généralement appel à l'utilisation de plusieurs DSML ayant des relations entre eux, restreignant ainsi l'ensemble des combinaisons valides des modèles conformes à ces différents métamodèles (c.-à-d. construits à l'aide de ces différents DSML). Il est ainsi récemment apparu la nécessité de représenter ces différents DSML et les relations entre eux. Une réponse logique à ce besoin a été de proposer des modèles dont les éléments de base sont d'une part les différents DSML et d'autre part les liens exprimant leurs dépendances. Ce type de modèle est appelé *mégamodèle* [BJV04, Fav04] et est déjà supporté dans l'outil de mégamodélisation AM3 [AM307].

Définition (*Mégamodèle*) Un mégamodèle est un modèle dont les éléments représentent des (méta)modèles ou d'autres *artefacts* (comme des DSML, des outils, des services, etc.).

Notons par ailleurs que le concept de DSML a donné lieu à la création de nombreux langages qu'il est maintenant urgent de maîtriser (documentation, hiérarchie, etc.) et de pouvoir manipuler facilement (combiner, transformer, etc.). Pour cela, certains *zoos*[6] proposent un recensement, une documentation et une classification de ces DSML et offrent certaines manipulations, comme de pouvoir les transformer vers différents espaces techniques.

2.2 La métamodélisation

Comme nous l'avons vu dans la section précédente, l'IDM préconise l'utilisation de petits langages de modélisation, dédiés chacun à un domaine particulier. Ainsi, la première problématique clé de l'IDM est la maîtrise de la définition de ces DSML, dont le nombre ne cesse de croître avec la multiplication des domaines d'application de l'informatique. La métamodélisation, activité correspondant à définir un DSML, doit donc être étudiée et maîtrisée. Pour cela, les premiers travaux ont consisté à définir précisément les différentes composantes d'un langage de modélisation et à offrir les outils permettant de les décrire.

6. Un zoo est (une vue d')un mégamodèle où tous les métamodèles qui le compose ont le même métamétamodèle [VBBJ06]. Nous citons par exemple les zoos http://www.eclipse.org/gmt/am3/zoos/ et http://www.zooomm.org/.

Définition (*Métamodélisation*) La métamodélisation est l'activité consistant à définir le métamodèle d'un langage de modélisation. Elle vise donc à bien modéliser un langage, qui joue alors le rôle de système à modéliser.

Nous détaillons maintenant les principes de la métamodélisation (section 2.2.1) et présentons ensuite les différents outils et techniques pour la spécification d'un DSML (section 2.2.2). Ceci est illustré par la définition de SIMPLEPDL, un petit langage de modélisation dédié au domaine des procédés de développement, c.-à-d. un langage permettant de définir des modèles de procédé de développement.

2.2.1 Qu'est-ce qu'un langage ?

Que ce soit en linguistique (langage naturel) ou en informatique (langage de programmation ou de modélisation), il est depuis longtemps établi qu'un langage est caractérisé par sa *syntaxe* et sa *sémantique*. La syntaxe décrit les différentes constructions du langage et les règles d'agencement de ces constructions (également appelées *context condition* [HR04]). La sémantique désigne le lien entre un signifiant (un programme, un modèle, etc.), et un signifié (p. ex. un objet mathématique) afin de donner un sens à chacune des constructions du langage. Il y a donc entre la sémantique et la syntaxe le même rapport qu'entre le fond et la forme.

Définition (*Langage*) Un langage (L) est défini selon le tuple $\{S, Sem\}$ où S est sa syntaxe et Sem sa sémantique.

Cette définition est très générale et assez abstraite pour caractériser l'ensemble des langages, quel que soit le domaine. De manière plus précise, dans le contexte de l'informatique, on distingue généralement la syntaxe concrète (CS sur la figure 2.4), manipulée par l'utilisateur du langage, de la syntaxe abstraite (AS sur la figure 2.4) qui est la représentation interne (d'un programme ou d'un modèle) manipulée par l'ordinateur [ASU86]. Dans les langages de programmation, la représentation interne (l'arbre abstrait) est dérivée de la syntaxe concrète. Ainsi, la syntaxe d'un langage de programmation est définie par les syntaxes concrète et abstraite et par un lien, dit de *dérivation* ou d'*abstraction*, entre la syntaxe concrète et la syntaxe abstraite (M_{ca} sur la figure 2.4). Ce lien d'abstraction permet d'enlever tout le « sucre » syntaxique inutile à l'analyse du programme.

D'autre part, on distingue également le *domaine sémantique* (SD sur la figure 2.4) qui représente l'ensemble des états atteignables (c.-à-d. les états possibles du système). La sémantique d'un langage de programmation est alors donnée en liant les constructions de la syntaxe concrète avec l'état auquel elles correspondent dans le domaine sémantique (M_{as} sur la figure 2.4).

Définition (*Langage de programmation*) Un langage de programmation (L_p) est défini selon le tuple $\{AS, CS, M_{ca}, SD, M_{cs}\}$ où AS est la syntaxe abstraite, CS est la syntaxe concrète, M_{ca} est le *mapping* de la syntaxe concrète vers sa représentation abstraite, SD est le domaine sémantique et M_{cs} est le *mapping* de la syntaxe concrète vers le domaine sémantique.

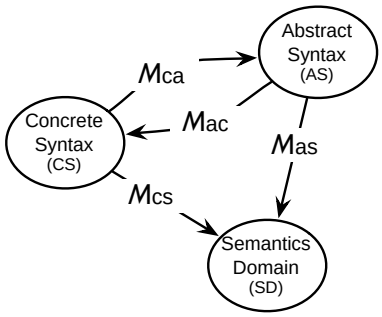


FIGURE 2.4: Composantes d'un langage

Dans le contexte de l'IDM, la syntaxe abstraite est placée au cœur de la description d'un langage de modélisation. Elle est donc généralement décrite en premier et sert de base pour définir la syntaxe concrète. La définition de la syntaxe concrète consiste alors à définir des décorations (textuelles ou graphiques) et à définir un lien entre les constructions de la syntaxe abstraite et leurs décorations de la syntaxe concrète (M_{ac} sur la figure 2.4). Ce changement de sens du lien par rapport aux langages de programmation permet d'envisager de définir plusieurs syntaxes concrètes (M^*_{ac}) pour une même syntaxe abstraite et donc d'avoir plusieurs représentations d'un même modèle. Le langage peut alors être manipulé avec différents formalismes mais avec les mêmes constructions et la même représentation abstraite. D'autre part, dans le cadre des langages de modélisation, la sémantique est exprimée à partir des constructions de la syntaxe abstraite par un lien vers un domaine sémantique (M_{as} sur la figure 2.4). Nous considérerons dans cette thèse qu'un DSML ne peut avoir qu'une seule sémantique (et donc un seul *mapping* vers un domaine sémantique). La définition d'un autre *mapping* engendrera la définition d'un nouveau DSML. Toutefois, un même modèle peut être simultanément ou successivement conforme à plusieurs DSML.

Définition (*Langage de modélisation*) Un langage de modélisation (L_m) est défini selon le tuple $\{AS, CS^*, M^*_{ac}, SD, M_{as}\}$ où AS est la syntaxe abstraite, CS^* est la (les) syntaxe(s) concrète(s), M^*_{ac} est l'ensemble des *mappings* de la syntaxe abstraite vers la (les) syntaxe(s) concrète(s), SD est le domaine sémantique et M_{as} est le *mapping* de la syntaxe abstraite vers le domaine sémantique.

2.2.2 Outils et techniques pour la spécification d'un DSML

Nous introduisons dans cette partie les techniques, standards, et outils actuellement disponibles pour décrire les différentes parties d'un DSML présentées dans la partie précédente.

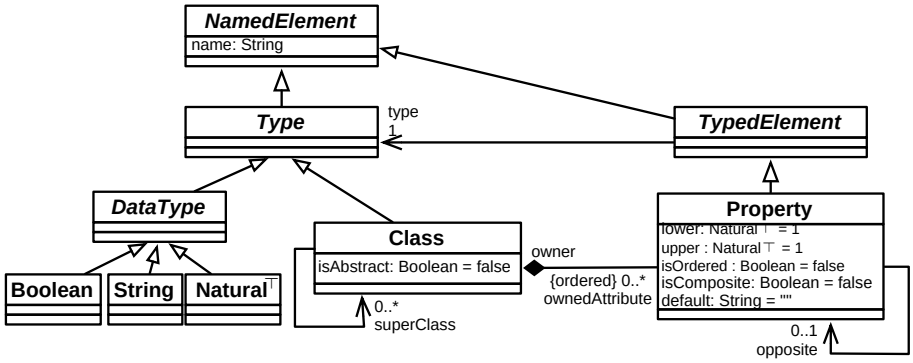


FIGURE 2.5: Concepts principaux de métamodélisation (EMOF 2.0)

Syntaxe abstraite

La syntaxe abstraite (AS) d'un langage de modélisation exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations. Les langages de métamodélisation tels que le standard MOF de l'OMG [OMG06b], offrent les concepts et les relations élémentaires qui permettent de décrire un métamodèle représentant la syntaxe abstraite d'un langage de modélisation. Pour définir cette syntaxe, nous disposons à ce jour de nombreux environnements et langages de métamodélisation : Eclipse-EMF/Ecore [BSE03], GME/MetaGME [LMB+01], AMMA/KM3 [JB06, ATL05], XMF-Mosaic/Xcore [CESW04] ou Kermeta [MFJ05]. Tous ces langages reposent toutefois sur les mêmes constructions élémentaires (cf. figure 2.5). S'inspirant de l'approche orientée objet, les langages de métamodélisation objet offre le concept de classe (*Class*) pour définir les concepts d'un DSML. Une classe est composée de propriétés (*Property*) qui la caractérisent. Une propriété est appelée *référence* lorsqu'elle est typée (*TypedElement*) par une autre classe, et *attribut* lorsqu'elle est typée par un type de donnée (p. ex. booléen, chaîne de caractère et entier).

Nous illustrons ces concepts par la définition de SIMPLEPDL, un langage simple de description de procédé, que nous utiliserons dans la suite de cette thèse pour illustrer les différentes techniques abordées. Il s'inspire du standard SPEM (*Software & Systems Process Engineering Metamodel*) [OMG05a] proposé par l'OMG mais aussi du métamodèle UMA (*Unified Method Architecture*) utilisé par le *plug-in* Eclipse EPF[7] (*Eclipse Process Framework*), dédié à la modélisation de procédé. Il est volontairement simplifié pour ne pas compliquer inutilement les expérimentations.

Pour définir la syntaxe abstraite de *SimplePDL*, nous avons utilisé l'éditeur graphique du projet TOPCASED permettant la description de métamodèles à l'aide

7. http://www.eclipse.org/epf/

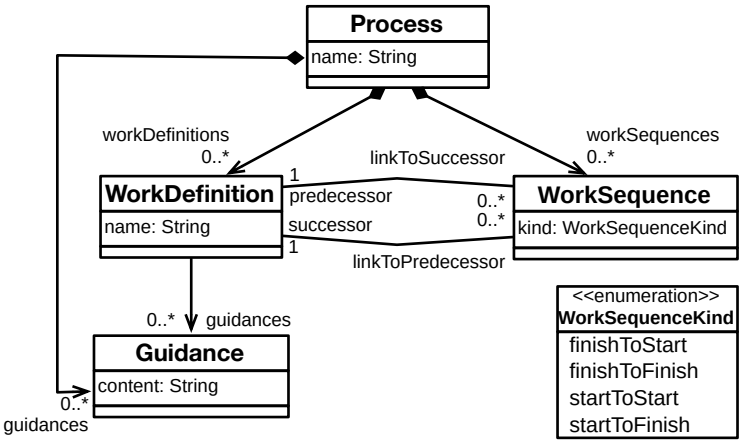


FIGURE 2.6: Syntaxe abstraite de SIMPLEPDL

du langage de métamodélisation Ecore. Le métamodèle SIMPLEPDL est donné figure 2.6. Il définit le concept de processus (*Process*) composé d'un ensemble d'activités (*WorkDefinition*) représentant les différentes tâches à réaliser durant le développement. Une activité peut dépendre d'une autre (*WorkSequence*). Une contrainte d'ordonnancement sur le démarrage ou la fin de la seconde activité est précisée (attribut *linkType*) grâce à l'énumération *WorkSequenceType*. Par exemple, deux activités A_1 et A_2 reliées par une relation de précédence de type *finishToStart* signifie que A_2 ne pourra commencer que quand A_1 sera terminée. Enfin, des annotations textuelles (*Guidance*) peuvent être associées aux activités pour donner plus de détails sur leurs réalisations.

La figure 2.7 donne un exemple de modèle de processus composé de quatre activités. Le développement ne peut commencer que quand la conception est terminée. La rédaction de la documentation ou des tests peut commencer dès que la conception est commencée (*startToStart*) mais la documentation ne peut être terminée que si la conception est terminée (*finishToFinish*) et les tests si le développement est terminé.

La représentation graphique offerte par les langages de métamodélisation ne permet pas de capturer formellement l'ensemble des propriétés du langage (c.-à-d. les *context conditions*). Dans le domaine des langages de programmation, la sémantique axiomatique est basée sur des logiques mathématiques et exprime une méthode de preuve pour certaines propriétés des constructions d'un langage [Cou90]. Celle-ci peut être très générale (e.g. triplet de Hoare) ou restreinte à la garantie de la cohérence de ces constructions (p. ex. le typage). Dans le cadre d'un langage de modélisation, cette seconde utilisation est exprimée par le biais de règles de bonne formation (*Well-Formed Rules* – WFR), au niveau du métamodèle. Ces règles devront être respectées par les modèles conformes à ce métamodèle.

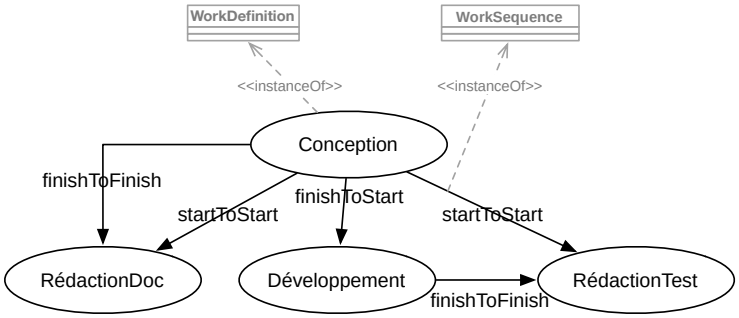


FIGURE 2.7: Exemple de modèle SIMPLEPDL

Pour exprimer ces règles, l'OMG préconise d'utiliser OCL (*Object Constraint Language*) [OMG06c, WK03]. Appliqué au niveau du métamodèle, il permet d'ajouter des propriétés, principalement structurelles, qui n'ont pas pu être capturées par les concepts fournis par le métamétamodèle. Il s'agit donc d'un moyen de préciser la sémantique du métamodèle en limitant les modèles conformes.

Par exemple, pour *SimplePDL*, on peut utiliser la contrainte OCL suivante pour imposer l'unicité du nom des activités dans un processus.

context Process **inv** :
self . activities −>**forAll**(a1, a2 : Activity |
a1 <> a2 **implies** a1.**name** <> a2.**name**)

Pour vérifier qu'un modèle respecte ces contraintes, on peut utiliser des vérificateurs OCL tels que Use [RG00], OSLO [8], TOPCASED, etc.

Syntaxe concrète

Les syntaxes concrètes (CS) d'un langage fournissent à l'utilisateur un ou plusieurs formalismes, graphiques et/ou textuels, pour manipuler les concepts de la syntaxe abstraite et ainsi en créer des « instances ». Le modèle ainsi obtenu sera conforme à la structure définie par la syntaxe abstraite. La définition d'une syntaxe concrète consiste à définir un des *mappings* de M^*_{ac} (cf. figure 2.4), $M_{ac} : AS \leftrightarrow CS$, et permet ainsi « d'annoter » chaque construction du langage de modélisation définie dans la syntaxe abstraite par une (ou plusieurs) décoration(s) de la syntaxe concrète et pouvant être manipulée(s) par l'utilisateur du langage.

La définition du modèle d'une syntaxe concrète est à ce jour bien maîtrisée et outillée. Il existe en effet de nombreux projets qui s'y consacrent, principalement basés sur EMF (*Eclipse Modeling Framework*) [EMF07] : GMF (*Generic Modeling Framework*) [GMF07], TOPCASED [Top07, FGC+06], Merlin Generator [Mer07], GEMS [GEM07], TIGER [Tig07, EEHT05a], etc. Si ces derniers sont

8. Open Source Library for OCL, http://oslo-project.berlios.de.

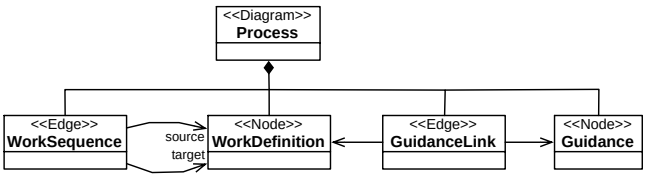


FIGURE 2.8: Modèle de configuration de la syntaxe concrète de SIMPLEPDL

principalement graphiques, des projets récents permettent de définir des modèles de syntaxe concrète textuelle. Nous citons par exemple les travaux des équipes INRIA Triskell (Sintaks [MFF+08, Sin07]) et ATLAS (TCS [JBK06, TCS07]) qui proposent des générateurs automatiques d'éditeurs pour des syntaxes concrètes textuelles. Ces approches génératives, en plus de leurs qualités de généricité, permettent de normaliser la construction des syntaxes concrètes.

Nous illustrons l'utilisation du générateur d'éditeur graphique de TOPCASED afin de définir la syntaxe concrète de SIMPLEPDL. Cet outil permet, pour un modèle Ecore donné, de définir une syntaxe concrète graphique et l'éditeur associé [9]. Cette génération d'éditeur s'appuie sur le résultat de la génération de la bibliothèque de sérialisation d'un modèle au format XMI fournie par EMF [BSE03]. La syntaxe concrète est décrite dans un *modèle de configuration* qui offre une grande liberté de personnalisation des éléments graphiques souhaités pour représenter les concepts. Dans TOPCASED, il a été utilisé pour engendrer les éditeurs pour les langages Ecore, UML2, AADL [10] et SAM [11].

Pour SIMPLEPDL, nous avons défini en premier lieu le modèle de notre syntaxe concrète (la figure 2.8 en présente une version simplifiée). *Activity* et *Guidance* sont définies comme *Node* (boîtes). *Precedes* est définie comme *Edge*, un arc entre deux boîtes. *Process* est représentée comme *Diagram* qui correspond à un paquetage qui contiendra les autres éléments. Définir la syntaxe concrète d'un langage peut nécessiter l'emploi d'éléments additionnels ne correspondant à aucun concept abstrait. Par exemple, ici il est indispensable d'ajouter *GuidanceLink* comme *Edge* pour relier une *Guidance* à une *Activity*. *GuidanceLink* ne correspond pourtant à aucun concept de *SimplePDL*. Il s'agit de « sucre » syntaxique dont la présence est obligatoire pour lier un élément graphique *Guidance* à la boîte représentant l'activité qu'il décrit. Il se traduit par la référence appelée *guidance* de *Activity* (cf. figure 2.6). Il faut noter que les concepts de la syntaxe abstraite (cf. figure 2.6) et ceux de la syntaxe concrète (cf. figure 2.8) sont des concepts différents qui doivent être mis en correspondance. Nous avons employé les mêmes noms

9. Nous proposons sur le site de documentation de TOPCASED, http://topcased-mm.gforge.enseeiht.fr, un tutoriel présentant les fonctionnalités du générateur de manière itérative et sur l'exemple de SIMPLEPDL

10. *Architecture Analysis & Design Language*, cf http://www.aadl.info/

11. *Structured automata Metamodel*, formalisme à base d'automates hiérarchiques utilisé par Airbus

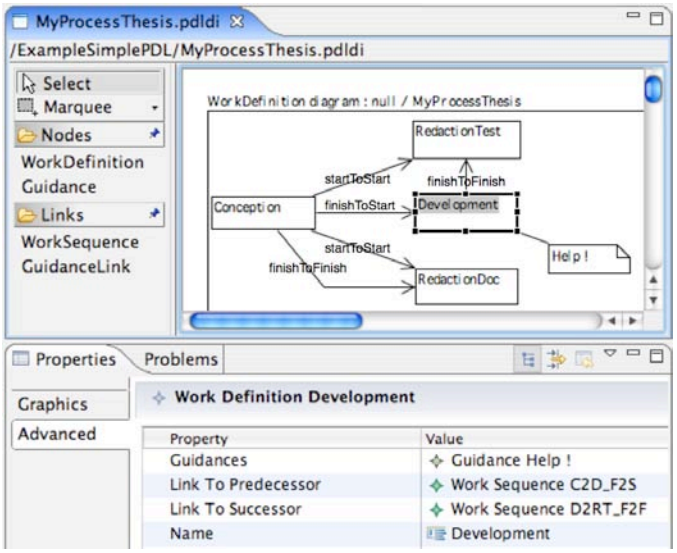


FIGURE 2.9: Éditeur graphique de SimplePDL généré avec TOPCASED

quand la correspondance était évidente. La figure 2.9 présente l'éditeur engendré. Tous les concepts du modèle de configuration sont présents dans la palette. Sélectionner un élément de la palette et le déposer sur le diagramme crée un élément graphique (*node* ou *edge*) et instancie, selon le modèle de configuration, la métaclasse correspondante du métamodèle SIMPLEPDL. Le modèle de configuration permet également de préciser la représentation graphique des différents éléments. Par exemple *WorkSequence* connecte deux *WorkDefinition* avec une flèche du côté de la cible.

Sémantique

Définir la sémantique d'un langage revient à définir le domaine sémantique et le *mapping* M_{as} entre la syntaxe abstraite et le domaine sémantique ($AS \leftrightarrow SD$). Le domaine sémantique définit l'ensemble des états atteignables par le système, et le *mapping* permet d'associer ces états aux éléments de la syntaxe abstraite.

Dans le contexte de l'IDM, au même titre que les autres éléments d'un langage de modélisation, la définition du domaine sémantique et du *mapping* prend la forme de modèle [Hau05, HbR00, HR04]. La sémantique des langages de modélisation est à ce jour rarement défini et fait actuellement l'objet d'intenses travaux de recherche. Nous détaillons cet aspect dans le chapitre 4.

2.3 La transformation de modèle

La deuxième problématique clé de l'IDM consiste à pouvoir rendre opérationnels les modèles à l'aide de transformations. Cette notion est au centre de l'ap-

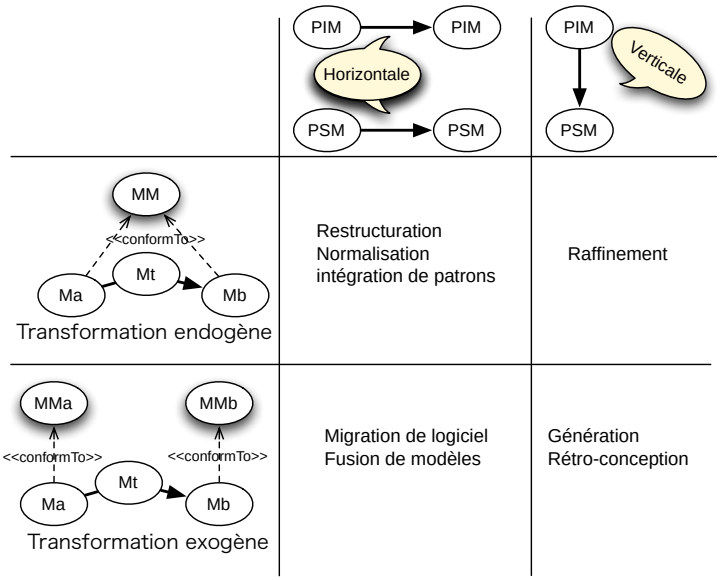


FIGURE 2.10: Types de transformation et leurs principales utilisations

proche MDA et plus généralement de celle des DSML. En effet, l'intérêt de transformer un modèle Ma en un modèle Mb que les métamodèles respectifs MMa et MMb soient identiques (transformation *endogène*) ou différents (transformation *exogène*) apparaît comme primordial (génération de code, *refactoring*, migration technologique, etc.) [Béz04b].

D'autre part, l'approche MDA repose sur le principe de la création d'un modèle indépendant de toute plateforme (PIM) pouvant être raffiné en un ou plusieurs modèle(s) spécifique(s) à une plateforme (PSM). Les méthodes de transformation sont là aussi indispensables pour changer de niveau d'abstraction (transformation *verticale*), dans le cas du passage de PIM à PSM et inversement, ou pour rester au même niveau d'abstraction (transformation *horizontale*) dans le cas de transformation PIM à PIM ou PSM à PSM [GLR+02]. Ces différentes classes de transformation sont reprises sur la figure 2.10 en indiquant leurs cas d'utilisation.

Enfin, la transformation de modèle est également utilisée dans la définition des langages de modélisation pour établir les *mappings* et des traductions entre différents langages. Ces différentes classes de transformation sont résumées sur la figure 2.11.

On comprend donc pourquoi le succès de l'IDM repose en grande partie sur la résolution du problème de la transformation de modèle. Cette problématique a donnée lieu ces dernières années à de nombreux travaux académiques, industriels et de normalisation que nous présentons ci-après.

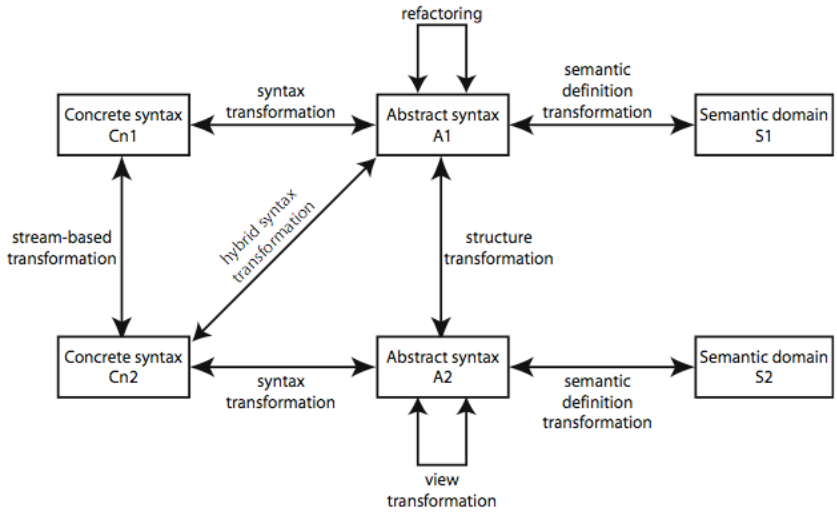


FIGURE 2.11: Classes de transformations dans la définition d'un DSML [Kle06]

2.3.1 Historique

Les travaux réalisés dans le domaine de la transformation de modèle ne sont pas récents et peuvent être chronologiquement classés selon plusieurs générations en fonction de la structure de donnée utilisée pour représenter le modèle [Béz03] :

- *Génération 1 : Transformation de structures séquentielles d'enregistrement.* Dans ce cas un script spécifie comment un fichier d'entrée est réécrit en un fichier de sortie (p. ex. des scripts Unix, AWK ou Perl). Bien que ces systèmes soient plus lisibles et maintenables que d'autres systèmes de transformation, ils nécessitent une analyse grammaticale du texte d'entrée et une adaptation du texte de sortie [GLR+02, Béz03].
- *Génération 2 : Transformation d'arbres.* Ces méthodes permettent le parcours d'un arbre d'entrée au cours duquel sont générés les fragments de l'arbre de sortie. Ces méthodes se basent généralement sur des documents au format XML et l'utilisation de XSLT [12] ou XQuery [13].
- *Génération 3 : Transformation de graphes.* Avec ces méthodes, un modèle en entrée (graphe orienté étiqueté) est transformé en un modèle en sortie. Ces approches visent à considérer l' « opération » de transformation comme un autre modèle (cf. figure 2.12) conforme à son propre métamodèle (lui-même défini à l'aide d'un langage de métamodélisation, par exemple le MOF). La transformation d'un modèle Ma (conforme à son métamodèle MMa) en un modèle Mb (conforme à son métamodèle MMb) par le modèle Mt peut donc être formalisée de la manière suivante :

$$Mb^* \leftarrow f(MMa^*, MMb^*, Mt, Ma^*)$$

12. *XSL (eXtensible StyleSheet Language) Transformation*, cf. http://www.w3.org/TR/xslt.
13. *An XML Query Language*, cf. http://www.w3.org/TR/xquery/.

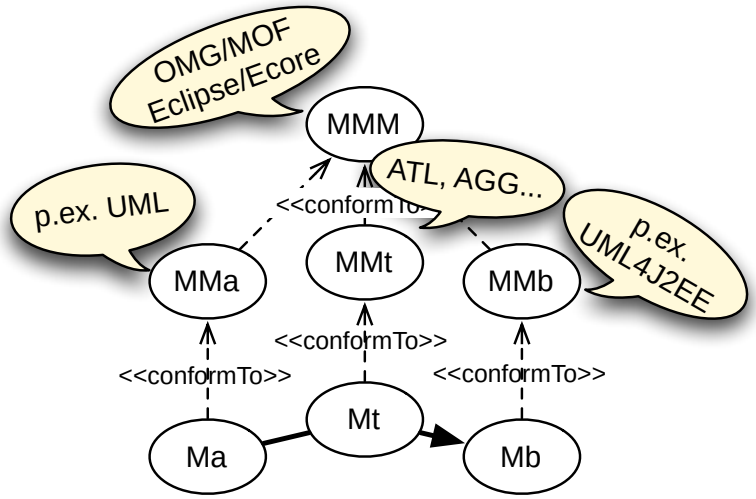


FIGURE 2.12: Principes de la transformation de modèle

Notons qu'il est possible d'avoir plusieurs modèles d'entrées (Ma^*) et de sorties (Mb^*).

Cette dernière génération a donné lieu à d'importants travaux de recherche et à la proposition de diverses approches [CH03]. Celles-ci peuvent être caractérisées à partir de critères comme le paradigme pour la définition des transformations, les scénarios de transformation, la directivité des transformations établies, le nombre de modèles sources et cibles, la traçabilité, le langage de navigation utilisé, l'organisation et l'ordonnancement des règles, etc. [Jou06].

2.3.2 Standards et langages pour la transformation de modèle

De nombreux langages sont à ce jour disponibles pour écrire des transformations de modèle de génération 3. On retrouve d'abord les langages généralistes qui s'appuient directement sur la représentation abstraite du modèle. On citera par exemple l'API [14] d'EMF [BSE03] qui, couplée au langage Java, permet de manipuler un modèle sous la forme d'un graphe. Dans ce cas, c'est à la charge du programmeur de faire la recherche d'information dans le modèle, d'expliciter l'ordre d'application des règles [15], de gérer les éléments cibles construits, etc.

Afin d'abstraire la définition des transformations de modèle et rendre transparent les détails de mise en œuvre, l'idée a été de définir des DSML dédiés à la transformation de modèle. Cette approche repose alors sur la définition d'un métamodèle dédié à la transformation de modèle et des outils permettant d'exécuter les

14. *Application Programming Interface* ou Interface de programmation.
15. Une règle est l'unité de structuration dans les langages de transformation de modèle.

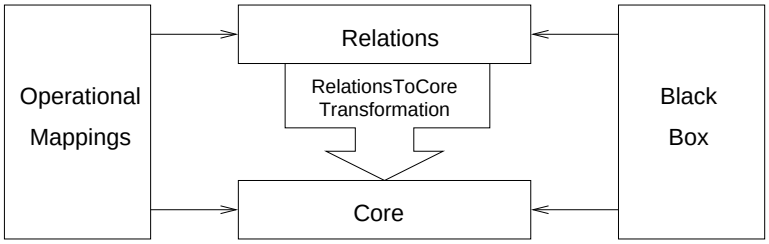


FIGURE 2.13: Architecture du standard QVT [OMG08]

modèles de transformation. Nous citerons par exemple ATL (*ATLAS Transformation Language*) [JK05] que nous utilisons tout au long de cette thèse. Il s'agit d'un langage hybride (déclaratif et impératif) qui permet de définir une transformation de modèle à modèle (appelée *Module*) sous la forme d'un ensemble de règle. Il permet également de définir des transformations de type modèle vers texte (appelée *Query*). Une transformation prend en entrée un ensemble de modèles (décrits à partir de métamodèles en Ecore ou en KM3).

Afin de donner un cadre normatif pour l'implantation des différents langages dédiés à la transformation de modèle, l'OMG a défini le standard QVT (*Query/-View/Transformation*) [OMG08]. Le métamodèle de QVT est conforme à MOF et OCL est utilisé pour la navigation dans les modèles. Le métamodèle fait apparaître trois sous-langages pour la transformation de modèles (cf. figure 2.13), caractérisés par le paradigme mis en œuvre pour la définition des transformations (déclaratif, impératif et hybride). Les langages *Relations* et *Core* sont tous deux déclaratifs mais placés à différents niveaux d'abstraction. L'un des buts de *Core* est de fournir une base pour la spécification de la sémantique de *Relations*. La sémantique de *Relations* est donnée comme une transformation de *Relations* vers *Core*. Il est parfois difficile de définir une solution complètement déclarative à un problème de transformation donné. Pour adresser cette question, QVT propose deux mécanismes pour étendre *Relations* et *Core* : un troisième langage appelé *Operational Mappings* et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage arbitraire (boîte noire ou *black box*). *Operational Mappings* étend *Relations* avec des constructions impératives et des constructions OCL avec effets de bord.

2.4 Discussion et synthèse

Nous avons introduit dans ce chapitre les principes généraux de l'IDM, c'est-à-dire la métamodélisation d'une part et la transformation de modèle d'autre part. Ces deux axes constituent les deux problématiques clé de l'IDM sur lesquelles la plupart des travaux de recherche se concentrent actuellement.

Les premiers résultats en métamodélisation ont permis d'établir les concepts (*modèle* et *métamodèle*) et relations (*représentationDe* et *conformeA*) de base dans

une architecture dirigée par les modèles. Malgré tout, les modèles sont actuellement construits à partir de DSML décrits principalement par leur structure. Nous avons vu que si la définition des syntaxes abstraites et concrètes était maîtrisée et outillée. Il est aussi possible de vérifier structurellement la conformité d'un modèle par rapport à son DSML. Cependant, nous n'avons pas abordé dans ce chapitre les aspects liés à la sémantique d'exécution qui font l'objet du chapitre 4. Notons qu'il n'y a toutefois pas de consensus sur la portée de la relation *conformeA*, en particulier si elle ne doit prendre en compte que la conformité *structurelle* (c.-à-d. le modèle respecte les *context conditions* de la syntaxe) ou une conformité plus large incluant la sémantique et vérifiant aussi la cohérence comportementale du modèle. Une formalisation précise de cette relation n'est à ce jour pas définie et la description précise de la sémantique des DSML reste une étape déterminante pour le passage à une approche complètement dirigée par les modèles (c.-à-d. une approche où les modèles seront nécessaires et suffisants pour décrire entièrement un système – logiciel par exemple – et dans laquelle, le recours à la programmation sera intégralement transparent pour le concepteur).

Les techniques de transformation de modèle, clé du succès de l'IDM afin de pouvoir rendre opérationnels les modèles, sont issues de principes qui sont bien antérieurs à l'IDM. Malgré tout, les travaux récents de normalisation et d'implantation d'outils ainsi que les nouveaux principes de l'IDM ont permis de faire évoluer ces techniques. Les transformations s'expriment maintenant directement entre les syntaxes abstraites des différents DSML et permettent ainsi de se concentrer sur les concepts et de gagner alors en abstraction. Malgré tout, des travaux sont encore nécessaires afin de formaliser ces techniques et pouvoir ainsi valider et vérifier les transformations écrites. Par exemple, les transformations permettant de générer du code à partir d'un modèle sont assimilables à une compilation qu'il est indispensable dans certains cas de certifier. D'autre part, si QVT offre un cadre très large pour la description de transformation de modèle en couvrant une large partie des approches possibles, ce n'est généralement pas le cas des implantations actuellement disponibles. En effet, la plupart des langages n'implantent qu'une partie du standard QVT, prévu grâce aux niveaux de conformité définis par l'OMG [OMG08, §2].

Chapitre 3

État de l'art sur l'ingénierie des procédés de développement

Nous avons abordé dans le chapitre précédent les principaux concepts de l'IDM. Nous montrons dans ce chapitre comment ils s'appliquent au domaine des procédés de développement (section 3.1) et plus particulièrement au travers du standard SPEM de l'OMG (section 3.2). Nous soulignons et analysons plus particulièrement les moyens proposés dans le standard pour exécuter un modèle de procédé (section 3.3). Nous terminons ce chapitre par une discussion sur les manques actuels et dégageons les objectifs de nos travaux pour participer à leurs résolutions (section 3.4).

3.1 Historique des procédés de développement

Dans le domaine du génie logiciel, il est depuis longtemps reconnu qu'une application est un produit manufacturé complexe dont la réalisation doit s'intégrer dans une démarche méthodologique : *le procédé de développement* [1]. Pour améliorer la qualité du produit développé, il est important de maîtriser son procédé de développement dont la problématique s'apparente à celle du logiciel lui-même [Ost87, Est05]. Plusieurs cycles de vie du logiciel (*Software Development Life Cycle* – SDLC) ont été proposés comme le cycle en cascade [Roy87], en spirale [Boe86] ou incrémental. Cependant, la communauté de la modélisation des procédés logiciels n'a pas été satisfaite de l'utilisation de ces descriptions de cycles de vie. La granularité des SDLC n'est pas assez fine et ne permet pas de décrire les parties élémentaires du procédé [CKO92]. Rapidement, le besoin a émergé de décrire avec plus de détails les procédés actuellement suivis pour le développement ou la maintenance de logiciel. L'idée a été de détailler la description de ces SDLC par les informations qui fournissent plus d'aide pour exécuter un projet de déve-

1. Dans la suite de cette thèse, nous emploierons indifféremment les termes de *procédé* et *processus*, qualifiant tous les deux une suite d'étapes réalisées dans un but donné [Ins90]

loppement logiciel. Ces travaux ont permis de faire apparaître la notion de modèle de procédé (*Process Model* – PM).

Dans les années 90, de nombreux travaux ont porté sur la modélisation de ce procédé à des fins de compréhension, d'évaluation, d'amélioration ou d'exécution. De nombreux AGL-P, Ateliers de Génie Logiciel centré Procédé, ont été proposés. Il s'agit alors de piloter un développement réel en fonction d'un procédé de développement défini dans un langage dédié [BEM94, CHL+94a, BFL+95, CC97, Bre02].

Les procédés peuvent être exprimés sous la forme d'un modèle décrit selon un langage de modélisation de procédé (*Process Modeling Language* – PML). Un modèle de procédé est une représentation des activités du monde réel. Le modèle de procédé est développé, analysé, raffiné, transformé et/ou exécuté conformément au métamodèle du procédé. Les PML et leurs Environnements de Génie Logiciel Sensibles au Procédé (PSEE) peuvent être classés en trois catégories suivant l'élément central du modèle de procédé :

- Les *PML centrés Produits* se concentrent surtout sur les données échangées, et ont développé d'importantes propriétés concernant ces données (données orientées objet, transactions, persistance, versionnement, etc.). Ces PML ont eu beaucoup de succès dans l'industrie, surtout dans le domaine de la gestion de configuration. Nous citons par exemple les systèmes ADELE [BEM94] et EPOS [CHL+94b].
- Les *PML centrés Rôles* mettent l'accent sur les rôles et leurs collaborations. Ces PML ont surtout eu du succès dans le domaine du travail coopératif.
- Les *PML centrés Activités* s'appuient sur les activités qui représentent les tâches à réaliser. Cette catégorie a eu beaucoup de succès dans le domaine de la recherche (p. ex. SPADE [BFGL94], MARVEL [KFP88], APEL [DEA98], etc.). Du côté industriel, la construction de *workflows* a commencé vers la fin des années 90, et ceci indépendamment de tous les travaux de recherche qui ont été faits sur les procédés. Les *workflows* sont des procédés exécutables, centrés activités et généralement simples.

3.2 SPEM, le standard de l'OMG pour la modélisation des procédés

À l'instar d'UML pour les notations objets, la tendance actuelle est à l'unification des PML dans l'optique industrielle indispensable de la réutilisation. Citons par exemple SPEM [OMG07a], le métamodèle défini par l'OMG pour la modélisation des procédés de développement, ou les langages basés sur XML tels que XPDL [WfM05] proposé par le WfMC (*Workflow Management Coalition*) ou BPML [Ark02] proposé par le BPMI (*Business Process Management Initiative*). Ces approches constituent pour l'essentiel une unification des concepts mais leur sémantique reste décrite informellement.

SPEM (*Software & Systems Process Engineering Metamodel*) est le standard

de l'OMG dédié à la modélisation des procédés logiciels et systèmes. Il vise à offrir un cadre conceptuel pour modéliser, échanger, documenter, gérer et présenter les processus et méthodes de développement.

Ses concepts sont décrits par un métamodèle qui repose sur l'idée qu'un procédé de développement est une collaboration entre des entités actives et abstraites, les *rôles*, qui représentent un ensemble de compétences et qui effectuent des opérations, les *activités*, sur des entités concrètes et réelles, les *produits*. Les différents rôles agissent les uns par rapport aux autres ou collaborent en échangeant des produits et en déclenchant l'exécution de certaines activités. Ce modèle conceptuel est synthétisé sur la figure 3.1.

Après la version 1 de SPEM [OMG05a], dont nous proposons une évaluation dans [CCOP06] et une formalisation dans [CCCC06, Com05], l'OMG est actuellement en train de finaliser la version 2 [OMG07a]. En plus de fournir un moyen standard pour représenter l'expertise et les procédés d'une organisation, SPEM2.0 est défini selon une vision attractive. Celle-ci consiste à séparer les aspects contenus et données relatifs aux méthodologies de développement, de leurs possibles instanciations dans un projet particulier. Ainsi, pour pleinement exploiter ce *framework*, la première étape devrait être de définir toutes les phases, activités, produits, rôles, guides, outils, etc., qui peuvent composer une méthode pour ensuite, dans une deuxième étape, choisir en fonction du contexte et du projet, le contenu de la méthode appropriée pour l'utiliser dans la définition du procédé.

SPEM2.0 est défini sous la forme d'un métamodèle MOF [OMG06b] qui s'appuie sur les spécifications UML2.0 Infrastructure [OMG07b] et UML2.0 Diagram Interchange [OMG06a]. Il réutilise de UML2.0 infrastructure les concepts de base comme *Classifier* ou *Package*. Aucun concept de UML2.0 Superstructure [OMG07b] n'est réutilisé. Le standard est défini également sous la forme d'un profil UML où chaque élément du métamodèle de SPEM2.0 est défini comme un stéréotype de UML2.0 Superstructure.

Le métamodèle de SPEM2.0 est composé de sept paquetages liés avec le mécanisme « merge » [OMG07b, §11.9.3], chaque paquetage traitant d'un aspect par-

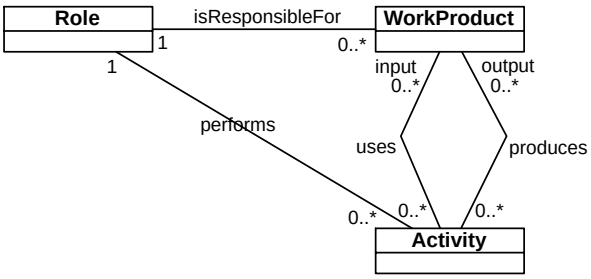


FIGURE 3.1: Modèle conceptuel de SPEM

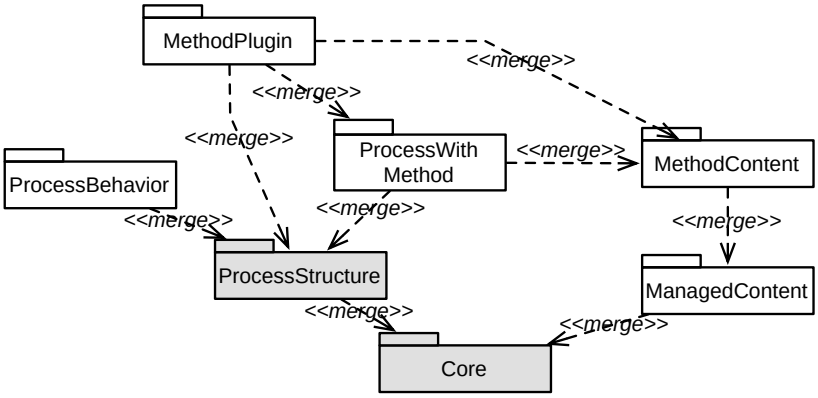


FIGURE 3.2: Structure du métamodèle de SPEM2.0 [OMG07a]

ticulier (cf. figure 3.2). Le paquetage *Core* introduit les classes et les abstractions qui définissent les fondations pour tous les autres paquetages. La structure de décomposition de ce paquetage est la classe *WorkDefinition*, qui généralise toutes les activités de travail de SPEM2.0. Le paquetage *ProcessStructure* définit les éléments permettant de représenter les modèles de procédé en terme de flots d'activités (*Activities*) avec leurs utilisations de produit (*WorkProductUses*) et de rôle (*RoleUses*). Cependant, la possibilité de documenter textuellement ces éléments (c.-à-d. ajouter les propriétés décrivant chaque élément) n'est pas fournie dans ce paquetage mais dans *ManagedContent*, qui définit les concepts pour gérer les descriptions textuelles des éléments de procédé. Des exemples de ces concepts sont les classes *ContentDescription* et *Guidance*. Le paquetage *MethodContent* définit les concepts de base pour spécifier le contenu de méthodes basiques, comme les rôles (*Roles*), les tâches (*Tasks*) et les produits (*WorkProducts*). Le paquetage *ProcessWithMethod* regroupe l'ensemble des éléments requis pour intégrer les processus définis avec les concepts du paquetage *ProcessStructure*, selon les contenus définis avec les concepts du paquetage *MethodContent*. Le paquetage *MethodPlugin* offre les mécanismes pour gérer et réutiliser des bibliothèques de contenus de méthode et procédé. Ceci est assuré grâce aux concepts *MethodPlugin* et *MethodLibrary*. Enfin, le paquetage *ProcessBehavior* offre le moyen de lier un élément de procédé SPEM2.0 avec un comportement externe comme des diagrammes d'activités UML2.0 ou des modèles BPMN (*Business Process Modeling Notation*).

3.3 Vers une exécutabilité des modèles de procédé

Même si l'exécution des modèles de procédé était une des principales exigences de l'appel à proposition (*Request For Proposal* – RFP) de la version 2 de SPEM [OMG04], la récente adoption de la spécification ne répond pas au problème

de l'exécutabilité. Néanmoins, la spécification suggère clairement deux solutions pour exécuter un modèle de procédé SPEM2.0. Nous présentons dans la suite de cette section ces deux solutions et donnons quelques remarques sur leur faisabilité.

3.3.1 Traduction d'un modèle de procédé SPEM2.0 dans un outil de planification

Dans la première solution, le standard propose de traduire un modèle de procédé dans un outil de planification et d'exécution comme *IBM Rational Portfolio Manager* ou *Microsoft Project*. Les procédés SPEM2.0 définis en utilisant les structures de décomposition (c.-à-d. *Activity*, *Role Use* et *WorkProduct Use* du paquetage *Process Structure*) offrent les attributs clés pour fournir le planning du projet à partir des aides (*Guidance*) nécessaires aux décisions pour l'instanciation du procédé. Des exemples de ces attributs sont *hasMultipleOccurrence*, qui indique qu'une activité (*Activity*) ou un produit (*WorkProduct*) devra être traduit plusieurs fois ; L'attribut *isRepeatable* pour une activité indique qu'elle devra être itérée plusieurs fois avant d'obtenir le résultat escompté ; L'attribut *isOptional* indique qu'une activité est optionnelle et pourra donc ne pas être réalisée au cours de l'exécution du procédé. Une fois que les procédés SPEM2.0 sont traduits dans un outil de planification celui-ci peut être utilisé pour affecter les ressources concrètes allouées au projet (ressources humaines, matérielles, logicielles, etc.).

Cependant, même si cette approche est très utile pour la planification de projet, elle ne peut pas être considérée comme une réelle exécution du procédé. Il est nécessaire d'affecter les durées de chaque tâche, les compétences de chaque personne, afin d'avoir à la fin de l'exécution (prédictive) une estimation des ressources et du temps pour la réalisation du procédé. Ces planifications sont utilisées par le chef de projet afin d'estimer si le procédé sera réalisable ou non, s'il est nécessaire d'affecter plus de ressources aux activités, etc. Il n'y a pas de support de l'exécution du procédé, pas d'affectation automatique des tâches aux rôles responsables, pas de contrôle automatique du flot de données et de l'état des produits après chaque activité, pas de moyen pour supporter la communication dans l'équipe, etc. En plus du fait que cette approche n'offre pas de support concret de l'exécution, elle présente le manque majeur de ne pas avoir de lien clair avec les outils de planification. Un autre aspect qui doit être pris en compte est l'impact d'une modification ou d'un ajout d'information dans l'outil de planification et comment cette modification peut être répercutée ou tracée dans le modèle de procédé SPEM2.0. Enfin, le modeleur de procédé doit s'assurer de la compatibilité du format de fichier de la définition du procédé de l'outil de planification.

3.3.2 Expression avec un autre formalisme du comportement des éléments de procédé SPEM2.0

Le standard SPEM2.0 n'offre pas de concept ou de formalisme pour exprimer le comportement ou l'exécution précise d'un procédé. Toutefois, prétendant à plus

de flexibilité, SPEM2.0 offre, à travers le paquetage *ProcessBehavior*, le moyen d'associer aux éléments de procédé SPEM2.0 un comportement exprimé avec un autre formalisme. L'objectif est de ne pas imposer un modèle comportemental précis mais de donner la possibilité, au concepteur des procédés, de choisir celui qui répond le mieux à ses besoins. Une activité SPEM2.0 peut, par exemple, être liée à un diagramme BPMN afin de représenter avec plus de détails les étapes de l'activité, le flot de contrôle, etc. Le moteur d'exécution BPMN est alors réutilisé. Une traduction vers BPEL (*Business Process Execution Language*) est également envisageable afin de réutiliser un moteur d'exécution BPEL existant. Par ailleurs, un produit peut par exemple être lié à un diagramme de machine à états UML afin de décrire les différents états d'un produit et les transitions entre ces états. Ici aussi, un moteur d'exécution de machines à états doit être intégré au moteur d'exécution du procédé. SPEM2.0 définit différents types de classe *proxy* (*Activity_ext*, *ControlFlow_ext*, *Transition_ext* et *State_ext*) afin de lier aux éléments de procédé SPEM2.0 (c.-à-d. *WorkProductUse*, *WorkDefinition*, *RoleUse*, *Activity* et *WorkSequence*) un comportement externe. Le concepteur du procédé doit alors lier les éléments du procédé avec leurs équivalents dans le modèle comportemental choisi. Puisqu'un modèle comportemental peut ne pas être assez expressif pour tous les aspects comportementaux d'un processus, plusieurs formalismes peuvent être combinés.

Même si cette approche peut offrir plus de flexibilité dans la représentation des aspects comportementaux des procédés SPEM2.0, elle présente différents manques. Le premier est que le standard n'est pas vraiment clair sur les moyens de lier les éléments de procédé avec un modèle comportemental. Il offre juste des classes *proxy* qui font référence à d'autres éléments dans un modèle comportemental externe. Nous supposons que cette tâche est alors à la charge des outilleurs. Ils auront donc à définir un modèle comportemental spécifique qui sera engendré automatiquement à partir du modèle de procédé. C'est déjà le cas dans l'outil du projet Eclipse EPF [2] qui est défini comme une implémentation de SPEM2.0. Dans EPF, un diagramme d'activité propriétaire est partiellement généré à partir de la définition du procédé. Il peut ensuite être raffiné afin d'offrir plus de détails sur les activités du procédé et leurs coordinations (flots de contrôle). Cependant, aucune exécution n'est fournie.

Le deuxième manque est que la traduction des éléments de procédé SPEM2.0 dans le modèle comportemental spécifique peut être différente d'une organisation à une autre, en fonction de l'interprétation du modeleur du procédé. Donc, un effort de standardisation est nécessaire afin d'harmoniser les règles de traduction entre les concepts de SPEM2.0 et un modèle comportemental. Nous citerons par exemple les travaux de Bendraou *et al.* [BSGB07] qui proposent une traduction vers BPEL.

Le troisième manque, qui est relatif au précédent, est que le plus souvent les concepts des modèles comportementaux sont plus riches que les concepts de SPEM2.0. Cela est dû au fait que la modélisation du comportement et de l'exécution offre des concepts supplémentaires relatifs au support technique et l'exécution

2. *Eclipse Process Framework* Project, http://www.eclipse.org/epf

du procédé tandis que SPEM2.0 se concentre sur les préoccupations « métiers » des procédés ou méthodologies de développement (c.-à-d., *Roles*, *Activities*, *Guidance*, etc.). En conséquence, une génération complète du code exécutable à partir de SPEM2.0 n'est pas possible sans des étapes préalables de raffinement. Ceci pose alors le problème de la traçabilité et de comment ces raffinements (changements) peuvent être répercutés dans le modèle de procédé SPEM2.0.

3.4 Discussion et synthèse

Nous avons présenté dans ce chapitre les concepts principaux de l'ingénierie des procédés de développement, plus particulièrement à travers le PML SPEM, proposé par l'OMG. Celui-ci s'inscrit dans l'IDM grâce à sa définition sous la forme d'un métamodèle (tel que nous l'avons présenté dans le chapitre 2).

L'étude détaillée de ce standard nous amène à souligner les manques actuels afin de pouvoir exécuter un modèle SPEM, par exemple pour étudier très tôt la faisabilité d'un projet, estimer les ressources, etc. Il nous semble pour cela important d'intégrer au standard les informations pour permettre d'appréhender dans le modèle son exécution, régie par une sémantique comportementale précise et explicite. A ce titre, cette problématique correspond plus généralement à celle de l'exécutabilité des modèles, et donc celle de la sémantique comportementale des DSML ainsi que de leur outillage. C'est pourquoi nous avons choisi d'utiliser ce domaine d'application de manière *académique* tout au long de cette thèse, pour appliquer nos travaux plus génériques visant à offrir une démarche outillée de métamodélisation. Pour cela, nous présentons tout d'abord dans le chapitre 4 des expérimentations avec les outils actuels pour exécuter des modèles décrits à partir du PML SIMPLEPDL présenté dans le chapitre 2. Nous présentons ensuite, dans le chapitre 5, une extension de SPEM2.0, XSPEM, dont les modèles supportent l'exécution du procédé pour un projet particulier. Les chapitres 6 et 7 présentent respectivement la définition des outils de vérification formelle et de simulation pour XSPEM. Enfin, le chapitre 8 présente un cadre formel pour la définition de DSML, et nous montrons la formalisation possible de XSPEM.

Nous évaluons la généricité de notre approche au sein du chapitre 9 en l'appliquant à d'autres domaines comme celui des systèmes embarqués critiques.

Chapitre 4

Définition de la sémantique d'exécution d'un DSML

L'IDM a permis d'établir une nouvelle approche, plus abstraite, pour le développement de systèmes complexes. Un système peut être décrit par différents modèles liés les uns aux autres. L'idée phare est d'utiliser autant de modèles différents que les aspects chronologiques ou technologiques du développement du système le nécessitent. La principale différence avec un programme est qu'un modèle privilégie la syntaxe abstraite alors qu'un programme favorise la syntaxe concrète. La syntaxe abstraite se focalise sur les concepts fondamentaux du domaine et laisse apparaître une sémantique à travers le vocabulaire choisi pour nommer les concepts et les relations ainsi que la structure de graphe établie par les relations. Cette syntaxe abstraite est suffisante pour offrir une représentation homogène du système. Elle facilite l'interopérabilité entre les outils en permettant de traduire les données utilisées par un outil dans la syntaxe utilisée par un autre outil. Cependant, lorsque l'on souhaite comprendre précisément la signification du modèle et la manière de l'interpréter sans ambiguïté, il est indispensable de définir la sémantique pour chaque langage de modélisation (chaque DSML dans notre cas). Ceci est particulièrement le cas pour les modèles utilisés dans le cadre de systèmes critiques (temps réel, embarqués, fiables, redondants, performants...) qui imposent de valider les solutions envisagées au plus tôt dans le processus de développement. Le projet TOPCASED se place dans ce contexte : les modèles construits à partir des différents DSML de l'atelier doivent pouvoir être simulés et vérifiés.

Il y a différents moyens d'exprimer la sémantique d'un langage. Toutefois, qu'il s'agisse d'un langage informatique ou du langage naturel, la définition du sens des concepts se rapporte à la définition d'un *mapping* vers un domaine sémantique comme nous l'avons décrit dans le chapitre 2. Pour le langage naturel, il se ramène à un mapping vers la connaissance et l'expérience que l'on a du concept dans le monde qui nous entoure et le plus souvent dans notre langue maternelle.

Pour les langages de modélisation, la sémantique comportementale est généralement décrite de manière *ad-hoc* pour chaque langage. Elle nécessite un tel

effort que seuls les langages les plus populaires sont pris en compte. C'est par exemple le cas pour UML, pour lequel de nombreuses machines virtuelles ont été définies (p. ex. OxUML [JZM07a], xtUML [Men07], iUML [xUM07, iUM07], XIS-xModels [LdS04], Rhapsody [GHP02] et *Executable* UML [MBB02]) avec pour la plupart un langage d'action (p. ex. OCL4X [JZM07b] dans OxUML). Il existe aussi de nombreux générateurs de code, souvent basés sur UML. Ce type de solution a aussi été utilisée dans le cadre de l'ingénierie des procédés pour des formalismes généralement spécifiques. Nous citerons par exemple les travaux de Bendraou [Ben07] proposant une spécialisation d'UML dédiée aux procédés de développement permettant d'exécuter un modèle de procédé à partir d'un comportement décrit en Java. Toutes ces solutions définissent précisément la sémantique du langage considéré mais celles-ci restent le plus souvent implicites, parfois même incompatibles entre différents outils supportant le même langage, et sans capitalisation entre les différents travaux. Elles ont également toutes recours à l'ingénierie de la programmation pour exprimer la sémantique et ne profite donc pas de l'abstraction offerte par les langages de modélisation.

Plus récemment, des travaux ont proposé des langages d'action plus abstraits permettant de définir une sémantique d'exécution pour un DSML donné. Ces langages ne sont alors plus spécifiques à un DSML particulier, mais s'appuient sur les concepts plus abstraits de la métamodélisation (cf. figure 2.5). Ce gain d'abstraction permet d'offrir des outils génériques (machine virtuelle, générateur de code) qui s'appliquent à n'importe quelle syntaxe abstraite et qui interprètent le comportement défini à l'aide du langage d'action.

De nombreux projets s'appuyant sur l'IDM sont en cours de réalisation pour la validation de systèmes complexes et/ou critiques. C'est par exemple le cas dans les projets TOPCASED [1], SPACIFY [2], GENEAUTO [3], SPICES [4], OPENEMBEDD [5], MODELWARE [6] et MODELPLEX [7]. L'expression de la sémantique est un point majeur de ces différentes initiatives avec des solutions différentes. Il devient important de capitaliser les solutions proposées pour construire une ingénierie de la sémantique en *(méta)*modélisation. Cette problématique est d'ailleurs au centre des discussions qui ont lieues dans le cadre de l'atelier SéMo (*Sémantique des (meta) Modèles*) que nous avons fondé et dont les deux premières éditions ont eu lieu en 2007 [8] à Toulouse et en 2008 [9] à Mulhouse conjointement à la conférence francophone IDM [10].

1. cf. http://www.topcased.org
2. cf. http://spacify.gforge.enseeiht.fr
3. cf. http://geneauto.gforge.enseeiht.fr
4. cf. http://www.spices-itea.org
5. cf. http://openembedd.inria.fr
6. cf. http://www.modelware-ist.org
7. cf. http://www.modelplex-ist.org
8. cf. http://semo2007.enseeiht.fr et compte rendu dans l'article [CCMP07].
9. cf. http://semo2008.enseeiht.fr.
10. *Ingénierie Dirigée par les Modèles*, cf. http://megaplanet.org/idm07 et http://megaplanet.org/idm08.

Nous introduisons dans ce chapitre une taxonomie des différents types de sémantique pour un DSML et les moyens de la définir au sein d'un métamodèle. Nous proposons pour cela une discussion au regard des travaux de la communauté des langages de programmation et des techniques utilisées. Nous mettons plus particulièrement l'accent sur les sémantiques permettant d'exécuter un modèle en vue de l'animer, le simuler ou le vérifier dynamiquement. Nous illustrons ensuite ces différentes techniques à travers des expérimentations pour définir la sémantique d'exécution d'un langage simple de modélisation de processus : SIMPLEPDL. Ces expérimentations permettent d'évaluer les différences entre les approches et d'introduire les outils qui peuvent être utilisés pour les mettre en œuvre. Un bilan de ces expérimentations et une discussion sur leurs mises en œuvre concluent ce chapitre et permettent de préciser la problématique de cette thèse.

> La taxonomie des sémantiques ainsi que les différentes mises en œuvres et discussions ont fait l'objet d'une publication au sein de la conférence francophone IDM 2006 [CRC+06a] et dans le workshop international MDEIS 2006 [CRC+06b].

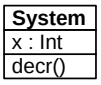
4.1 Taxonomie des sémantiques dans l'IDM

La sémantique des langages de programmation est formalisée depuis de nombreuses années et il semble pertinent de profiter de l'expérience acquise pour explorer les moyens d'exprimer la sémantique des langages de modélisation. La sémantique d'un langage définit de manière précise et non ambiguë la signification des constructions de ce langage. Elle permet ainsi de donner un sens précis aux programmes construits à partir de celui-ci. On dit qu'une sémantique est *formelle* lorsqu'elle est exprimée dans un formalisme mathématique et permet de vérifier la cohérence et la complétude de cette définition. On distingue la *sémantique statique* qui correspond à des propriétés indépendantes de l'exécution ou valables pour toutes les exécutions. Celle-ci est en général vérifiée statiquement lors de la compilation des programmes (et exprime des règles, comme le typage, qui ne peuvent pas être exprimées par la syntaxe) et la *sémantique dynamique* (ou *comportementale*) qui permet de décrire le comportement des programmes à l'exécution. Pour aller de la plus concrète à la plus abstraite, une sémantique comportementale peut être opérationnelle, dénotationnelle ou axiomatique [Win93]. Une *sémantique opérationnelle* donne une vision impérative en décrivant un programme par un ensemble de transitions (ou transformations) entre les états du contexte d'exécution (p. ex. de la mémoire). Une *sémantique dénotationnelle* décrit, sous forme de fonctions, l'effet d'un programme et non la manière dont celui-ci est exécuté. Une *sémantique axiomatique* propose une vision déclarative en décrivant l'évolution des caractéristiques d'un élément lors du déroulement d'un programme.

Par analogie avec l'approche suivie pour les langages de programmation, le défi actuel dans l'IDM consiste à donner les moyens d'exprimer une sémantique précise pour les concepts de la syntaxe abstraite des langages de modélisation (de

```
int x;
void decr () {
        if ( x>0 )
                x = x − 1;
}
```

(a) Programme



(b) Modèle

FIGURE 4.1: Fonction *decr*

manière à pouvoir en exécuter les modèles). Tel que nous avons défini un DSML (cf. section 2.2), l'expression d'une sémantique correspond à définir un *mapping* de chaque état du modèle, capturé par la syntaxe abstraite, vers un domaine sémantique qui définit l'ensemble des états possibles du système. Dans le contexte de l'IDM, cette fonction prend naturellement la forme d'un modèle (cf. M_{as}, figure 2.4) et il est donc indispensable d'étudier le langage de modélisation adéquat pour le définir. Notons que ce langage (L_{a2s}) doit permettre de manipuler à la fois la syntaxe abstraite du DSML considéré et celle du domaine sémantique choisi. Cette propriété peut être formalisée de la manière suivante : $L_{AS}, L_{SD} \subseteq L_{a2s}$ [HR04].

On peut classer les différentes sémantiques selon les catégories définies pour les langages de programmation (sémantique axiomatique, dénotationnelle et opérationnelle), et appliquer celles-ci selon les besoins (vérification de modèle, animation de modèle, etc.) et les outils disponibles. Nous les présentons dans la suite de cette partie en les illustrant à travers l'exemple simple de la fonction *decr* dont le programme et un de ses modèles sont fournis sur la figure 4.1. Ce programme décrit une fonction qui décrémente de un la valeur de x à condition qu'elle soit strictement positive. Nous exprimons par la suite la sémantique selon les trois niveaux d'abstraction cités.

La *sémantique axiomatique* ainsi que la sémantique statique (également appelée *context condition* par l'OMG et correspondant à une sémantique axiomatique très simple pouvant être vérifiée statiquement [11]) peuvent être exprimées sur la syntaxe abstraite, soit à l'aide du langage de métamodélisation lui-même (p. ex. les multiplicités) soit en la complétant de contraintes exprimées à l'aide d'un langage comme OCL (invariant, pré- ou post-condition). Les pré- et post-conditions supposent d'avoir identifiées des opérations sur le modèle. Une spécification axiomatique de la sémantique de la fonction *decr* peut, par exemple, être définie par la propriété OCL donnée sur le *listing* 4.1.

Listing 4.1: Sémantique axiomatique de la fonction *decr*

```
context System::decr()
        post : self.x = if ( self.x@pre > 0 ) then
```

11. Notons qu'en UML il est possible d'exprimer des propriétés OCL portant sur des séquences d'envois de messages. Cependant, ces propriétés ne sont pas vérifiées par les vérificateurs OCL car elles nécessitent la formalisation de la sémantique d'exécution du langage, par exemple UML, et la mémorisation de la trace d'exécution

self .x@**pre** − 1
else
self .x@**pre**
endif

Les sémantiques opérationnelle et dénotationelle peuvent également être exprimées pour un DSML et correspondent toutes les deux à la définition d'un *mapping* vers un domaine sémantique. Dans le cadre de nos travaux, nous différencions dans l'IDM ces deux types de sémantique d'exécution de la manière suivante.

La *sémantique opérationnelle* s'exprime sur une représentation abstraite identique à celle de la syntaxe abstraite du langage considéré. Elle est dans ce cas exprimée sur des concepts instanciés à partir des paradigmes utilisés pour la définition de la syntaxe abstraite (dans notre cas, les concepts de métamodélisation). Ces concepts sont dénués de sémantique mais un *langage d'action* permet de l'exprimer et ainsi de définir les outils support à l'exécution. Un tel langage permet de décrire l'évolution du modèle et de produire, à partir d'un état donné, un ou plusieurs autres états (dit successeurs). Dans le contexte des langages naturels, une sémantique opérationnelle revient à manipuler une phrase (le plus souvent mentalement), dans la même langue que celle utilisée pour l'écrire initialement. On lui donne dans ce cas un sens selon l'expérience acquise (propre à chacun). De manière plus formelle, notons qu'une sémantique opérationnelle est, par définition, fortement bisimilaire [12] au système de transition de référence du système. En effet, la représentation abstraite sur laquelle elle est exprimée étant identique à la syntaxe abstraite du langage, chaque état exprimé par la sémantique opérationnelle est observable par le système de transition de référence.

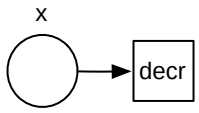
Pour l'exemple de la fonction *decr*, la sémantique opérationnelle peut être spécifiée de la manière suivante : $\mathbb{Z} \mapsto \mathbb{Z}$ tel que $x \mapsto if(x > 0)?x - 1 : x$. Le domaine sémantique choisi dans ce cas correspond à $\mathbb{Z}$.

La *sémantique dénotationnelle* s'exprime sur une représentation abstraite différente de celle définie dans la syntaxe abstraite du langage considéré. Les paradigmes utilisés sont fixes et sémantiquement bien définis. Ils sont adaptés à la construction d'outils efficaces pour le support de l'exécution (p. ex. des outils de *model-checking*). Dans le contexte des langages naturels, cela revient à interpréter une phrase dans une langue différente que celle dans laquelle elle a été initialement écrite. L'expression de la sémantique correspond à trouver alors une traduction dans la nouvelle langue pour que la phrase garde le même sens. De manière plus formelle et par expérience, le domaine choisi est alors faiblement bisimilaire au système de transition de référence du système. Il est alors nécessaire de retrouver les états que l'on souhaite observer dans le formalisme initial. Ce type de sémantique correspond dans l'IDM à une *sémantique par traduction* qui définit un langage par sa transformation vers un autre langage formellement défini [CESW04]. Nous utilisons dans la suite de ce mémoire cette terminologie.

Une sémantique par traduction de la fonction *decr* peut correspondre à la fonc-

12. Selon la définition de la bisimulation donnée dans [Mil95].

tion qui traduit ce programme dans un domaine formel tel que les réseaux de Petri [Rei85]. Une telle fonction pourrait alors fournir le réseau suivant pour la fonction *decr*, où la valeur de x est représentée par le marquage de la place de même nom :



Les réseaux de Petri ayant une sémantique formelle, le programme prend alors le sens du réseau de Petri généré.

Nous nous concentrons dans la suite de cette thèse sur la notion de *sémantique d'exécution* qui permet de donner un caractère opératoire à un DSML. Le métamodèle est alors dit *exécutable* et permet de faire évoluer les modèles qui lui sont conformes au cours d'une exécution conformément à la sémantique du langage. Nous nous restreignons pour cela à l'étude des sémantiques opérationnelles et par traduction vers un domaine sémantique opérationnel.

Nous présentons dans les sections suivantes les technologies actuellement disponibles dans la communauté de l'IDM pour définir les différents types de sémantique décrits précédemment. Pour les illustrer, nous nous appuyons sur SIMPLEPDL, un langage simplifié de modélisation de processus dont les syntaxes (abstraite et concrète) ont été définies dans le chapitre 2. Dans cet exemple, l'exécution d'un processus consiste à réaliser toutes les activités qui le composent. Le processus est dit terminé quand toutes ses activités sont terminées. Une activité ne peut être commencée que si les activités dont elle dépend sont commencées (précédence *start-to-start*) ou terminées (précédence *finish-to-start*). Au fur et à mesure du développement, le taux de réalisation d'une activité augmente jusqu'à ce qu'elle soit terminée. Elle ne peut être terminée que si les activités précédentes de type *finish-to-finish* sont terminées et les activités précédentes de type *start-to-finish* sont commencées. Ce sont les équipes de développement qui décident de quelle activité commencer, continuer (i.e. augmenter le taux d'avancement) ou terminer.

4.2 Sémantique opérationnelle : expérimentations

La sémantique opérationnelle permet de décrire le comportement dynamique des constructions d'un langage. Dans le cadre de l'IDM, elle vise à exprimer la sémantique comportementale de la syntaxe abstraite, à l'aide d'un langage d'action, afin de permettre l'exécution des modèles qui lui sont conformes.

Dans le cadre de notre exemple SIMPLEPDL, exprimer la sémantique d'exécution nécessite de compléter notre syntaxe abstraite initiale (cf. figure 2.6) avec les informations liées à l'état des activités. Nous avons donc ajouté l'attribut *progress* sur la métaclasse *WorkDefinition* (cf. figure 4.2). Il représente le taux de progression d'une activité avec le codage suivant : non commencée (-1), en cours (0..99),

ou terminée (100).

Pour exprimer la sémantique opérationnelle, nous avons envisagé deux approches. La première consiste à définir le comportement de chaque concept de manière impérative à l'aide d'un langage de métaprogrammation (p. ex. Kermeta [MFJ05], xOCL [CSW08b] ou l'API Java d'EMF) ou la spécification d'un langage d'action (p. ex. AS-MOF [Bre02, PKP06]). Notre expérimentation a été réalisée avec Kermeta. La seconde approche s'inspire des règles de réduction des langages de programmation (*Structural Operational Semantics* [Plo81], sémantique naturelle [Kah87]). Elle consiste à définir le comportement par des transformations endogènes exprimées sur la syntaxe abstraite à l'aide d'un langage comme ATL [JK05], ou issues de la transformation de graphe comme AGG [Tae03]. Celles-ci expriment les modifications de l'environnement d'exécution et du système en fonction de l'état courant du modèle.

La principale différence est qu'une approche par métaprogrammation consiste à enrichir la syntaxe abstraite par des opérations décrivant l'évolution du modèle, par exemple de manière impérative. Une approche par transformation de modèle endogène permet de définir de manière déclarative le comportement sous la forme d'un système de transition, en fonction des états possibles du modèle.

4.2.1 Mise en œuvre avec Kermeta

Kermeta [MFJ05] est défini comme un langage de métamodélisation exécutable ou de métaprogrammation objet : il permet de décrire des métamodèles dont les modèles sont exécutables. Kermeta est basé sur le langage de métamodélisation EMOF [OMG06b] et est intégré à l'IDE Eclipse sous la forme d'un *plugin*. Il a été conçu comme un tissage entre un métamodèle comportemental et EMOF. Le métamodèle de Kermeta est donc composé de deux packages, *Core* et *Behavior*. Le premier correspond à EMOF et le second est une hiérarchie de métaclasses représentant des expressions impératives. Ces expressions sont utilisées pour décrire la sémantique comportementale des métamodèles. En effet, chaque concept d'une syntaxe abstraite définie avec Kermeta peut contenir des opérations, le corps de ces opérations étant constitué de ces expressions impératives (package *Behavior*). On peut ainsi définir la structure aussi bien que la sémantique opérationnelle d'un métamodèle, rendant ainsi exécutables les modèles qui s'y conforment. Une machine virtuelle permet d'interpréter le code des opérations définies.

Pour décrire notre sémantique, nous avons ajouté à la syntaxe abstraite, en plus de l'attribut *progress*, des opérations (cf. figure 4.2) dont l'exécution fait évoluer le modèle de procédé.

Les opérations *start*, *setProgression* et *complete* permettent respectivement de démarrer une activité, de changer son taux de progression et de la marquer comme terminée. Ces opérations sont régies par des contraintes de précédence exprimées grâce à des gardes (conditionnelles) utilisant des opérations auxiliaires faisant référence à l'état des activités. Ainsi, *startable* indique si une *WorkDefinition* peut être démarrée en fonction de l'état des *WorkDefinition* qui la précèdent. De ma-

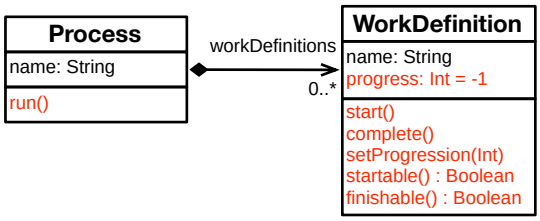


FIGURE 4.2: Métamodèle étendu pour la définition de la sémantique opérationnelle

nière analogue, *finishable* indique si une activité peut être terminée (les activités précédentes de type *finish-to-finish* sont terminées et les activités précédentes de type *start-to-finish* sont commencées). Les structures conditionnelles dans les opérations *start*, *complete* et *setProgression* sont assimilables à des règles de *pattern matching* : elles filtrent les éléments du modèle sur lesquels l'opération s'applique effectivement.

Kermeta introduit un mécanisme d'aspect [Jéz08] permettant de décrire la sémantique opérationnelle comme une extension de la syntaxe abstraite. On obtient alors le code Kermeta donné dans le *listing* 4.2.

Listing 4.2: Sémantique opérationnelle de SIMPLEPDL avec Kermeta

```
@mainClass "SimplePDL::Process"
@mainOperation "run"

package SimplePDL;

require kermeta
require "SimplePDL.kmt"
using kermeta::standard

@aspect "true"
class WorkDefinition {
        attribute progress : Integer

        operation startable() : Boolean is do
            var start_ok : kermeta::standard::Boolean
            var previousActivities : seq WorkDefinition [0..*]
            var prevPrecedes : seq WorkSequence [0..*]

            if progress==-1 then
                // Getting the activities which have to be started
                prevPrecedes := linkToPredecessor.select{p |
                                p.kind == WorkSequenceKind.startToStart}
                previousActivities := prevPrecedes.collect{p | p.predecessor}
                start_ok := previousActivities.forAll{a | a.progress >= 0}
```

```
            // Getting the activities which have to be finished
            prevPrecedes := linkToPredecessor.select{p |
                                p.kind == WorkSequenceKind.finishToStart}
            previousActivities := prevPrecedes.collect{p | p.predecessor}
            start_ok := start_ok
                    and (previousActivities.forAll{a | a.progress==100})
            result := start_ok or (linkToPredecessor.size() == 0)
        else
            result := false
        end
    end

    operation finishable() : Boolean is do
        var finish_ok : kermeta::standard::Boolean
        var previousActivities : seq WorkDefinition [0..*]
        var prevPrecedes : seq WorkSequence [0..*]
        // Activities must be started
        if progress < 100 and progress >= 0 then
            // Testing previous activities
            prevPrecedes := linkToPredecessor.select{p |
                    p.kind == WorkSequenceKind.finishToFinish }
            previousActivities := prevPrecedes.collect{p | p.predecessor}
            finish_ok := previousActivities.forAll{a | a.progress==100}
            result := (finish_ok or linkToPredecessor.size()==0)
        else
            result := false
        end
    end

    operation start() : Void is do
        if startable() then
            progress := 0
        endif
    end

    operation complete() : Void is do
        if finishable() then
            progress := 100
        endif
    end

    operation setProgression(newProgress : Integer) is do
        if (newProgress > progress)
                and (progress > 0) and (progress < 100) then
```

```
                progress := newProgress
            endif
        end
}

@aspect "true"
class Process {
        operation run() : Void is do
                // ...
        end
}
```

L'opération *run* de *Process* simule le démarrage du procédé en positionnant à -1 l'attribut *progress* de toutes les activités. Elle contrôle également l'évolution du processus en intégrant une interface utilisateur sommaire. Elle s'appuie pour cela sur les possibilités de Kermeta d'intégrer du code Java. La personne en charge de gérer le processus peut ainsi *démarrer une activité* sélectionnée parmi les activités éligibles (*startable*), *changer le taux de progression* d'une activité commencée, *terminer une activité* sélectionnée parmi les activités qui peuvent être terminées (*finishable*) ou *stopper* l'exécution.

4.2.2 Mise en œuvre avec ATL

L'intérêt principal de l'utilisation des transformations de modèle endogènes est de pouvoir exprimer la sémantique opérationnelle de manière déclarative, sous la forme par exemple d'un système de transition. La syntaxe abstraite est alors utilisée comme « donnée » des transformations définissant la sémantique.

Rappelons qu'ATL (*ATLAS Transformation Language*) est un langage de transformation de modèle « hybride », déclaratif et impératif. Il permet donc en particulier de définir une transformation par le biais de règles déclaratives (*rule*) pouvant faire appel à des fonctions auxiliaires (*helper*) qui peuvent être récursives.

Contrairement à Kermeta, il n'est dans ce cas pas possible de faire d'interaction avec l'utilisateur. Aussi, nous avons adapté le modèle d'exécution exploratoire : une activité est démarrée et terminée dès que possible et le taux de progression d'une activité commencée est augmenté d'une valeur arbitraire (dans notre cas 33) à chaque évolution. L'évolution d'un modèle est décrit par une succession d'appels à la transformation. Pour conserver le même modèle d'exécution qu'avec Kermeta, il faudrait que l'interface génère un deuxième modèle, dit d'interaction, à prendre en compte dans la transformation pour identifier l'événement à traiter.

La seule partie du modèle qui évolue est la valeur de l'attribut *progress* des activités. En ATL, on peut pour cela utiliser le mode *refine* (défini dans l'entête de la transformation) qui réalise automatiquement une copie du modèle d'entrée et permet de spécifier uniquement les modifications souhaitées sur celui-ci. Cela permet ainsi de définir plus simplement les transformations endogènes.

Pour notre exemple la transformation est constituée de trois règles (*start*, *setProgression* et *complete*) correspondant chacune à une action possible sur une activité (cf. *listing* 4.3), de la même manière que les opérations avec Kermeta. Remarquons que ces actions (et donc ces règles) sont bien exclusives mais que plusieurs activités peuvent évoluer simultanément. Les règles sont appliquées sur les activités du modèle en fonction du filtre (*pattern matching*) défini dans chacune des règles.

Listing 4.3: Sémantique opérationnelle de SIMPLEPDL avec ATL

```
rule start {
  from
    wd_in : SimplePDL!WorkDefinition ( wd_in. startable () )
  to
    wd_out : SimplePDL! WorkDefinition (
        -- start the activity
        progress <- 0 )

rule complete {
  from
    wd_in : SimplePDL!WorkDefinition ( wd_in. finishable () )
  to
    wd_out : SimplePDL! WorkDefinition (
        -- complete the activity
        progress <- 100 )

rule setProgression {
  from
    wd_in : SimplePDL! WorkDefinition(wd_in.progress >= 0
                and wd_in.progress < 90)
  to
    wd_out : SimplePDL! WorkDefinition (
        -- compute new progress value
        progress <- wd_in.newProgress() )
```

Ces règles utilisent trois *helpers* (cf. *listing* 4.4) : *newProgress* calcule une nouvelle valeur du taux de progression (en fonction de l'évolution arbitraire choisie pour l'expérimentation, dans notre cas 33), *startable* et *finishable* sont équivalentes aux opérations définies en Kermeta.

Listing 4.4: *Helpers* pour la sémantique opérationnelle de SIMPLEPDL avec ATL

```
helper context SimplePDL! WorkDefinition
    def : startable () : Boolean = (
        self . progress < 0                          -- not already started
        and self . linkToPredecessor ->select(p | p.kind = # startToStart )
                ->collect(p | p. predecessor )       -- previous start / start
```

```
            ->forAll(a | a.progress >= 0)     -- are started
        and self. linkToPredecessor->select(p | p.kind = # finishToStart )
            ->collect(p | p.predecessor)      -- previous finish / start
            ->forAll(a | a.progress = 100)    -- are finished
    );

helper context SimplePDL! WorkDefinition
    def : finishable () : Boolean = (
        self.progress >= 0 and self.progress < 100
        and self. linkToPredecessor->select(p | p.kind = # finishToFinish )
            ->collect(p | p.predecessor)      -- previous finish / finish
            ->forAll(a | a.progress = 100)    -- are finished
        and self. linkToPredecessor->select(p | p.kind = # startToFinish )
            ->collect(p | p.predecessor)      -- previous start / finish
            ->forAll(a | a.progress >= 0)     -- are started
    );

helper context simplepdl! Activity
        def : newProgress () : Integer = self.progress + 33;
```

Notons que nous avons choisi de privilégier une exécution particulière pour garder un exemple simple. Pour exprimer la sémantique générale, il faut prévoir de piloter la sémantique par un modèle (correspondant aux entrées/sorties de l'utilisateur dans les expérimentations avec Kermeta).

En dehors des travaux sur la théorie des graphes présentés dans la section suivante, nous n'avons pas trouvé de travaux utilisant les transformations endogènes pour exprimer la sémantique opérationnelle d'un DSML. Cette technique s'apparente toutefois aux systèmes de transition utilisés dans les langages de programmation.

4.2.3 Mise en œuvre avec AGG

AGG [AGG07] (*The Attributed Graph Grammar System*) est un langage visuel basé sur les règles et supportant une approche algébrique pour la transformation de graphe [Tae03]. Il permet la spécification et l'implantation de prototypes avec des structures complexes de graphe. AGG peut être utilisé comme un moteur général de transformation de graphe pour des applications Java de haut niveau d'abstraction (comme la génération à base de transformations d'environnements de modélisation, p. ex. Tiger [EEHT05b]).

La combinaison d'une notation graphique, d'une potentielle interprétation intuitive et d'un fondement mathématique rigoureux [Roz97] fait de la transformation de graphe un candidat intéressant pour la spécification d'un comportement à travers des transformations endogènes sur les modèles. De nombreux travaux s'accordent sur ce fait [Pad82, EEKR99, GR01]. Nous avons repris les expérimen-

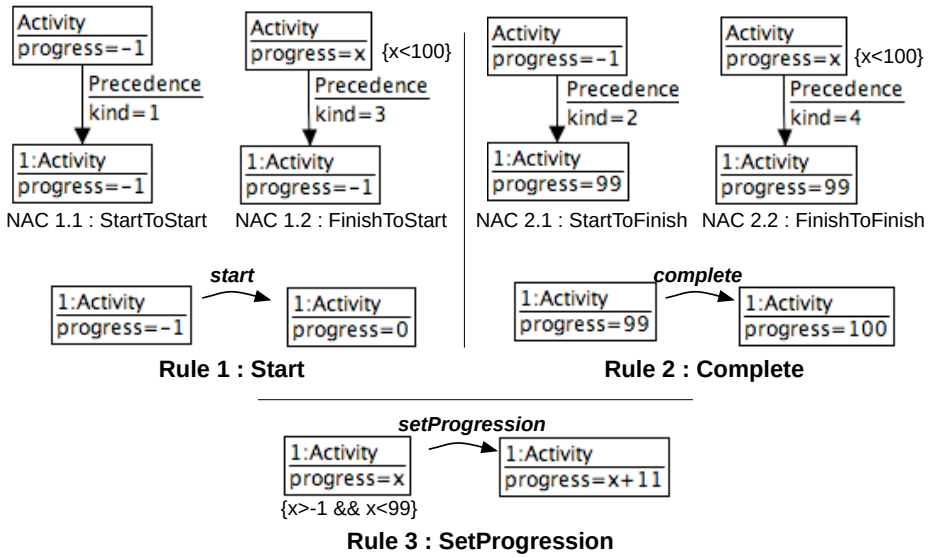


FIGURE 4.3: Sémantique opérationnelle de SIMPLEPDL avec AGG

tations faites pour l'exécution de modèles SIMPLEPDL avec AGG. Pour cela, nous avons suivi le même modèle d'exécution exploratoire que celui retenu pour ATL.

La description en AGG est composée de trois règles similaires aux règles définies en ATL : *Start* initialise une activité qui peut démarrer, *Complete* termine une activité qui peut l'être et *SetProgression* incrémente d'une valeur arbitraire le taux de progression d'une activité démarrée (cf. figure 4.3). Chacune des règles peut être munie de conditions d'application positives (AC - *Application Condition* - exprimée entre accolades sur la figure 4.3) ou négatives (NAC - *Negative Application Condition*, exprimée graphiquement). Par exemple, la règle permettant de démarrer une activité est conditionnée par les activités qui la précèdent. *NAC 1.1* exprime qu'il ne doit pas y avoir d'activité précédente de type *StartToStart* (*kind* = 1) non démarrée (*progress* = -1). *NAC 1.2* exprime qu'il ne faut pas d'activité précédente de type *FinishToStart* (*kind* = 3) non finie (*progress* < 100). AGG offre ainsi une utilisation très facile du principe de *pattern matching*. Les NAC ont permis d'exprimer facilement les différents types de précédence pris en compte dans SIMPLEPDL.

AGG offre ensuite la possibilité d'appliquer les différentes règles sur un graphe particulier selon deux modes : un mode pas-à-pas permettant de choisir les règles à appliquer au fur et à mesure et de visualiser l'interprétation de chaque AC/NAC. Le deuxième mode d'interprétation automatique (*batch*) permet de lancer une simulation en prenant en compte un ensemble de règles à chaque pas. On peut alors exprimer des priorités sur les règles pour contrôler leur application. L'interprétation est faite tant que des règles sont applicables sur le graphe.

Le choix d'AGG pour la modélisation du comportement d'un DSML se révèle

particulièrement intéressant à différents niveaux. Tout d'abord, la syntaxe concrète graphique qu'offre AGG couplée au *pattern matching* naturel dans la théorie des graphes ont permis de modéliser naturellement et simplement la sémantique comportementale de SIMPLEPDL. D'autre part, avoir une grammaire de graphe pour AGG permet de faire certaines validations en utilisant les techniques d'analyse. Entre autres, AGG offre la possibilité de définir des conditions de conformité et de consistance pouvant être évaluées sur un graphe, des techniques d'analyse de paires critiques et des techniques d'analyse des critères de terminaison [EEdL+05].

Les outils de transformation de graphe comme AGG offrent un pont entre l'IDM et la théorie des graphes qui apporte de nombreux avantages. Ce pont est d'ailleurs rendu transparent grâce à des travaux récents qui intègrent le moteur de transformation d'AGG au sein du *framework* EMF d'Eclipse [BEK+06]. Ces travaux permettent ainsi d'utiliser AGG pour exprimer des règles de transformation endogènes directement sur des modèles conformes à des métamodèles décrits en Ecore. Cette approche constitue donc une solution pertinente pour la mise en œuvre d'une sémantique d'exécution d'un DSML par transformations endogènes. Elle est par exemple utilisée dans les travaux de [KKR06a] (et détaillée dans [KKR06b]). L'approche est similaire mais ils utilisent l'outil de transformation de graphe GROOVE [Ren07] qui offre entre autre un simulateur permettant de définir les règles de transformation et de les exécuter sur un graphe donné [Ren04]. Les travaux de Kuske, Gogolla et Ziemann ont également utilisé la transformation de graphe (et la notion d'unité de transformation [Kus00]) pour exprimer la sémantique comportementale de certains diagrammes UML et leurs relations [Kus01, KGKK02, GZK02, ZHG05]. Enfin, on retrouve l'utilisation de la transformation de graphe pour l'expression de la sémantique comportementale dans les travaux de Hausmann [Hau05]. Il introduit la notion de métamodélisation dynamique (*Dynamic Meta Modeling* – DMM) comme une technique pour spécifier la sémantique des langages de modélisation graphiques. Au sein de celle-ci, il utilise la transformation de graphe pour exprimer le comportement d'un système de transition issue d'une traduction à partir d'un modèle.

4.3 Sémantique par traduction : expérimentations

Le principe de la sémantique dénotationnelle est de s'appuyer sur un formalisme rigoureusement (i.e. mathématiquement) défini pour exprimer la sémantique d'un langage donné [Mos90, GS90]. Une traduction des concepts du langage d'origine est alors réalisée vers ce formalisme. C'est cette traduction qui donne la sémantique du langage d'origine.

Dans le cadre de l'IDM, il s'agit d'exprimer des transformations vers un autre espace technique [FEB06], c'est-à-dire définir un pont entre les espaces techniques source et cible. On parle ainsi de sémantique par traduction [CESW04]. Ces ponts technologiques permettent ainsi d'utiliser les outils de simulation, de vérification et d'exécution fournis par l'espace technique cible.

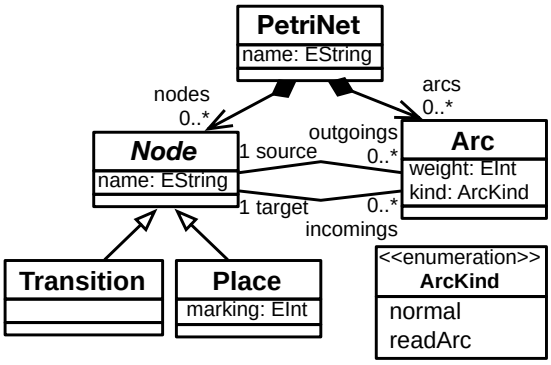


FIGURE 4.4: Syntaxe abstraite des réseaux de Petri

Afin d'illustrer cette approche, nous présentons dans cette partie une traduction des concepts de SIMPLEPDL vers les réseaux de Petri. Nous détaillons les réseaux de Petri temporels à priorités (concepts, sémantique et outils) dans le chapitre 6. Nous proposons toutefois dans la figure 4.4 une syntaxe abstraite très simple utilisée dans les expérimentations suivantes. Un réseau de Petri (*PetriNet*) est composé de noeuds (*Node*) pouvant être des places (*Place*) ou des transitions (*Transition*). Les noeuds sont reliés par des arcs (*Arc*) pouvant être des arcs normaux ou des *read-arcs* (*ArcKind*). Le poids d'un arc (*weight*) indique le nombre de jetons consommés dans la place source ou ajoutés à la place destination (sauf dans le cas d'un *read-arc* où il s'agit simplement de tester la présence dans la place source du nombre de jetons correspondant au poids). Le marquage du réseau de Petri est capturé par l'attribut *marking* d'une place.

Le schéma de traduction pour transformer un modèle de processus en un modèle de réseau de Petri est présenté sur la figure 4.5. Un exemple de modèle de procédé et de réseau de Petri correspondant est donné sur la figure 4.6. La transformation correspondante est écrite en ATL et est détaillée dans un « *Use Case* » que nous avons publié sur le site officiel Eclipse d'ATL [13]. Elle comprend trois règles déclaratives (cf. listing 4.5). La première traduit chaque activité en quatre places caractérisant son état (*NotStarted*, *Started*, *InProgress* et *Finished*) et deux transitions (*Start* et *Finish*). L'état *Started* permet de mémoriser qu'une activité a démarré.

Ces places et transitions sont utilisées dans la deuxième règle pour traduire chaque *WorkSequence* en un *read-arc* entre une place de l'activité source et une transition de l'activité cible, la place et la transition dépendant de la valeur de l'attribut *kind*.

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

13. cf. http://eclipse.org/m2m/atl/usecases/SimplePDL2Tina

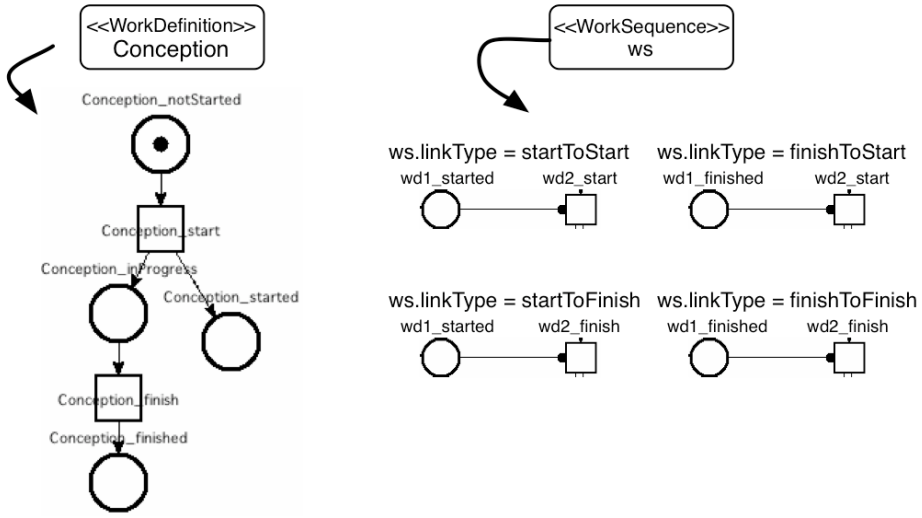


FIGURE 4.5: Schéma de traduction de SIMPLEPDL vers PETRINET

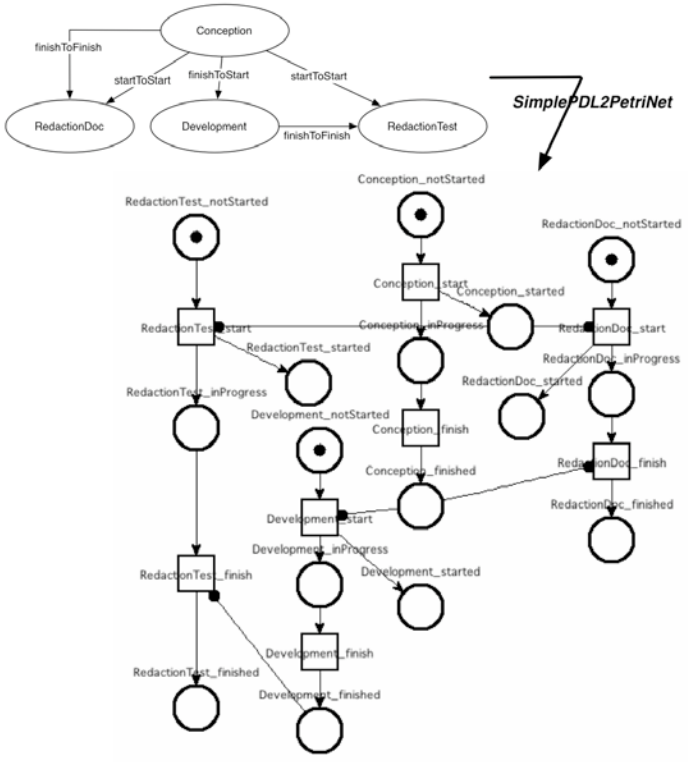


FIGURE 4.6: Exemple de transformation d'un processus en réseau de Petri

Listing 4.5: Extrait de la transformation ATL de SIMPLEPDL vers PETRINET

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

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

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

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

Afin de maîtriser la complexité pour définir une sémantique par traduction et prendre plus facilement en compte l'évolution de la définition d'un langage, Cleenewerck *et al.* [CK07] préconisent de séparer les préoccupations. Ils définissent pour cela la notion de *module logiciel* correspondant à des constructions du langage pour lesquelles la sémantique par traduction est définie. Les différentes constructions (préoccupations) peuvent ensuite être assemblées de manière à définir la sémantique du DSML complet.

Gurevich *et al.* [Gur01] a le même objectif mais avec une approche différente qui consiste à définir des unités formelles dans le langage cible. En utilisant le langage de transformation GReAT (*Graph Rewriting And Transformation language*) [AKK+05], ils proposent un ancrage sémantique (*Semantic Anchoring*) d'unités formelles définies à l'aide d'ASM (*Abstract State Machine*) [Gur01].

Enfin, notons que les techniques de sémantique par traduction ont également été utilisées dans le groupe pUML [14], sous le nom de *Denotational Meta Modeling*, pour la formalisation de certains diagrammes d'UML [CEK01].

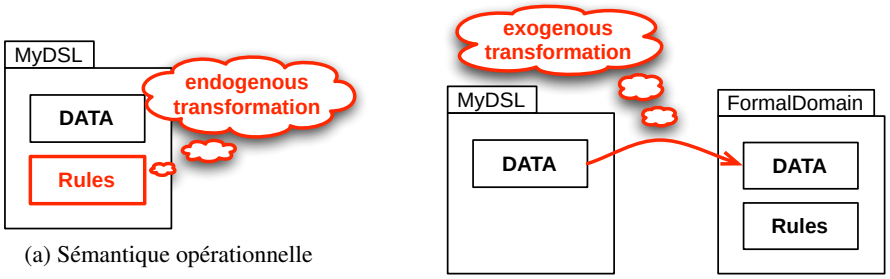
4.4 Discussions et problématiques

Ce chapitre introduit une taxonomie des différents types de sémantique dans l'IDM et des expérimentations sur leurs implantations possibles. Nous discutons dans la suite de cette partie les expérimentations réalisées de manière à introduire les contributions de cette thèse.

4.4.1 Autres taxonomies des sémantiques

Les nombreux travaux sur la sémantique des langages de programmation ont permis d'établir une taxonomie des différentes techniques utilisées pour exprimer

14. *The precise UML group*, cf. http ://www.cs.york.ac.uk/puml/



(a) Sémantique opérationnelle

(b) Sémantique par traduction

FIGURE 4.7: Sémantique opérationnelle Vs. sémantique par traduction

une sémantique selon des besoins différents [Win93]. Cette préoccupation est beaucoup plus récente pour les langages de modélisation.

Les travaux les plus proches des nôtres sont ceux de Clark *et al.* [CESW04], récemment mis à jour dans [CSW08a]. Nous partageons la distinction entre une sémantique opérationnelle et une sémantique par traduction. Ces travaux sont toutefois focalisés sur la sémantique d'exécution et n'évoquent donc pas la sémantique axiomatique. Ils définissent également la notion de sémantique par extension comme une extension des concepts et de la sémantique d'un langage existant. Nous pensons qu'il s'agit d'un moyen de factoriser des descriptions sémantiques mais que la définition de la sémantique elle-même (nouvelle ou existante) se ramène à un des types de notre taxonomie. Enfin, nous ne partageons pas la définition d'une sémantique dénotationnelle, comme un *mapping* vers un domaine sémantique. Il s'agit là pour nous de la définition générale d'une sémantique d'exécution qui généralise l'ensemble des autres types.

Hausmann introduit également dans sa thèse une taxonomie des techniques pour exprimer une sémantique comportementale [Hau05]. Il distingue les techniques propres à des objectifs particuliers (vérification de propriétés, analyse de la consistance ou génération de code) des techniques générales dont le but est d'exprimer la sémantique. Ces dernières distinguent également les techniques de sémantique opérationnelle et celles de sémantique dénotationnelle.

4.4.2 Sémantique opérationnelle Vs. sémantique par traduction

Une sémantique opérationnelle permet de rester dans le même espace technique (c.-à-d. de manipuler les concepts de la syntaxe abstraite avec les mêmes paradigmes) alors qu'une sémantique par traduction, par définition, met en œuvre une traduction vers d'autres paradigmes formellement définis dans l'espace technique cible. Cette différence entraîne un changement radical dans la manière de définir une sémantique d'exécution et nécessite un travail différent. En effet, dans le cas d'une sémantique opérationnelle, on s'attache à définir l'évolution du modèle au cours de son exécution (c.-à-d. la sémantique) à l'aide de transformations

endogènes (cf. figure 4.7a). En revanche, dans le cas d'une sémantique par traduction, il s'agit de construire une équivalence entre les concepts du DSML d'origine et ceux du domaine cible, dont le langage et la sémantique existent déjà. Il s'agit alors de définir une transformation exogène (cf. figure 4.7b).

Pour chacune de ces approches, le langage utilisé pour exprimer la sémantique à travers des règles de transitions (les opérations en Kermeta ou les règles endogènes en ATL et AGG) ou des transformations exogènes (p. ex. en ATL) nécessite de sélectionner les éléments du modèle d'entrée (appelé également *patterns*) pour lesquels la règle doit s'appliquer. Dans le cas des transformations (de modèle, de graphe, etc.), il s'agit alors d'appliquer la règle sur les différents *patterns* que l'on retrouve au sein du modèle. On parle alors de *pattern matching*. Dans le cas d'un langage de métaprogrammation comme Kermeta, la définition d'un *pattern* est plus difficile car les opérations doivent être définies au sein d'un seul concept de la syntaxe abstraite. Il est donc nécessaire de conditionner l'exécution de chaque opération par le *pattern* attendu.

Par ailleurs, aller vers un autre espace technique permet de bénéficier de tous les outils disponibles (p. ex. simulateurs, vérificateurs, etc.). Inversement, dans le cadre d'une sémantique opérationnelle il est nécessaire de développer les outils supports à la sémantique. Malgré tout, dans le cas d'une sémantique par traduction, l'utilisation des outils sur le domaine formel doit rester transparente pour le concepteur. Il est alors indispensable d'interpréter les résultats obtenus dans l'espace cible en terme des concepts présents dans le modèle initial. De plus, la distance sémantique entre le DSML d'origine et le domaine formel choisi exige une transformation plus ou moins complexe et l'utilisation éventuelle de langages intermédiaires.

Telle qu'expérimentée avec Kermeta, ATL ou AGG, une sémantique opérationnelle nous semble plus simple à mettre en œuvre, en particulier parce que, restant dans le même espace technique, les concepts manipulés sont plus proches du domaine métier. Ainsi, dans un objectif d'animation/simulation une approche opérationnelle est préférable, en particulier si le modèle de calcul est simple (p. ex. à événements discrets). Pour de la vérification, une approche par traduction est plus adaptée pour tirer partie des outils de vérification disponibles dans l'espace technique cible.

4.4.3 Modifications du métamodèle

Dans l'ensemble de ces expérimentations, nous avons dû modifier à de nombreuses reprises notre syntaxe abstraite initiale de SIMPLEPDL.

La première modification est la définition des informations (attributs, références ou métaclasses) permettant de capturer chaque état souhaité du modèle au cours des exécutions possibles. Cela correspond dans notre exemple à l'attribut *progress* défini sur les activités. D'autre part, il est intéressant de pouvoir conserver l'ensemble des événements qui ont fait évoluer le système au cours de l'exécution. Cela correspond par exemple aux interactions avec l'utilisateur dans le cas

de Kermeta. Afin de ne pas les définir de manière implicite sous forme de données internes au moteur d'exécution, il faut définir au sein de la syntaxe abstraite les différents événements et le moyen de les conserver tout au long d'une exécution.

Toutes ces modifications sont capitales dans la définition d'un DSML et sont souvent très difficiles à définir de manière pertinente. Il est donc indispensable de fournir des approches rigoureuses et génériques pour aider le concepteur du langage.

4.4.4 Approche outillée pour la définition d'une sémantique

Nous avons montré dans ce chapitre qu'il était possible d'appliquer la taxonomie des techniques pour exprimer une sémantique d'un langage de programmation à l'IDM et donc aux langages de modélisation. Chaque technique répond à des besoins différents sur les modèles. D'autre part, les expérimentations montrent que le domaine de l'IDM et les outils actuellement disponibles sont assez mûrs pour être utilisés dans la description de la sémantique des DSML, sans recourir à un langage de programmation. C'est une étape indispensable qui prouve la pertinence des outils et des concepts introduits par l'IDM.

Malgré tout, la définition de la sémantique d'un langage (de programmation comme de modélisation) reste une tâche compliquée qui, dans un contexte où l'on préconise la définition de petits langages dédiés, est amenée à être répétée de multiple fois. Afin d'aider le concepteur du langage, nous pensons qu'il est donc indispensable d'intégrer l'expression de la sémantique d'un DSML dans une démarche plus globale et outillée. Pour cela, nous introduisons dans la suite de cette thèse une démarche dans laquelle l'expression de la sémantique est fortement couplée à une structuration précise et générique de la syntaxe abstraite.

Par ailleurs, la multiplication des DSML nécessite un effort considérable pour fournir les moyens de valider et vérifier les modèles construits à partir de ces différents DSML. Pour cela, il semble pertinent de profiter de l'abstraction offerte par un DSML, en se concentrant uniquement sur le domaine considéré, afin d'offrir des outils génériques qui s'appuient sur la structure d'un DSML exécutable introduite dans cette thèse.

Enfin, au même titre que pour les langages de programmation, il est important de pouvoir vérifier la cohérence de plusieurs sémantiques d'un même DSML définies pour des besoins différents (simulation, vérification, etc.).

Deuxième partie

Contributions pour la simulation & la vérification de modèle

Chapitre 5

Démarche outillée pour la définition d'un DSML exécutable

La métamodélisation vise à définir les langages de modélisation à partir de modèles, appelés métamodèles. Comme nous l'avons vu dans le chapitre 2, la définition des différentes constructions du langage (sous la forme de concepts et de relations entre eux) est une tâche très proche de celle de la modélisation elle-même, en considérant le langage lui-même comme le système à modéliser.

Dans l'objectif d'exécuter les modèles construits, il est nécessaire d'expliciter la sémantique d'exécution. Les expérimentations présentées au chapitre 4 pour exprimer la sémantique d'exécution d'un DSML ont montré que de nouvelles informations doivent être prises en compte. L'exécution d'un modèle correspond à le faire évoluer d'un état observable du système à un autre, conformément à la sémantique d'exécution définie. Lorsque l'on souhaite exécuter un modèle, il faut capturer les informations qui permettent d'appréhender son évolution. Cela correspond à pouvoir distinguer dans le modèle chacun des états possibles du système et ainsi pouvoir les présenter à l'utilisateur. En plus des informations structurelles du modèle, il est donc nécessaire de compléter la syntaxe abstraite par des informations qui vont évoluer au cours de l'exécution. Par abus de langage, nous parlerons des *informations dynamiques* du modèle. Ces informations doivent être définies au niveau d'abstraction adéquat. En d'autres termes, elles doivent être nécessaires et suffisantes pour répondre aux questions que le concepteur se pose sur le système au cours de son exécution. Nous présentons pour cela, dans la section 5.2, une approche outillée et basée sur le principe de substituabilité des modèles [1] dans laquelle les propriétés comportementales du système sont prises en compte pour déterminer le bon niveau d'abstraction des informations dynamiques.

Par ailleurs, nous nous concentrons dans cette thèse sur une des utilisations possibles de la sémantique d'exécution, qui consiste à pouvoir valider les modèles par simulation ou les vérifier à l'aide des techniques de *model-checking*. En plus de la définition des informations dynamiques et de la formalisation des propriétés

1. Le principe de substituabilité est présenté dans la section 2.1 de cette thèse.

temporelles que l'on souhaite vérifier, d'autres préoccupations sont à prendre en compte au niveau de la syntaxe abstraite. Il s'agit en particulier des différents types d'interaction possibles avec l'utilisateur (gérés par l'interface graphique dans nos expérimentations avec Kermeta) et la séquence particulière qui correspond à une certaine exécution du système. Ces préoccupations nécessitent d'être explicitées et structurées dans la syntaxe abstraite de manière à être capitalisées pour la mise en place d'outils génériques, support à l'exécution des modèles. Nous proposons dans la section 5.3 d'étendre la syntaxe abstraite pour prendre en compte ces nouvelles préoccupations.

Nous décrivons également une architecture générique pour construire de manière structurée une syntaxe abstraite permettant de capturer l'exécution d'un modèle et les informations dédiées à la validation dynamique des modèles.

La démarche est illustrée tout au long de ce chapitre par une des applications principales de nos travaux, la définition d'une extension exécutable de SPEM2.0, appelée XSPEM. XSPEM permet de modéliser un procédé puis de l'exécuter de manière à le vérifier formellement ou l'animer graphiquement. Pour cela, nous avons dans un premier temps sélectionné les informations actuellement définies dans SPEM2.0 qui étaient pertinentes pour l'exécution d'un procédé (section 5.1). Nous présentons ensuite, dans la section 5.2, les moyens de définir au sein de la syntaxe abstraite l'ensemble des états possibles d'un procédé. La section 5.3 détaille les autres préoccupations à prendre en compte pour la validation des modèles. Nous proposons dans la section 5.4 d'organiser l'ensemble de ces préoccupations selon une architecture générique. Enfin, nous discutons les apports de la démarche et de l'architecture générique dans la section 5.5.

> La définition d'un SPEM2.0 exécutable a fait l'objet d'une publication dans la conférence internationale APSEC 2007 [BCCG07]. La démarche pour la définition d'un DSML exécutable a pour sa part été publiée dans la conférence internationale ICEIS 2007 [CGCT07] et a été retenue pour être éditée dans l'ouvrage *Enterprise Information System, vol. IX* [CCG+08a]. La structuration des différentes préoccupations a été publiée à ERTS 2008 [CCG+08b].

5.1 xSPEM, une extension eXécutable de SPEM2.0

Dans l'ensemble du chapitre, nous illustrons la démarche proposée de manière à étendre la spécification de SPEM2.0 et ainsi prendre en compte l'exécution des procédés qui n'est actuellement pas adressée par le standard de l'OMG. Nous appelons notre proposition XSPEM, pour *eXecutable SPEM*. Dans un premier temps, nous présentons dans cette partie les éléments de SPEM2.0 qui jouent un rôle dans l'exécution d'un procédé et que nous réutilisons donc dans XSPEM (section 5.1.1). Nous introduisons également une première extension du standard permettant d'appliquer un modèle de procédé à un projet particulier (section 5.1.2). Enfin, nous illustrons par un exemple de modèle de procédé, les concepts structurels de l'in-

génierie des procédés repris dans XSPEM (section 5.1.3). A l'instar de Kurtev *et al.* [KBJV06], nous appelons DDMM (*Domain Definition MetaModel*) cette partie de la syntaxe abstraite du DSML.

5.1.1 XSPEM *Core*

Les concepts de SPEM2.0 qui ont une influence sur l'exécution d'un procédé sont regroupés dans le paquetage *xSPEM_Core* de la syntaxe abstraite (cf. figure 5.1). Ce paquetage intègre des concepts offerts par les paquetages *Core* et *ProcessStructure* de SPEM2.0 (en gris sur la figure 3.2 présentée dans la section 3.2). Une activité (*Activity*) est une définition de travail (*WorkDefinition*) concrète qui représente une unité de travail générale et assignable (*ProcessPerformerMap*) à des ensembles de compétences représentés par des rôles (*RoleUse*). Une activité peut s'appuyer (*ProcessParameter*) sur des produits (*WorkProductUses*) en entrée ou en sortie (attribut *direction*). Elle peut également être composée (*nestedBreakdownElement*) de *BreakdownElement*, métaclasse abstraite généralisant le concept d'activité (permettant ainsi de définir un procédé hiérarchique) et le concept de ressource (*RoleUse* et *WorkProductUse*). Ces éléments peuvent être caractérisés par les attributs *isOptional* et *hasMultipleOccurrences*. Le premier indique que l'activité n'est pas obligatoire, c.-à-d. que l'activité parente peut se terminer sans que la sous-activité optionnelle n'ait été réalisée. Dans le cas d'une ressource, cela signifie qu'une activité peut être réalisée même si cette ressource n'est pas disponible. L'attribut *hasMultipleOccurrences* indique qu'une activité peut avoir plusieurs occurrences. Dans ce cas, toutes les occurrences doivent être réalisées en parallèle, avec des ressources différentes. Cet attribut indique pour un rôle ou un produit s'il est également possible d'en avoir plusieurs occurrences.

Les activités, en tant que spécialisation de *WorkBreakdownElement*, peuvent également être liées par des contraintes d'ordonnancement (*WorkSequence*) dont l'attribut *linkKind* indique quand une activité peut être démarrée ou terminée. Par exemple, une *WorkSequence* définie comme *finishToStart* implique que l'activité suivante ne pourra démarrer que quand l'activité précédente sera terminée.

L'attribut *isRepeatable* indique qu'une activité peut être réalisée plusieurs fois. A la différence de l'attribut *hasMultipleOccurrences*, il ne s'agit pas de plusieurs occurrences de la même activité qui sont réalisées en parallèle, mais la même activité qui pourra être réalisée plusieurs fois successivement. Une activité qui peut être répétée peut donc être redémarrée quand elle a été accomplie. Nous distinguerons donc l'état accompli (*completed*) de l'état terminé (*finished*) qui indique qu'elle ne peut plus être redémarrée.

L'attribut *isOngoing* indique que l'activité devra être réalisée tout au long de l'exécution du procédé. Elle démarrera quand le procédé démarrera et s'arrêtera quand le procédé se terminera.

Comme SPEM2.0, XSPEM ne définit pas explicitement la notion de procédé. Un procédé est implicitement défini par l'activité (unique) au sommet de la hiérarchie.

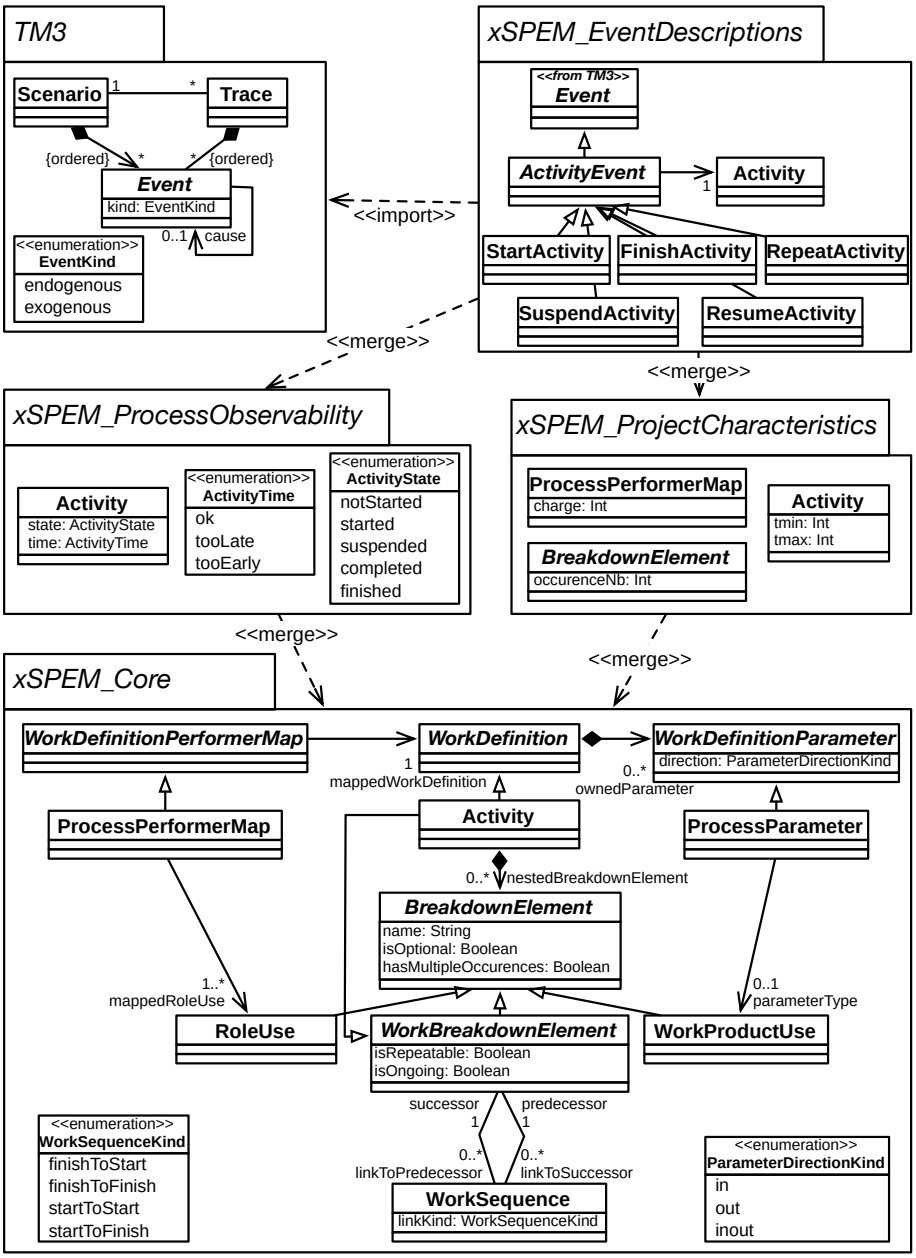


FIGURE 5.1: Syntaxe abstraite de XSPEM

D'autre part, dans le cas de XSPEM nous considérerons que *RoleUse* et *WorkProductUse* peuvent être considérés comme des ressources requises pour réaliser une activité. L'attribut *direction* défini dans *WorkDefinitionParameter* peut être utilisé pour compléter les contraintes d'ordonnancement des activités exprimées à travers le concept de *WorkSequence*. Pour des raisons de simplification, il ne sera donc pas détaillé dans la suite de ce chapitre.

5.1.2 XSPEM *ProjectCharacteristics*

L'application d'un procédé à un projet particulier nécessite de compléter (de raffiner) le procédé avec des informations spécifiques au projet. Cela inclut par exemple la définition des propriétés spécifiques au dimensionnement des activités, c.-à-d. pour chaque activité le nombre de ressources nécessaires, la durée estimée, etc. Il faut également définir le nombre total de ressources allouées au projet.

Actuellement, ces informations sont généralement définies au sein d'un outil de planification (p. ex. *MS Project*[2]) dans lequel il est préalablement nécessaire de ressaisir l'ensemble du procédé. Le chef de projet est alors face à deux formalismes : le procédé décrit en SPEM à partir duquel il peut obtenir l'ensemble des informations générales sur le procédé, et celui de l'outil à partir duquel il peut réaliser la gestion de ses ressources et du planning.

Nous proposons de définir dans la syntaxe abstraite de XSPEM les informations permettant de faire la gestion des ressources et du planning. Ces informations seront donc renseignées sur le modèle de procédé. En plus de regrouper les informations au sein d'un seul formalisme et éviter ainsi les problèmes d'incohérence, cette extension offre la possibilité de prendre en compte l'expérience issue de la réalisation d'un projet au niveau du procédé général.

Dans XSPEM, les informations spécifiques à chaque projet sont introduites dans le paquetage *xSPEM_ProjectCharacteristics*, défini comme une extension du paquetage *xSPEM_Core* en utilisant l'opérateur *merge* de l'OMG [OMG06b, §7] (cf. figure 5.1). Il redéfinit les concepts *Activity*, *BreakdownElement* et *ProcessPerformerMap* en ajoutant :

- l'intervalle de temps au cours duquel l'activité devra se terminer (*tmin* et *tmax* sur *Activity*) ;
- le nombre d'occurences (*occurrencesNb* sur *BreakdownElement*) permettant de définir :
  - dans le cas d'une activité, le nombre d'instances de celle-ci pour le projet particulier ou l'activité englobante,
  - dans le cas d'un rôle, le nombre d'occurrences allouées au projet,
  - dans le cas d'un produit, le nombre d'instances nécessaires dans le projet.
- la charge de travail affectée à un role pour une activité (*charge* sur *ProcessPerformerMap*).

2. cf. http://office.microsoft.com/project

5.1.3 Exemple : modèle XSPEM de la méthode MACAO

Nous illustrons XSPEM à travers un exemple de modèle (cf. figure 5.2) qui propose une modélisation de la *Méthode d'Analyse et de Conception d'Applications orientées-Objet* (MACAO) [Cra02].

MACAO est une méthode complète de génie logiciel qui s'appuie sur la notation et les modèles UML. Elle se base sur UP (*Unified Process*) et se place ainsi au même niveau que le RUP (*Rational Unified Process*) proposé par IBM [KK03].

MACAO s'en différencie cependant sur de nombreux points et propose :

- une approche utilisateur proche des méthodes systémiques telle que MERISE, et basée sur les cas d'utilisation d'UML,
- une démarche itérative de conception-développement, basée sur la réalisation de prototypes soumis aux utilisateurs pour validation (réduction des risques),
- une règle de non-régression des prototypes afin de limiter les propagations de bugs liés à des modifications intempestives de prototypes déjà recettés,
- une documentation type pour chaque étape du déroulement du projet,
- trois modèles d'IHM [3] : le Schéma Navigationnel d'Interfaces (SNI), le Schéma d'Enchaînement des Fenêtres (SEF) pour les IHM de type Windows et le Schéma d'Enchaînement des Pages (SEP) pour les IHM de type WEB,
- des patrons de conception de l'IHM s'appuyant sur le diagramme des classes métiers.

Ces concepts sont définis au sein d'un procédé de développement complet composé de quatre phases ordonnées et présentées dans la figure 5.2 : l'analyse globale (*GlobalAnalisys*), la conception globale (*GlobalDesign*), le développement (*Development*) et la finalisation (*Finalization*). L'ensemble du projet est géré par le chef de projet à travers une activité de gestion de projet (*ProjectManagement*) qui est active tout au long du procédé (*isOngoing*). L'ordre entre les activités est décrit par les relations de précédence.

Notons que la phase de développement est elle-même composée d'un cycle de vie en spirale composé des phases de définition, conception détaillée, codage, intégration et *beta*-test. Chaque itération au sein de ce cycle de vie donne lieu à la création d'un prototype opérationnel recetté par le client (phase de *beta*-test). Cette démarche permet d'avoir très rapidement un retour de la part des futurs utilisateurs et de pouvoir ainsi rectifier très tôt les erreurs de conception.

La figure 5.2 propose un exemple d'application de la méthode MACAO sur un projet particulier et définit donc les contraintes de temps (entre crochet) et de ressources (pour des raisons de lisibilité, nous ne faisons pas apparaître les produits sur la figure).

3. Interface Homme-Machine

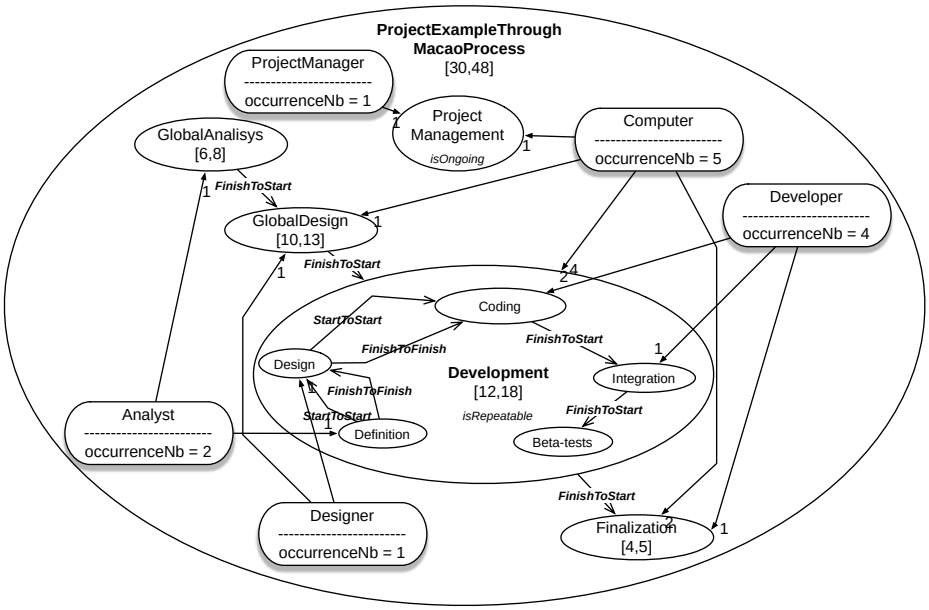


FIGURE 5.2: Modèle XSPEM de la méthode MACAO pour un projet particulier

5.2 Approche dirigée par les propriétés pour la définition des informations dynamiques

Pour exécuter un modèle il est nécessaire de compléter la syntaxe abstraite avec les informations dynamiques qui permettent de capturer les états possibles du système (du procédé dans notre cas) au cours de l'exécution. Celles-ci doivent être définies à un niveau d'abstraction adéquat pour répondre au principe de substituabilité qui est au centre d'une approche dirigée par les modèles et respecter ainsi la définition d'un modèle [Min68]. En effet, un modèle est défini comme une abstraction nécessaire et suffisante pour répondre aux questions que le concepteur se pose sur le système. Il semble donc pertinent de prendre comme point de départ les questions auxquelles les utilisateurs souhaitent répondre pour définir le bon niveau d'abstraction des informations nécessaires pour y répondre.

Nous proposons pour cela une approche outillée qui vise à spécifier au niveau du métamodèle les propriétés temporelles que l'on souhaite vérifier au sein des modèles au cours de leurs exécutions. Ces propriétés permettent d'identifier les états du système que l'on souhaite observer, et ainsi de déterminer la bonne granularité (c.-à-d. abstraction) des informations dynamiques nécessaires dans la syntaxe abstraite.

Nous illustrons cette approche en montrant comment compléter la syntaxe abstraite de *xSPEM* avec les informations dynamiques obtenues à partir d'un ensemble donné de propriétés.

5.2.1 Caractérisation des propriétés temporelles

Le premier travail, à faire par l'expert du domaine, consiste à exprimer les propriétés temporelles attendues sur le système au cours de son exécution. Elles viennent compléter les propriétés structurelles définies au niveau de la syntaxe abstraite et qui peuvent être vérifiées statiquement. Par définition, les propriétés temporelles permettent de spécifier l'évolution d'une exécution (ou de toutes les exécutions). Leur évaluation nécessite donc de pouvoir exécuter le modèle.

Nous avons identifié certaines propriétés que tout modèle XSPEM doit vérifier. Nous les avons séparées en deux classes : les *propriétés universelles* qui doivent être vérifiées pour chaque exécution possible du procédé :

- toute activité non optionnelle doit démarrer,
- toute activité suspendue doit redémarrer,
- toute activité démarrée doit s'accomplir,
- toute activité accomplie peut redémarrer mais doit finir par se terminer,
- quand une activité est terminée, elle doit rester dans cet état,
- une activité peut démarrer ou terminer à condition de respecter les contraintes d'ordonnancement,
- ...

Les *propriétés existentielles* doivent être vraies pour au moins une des exécutions possibles du procédé :

- chaque activité doit être réalisée dans l'intervalle de temps *tmin* et *tmax*, sauf si elle est optionnelle,
- l'ensemble du procédé doit se terminer correctement, c.-à-d. que toutes ses activités obligatoires sont réalisées dans les temps, c.-à-d.entre *tmin* et *tmax*,
- ...

Toutes les propriétés ne sont, bien entendu, pas mentionnées ici. Des propriétés similaires ont été définies pour affiner la sémantique des différents attributs d'une activité, d'un produit et d'un rôle. Par exemple, une activité ne peut démarrer que si chaque produit qu'elle a en entrée est commencé (au moins une activité qui le réalise a démarré) ou terminé (toutes les activités qui doivent le construire sont terminées). Il pourrait également être intéressant de savoir s'il est possible d'achever tous les produits d'un procédé.

5.2.2 Caractérisation des états possibles

A partir des propriétés identifiées dans la partie précédente, nous pouvons déduire le bon niveau d'abstraction des informations permettant de les évaluer et qu'il est nécessaire d'ajouter à la syntaxe abstraite.

Ainsi, pour XSPEM, nous identifions deux aspects orthogonaux pour le concept *Activity*. Premièrement, une activité peut être non démarrée, démarrée, suspendue, accomplie et finalement terminée. Deuxièmement, il y a une notion de temps et d'horloge associée à chaque activité ; mais ce temps n'est pertinent que pour les conditions de franchissement des transitions (dans notre cas, la transition qui dé-

marre, suspend, reprend, répète ou termine une activité) et n'est pas explicite dans les propriétés d'états. Cet aspect peut donc être représenté dans l'ensemble fini d'états {*tooEarly*, *ok*, *tooLate*}. Il n'est pertinent que quand l'activité est terminée.

5.2.3 Extension de la syntaxe abstraite : xSPEM *ProcessObservability*

Maintenant, nous caractérisons l'ensemble de ces états en étendant la syntaxe abstraite de xSPEM et plus particulièrement ici l'élément *Activity* pour lequel nous introduisons les attributs nécessaires pour refléter les informations dynamiques. Nous avons choisi d'ajouter deux attributs, *state* et *time*, sur la métaclasse *Activity* (paquetage *xSPEM_ProcessObservability* sur la figure 5.1) :

$state \in \{notStarted,\ started,\ suspended,\ completed,\ finished\}$
et
$time \in \{tooEarly,\ ok,\ tooLate\}$

Nous appelons SDMM (*States Definition MetaModel*) cette partie de la syntaxe abstraite du DSML qui capture les informations dynamiques caractérisant l'état du modèle à l'exécution.

5.2.4 Formalisation des propriétés temporelles avec TOCL

Maintenant que nous avons étendu la syntaxe abstraite avec les informations dynamiques, nous pouvons formaliser les propriétés temporelles définies dans la partie 5.2.1 sur la syntaxe abstraite elle-même. Celles-ci devront être vérifiées dans le cadre de la validation dynamique du modèle. Afin de les exprimer directement sur la syntaxe abstraite, nous avons besoin d'un langage qui s'appuie sur les concepts de métamodélisation. Nous présentons dans un premier temps les langages existants. Nous formalisons ensuite les propriétés décrites pour xSPEM avec TOCL (*Temporal OCL*) [ZG02, ZG03], le langage retenu, et montrons comment il peut s'intégrer à un langage de métamodélisation comme MOF.

Choix du langage pour la spécification des propriétés temporelles

OCL a été introduit par l'OMG afin de spécifier les propriétés structurelles sur des modèles. Il est devenu le langage standard pour exprimer des propriétés structurelles sur les modèles UML. Il existe aussi de nombreux outils permettant de vérifier ces propriétés. Pour la spécification de propriétés temporelles, des travaux plus récents proposent d'étendre la syntaxe et la sémantique usuelle d'OCL pour prendre en compte des opérateurs temporels. Tous ces travaux sont réalisés dans un contexte UML et n'abordent pas la manière dont le système de transition peut être dérivé du modèle.

Dans le cadre de notre état de l'art, nous avons plus particulièrement étudié les travaux de [ZG02, ZG03] qui proposent *TOCL*, pour *Temporal OCL*, une extension d'OCL avec les opérateurs de la logique temporelle linéaire. Ils définissent la sémantique à partir d'une sémantique de trace du modèle UML. Ce travail a été

implanté pour l'évaluation de propriétés temporelles sur les traces dans l'outil USE [USE07].

D'autres travaux, comme [Fla03] et [FM03], se sont focalisés sur l'expression de contraintes temps-réel en gardant la syntaxe initiale d'OCL. Ils s'appuient sur des machines à états pour exprimer les contraintes dynamiques du système. Ils proposent de traduire leurs contraintes en *Clocked-CTL*.

Dans [CK02], les auteurs proposent d'exprimer les contraintes temps-réel en utilisant deux nouvelles classes, *Time* et *Events*. Un nouveau *template* OCL est également introduit afin d'y traduire les contraintes usuelles (pre-, post-, inv et action). La sémantique est là aussi définie comme une sémantique de trace.

Dans [DKR00], les auteurs expriment les contraintes OCL non temporelles dans une version orienté-objet de CTL. Ils ont défini formellement l'état d'un modèle UML et sont en mesure de vérifier si une propriété exprimée en OCL peut être vérifiée dans chaque état accessible du système.

Les travaux de [BFS02] introduisent également un nouveau *template* OCL. Ils traduisent les contraintes en $\mathcal{O}\mu(OCL)$-calculus, un μ-calcul observationnel dont les expressions sont des expressions OCL. La sémantique de $\mathcal{O}\mu(OCL)$-calculus est définie à partir des états de [DKR00]. En utilisant les outils de vérification de modèle (*model-checking*), les auteurs peuvent vérifier les propriétés sur un terme CCS qui modélise le modèle UML.

Tous ces travaux présentent le moyen d'étendre OCL pour spécifier des contraintes temporelles afin de pouvoir les vérifier ou les simuler plus tard, mais ne présentent pas cette dernière étape. D'autre part, ces différentes approches ont toutes été introduites pour UML.

Afin de formaliser les propriétés temporelles définies pour XSPEM, nous avons choisi TOCL (la proposition de P. Ziemann & M. Gogolla) pour deux raisons principales :

1. La sémantique des expressions temporelles est formellement définie à partir d'une sémantique de trace. Chaque trace est une séquence finie d'états du système, « capturant » le système en cours d'exécution. Même si ce travail a initialement été défini sur des modèles UML, la sémantique de trace peut facilement être généralisée à une séquence d'états arbitraires tout en conservant la sémantique initiale des opérateurs temporels.
2. La syntaxe de cette extension d'OCL est particulièrement naturelle. Elle introduit aussi bien les opérateurs temporels classiques du futur comme *next*, *existsNext*, *always*, *sometimes* que les opérateurs duaux pour le passé. Nous nous concentrerons dans la suite de notre travail sur la manipulation des opérateurs temporels portant sur les états futurs du système.

Le langage TOCL

Pour un modèle XSPEM, l'état d'un procédé est décrit précisément à partir de l'état de chacune des activités. Nous définissons l'ensemble fini $\mathcal{S}$ de ces états. Soit

$\mathcal{A}$ l'ensemble des activités (c.-à-d. instance de *Activity*) du modèle. Soit également Σ l'ensemble décrivant l'état du procédé : $\Sigma = \mathcal{A} \mapsto \mathcal{S}$. Notons que dans notre étude, l'ensemble $\mathcal{A}$ est fixe au cours de l'exécution. Nous ne considérons donc pas la création et la suppression dynamique d'objets (c.-à-d. d'activités, dans notre cas). Nous reviendrons sur cet aspect dans le dernier chapitre.

Une trace $\hat{\sigma} \in \Sigma^*$ du procédé est une séquence finie maximale des états du procédé $\langle \sigma_0, \ldots, \sigma_n \rangle$, où $\sigma_i \in \Sigma$ et σ_0 spécifie l'état initial du procédé. Sémantiquement, nous avons deux types de transitions. Premièrement, les transitions de passage du temps continu qui sont ici inobservables et qui consistent à incrémenter simultanément toutes les horloges des activités par une quantité dt. Deuxièmement, les transitions basées sur les événements qui changent l'état des activités en cours tel que défini par l'expert dans la figure 5.4. Deux états consécutifs d'une exécution sont liés par une combinaison de transition de passage du temps, suivie d'une transition basée sur les événements. Par exemple, considérons une activité $A1$. Quand elle est démarrée (événement *StartActivity*), elle est dans l'état $< started, ok, 0 >$. Son horloge interne va ensuite évoluer en fonction des transitions de passage de temps. Enfin, l'événement *FinishActivity* fera évoluer l'état de l'activité en fonction de la valeur de l'horloge interne.

Notons que cette sémantique de trace peut facilement être réutilisée pour étendre TOCL par la définition de nouveaux opérateurs. Par exemple, pour faciliter la définition de nos propriétés liées à l'ordonnancement, nous introduisons un nouvel opérateur *precedes*. Cet opérateur peut être décrit en utilisant les opérateurs précédents :

$$e_1 \textit{ precedes } e_2 = \textit{always}!(e_2) \textit{ until } e_1$$

Cet opérateur peut formellement être défini par la sémantique de trace suivante :

$$I[\![e_1 \textit{ precedes } e_2]\!](\hat{\sigma}, i) = \begin{cases} \text{true} & \text{if } I[\![e_1]\!](\hat{\sigma}, i) = \text{ false } \forall j | j \in [i, l[\\ & \text{with } l = \min (\{k \geq i | I[\![e_2]\!](\hat{\sigma}, k) = \text{ true}\} \\ & \qquad \cup \{ |\hat{\sigma}|\}) \\ \text{false} & \text{otherwise.} \end{cases}$$

Les expressions sont évaluées dans un environnement composé d'une séquence $\hat{\sigma}$ d'état, et d'un indice de référence i dans la trace. La sémantique d'une expression e est définie récursivement par la fonction $I[\![e]\!] : \Sigma^* \times \mathbb{N} \rightarrow \text{boolean}$.

Les expressions de notre extension de TOCL sont maintenant des expressions OCL sur les éléments d'un modèle utilisant ces opérateurs temporels. Il est donc possible de définir ces expressions à l'aide des noms d'états définis précédemment dans l'ensemble $\mathcal{S}$.

Formalisation en TOCL des propriétés sur xSPEM

Nous pouvons maintenant reprendre les propriétés temporelles exprimées précédemment afin de les formaliser en fonction de la syntaxe abstraite complétée par les informations dynamiques. Par exemple, pour les propriétés universelles suivantes :

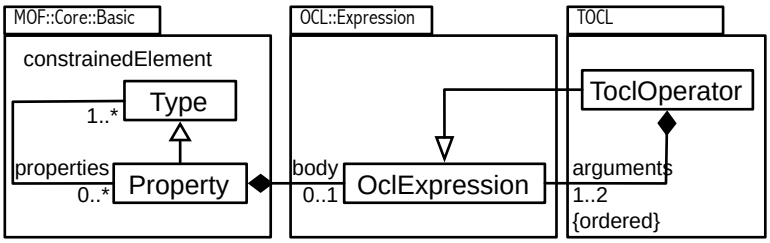


FIGURE 5.3: Promotion de *TOCL* au niveau du MOF

– toute activité a non optionnelle doit démarrer :
  always $((\mathrm{a.state} = \mathrm{notStarted})\ \mathrm{and}\ (\mathrm{not\ a.isOptional})$
  $\Longrightarrow$ *sometime* $(\mathrm{a.state} = \mathrm{started}))$,
– toute activité a démarrée doit s'accomplir :
  always $((\mathrm{a.state} = \mathrm{started}) \implies$ *sometime* $(\mathrm{a.state} = \mathrm{completed}))$,
– quand une activité a est terminée, elle doit rester dans cet état :
  $(\mathrm{a.state} = \mathrm{finished}) \implies$ *always* $(\mathrm{a.state} = \mathrm{finished})$,
– ....

Pour une propriété existentielle, nous vérifions sa négation. Si l'analyse donne une réponse positive, il n'y a pas de trace qui satisfasse la propriété. Au contraire, si l'analyse donne une réponse négative, la propriété existentielle est vérifiée et le contre-exemple identifié est une des traces qui satisfait la propriété temporelle. Par exemple, nous formalisons le fait qu'une activité a doit terminer dans l'intervalle de temps *tmin* et *tmax* de la manière suivante :

$$always\ not\ (a.time = ok)\ \equiv\ always\ (a.time = tooEarly\ ||\ a.time = tooLate)$$

Intégration de TOCL dans une approche de métamodélisation

Nous avons illustré ci-dessus la syntaxe concrète textuelle et la sémantique associée de notre extension de TOCL. Afin de l'intégrer maintenant dans une approche de métamodélisation, et donc de pouvoir définir les propriétés au niveau du métamodèle, il est nécessaire de définir sur le langage de niveau M3 (p. ex. le MOF [OMG06b]) la syntaxe abstraite d'OCL et de l'extension temporelle TOCL. Pour offrir la possibilité d'utiliser TOCL dans la définition de n'importe quel DSML, nous partons de la syntaxe abstraite d'OCL définie dans [RG99] et définissons sa promotion (structurelle) au niveau du MOF (paquetage *OCL : :Expression* sur la figure 5.3). Nous ajoutons également l'ensemble des opérateurs temporels définis dans TOCL et dans l'extension définie ci-dessus (paquetage *TOCL* sur la figure 5.3). TOCL peut ainsi facilement s'implanter, par exemple dans l'environnement Eclipse [Ecl07], comme une extension du projet EMF [EMF07] fournissant le langage de métamodélisation Ecore [BSE03].

Nous avons maintenant introduit les syntaxes abstraite et concrète ainsi que la sémantique de notre extension d'OCL pour l'expression de propriétés temporelles. Nous sommes maintenant en mesure d'exprimer des propriétés complexes sur le comportement du modèle à vérifier.

En plus des propriétés dynamiques exprimées ci-dessus en TOCL, une autre application immédiate de cette extension est la transformation de chaque invariant défini avec OCL comme une propriété universelle. Il est ainsi possible de vérifier que chaque invariant est vérifié dans tous les états de la trace d'exécution du procédé. Par conséquent, nous transformons l'ensemble des invariants e en *always* e.

5.3 Explicitation du comportement de référence

Maintenant que nous avons identifié les états à travers les informations dynamiques et avons formalisé les propriétés, nous proposons dans cette partie d'exprimer la sémantique de référence. Celle-ci peut alors être directement exprimée sur la syntaxe abstraite du DSML, qui correspond aux concepts du domaine, et est donc proche de l'expérience acquise par les experts. Nous proposons de représenter cette sémantique de référence par un ensemble de règles de transition, étiquetées par les événements qui déclencheront les changements de transition. Ces événements correspondent en particulier aux différentes interactions possibles entre le système et l'environnement (utilisateurs, capteurs, etc.) qui avaient été prises en compte de manière *ad-hoc* dans l'interface d'administration mise en place dans nos expérimentations avec Kermeta (cf. chapitre 4). Nous pensons qu'il est donc souhaitable d'expliciter au préalable ces interactions au niveau de la syntaxe abstraite en définissant les différents types d'événements possibles.

Il s'agit donc de caractériser les événements (section 5.3.1) et la sémantique de référence (section 5.3.2), ces activités pouvant être réalisées conjointement. Pour décrire des exécutions particulières, nous définissons également un métamodèle générique permettant de définir les scénarios correspondants (sous la forme d'une séquence d'événement) et la trace des résultats de l'exécution associée (section 5.3.3).

5.3.1 Extension de la syntaxe abstraite : XSPEM *EventDescriptions*

Les états du modèle ont été capturés à travers les éléments rajoutés dans la syntaxe abstraite du métamodèle, et plus particulièrement au sein du SDMM (pour XSPEM, le paquetage *xSPEM_ProcessObservability*). Les changements d'état sont provoqués par des événements qui font évoluer le système (dans notre cas, le procédé). Les événements peuvent être exogènes (produits par l'environnement du procédé) ou endogènes (produits par l'exécution du procédé). Par exemple, l'événement *StartActivity* appliqué sur une activité la fait passer de l'état *notStarted* à l'état *started*. Les autres événements sur une activité sont *repeat*, *finish*, *suspend* et *resume*. Ils sont définis dans le paquetage *xSPEM_EventsDescription* comme

Let a be the considered activity.

$$\forall ws \in a.predecessor,\\ (ws.linkKind = startToStart \;\&\&\; ws.linkToPredecessor.state = started)\\ ||(ws.linkKind = finishToStart \;\&\&\; ws.linkToPredecessor.state = finished)$$

$$notStarted, ok, clock \xrightarrow{StartActivity} started, ok, 0$$

$$started, ok, clock \xrightarrow{SuspendActivity} suspended, ok, clock$$
$$suspended, ok, clock \xrightarrow{ResumeActivity} started, ok, clock$$
$$completed, ok, clock \xrightarrow{RepeatActivity} started, ok, clock$$
$$started, ok, clock \xrightarrow{CompleteActivity} completed, ok, clock$$

$$\forall ws \in a.predecessor,\\ (ws.linkKind = startToFinish \;\&\&\; ws.linkToPredecessor.state = started)\\ ||(ws.linkKind = finishToFinish \;\&\&\; ws.linkToPredecessor.state = finished)$$

$$completed, ok, clock < tmin \xrightarrow{FinishActivity} finished, tooEarly, clock$$
$$completed, ok, clock \in [tmin, tmax] \xrightarrow{FinishActivity} finished, ok, clock$$
$$completed, ok, clock > tmax \xrightarrow{FinishActivity} finished, tooLate, clock$$

FIGURE 5.4: Règles de transition (basé sur les événements) pour les activités

spécialisation de *ActivityEvent*, un événement abstrait qui fait évoluer une activité.

Nous appelons EDMM (*Events Definition MetaModel*) cette partie de la syntaxe abstraite du DSML qui décrit les événements.

5.3.2 Définition du système de transition du DSML

Étant donné la caractérisation des états et l'identification des événements, une abstraction observationnelle de la sémantique opérationnelle de nos procédés peut maintenant être définie par un système de transition respectant les propriétés caractérisées dans la partie 5.2.1. Pour cela, l'expert doit définir l'état initial et la relation de transition. Dans notre cas l'état initial est $\{a \mapsto (notStarted,\ ok) | a \in \mathcal{A}\}$ où $\mathcal{A}$ est l'ensemble de toutes les activités. La relation de transition est définie pour *Activity* dans la figure 5.4. Nous nous appuyons sur une notion d'horloge ($clock \in \mathbb{R}^+$), interne à chaque activité et permettant de formaliser les changements d'état du système. Cette horloge n'est toutefois pas représentée dans la syntaxe abstraite parce que seule son abstraction est nécessaire (attribut $time$), l'horloge étant prise en compte par le moteur d'exécution. L'expression par l'expert de la sémantique observationnelle permet d'une part d'établir le lien entre l'horloge et son abstraction définie par l'attribut $time$ et d'autre part de faire apparaître les événements qui permettent de changer d'état le système. Par exemple, l'événement *StartActivity* permet de faire passer une activité de l'état *notStarted* à l'état *started*. Cela correspond à positionner l'attribut $state$ à *started* et à initialiser l'horloge à 0.

5.3.3 Métamodèle pour la gestion des traces et scénarios

Enfin, nous complétons notre métamodèle par une quatrième partie capturant l'ensemble des événements d'un scénario ou d'une trace. Cette partie est générique (c.-à-d. ne dépend pas de la définition du DSML en question car elle s'appuie uniquement sur une notion abstraite d'événement) et peut donc être importée dans la définition de chaque DSML. Nous appelons cette partie TM3, *Trace Management MetaModel*, qui définit les concepts suivants (cf. figure 5.1) :

- *Event* : un événement qui peut être exogène (il vient de l'environnement) ou endogène (il est produit par le système lui-même, généralement en réaction à un événement précédent, c.-à-d. la $cause$).
- *Trace* : décrit un ensemble ordonné d'événements, aussi bien exogènes qu'endogènes qui sont survenus au cours d'une exécution particulière. Une trace permet ainsi de pouvoir « rejouer » à l'identique cette exécution.
- *Scenario* : il décrit un ensemble ordonné d'événements exogènes. Un scénario pourra donc potentiellement donner lieu à différentes exécutions (et donc différentes traces) si le système est indéterministe.

Pour chaque DSML considéré, les événements décrits dans son EDMM sont en fait des spécialisations du concept *Event* défini dans TM3. Les événements de ce DSML peuvent ainsi être enregistrés dans une trace afin de garder les étapes de l'exécution d'un procédé. Ils peuvent aussi être utilisés pour construire un scénario qui pourra guider la simulation d'un modèle.

5.4 Architecture générique pour la définition d'un DSML exécutable

Les expérimentations menées dans le chapitre 4 ont permis de mettre en évidence le besoin d'étendre la syntaxe abstraite pour prendre l'exécution de modèle. Dans ce chapitre nous avons mis en évidence quatre aspects différents de la syntaxe abstraite d'un DSML exécutable, trois sont spécifiques du DSML considéré et le quatrième est générique. Le modèle de la syntaxe abstraite d'un DSML exécutable (M_{xAS}) est donc défini de la manière suivante :

$$M_{xAS} = \{\text{DDMM}, \text{SDMM}, \text{EDMM}\} \cup \{\text{TM3}\}$$

En effet, nous avons dans un premier temps défini pour xSPEM les concepts et relations du domaine (c.-à-d. l'ingénierie des procédés) qui permettent structurellement de définir les constructions du DSML. Comme dans [KBJV06], nous appelons le DDMM, *Domain Definition MetaModel*, cette partie de la syntaxe abstraite. Il représente une conceptualisation du domaine adressé par le DSML. Notons que cette partie de la syntaxe abstraite est indispensable et représente le point de départ pour la définition d'un DSML (et n'est donc pas liée à la propriété d'exécutabilité du DSML).

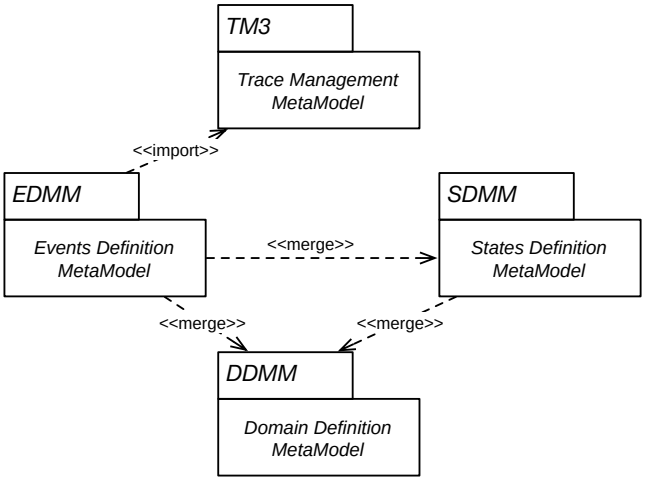


FIGURE 5.5: Architecture générique d'une syntaxe abstraite pour l'exécution des modèles

Afin de pouvoir exécuter un modèle, par exemple un modèle de procédé XSPEM, il est nécessaire de compléter la syntaxe abstraite par les informations permettant de prendre en compte les états possibles du système. Cette partie de la syntaxe abstraite n'est pas indispensable pour la définition de tous les DSML mais uniquement pour ceux qui ont l'objectif de permettre d'exécuter leurs modèles. Nous appelons cette partie de la syntaxe abstraite le SDMM, *States Definition MetaModel*. Il s'agit d'une extension du DDMM. Pour cela nous utilisons dans notre architecture l'opérateur *merge* de l'OMG [OMG06b, §7] (cf. figure 5.5).

La troisième partie de la syntaxe abstraite correspond à la caractérisation des événements qui déclencheront les changements d'états sur le modèle. Nous appelons EDMM, *Events Definition MetaModel*, cette partie de la syntaxe abstraite. Elle s'appuie sur le DDMM et le SDMM (cf. figure 5.5). Notons que la notion d'événement est fortement liée à la notion de transition dans le système. On parlera d'un système *déterministe* lorsque chaque événement a au plus une transition depuis un état donné, et on parlera de système *indéterministe* lorsqu'un événement peut déclencher plusieurs transitions depuis un même état.

Enfin, nous complétons cette architecture par une quatrième partie permettant de capturer une exécution sous la forme d'une séquence d'événements. Nous appelons TM3 (*Trace Management MetaModel*) cette partie de la syntaxe abstraite, générique et donc réutilisable pour chaque DSML.

Les trois premières parties de la syntaxe abstraite sont spécifiques au domaine considéré par le DSML. Ils sont donc à définir chaque nouveau langage. Pour cela, nous proposons dans ce chapitre une démarche décrivant les étapes nécessaires à leurs constructions (cf. figure 5.6) et l'illustrons par la définition de XSPEM. Cette démarche se compose de trois étapes :

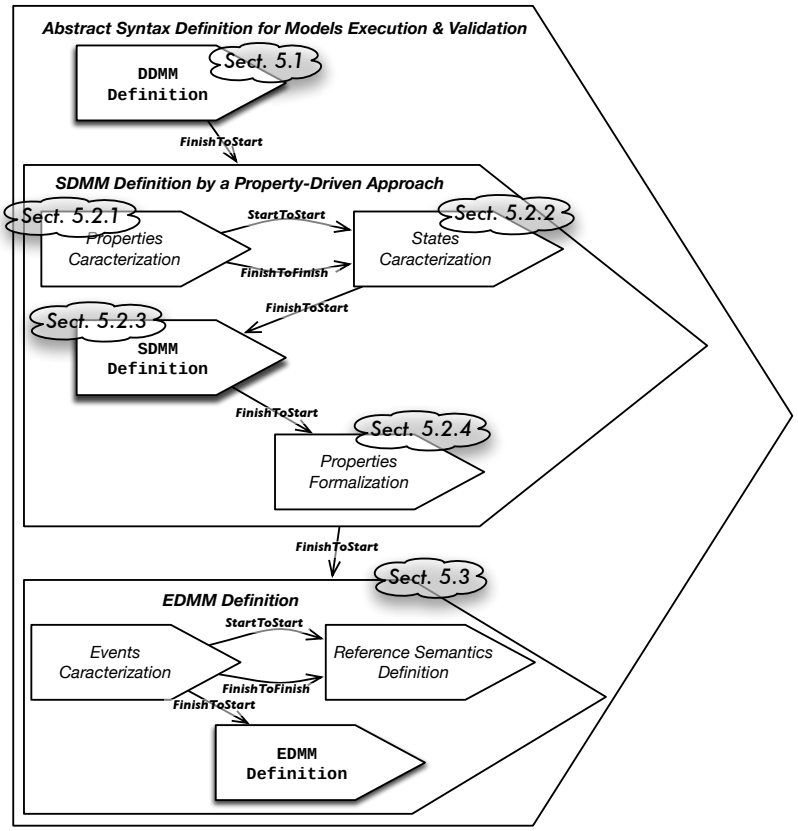


FIGURE 5.6: Démarche pour définir de la syntaxe abstraite d'un DSML exécutable

1. La première consiste à définir le DDMM. Pour XSPEM, le DDMM est défini dans la section 5.1 à partir des concepts du domaine des procédés issus de SPEM et par une première extension permettant de caractériser un procédé pour un projet particulier.

2. La deuxième consiste à définir les informations dynamiques (c.-à-d. le SDMM). Pour cela, nous proposons dans la section 5.2 une approche dirigée par les propriétés grâce à laquelle le niveau d'abstraction des informations dynamiques est défini en partant des questions que l'on se pose au cours de l'exécution du système (ici, le procédé).

3. La troisième consiste à expliciter le comportement connu des experts au moyen d'une sémantique de référence exprimée sous la forme d'un système de transition. Elle permet alors de caractériser les événements du système et ainsi de les expliciter dans le EDMM de la syntaxe abstraite (section 5.3).

5.5 Discussion et synthèse

L'approche que nous décrivons dans ce chapitre permet d'aborder rigoureusement et de manière outillée la définition d'une syntaxe abstraite pour un DSML exécutable. Elle décrit pour cela les différentes préoccupations à prendre en compte et les organise selon une architecture générique capitalisant les parties qui ne sont pas spécifiques au domaine considéré par le DSML.

Notre approche repose sur trois idées centrales. Tout d'abord, la métamodélisation est une activité proche de la modélisation pour laquelle le système considéré correspond au langage que l'on souhaite décrire. A ce titre, il semble pertinent d'intégrer à la métamodélisation la séparation des préoccupations pour en réduire la complexité et en augmenter la maîtrise. Ce point est déjà admis dans la modélisation avec, par exemple, les différents diagrammes UML utilisés pour la définition d'un même système. Le diagramme de classe est généralement au centre de la projection de différentes vues, chacune mettant en avant une préoccupation particulière. De manière similaire, l'approche que nous proposons pour la métamodélisation place le DDMM au centre de la syntaxe abstraite d'un DSML. Il décrit les concepts du domaine et leurs relations. Le SDMM capture les états observables du système et le EDMM décrit les événements observables permettant de faire évoluer le système. Ce sont des préoccupations supplémentaires nécessaires pour la validation dynamique des modèles construits.

D'autre part, la pertinence des informations définies dans la syntaxe abstraite relève de l'évaluation du principe de substituabilité. Pour être pertinentes, il faut qu'elles soient nécessaires et suffisantes afin de répondre aux questions que se pose l'expert du domaine considéré. Nous offrons pour cela un environnement d'expression des propriétés temporelles du système permettant à l'utilisateur d'en spécifier le comportement et d'en déduire les états du système observables sur les modèles.

Enfin, il nous semble important de continuer à placer au centre de la définition d'un DSML sa syntaxe abstraite. Ce principe a déjà permis d'établir des outils génériques ou génératifs pour faciliter la définition de syntaxes concrètes. Ainsi, la suite des travaux que nous présentons dans cette thèse repose sur l'architecture de la syntaxe abstraite décrite dans ce chapitre. Nous étudions à partir de celle-ci la possibilité d'en dériver des approches outillées permettant de simuler et vérifier les modèles construits avec le DSML. Il s'agit alors de décrire la sémantique d'exécution et de définir ou d'intégrer les outils supports à la simulation et la vérification.

A l'image des travaux sur les syntaxes concrètes, nous pensons qu'il est possible d'établir une ingénierie de la sémantique permettant au métamodeleur de n'exprimer que les informations spécifiques à son domaine. Celles-ci doivent ensuite être prises en compte par un ensemble d'outils génériques offrant aux concepteurs les moyens de vérifier et de valider leurs modèles.

Chapitre 6

Approche par traduction pour la vérification de modèle

Dans le cadre du projet TOPCASED, une fonctionalité importante est de pouvoir vérifier formellement les modèles construits à partir des différents DSML de l'atelier vis-à-vis d'un ensemble donné de propriétés. Ces propriétés peuvent être soit générales au langage et définies alors sur le DSML, soit spécifiques au système considéré et exprimées alors sur le modèle.

Pour cela, nous avons dans un premier temps proposé dans le chapitre 5 une démarche pour étendre la syntaxe abstraite et ainsi prendre en compte les informations nécessaires pour l'opérationalisation des modèles ; principalement en vu d'être validés dynamiquement. Cette démarche nous a amené à définir le modèle TOCL des propriétés temporelles du système qui devront être vérifiées, soit au cours de toutes les exécutions possibles (propriétés universelles), soit au cours d'au moins une exécution (propriétés existentielles). Elle a également permis de définir le système de transition, étiquetée par les événements, qui spécifie la sémantique comportementale du DSML, du point de vue de l'expert du domaine.

Par ailleurs, les expérimentations menées dans le chapitre 4 ont permis de mettre en évidence que l'expression d'une sémantique d'exécution par traduction vers un autre espace technique sémantiquement défini permettait de réutiliser l'ensemble des outils disponibles dans le domaine cible. Ici, notre objectif est de réutiliser les outils de vérification. Toutefois, la définition d'une telle sémantique reste délicate. Elle nécessite de bien connaître les domaines source et cible, de manière à trouver une équivalence sémantique entre les éléments sources et cibles.

C'est pourquoi nous étudions dans ce chapitre les moyens de valider une sémantique par traduction en fonction de la sémantique exprimée par l'expert du domaine au moyen du système de transition établi au début de l'approche. Pour cela, nous utilisons la preuve de bisimulation [Mil95] pour montrer l'équivalence de comportement entre le modèle initial et le modèle obtenu par traduction. Nous présentons également les moyens de traduire les propriétés exprimées sur le modèle initial en propriétés exprimées sur le modèle obtenu après traduction.

Notre objectif est de définir une sémantique par traduction pour XSPEM et permettre la vérification des propriétés TOCL exprimées dans le chapitre précédent.

> L'approche présentée dans ce chapitre pour vérifier les modèles issus d'un DSML a fait l'objet de deux publications. La première, lors de la conférence francophone IDM 2007, présente nos travaux sur la vérification de modèle XSPEM [CCBV07]. La deuxième, lors de la conférence européenne ECMDA 2008, présente l'application de cette même méthode pour la validation de programme Ladder [BCC+08].

6.1 Présentation générale de l'approche

L'approche que nous avons suivie se compose de deux étapes que nous illustrons par la figure 6.1. La première consiste à définir la traduction sémantique de manière à pouvoir exploiter le modèle avec les outils du langage cible. La deuxième étape consiste à traduire les propriétés TOCL dans le formalisme attendu par les outils de vérification afin de pouvoir les vérifier sur la traduction du modèle.

La définition d'une sémantique par traduction nécessite de trouver une équivalence sémantique pour les concepts du DSML de départ en fonction des concepts offerts par le langage ciblé. D'autre part, afin de pouvoir manipuler le modèle obtenu par traduction à partir d'outils de vérification, il est nécessaire de le convertir en fonction de la syntaxe concrète attendue.

Afin d'avoir une définition modulaire de la traduction, nous proposons de définir un métamodèle du langage cible et d'établir ainsi la traduction en deux temps. Tout d'abord, nous préconisons de définir la transformation (modèle à modèle) qui traduit les concepts du DSML initial en fonction des concepts métamodélisés et sémantiquement définis du langage visé. C'est cette transformation qui capture la sémantique d'exécution du DSML de départ. Il faut ensuite définir les projecteurs (et éventuellement les injecteurs) permettant de transformer le modèle obtenu par traduction vers (et depuis) la syntaxe concrète attendue par les outils visés pour la vérification. On peut pour cela définir une transformation de type modèle vers texte à l'aide d'un langage général comme ATL [ATL07a, JK05] ou dédié à la génération de code (également appelé *langage de template*) comme JET [JET07], OaW [oAW07] ou Acceleo [Acc07]. On peut également utiliser des langages comme TCS [TCS07, JBK06] ou Sintaks [Sin07, MFF+08] dont l'objectif est de définir une syntaxe concrète textuelle pour un DSML et de générer ensuite les injecteurs et projecteurs permettant de passer du modèle (au format XMI) au texte et inversement. Ces derniers permettent ainsi de rendre transparent le pont entre l'espace technique de l'IDM et celui sur lequel reposent les outils.

Cette approche en deux temps permet ainsi de ne pas mélanger la définition de la sémantique d'exécution du DSML avec le changement purement syntaxique nécessaire pour pouvoir exploiter les outils de l'espace technique cible. Par ailleurs, cette approche permet de s'abstraire des outils utilisés pour le langage cible. En effet, plusieurs outils sont généralement disponibles pour un même langage. Cette

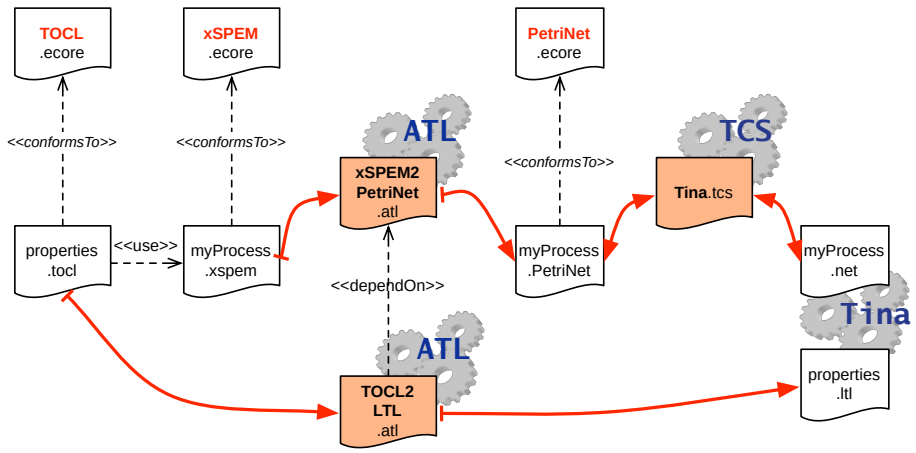


FIGURE 6.1: Approche pour la définition d'une sémantique par traduction

approche permet pour une même expression de la sémantique (c.-à-d. traduction) d'utiliser différents projecteurs, offrant ainsi la possibilité d'utiliser différents outils. Par ailleurs, une modification de la syntaxe concrète d'un outil (par exemple, lors d'un changement de version) n'entraîne qu'une modification du projecteur associé, sans devoir modifier la traduction sémantique de notre DSML de départ. De manière analogue, un changement de la sémantique d'exécution ne nécessite pas de revoir les projecteurs vers des outils spécifiques et permet de partager ces projecteurs entre plusieurs DSML, s'appuyant sur le même langage formel cible.

Par ailleurs, nous proposons de réutiliser les propriétés exprimées en TOCL pour définir les informations dynamiques. En fonction des conventions de nommage définies dans la traduction sémantique du modèle sur lequel s'applique les propriétés, elles peuvent être traduites dans le langage attendu par les outils de vérification visés, par exemple une logique temporelle. De la même manière que pour le langage cible, on peut faire un métamodèle du langage d'expression de propriété afin de s'abstraire des différentes syntaxes concrètes supportées par les outils. Malgré tout, certaines logiques font l'usage d'un formalisme communément admis sur lequel on peut s'appuyer pour traduire directement les propriétés TOCL.

Notons qu'afin de permettre de s'abstraire d'un domaine formel particulier, des travaux en cours au sein du projet TOPCASED visent à définir le langage pivot FIACRE [BBF+08]. Les concepts de ce langage sont d'une abstraction intermédiaire entre ceux des différents DSML de l'atelier et ceux des domaines formels visés par les traductions. Il permettra ainsi de réduire la difficulté pour exprimer la sémantique par traduction des différents DSML et permettra de s'abstraire des différents domaines formels (comme les réseaux de Petri temporels ou les automates temporisés), pouvant ainsi être utilisés indifféremment en fonction des outils qu'ils proposent.

Malgré tout, l'expression d'une sémantique par traduction reste délicate et il est

difficile d'assurer sa cohérence vis-à-vis du comportement attendu par les experts du domaine. Pour cela, nous proposons dans la suite ce chapitre d'explorer la possibilité de valider la traduction exprimée vers un domaine formel par rapport aux règles de transition exprimées par l'expert. Il s'agit de faire une preuve de bisimulation entre le modèle initial et le modèle obtenu par traduction afin d'assurer leur équivalence de comportement. Cette preuve doit être construite par induction sur la structure des modèles de manière à assurer l'équivalence de tous les modèles obtenus par traduction et donc la validité de la transformation (c.-à-d. la sémantique par traduction).

6.2 Application à la vérification de modèle xSPEM

Nous avons utilisé l'approche présentée ci-dessus pour définir et valider une sémantique d'exécution de xSPEM par traduction vers les réseaux de Petri et plus particulièrement les réseaux de Petri temporels à priorités (*Prioritized Time Petri Nets*, PrTPN) [BPV07]. Nous avons également défini une traduction des contraintes TOCL en logique temporelle linéaire (LTL) afin de pouvoir vérifier les modèles traduits en réseaux de Petri vis-à-vis des propriétés établies en amont par le concepteur du langage ou du modèle. La vérification est réalisée par *model-checking* à l'aide de la boîte à outils TINA [TIN07], développée au sein du laboratoire LAAS CNRS. Cette approche est illustrée sur la figure 6.1.

Dans cette partie, pour des raisons de simplification et de clarté des explications, nous présentons ce travail à travers un sous-ensemble d'xSPEM (cf. figure 6.2) dans lequel nous avons unifié les notions de *RoleUse* et *WorkProductUse*, que nous appelons *Resource*. Nous ne traiterons pas non plus les notions d'instances multiples, et de répétition sur les activités ni les états *completed* et *suspended*. La répétition, les activités *ongoing* et les états *completed* et *suspended* sont toutefois disponibles dans l'atelier TOPCASED et à l'adresse $\mathrm{http://combemale.perso.enseeiht.fr/xSPEM/}$.

Notons que la sémantique d'exécution observationnelle et l'état initial d'un modèle xSPEM sont donnés dans la partie 5.2.3. L'état d'un modèle xSPEM reste identique à celui défini dans le chapitre précédent et peut être formalisé de la manière suivante :

Définition (*État d'un modèle* xSPEM) Nous définissons l'état d'un modèle xSPEM comme un ensemble de triplets $(state, inTime, clock) \in \{notStarted, started, finished\} \times \{tooEarly, ok, tooLate\} \times \mathbb{R}^+$. Chaque triplet est associé à une activité du modèle concerné.

La section 6.2.1 introduit les PrTPN, la LTL et la boite à outils TINA utilisés dans nos travaux. Nous détaillons ensuite dans la section 6.2.2 la traduction d'un modèle xSPEM en PrTPN et dans la section 6.2.3 la traduction des propriétés TOCL en LTL.

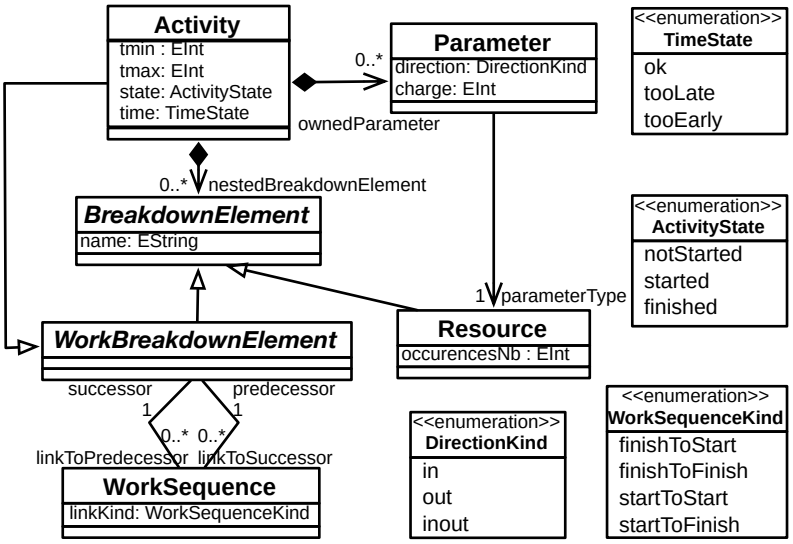


FIGURE 6.2: Syntaxe abstraite de XSPEM (simplifié)

6.2.1 Réseaux de Petri, logique temporelle et boîte à outils TINA

Cette section présente les différents éléments utilisés pour la vérification de modèle XSPEM. Parmi ceux-ci, sont utilisés les PrTPN pour donner une sémantique par traduction au langage XSPEM, la logique temporelle SE-LTL (une variante de la LTL) pour exprimer les propriétés à vérifier sur les réseaux de Petri obtenus par traduction des modèles XSPEM, et l'environnement de description et de vérification TINA qui supporte ces différents éléments. Nous ne décrirons pas dans le détail chacune de ces parties et nous renvoyons le lecteur intéressé à [CCO+04] pour une présentation de SE-LTL, [BV06] pour une présentation des réseaux temporels et [BRV04] pour une description de TINA.

Réseaux de Petri temporels à priorités

Les réseaux de Petri temporels à priorité (*Prioritized Time Petri Nets*, PrTPN) [BPV07] étendent les réseaux temporels (*Time Petri Nets*, TPN) [Mer74] par une relation de priorité $\succ$ sur les transitions. Les TPN sont obtenus depuis les réseaux de Petri [Rei85] en associant deux dates min et max à chaque transition. Supposons que la transition t soit devenue sensibilisée pour la dernière fois à la date θ, alors t ne peut être tirée avant la date $\theta + min$ et doit l'être au plus tard à la date $\theta + max$, sauf si le tir d'une autre transition a désensibilisé t avant que celle-ci ne soit tirée. Le tir des transitions est de durée nulle. Les TPN expriment nativement des spécifications « en délais ». En explicitant débuts et fins d'actions, ils peuvent aussi exprimer des spécifications « en durées ». Leur domaine d'application est

donc large.

D'un point de vue plus formel, les TPN sont des réseaux de Petri dans lesquels un intervalle réel non négatif $I_s(t)$ est associé à chaque transition t d'un réseau. La fonction I_s est appelée *fonction intervalle de tir statique*.

$\mathbf{R}^+$ et $\mathbf{Q}^+$ sont, respectivement, les ensembles des nombres réels et des rationnels non négatifs. Soit $\mathbf{I}^+$ l'ensemble non vide des intervalles réels. Pour $i \in \mathbf{I}^+$, $\downarrow i$ exprime l'extrémité de gauche, et $\uparrow i$ l'extrémité de droite (si i est fini) ou ∞. Pour tout $\theta \in \mathbf{R}^+$, $i \dot{-} \theta$ définit l'intervalle de temps $\{x - \theta | x \in i \wedge x \geq \theta\}$.

Définition (*Réseaux de Petri temporels à priorités (PrTPN)*) Un PrTPN est un tuple $\langle P, T, \mathbf{Pre}, \mathbf{Post}, \succ, m_0, I_s \rangle$, dans lequel $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0 \rangle$ est un réseau de Petri, $I_s : T \rightarrow \mathbf{I}^+$ est une fonction appelée *intervalle de tir statique* et $\succ$ un pré-ordre sur les transitions.

P est un ensemble de *places*, T est un ensemble de *transitions*, $\mathbf{Pre}$, $\mathbf{Post}$: $T \rightarrow P \rightarrow \mathbf{N}^+$ sont les fonctions de *precondition* (arcs entrants) et *postcondition* (arcs sortants), $m^0 : P \rightarrow \mathbf{N}^+$ est le *marquage initial*. Les TPN ajoutent aux réseaux de Petri la fonction *intervalle statique* I_s, qui associe un intervalle temporel $I_s(t) \in \mathbf{I}^+$ à chaque transition du réseau. $Eft_s(t) = \downarrow I_s(t)$ et $Lft_s(t) = \uparrow I_s(t)$ sont respectivement appelés le *temps de tir statique au plus tôt* et le *temps de tir statique au plus tard* de t. Les PrTPN étendent les TPN à l'aide de la relation de priorité $\succ$ applicable aux transitions. Les priorités sont représentées par un arc dirigé entre les transitions, la transition source possédant la plus forte priorité.

Un exemple de PrTPN est donné sur la figure 6.3. Dans cet exemple, la place $p0$ a un marquage équivalent aux poids des arcs vers les transitions t et t'. Ces deux transitions sont donc sensibilisées et peuvent être franchies dans l'intervalle de temps qui leur est associé. Malgré tout, lorsque t et t' peuvent être toutes les deux franchies, t' est prioritaire par rapport à t.

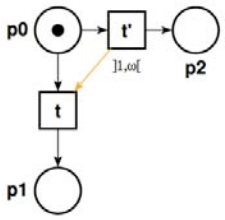


FIGURE 6.3: Exemple de PrTPN

L'état d'un PrTPN et sa relation de transition d'état temporelle $\xrightarrow{t@\theta}$, sont définis de la manière suivante :

Définition (*État et sémantique d'un PrTPN*) L'état d'un PrTPN est une paire $s = (m, I)$ dans laquelle m est le marquage et I est la fonction à intervalle statique. La

fonction $I : T \rightarrow \mathbf{I}^+$ associe un intervalle temporel à chaque transition activée de m.
Nous écrivons $(m, I) \xrightarrow{t@\theta} (m', I')$ ssi $\theta \in \mathbf{R}^+$ et :

1. $m \geq \mathbf{Pre}(t) \ \wedge \ \theta \geq \downarrow I(t)$
   $\wedge \ (\forall k \in T)(m \geq \mathbf{Pre}(k) \Rightarrow \theta \leq \uparrow I(k))$
   $\wedge \ (\forall k \in T)(m \geq \mathbf{Pre}(k) \wedge \theta \geq \downarrow I(k) \Rightarrow \neg k \succ t)$
2. $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
3. $(\forall k \in T)(m' \geq \mathbf{Pre}(k) \Rightarrow$
   $I'(k) = \textbf{ if } \ k \neq t \ \wedge \ m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k)$
   then $\ I(k) \dot{-} \theta$
   else $\ I_s(k))$

Les transitions peuvent être « tirées » à tout moment dans l'intervalle de temps. Un état admet alors généralement un nombre infini d'états successeurs. Comme pour beaucoup de domaines formels pour les systèmes temps réels, l'espace d'état d'un PrTPN est généralement infini. La vérification de réseaux temporels nécessite d'obtenir une abstraction finie de l'espace d'états temporels associé qui est en général infini. Les méthodes permettant d'obtenir ces abstractions finies sont basées sur la technique des classes d'états, et ont été initiées par [BM83, BD91]. Une présentation des résultats les plus récents peut-être trouvée dans [BV06].

Métamodèle des PrTPN Nous avons défini un métamodèle des réseaux temporels à priorités présenté dans la figure 6.4. Un réseau de Petri (*PetriNet*) est composé de noeuds (*Node*) pouvant être des places (*Place*) ou des transitions (*Transition*). Les noeuds sont reliés par des arcs (*Arc*) pouvant être des arcs normaux ou des *read-arcs* (*ArcKind*). Le nombre de jetons d'un arc, appelé également le poids (*weight*), indique le nombre de jetons consommés dans la place source ou ajoutés à la place destination (sauf dans le cas d'un *read-arc* où il s'agit simplement de tester si la place source contient le nombre spécifié de jetons). Le marquage du réseau de Petri est capturé par l'attribut *marking* de chaque place. Les attributs *tmin* et *tmax* permettent d'exprimer un intervalle de temps sur les transitions et la relation réflexive *prioritize* d'exprimer les priorités entres-elles.

Ce métamodèle permet de construire des modèles incorrects. Par exemple on peut mettre un arc entre deux places ou deux transitions. Il a donc été complété par des contraintes OCL pour restreindre les instances valides. Une approche identique a été appliquée pour restreindre le métamodèle SPEM [CCCC06].

La logique temporelle SE-LTL

La logique LTL étend le calcul propositionnel en permettant l'expression de propriétés spécifiques sur les séquences d'exécution d'un système. $SE{-}LTL$ (State/Event LTL) est une variante de LTL récemment introduite [CCO$^+$04], qui permet de traiter de façon homogène des propositions d'états et des propositions de

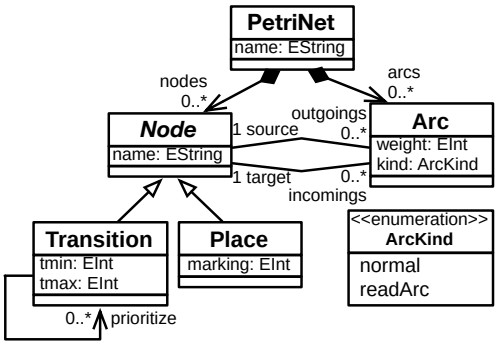


FIGURE 6.4: Métamodèle des réseaux de Petri temporels à priorités

transitions. Les modèles pour la logique $SE-LTL$ sont des structures de Kripke étiquetées (ou SKE), aussi appelées systèmes de transitions de Kripke (ou KTS).

En voici quelques formules :

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

Boite à outils Tina pour les réseaux de Petri

TINA (*TIme Petri Net Analyzer*) est un environnement logiciel permettant l'édition et l'analyse de réseaux de Petri, de TPN et, plus récemment, de PrTPN. Les différents outils constituant l'environnement peuvent être utilisés de façon combinée ou indépendante. Les outils utilisés dans le cadre de cette étude sont :

- nd (*NetDraw*) : nd est un outil d'édition de réseaux temporels et d'automates, sous forme graphique ou textuelle. Il intègre aussi un simulateur « pas à pas » (graphique ou textuel) pour les réseaux temporels et permet d'invoquer les outils ci-dessous sans sortir de l'éditeur.
- tina : cet outil construit des représentations de l'espace d'états d'un réseau de Petri, temporel ou non, avec ou sans priorité. Aux constructions classiques (graphe de marquages, arbre de couverture), tina ajoute la construction d'espaces d'états abstraits, basés sur les techniques d'ordre partiel. Pour les réseaux temporels, tina propose toutes les constructions de graphes de classes discutées dans [BV06]. Tous ces graphes peuvent être produits dans divers formats : « en clair » (à but pédagogique), dans un format d'échange compact à destination des autres outils de l'environnement, ou bien dans les formats acceptés par certains vérificateurs de modèle externes, comme MEC [GV04] pour la vérification de formules du μ-calcul, ou les outils

$CADP$ [BDJM05] pour notamment la vérification de pré-ordres ou d'équivalences de comportements.

– selt : en plus des propriétés générales d'accessibilité vérifiées à la volée par tina (caractère borné, présence de blocage, pseudo-vivacité et vivacité), il est le plus souvent indispensable de pouvoir garantir des propriétés spécifiques relatives au système modélisé. L'outil selt est un vérificateur de modèle (*model-checker*) pour les formules d'une extension de la logique temporelle $SE-LTL$ de [CCO+04]. En cas de non-satisfaction d'une formule, selt peut fournir une séquence contre-exemple en clair ou sous un format exploitable par le simulateur de TINA.

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

Quelques formules de selt :

$\Box\,(p2 + p4 + p5 = 2)$	invariant de marquage linéaire sur les places,
$\Box\,(p2 * p4 * p5 = 0)$	invariant de marquage non linéaire,
infix $q\ R\ p = \Box\,(p \Rightarrow \Diamond\ q)$	déclare l'opérateur « répond à », noté R.

6.2.2 Traduction xSPEM2PrTPN

Nous décrivons maintenant la sémantique par traduction que nous avons implantée pour xSPEM. Nous avons pour cela défini une transformation vers les réseaux de Petri que nous illustrons sur la figure 6.5.

Chaque activité (*Activity*) est traduite par quatre places caractérisant son état (*NotStarted*, *Started*, *InProgress* ou *Finished*). L'état *Started* mémorise qu'une activité a démarré, aussi bien pendant qu'elle est en cours qu'une fois terminée. Nous ajoutons également trois places qui définissent l'horloge locale. L'horloge sera dans l'état *TooEarly* si l'activité finit avant *tmin* et dans l'état *TooLate* si l'activité se termine après *tmax*. Quatre transitions entre ces sept places définissent le comportement de l'activité modélisée. Nous nous appuyons sur les priorités pour séquencer les transitions de manière à exprimer le temps d'exécution de chaque activité. Ainsi, la transition *a_deadline* est définie comme prioritaire à la transition *a_finish* (en gris sur la figure 6.5).

Chaque ressource (*Resource*) est représentée par une place où le marquage initial correspond au nombre d'occurrences (*occurrencesNb*). Chaque paramètre d'une activité (*Parameter*) est traduit par un arc dont le poids (*weight*) correspond

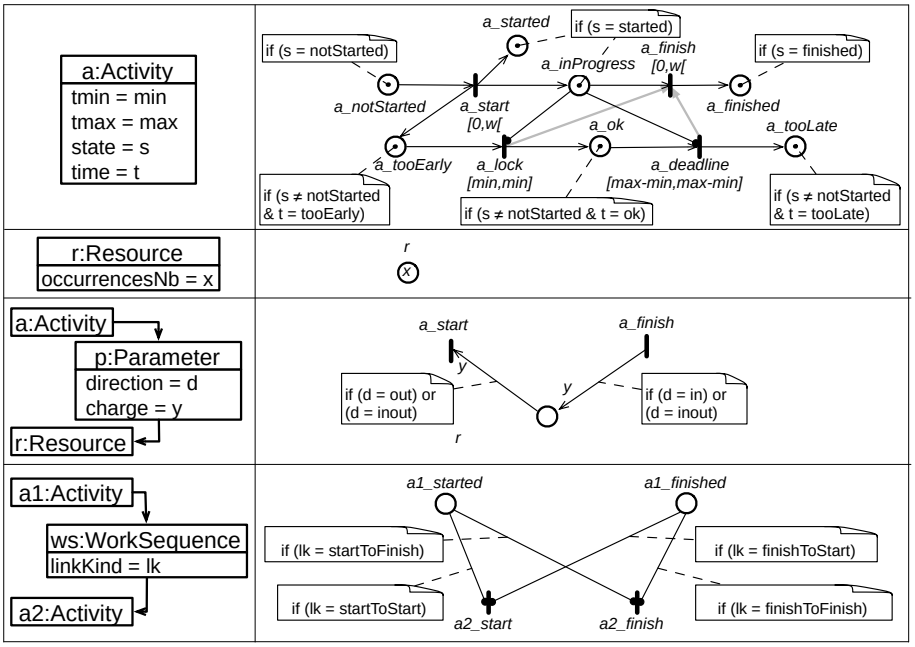


FIGURE 6.5: Schéma de traduction de XSPEM vers PETRINET

à la charge (*charge*). Cet arc est lié à une des transitions d'une activité en fonction de l'attribut *direction*.

Un lien d'ordonnancement (*WorkSequence*) devient un *read-arc* d'une place de l'activité source vers une transition de l'activité cible, en fonction de l'attribut *linkKind*.

La décomposition hiérarchique des activités est représentée sous la forme de contraintes d'ordonnancement. Les activités filles peuvent démarrer uniquement quand l'activité englobante a démarré. Celle-ci ne pourra se terminer qu'une fois les activités filles terminées. Ainsi :

$$(A1 \blacklozenge\!\!- A2) = ((A1 \overset{startToStart}{\longrightarrow} A2) \,\&\, (A2 \overset{finishToFinish}{\longrightarrow} A1))$$

Enfin, l'état du processus est caractérisé par le marquage des places correspondant aux états des activités et de leur horloge locale. Les différents marquages possibles pour une activité sont exprimés sur la figure 6.5 au moyen d'annotations.

La traduction a été implantée par une transformation de modèle en utilisant ATL. Elle a été décrite de manière déclarative, chaque règle exprimant la traduction d'un *pattern* particulier du modèle d'entrée (c.-à-d. d'une construction du DSML initial). Nous profitons de l'héritage de règles offert par ATL pour distinguer certains *patterns* que l'on ne retrouve pas au niveau de la structure de la syntaxe abstraite, mais au niveau de la valeur de certains attributs. Ainsi, nous avons distingué les notions d'activité principale (c.-à-d. le processus) et de sous-activités. Ces dernières sont également distinguées selon qu'elles sont répétables ou *ongoing*. A

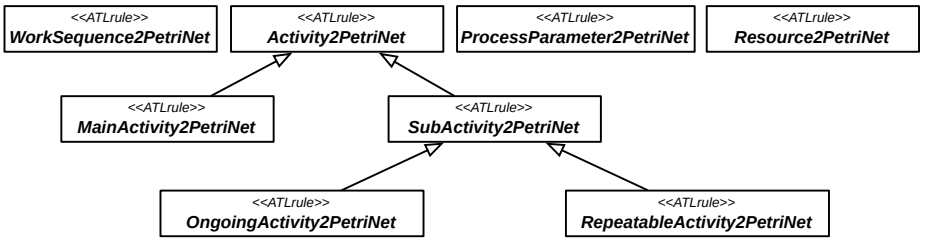


FIGURE 6.6: Structuration de la transformation XSPEM2PETRINET en ATL

chaque type d'activité correspond une règle ATL. Ces règles sont organisées grâce à la relation d'héritage comme le montre la figure 6.6.

Définition (*Traduction* Π) Nous appelons Π la transformation ATL ci-dessus. Elle est définie pour tout modèle conforme à XSPEM.

Le modèle de réseau de Petri obtenu par l'application de la traduction Π sur un modèle XSPEM est ensuite transformé dans la syntaxe concrète attendue par TINA. Nous avons expérimenté pour cela à la fois la définition d'une requête ATL (PETRINET2TINA) et la définition d'un modèle TCS.

L'ensemble des transformations ATL et des modèles TCS est accessible à l'adresse http://combemale.perso.enseeiht.fr/xSPEM.

6.2.3 Expression des propriétés sur les réseaux de Petri

Après la transformation du modèle de processus en réseaux de Petri, nous souhaitons vérifier les propriétés associées à XSPEM en utilisant TINA. Le principe est d'engendrer les propriétés LTL au format texte en utilisant des requêtes ATL. Il faut donc connaître le schéma de traduction de XSPEM vers TINA pour assurer la cohérence. Travailler directement au niveau de PETRINET aurait signifié s'appuyer également sur le métamodèle de LTL.

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

Traduction TOCL2LTL

Les propriétés exprimées au sein du modèle TOCL s'appuient sur les concepts définis dans XSPEM. Elles s'appliquent ainsi sur l'ensemble des modèles de processus conformes à XSPEM. Ces propriétés TOCL peuvent être traduites automatiquement en propriétés LTL portant sur le réseaux de Petri résultant de la traduction. Néanmoins, il faut pour cela connaître le modèle XSPEM à l'origine de la traduction et les conventions de nommage utilisées pour le traduire en PrTPN.

Nous avons dans un premier temps défini une transformation *ad-hoc* spécifique à XSPEM et aux conventions de nommage utilisé dans la transformation

Listing 6.1: Génération des propriétés pour l'étude de la terminaison

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

XSPEM2PETRINET. Nous travaillons actuellement à l'établissement d'une transformation d'ordre supérieur générique qui prendrait en entrée le modèle TOCL des propriétés et une transformation ATL (considérée ici comme un modèle) décrivant une sémantique par traduction (p. ex., dans notre cas, XSPEM2PETRINET).

Consistance des modèles xSPEM

En plus du modèle TOCL défini directement sur XSPEM, l'utilisateur décrit au moment de l'élaboration de son modèle XSPEM les contraintes qui vont régir son procédé. Celles-ci peuvent être de nature causale (c.à-d. stipuler qu'une activité ne peut débuter qu'après la fin d'une autre), liées à la possession de ressources nécessaires pour la réalisation d'une activité, ou encore temporelles. Une fois l'ensemble des contraintes associé au procédé décrit, l'utilisateur doit être capable de savoir si le procédé qu'il vient de décrire est satisfaisable (c.-à-d. réalisable). En d'autres termes, existe-t-il une exécution permettant de réaliser le procédé en respectant les contraintes fixées (temporelles, causales, etc.) ?

Les techniques de *model-checking* vont lui permettre de résoudre ce problème. La question de savoir si un procédé est réalisable peut se ramener à une étude de la terminaison de l'exécution. Dans le contexte de cette étude, nous considérerons que le procédé est terminé si chacune des activités prévues a effectivement été exécutée [1]. A cet effet, la macro-proposition « $\mathrm{finished}$ » est générée automatiquement lors de la traduction du modèle XSPEM (cf. listing 6.1).

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

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

On peut donc s'intéresser à une notion de « consistance faible » permettant de s'assurer de l'existence d'au moins une exécution du procédé. On évalue la

1. D'autres alternatives pourraient bien sûr être considérées, la définition de cette notion de « terminaison » devant se faire au niveau de XSPEM.

propriété « $-\ \Diamond$ finished » qui, littéralement, exprime qu'aucune exécution ne se termine. Si le procédé est faiblement consistant, cette propriété est évaluée à *False* et le contre-exemple produit par selt donne une exécution correcte du procédé. Si la propriété est évaluée à *True* alors le procédé est inconsistant et n'admet aucune solution.

Il est à noter que l'étude de la consistance est générique. Le seul paramètre à fournir est la donnée de la proposition « $\mathrm{finished}$ ». Dans le cas de descriptions temporisées, la démarche est du même type.

Correction de la transformation

Dans le contexte de cette étude, la transformation vers les PrTPN revient à proposer pour XSPEM une sémantique par traduction. Si la transformation exprimée en ATL est à un niveau d'abstraction suffisant, il est toutefois important de pouvoir vérifier que la transformation est consistante et produit une traduction en réseau de Petri conforme à l'« esprit » de la spécification XSPEM. La complexité du réseau de Petri obtenu à partir de la spécification XSPEM rend difficile cette analyse au moment de la mise au point de la traduction.

Pour faciliter la mise au point de la transformation ATL nous engendrons automatiquement un ensemble de formules LTL qui doivent être satisfaites par le réseau de Petri correspondant au processus XSPEM. Ces formules permettent par exemple de s'assurer que les différents états d'une activité sont mutuellement exclusifs, ou de s'assurer des contraintes de précédence entre les différentes activités du procédé, exprimées en XSPEM par l'attribut *linkKind* de *WorkSequence*. Ainsi pour tout couple d'activités $(A1, A2)$ et tout couple de statuts $(status1, status2) \in \{finish, start\}^2$, la contrainte $A1\ status1ToStatus2\ A2$ exprimée en XSPEM donnera lieu à la propriété LTL suivante : $A1\_status1ed\ Precede\ A2\_status2ed$.

L'ensemble de ces propriétés correspond à des « obligations de preuve » (c.-à-d. dans notre cas des invariants) qui doivent être vérifiées. Ces propriétés sont engendrées lors de la phase de transformation du modèle XSPEM en réseaux de Petri et s'appuient sur les conventions de nommage utilisées pour les places et les transitions dans le mapping XSPEM2PETRINET. Dans le cadre de cette étude, nous avons uniquement mis en oeuvre une vérification de ces propriétés par *model-checking* (pour chaque instance transformée), il serait aussi possible et intéressant d'établir (une fois pour toutes) une preuve structurelle de leur validité.

6.3 Validation de la sémantique par traduction

Les obligations de preuve présentées dans la section précédente et engendrées au moment de la mise au point de la transformation ne sont qu'un moyen de détecter des erreurs dans le modèle transformé. Ainsi, nous proposons maintenant de valider formellement la traduction sémantique que nous avons définie entre XSPEM et PrTPN. Dans cette partie, nous établissons pour cela une relation de bisimulation

entre les modèles XSPEM et les modèles de réseaux de Petri obtenus par traduction en fonction de leurs sémantiques respectives (la sémantique de référence de XSPEM et celle des réseaux de Petri). Cette relation assure que les conclusions obtenues sur les réseaux de Petri sont maintenues sur le modèle XSPEM initial. Ainsi, une propriété vérifiée par le *model-checker* selt est une propriété valide sur le modèle XSPEM.

Nous comparons d'abord le nombre possible de transition dans le modèle XSPEM et dans le PrTPN associé. Dans le premier, nous considérons deux transitions applicables sur chaque activité (*StartActivity* et *FinishActivity*), tandis que nous avons quatre transitions pour chaque activité encodée en PrTPN (*a_start*, *a_finish*, *a_lock* et *a_deadline*). Nous avons donc besoin de prouver une bisimulation faible entre ces deux modèles.

Nous décrivons dans la suite les lemmes principaux de la preuve qui est fournie intégralement dans l'annexe A.

Lemme 1 *Soit* MS *l'ensemble des états possibles d'un modèle* XSPEM *et* PNS *l'ensemble des états possibles d'un PrTPN. Pour tout état du modèle* $S \in$ MS *avec une et une seule activité et pour toute séquence* $u \in T^*$ *tel que* $S_0 \rightarrow^{u*} S$ *:*

1. $\forall \lambda \in T, S' \in \text{MS}, S \rightarrow^{\lambda} S' \implies \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} \Pi(S')$
2. $\forall \lambda \in T, P \in \text{PNS}, \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} P \implies \exists S' \in \text{MS}\, t.\, q. \begin{cases} S \rightarrow^{\lambda} S' \\ \Pi(S') \equiv P \end{cases}$

   où $\rightarrow^{\epsilon}$ *décrit une transition non observable.*

Preuve 1 *Soit* S *l'état d'un modèle et* $u \in T^*$ *tel que* $S_0 \rightarrow^{u*} S$*,*

1. *Soit* S' *l'état d'un modèle contenant une seule activité telle que* $S \rightarrow^{\lambda} S'$*. Donc* $\Pi(S)$ *est le réseau de Petri décrit dans la figure 6.5 décrivant seulement une activité.*

$$
\begin{array}{rcl}
I_s & = & \{s \mapsto (0,0), f \mapsto (0,w), l \mapsto (tmin, tmin), \\
 & & d \mapsto (tmax - tmin, tmax - tmin)\} \\
P & = & \{a\_notStarted, a\_started, a\_inProgress, a\_finished\} \\
 & & \cup \{a\_tooEarly, a\_ok, a\_tooLate\} \\
T & = & \{a\_start, a\_finish\} \cup \{a\_lock, a\_deadline\}
\end{array}
$$

   Nous associons respectivement aux transitions StartActivity *et* FinishActivity *de la sémantique de* XSPEM *les transitions* a_start *et* a_finish*. Les deux transitions restantes dans le réseaux de Petri,* a_lock *et* $a\_deadline$*, sont nos transitions epsilon, c.-à-d. les transitions qui ne sont pas observables.*

   Nous montrons que pour toute transition applicable dans MS*, si la transition* λ *peut arriver dans* S *alors la même transition peut arriver dans le réseaux de Petri obtenu par* $\Pi(S)$*.*

2. *Soit P' l'état d'un réseau de Petri tel que $\Pi(S) \rightarrow^{\lambda} P'$.*
   Nous montrons que pour toutes les transitions possibles λ sur $\Pi(S)$, la même transition peut arriver dans S.
3. *Cas initial. De manière triviale $(m, I) = \Pi(S_0)$ est défini et satisfait la propriété.*

Lemme 2 *Soit l'état d'un modèle de procédé $S \in$ MS avec un nombre fini n d'activités, avec des règles de dépendance entre elles, tel que S et $\Pi(S)$ sont faiblement bisimilaires. Soit l'état d'un modèle de procédé $S' \supseteq S \in$ MS défini comme l'état S avec une activité A de plus sans dépendance. Donc S' et $\Pi(S')$ sont faiblement bisimilaires.*

Lemme 3 *Soit l'état d'un modèle de procédé $S \in$ MS avec un nombre fini n d'activités, avec des règles de dépendance entre elles, tel que S et $\Pi(S)$ sont faiblement bisimilaires. Soit l'état d'un modèle de procédé $S' \supseteq S \in MS$ défini comme l'état S avec en plus un lien de dépendance entre les deux activités A_1 et $A_2 \in S$. Donc S' et $\Pi(S')$ sont faiblement bisimilaires.*

Théorème 4 (Bisimulation faible) *Pour tout état de modèle $S \in$ MS et pour toute séquence $u \in T*$ tel que $S_0 \rightarrow^{u*} S$:*

1. $\forall \lambda \in T, S' \in \text{MS}, S \rightarrow^{\lambda} S' \implies \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} \Pi(S')$
2. $\forall \lambda \in T, P \in \text{PNS}, \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} P \implies \exists S' \in \text{MS}\, t.\, q. \begin{cases} S \rightarrow^{\lambda} S' \\ \Pi(S') \equiv P \end{cases}$

Preuve 2 *Par induction sur la structure du modèle de procédé :*

– *L'état initial est prouvé grâce au lemme 1 ;*
– *L'ajout d'une activité préserve la propriété (lemme 2) ;*
– *L'ajout d'un lien de dépendence préserve la propriété (lemme 3) ;*

La décomposition hiérarchique des activités est traduite par des liens de dépendance et préserve donc la propriété de bisimulation.

Dans le processus de validation d'une transformation, la notion de bisimulation est une préoccupation centrale. Actuellement, plusieurs définitions pour prouver la bisimulation existent, choisies en fonction de la granularité entre les événements observables des deux systèmes que l'on souhaite prouver comme étant bisimilaires.

Ces bisimulations ont été sémantiquement définies, comme relations entre des systèmes de transition. Pour la majorité des travaux, elles ont été appliquées sur des calculs de processus, et en particulier sur le π-calcul [San96]. L'accent est principalement mis sur la définition de nouvelles variantes et les propriétés auxquelles elles répondent, sans regarder la possibilité d'automatiser les preuves de bisimulation.

Les travaux récents sur l'automatisation portent encore sur des termes π et définis dans le cadre de l'assistant de preuve COQ [COQ07]. Sous leur forme actuelle, les résultats présentés dans [Hir01, Pou07] sont inapplicables à notre problématique. Pour autant que nous sachions, le nombre de travaux relatifs semble assez

faible. Par ailleurs, le cas très particulier d'un système obtenu grâce à la traduction d'un autre est, de loin, une voie inexplorée, même pour des traductions structurelles et modulaires comme la nôtre.

Malgré tout, les résultats autant théoriques que pratiques sur l'automatisation de la preuve de bisimulation portent sur des systèmes d'état fini, une classe à laquelle les modèles XSPEM se ramènent (dans le cas où il y a pas de création dynamique au cours de l'exécution du processus). Dans ce contexte, de nombreuses questions sur la bisimulation sont décidables, mais nécessitent beaucoup de ressources, comme on peut le voir dans les outils [Bou98, BDJM05].

6.4 Discussion et synthèse

L'approche que nous avons suivie pour valider par *model-checking* les modèles issus de DSML tels que XSPEM a consisté à définir une sémantique par traduction vers les réseaux de Petri et plus particulièrement vers la boite à outils TINA.

La sémantique de référence issue de l'approche décrite dans le chapitre 5 a permis de donner un cadre à la définition de la traduction et d'en valider ensuite formellement l'équivalence par bisimulation. Cette approche permet ainsi de valider l'expression de la traduction à partir d'une sémantique exprimée par l'expert du domaine directement sur les concepts qu'il connaît. Elle permet également de garantir que les résultats obtenus sur le réseau de Petri sont pertinents sur le modèle XSPEM initial.

Pour implanter la sémantique par traduction de XSPEM, nous avons utilisé le langage de transformation de modèle ATL. Un tel langage offre des constructions déclaratives permettant de définir une transformation sous la forme de règles qui seront appliquées sur chacun des *patterns* trouvés dans le modèle d'entrée. De cette manière, le métamodeleur peut décrire sa transformation en fonction des traductions sémantiques de chaque *pattern*, sans se soucier de l'application des règles sur le modèle d'entrée et le séquencement à suivre pour cela. D'autre part, ATL offre la possibilité de définir un héritage entre les règles. Cet aspect du langage permet de faire ainsi de l'héritage de *pattern*, venant compléter l'héritage de concept déjà présent dans la syntaxe abstraite. Enfin, notons que les techniques de *model-checking* sont très coûteuses en ressources. Il est donc généralement indispensable d'optimiser la traduction établie entre le DSML initial et le langage ciblé. Il est alors d'autant plus difficile de s'assurer que la traduction est bien cohérente par rapport au comportement attendu par l'expert et il semble donc indispensable d'en valider l'équivalence avec le système de transitions initial par bisimulation.

Par ailleurs, les contraintes exprimées en TOCL pour la caractérisation des états sont traduites en LTL afin d'être vérifiées automatiquement sur le réseau de Petri issu de la traduction d'un modèle. Un ensemble d'obligations de preuve est également engendré pour aider à la définition de la transformation par la détection d'erreurs.

L'approche suivie dans ce chapitre nécessite de maîtriser le DSML source, le

domaine cible, le langage d'expression des propriétés et le langage de transformation. Malheureusement, des solutions automatiques semblent difficiles à envisager. Des travaux sont actuellement en cours dans le projet TOPCASED pour définir le langage pivot FIACRE [BBF+08]. Il permettra d'abstraire différents domaines formels comme les réseaux de Petri et les automates. Ce langage offre des concepts plus abstraits réduisant l'écart avec les DSML de l'atelier et simplifiant donc la définition d'une sémantique par traduction.

Un autre problème actuellement beaucoup étudié par la communauté est celui du retour des informations d'analyse, du domaine formel vers le DSML initial (p. ex. traduire le contre-exemple fourni par selt sur le PrTPN en une trace pouvant être simulée sur le modèle XSPEM). Différentes approches sont actuellement étudiées et reposent toutes sur la traçabilité des transformations de modèle. La solution qui nous semble être la plus pertinente repose sur l'utilisation d'un modèle de liens [FBJ+05] définissant les liens entre le DSML initial et le domaine formel cible. Ce modèle de lien pourrait ensuite être utilisé pour engendrer automatiquement les transformations dans les deux sens, chacune s'appuyant sur le modèle de trace fourni par la transformation inverse. Certaines expérimentations ont déjà été menées dans [ATL07b]. Notons qu'un retour *ad-hoc* peut toutefois être défini en s'appuyant sur les conventions de nommage utilisées dans la sémantique par traduction.

L'approche que nous avons présentée dans ce chapitre est intéressante car elle permet de se concentrer uniquement sur les concepts du domaine considéré. Ainsi, cette approche semble pertinente aussi bien pour des langages de modélisation que pour d'autres types de langage. Par exemple, nous avons mené une étude similaire afin de pouvoir vérifier des programmes Ladder [BCC+08]. Nous avons pour cela défini le métamodèle du langage Ladder et avons implanté sa sémantique sous la forme d'une traduction vers les PrTPN.

Chapitre 7

Cadre générique pour la définition de simulateurs de modèle

Dans le chapitre précédent, nous avons vu que l'effort pour définir une chaîne de vérification formelle des modèles était important et nécessitait d'être répété pour chaque DSML. De plus, la vérification par *model-checking* fait face à la difficulté du « passage à l'échelle ». En effet, il est encore très coûteux en terme de ressources (de calcul et de mémoire) de vérifier exhaustivement des systèmes comptant de nombreux états atteignables.

Pour valider les modèles, un autre moyen considéré dans le projet TOPCASED consiste à simuler le comportement dynamique du système et obtenir ainsi une trace d'exécution. Cette simulation peut être réalisée en *batch* à partir d'un scénario pré-défini ou de manière interactive. Cette dernière solution permet alors à l'utilisateur de construire progressivement la trace d'exécution à partir des événements qu'il injecte. L'utilisateur peut également voir évoluer son modèle tout au long de l'exécution et ainsi contrôler visuellement le comportement du système pour une exécution donnée. On parle alors d'animation de modèle.

Dans ce chapitre, nous décrivons dans un premier temps les objectifs de la simulation, les besoins des utilisateurs de TOPCASED et les approches existantes (section 7.1). Nous présentons ensuite l'architecture générique que nous avons définie de manière à pouvoir partager certains composants d'un simulateur et pouvoir engendrer les autres (section 7.2). Un prototype de simulateur de processus XSPEM illustre cette approche (section 7.3). Nous concluons enfin par une discussion et une synthèse des travaux (section 7.4).

> Les travaux présentés dans ce chapitre ont fait l'objet de deux rapports livrés dans le lot n°2 du projet TOPCASED, lot en charge des aspects métamodélisation et des travaux sur la simulation [1]. Le pre-

1. TOPCASED *Model Simulation*, cf. https://gforge.enseeiht.fr/projects/topcased-ms

mier rapport présente l'état de l'art sur les besoins et les outils pour la simulation de modèle [Lab06b]. Le deuxième décrit l'approche retenue dans TOPCASED pour faciliter la définition de simulateurs pour les différents DSML considérés dans l'atelier [Lab06a].

7.1 Description des besoins et des approches pour la simulation

7.1.1 Objectifs de la simulation de modèle

La simulation permet d'améliorer la compréhension d'un système sans devoir le manipuler réellement, soit parce qu'il n'est pas encore défini ou disponible, soit parce qu'il ne peut pas être manipulé directement en raison des coûts, du temps, des ressources ou du risque. La simulation est donc réalisée sur un modèle du système.

La simulation est généralement définie selon trois étapes (cf. figure 7.1). La première consiste à générer une représentation de la charge de travail, c.-à-d. l'ensemble des entrées à appliquer sur le système étudié. Cette représentation peut être une trace [2] (ou un scénario [2]) décrivant une charge de travail réelle, ou plus synthétique et générée par une heuristique ou une fonction stochastique. La deuxième étape consiste à simuler le modèle à partir de la charge de travail définie en entrée pour produire les résultats. Enfin, la troisième étape consiste à analyser les résultats de la simulation pour acquérir une meilleure compréhension du système considéré.

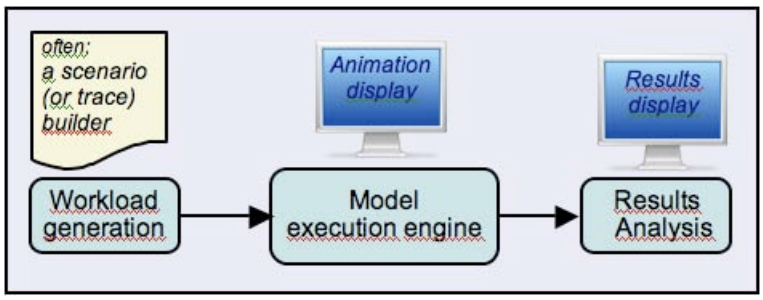


FIGURE 7.1: Les trois étapes de la simulation

Ces trois étapes peuvent être réalisées par un ou plusieurs outils, comme un constructeur de scénario (*scenario builder*) pour engendrer la charge de travail, un moteur de simulation (*simulation engine*) et un outil d'analyse des résultats. Il est également possible de combiner dans un même outil deux ou trois de ces étapes. Par exemple, il est possible de créer interactivement une trace au cours de l'exécution d'un modèle. Il est aussi possible de coupler le moteur d'exécution

2. tel que défini dans la partie 5.4

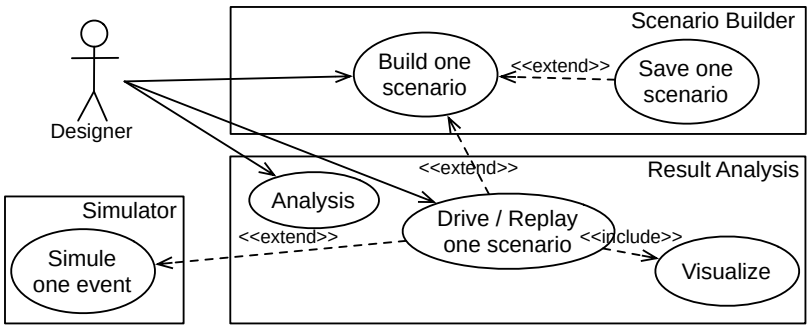


FIGURE 7.2: Exigences pour la simulation de modèle dans TOPCASED

avec l'outil d'analyse des résultats afin d'offrir une représentation « à la volée » (c.-à-d. au cours d'une simulation) de l'évolution du modèle.

7.1.2 Exigences pour la simulation dans l'atelier TOPCASED

Les principales fonctionnalités exprimées dans le cadre de TOPCASED sont synthétisées sur la figure 7.2. La simulation vise à valider les modèles construits à partir des DSML de haut niveau d'abstraction définis dans l'atelier. Il s'agit principalement de fournir les moyens au concepteur d'animer son modèle afin d'en avoir une meilleure compréhension et de le valider vis-à-vis des exigences de l'utilisateur. Il souhaite pour cela pouvoir visualiser l'évolution du modèle dans le même formalisme que celui utilisé pour le construire (*Result Analysis*). Cette animation peut être guidée par un scénario pré-défini ou réalisée de manière interactive à l'aide d'un constructeur de scénario (*Scenario Builder*) permettant d'injecter des événements, dit *exogènes*. Le concepteur doit pouvoir contrôler l'exécution, à l'aide d'un panneau de contrôle permettant de la lancer, la stopper, revenir en arrière ou avancer.

Actuellement, les DSML envisagés dans TOPCASED se concentrent principalement sur des modèles asynchrones ou synchrones à événements discrets. Un modèle de calcul à événements discrets peut donc être utilisé. Nous le présentons dans la partie suivante.

7.1.3 Simulation à événements discrets

Dans la simulation de systèmes discrets, le temps simulé est un temps virtuel discret. Deux mécanismes peuvent être considérés pour l'évolution du temps discret :

– une approche basée sur une horloge périodique (évolution du temps par incrément fixe) : le temps avance dans le modèle selon des intervalles égaux de temps réel,

– une approche basée sur les événements : le modèle est pris en compte et mis à jour uniquement à l'arrivée d'un nouvel événement (ou quand le système change d'état) ; le temps avance d'un événement à l'autre.

Les formalismes de modélisation traditionnels s'appuient sur des bases mathématiques et ont largement précédé l'arrivée des ordinateurs. Deux types de démarche ont été proposées pour modéliser les systèmes discrets [ZKP00] :

– La spécification de système à temps discret (en anglais, *Discrete Time System Specification* – DTSS) a été proposée pour modéliser les systèmes dont l'évolution du temps est basée sur une horloge périodique, comme les automates.
– La spécification de systèmes à événements discrets (en anglais, *Discrete Event System Specification* – DEVS ou DE) est apparue plus récemment et a été définie pour la simulation sur ordinateur. Il s'agit d'une approche basée sur les événements, plus facile à gérer par un programme exécutable qu'un modèle mathématique "pur".

Ainsi, nous avons deux principaux paradigmes pour modéliser la dynamique des systèmes discrets : les formalismes discrets périodiques (DTSS, également appelé "synchrone"), et discrets basés sur les événements (ou DEVS DE, aussi nommé "asynchrone"). Ces formalismes permettent de concevoir des systèmes modulaires et hiérarchiques. Une telle construction consiste à coupler des modèle existants afin de construire une modélisation plus large du système. Ces formalismes permettent de traiter un ensemble de modèles comme un modèle de base dans un ensemble de plus haut niveau d'abstraction.

7.1.4 Ateliers de simulation existants

Plusieurs outils supportent la simulation à événements discrets. Toutefois, il est assez difficile d'interfacer ces outils avec l'environnement TOPCASED, c'est-à-dire utiliser directement l'un de ces outils pour simuler et animer un modèle défini à l'aide d'un éditeur de l'atelier TOPCASED. En effet, il faudrait pour cela définir une traduction bidirectionnelle entre le DSML initial et le langage de l'outil de simulation. Nous avons mentionné la difficulté de définir une telle traduction dans le chapitre précédent. Néanmoins, il est intéressant d'analyser les principales caractéristiques de ces outils et d'en souligner les plus utiles pour les utilisateurs de TOPCASED.

Matlab/Simulink [Mat07a] et Scilab/Scicos [CCN06] sont principalement dédiés à la simulation de systèmes physiques avec des modèles de temps continu. Ils offrent toutefois certaines possibilités pour supporter un modèle à événements discrets, comme le module *Stateflow* dans l'atelier Matlab/ Simulink.

Dans le cadre de TOPCASED, nous avons profité de l'expérience du projet RNTL COTRE [BRV+03, FBB+03] et des études sur l'animation de modèle *états/ transitions* à travers les fonctionnalités de Hyper-formix Workbench [Wor07], Sildex [Sil07], StateMate [Sta07] et Uppaal [BDL04]. Ces outils, basés sur des automates simples ou temporels, offrent une représentation graphique mettant en avant

les états actifs et les transitions franchissables. Ces outils sont également couplés avec des moyens de visualisation et d'enregistrement les traces.

Une autre source d'inspiration a été l'environnement Ptolemy II [Pto07] développé à l'Université de Berkeley. Il supporte la modélisation hétérogène, la simulation et la conception de systèmes concurrents. Les modèles de simulation sont construits selon des modèles de calcul qui dirigent les interactions des composants dans le modèle. Le choix du modèle de calcul dépend du type de modèle à construire [LSV98]. Dans le cas du modèle à événements discrets, la communication entre les composants se fait à l'aide d'une séquence d'événements datés. Un événement est caractérisé par une valeur et une date. Les composants peuvent être des processus qui réagissent aux événements. Le modèle de calcul à événement discret offre une sémantique déterministe par entrelacement pour la gestion des événements simultanés. Après 10 ans d'existence, Ptolemy II est un outil mature et riche, du moins pour les modèles de calcul les plus importants, y compris le modèle à événements discrets.

L'efficacité d'un outil en terme de puissance de calcul pour réaliser la simulation et la capacité de l'intégrer dans un processus industriel sont des critères importants à prendre en compte pour le choix d'un outil. Les outils interprétant graphiquement les modèles au cours de leur simulation manquent d'efficacité. Aussi, ils sont le plus souvent utilisés comme outils de prototypage. Un outil de simulation par animation est donc le plus souvent utilisé pour concevoir efficacement et rapidement un premier modèle dans le but de l'améliorer et le valider par simulation. Une fois validé, ce modèle est traduit dans un langage de programmation (p. ex. Java) et couplé à un moteur de simulation efficace supportant le modèle à événement discret.

7.2 Architecture générique pour la définition d'un simulateur de modèle

Les principales exigences retenues dans le premier périmètre de TOPCASED, celui pris en compte par nos travaux, sont résumées sur le diagramme des cas d'utilisation de la figure 7.2. Bien sûr, ces exigences sont décrites du point de vue de l'utilisateur. Néanmoins, nous avons essayé de séparer les différentes préoccupations et présentons dans la suite de cette partie les différents composants définis pour la simulation de modèle (figure 7.3). Chaque composant est fortement couplé à une ou plusieurs parties de la syntaxe abstraite et est défini soit de manière générique (pour ceux qui s'appuient sur TM3, *Trace Management MetaModel*), soit de manière générative en complétant le modèle de la syntaxe abstraite par les informations de la syntaxe concrète.

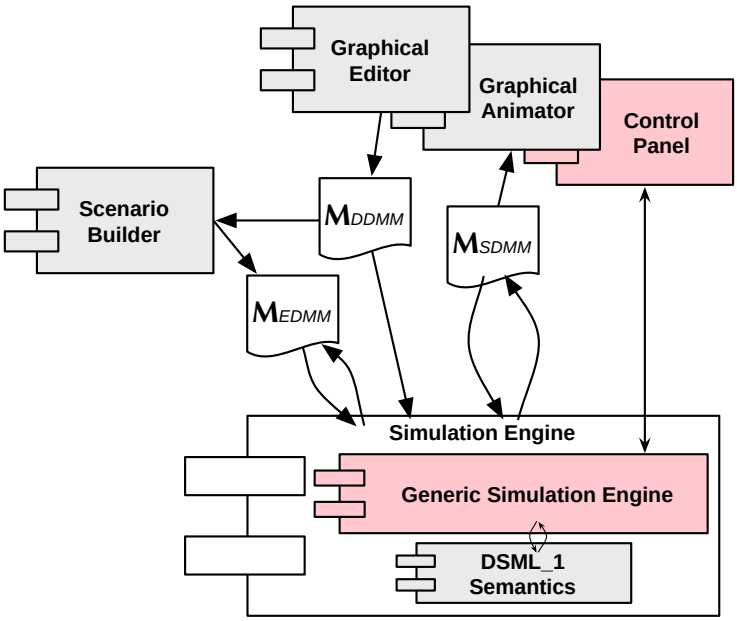


FIGURE 7.3: Composants d'un simulateur dans TOPCASED

7.2.1 Éditeur graphique

Du point de vue d'un utilisateur de TOPCASED, le premier composant utilisé est l'éditeur graphique (*Graphical Editor*). Même s'il ne rentre pas directement dans le processus de simulation, il permet de construire le modèle structurel (M_{DDMM}) en s'appuyant sur des représentations principalement graphiques.

7.2.2 Constructeur de scénario

Un scénario est défini comme une séquence d'occurrences d'événements exogènes. L'occurrence d'un événement est définie par sa date d'occurrence, les éléments du modèle simulé sur lesquels il agit et, parfois, certains paramètres. Par exemple, dans le cas de l'animation d'un processus XSPEM, nous avons défini plusieurs événements. Sur les activités, l'événement *startActivity* démarre une activité. D'autres événements peuvent prendre des paramètres. C'est le cas de l'événement *setProgression* qui permet d'augmenter le taux de réalisation d'une activité.

Un constructeur de scénario (*Scenario Builder*) est un outil qui permet de définir une séquence d'occurrences d'événements, soit en amont de la simulation (à partir des exigences, d'un générateur de scénario, d'un générateur de tests, ou d'un contre-exemple fourni par un *model-checker*), soit de manière interactive au cours de l'exécution du modèle. Cette séquence dirige l'évolution souhaitée du modèle. Le constructeur de scénario permet également de sauvegarder le scénario ou d'en charger un précédemment défini.

La définition des événements est spécifique à chaque DSML et est prise en compte dans le cadre de notre approche au sein du EDMM. Un constructeur de scénario permet donc de construire un modèle M_{EDMM} qui s'appuie sur le modèle structurel (M_{DDMM}) issu de l'éditeur. Au sein de l'atelier TOPCASED, deux approches, potentiellement complémentaires, sont envisageables :

- Préalablement à une simulation, en utilisant un générateur d'éditeur afin de pouvoir définir les scénarios. On peut alors envisager des éditeurs soit graphiques à l'aide d'outils tels que TOPCASED ou GMF, soit textuels à l'aide d'outils tels que TCS ou Sintaks.
- Pendant la simulation, en s'appuyant sur l'éditeur graphique de l'atelier TOPCASED pour animer le modèle. Le constructeur de scénario peut alors prendre la place de la palette et proposer les différents types d'événements que l'on peut introduire dans le scénario pendant la simulation.

Si l'interface du constructeur de scénario est spécifique aux événements de chaque DSML, toute la gestion et la représentation de l'évolution (scénario ou trace) peuvent être capitalisées. D'autre part, nous travaillons actuellement pour étendre le TM3 de manière à s'appuyer sur un standard, tel que le Profil de Test de l'OMG [OMG05b].

7.2.3 Moteur de simulation

Les événements définis dans le scénario sont successivement interprétés par un moteur de simulation (*Simulation Engine*). Pour cela, il charge dans un premier temps le modèle structurel du système et construit un modèle dynamique (c.-à-d. représentant l'état du système en cours d'exécution) qu'il initialise en fonction de l'état initial (M_{SDMM}).

Par la suite, le moteur de simulation a en charge de mettre à jour les informations dynamiques du modèle (c.-à-d. celles définies dans le SDMM) en fonction des occurrences d'événements définies dans le scénario. Dans le cadre de notre architecture, il est composé de deux composants principaux : un moteur générique et une définition de la sémantique pour chaque DSML. Le moteur générique définit le modèle de calcul à événements discrets en fonction des concepts génériques définis dans le TM3. Il est composé des trois éléments principaux suivants :

- *Le pilote* (*Driver*) permet de contrôler la simulation et de faire l'interface avec les composants extérieurs. Il offre une API permettant de réaliser la simulation en mode *batch* ou interactive.
- *L'agenda* conserve les événements de l'exécution en cours, ordonnés en fonction de leur date d'échéance. Il est initialisé au début de la simulation par les événements du scénario et offre l'interface nécessaire au pilote pour obtenir un événement particulier et rajouter de nouveaux événements.
- *L'interpréteur* offre une interface avec la sémantique des différents DSML. Il interprète un événement abstrait (*Event*). Le comportement en réaction à un événement particulier sera défini dans la sémantique du DSML et est chargé en fonction du contexte.

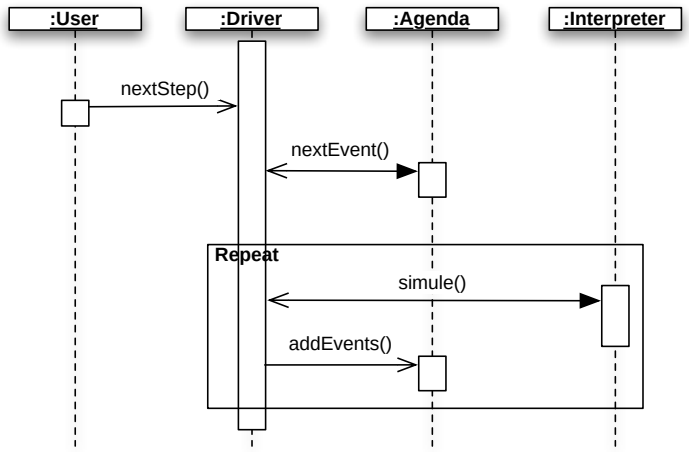


FIGURE 7.4: Interactions (simplifiées) pour la gestion d'un événement

Un exemple d'interactions entre ces éléments est présenté sur la figure 7.4. Il représente (de manière simplifiée) les échanges pour le traitement d'un nouvel événement de l'agenda. La sémantique précise que nous avons implantée pour le modèle de calcul à événement discret est décrite dans [Gro07].

Le composant spécifique à chaque DSML implante la sémantique opérationnelle du langage. Plus précisément, il définit comment les informations dynamiques évoluent en réaction à chacun des événements. Par exemple pour XSPEM, démarrer une activité correspond à modifier son état interne (attribut *state*), si les conditions sont remplies pour qu'elle puisse être démarrée.

L'atelier TOPCASED s'appuie sur l'environnement Eclipse implanté en Java. C'est pourquoi nous avons dans un premier temps utilisé l'API Java offerte par EMF pour définir les composants spécifiques à chaque DSML. Chacun spécialise l'interpréteur du moteur générique et est défini comme un *plugin* Eclipse. Il s'appuie sur un *plugin* générique implantant le modèle de calcul à événement discret.

Notons que cette approche est très facilement adaptable de manière à définir la sémantique opérationnelle avec un autre langage d'action. De la même manière, l'API générique du moteur peut facilement permettre de le réutiliser dans un autre environnement.

7.2.4 Panneau de contrôle

Le panneau de contrôle (*control panel*) permet à l'utilisateur d'interagir avec le moteur d'exécution pour lancer ou stopper la simulation, revenir en arrière ou avancer dans le scénario. Il est défini sous la forme d'un *plugin* Eclipse générique communiquant avec l'interface du moteur de simulation. Il offre à l'utilisateur une vue graphique sous la forme d'un *player* classique, intégré dans l'environnement

d'Eclipse.

7.2.5 Animateur de modèle

L'animateur de modèle (*Graphical Animator*) permet de visualiser l'évolution du système en s'appuyant sur le modèle dynamique (M_{SDMM}) mis à jour au fur et à mesure de l'exécution par le moteur de simulation.

Il est spécifique au DSML considéré. Dans le cadre de TOPCASED nous avons décidé de nous appuyer sur l'éditeur graphique utilisé pour construire le modèle en le complétant par la description graphique des informations dynamiques. Comme pour la description de la syntaxe concrète graphique du DDMM, il est possible de capitaliser des composants graphiques classiques tels que des jauges, des barres de progression, des jeux de couleurs, etc. Ces éléments permettent de compléter la visualisation du modèle pour mettre en évidence l'état de la simulation en cours. Techniquement, un configurateur pourrait permettre d'associer aux éléments du SDMM de la syntaxe abstraite ces composants graphiques. L'animateur pourra être alors engendré sous la forme d'un *plugin* Eclipse qui s'appuie sur la syntaxe concrète graphique existante de l'éditeur.

7.2.6 Outils d'analyse

L'analyse d'une simulation ressemble à la visualisation de l'exécution. La principale différence est que dans le cas d'une animation, c'est l'utilisateur qui doit interpréter la visualisation proposée, alors que dans le cas de l'analyse, c'est un outil qui offre les résultats d'une analyse à l'utilisateur. Par exemple, un outil d'analyse peut vérifier que la trace d'exécution respecte certaines propriétés temporelles. L'outil d'analyse peut être couplé avec le moteur de simulation de manière à vérifier au cours de l'exécution le respect d'une ou plusieurs propriétés et avertir l'utilisateur dès que le modèle n'est plus valide.

Nous n'avons pas abordé cet aspect dans nos premiers travaux. L'approche présentée dans cette thèse permet toutefois de réutiliser facilement les propriétés temporelles exprimées en TOCL de manière à les vérifier sur les traces d'exécution construites par simulation.

7.3 Prototype de simulateur de modèle XSPEM

La principale application de nos travaux sur la simulation de modèle a consisté à définir un simulateur de machines à états UML2.0 au sein de l'atelier TOPCASED. Une étude est actuellement en cours pour prendre également en compte le langage d'automates SAM (*Structured Analysis Model*). Ces travaux sont présentés dans le chapitre 9 consacré aux transferts des travaux dans l'atelier TOPCASED.

Néanmoins, nous avons également profité de notre architecture générique pour offrir la possibilité de simuler les modèles de processus XSPEM. Cela permet entre autre de valider graphiquement la faisabilité du processus vis-à-vis des contraintes

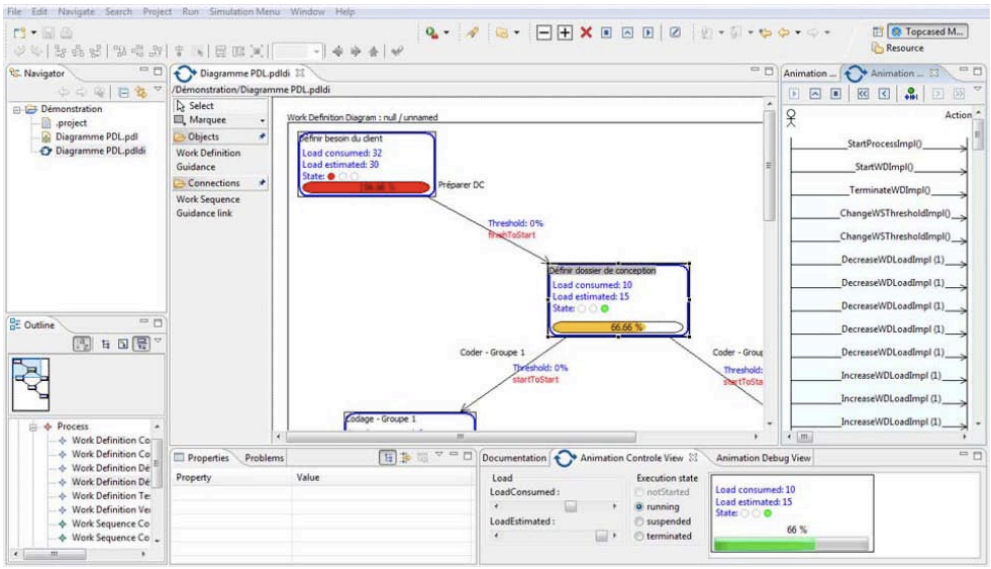


FIGURE 7.5: Capture écran du prototype de simulateur pour les modèles XSPEM

exprimées dans le modèle (ordonnancement, ressources et temps). Une capture écran du prototype est présentée sur la figure 7.5.

Nous avons travaillé sur la définition des composants graphiques pertinents pour l'animation d'un processus [CM06]. Ces différents composants graphiques ont été, dans un premier temps, intégrés directement au sein de l'éditeur graphique préalablement défini (vue graphique du processus sur la figure 7.5). Ils pourraient toutefois être intégrés au générateur d'éditeur défini dans le cadre du projet TOPCASED de manière à les capitaliser pour différents animateurs de modèle.

Pour le constructeur de scénario, différentes vues ont été expérimentées dans Eclipse. La figure 7.5 montre sur la droite le diagramme de séquence correspondant aux interactions entre l'utilisateur et le système. En bas, la vue de contrôle de l'animation permet d'envoyer les événements aux éléments sélectionnés dans la vue graphique. Enfin, des actions similaires ont également été intégrées à la barre d'action d'Eclipse.

Le panneau de contrôle est pour sa part accessible à partir d'un menu dédié à la simulation dès que l'animateur est lancé (*SimulationMenu* sur la figure 7.5).

Enfin, la sémantique d'XSPEM a été implantée sous la forme d'un *plugin* qui spécialise l'interpréteur abstrait du moteur de simulation générique. Ce *plugin* décrit le comportement d'un modèle XSPEM en réaction à chacun des événements décrits dans le EDMM (XSPEM *ProcessObservability*).

7.4 Discussion et synthèse

Dans le cadre du projet TOPCASED, on souhaite définir un simulateur de modèle pour différents langages de l'atelier. Afin de réduire l'effort nécessaire pour définir ces simulateurs, nous utilisons dans ce chapitre l'architecture de la syntaxe abstraite proposée dans le chapitre 5. Celle-ci nous permet :

1. d'offrir un moteur de simulation à événements discrets et un panneau de contrôle génériques reposant sur la partie de la syntaxe abstraite qui est partagée entre les différents DSML : TM3. Le moteur est ensuite paramétré par la sémantique du langage considéré en définissant la réaction à chaque événement décrit dans le EDMM.
2. de réutiliser l'approche suivie dans TOPCASED pour la génération des éditeurs graphiques de manière à pouvoir générer :
   – le constructeur de scénario à partir d'un modèle de la syntaxe concrète souhaitée pour les événements définis dans le EDMM,
   – l'animateur de modèle qui vient compléter la représentation graphique définie pour l'éditeur par les informations dynamiques exprimées dans le SDMM.

Cette approche profite de l'abstraction offerte par l'IDM afin de partager les outils indépendants des concepts d'un domaine particulier. Chaque composant présenté a été implanté sous la forme de *plugin* Eclipse s'appuyant sur la plate-forme elle-même, mais aussi sur la librairie graphique GEF et sur l'environnement de métamodélisation EMF.

Actuellement, l'animation s'appuie sur l'éditeur disponible dans l'atelier TOPCASED pour visualiser les modèles. Ceci suppose que les modèles doivent être de taille raisonnable pour que l'utilisateur puisse suivre leurs évolutions. Dans le cas de modèles de taille importante, il est nécessaire de développer des analyseurs spécifiques qui collecteront et agrégerons les informations pertinentes. Ce module a été identifié dans l'architecture mais n'a pas été implanté pour l'instant.

D'autres travaux ont proposé des approches génériques pour la simulation de modèle dans un contexte de DSML. Les plus proches sont les travaux réalisés dans le cadre du projet Modelplex et présentés dans [KDH07]. Toutefois, la définition du moteur de simulation n'est pas intégrée dans une approche plus globale. Celle-ci nous permet d'avoir un couplage fort avec la structure générique de la syntaxe abstraite et de prendre en compte l'ensemble des composants d'un simulateur.

Chapitre 8

Cadre formel pour la métamodélisation

Les travaux présentés dans cette thèse s'inscrivent dans un champ de recherche encore très exploratoire où aucun consensus n'a été atteint. Néanmoins, l'approche proposée a permis d'explorer certaines pistes et surtout de mettre l'accent sur l'importance d'une ingénierie de la sémantique dans le contexte des langages de modélisation. Les technologies actuelles permettent d'implanter les différentes approches proposées dans cette thèse pour exprimer une sémantique d'exécution et les premières applications de ces travaux sont très prometteuses. Cependant, chacune d'entre elles représente une implantation différente de la sémantique d'un même langage. Il est donc important de pouvoir valider leurs cohérences. Il nous semble pour cela indispensable que chacune soit équivalente, voire le résultat d'une transformation à partir d'une *sémantique de référence*, exprimée de manière plus abstraite et qui traduit directement la connaissance et l'expérience des experts en s'appuyant sur les concepts de leur domaine.

Nous présentons dans ce chapitre les travaux préliminaires que nous avons réalisés dans l'objectif de définir un cadre pour la réalisation d'un environnement formel de métamodélisation. Celui-ci doit permettre d'exprimer aussi bien sa syntaxe abstraite que sa sémantique, statique et dynamique. Le cadre formel doit permettre de traiter aussi bien des modèles que leurs langages de modélisation et permettre de les replacer formellement vis-à-vis d'architectures telles que le MDA.

L'idée centrale de notre approche est la distinction forte entre le rôle (c.-à-d. la fonction [1]) d'un modèle dans une démarche globale (p. ex. le MDA) et sa nature décrivant son implantation formelle adéquate. Après avoir distingué précisément ces deux propriétés et les travaux relatifs qui ont servi de base à nos réflexions (section 8.1), nous détaillons le cadre formel que nous avons défini (section 8.2) et montrons comment il pourrait s'appliquer à une approche telle que celle de l'OMG à travers le MDA pour les aspects syntaxique et statique (section 8.3). Nous expliquons ensuite comment ce cadre peut être utilisé comme représentation abs-

1. Nous emploierons indifféremment les termes de *rôle* et de *fonction*.

traite pour formaliser des modèles construits à partir d'environnements existants tels qu'EMF (section 8.4). Enfin, bien qu'encore préliminaire, nous terminons sur une présentation des possibilités pour réutiliser ce cadre afin de formaliser la sémantique dynamique du langage, et pouvoir ainsi en exécuter les modèles (section 8.5).

> La formalisation du cadre de métamodélisation présenté dans ce chapitre a donné lieu à une publication dans le workshop international Towers 2007 [TCCG07].

8.1 *Fonction* et *Nature* des modèles

Pour formaliser un environnement de *(méta)*modélisation, il faut dans un premier temps étudier précisément la fonction et la nature de chaque élément [2] manipulé. *La fonction* permet de comprendre exactement le rôle que joue un élément par rapport aux autres auxquels il est relié. La compréhension de la fonction des éléments permet de définir précisément les paradigmes de l'espace technique [BK05] dans lequel ils apparaissent. *La nature* permet de décrire précisément les informations que véhicule chaque élément et les services qu'il rend. Elle reste invariable par rapport à l'environnement. Elle permet de comprendre de quelle manière chaque élément peut être formellement représenté et comment il peut être manipulé. Notons que la nature et la fonction sont deux propriétés orthogonales, qu'il nous semble primordial de bien définir pour comprendre et formaliser un espace technique.

Dans un premier temps, de nombreux travaux ont porté sur la définition du concept de modèle et sur l'étude de la relation de représentation, qui permet de le construire à partir du système qu'il modélise [Sei03, SJ07]. Avec l'évolution de l'IDM et l'arrivée de la métamodélisation, d'autres travaux ont étudié les différents niveaux d'abstraction de manière à décrire le rôle d'un modèle à chaque niveau. Les premiers d'entre eux sont certainement ceux qui ont établi les différentes approches de métamodélisation comme MDA [MM03], MIC [SK97] et *Software Factories* [GSCK04]. En effet, les premiers rôles importants définis pour les modèles sont ceux qui qualifient leur niveau d'abstraction, correspondant plus exactement dans le cas du MDA au niveau d'« utilisation » du modèle. On distingue généralement ce niveau par le nombre de « méta » que l'on inscrit devant le mot « modèle ». Le premier niveau (sans le mot « méta », M_1) correspond aux modèles utilisés pour décrire un système réel. Le deuxième niveau (avec une fois le mot « méta », M_2) correspond aux modèles utilisés pour décrire les langages, et ainsi de suite...

Ces travaux ont permis de décrire différentes approches supportant l'IDM. Elles introduisent chacune des fonctions, souvent similaires, mais jamais formalisées et laissant part à certaines ambiguïtés. Par la suite, des travaux ont défini pré-

2. Nous emploierons le terme « *élément* » pour nommer un concept défini dans un espace technique.

cisément les relations de représentation (μ) et de conformité (χ) [Fav04, BB04]. Ils ont également permis de bien différencier la relation d'instanciation, entre un élément d'un modèle et un élément du langage, de la relation de conformité entre un modèle et son langage. Ces travaux sont décrits dans le chapitre 2. Enfin, plus récemment, les travaux de [Küh06b] (complétés dans [Hes06] et [Küh06a]) introduisent les rôles *token* et *type*. Le premier qualifie un modèle permettant de filtrer les éléments d'un système ou d'un autre modèle (projection un à un de certains éléments). Le rôle *type* qualifie un modèle qui classe un ensemble d'éléments d'un système ou d'un autre modèle. Ces fonctions leurs permettent de distinguer la métamodélisation *linguistique* et *ontologique*. Ces fonctions permettent de qualifier un modèle uniquement dans un contexte particulier. Un modèle peut donc jouer différents rôles selon les éléments par rapport auxquels on le voit (c.-à-d. du point de vue).

Grâce à ces travaux, l'espace technique de l'IDM a été délimité et ses principaux concepts définis. Ces travaux sont indispensables pour mieux comprendre les approches de métamodélisation. La compréhension de ces approches permet alors d'en proposer des environnements (EMF, MDR, etc.) dont les implantations peuvent différer.

Lorsque l'on souhaite exprimer formellement la sémantique d'un DSML, il faut dans un premier temps définir formellement la représentation abstraite du DSML (sur laquelle sera exprimée la sémantique) et des modèles (sur lesquels la sémantique s'appliquera). Ces représentations définissent la nature des éléments. Elles peuvent être définies de manière *ad-hoc*, par exemple pour chacun des rôles que peut jouer un modèle, ou être abstraites dans un cadre plus générique. Les premiers travaux (p. ex. [SJ05]) ont porté sur la définition formelle de tous les modèles mais sans prendre en compte leurs langages ni des approches plus générales comme le MDA.

Plus récemment, et comme base de nos travaux, Jouault *et al.* proposent, à travers la formalisation du langage de métamodélisation KM3, une première étude sur la formalisation possible de la nature des modèles en leur donnant un codage formel [JB06]. Cette étude permet également de replacer cette formalisation par rapport aux fonctions que peuvent jouer les modèles dans une architecture plus globale comme le MDA. Ils introduisent pour cela le concept de modèle de référence (*ReferenceModel*) comme un modèle de la syntaxe abstraite du langage permettant de définir des modèles (à gauche sur la figure 8.1). Un modèle doit être conforme à son modèle de référence. Ils ont également formalisé les concepts de *TerminalModel*, *MetaModel* et *MetaMetaModel* (à droite sur la figure 8.1) avec le rôle communément admis dans la communauté de l'IDM (cf. chapitre 2). Métamodèles et métamétamodèles sont des modèles de références dont la fonction est définie par le modèle auquel ils sont conformes. Un modèle est défini comme un triplet (G, ω, μ) où G est un graphe, ω un autre modèle qui est le modèle de référence pour le modèle donné et μ est la fonction qui, à partir d'un élément de G, permet de connaître le nœud du graphe ω dont il est instance. Comme cette définition ne permet pas une représentation finie des modèles, les auteurs proposent de

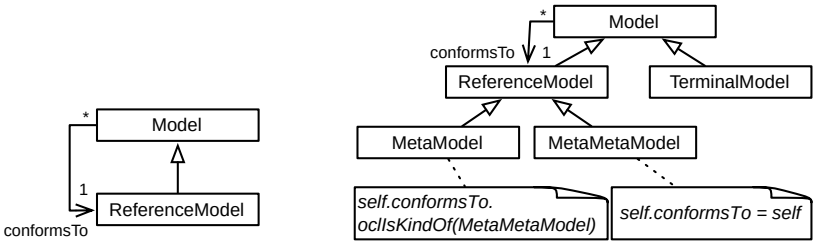


FIGURE 8.1: Formalisation des concepts de l'IDM selon [JB06]

définir un modèle initial de manière réflexive. Ce modèle est le métamétamodèle et permet de définir des métamodèles qui permettront de créer des modèles (cf. figure 8.1). Le cœur de l'article présente la formalisation de KM3. Ceci est fait de manière incrémentale, en commençant par un langage réduit aux seuls concepts de *classe* et *référence*.

L'approche distingue la fonction d'un *ReferenceModel*, qui permet de créer des modèles qui y seront conformes, de celle d'un *TerminalModel* qui représente un système. Néanmoins, ils tentent d'unifier leur nature dans celle définie pour tout modèle. Bien que cette approche soit pratique à bien des points de vue, sa formalisation pose des problèmes et impose des distorsions que l'on peut observer dans l'annexe proposant une formalisation en Prolog. En effet, l'approche consistant à tout voir comme modèle et à en unifier la nature, ne permet pas de définir proprement un modèle réflexif. La formalisation des modèles ne peut pas être bien fondée sans l'existence de modèles initiaux. Introduire ce concept de modèle initial, comme un modèle réflexif, le différencie par sa nature des autres modèles « non initiaux ». Cela casse alors la vue hégémonique du « tout est modèle » dans laquelle tout les modèles sont de nature homogène.

Le code Prolog permettant de valider la réflexivité de KM3 est fourni en annexe mais ne suit pas la formalisation décrite dans l'article. Par exemple, la fonction μ n'est pas explicitée et est dispersée au sein des différentes règles Prolog. Par ailleurs, le choix de Prolog pour l'implantation ne permet pas d'accéder à des mécanismes de typage et de structuration (modularité et hiérarchisation).

Sur la base de ces travaux, nous proposons dans la partie suivante un cadre formel minimal et bien fondé dans lequel nous distinguons clairement la nature d'un modèle par rapport aux différentes fonctions qu'il peut jouer dans un cadre plus général (p. ex. la pyramide de l'OMG). Nous souhaitons également que la formalisation définie soit facilement réutilisable par un langage d'action afin d'exprimer sur un DSML aussi bien la sémantique statique que dynamique. Nous exposons dans la dernière partie du chapitre les approches possibles pour prendre en compte les aspects dynamiques.

8.2 Description du cadre proposé

Nous présentons dans cette section les fondations de notre cadre formel. Nous définissons dans un premier temps les concepts de *Model* et *ModelClass* distinguant les différentes natures puis les relations de conformité et de promotion les liants.

8.2.1 *Model* & *Model Class*

En suivant les paradigmes de l'approche orientée objet (OO), l'IDM s'appuie sur deux niveaux différents :

- le premier permet de définir les concepts et les relations entre eux (appelé *Reference* en IDM). Chaque système à modéliser est alors défini à partir de ces concepts et de ces références. Dans l'approche OO, les concepts sont représentés par des *Classes* et les références par des *(relation d') Associations*.
- le deuxième niveau correspond à la définition des systèmes. Il est composé d'un ensemble d'éléments et de relations entre eux. Dans l'approche OO, ces éléments sont appelés *Objects* et ces relations sont appelées *Links*.

Ces deux niveaux d'abstraction sont liés par une *relation d'instanciation* ou *relation de typage* suivant le sens considéré. Dans l'approche OO, un objet O est une instance d'une classe A. On dit aussi que O est de type A. Chaque lien entre deux objets est tel qu'il existe, dans le langage associé, une association entre les deux types (c.-à-d. les deux classes) des objets considérés.

Notre approche consiste à séparer le niveau des instances du niveau des types, en définissant des structures différentes pour caractériser les deux natures de modèle. Aussi, nous utilisons les deux termes *Model* et *ModelClass*. Un modèle (*Model*, M) correspond au niveau des instances et une classe de modèles (*ModelClass*, MC) au langage de modélisation à partir duquel on peut créer une famille de modèles (cf. figure 8.2). Ces modèles sont alors dit conformes à la classe de modèle (cf. figure 8.2). Une classe de modèles définit aussi bien la structure que la sémantique (statique et dynamique) des modèles qu'elle permet de décrire. Par exemple, en UML, la multiplicité définit le nombre d'objets qui peuvent être liés. De plus, on peut utiliser OCL pour définir des contraintes plus complexes qui n'ont pas de notation graphique. La sémantique dynamique est pour l'essentiel décrite informellement.

Dans notre cadre, le concept de *ModelClass* n'est pas une spécialisation de *Model*. Ce sont deux éléments de nature différente. Un modèle est défini comme un multigraphe dont les nœuds représentent les objets du modèle et les arcs représentent les liens entre ces objets. Chaque arc est caractérisé par une étiquette portant le nom de la référence dont il est instance. A titre d'exemple, un sous-ensemble du modèle XSPEM est donné sur la figure 8.3. Il est créé à partir de la classe de modèles MOF présentée dans le chapitre 2 et sur laquelle nous reviendrons formellement dans la section 8.3.1 (cf. figure 8.5). On constate que chaque élément est bien une instance d'un élément du MOF : les concepts sont des ins-

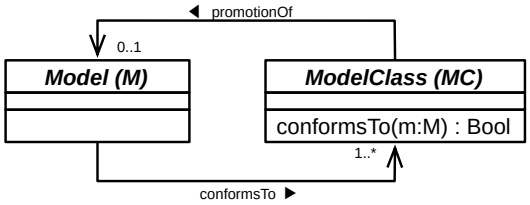


FIGURE 8.2: Diagramme de classe de *Model* & *ModelClass*

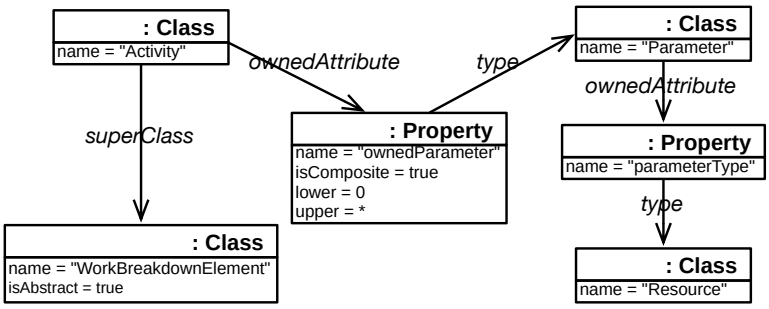


FIGURE 8.3: Modèle (en MOF) d'un sous-ensemble de XSPEM

tances de $Class$, les références sont des instances de $Property$ et l'ensemble des liens sont établis à partir des références du MOF. Par exemple, le lien d'héritage entre le concept $WorkBreakdownElement$ et $Activity$ est étiquetté par le nom de la référence $superClass$.

Une classe de modèles est également définie comme un multigraphe où les nœuds représentent les classes et les arcs représentent les références. Elle est également complétée par les propriétés sémantiques du langage. Sur la figure 8.4 est donnée la classe de modèles XSPEM correspondant au même sous-ensemble que celui pris en compte dans la figure 8.3. Elle ne représente que le multigraphe avec certaines propriétés sémantiques statiques comme, par exemple, les multiplicités. Le paragraphe 8.2.3 expliquera comment obtenir la classe de modèles par *promotion* du modèle.

Définissons formellement les notions de $Model$ et $ModelClass$.

Soit deux ensembles, $\mathrm{Classes}$ représentant l'ensemble de toutes les classes possibles et $\mathrm{References}$ correspondant à l'ensemble des étiquettes de référence (c.-à-d. l'ensemble des noms des références). Nous considérons également l'ensemble $\mathrm{Objects}$ des instances de ces classes.

Définition (*Model*) Soit $\mathscr{C} \subseteq \mathrm{Classes}$ un ensemble de classes. Soit $\mathscr{R} \subseteq \{\langle c_1, r, c_2\rangle \mid c_1, c_2 \in \mathscr{C}, r \in \mathrm{References}\}$ un ensemble de références entre les classes tel que : $\forall c_1 \in \mathscr{C}, \forall r \in \mathrm{References}, card\{c_2 \mid \langle c_1, r, c_2\rangle \in \mathscr{R}\} \leq 1$.

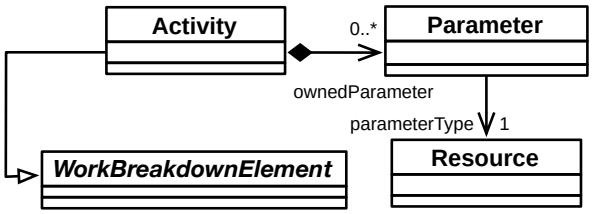


FIGURE 8.4: Classe de modèles d'un sous-ensemble de XSPEM

Un modèle $\langle MV, ME \rangle \in \mathrm{Model}(\mathscr{C}, \mathscr{R})$ est un multigraphe construit à partir de l'ensemble fini MV d'objets typés, les nœuds, et de l'ensemble fini ME d'arcs étiquetés tel que :

$$MV \subseteq \{\langle o, c\rangle \mid o \in \mathrm{Objects}, c \in \mathscr{C}\}$$
$$ME \subseteq \{\ \langle\langle o_1, c_1\rangle, r, \langle o_2, c_2\rangle\rangle \mid \ \langle o_1, c_1\rangle, \langle o_2, c_2\rangle \in MV, \langle c_1, r, c_2\rangle \in \mathscr{R}\ \ \}$$

Notons que dans notre cadre, un même objet peut correspondre à différents nœuds dans le graphe d'un modèle, associé à des classes différentes. La sémantique de cette duplication éventuelle est relative à l'héritage, et traduit le fait qu'un même objet a plusieurs types. Ces deux notions sont usuellement présentes dans les langages orientés objets. Cet aspect est détaillé dans la partie 8.3.1.

Définition (*Model Class*) Une classe de modèles (*ModelClass*) est un multigraphe représentant des classes et des références. Elle définit également les propriétés sémantiques relatives à l'instanciation des classes et des références. Elle est défini comme une paire composée d'un multigraphe (RV, RE) construit à partir d'un ensemble fini RV de classes et d'un ensemble fini RE d'étiquettes de référence, et d'une fonction prédicat appelée *conformsTo* dont le domaine est les modèles.

Une classe de modèles est un tuple $\langle (RV, RE), conformsTo\rangle \in \mathrm{ModelClass}$ tel que :

$$RV \subseteq \mathrm{Classes}$$
$$RE \subseteq \{\langle c_1, r, c_2\rangle \mid c_1, c_2 \in RV, r \in \mathrm{References}\}$$
$$conformsTo : \mathrm{Model}(RV, RE) \rightarrow Bool$$

tel que $\forall c_1 \in RV, \forall r \in \mathrm{References}, card\{c_2 \mid \langle c_1, r, c_2\rangle \in RE\} \leq 1$

8.2.2 Conformité

Étant donné un modèle M et une classe de modèles MC, nous pouvons vérifier la conformité de M par rapport à MC. C'est l'objectif de la fonction $conformsTo$ définie dans MC. Elle identifie l'ensemble des modèles valides par rapport à MC.

Dans notre cadre, la conformité vérifie sur le modèle M que :

– Tout objet o de M est instance d'une classe C de MC. On dit aussi que o a pour type C.

- Tout lien entre deux objets est tel qu'il existe, dans MC, une référence entre les classes typant les deux objets. Dans la suite, nous dirons que ces liens sont instances des références entre les classes de MC.
- Enfin, toutes les propriétés sémantiques définies dans MC sont satisfaites dans M. Par exemple, les multiplicités définies sur les références entre les concepts expriment des contraintes sur le nombre de liens possibles entre les objets de ces classes (c.-à-d. concepts). Nous le détaillons dans la partie 8.3.1.

La notion de conformité est représentée sur la figure 8.2 par la relation entre M et le MC auquel il est conforme. Un modèle peut être conforme à plusieurs classes de modèles différentes, c.-à-d. que le modèle est conforme à chacune des classes de modèles.

8.2.3 Promotion

Une classe de modèles peut être définie *from scratch* ou construite à partir de la promotion d'un modèle. Même si nous l'avons représenté sur la figure 8.2 par un prédicat *promotionOf* par symétrie avec le prédicat *conformsTo*, *promotion* est une opération qui permet de construire une classe de modèles à partir d'un modèle. Celle-ci peut ensuite être utilisée pour définir de nouveaux modèles.

L'opération de promotion doit donc construire le multigraphe de la classe de modèles et définir les propriétés sémantiques encodées dans la fonction *conformsTo*. Les différentes étapes de l'opération de promotion sont les suivantes :

1. identifier dans le graphe du modèle, les éléments qui correspondent à des concepts et seront donc traduits en nœuds dans la classe de modèles.
2. identifier les relations entre les concepts précédents. Le graphe du modèle est encore utilisé pour identifier les chemins entre les éléments promus en concepts qui correspondent à des références dans la classe de modèles. Ces chemins seront traduits en arcs au niveau de la classe de modèles.
3. définir les propriétés sémantiques qui doivent être vérifiées par les modèles se conformant à la classe de modèles. La conjonction de ces propriétés est intégrée dans la fonction $conformsTo$ de la classe de modèles résultant.

8.3 Application : formalisation de MOF et de la pyramide de l'OMG

Nous illustrons dans cette section notre approche à travers la formalisation de l'architecture proposée par l'OMG. Nous nous concentrons sur *EMOF_Core*, un sous-ensemble d'EMOF [OMG06b, §12], comme classe de modèles initiale. Elle est présentée dans la partie 8.3.1 avec la formalisation des propriétés sémantiques intégrées à la fonction *conformsTo*. Dans la partie 8.3.2, nous formalisons la métacircularité de *EMOF_Core* en définissant le modèle conforme à la classe de mo-

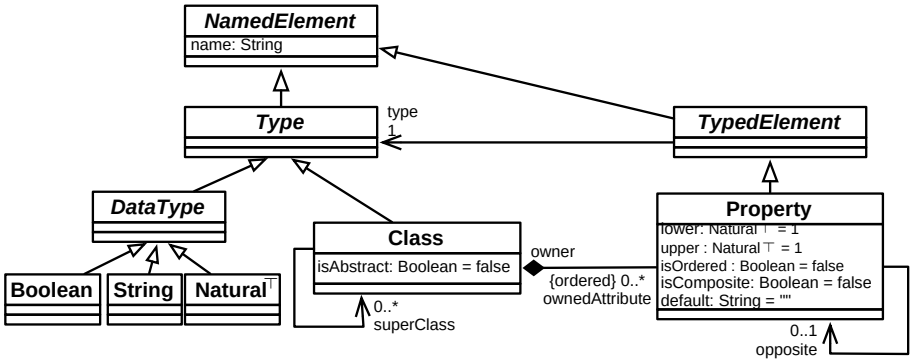


FIGURE 8.5: Classe de modèles de *EMOF_Core* (notation du diagramme de classe UML)

dèles définie dans la partie 8.3.1 et la promotion permettant à partir de celui-ci de redéfinir la classe de modèles initiale.

8.3.1 Définition du *ModelClass EMOF_Core*

Le sous-ensemble d'EMOF est présenté dans la figure 8.5 en utilisant la notation usuelle du diagramme de classe UML. Le *ModelClass* associé peut être défini de la manière suivante :

$$
\begin{aligned}
RV \triangleq\ & \{\ NamedElement, Type, TypedElement, DataType, Boolean, \\
& String, Natural^{\top}, Class, Property\ \} \\
RE \triangleq\ & \{\ \langle Class, \mathrm{ownedAttribute}, Property\rangle, \langle Class, \mathrm{isAbstract}, Boolean\rangle, \\
& \langle Class, \mathrm{superClass}, Class\rangle, \langle Class, \mathrm{inh}, Type\rangle, \ldots\ \}
\end{aligned}
$$

où "inh" représente la relation standard d'héritage.

Nous détaillons dans la suite de cette partie la formalisation de la classe de modèles. La fonction $conformsTo$ est pour sa part présentée dans la partie 8.3.2 comme le résultat de la promotion définie.

Une classe ($Class$ dans la figure 8.5) est une entité conceptuelle qui permet de définir une famille d'objets. Elle peut être instanciée pour construire des *instances*. Une instance est un objet particulier qui a été créé en suivant les contraintes de structure définies par sa classe. Chaque instance est liée exactement à une classe par la relation d'instanciation.

Une classe regroupe un ensemble ordonné de propriétés ($Property$) qui sont instanciées de multiples fois pour construire la structure de chaque objet. Chaque propriété est définie entre une classe et un type ($Type$) qui peut être :

- un type de donnée ($DataType$), associé à une sémantique de valeur. Elle est alors appelée attribut.
- une autre classe. Elle est alors appelée référence.

Description d'une classe

Héritage Il s'agit du moyen de définir une classe (une sous-classe) en réutilisant d'autres classes (les classes parentes) déjà définies. Une sous-classe est liée à ses classes parentes par un lien de type *superClass*. La nouvelle classe hérite des propriétés des classes parentes.

Dans le cadre que nous avons défini, nous avons identifié différentes approches possibles selon que la classe de modèles doit intégrer ou non la notion d'héritage.

Dans le cas sans héritage les objets ne peuvent pas être dupliqués et nous formalisons la propriété ainsi :

$$\begin{aligned} &noInheritance \triangleq \langle MV, ME\rangle \mapsto \\ &\quad \forall \langle o_1, c_1\rangle, \langle o_2, c_2\rangle \in MV, o_1 = o_2 \Rightarrow c_1 = c_2 \end{aligned}$$

Dans le cas d'un héritage classique, les objets peuvent être dupliqués, à condition que les classes de ces doublons est un sous-type commun, la classe de base. La propriété suivante est paramétrée par inh, prenant en compte la relation d'héritage standard. Nous définissons d'abord un prédicat auxiliaire qui exprime qu'un objet o de type c_1 a un dupliqué de type c_2, avec c_2 sous-type de c_1.

$$\begin{aligned} &hasSub(inh \in RE, o \in \text{Objects}, c_1, c_2 \in \text{Classes}, \langle MV, ME\rangle) \triangleq \\ &\quad c_1 = c_2 \vee \exists c_3 \in \text{Classes}, \langle\langle o, c_2\rangle, inh, \langle o, c_3\rangle\rangle \in ME \\ &\quad \wedge hasSub(inh, o, c_1, c_3, \langle MV, ME\rangle) \end{aligned}$$

Nous définissons ensuite la notion d'héritage standard. Le premier prédicat exprime que la relation d'héritage se transmet à travers les objets dupliqués. Le deuxième exprime que chaque ensemble d'objets dupliqués a un objet dupliqué de base, c'est à dire dont la classe est la classe de classe.

$$\begin{aligned} &standardInheritance(inh \in RE) \triangleq \langle MV, ME\rangle \mapsto \\ &\quad \forall \langle\langle o_1, c_1\rangle, inh, \langle o_2, c_2\rangle\rangle \in ME, o_1 = o_2 \\ &\quad \wedge \forall \langle o_1, c_1\rangle, \langle o_2, c_2\rangle \in MV, o_1 = o_2 \Rightarrow \exists c \in \text{Classes}, \\ &\quad\quad hasSub(inh, o_1, c_1, c, \langle MV, ME\rangle) \\ &\quad\quad \wedge hasSub(inh, o_2, c_2, c, \langle MV, ME\rangle) \end{aligned}$$

Enfin, la propriété suivante exprime que c_2 est une sous-classe de c_1.

$$\begin{aligned} &subClass(inh \in RE, c_1, c_2 \in RV) \triangleq \langle MV, ME\rangle \mapsto \\ &\quad \forall \langle o, c\rangle \in MV, c = c_2 \Rightarrow \langle\langle o, c_2\rangle, inh, \langle o, c_1\rangle\rangle \in ME \end{aligned}$$

Les classes abstraites Une classe est abstraite si son attribut *isAbstract* a la valeur vraie. Elle ne peut pas être instanciée. Les classes abstraites sont utilisées pour représenter les entités et concepts abstraits.

Dans un modèle M conforme à MC, chaque objet instance d'une classe abstraite doit avoir un objet dupliqué instance d'une des sous-classes concrètes. Pour

vérifier cette propriété il suffit d'appliquer le prédicat $isAbstract$ suivant sur toutes les classes identifiées comme abstraites.

$$isAsbstract(inh \in RE, c_1 \in RV) \triangleq \langle MV, ME\rangle \mapsto \\ \forall\langle o, c\rangle \in MV, c = c_1 \Rightarrow \exists c_2 \in RV, \langle\langle o, c_2\rangle, inh, \langle o, c_1\rangle\rangle \in ME$$

Description des propriétés

Lower & Upper Pour un attribut ou une référence, le nombre minimum et maximum d'instances du concept cible peut être défini en utilisant les attributs $lower$ et $upper$. Cette paire est usuellement appelée multiplicité. Ces deux attributs sont utilisés afin d'exprimer un intervalle pour le nombre d'instances possibles. Un intervalle non borné est modélisé en utilisant la valeur $\top$ pour l'attribut $upper$. Nous introduisons pour cela le type Natural$^\top = \mathbb{N} \cup \{\top\}$ où $\forall e \in \mathbb{N}, e < \top$. Ce type est également défini par l'OMG sous le nom de *UnlimitedNatural* [OMG06b, §12]).

$$lower(c_1 \in RV, r_1 \in RE, n \in Natural^\top) \triangleq \langle MV, ME\rangle \mapsto \\ \forall\langle o, c\rangle \in MV, c = c_1 \Rightarrow card(\{m_2 \in MV \mid \langle\langle o, c_1\rangle, r_1, m_2\rangle \in ME\}) \geq n$$

Une formalisation analogue est définie pour la propriété $upper$ en remplaçant $\geq$ par $\leq$.

Opposite Une référence peut être associée à une autre référence, dite *opposite*. Cela implique qu'un modèle valide devra, pour chaque lien entre deux objets instances de cette référence, avoir un lien en sens inverse entre ces deux mêmes objets typée par la relation opposée.

$$isOpposite(r_1, r_2 \in RE) \triangleq \langle MV, ME\rangle \mapsto \\ \forall m_1, m_2 \in MV, \langle m_1, r_1, m_2\rangle \in ME \Leftrightarrow \langle m_2, r_2, m_1\rangle \in ME$$

Composite Une référence peut être composite. Une instance d'un concept ne peut être cible que d'au plus une instance d'une référence composite. La formalisation de cette propriété nécessite de considérer l'ensemble R des références composites et est donnée par le prédicat *areComposite* pour les objets d'une classe c_1. Une référence composite est également caractérisée par une propriété temporelle qui limite la durée de vie d'un composé à celle du composant. Cette propriété n'est actuellement pas prise en compte par l'expressivité de notre langage de contrainte et sera donc à prendre en compte dans la sémantique d'exécution du DSML.

$$areComposite(c_1 \in RV, R \subseteq RE) \triangleq \langle MV, ME\rangle \mapsto \\ \forall\langle o, c\rangle \in MV, c = c_1 \Rightarrow \\ card(\{m_1 \in MV \mid \langle m_1, r, \langle o, c\rangle\rangle \in ME, r \in R\}) \leq 1$$

Il existe d'autres propriétés que nous ne détaillons pas ici, comme par exemple de ne pas avoir deux références *opposites* qui sont toutes les deux composites. Une autre est la notion d'ensemble ordonné (*isOrdered*). Sa sémantique est temporelle

et n'est pas traitée dans cette partie. Elle doit suivre la définition d'un comportement à travers une sémantique d'exécution (ordre d'insertion et d'accès aux objets de l'ensemble).

8.3.2 Vérification de la métacircularité du MOF

Nous décrivons maintenant comment *EMOF_Core* peut formellement se décrire en lui-même. La vue usuelle de EMOF présentée sur la figure 8.5 correspond à la classe de modèles, sans la fonction de conformité. La notation utilisée (celle du diagramme de classe UML) ne permet de représenter qu'une partie de la sémantique, par exemple les multiplicités sur les références. Nous introduisons maintenant le modèle de EMOF, puis la fonction de promotion permettant d'obtenir la classe de modèles correspondant à EMOF.

Création du modèle MOF à partir de la classe de modèles MOF

La figure 8.6 présente le modèle de *EMOF_Core* avec la notation du diagramme d'objet d'UML. Dans ce diagramme, les attributs avec les valeurs par défaut ne sont pas présentés et les liens ont une notation graphique en fonction de leur type. Tout élément du modèle est une instance d'un concept de la classe de modèles du MOF présenté sur la figure 8.5. Le modèle respecte également les propriétés sémantiques de la classe de modèles.

Chaque concept est instancié comme un objet de type $Class$ et chaque référence ou attribut comme un objet de type $Property$. Une propriété d'un concept, référence ou attribut, est représentée par un lien de l'objet instanciant le concept vers l'objet instanciant la propriété. Il y a aussi un lien de l'objet instanciant la propriété vers l'objet instanciant le type de la propriété. Par exemple, un attribut devra pointer vers un objet instanciant le concept Boolean, String ou Natural$^{\top}$ (c.-à-d. *DataType*).

Promotion du modèle MOF vers la classe de modèles MOF

Nous décrivons maintenant la fonction de promotion qui, à partir du modèle de la figure 8.6 représenté comme un multigraphe typé, construit une classe de modèles. Celle-ci est représentée par une paire composée d'un multigraphe et d'une conjonction de propriétés sémantiques sur les éléments du multigraphe. Cette fonction est définie en trois étapes :

1. nous construisons d'abord les nœuds du graphe de la classe de modèles,
2. nous définissons les références entre ces nœuds,
3. nous enrichissons finalement le graphe résultant avec les propriétés exprimées sur les éléments du graphe.

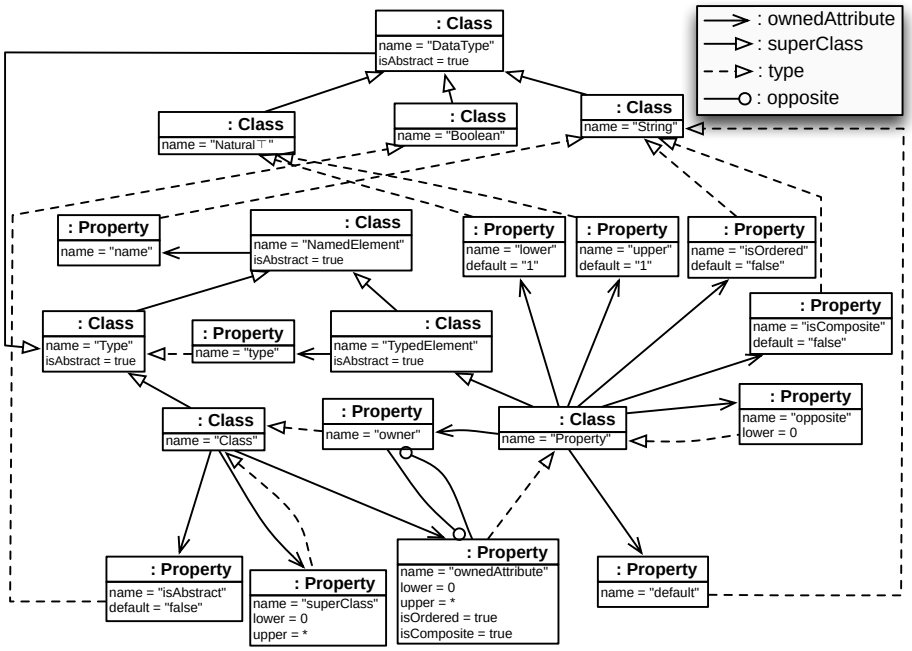


FIGURE 8.6: Modèle (en MOF) de *EMOF_Core* (notation du diagramme d'objet UML)

Construction des nœuds : concepts de la classe de modèles Cette première étape est simple. Nous devons identifier dans le modèle les nœuds qui sont les concepts dans la classe de modèles.

Dans le modèle MOF, nous identifions les objets qui sont de type *Class* comme concept dans la classe de modèles cible. Le champ *name* des objets est utilisé pour identifier les classes. Nous avons donc les concepts suivants : DataType, Natural$^{\top}$...

Plus formellement, soit (MV, ME) le graphe du modèle. Nous construisons l'ensemble RV de la classe de modèles de la manière suivante :

$$RV \triangleq \left\{ \; o.name \,\middle|\, \langle o, Class\rangle \in MV \; \right\}$$

Constructions des arcs : références de la classe de modèles Une fois que les classes ont été construites, la seconde étape consiste à enrichir le graphe avec les arcs entre les nœuds, représentant les références entre les classes. Les arcs sont construits à partir d'une séquence de liens et d'objets dans le modèle MOF.

La règle traduit les chemins d'un objet o_1 de type *Class* à un objet o_2 de type *Property*, par un arc de type *ownedAttribute*, suivi par un arc de type *type* vers une objet o_3 de type $Class$. Chaque chemin est traduit en référence, de la classe associée à l'objet o_1 vers la classe associée à la l'objet o_3. Ils sont étiquetés par l'attribut *name* de l'objet o_2. Ces références seront plus tard associées à des propriétés sémantiques.

Soit $Rule$ le prédicat utilisé comme règle de *matching* pour identifier les chemins.

$$Rule(o_1, o_2, o_3) \triangleq \begin{array}{l} \langle\langle o_1, Class\rangle, \text{ownedAttribute}, \langle o_2, Property\rangle\rangle \in ME \\ \wedge\langle\langle o_2, Property\rangle, \text{type}, \langle o_3, Class\rangle\rangle \in ME \end{array}$$

RE est l'ensemble des arcs dans la classe de modèles construite avec cette règle.

$$RE \triangleq \{\ \langle o_1.name, o_2.name, o_3.name\rangle \,\big|\, Rule(o_1, o_2, o_3)\ \}$$

Enrichissement de la classe de modèles avec les propriétés sémantiques La dernière étape consiste à enrichir le graphe de la classe de modèles résultant par les propriétés sémantiques du modèle. Ces propriétés sémantiques (intégrées à la fonction $conformsTo$) portent sur les éléments du graphe. Chacune de ces propriétés est définie formellement et correspond à une propriété du MOF introduite dans la partie 8.3.1.

Héritage Chaque lien avec un label *superClass* entre deux objets de type $Class$ est traduit comme une relation d'héritage entre les deux classes correspondantes.

$$\bigwedge \begin{array}{l} \{\ subClass(inh, o_1.name, o_2.name)\ \mid \\ \langle\langle o_1, Class\rangle, \text{superClass}, \langle o_2, Class\rangle\rangle \in ME\ \} \end{array}$$

Classes abstraites Chaque classe c résultant de la promotion d'un objet o avec un attribut isAbstract dont la valeur est true a la propriété $isAbstract(c)$. Elle dénote qu'une classe abstraite ne peut pas être instanciée.

$$\bigwedge \begin{array}{l} \{\ isAbstract(inh, o.name)\ \mid \\ \langle\langle o, Class\rangle, \text{isAbstract}, \langle \text{true}, Boolean\rangle\rangle \in ME\ \} \end{array}$$

lower Chaque référence $o_2.name$ entre les classes $o_1.name$ et $o_3.name$, formant le triplet (o_1, o_2, o_3) traduit par la règle de *matching*, a un nombre minimum de $o_2.\text{lower}$ instanciations dans un modèle valide.

$$\bigwedge \{\ lower(o_1.name, o_2.name, o_2.\text{lower}) \,\big|\, Rule(o_1, o_2, o_3)\ \}$$

upper Cette propriété sémantique est définie de manière similaire.

$$\bigwedge \{\ upper(o_1.name, o_2.name, o_2.\text{upper}) \,\big|\, Rule(o_1, o_2, o_3)\ \}$$

ordered Chaque référence construite en utilisant la règle de *matching* et pour laquelle l'objet o_2 a un attribut appelé *isOrdered* avec la valeur à $true$, doit satisfaire la propriété d'ordre. Cette propriété est relative à l'ordre dans lequel l'ensemble des objets cibles de la référence peuvent être accédés et donc implique que les notions de temps et d'exécution soient définies. Cette propriété n'est donc pas traitée dans un premier temps par notre langage de requête.

opposite Chaque référence promue d'un objet o_1 avec un lien typé $\mathrm{opposite}$ vers un autre objet o_2 possède également une référence inverse.

$$\bigwedge \begin{array}{l} \{\, isOpposite(o_1.name, o_2.name) \mid \\ \langle\langle o_1, Property\rangle, \mathrm{opposite}, \langle o_2, Property\rangle\rangle \in ME \,\} \end{array}$$

composite Chaque référence entre classes qui a été construite en utilisant la règle de *matching* et pour laquelle l'objet o_2 a un attribut nommé *isComposite* avec la valeur à $true$ satisfait la propriété de composition. Nous définissons d'abord l'ensemble des relations de composition.

$$\begin{array}{l} compRelations \triangleq \{\, o_2.name \mid \\ \qquad \langle o_2, Property\rangle, \mathrm{isComposite}, \langle \mathrm{true}, Boolean\rangle\rangle \in ME \,\} \end{array}$$

Puis nous définissons la propriété à ajouter à *conformsTo*.

$$\bigwedge \{\; areComposite(o_1.name, compRelations) \,|\, \langle o_1, Class\rangle \in MV \;\}$$

default value Chaque lien entre objets promu en utilisant la règle de *matching* et pour laquelle l'objet o_2 a un attribut nommé *default* ayant pour valeur v permet d'initialiser la propriété. Cette propriété n'est valide que pour l'état initial des objets, et nous ne détaillons donc pas ici sa gestion qui doit être prise en charge par la sémantique d'exécution.

8.3.3 Formalisation de la pyramide de l'OMG

Le cadre de métamodélisation défini permet de formaliser la pyramide proposée par l'OMG (cf. figure 8.7). Les niveaux *Model* et *ModelClass* sont suffisants et nécessaires pour définir de manière claire et formelle les niveaux d'utilisation des modèles (*model*, *metamodel* et *metametamodel*).

Le langage de métamodélisation MOF (c.-à-d. le métamétamodèle) permet de standardiser les concepts utilisés pour décrire les différents langages de modélisation (c.-à-d. les métamodèles). Comme introduit par l'OMG, le MOF a la propriété d'être réflexif, c.-à-d. qu'il peut se décrire lui-même. Cette propriété permet de limiter à trois le nombre de niveau d'utilisation des modèles dans la pyramide de l'OMG. Cette propriété signifie que les classes (c.-à-d. les types) définies dans le MOF sont instances des classes de ce même MOF. D'un point de vue formel, ce n'est pas possible qu'un type soit instance d'un type !

Dans notre approche, nous définissons le MOF (cf. figure 8.7, M3) comme une classe de modèles ($MOF : MC$) telle qu'il est possible de définir un modèle $MOF : M$ conforme à $MOF : MC$ ainsi qu'une *promotion* qui permet exactement d'obtenir la classe de modèles $MOF : MC$.

La description des autres niveaux est similaire. Un langage de modélisation (c.-à-d. un métamodèle) décrit les concepts (c.-à-d. les types) qui peuvent être instanciés en utilisant ce langage. Donc un métamodèle (p. ex. figure 8.7, M2) est

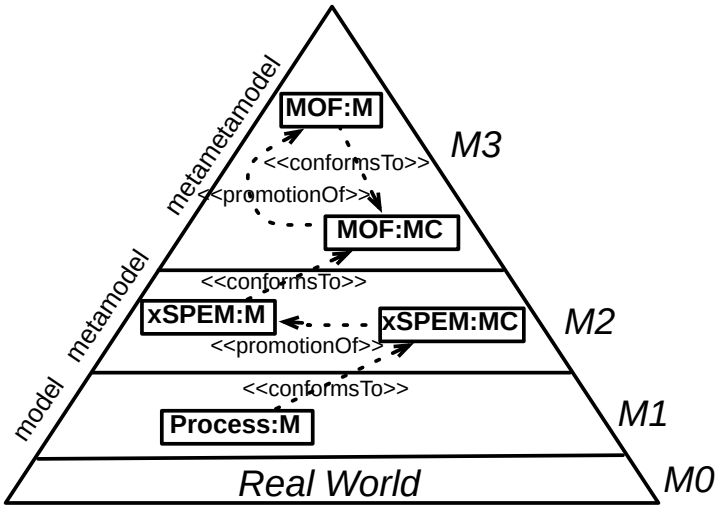


FIGURE 8.7: Formalisation de la pyramide de l'OMG

défini comme une classe de modèles (p. ex. $xSPEM : MC$ pour le langage de modélisation XSPEM) obtenu à partir de la promotion d'un modèle $xSPEM : M$ conforme à $MOF : MC$.

De la même manière, un modèle (p. ex. $Process : M$ au niveau M1 dans la figure 8.7) est conforme à une classe de modèles (p. ex. $xSPEM : MC$) du niveau M2.

Ainsi, le cadre présenté dans ce chapitre permet de formaliser la pyramide de l'OMG et les différentes fonctions des modèles introduites pour distinguer chacun des niveaux (M1, M2 et M3). Néanmoins, la formalisation que nous proposons et qui s'appuie sur la nature des modèles offre un cadre plus général. En particulier, elle n'est pas limitée à un nombre prédéfini de niveaux. Malgré tout, la classification de l'OMG est pertinente et correspond à celle que l'on retrouve dans d'autres espaces techniques comme le *grammarware*.

8.4 Intégration dans un environnement de métamodélisation

Le cadre générique présenté dans ce chapitre en utilisant la théorie des ensembles offre une représentation abstraite et formelle des modèles et des langages de modélisation. Il offre la possibilité de vérifier formellement les modèles construits, en particulier leur conformité par rapport à leur classe de modèles. Ce cadre doit toutefois être intégré dans un environnement de métamodélisation existant de manière à rester transparent pour les (méta)modeleurs.

Nous avons pour cela implanté les différents types présentés en COQ [COQ06] sous la forme d'une bibliothèque que nous appelons COQ4MDE. Nous l'avons également complété d'un sous-ensemble d'OCL (opérateurs de navigation et principaux itérateurs) afin de pouvoir représenter formellement les contraintes OCL exprimées sur les classes de modèles comme sur les modèles (c.-à-d. la sémantique statique, également appelée *context condition*). Nous avons également établi un pont entre l'environnement de métamodélisation EMF et notre cadre formel en COQ. Celui-ci a été établi à l'aide d'une transformation de modèle du sous-ensemble d'Ecore correspondant à *EMOF_Core* vers le code COQ équivalent à l'aide de COQ4MDE. La définition des langages de modélisation et la construction des modèles au sein de l'environnement EMF s'appuient alors sur une représentation abstraite formelle qui reste transparente pour l'utilisateur. Le langage de requête défini permet d'exprimer formellement la sémantique statique.

Cette représentation abstraite en COQ des modèles construits dans l'environnement EMF permet alors de faire la vérification formelle de la conformité d'un modèle par rapport à une classe de modèle. Cette vérification est similaire à celle qui est déjà proposée par EMF et qui vérifie le respect des contraintes statiques à l'aide d'un *model-checker* OCL. Néanmoins, l'approche plus générique de notre cadre et le choix de la formalisation en COQ offrent des perspectives plus larges, comme l'expression de la sémantique d'exécution et donc la possibilité de faire des vérifications dynamiques et formelles. Pour cela, nous travaillons actuellement à l'extension de notre langage de requête afin de pouvoir exprimer des règles de transition sur la structure des classes de modèles. Nous envisageons pour cela de le compléter par les constructions d'un langage de transformations de modèle endogènes. Chaque règle permettrait alors de décrire une transition d'état dans le système.

8.5 Discussion et synthèse

Nous proposons dans ce chapitre une formalisation mathématique d'un cadre générique pour l'IDM reposant sur la nature des modèles. Nous proposons un codage formel basé sur des multigraphes typés, distinguant la nature d'un modèle ($Model$) de celle du langage qui a permis de le construire ($ModelClass$). Nous avons également formalisé deux opérations fondamentales : la promotion et la conformité. Enfin, nous décrivons le noyau d'un langage de requête de manière à pouvoir exprimer la sémantique statique des langages. La formalisation est rendue transparente grâce à une transformation permettant de traduire dans l'environnement COQ un modèle Ecore conforme à une classe de modèles.

Nous prévoyons à court terme d'inclure dans la bibliothèque COQ4MDE les opérations courantes sur les modèles, comme les opérateurs *merge*, *import* et *fusion* et les modèles paramétrés. Nous souhaitons également refléter le formalisme OCL au sein de notre langage de requête de manière à pouvoir le rendre facilement utilisable.

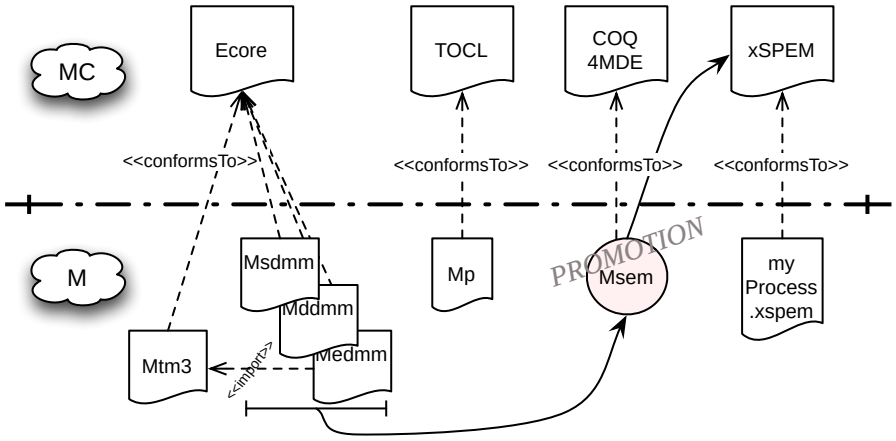


FIGURE 8.8: Description de la promotion dans COQ4MDE

La principale motivation de notre travail réside dans la volonté d'offrir un cadre formel à partir duquel il est envisageable d'utiliser un langage d'action pour formaliser la sémantique d'exécution d'un langage de modélisation. Nous envisageons pour cela d'étendre le langage de requête actuel de manière à pouvoir exprimer des transformations endogènes sur la structure d'une classe de modèles. Le métamodeleur pourrait alors utiliser au sein de l'environnement EMF le langage de métamodélisation Ecore pour décrire la syntaxe abstraite du DSML. Une syntaxe concrète de COQ4MDE lui permettrait ensuite de décrire la sémantique d'exécution de la classe de modèles et de l'intégrer dans la promotion du modèle en MOF permettant d'obtenir le DSML. Cette approche est présentée sur la figure 8.8.

La formalisation d'une sémantique opérationnelle permet alors de la considérer comme celle de référence pour le DSML donné, et ainsi de pouvoir générer automatiquement différentes implantations de la sémantique opérationnelle (p. ex. en Java, en Kermeta, etc.) en fonction des besoins dans l'opérationalisation des modèles. C'est également une condition préalable pour envisager de valider automatiquement l'équivalence des sémantiques par traduction définies pour le même DSML.

Chapitre 9

Transfert des contributions dans le projet TOPCASED

L'ensemble des travaux présenté dans cette thèse se nourrit des besoins exprimés au sein du projet TOPCASED pour répondre aux exigences en termes de validation et de vérification de modèle. Nous avons également trouvé dans cet atelier un cadre idéal pour mettre en application l'approche proposée dans cette thèse de manière à coupler les outils existants à des fonctionnalités de simulation et de vérification. L'atelier offre une solution *open source* et pérenne pour la modélisation et l'opérationalisation des modèles (simulation, vérification, génération de code, génération de documentation, traçabilité, etc). L'atelier s'appuie sur la plate-forme Eclipse [Ecl07], les API graphiques comme GEF [GEF07] et les technologies de l'IDM comme EMF [EMF07], GMF [GMF07], Acceleo [Acc07], ATL [ATL07a], Kermeta [Ker07] ou OpenArchitectureWare [oAW07] (cf. figure 9.1).

Chaque partie de cette thèse a dans un premier temps donné lieu à l'élaboration de prototypes au sein du laboratoire puis à un transfert vers l'atelier TOPCASED, au travers des sociétés partenaires du projet. Nous présentons dans ce chapitre les travaux réalisés ou en cours de réalisation dans TOPCASED et couvrant en partie l'approche proposée par cette thèse. La partie 9.1 présente rapidement les travaux sur l'édition de modèle XSPEM et leur intégration dans TOPCASED. Ces travaux sont réalisés en collaboration avec la société *Tectosages* [1] dans le cadre du lot n°1 de TOPCASED en charge du processus global de l'atelier. La partie 9.2 décrit l'intégration de la chaîne de vérification de processus XSPEM dans l'atelier TOPCASED par l'intermédiaire du bus de modèle. Ces travaux sont en cours de réalisation au travers d'une collaboration avec la société *Sogeti* [2] au sein du lot n°3 de TOPCASED. Enfin, nous détaillons plus particulièrement dans la partie 9.3 le simulateur de machine à états UML2.0 défini dans le cadre du lot n°2 de TOPCASED et en cours

1. cf. http://www.tectosages.com
2. cf. http://www.sogeti.com

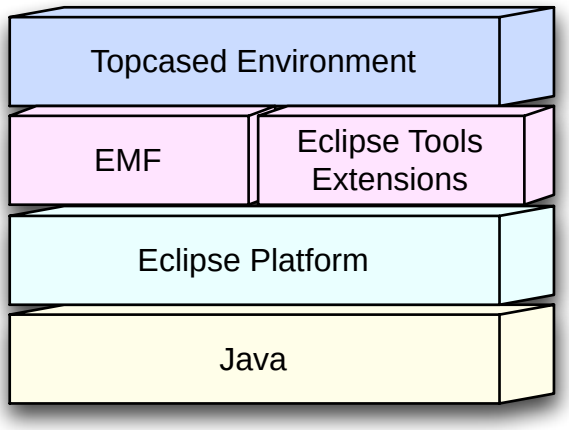


FIGURE 9.1: Intégration de TOPCASED à la plateforme Eclipse

d'implantation par les sociétés *Anyware Technologies* [3] et *Atos Origin* [4].

Les transferts de technologie présentés dans ce chapitre sont le fruit d'une forte volonté d'intégrer au sein de l'atelier TOPCASED les réflexions issues de cette thèse. Ce travail n'aurait pas pu avoir lieu sans le soutien des partenaires du projet. Il est également le résultat d'efforts technologiques fournis par de nombreux stagiaires pour proposer des prototypes supports de nos réflexions. Je tiens particulièrement à souligner les travaux de :

- Jean-Noël Guyot [Guy07] et Boris Libert [Lib07] pour leurs apports dans la modélisation de processus,
- Darlam Bender [Ben08] et Youssef Dkiouak [Dki07] pour leurs contributions respectives dans la vérification de modèle,
- Jonathan Math [Mat07b], Christophe Bez [Bez07] et Vincent Gaillaud pour leurs développements sur les outils de simulation de modèle.

Les travaux présentés dans ce chapitre ont donné lieu à deux publications à ERTS 2008 (*4th European Congress* ERTS – EMBEDDED REAL TIME SOFTWARE). La première sur l'environnement de modélisation de processus XSPEM [GCC+08], et la deuxième sur la mise en oeuvre d'un simulateur de machine à états UML2.0 [CCG+08b].

3. cf. http://www.anyware-tech.com
4. cf. http://www.atosorigin.com

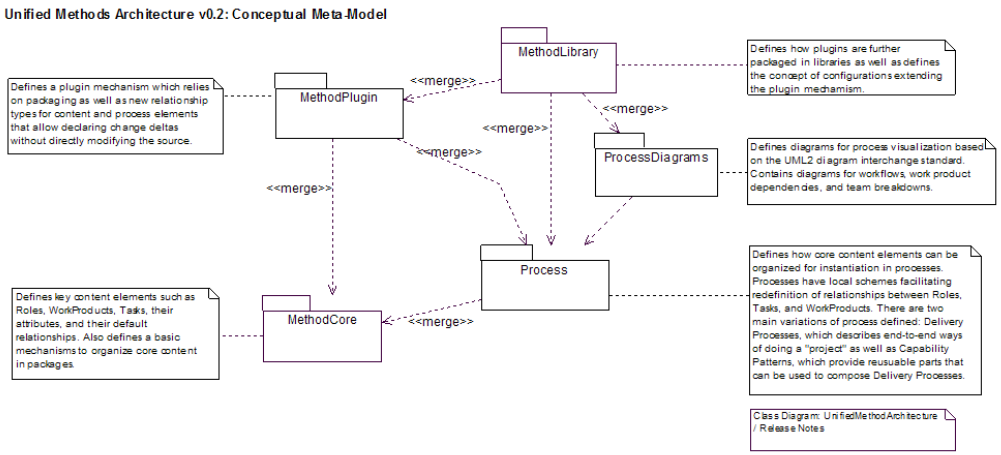


FIGURE 9.2: Architecture du métamodèle UMA

9.1 *TopProcess* : Outils de modélisation de processus pour xSPEM

TopProcess [5] est un sous-projet du lot n°1 de TOPCASED, au sein duquel nous collaborons plus particulièrement avec la société *Tectosages*. Ces travaux ont pour objectif de fournir les outils nécessaires à la mise en place d'un développement dirigé par les modèles et guidé par les processus.

Nous avons dans un premier temps contribué à l'implantation d'xSPEM sous la forme d'un métamodèle décrit en Ecore. *xSPEM_Core* a été défini de manière à être compatible avec SPEM2.0. Nous avons toutefois repris la structuration du métamodèle proposée dans UMA (*Unified Method Architecture*), le métamodèle compatible SPEM2.0 implanté dans le *plugin* EPF (*Eclipse Process Framework project*) [EPF07]. La figure 9.2 décrit cette structuration dont les principales différences avec le standard de l'OMG sont :

- UMA contient moins de paquetages que SPEM2.0.
- SPEM2.0 utilise UML2.0, alors qu'UMA redéfinit le noyau minimal nécessaire. Il est ainsi défini comme un DSML, indépendant de l'évolution d'UML.
- SPEM2.0 ne contient que des types de guides (*Guidance*) très génériques, alors qu'UMA définit des types plus spécifiques.
- Un ensemble de 21 classes a été défini dans UMA pour les éléments graphiques de manière à les standardiser.

Les autres paquetages décrits dans le chapitre 5 ont également été définis de manière à compléter *xSPEM_Core*.

Nous avons également réalisé deux prototypes d'éditeur. Le premier à partir du

5. cf. https://gforge.enseeiht.fr/projects/topcased-spem

générateur de TOPCASED et le second à partir du générateur de GMF. Ils permettent soit de définir entièrement un processus, soit de compléter un modèle défini à l'aide d'EPF. Ils offrent une modélisation différente des processus. A la différence d'EPF qui propose de saisir les informations sous la forme de tables et de générer à partir d'elles les modèles graphiques, nous proposons des éditeurs graphiques supportant plusieurs vues (c.-à-d. diagrammes). Le premier périmètre était de se concentrer sur deux vues principales :

- *Le diagramme d'activité* qui permet de décrire un processus en fonction d'un enchaînement partiellement ordonné d'activités à réaliser.
- *Le diagramme des tâches* qui permet de décrire les activités en fonction des états des produits qu'elle a en entré et/ou en sortie, des rôles, des guides, etc.

Ces éditeurs nous ont permis de construire les modèles de processus utilisés dans nos travaux sur la simulation et la vérification. Ils constituent également un composant utilisé pour le développement de l'animateur de processus.

9.2 *Tina Bridges* : Intégration d'une chaîne de validation pour les modèles de processus

TinaBridges [6] est un sous-projet du lot n°3 de TOPCASED qui vise à fournir les *ponts* des différents DSML de l'atelier vers des domaines formels tels que les réseaux de Petri ou les automates en s'appuyant sur les services offerts par l'atelier. Plusieurs approches ont pour cela été envisagées. L'une d'entre elles vise à définir les services sur le bus de modèles permettant de traduire un modèle et ses propriétés de manière transparente pour l'utilisateur vers la syntaxe concrète d'un outil tel que TINA [TIN07] ou CADP [CAD07]. Les transformations définies dans le chapitre 6 qui permettent la vérification de modèle de processus grâce au *model-checker* de TINA sont actuellement en cours d'intégration dans l'atelier TOPCASED. Les premiers objectifs sont de pouvoir appeler les outils de *model-checking* de manière transparente pour l'utilisateur et de partager les projecteurs (transformations ou modèles TCS) vers les syntaxes concrètes particulières (figure 9.3). Les propriétés définies en TOCL sont traduites automatiquement en LTL et vérifiées sur le réseau de Petri résultant de la traduction du modèle. Le premier objectif est de fournir à l'utilisateur le résultat booléen de la vérification de chacune des propriétés sur son modèle. D'autres perspectives sont détaillées dans le dernier chapitre de cette thèse et visent à fournir un retour automatique des résultats de la vérification vers le modèle initial (comme les contre-exemples qui pourraient ensuite être injectés dans le simulateur).

Ces travaux s'inscrivent dans le cadre d'un transfert technologique de la part des laboratoires IRIT et LAAS auprès de la société *Sogeti*.

6. cf. https://gforge.enseeiht.fr/projects/tina-bridges

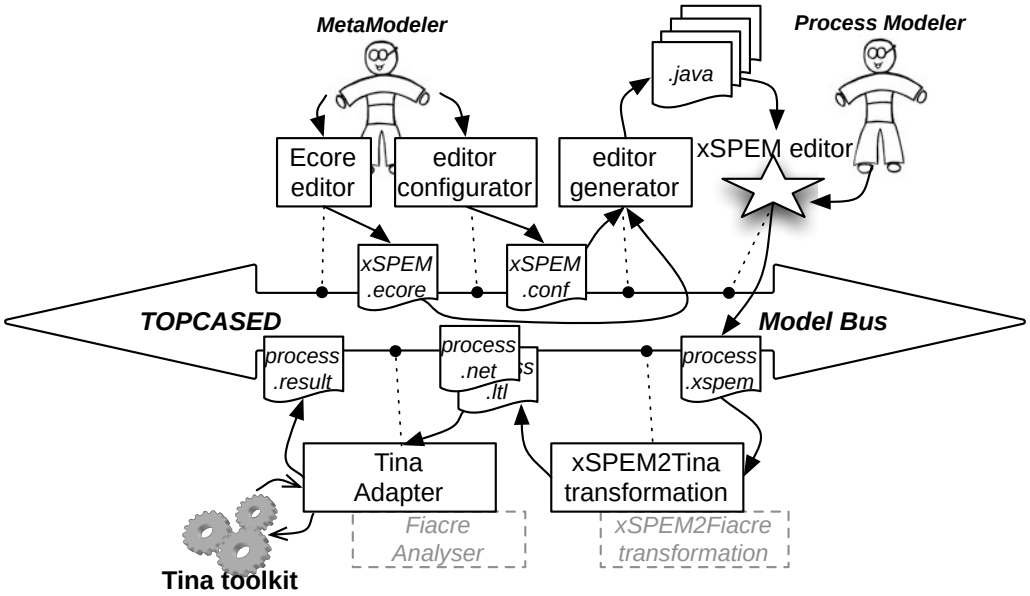


FIGURE 9.3: Intégration d'une chaîne de vérification de modèle dans l'atelier TOPCASED

9.3 TOPCASED *Model Simulation* : Simulation de machines à états UML2.0

TOPCASED *Model Simulation*[7] fait partie du lot n°2 du projet TOPCASED. Les travaux préliminaires ont permis de développer une première version d'un simulateur de machine à états UML2.0. Il permet d'animer un modèle construit à partir de l'éditeur fourni par l'atelier TOPCASED (basé sur le *plugin* Eclipse UML2 [MDT07]). Ces résultats s'inspirent des réflexions présentées dans le chapitre 7 et résultent d'un effort de transfert technologique auprès des sociétés *Anyware Technologies* et *Atos Origin*. A partir d'un prototype développé au sein de l'équipe, un outil industriel et intégré à l'atelier TOPCASED est en cours d'élaboration. Nous présentons, au regard de l'approche de cette thèse, l'implication dans la mise au point de celui-ci. Nous détaillons dans un premier temps le métamodèle retenu (section 9.3.1) et présentons ensuite les différents composants développés (section 9.3.2).

7. cf. https://gforge.enseeiht.fr/projects/topcased-ms

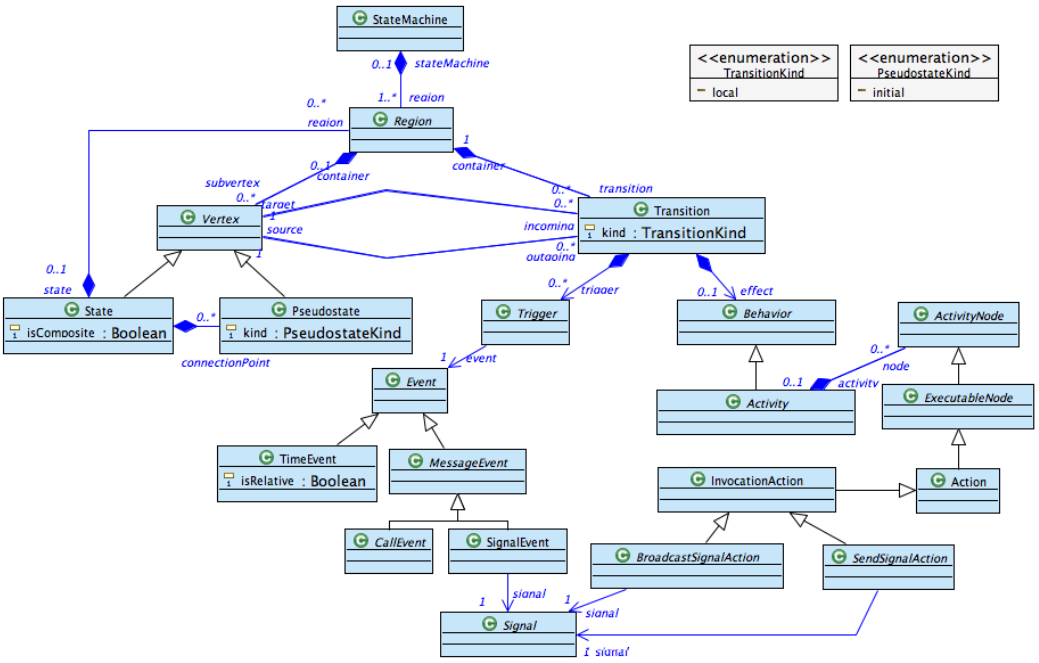


FIGURE 9.4: DDMM retenu dans TOPCASED pour la simulation de machine à états UML2.0

9.3.1 Métamodèle retenu dans TOPCASED pour la simulation de machine à états UML2.0

Domain Definition MetaModel

Pour la première version de l'animateur, nous avons ciblé un sous-ensemble des machines à états d'UML2.0 dont le métamodèle est donné sur la figure 9.4. Ce métamodèle décrit les propriétés structurelles des machines à états. Afin de clarifier leur présentation, nous allons l'illustrer à travers un exemple qui introduit les concepts pris en compte pour la simulation. Cet exemple est donné sur la figure 9.5 et décrit une unique machine à états car l'animateur n'est actuellement capable que de simuler une seule machine à la fois. Le prochain paragraphe détaille cet exemple et souligne les concepts pris en compte dans la simulation et donc présents dans le métamodèle (entre parenthèse et en *italique*).

La machine à états (*StateMachine*) de la figure 9.5 modélise les feux clignotants d'une voiture. C'est une machine à états concurrente composée de quatre régions (*Region*) : une pour le feux gauche, une pour le feux droit, une pour la poignée et une pour l'horloge interne. Les feux droit et gauche partagent la même machine à états et pourraient être considérés comme deux instances du même concept *Feu*. Les travaux sur la gestion d'instances dans le simulateur sont actuellement en cours, et nous avons donc dû dans un premier temps dupliquer la machine à états et renommer les signaux (*Signal*) qui déclenchent (*Trigger*) les transitions (*Tran-*

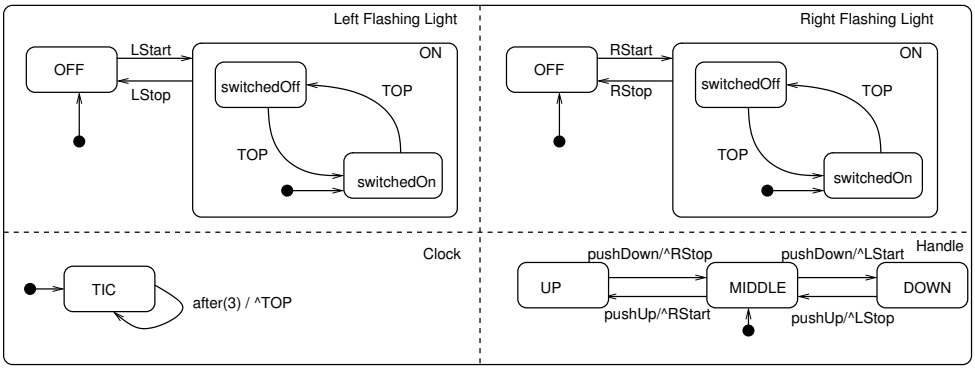


FIGURE 9.5: Machine à états pour des feux clignotants

sition). Aussi, le signal *Lstart* démarre le feux gauche et le met dans l'état appelé *ON*. Cet état est composé d'une région avec les deux sous-états *switchedOff* et *switchedOn* qui indiquent si la lumière est allumée ou non. Le feux passe d'un état à l'autre en fonction du signal *TOP* généré par l'horloge. Enfin, la poignée réagit en fonction des signaux envoyés par l'utilisateur *pushUp* et *pushDown* et de sa position – *down*, *middle* ou *up* – pour allumer ou éteindre le feux gauche ou droit.

Nous avons considéré dans un premier temps uniquement les transitions locales, c.-à-d. les transitions dont l'état source et l'état cible appartiennent à la même région. Une transition est déclenchée par des événements (*Event*) et peut exécuter une activité quand elle est franchie. Les activités (*Activity*) sont décomposées en actions (*Action*). La première action que nous avons prise en compte (*BroadCastSignalAction*) permet d'envoyer le signal de manière globale à tout le modèle. Un événement correspond soit à un signal (*SignalEvent*) soit à une date (*TimeEvent*). La transition dans la région correspondant à l'horloge définit un événement temporel indiquant qu'elle sera franchie au bout de 3 unités de temps à partir de l'entrée dans l'état *TIC*. Les transitions dans la région *Handle* correspondant à la manette sont déclenchées soit par le signal *pushDown*, soit pas le signal *pushUp*. Quand elles sont franchies, elles exécutent une action qui émet en diffusion un signal faisant évoluer les régions qui correspondent aux feux.

Notons que tous les éléments qui ne sont pas pris en compte par la simulation peuvent être présents dans le modèle construit à partir de l'éditeur graphique (M_{DDMM}) mais seront ignorés dans le modèle dynamique (M_{SDMM}). Une analyse en utilisant des contraintes OCL devra être réalisée pour avertir l'utilisateur des éléments non supportés (et du risque d'échecs de la simulation).

States Definition MetaModel

Conformément à l'approche définie dans cette thèse, nous avons complété le métamodèle défini par l'OMG de manière à prendre en compte les informations dynamiques caractérisant l'état du système (M_{SDMM}). Par exemple, nous avons

besoin de connaître l'état courant de chaque région active, la liste des signaux en attente, de pouvoir dire si une transition peut être franchie ou pas, etc.

Cette partie de la syntaxe abstraite vient compléter celle du DDMM. L'opérateur *merge* utilisé dans l'approche présentée dans cette thèse pour structurer les différentes préoccupations n'est actuellement pas disponible dans EMF. C'est pourquoi nous avons préféré définir des métamodèles différents avec des liens entre eux (le SDMM a des liens vers le DDMM). Une transformation de modèle en ATL permet de construire un M_{SDMM} à partir d'un M_{DDMM}.

Events Definition MetaModel

Pour l'exécution des machines à états UML2.0, le EDMM contient dans un premier temps un seul événement « injecter un signal ». Il rajoute un signal à ceux qui doivent être traités par la machine à états. Cet événement peut être à la fois exogène et endogène. Dans notre exemple, cet événement est exogène pour les signaux *pushDown* et *pushUp* de la manette. Il est endogène pour les signaux comme *RStart* qui sont émis lors du franchissement d'une transition de la région *Handle*.

Des travaux en cours intègrent les notions de classe et d'instance. Cela conduit à distinguer deux événements différents pour injecter un signal. Le premier diffuse le signal à toutes les machines du système (*broadcast signal*) et le second cible la machine à états d'une instance particulière (*send signal*).

Grâce à ces événements et en s'appuyant sur le TM3, on peut définir des scénarios et des traces d'exécution comme des séquences d'instances des événements décrits ci-dessus. La figure 9.6 présente un exemple de scénario (figure 9.6a) qui consiste à lever puis baisser la manette. Elle donne aussi une trace possible exprimée sous forme textuelle (figure 9.6c) et de diagramme de séquence UML (figure 9.6b).

9.3.2 Prototype du simulateur de machines à états de TOPCASED

A partir de la définition du métamodèle ci-dessus, nous avons établi les différents composants d'un simulateur selon l'architecture décrite dans le chapitre 7.

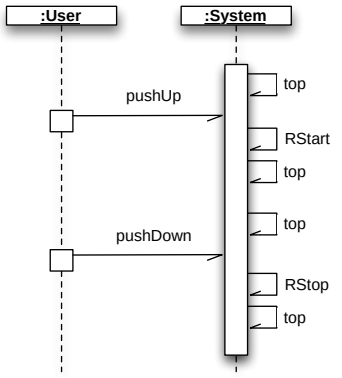
Constructeur de scénario

Le EDMM définit les différents événements qui peuvent survenir au cours d'une simulation. Ils sont spécifiques à un DSML donné. Par exemple, pour les machines à états, le seul événement externe consiste à pouvoir injecter un nouveau signal à prendre en compte par les différentes régions.

Un éditeur graphique d'événement peut être utile pour déclencher des événements au cours de l'animation. Une solution est de remplacer ou de compléter la palette d'édition de l'éditeur par ces événements. Ceci peut être fait automatiquement à partir de la définition du EDMM.

injecter(pushUp) @ 2
injecter(pushDown) @ 8

(a) Scénario



(b) Diagramme de séquence de la trace

injecter(top) @ 1
injecter(pushUp) @ 2
injecter(RStart) @ 3
injecter(top) @ 4
injecter(top) @ 7
injecter(pushDown) @ 8
injecter(RStop) @ 9
injecter(top) @ 10
...

(c) Trace

FIGURE 9.6: Exemple d'un scénario et d'une trace pour les machines à états

Moteur de simulation

Le moteur de simulation à événements discrets décrit dans la section 7.2.3 a été implanté sous la forme d'un *plugin* Eclipse générique pouvant être réutilisé par les différents simulateurs. Il offre pour cela le point d'extension [8] permettant dans un autre *plugin* de spécialiser l'interpréteur par la sémantique particulière à chaque DSML. La sémantique est donc décrite dans un deuxième *plugin* spécifique à chaque DSML. Dans le cas des machines à états UML2.0, la sémantique retenue est décrite dans [Gro07] et s'inspire des travaux d'Harel *et al.* [HN96] dans l'outil Statemate.

Panneau de contrôle

Le moteur de simulation interprète les événements du scénario en fonction des interactions de l'utilisateur à travers le panneau de contrôle. Celui-ci permet de démarrer ou stopper la simulation, la faire avancer pas à pas, etc. Dans la première version du simulateur, nous avons défini une vue spécifique sous la forme d'un *plugin* qui pilote le moteur à partir de son API.

Animateur de modèle

Comme le concepteur a défini son modèle en utilisant l'éditeur de TOPCASED, nous avons réutilisé la même représentation graphique afin de montrer l'état courant du modèle au cours de l'animation. Les informations dynamiques sont affichées sur la représentation graphique du modèle. Par exemple, les états courants sont affichés en rouge, les transitions franchissables en vert, etc.

8. Dans Eclipse, les points d'extension sont les points de communication avec d'autres *plugin*

L'animation est réalisée à l'aide du moteur de simulation qui modifie les informations dynamiques du modèle. A chaque modification, le mécanisme d'observateurs d'EMF est utiliser pour mettre à jour l'affichage de l'animateur. Le principal avantage de cette approche est qu'elle permet de réutiliser les configurateurs définis pour les éditeurs graphiques. Les éléments graphiques pour l'animation sont seulement rajoutés comme décoration sur les éléments graphiques de l'éditeur.

Le modèle animé peut également être exploré à travers la vue des propriétés fournie par la plateforme Eclipse et qui est automatiquement active quand on sélectionne la *perspective* TOPCASED.

D'autre outils de visualisation peuvent être élaborés à partir de cette approche mais ils ne sont pas encore implémentés.

Interactions entre les outils

Les différentes parties de la syntaxe abstraite (DDMM, SDMM, EDMM et TM3) ont été définies en utilisant EMF à l'aide du langage de métamodélisation Ecore [BSE03]. Les modèles sont ainsi soit stockés en mémoire, comme des objets Java instances des classes générées par EMF (ou créés par l'API réflexive dynamique), soit sérialisés en fichiers XMI.

TOPCASED s'appuie sur un bus de modèles afin de faciliter l'échange entre les composants mais nécessite pour cela de sérialiser les modèles. Pour que l'échange entre les différents éléments du simulateur ne soit pas trop coûteux, nous avons préféré partager les modèles en mémoire selon une représentation EMF. Cette approche s'appuie sur les observateurs générés par EMF pour garder un couplage fort entre les composants.

Prototype courant

Un premier prototype de simulateur de machine à états UML2.0 a été développé. Une capture écran du simulateur en cours d'exécution est montrée sur la figure 9.7. Il réutilise la visualisation graphique offerte par l'éditeur développé dans l'atelier TOPCASED. Au cours de la simulation d'un modèle, il est possible de visualiser les informations dynamiques. Il est aussi possible d'utiliser la vue des propriétés fournie par Eclipse. Elle est particulièrement utile pour les propriétés qui ne sont pas représentées graphiquement. Un panneau de contrôle est également disponible sous la forme d'une vue indépendante. Il permet de démarrer, arrêter et avancer pas-à-pas la simulation. La simulation est pilotée interactivement ou par un scénario préétabli choisi à l'avance. Un mode *batch* est également disponible.

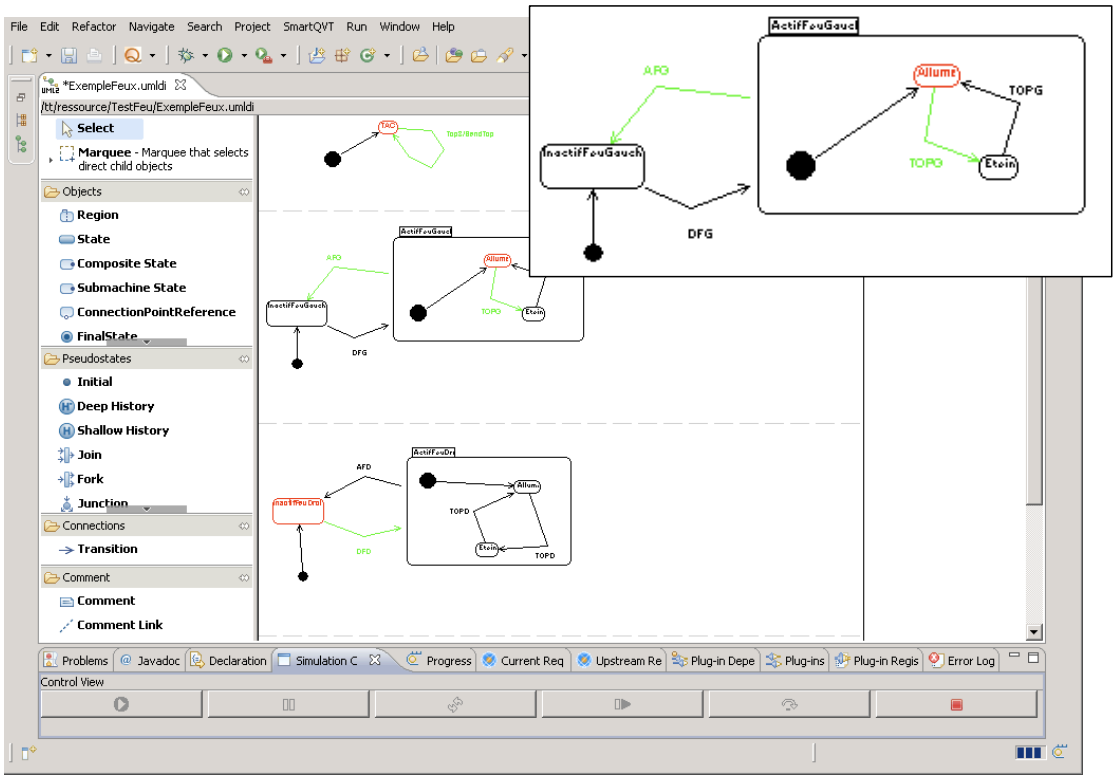


FIGURE 9.7: Capture écran du prototype de simulateur pour les machines à états UML2.0

Chapitre 10

Conclusion générale

Ce chapitre présente un bilan des contributions apportées par cette thèse (section 10.1) que l'on retrouve de manière détaillée à la fin de chacun des chapitres précédents. Ce bilan est illustré à travers la figure 10.1 qui reprend l'ensemble des étapes de la démarche que nous avons introduite pour définir un DSML « exécutable » et les outils permettant de vérifier et simuler ses modèles. Nous faisons ensuite une critique de nos travaux et présentons les perspectives qu'ils ont permis d'ouvrir ainsi que les voies de recherche que nous suivons actuellement (section 10.3).

10.1 Bilan et applications des travaux

Nous proposons dans cette thèse une démarche outillée pour construire un DSML « exécutable » (c.-à-d. dont il est possible d'exécuter les modèles) ainsi que les outils permettant de simuler et de vérifier formellement les modèles. L'objectif principal est de réduire l'effort au maximum de manière à pouvoir prendre en compte différents DSML, comme par exemple au sein de l'atelier TOPCASED.

Les contributions issues de ces travaux de thèse sont :

- la définition d'une taxonomie précise des techniques pour exprimer la sémantique d'exécution d'un DSML.
- une démarche méthodologique pour la construction d'une syntaxe abstraite et plus globalement d'un DSML exécutable.
- une démarche pour exprimer et valider la traduction vers un domaine formel à des fins de vérification de modèle.
- une architecture générique pour la simulation de modèle par sémantique opérationnelle.
- un cadre formel de métamodélisation implanté à travers l'assistant de preuve COQ pour formaliser les modèles et leurs DSML et ainsi pouvoir valider les différentes implantations de la sémantique.

L'ensemble de ces travaux sont illustrés dans les différents chapitres de cette thèse à travers le domaine des procédés de développement. Après une étude ap-

profondie de cette ingénierie, et plus particulièrement du langage de modélisation SPEM proposé par l'OMG, nous définissons l'extension XSPEM à partir de laquelle il est possible de construire des modèles exécutables. Nous décrivons également l'ensemble des outils nécessaires à la vérification formelle et à la simulation d'un modèle de procédé. La description de XSPEM et les différents outils sont disponibles à l'adresse http://combemale.perso.enseeiht.fr/xSPEM.

La démarche est toutefois générique. Elle a ainsi contribué à l'établissement des fonctionnalités de vérification et de simulation pour différents langages de l'atelier TOPCASED. Les transferts de nos travaux au sein de ce projet sont présentés dans le chapitre 9 et comprennent :

- l'intégration des travaux sur l'ingénierie des procédés et sur l'opérationalisation des modèles XSPEM dans le *WorkPackage* 1 du projet TOPCASED et plus particulièrement dans le sous-projet TOPPROCESS [1].
- l'intégration sur le bus de modèles TOPCASED des projecteurs pour TINA dans le *WorkPackage* 3 du projet TOPCASED et plus particulièrement dans le sous-projet TINABRIDGES [2].
- la définition de simulateurs de modèles pour les langages UML2 et SAM dans le *WorkPackage* 2 du projet TOPCASED [3].

10.2 Description générale de la démarche de métamodélisation pour la simulation et la vérification de modèle

Résultant d'expérimentations menées pour définir et évaluer les différentes techniques permettant d'exprimer une sémantique d'exécution pour un DSML, la démarche que nous proposons dans cette thèse se compose de quatre étapes principales (cf. figure 10.1), offrant chacune des outils génériques ou génératifs pour aider le (méta)concepteur à la réaliser.

- La première étape consiste à **définir de manière structurée la syntaxe abstraite** du DSML en prenant en compte les différentes préoccupations nécessaires pour exécuter, simuler et vérifier un modèle. A partir de la description des concepts du domaine (*Domain Definition MetaModel* – DDMM), nous proposons une approche dirigée par les propriétés afin de déterminer la bonne abstraction des informations dynamiques qui permettront ensuite de capturer l'état du système au niveau du modèle (*States Definition MetaModel* – SDMM). Enfin, ces deux premiers éléments sont étendus par le EDMM (*Events Definition MetaModel*) décrivant les interactions (c.-à-d. les événements) possibles entre l'environnement et le système au cours de son exécution. Il utilise pour cela TM3 (*Trace Management MetaModel*), un métamodèle générique permettant de capturer la séquence d'interaction d'une exécution particulière.

1. cf. http://gforge.enseeiht.fr/projects/topcased-pe/
2. cf. http://gforge.enseeiht.fr/projects/tina-bridges/
3. cf. http://gforge.enseeiht.fr/projects/topcased-ms/

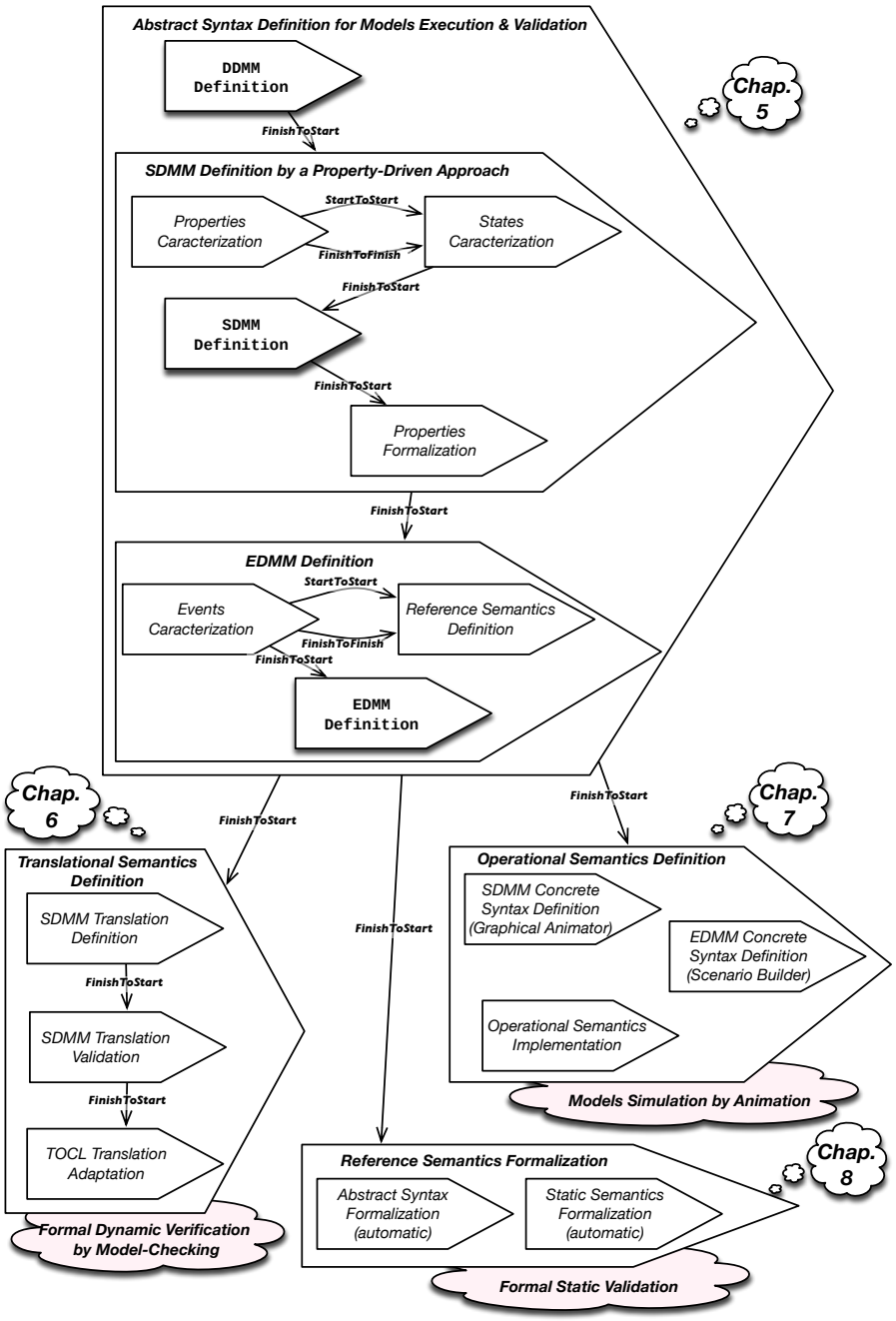


FIGURE 10.1: Modèle XSPEM de la démarche globale pour la validation et la vérification de modèle

- La deuxième étape a pour objectif de permettre de vérifier formellement les modèles. Elle consiste à **définir une sémantique par traduction vers un domaine formel** tel que les réseaux de Petri. Cette étape correspond à la définition d'un compilateur de modèle. Les propriétés définies dans l'étape précédente peuvent être traduites automatiquement de manière à être vérifiées sur le modèle obtenu par traduction. Toutefois, afin de s'assurer que les conclusions obtenues sont cohérentes, nous détaillons la preuve de bisimulation qu'il est nécessaire d'établir entre le modèle initial et le modèle obtenu par traduction.
- La troisième étape a pour objectif de permettre d'animer des modèles. Elle consiste à **définir la sémantique d'exécution comme l'ensemble des réactions à chaque événement défini dans le EDMM.** Ceci équivaut à décrire la sémantique comportementale par un système de transition. Pour simuler le modèle, cette sémantique est prise en compte par un ensemble de composants réutilisables que nous avons conçus (moteur de simulation à événements discrets et panneau de contrôle) s'appuyant sur les concepts génériques du TM3. Les outils graphiques comme l'animateur et le constructeur de scénario sont engendrés à partir des décorations graphiques définies sur les éléments du SDMM et du EDMM et viennent compléter l'éditeur graphique obtenu à partir du DDMM comme c'est actuellement le cas dans TOPCASED.
- La quatrième étape consiste à **traduire automatiquement les DSML et les modèles vers une représentation abstraite formelle** que nous avons définie à travers la bibliothèque COQ4MDE. L'environnement COQ permet alors de vérifier formellement la conformité d'un modèle par rapport à son langage. COQ4MDE a été défini de manière à pouvoir être étendu facilement pour exprimer la sémantique d'exécution d'un DSML et utiliser alors l'environnement COQ pour vérifier le comportement d'un modèle.

10.3 Élargissement des résultats

Les travaux présentés dans cette thèse s'inscrivent dans un champ encore peu exploré de la recherche. Ils contribuent à l'étude de l'ingénierie de la sémantique pour les langages de modélisation. Un consensus est encore loin d'être trouvé, et nécessite pour cela de poursuivre les études sur ce domaine.

Nous pensons que l'approche présentée dans cette thèse permet de donner un cadre rigoureux pour définir les outils de vérification et de simulation pour un DSML. Ces travaux ouvrent maintenant de nombreuses perspectives, à court terme comme à long terme, afin d'aider le métaconcepteur à définir l'ensemble des aspects d'un DSML.

10.3.1 Environnement d'expression des propriétés temporelles

Il nous semble intéressant d'étendre notre étude sur TOCL afin de pouvoir exprimer des propriétés aussi bien sur la syntaxe abstraite du langage que sur les modèles construits. Ces propriétés peuvent ensuite être prises en compte de la même manière dans la traduction automatique vers un langage tel que LTL. L'ensemble de ces propriétés peut également être pris en compte dans la simulation afin de mieux appréhender les erreurs, en complément de l'animation graphique du modèle sur des exécutions particulières. Pour cela, il faut fournir les outils d'analyse qui permettront d'évaluer les propriétés sur chacun des états de l'exécution. Ces outils peuvent être génériques et intégrés à l'architecture d'un simulateur de modèle proposé dans le chapitre 7.

10.3.2 Analyse des résultats de vérification

Notre approche consiste à compiler le modèle dans un domaine exécutable au sein duquel on profite des outils disponibles pour exécuter, vérifier ou simuler le modèle. Ainsi, au même titre que le *debugger* d'un langage de programmation permet de visualiser, au cours de l'exécution, les informations du programme, il est souhaitable de pouvoir afficher les résultats des outils sur le modèle initial. Cependant, nous n'avons pas abordé dans cette thèse la problématique de l'analyse des résultats fournis par les outils réutilisés grâce à une sémantique par traduction.

Cette problématique fait actuellement l'objet de nombreux travaux, qui abordent aussi bien la bidirection que la traçabilité des transformations de modèle. L'approche qui nous semble la plus pertinente consiste à définir la sémantique par traduction non pas comme une transformation de modèle mais comme un modèle de lien [FBJ+05] entre la syntaxe abstraite du DSML source et celle du domaine cible. Un tel modèle pourrait être construit à partir d'un outil comme AMW (*Atlas Model Weaver*) [AMW07]. Un interpréteur dédié à la sémantique par traduction, mais générique à tous les DSML grâce à l'architecture introduite dans cette thèse, pourrait générer à partir d'un modèle de lien l'ensemble des transformations, dépendantes les unes des autres grâce à un modèle de trace (construit par certaines et utilisé par d'autres).

Cette approche permettrait de rendre transparente l'utilisation d'un formalisme différent pour la vérification formelle d'un modèle. Elle permettrait également de coupler la vérification (par compilation) et la simulation (par interprétation). En effet, le contre-exemple fourni par le *model-checker* pourrait être interprété comme une trace sur le modèle initial et ainsi être simulé, par exemple en mode pas à pas.

10.3.3 Extension du cadre de simulation

Les travaux que nous avons menés dans TOPCASED ont permis d'établir le prototype pour le premier périmètre de la simulation de machine à états UML2. Ces travaux sont actuellement poursuivis pour étendre le métamodèle d'UML2 pris en compte. Nous travaillons plus particulièrement sur le lien entre les machines à

états et le diagramme de classe afin de pouvoir instancier la même machine à états pour chaque objet d'une même classe.

La structuration de la syntaxe abstraite introduite dans cette thèse nous permet de bien distinguer les informations spécifiques à chaque instance de machine à états (celles du SDMM), des informations qui peuvent être partagées entre chaque instance (celles du DDMM). Les premiers résultats obtenus sont présentés dans [Bez07].

10.3.4 Sémantique d'exécution avec création dynamique

Une des limites de nos travaux réside dans les sémantiques d'exécution pour lesquelles il y a, à l'exécution, modification de la structure du modèle. Nous nous sommes restreints aux sémantiques d'exécution où seule la valeur des informations dynamiques évolue. Par exemple, dans XSPEM, nous ne permettons pas de créer à l'exécution de nouvelles activités.

Nous avons choisi de nous restreindre à de telles sémantiques en raison des limitations imposées par les méthodes de *model-checking* et des difficultés ergonomiques pour l'animation graphique de modèles. Malgré tout, dans le cas d'une simulation en mode *batch*, il est intéressant de pouvoir étendre l'expressivité de l'approche afin de définir des sémantiques permettant la création dynamique. Par exemple, pour les machines à états UML2, il est souvent indispensable de créer dynamiquement de nouveaux objets, instanciant alors au cours de l'exécution de nouvelles machines à états à simuler.

10.3.5 Environnement de métamodélisation formelle

La cadre formel que nous avons défini permet d'exprimer la syntaxe abstraite et la sémantique statique des DSML. L'objectif majeur de ce travail est de fournir un environnement de métamodélisation formelle, qui peut facilement être étendu pour exprimer la sémantique d'exécution des DSML. C'est pour cela que nous avons fait le choix d'implanter ce cadre sous la forme d'une bibliothèque COQ et d'un langage de requête dont la syntaxe est proche d'OCL. Nous travaillons actuellement à l'extension de ce langage afin de pouvoir l'utiliser pour exprimer formellement la sémantique d'exécution.

Nous avons choisi d'étendre ce langage de requête à un langage de transformation dédié pour l'expression de transformations endogènes sur la syntaxe abstraite des DSML. Nous pensons ainsi permettre l'expression formelle de la sémantique d'exécution sous la forme d'un système de transition couplé aux événements décris dans le EDMM. Cette sémantique pourrait être considérée comme la référence afin de valider la cohérence entre les différentes sémantiques (opérationnelle et par traduction) d'un même DSML. Il est également envisageable de l'utiliser pour générer automatiquement différentes sémantiques opérationnelles, visant chacune une plateforme d'exécution et des besoins différents (p. ex. Kermeta, Java, etc.). Cette approche est illustrée sur la figure 10.2.

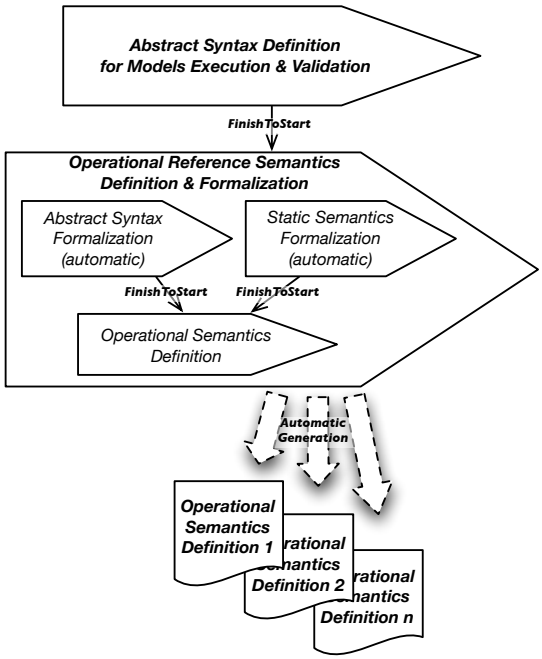


FIGURE 10.2: Formalisation de la sémantique opérationnelle de référence

10.3.6 Administration de systèmes dirigée par les modèles

La démarche introduite dans cette thèse pour construire un DSML exécutable a été exploitée pour valider et vérifier un système au plus tôt dans le cycle de développement à partir des modèles construits. Nous travaillons actuellement sur une autre utilisation de l'exécutabilité des modèles et de nos travaux sur la simulation. Il s'agit d'utiliser les modèles pour administrer de manière homogène et plus abstraite des systèmes de plus en plus complexes, hétérogènes et distribués. Ces modèles nécessitent alors de capturer l'évolution du système au cours de son exécution. L'architecture générique que nous proposons pour structurer la syntaxe abstraite peut être utilisée pour fournir des outils génériques, par exemple de reconfiguration automatique. Nous travaillons pour cela sur l'établissement de modèles de reconfiguration, exécutés en fonction d'événements endogènes particuliers envoyés par le système (par exemple lors de panne) [BAH+08, CBTH08, CBC+08].

Bibliographie

[Acc07] Acceleo : MDA Generator. http://www.acceleo.org, 2007. (Cité pages 98 et 145.)

[AGG07] The Attributed Graph Grammar system (AGG). http://tfs.cs.tu-berlin.de/agg/, 2007. (Cité page 66.)

[AK03] Colin ATKINSON et Thomas KUHNE : Model-Driven Development : A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. (Cité page 28.)

[AKK+05] Aditya AGRAWAL, Gabor KARSAI, Zsolt KALMAR, Sandeep NEEMA, Feng SHI et Attila VIZHANYO : The Design of a Language for Model Transformations. Rapport technique, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37235, USA, 2005. (Cité page 72.)

[AM307] The ATLAS MegaModel Management (AM3) Project Home Page. http://www.eclipse.org/gmt/am3, 2007. INRIA ATLAS. (Cité page 33.)

[AMW07] The ATLAS Model Weaver (AMW) Project Home Page. http://www.eclipse.org/gmt/amw, 2007. INRIA ATLAS. (Cité page 161.)

[Ark02] Assaf ARKIN : *Business Process Modeling Language*. Business Process Management Initiative, novembre 2002. (Cité page 48.)

[ASU86] Alfred V. AHO, Ravi SETHI et Jeffrey D. ULLMAN : *Compilers : Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. (Cité page 34.)

[ATL05] INRIA ATLAS : KM3 : Kernel MetaMetaModel. Rapport technique, LINA & INRIA, Nantes, août 2005. (Cité page 36.)

[ATL07a] The ATLAS Transformation Language (ATL) Project Home Page. http://www.eclipse.org/m2m/atl, 2007. INRIA ATLAS. (Cité pages 98 et 145.)

[ATL07b] AMW Use Case - Traceability. http://www.eclipse.org/gmt/amw/usecases/traceability/, 2007. INRIA ATLAS. (Cité page 113.)

[BAH+08] Laurent BROTO, Estella ANNONI, Daniel HAGIMONT, Benoît COMBEMALE et Jean-Paul BAHSOUN : A Model Driven Autonomic Management System. *In Proceedings of the 5th International Conference on*

Information Technology : New Generations (ITNG), Las Vegas, USA, avril 2008. IEEE Computer Society. (Cité page 163.)

[BB04] Jean BÉZIVIN et Erwan BRETON : Applying the basic principles of model engineering to the field of process engineering. *Novatica – Special Issue on Software Process Technologies*, 5(171):27–33, 2004. (Cité page 129.)

[BBF+08] Bernard BERTHOMIEU, Jean-Paul BODEVEIX, Mamoun FILALI, Patrick FARAIL, Pierre GAUFILLET, Hubert GARAVEL et Frederic LANG : FIACRE : an Intermediate Language for Model Verification in the TOPCASED Environment. *In 4th European Congress* EMBEDDED REAL TIME SOFTWARE *(ERTS)*, janvier 2008. (Cité pages 99 et 113.)

[BCC+08] Darlam BENDER, Benoît COMBEMALE, Xavier CRÉGUT, Jean-Marie FARINES et François VERNADAT : Ladder Metamodeling & PLC Program Validation through Time Petri Nets. *In Proceedings of the 4th European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*, Lecture Notes in Computer Science, Berlin, Germany, juin 2008. Springer. (Cité pages 98 et 113.)

[BCCG07] Réda BENDRAOU, Benoit COMBEMALE, Xavier CRÉGUT et Marie-Pierre GERVAIS : Definition of an eXecutable SPEM2.0. *In Proceedings of the 14th Asian-Pacific Software Engineering Conference (APSEC)*, pages 390–397, Nagoya, Japan, décembre 2007. IEEE Computer Society. (Cité page 80.)

[BD91] Bernard BERTHOMIEU et Michel DIAZ : Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991. (Cité page 103.)

[BDJM05] Damien BERGAMINI, Nicolas DESCOUBES, Christophe JOUBERT et Radu MATEESCU : BISIMULATOR : A Modular Tool for On-the-Fly Equivalence Checking. *In* N. HALBWACHS et L. ZUCK, éditeurs : *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, volume 3440 de *Lecture Notes in Computer Science*, pages 581–585. Springer, avril 2005. (Cité pages 105 et 112.)

[BDL04] Gerd BEHRMANN, Alexandre DAVID et Kim G. LARSEN : A tutorial on UPPAAL. *In* M. BERNARDO et F. CORRADINI, éditeurs : *Formal Methods for the Design of Real-Time Systems : 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT)*, volume 3185 de *Lecture Notes in Computer Science*, pages 200–236. Springer, septembre 2004. (Cité page 118.)

[Bec02] Kent BECK : *eXtreme Programming*. Génie logiciel. Campus Press, Paris, France, août 2002. (Cité page 18.)

[BEK+06] Enrico BIERMANN, Karsten EHRIG, Christian KÖHLER, Günter KUHNS, Gabriele TAENTZER et Eduard WEISS : Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. *In* O. NIERSTRASZ, J. WHITTLE, D. HAREL et G. REGGIO, éditeurs : *Proceedings of the 9th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 de *Lecture Notes in Computer Science*, pages 425–439. Springer, octobre 2006. (Cité page 68.)

[BEM94] Noureddine BELKHATIR, Jacky ESTUBLIER et Walcédio L. MELO : The ADELE-TEMPO experience : an environment to support process modeling and enaction. *In Software Process Modelling and Technology*, pages 187–222. Research Studies Press/John Wiley & Sons, 1994. (Cité page 48.)

[Ben07] Réda BENDRAOU : *UML4SPM : Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle*. Thèse de doctorat, Université Pierre & Marie Curie – Paris VI, septembre 2007. (Cité page 56.)

[Ben08] Darlam BENDER : Application of the MDE for modeling and formal verification of PLC programs written in Ladder Diagram Language. Mémoire de D.E.A., Universidade Federal de Santa Catarina, Santa Catarina, Brasil, avril 2008. (Cité page 146.)

[Béz03] Jean BÉZIVIN : *La transformation de modèles*. INRIA-ATLAS & Université de Nantes, 2003. Ecole d'Eté d'Informatique CEA EDF INRIA 2003, cours #6. (Cité pages 13, 31, et 42.)

[Bez07] Christophe BEZ : Contribution à la simulation de machines à états UML2 : Gestion de l'instanciation. Mémoire de D.E.A., Master SCR, INPT ENSEEIHT, Toulouse, France, juin 2007. (Cité pages 146 et 162.)

[BFGL94] Sergio BANDINELLI, Alfonso FUGGETTA, Carlo GHEZZI et Luigi LAVAZZA : SPADE : an environment for software process analysis, design, and enactment. *Software process modelling and technology*, pages 223–247, 1994. (Cité page 48.)

[BFL+95] Sergio BANDINELLI, Alfonso FUGGETTA, Luigi LAVAZZA, Maurizio LOI et Gian Pietro PICCO : Modeling and Improving an Industrial Software Process. *IEEE Transactions on Software Engineering*, 21(5):440–454, mai 1995. (Cité page 48.)

[BFS02] Julian C. BRADFIELD, Juliana Kuster FILIPE et Perdita STEVENS : Enriching OCL Using Observational Mu-Calculus. *In Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2306 de *Lecture Notes in Computer Science*, pages 203–217, Grenoble, France, avril 2002. Springer. (Cité page 88.)

[BG01] Jean BÉZIVIN et Olivier GERBÉ : Towards a Precise Definition of the OMG/MDA Framework. *In Proceedings of the 16th IEEE international*

conference on Automated Software Engineering (ASE), page 273, San Diego, USA, 2001. IEEE Computer Society Press. (Cité page 28.)

[BJRV05] Jean BÉZIVIN, Frédéric JOUAULT, Peter ROSENTHAL et Patrick VALDURIEZ : Modeling in the Large and Modeling in the Small. *In* U. ASSMANN, M. AKSIT et A. RENSINK, éditeurs : *Model Driven Architecture, European MDA Workshops : Foundations and Applications, MDAFA 2003 and MDAFA 2004, , June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, volume 3599 de *Lecture Notes in Computer Science*, pages 33–46, Twente, The Netherlands, 2005. Springer. (Cité page 30.)

[BJV04] Jean BÉZIVIN, Frédéric JOUAULT et Patrick VALDURIEZ : On the Need for Megamodels. *In Proceedings of the OOPSLA/GPCE : Best Practices for Model-Driven Software Development workshop*, 2004. (Cité page 33.)

[BK05] Jean BEZIVIN et Ivan KURTEV : Model-based Technology Integration with the Technical Space Concept. *In Metainformatics Symposium*, Esbjerg, Denmark, 2005. Springer-Verlag. (Cité pages 30 et 128.)

[BKK03] Grady BOOCH, Per KROLL et Philippe KRUTCHTEN : *The Rational Unified Process Made Easy : A Practitioner's Guide to the RUP*. Addison-Wesley Professional, avril 2003. (Cité page 18.)

[Bla05] Xavier BLANC : *MDA en action*. EYROLLES, mars 2005. (Cité page 32.)

[BM83] Bernard BERTHOMIEU et Miguel MENASCHE : An Enumerative Approach for Analyzing Time Petri Nets. *IFIP Congress Series*, 9:41–46, 1983. (Cité page 103.)

[Boe86] Barry W. BOEHM : A Spiral Model of Software Development and Enhancement. *In Software Engineering Notes*, volume 11/4. ACM SIGSOFT, août 1986. (Cité page 47.)

[Bou98] Amar BOUALI : XEVE, an ESTEREL Verification Environment. *In* A. HU et M. Y. VARDI, éditeurs : *Proceedings of the 10th International Conference on Computer Aided Verification (CAV)*, volume 1427 de *Lecture Notes in Computer Science*, pages 500–504, Vancouver, BC, Canada, juin 1998. Springer. (Cité page 112.)

[BPV07] Bernard BERTHOMIEU, Florent PERES et François VERNADAT : Model Checking Bounded Prioritized Time Petri Nets. *In* K. NAMJOSHI, T. YONEDA, T. HIGASHINO et Y. OKAMURA, éditeurs : *Proceedings of the 5th International Symposium Automated Technology for Verification and Analysis (ATVA)*, volume 4762 de *Lecture Notes in Computer Science*, pages 523–532, Tokyo, Japan, octobre 2007. Springer. (Cité pages 100 et 101.)

[Bre02] Erwan BRETON : *Contribution à la représentation de processus par des techniques de méta-modélisation*. Thèse de doctorat, Université de Nantes, juin 2002. (Cité pages 48 et 61.)

[BRV+03] Bernard BERTHOMIEU, Pierre-Olivier RIBET, François VERNADAT, Jean BERNARTT, Jean-Marie FARINES, Jean-Paul BODEVEIX, Mamoun FILALI, Gérard PADIOU, Pierre MICHEL, Patrick FARAIL, Pierre GAUFILLET, Pierre DISSAUX et Jean-Luc LAMBERT : Towards the verification of real-time systems in avionics : The Cotre approach. *In Eigth International workshop for industrial critical systems*, pages 201–216. Thomas Arts, Wan Fokkink, juin 2003. (Cité page 118.)

[BRV04] Bernard BERTHOMIEU, Pierre-Olivier RIBET et François VERNADAT : The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14):2741–2756, juillet 2004. (Cité page 101.)

[BSE03] Franck BUDINSKY, David STEINBERG et Raymond ELLERSICK : *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley Professional, 2003. (Cité pages 18, 36, 39, 43, 90, et 154.)

[BSGB07] Reda BENDRAOU, Andrey SADOVYKH, Marie-Pierre GERVAIS et Xavier BLANC : Software Process Modeling and Execution : The UML4SPM to WS-BPEL Approach. *In Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 314–321, Washington, DC, USA, 2007. IEEE Computer Society. (Cité page 52.)

[BV06] Bernard BERTHOMIEU et François VERNADAT : *Réseaux de Petri temporels : méthodes d'analyse et vérification avec TINA*. Traité IC2, 2006. (Cité pages 101, 103, et 104.)

[Béz04a] Jean BÉZIVIN : In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2):21–24, 2004. (Cité page 28.)

[Béz04b] Jean BÉZIVIN : Sur les principes de base de l'ingénierie des modèles. *RSTI-L'Objet*, 10(4):145–157, 2004. (Cité pages 28, 32, et 41.)

[Béz05] Jean BÉZIVIN : On the unification power of models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005. (Cité page 28.)

[CAD07] Construction and Analysis of Distributed Processes (CADP). http://www.inrialpes.fr/vasy/cadp/, 2007. INRIA VASY. (Cité page 148.)

[CBC+08] Benoît COMBEMALE, Laurent BROTO, Xavier CRÉGUT, Michel DAYDÉ et Daniel HAGIMONT : Autonomic Management Policy Specification : from UML to DSML. *In ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 5301, pages 584–599, Toulouse, France, octobre 2008. LNCS. (Cité page 163.)

[CBTH08] Benoît COMBEMALE, Laurent BROTO, Alain TCHANA et Daniel HAGIMONT : Metamodeling Autonomic System Management Policies. *In Model-Driven Development of Autonomic Systems (MDDAS), Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, Turku, Finland, juillet 2008. IEEE Computer Society. (Cité page 163.)

[CC97] Xavier CRÉGUT et Bernard COULETTE : PBOOL : an Object-Oriented Language for Definition and Reuse of Enactable Processes. *Software Concepts & Tools*, 18(2):47–62, juin 1997. (Cité pages 23 et 48.)

[CCBV07] Benoit COMBEMALE, Xavier CRÉGUT, Bernard BERTHOMIEU et Francois VERNADAT : SimplePDL2Tina : Mise en oeuvre d'une Validation de Modèles de Processus. *In 3ieme journées sur l'Ingénierie Dirigée par les Modeles (IDM)*, Toulouse, France, mars 2007. (Cité page 98.)

[CCCC06] Benoit COMBEMALE, Xavier CRÉGUT, Alain CAPLAIN et Bernard COULETTE : Modélisation rigoureuse en SPEM de procédé de développement. *In* Roger ROUSSEAU, Christelle URTADO et Sylvain VAUTTIER, éditeurs : *12ième conférence sur les Langages et Modèles à Objets (LMO)*, pages 135–150, Nîmes - France, mars 2006. Hermes Sciences/-Lavoisier. (Cité pages 49 et 103.)

[CCG+08a] Benoît COMBEMALE, Xavier CRÉGUT, Pierre-Loïc GAROCHE, Xavier THIRIOUX et Francois VERNADAT : A Property-Driven Approach to Formal Verification of Process Models. *In* Jorge CARDOSO, José CORDEIRO, Joaquim FILIPE et Vitor PEDROSA, éditeurs : *Enterprise Information System IX*. Springer-Verlag, 2008. (Cité page 80.)

[CCG+08b] Benoît COMBEMALE, Xavier CRÉGUT, Jean-Pierre GIACOMETTI, Pierre MICHEL et Marc PANTEL : Introducing Simulation and Model Animation in the MDE TOPCASED Toolkit. *In Proceedings of the 4th European Congress* EMBEDDED REAL TIME SOFTWARE *(ERTS)*, Toulouse, France, janvier 2008. (Cité pages 80 et 146.)

[CCMP07] Benoît COMBEMALE, Xavier CRÉGUT, Pierre MICHEL et Marc PANTEL : SéMo'07 : premier atelier sur la Sémantique des Modèles. *L'Objet*, 13(4/2007):137–144, 2007. (Cité page 56.)

[CCN06] Stephen CAMPBELL, Jean-Philippe CHANCELIER et Ramine NIKOUKHAH : *Modeling and Simulation in Scilab/Scicos*. Springer, 2006. (Cité page 118.)

[CCO+04] Sagar CHAKI, Edmund M. CLARKE, Joël OUAKNINE, Natasha SHARYGINA et Nishant SINHA : State/Event-based Software Model Checking. *In* E. BOITEN, J. DERRICK et G. SMITH, éditeurs : *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM)*, volume 2999 de *Lecture Notes in Computer Science*, pages 128–147. Springer, avril 2004. (Cité pages 101, 103, et 105.)

[CCOP06] Benoit COMBEMALE, Xavier CRÉGUT, Ileana OBER et Christian PERCEBOIS : Evaluation du standard SPEM de représentation des processus. *Génie Logiciel - Magazine de l'ingéniérie du logiciel et des systèmes*, 77, juin 2006. Presented at NEPTUNE'06 Conference, Paris - France, May 2006. (Cité page 49.)

[CEK01] Tony CLARK, Andy EVANS et Stuart KENT : The Metamodelling Language Calculus : Foundation Semantics for UML. *In Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2029 de *Lecture Notes In Computer Science*, pages 17–31, London, UK, 2001. Springer. (Cité page 72.)

[CESW04] Tony CLARK, Andy EVANS, Paul SAMMUT et James WILLANS : Applied Metamodelling – A Foundation for Language Driven Development. version 0.1, 2004. (Cité pages 36, 59, 68, et 73.)

[CGCT07] Benoît COMBEMALE, Pierre-Loïc GAROCHE, Xavier CRÉGUT et Xavier THIRIOUX : Towards a Formal Verification of Process Model's Properties – SimplePDL and TOCL case study. *In Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS)*, pages 80–89, Funchal, Madeira - Portugal, juin 2007. INSTICC press. (Cité page 80.)

[CH03] Krzysztof CZARNECKI et Simon HELSEN : Classification of Model Transformation Approaches. *In OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003. (Cité page 43.)

[CHL+94a] Reidar CONRADI, Marianne HAGASETH, Jens-Otto LARSEN, Minh Ngoc NGUYÊN, Bjorn P. MUNCH, Per H. WESTBY, Zhu WEICHENG, Letizia JACCHERI et Chunnian LIU : Object-Oriented and Cooperative Process Modelling in EPOS. *In Software Process Modelling and Technology*, pages 33–70. Research Studies Press/John Wiley & Sons, 1994. (Cité page 48.)

[CHL+94b] Reidar CONRADI, Marianne HAGASETH, Jens-Otto LARSEN, Minh Ngoc NGUYÊN, Bjorn P. MUNCH, Per H. WESTBY, Weicheng ZHU, M. Letizia JACCHERI et Chunnian LIU : EPOS : object-oriented cooperative process modelling. *Software process modelling and technology*, pages 33–70, 1994. (Cité page 48.)

[CK02] Maria Victoria CENGARLE et Alexander KNAPP : Towards OCL/RT. *In International Symposium of Formal Methods Europe (FME) - Getting IT Right*, pages 390–409, London, UK, 2002. Springer-Verlag. (Cité page 88.)

[CK07] Thomas CLEENEWERCK et Ivan KURTEV : Separation of concerns in translational semantics for DSLs in model engineering. *In Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, pages 985–992, New York, NY, USA, 2007. ACM Press. (Cité page 72.)

[CKO92] Bill CURTIS, Marc I. KELLNER et Jim OVER : Process modeling. *Commun. ACM*, 35(9):75–90, 1992. (Cité page 47.)

[CM06] Eric NGuyen et Nicolas Pégourié CHRISOPHE MANCINO, Michel Martin : Plugin Eclipse de gestion de projet en ingénierie dirigée par les modèles. Mémoire de D.E.A., IUT de Blagnac, Toulouse, France, juin 2006. (Cité page 124.)

[Com05] Benoît COMBEMALE : Spécification et vérification de modèles de procédés de développement. Mémoire de D.E.A., Université Toulouse II & INPT ENSEEIHT, juin 2005. (Cité page 49.)

[COQ06] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, 2006. (Cité page 143.)

[COQ07] The Coq proof assistant. http://coq.inria.fr/, 2007. INRIA. (Cité page 111.)

[Cou90] Patrick COUSOT : Methods and Logics for Proving Programs. *In Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 841–994. MIT Press, Cambridge, MA, USA, 1990. (Cité page 37.)

[Cra02] Jean-Bernard CRAMPES : *Méthode orientée-objet intégrale MACAO, Démarche participative pour l'analyse, la conception et la réalisation de logiciels*. Génie logiciel. Technosup, Ellipse édition, décembre 2002. (Cité page 84.)

[CRC+06a] Benoit COMBEMALE, Sylvain ROUGEMAILLE, Xavier CRÉGUT, Frédéric MIGEON, Marc PANTEL et Christine MAUREL : Expériences pour décrire la sémantique en ingénierie des modèles. *In* Hermes SCIENCES/LAVOISIER, éditeur : *2ième journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, pages 17–34, Lilles, France, juin 2006. (Cité page 57.)

[CRC+06b] Benoit COMBEMALE, Sylvain ROUGEMAILLE, Xavier CRÉGUT, Frédéric MIGEON, Marc PANTEL, Christine MAUREL et Bernard COULETTE : Towards a rigorous metamodeling. *In* Luis F. PIRES et Slimane HAMMOUDI, éditeurs : *Proceedings of the 2nd International Workshop on Model-Driven Enterprise Information Systems (MDEIS)*, pages 5–14, Paphos, Cyprus, mai 2006. INSTICC press. (Cité page 57.)

[Cré97] Xavier CRÉGUT : *Un environnement d'assistance rigoureuse pour la description et l'exécution de processus de conception - Application à l'approche objet*. Thèse de doctorat, Institut National Polytechnique de Toulouse, juillet 1997. (Cité page 23.)

[CSW08a] Tony CLARK, Paul SAMMUT et James WILLANS : Applied Metamodelling – A Foundation for Language Driven Development. Second Edition, 2008. (Cité page 73.)

[CSW08b] Tony CLARK, Paul SAMMUT et James WILLANS : SUPERLANGUAGES – Developing Languages and Applications with XMF. First Edition, 2008. (Cité pages 18, 20, et 61.)

[DEA98] S. DAMI, J. ESTUBLIER et M. AMIOUR : Apel : A Graphical Yet Executable Formalism forProcess Modeling. *Automated Software Engineering : An International Journal*, 5(1):61–96, janvier 1998. (Cité page 48.)

[Dki07] Youssef DKIOUAK : Validation et Simulation de Modèles au sein des projets Up2UML et TOPCASED. Mémoire de D.E.A., Diplôme des Etudes Supérieur Approfondies, ENSIAS, Rabat, Maroc, juin 2007. (Cité page 146.)

[DKR00] Dino DISTEFANO, Joost-Pieter KATOEN et Arend RENSINK : Towards model checking OCL. *In Proceedings of the ECOOP Workshop on Dening a Precise Semantics for UML*, 2000. (Cité page 88.)

[Ecl07] Eclipse, An Open Development Platform. http://www.eclipse.org, 2007. Eclipse. (Cité pages 90 et 145.)

[EEdL+05] Hartmut EHRIG, Karsten EHRIG, Juan de LARA, Gabriele TAENTZER, Dániel VARRÓ et Szilvia VARRÓ-GYAPAY : Termination Criteria for Model Transformation. *In* M. CERIOLI, éditeur : *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 de *Lecture Notes in Computer Science*, pages 49–63, Edinburgh, UK, avril 2005. Springer. (Cité page 68.)

[EEHT05a] Karsten EHRIG, Claudia ERMEL, Stefan HÄNSGEN et Gabriele TAENTZER : Towards Graph Transformation Based Generation of Visual Editors Using Eclipse. *Electr. Notes Theor. Comput. Sci*, 127(4), 2005. (Cité page 38.)

[EEHT05b] Karsten EHRIG, Claudia ERMEL, Stefan HÄNSGEN et Gabriele TAENTZER : Generation of visual editors as eclipse plug-ins. *In Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering (ASE)*, pages 134–143, New York, NY, USA, 2005. ACM Press. (Cité pages 19 et 66.)

[EEKR99] H. EHRIG, G. ENGELS, H.-J. KREOWSKI et G. ROZENBERG, éditeurs. *Handbook of graph grammars and computing by graph transformation. Volume II : applications, languages, and tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. (Cité page 66.)

[EMF07] The Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf, 2007. Eclipse. (Cité pages 18, 38, 90, et 145.)

[EPF07] The Eclipse Process Framework (EPF). http://www.eclipse.org/epf, 2007. Eclipse. (Cité page 147.)

[Est05] Jacky ESTUBLIER : Software are Processes Too. *In Invited Paper at Software Process Workshop*, Bejing, China, mai 2005. (Cité pages 22 et 47.)

[Fav04] Jean-Marie FAVRE : Towards a Basic Theory to Model Model Driven Engineering. *In Workshop on Software Model Engineering (WISME), joint event with UML ?2004*, Lisboa, 2004. (Cité pages 33 et 129.)

[FBB+03] Jean-Marie FARINES, Bernard BERTHOMIEU, Jean-Paul BODEVEIX, Pierre DISSAUX, Patrick FARAIL, Mamoun FILALI, Pierre GAUFILLET, Hicham HAFIDI, Jean-Luc LAMBERT, Pierre MICHEL et François VERNADAT : *The Cotre Project : Rigorous Software Development for Real Time Systems in Avionics*. COLNARIC, ADAMSKI & WEGRZYN, novembre 2003. (Cité page 118.)

[FBJ+05] Marcos D. FABRO, Jean BÉZIVIN, Frédéric JOUAULT, Erwan BRETON et Guillaume GUELTAS : AMW : a generic model weaver. *In Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM)*, 2005. (Cité pages 113 et 161.)

[FEB06] Jean-Marie FAVRE, Jacky ESTUBLIER et Mireille BLAY : *L'Ingénierie Dirigée par les Modèles : au-delà du MDA*. Informatique et Systèmes d'Information. Hermes Science, lavoisier édition, février 2006. (Cité pages 13, 29, 31, et 68.)

[FGC+06] Patrick FARAIL, Pierre GAUFILLET, Agusti CANALS, Christophe Le CAMUS, David SCIAMMA, Pierre MICHEL, Xavier CRÉGUT et Marc PANTEL : The TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design. *In 3rd European Congress* EMBEDDED REAL TIME SOFTWARE *(ERTS)*, Toulouse, France, janvier 2006. (Cité pages 19 et 38.)

[Fla03] Stephan FLAKE : Temporal OCL extensions for specification of real-time constraints. *In SVERTS – Proceedings of the UML'03 workshop*, San Francisco, CA, USA, octobre 2003. (Cité page 88.)

[FM03] S. FLAKE et W. MUELLER : Formal Semantics of Static and Temporal State-Oriented OCL Constraints. *Journal on Software and System Modeling (SoSyM)*, 2(3), octobre 2003. (Cité page 88.)

[GCC+08] Ange GARCIA, Benoît COMBEMALE, Xavier CRÉGUT, Jean-Noël GUYOT et Boris LIBERT : topPROCESS : a Process Model Driven Approach Applied in TOPCASED for Embedded Real-Time Software. *In Proceedings of the 4th European Congress* EMBEDDED REAL TIME SOFTWARE *(ERTS)*, Toulouse, France, janvier 2008. (Cité page 146.)

[GEF07] The Graphical Editing Framework (GEF). http://www.eclipse.org/gef, 2007. Eclipse. (Cité page 145.)

[GEM07] The Generic Eclipse Modeling System (GEMS). http://sourceforge.net/projects/gems, 2007. (Cité page 38.)

[GHJ95] Erich GAMMA, Richard HELM et Ralph JOHNSON : *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. (Cité page 28.)

[GHP02] Eran GERY, David HAREL et Eldad PALACHI : Rhapsody : A Complete Life-Cycle Model-Based Development System. *In Proceedings of the Third International Conference on Integrated Formal Methods (IFM)*, volume 2335 de *Lecture Notes In Computer Science*, pages 1–10, Turku, Finland, mai 2002. Springer. (Cité page 56.)

[GLR+02] Anna GERBER, Michael LAWLEY, Kerry RAYMOND, Jim STEEL et Andrew WOOD : Transformation : The Missing Link of MDA. *In* A. CORRADINI, H. EHRIG, H. KREOWSKI et G. ROZENBERG, éditeurs : *Proceedings of the First International Conference on Graph Transformation (ICGT)*, volume 2505 de *Lecture Notes in Computer Science*, pages 90–105, Barcelona, Spain, octobre 2002. Springer. (Cité pages 41 et 42.)

[GMF07] The Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf, 2007. Eclipse. (Cité pages 19, 38, et 145.)

[GR01] Martin GROSSE-RHODE : Formal Concepts for an Integrated Internal Model of the UML. *Electronic Notes in Theoretical Computer Science*, 44(4), 2001. (Cité page 66.)

[Gro07] CNRS-IRIT & ONERA-CERT Labs & Sopra GROUP : Simulations de Machine à états UML2.0, v1. Rapport technique, Report for the TOPCASED projet, WP2 (Models Simulation), 2007. (Cité pages 122 et 153.)

[GS90] Carl A. GUNTER et Dana S. SCOTT : Semantic Domains. *In Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 633–674. MIT Press, Cambridge, MA, USA, 1990. (Cité page 68.)

[GSCK04] Jack GREENFIELD, Keith SHORT, Steve COOK et Stuart KENT : *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, septembre 2004. (Cité page 128.)

[Gur01] Yuri GUREVICH : The Abstract State Machine Paradigm : What Is in and What Is out. *In Ershov Memorial Conference*, 2001. (Cité page 72.)

[Guy07] Jean-Noël GUYOT : Modélisation du processus qualité du projet TOPCASED. Mémoire de D.E.A., Licence SIL, Université Paul Sabatier, Toulouse, France, juin 2007. (Cité page 146.)

[GV04] Alain GRIFFAULT et Aymeric VINCENT : The Mec 5 Model-Checker. *In* R. ALUR et D. PELED, éditeurs : *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 de *Lecture Notes in Computer Science*, pages 488–491, Boston, MA, USA, juillet 2004. Springer. (Cité page 104.)

[GZK02] Martin GOGOLLA, Paul ZIEMANN et Sabine KUSKE : Towards an Integrated Graph Based Semantics for UML. *In* P. BOTTONI et M. MINAS,

éditeurs : *Proceedings of the Graph Transformation and Visual Modeling Techniques (GT-VMT)*, volume 72(3) de *ENTCS*, Barcelona, Spain, octobre 2002. Elsevier. (Cité page 68.)

[Hau05] Jan Hendrik HAUSMANN : *Dynamic Meta Modeling – A Semantics Description Technique for Visual Modeling Languages*. Thèse de doctorat, University of Paderborn, 2005. (Cité pages 40, 68, et 73.)

[HbR00] David HAREL et bernhard RUMPE : Modeling Languages : Syntax, Semantics and All That Stuff, Part I : The Basic Stuff. Rapport technique, Mathematics & Computer Science, Weizmann Institute Of Science, Weizmann Rehovot, Israel, août 2000. (Cité page 40.)

[Hes06] Wolfgang HESSE : More matters on (meta-)modelling : remarks on Thomas Kühne's "matters". *Software and Systems Modeling (SoSyM)*, 5(4):387–394, 2006. (Cité page 129.)

[Hir01] Daniel HIRSCHKOFF : Bisimulation verification using the up to techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):271–285, 2001. (Cité page 111.)

[HN96] David HAREL et Amnon NAAMAD : The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996. (Cité page 153.)

[HR04] David HAREL et Bernhard RUMPE : Meaningful Modeling : What's the Semantics of "Semantics" ? *Computer*, 37(10):64–72, 2004. (Cité pages 34, 40, et 58.)

[Ins90] Institute of Electrical and Electronics Engineers, New York, NY. *IEEE Standard Computer Dictionary : A Compilation of IEEE Standard Computer Glossaries*, 1990. (Cité page 47.)

[iUM07] iUML executable UML modeling tool. http://www.kc.com/products.php, 2007. Kennedy Carter. (Cité page 56.)

[JB06] Frédéric JOUAULT et Jean BÉZIVIN : KM3 : a DSL for Metamodel Specification. *In Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 de *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006. (Cité pages 14, 18, 36, 129, et 130.)

[JBK06] Frédéric JOUAULT, Jean BÉZIVIN et Ivan KURTEV : TCS : a DSL for the specification of textual concrete syntaxes in model engineering. *In* S. JARZABEK, D. SCHMIDT et T. VELDHUIZEN, éditeurs : *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 249–254, Portland, Oregon, USA, octobre 2006. ACM. (Cité pages 19, 39, et 98.)

[JET07] JET code generator. http://www.eclipse.org/modeling/m2t/?project=jet, 2007. (Cité page 98.)

[JK05] Frédéric JOUAULT et Ivan KURTEV : Transforming Models with ATL. *In Satellite Events at the MoDELS 2005 Conference, Proceedings of the Model Transformations in Practice Workshop*, volume 3844 de *Lecture Notes in Computer Science*, pages 128–138, Montego Bay, Jamaica, 2005. Springer. (Cité pages 44, 61, et 98.)

[Jou06] Frédéric JOUAULT : *Contribution à l'étude des langages de transformation de modèles*. Thèse de doctorat, Université de Nantes, septembre 2006. (Cité page 43.)

[JZM07a] Ke JIANG, Lei ZHANG et Shigeru MIYAKE : An Executable UML with OCL-based Action Semantics Language. *In Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC)*, pages 302–309, Nagoya, Japan, décembre 2007. IEEE Computer Society. (Cité page 56.)

[JZM07b] Ke JIANG, Lei ZHANG et Shigeru MIYAKE : OCL4X : An Action Semantics Language for UML Model Execution. *In Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 633–636, Beijing, China, juillet 2007. IEEE Computer Society. (Cité page 56.)

[Jéz08] Jean-Marc JÉZÉQUEL : Model Driven Design and Aspect Weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2), mars 2008. (Cité page 62.)

[Kah87] Gilles KAHN : Natural Semantics. Report no. 601, INRIA, février 1987. (Cité page 61.)

[KBA02] Ivan KURTEV, Jean BÉZIVIN et Mehmet AKSIT : Technological Spaces : An Initial Appraisal. *In CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002. (Cité page 30.)

[KBJV06] Ivan KURTEV, Jean BÉZIVIN, Frédéric JOUAULT et Patrick VALDURIEZ : Model-based DSL Frameworks. *In Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 602–616, Portland, OR, USA, octobre 2006. ACM. (Cité pages 81 et 93.)

[KDH07] Andrei KIRSHIN, Dolev DOTAN et Alan HARTMAN : A UML Simulator Based on a Generic Model Execution Engine. *Models in Software Engineering*, pages 324–326, 2007. (Cité page 125.)

[Ker07] The KerMeta Project Home Page. http://www.kermeta.org, 2007. INRIA TRISKELL. (Cité page 145.)

[KFP88] Gail E. KAISER, Peter H. FEILER et Steven S. POPOVICH : Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, 5(3):40–49, 1988. (Cité page 48.)

[KGKK02] Sabine KUSKE, Martin GOGOLLA, Ralf KOLLMANN et Hans-Jörg KREOWSKI : An Integrated Semantics for UML Class, Object and State

Diagrams Based on Graph Transformation. *In Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM)*, volume 2335 de *Lecture Notes In Computer Science*, pages 11–28, London, UK, 2002. Springer. (Cité page 68.)

[KK03] Per KROLL et Philippe KRUCHTEN : *Guide pratique du RUP*. PEARSON Education, 2003. (Cité page 84.)

[KKR06a] Harmen KASTENBERG, Anneke KLEPPE et Arend RENSINK : Defining Object-Oriented Execution Semantics Using Graph Transformations. *In* R. GORRIERI et H. WEHRHEIM, éditeurs : *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 de *Lecture Notes in Computer Science*, pages 186–201, Bologna, Italy, juin 2006. Springer-Verlag. (Cité page 68.)

[KKR06b] Harmen KASTENBERG, Anneke KLEPPE et Arend RENSINK : Engineering Object-Oriented Semantics Using Graph Transformations. CTIT Technical Report 06-12, University of Twente, March 2006. (Cité page 68.)

[Kle06] Anneke KLEPPE : MCC : A Model Transformation Environment. *In Proceedings of the First European Conference Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 173–187, Bilbao, Spain, juillet 2006. (Cité pages 13 et 42.)

[KLM+97] Gregor KICZALES, John LAMPING, Anurag MENHDHEKAR, Chris MAEDA, Cristina LOPES, Jean-Marc LOINGTIER et John IRWIN : Aspect-Oriented Programming. *In* M. AKSIT et S. MATSUOKA, éditeurs : *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 de *Lecture Notes in Computer Science*, pages 220–242. Springer, Jyväskylä, Finland, juin 1997. (Cité page 28.)

[Küh06a] Thomas KÜHNE : Clarifying matters of (meta-) modeling : an author's reply. *Software and Systems Modeling (SoSyM)*, 5(4):395–401, 2006. (Cité page 129.)

[Küh06b] Thomas KÜHNE : Matters of (Meta-)Modeling. *Software and Systems Modeling (SoSyM)*, 5(4):369–385, décembre 2006. (Cité page 129.)

[Kus00] Sabine KUSKE : *Transformation Units—A structuring Principle for Graph Transformation Systems*. Thèse de doctorat, University of Bremen, 2000. (Cité page 68.)

[Kus01] Sabine KUSKE : A Formal Semantics of UML State Machines Based on Structured Graph Transformation. *In Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 de *Lecture Notes In Computer Science*, pages 241–256, London, UK, 2001. Springer. (Cité page 68.)

[KWB03] Anneke KLEPPE, Jos WARMER et Wim BAST : *MDA Explained. The Model Driven Architecture : Practice and Promise*. Addison-Wesley, 2003. (Cité page 32.)

[Lab06a] CNRS-IRIT & ONERA-CERT LABS : Synthesis on Methods and Tools for Simulation. Rapport technique, Report D02 for the TOPCASED projet, WP2 (Models Simulation), 2006. (Cité page 116.)

[Lab06b] CNRS-IRIT & ONERA-CERT LABS : Synthesis on Simulation Needs. Rapport technique, Report D01 for the TOPCASED projet, WP2 (Models Simulation), 2006. (Cité page 116.)

[LdS04] Miguel LUZ et Alberto Rodrigues da SILVA : Executing UML Models. *In Proceedings of the 3rd Workshop in Software Model Engineering (WiSME)*, Portugal, Lisbon, octobre 2004. (Cité page 56.)

[Lib07] Boris LIBERT : Utilisation de GMF pour réaliser un éditeur SPEM sous Eclipse. Mémoire de D.E.A., Licence NTIE, Université Le Mirail, Toulouse, France, juin 2007. (Cité page 146.)

[LMB+01] A. LEDECZI, M. MAROTI, A. BAKAY, G. KARSAI, J. GARRETT, C. Thomason IV, G. NORDSTROM, J. SPRINKLE et P. VOLGYESI : The Generic Modeling Environment. *In Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP)*, Budapest, Hungary, mai 2001. (Cité page 36.)

[LSV98] Edward A. LEE et Alberto L. SANGIOVANNI-VINCENTELLI : A framework for comparing models of computation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998. (Cité page 119.)

[Mat07a] Matlab/Simulink. http://www.mathworks.com, 2007. The MathWorks. (Cité page 118.)

[Mat07b] Jonathan MATH : Implémentation de la première version d'un simulateur de modèles générique au sein de l'atelier TOPCASED. Mémoire de D.E.A., Projet de fin d'études, INPT ENSEEIHT, Toulouse, France, juin 2007. (Cité page 146.)

[MBB02] Stephen MELLOR, Marc BALCER et Marc BALCER : *Executable UML : A Foundation for Model-Driven Architecture*. Pearson Education, 2002. (Cité page 56.)

[MDT07] MDT-UML2, an EMF-based implementation of the Unified Modeling Language (UML) 2.x OMG metamodel for the Eclipse platform. http://www.eclipse.org/uml2, 2007. Eclipse. (Cité page 149.)

[Men07] BridgePoint Development Suite. http://www.mentor.com/products/sm/uml_suite/, 2007. Mentor Graphics. (Cité page 56.)

[Mer74] Philip M. MERLIN : *A Study of the Recoverability of Computing Systems*. University of California, Irvine, PhD Thesis, 1974. (Cité page 101.)

[Mer07] Merlin generator. http://sourceforge.net/projects/merlingenerator, 2007. (Cité page 38.)

[MFF+08] Pierre-Alain MULLER, Frédéric FONDEMENT, Franck FLEUREY, Michel HASSENFORDER, Rémi SCHNECKENBURGER, Sébastien GÉRARD et Jean-Marc JÉZÉQUEL : Model-driven analysis and synthesis of textual concrete syntax. *Journal of Software and Systems Modeling (SoSyM)*, 2008. Online first. (Cité pages 19, 39, et 98.)

[MFJ05] Pierre-Alain MULLER, Franck FLEUREY et Jean-Marc JÉZÉQUEL : Weaving Executability into Object-Oriented Meta-Languages. *In* L. BRIAND et C. WILLIAMS, éditeurs : *Proceedings of the 8th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 de *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, octobre 2005. Springer. (Cité pages 18, 20, 36, et 61.)

[Mil95] Robin MILNER : *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, c. a. r. hoare édition, 1995. (Cité pages 59 et 97.)

[Min68] Marvin MINSKY : Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968. (Cité pages 28, 29, et 85.)

[MM03] Joaquin MILLER et Jishnu MUKERJI : *Model Driven Architecture (MDA) 1.0.1 Guide*. Object Management Group, Inc., juin 2003. (Cité pages 31 et 128.)

[Mos90] Peter D. MOSSES : Denotational Semantics. *In Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 575–631. MIT Press, Cambridge, MA, USA, 1990. (Cité page 68.)

[oAW07] The openArchitectureWare (oAW) generator framework. http://www.openarchitectureware.org/, 2007. (Cité pages 98 et 145.)

[OMG04] Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 2.0 RFP*, novembre 2004. (Cité page 50.)

[OMG05a] Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 1.1*, janvier 2005. (Cité pages 36 et 49.)

[OMG05b] Object Management Group, Inc. *UML Testing Profile 1.0 Specification*, juillet 2005. Final Adopted Specification. (Cité page 121.)

[OMG06a] Object Management Group, Inc. *Diagram Interchange 1.0 Specification*, avril 2006. (Cité page 49.)

[OMG06b] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, janvier 2006. Final Adopted Specification. (Cité pages 18, 29, 30, 36, 49, 61, 83, 90, 94, 134, et 137.)

[OMG06c] Object Management Group, Inc. *Object Constraint Language (OCL) 2.0 Specification*, mai 2006. (Cité pages 19 et 38.)

[OMG07a] Object Management Group, Inc. *Software & Systems Process Engineering Metamodel (SPEM) 2.0 (Beta 2)*, octobre 2007. (Cité pages 13, 22, 48, 49, et 50.)

[OMG07b] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.2 Infrastructure*, novembre 2007. Final Adopted Specification. (Cité pages 18, 30, et 49.)

[OMG07c] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.2 Superstructure*, novembre 2007. Final Adopted Specification. (Cité pages 18, 30, et 33.)

[OMG08] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0*, avril 2008. (Cité pages 13, 44, et 45.)

[Ost87] Leon J. OSTERWEIL : Software Processes Are Software Too. *In Proceedings of the 9th International Conference on Software Engineering (ICSE)*, pages 2–13, Monterey, California, USA, mars 1987. (Cité pages 22 et 47.)

[Pad82] Peter PADAWITZ : Graph Grammars and Operational Semantics. *Theorical Computer Science*, 19:117–141, 1982. (Cité page 66.)

[PKP06] Richard F. PAIGE, Dimitrios S. KOLOVOS et Fiona A. C. POLACK : An action semantics for MOF 2.0. *In Proceedings of the 2006 ACM symposium on Applied computing (SAC)*, pages 1304–1305, New York, NY, USA, 2006. ACM. (Cité page 61.)

[Plo81] Gordon D. PLOTKIN : A Structural Approach to Operational Semantics. Rapport technique FN-19, DAIMI, University of Aarhus, Denmark, septembre 1981. (Cité page 61.)

[Pou07] Damien POUS : New up-to techniques for weak bisimulation. *Theoretical Computer Science*, 380(1-2):164–180, 2007. (Cité page 111.)

[Pto07] Ptolemy II, Heterogeneous Modelling and Design. http://ptolemy.berkeley.edu/ptolemyII/, 2007. (Cité page 119.)

[Rei85] Wolfgang REISIG : *Petri nets : an introduction*. Springer-Verlag, New York, NY, USA, 1985. (Cité pages 60 et 101.)

[Ren04] Arend RENSINK : The GROOVE Simulator : A Tool for State Space Generation. *In* J. PFALTZ, M. NAGL et B. BÖHLEN, éditeurs : *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, (AGTIVE'03), Revised Selected and Invited Papers*, volume 3062 de *Lecture Notes in Computer Science*, pages 479–485, Charlottesville, VA, USA, septembre 2004. Springer. (Cité page 68.)

[Ren07] Arend RENSINK : GROOVE : A Graph Transformation Tool Set for the Simulation and Analysis of Graph Grammars. http://groove.cs.utwente.nl/, 2007. (Cité page 68.)

[RG99] Mark RICHTERS et Martin GOGOLLA : A Metamodel for OCL. *In* R. FRANCE et B. RUMPE, éditeurs : *UML'99 - The Unified Modeling Language. Second International Conference.*, volume 1723 de *Lecture Notes in Computer Science*, pages 156–171, USA, octobre 1999. Springer. (Cité page 90.)

[RG00] Mark RICHTERS et Martin GOGOLLA : Validating UML Models and OCL Constraints. *In* A. EVANS, S. KENT et B. SELIC, éditeurs : *Proceedings of the 3rd International Conference UML'00 – The Unified Modeling Language*, volume 1939 de *Lecture Notes In Computer Science*, pages 265–277. Springer, octobre 2000. (Cité page 38.)

[Roy87] Winston W. ROYCE : Managing the Development of Large Software Systems : Concepts and Techniques. *In Proceedings of the 9th International Conference on Software Engineering (ICSE)*, Monterey, California, USA, mars 1987. (Cité page 47.)

[Roz97] G. ROZENBERG, éditeur. *Handbook of graph grammars and computing by graph transformation : volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. (Cité page 66.)

[San96] Davide SANGIORGI : A Theory of Bisimulation for the pi-Calculus. *Acta Informatica*, 33(1):69–97, 1996. (Cité page 111.)

[Sei03] Ed SEIDEWITZ : What models mean. *IEEE Software*, 20(5):26–32, 2003. (Cité pages 28 et 128.)

[Sil07] Sildex. http://www.tni-software.com, 2007. TNI-Software. (Cité page 118.)

[Sin07] The Sintaks Project Home Page. http://www.kermeta.org/sintaks, 2007. INRIA TRISKELL. (Cité pages 39 et 98.)

[SJ05] Jim STEEL et Jean-Marc JÉZÉQUEL : Model Typing for Improving Reuse in Model-Driven Engineering. *In* L. BRIAND et C. WILLIAMS, éditeurs : *Proceedings of the 8th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 de *Lecture Notes in Computer Science*, pages 84–96, Montego Bay, Jamaica, octobre 2005. Springer. (Cité page 129.)

[SJ07] Jim STEEL et Jean-Marc JÉZÉQUEL : On Model Typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):452–468, décembre 2007. (Cité page 128.)

[SK97] Janos SZTIPANOVITS et Gabor KARSAI : Model-Integrated Computing. *Computer*, 30(4):110–111, 1997. (Cité page 128.)

[Sol00] Richard SOLEY : *Model Driven Architecture (MDA), Draft 3.2*. Object Management Group, Inc., novembre 2000. (Cité page 31.)

[Sta07] Statemate. http://modeling.telelogic.com/products/statemate/, 2007. Telelogic. (Cité page 118.)

[Tae03] Gabriele TAENTZER : AGG : A Graph Transformation Environment for Modeling and Validation of Software. *In Proceedings of the Second International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 de *Lecture Notes in Computer Science*, pages 446–453, Charlottesville, VA, USA, octobre 2003. Springer. (Cité pages 61 et 66.)

[TCCG07] Xavier THIRIOUX, Benoit COMBEMALE, Xavier CRÉGUT et Pierre-Loïc GAROCHE : A Framework to formalise the MDE Foundations. *In* Richard PAIGE et Jean BÉZIVIN, éditeurs : *Proceedings of the International Workshop on Towers of Models (TOWERS)*, pages 14–30, Zurich, juin 2007. (Cité page 128.)

[TCS07] TCS. http://www.eclipse.org/gmt/tcs/, 2007. INRIA ATLAS. (Cité pages 39 et 98.)

[Tig07] Tiger. http://tfs.cs.tu-berlin.de/~tigerprj, 2007. (Cité page 38.)

[TIN07] TIme petri Net Analyzer (TINA). http://www.laas.fr/tina/, 2007. LAAS CNRS. (Cité pages 100 et 148.)

[Top07] Toolkit in OPen source for Critical Application & SystEms Development. http://www.topcased.org/, 2007. TOPCASED Consortium. (Cité page 38.)

[USE07] A UML-based Specification Environment (USE). http://www.db.informatik.uni-bremen.de/projects/USE/, 2007. (Cité page 88.)

[VBBJ06] Eric VÉPA, Jean BÉZIVIN, Hugo BRUNELIÈRE et Frédéric JOUAULT : Measuring Model Repositories. *In Proceedings of the Model Size Metrics Workshop at the MoDELS/UML 2006 conference*, Lecture Notes In Computer Science, Genoava, Italy, 2006. Springer. (Cité page 33.)

[WfM05] WFMC : *Process Definition interface – XML Process Definition Language v2.0*, octobre 2005. (Cité page 48.)

[Win93] Glynn WINSKEL : *The formal semantics of programming languages : an introduction*. MIT Press, Cambridge, MA, USA, 1993. 0-262-23169-7. (Cité pages 57 et 73.)

[WK03] Jos WARMER et Anneke KLEPPE : *The Object Constraint Language : Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003. (Cité page 38.)

[Wor07] Workbench. http://www.hyperformix.com/products/workbench, 2007. HyPerformix. (Cité page 118.)

[XMF07] The XMF extensible programming language. http://www.ceteva.com, 2007. Ceteva. (Cité page 18.)

[xUM07] Executable UML (xUML). http://www.kc.com/xuml.php, 2007. Kennedy Carter. (Cité page 56.)

[ZG02] Paul ZIEMANN et Martin GOGOLLA : An Extension of OCL with Temporal Logic. *In* J. JÜRJENS, M.-V. CENGARLE, E. FERNANDEZ, B. RUMPE et R. SANDNER, éditeurs : *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, volume TUM-I0208, pages 53–62. Université Technique de Munich, Institut d'Informatique, septembre 2002. (Cité page 87.)

[ZG03] Paul ZIEMANN et Martin GOGOLLA : An OCL Extension for Formulating Temporal Constraints. Rapport technique, Universitat Bremen, 2003. (Cité page 87.)

[ZHG05] Paul ZIEMANN, Karsten HÖLSCHER et Martin GOGOLLA : From UML Models to Graph Transformation Systems. *In* M. MINAS, éditeur : *Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM)*, volume 127(4) de *ENTCS*. Elsevier, 2005. (Cité page 68.)

[ZKP00] Bernard P. ZEIGLER, Tag Gon KIM et Herbert PRAEHOFER : *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000. (Cité page 118.)

Annexes

Annexe A

Preuve de bisimulation faible entre xSPEM et PrTPN

Nous détaillons dans cette annexe les différentes étapes de la preuve de bisimulation introduite dans le chapitre 6.

Lemme 1 *Soit* MS *l'ensemble des états possibles d'un modèle* xSPEM *et* PNS *l'ensemble des états possibles d'un PrTPN. Pour tout état du modèle* $S \in$ MS *avec une et une seule activité et pour toute séquence* $u \in T^*$ *tel que* $S_0 \rightarrow^{u*} S$ *:*

1. $\forall \lambda \in T, S' \in \text{MS}, S \rightarrow^{\lambda} S' \implies \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} \Pi(S')$
2. $\forall \lambda \in T, P \in \text{PNS}, \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} P \implies \exists S' \in \text{MS}\, t.\, q. \begin{cases} S \rightarrow^{\lambda} S' \\ \Pi(S') \equiv P \end{cases}$

   où $\rightarrow^{\epsilon}$ *décrit une transition non observable.*

Preuve 3 *Soit* S *l'état d'un modèle et* $u \in T^*$ *tel que* $S_0 \rightarrow^{u*} S$*,*

1. *Soit* S' *l'état d'un modèle contenant une seule activité tel que* $S \rightarrow^{\lambda} S'$*.*
   Donc $\Pi(S)$ *est le réseau de Petri décrit dans la figure 6.5 et contenant une seule activité.*

$$\begin{array}{rcl} I_s & = & \begin{cases} s \mapsto (0,0), f \mapsto (0,w), l \mapsto (min, min), \\ d \mapsto (max - min, max - min) \end{cases} \\ P & = & \{a\_notStarted, a\_started, a\_inProgress, a\_finished\} \\ & & \cup \{a\_tooEarly, a\_ok, a\_tooLate\} \\ T & = & \{a\_start, a\_finish\} \cup \{a\_lock, a\_deadline\} \end{array}$$

   Nous associons aux transitions $StartActivity$ *et* $FinishActivity$ *de la sémantique de* xSPEM *les transitions respectives* a_start *et* a_finish *du réseau de Petri. Les transitions* a_lock *et* $a\_deadline$ *restantes dans le réseau de Petri sont nos transitions epsilons (*ϵ*), c'est-à-dire les transitions qui ne sont pas observables.*

   Nous considérons maintenant les différents cas possibles de transition applicable sur S *:*

– $StartActivity$, *quelque soit la valeur de l'horloge* ($\forall\theta$)
Dans ce cas, S *est tel que* $\exists\ clock, (notStarted, ok, clock)$. *La précondition sur les prédécesseurs est satisfaite car l'activité est considérée comme la seule dans l'état.*
Donc S *est tel que le marquage obtenu par* Π *contient un seul jeton dans la place* nS.

$$m = \{a\_notStarted \mapsto 1\}, I = \{a\_start \mapsto (0, w)\}$$

Selon la sémantique des PN, $m > Pre(s)$. *Après le franchissement de cette transition, le réseau de Petri* (m', I') *obtenu est tel que :*

$$\begin{array}{rcl} m' & = & \{a\_started \mapsto 1, a\_inProgress \mapsto 1, a\_tooEarly \mapsto 1\}, \\ I' & = & \{a\_lock \mapsto (min, min), a\_finish \mapsto (0, w)\}. \end{array}$$

Les transitions epsilons ne sont pas franchies ici.
Revenons sur S'. *Selon la sémantique de MS,* S' *est notre activité avec les valeurs* $(started, ok, 0)$. *Son image par la fonction* Π *donne le réseau de Petri* (m'', I'') *tel que :*

$$\begin{array}{rcl} m'' & = & \{a\_started \mapsto 1, a\_inProgress \mapsto 1, a\_tooEarly \mapsto 1\}, \\ I'' & = & \{a\_lock \mapsto (min, min), a\_finish \mapsto (0, w)\}. \end{array}$$

Nous avons $(m', I') = (m'', I'')$.

– $FinishActivity$ *avec* $\theta < min$
Nous considérons le deuxième cas. S *est tel que l'état est décrit par le triplet* $\exists\ clock$, *tel que* $(started, ok, clock)$ *et* $clock < min\_time$.
Le S' *obtenu est tel que son état est décrit par le triplet* $(finished, tooEarly, clock)$.
L'image de S *par* Π *est* (m, I) *avec*

$$\begin{array}{rcl} m & = & \{a\_started \mapsto 1, a\_inProgress \mapsto 1, a\_tooEarly \mapsto 1\}, \\ I & = & \{a\_lock \mapsto (min, min), a\_finish \mapsto (0, w)\} \end{array}$$

$\Pi(S')$ *est défini comme* (m', I') *où*

$$m' = \{a\_started \mapsto 1, a\_finished \mapsto 1, a\_tooEarly \mapsto 1\}, I = \{\}$$

Nous montrons que

$$(m, I) \xrightarrow{(\epsilon,\theta_1)*} \xrightarrow{(f,\theta_2)} \xrightarrow{(\epsilon,\theta_3)*} (m'', I'')$$

avec $\theta_1 + \theta_2 + \theta_3 = \theta = clock$
La première transition sur $a\_lock \in \epsilon$ *n'est pas applicable, car* $\theta_1 < min$.

$$(m, I) \xrightarrow{(f,\theta_2)} (m''', I''')$$

avec $m''' = \{a\_started \mapsto 1, a\_tooEarly \mapsto 1, a\_finished \mapsto 1\}, I''' = \{\}$ *Donc aucune transition epsilon (*ϵ*) est applicable.*
Et nous avons $(m''', I''') = (m'', I'') = (m, I)$.

– $FinishActivity$ *avec* $min \leq \theta < max$
$S = (started, ok, clock)$ *et* $min\_time \leq clock < max\_time$.
$S' = (finished, ok, clock)$.
$\Pi(S) = (m, I)$ *avec*

$$\begin{array}{rcl} m & = & \{a\_started \mapsto 1, a\_inProgress \mapsto 1, a\_tooEarly \mapsto 1\}, \\ I & = & \{a\_lock \mapsto (min, min), a\_finish \mapsto (0, w)\} \end{array}$$

$\Pi(S') = (m', I')$ *avec*

$$m' = \{a\_started \mapsto 1, a\_finished \mapsto 1, a\_ok \mapsto 1\}, I = \{\}$$

Nous montrons maintenant que

$$(m, I) \xrightarrow{(\epsilon,\theta_1)*} \xrightarrow{(f,\theta_2)} \xrightarrow{(\epsilon,\theta_1)*} (m', I')$$

avec $\theta_1 + \theta_2 + \theta_3 = \theta = clock$
Les transitions a_lock *et* a_finish *sont applicables sur* (m, I).
La transition $a\_lock \in \epsilon$ *pourrait être applicable car* $m > Pre(l)$. *Si* $\theta_1 < min$ *alors la transition* a_lock *n'est pas applicable. Un premier cas est quand* $(m, I) \xrightarrow{(f,\theta_2)} (m_2, I_2)$ *avec* $m_2 = \{a\_started \mapsto 1, a\_finished \mapsto 1, a\_tooEarly \mapsto 1\}$, $I_2 = \{\}$ *et* $\theta_2 = clock$ *Mais selon la sémantique des PN,* θ_2 *doit donc être* $< min$. *La transition n'est pas franchissable.*
Donc θ_1 *doit être* $\geq min$. *En outre* $\theta_1 \leq w$. *La transition se produit.*

$$(m, I) \xrightarrow{(l,\theta_1)} (m'_2, I'_2)$$

avec $m'_2 = \{a\_started \mapsto 1, a\_ok \mapsto 1, a\_inProgress \mapsto 1\}, I'_2 = \{a\_deadline \mapsto (max - min, max - min), a\_finish \mapsto (0, w)\}$ *(nous avons* $m - Pre(l) < Pre(d)$ *&* $m - Pre(l) < Pre(f)$*)*
Nous avons $0 \leq \theta_2 + \theta_3 < max - min$. *Nous franchissons maintenant la transition* f.

$$(m'_2, I'_2) \xrightarrow{(f,\theta_2)} o(m_3, I_3)$$

avec $m_3 = \{a\_started \mapsto 1, a\_ok \mapsto 1, a\_finish \mapsto 1\} et I_3 = \{\}$.
Nous obtenons $(m', I') = (m_3, i_3)$.
– $FinishActivity$ *avec* $\theta > max$
$S = (started, ok, clock)$ *et* $clock > max\_time$.
$S' = (finished, tooLate, clock)$.
$\Pi(S) = (m, I)$ *avec*

$$\begin{array}{l} m = \{a\_started \mapsto 1, a\_inProgress \mapsto 1, a\_tooEarly \mapsto 1\}, \\ I = \{a\_lock \mapsto (min, min), a\_finish \mapsto (0, w)\} \end{array}$$

$\Pi(S') = (m', I')$ *avec*

$$m' = \{a\_started \mapsto 1, a\_finish \mapsto 1, a\_tooLate \mapsto 1\}, I = \{\}$$

Nous montrons maintenant que

$$(m, I) \xrightarrow{(\epsilon,\theta_1)*} \xrightarrow{(f,\theta_2)} \xrightarrow{(\epsilon,\theta_3)*} (m', I')$$

avec $\theta_1 + \theta_2 + \theta_3 = \theta = clock$
Les transitions a_lock *et* a_finish *sont applicables sur* (m, I).
Le même raisonnement que le cas précédent s'applique ici. Donc nécessairement, une première transition a_lock *se produit quand* $\theta_1 \geq min$ *et* $\theta_1 < w$.

$$(m, I) \xrightarrow{(a\_lock,\theta_1)} (m'_2, I'_2)$$

avec

$$\begin{array}{rcl} m'_2 & = & \{a\_started \mapsto 1, a\_ok \mapsto 1, a\_inProgress \mapsto 1\}, \\ I'_2 & = & \left\{ \begin{array}{l} a\_deadline \mapsto (max - min, max - min), \\ a\_finish \mapsto (0, w) \end{array} \right\} \end{array}$$

$(m - Pre(l) < Pre(d)$ *&* $m - Pre(l) < Pre(f))$
Nous regardons si la transition f *peut s'appliquer. Donc* $\theta_2 < max - min$ *et aucune autre transition pourrait être franchie. Mais* $clock = \theta_1 + \theta_2 = max$ *et nous considérons le cas* $clock > max$
Donc nous devons franchir la transition sur d $(m'_2, I'_2) \xrightarrow{(a\_deadline, max-min)} (m''_2, I''_2)$ *avec*

$$\begin{array}{rcl} m''_2 & = & \{a\_started \mapsto 1, a\_tooLate \mapsto 1, a\_inProgress \mapsto 1\}, \\ I''_2 & = & \{a\_finish \mapsto (0, w)\} \end{array}$$

Donc la transition sur f *peut être franchie.* $(m''_2, I''_2) \xrightarrow{(a\_finish,\theta_2)} (m_3, I_3)$ *avec* $m_3 = \{a\_started \mapsto 1, a\_tooLate \mapsto 1, a\_finish \mapsto 1\}$ *et* $I_3 = \{\}$.
Nous obtenons $(m', I') = (m_3, i_3)$.

2. *Soit* P' *l'état d'un réseau de Petri tel que* $\Pi(S) \rightarrow^{\lambda} P'$.
   - $S = (notStarted, notFinished, clock)$

$$(m, I) = (\{a\_notStarted \mapsto 1\}, \{a\_start \mapsto (0, w)\})$$

Il n'y a qu'une seule transition possible $\exists\theta$ *t.q.* $(m, I) \xrightarrow{(a\_start,\theta)} (m', I')$

$$\begin{array}{rcl} m' & = & \{a\_start \mapsto 1, s\_inProgress \mapsto 1, s\_tooEarly \mapsto 1\} \\ I' & = & \{a\_lock \mapsto (min, min), a\_finish \mapsto (0, w)\} \end{array}$$

L'image de S par la même transition donne $S' = (started, notFinished, 0)$ *et* $\Pi(S') = (m', I')$.

– $S = (started, notFinished, clock)$
$\Pi(S) = (m, i)$ *with*

$$\begin{array}{rcl} m & = & \{a\_started \mapsto 1, a\_inProgress \mapsto 1, a\_tooEarly \mapsto 1\}, \\ I & = & \{a\_lock \mapsto (min, min), a\_finish \mapsto (0, w)\} \end{array}$$

Nous avons deux possibilités :

– *appliquons* a_finish *ssi* $\theta < min$ *donc nécessairement,* $(m, I) \xrightarrow{(f,\theta)}$ $(\{a\_started \mapsto 1, a\_finished \mapsto 1, a\_tooEarly \mapsto 1\}, \{\}) = (m', I')$
et $clock = \theta < min\ S \xrightarrow{\lambda_2} (finished, tooEarly, clock)$
$\Pi((finished, tooEarly, clock)) = (m', I')$

– *appliquons* a_lock *ssi* $\theta_1 = min\ (m, i) \xrightarrow{(a\_lock,\theta_1)}$ $(\{a\_started \mapsto 1, a\_inProgress \mapsto 1, a\_ok \mapsto 1\}, \{a\_deadline \mapsto (max-min, max-min), a\_finish \mapsto (0, w)\}) = (m_2, I_2)$ *Nous avons de nouveau deux cas possibles :*

(a) $a\_deadline$ *ssi* $\theta_2 = max - min$ *donc* $(m_2, I_2) \xrightarrow{(a\_deadline,\theta_2)}$ $(\{a\_started \mapsto 1, a\_tooLate \mapsto 1, a\_inProgress \mapsto 1\}, \{a\_finish \mapsto (0, w)\}) = (m_3, I_3)$
Enfin la transition f *peut être appliquée.*
$(m_3, I_3) \xrightarrow{(a\_finish,\theta_3)} (\{a\_started \mapsto 1, a\_tooLate \mapsto 1, a\_finished \mapsto 1\}, \{\}) = (m', I')$
et $clock = \theta_1{+}\theta_2{+}\theta_3 = max{+}\theta_3\ S \xrightarrow{\lambda_4} (finished, tooLate, clock)$
$\Pi((finished, tooLate, clock)) = (m', I')$

(b) a_finish *ssi* $\theta_2 < max - min$ *donc* $(m_2, I_2) \xrightarrow{(a\_finish,\theta_2)}$ $(\{a\_started \mapsto 1, a\_ok \mapsto 1, a\_finish \mapsto 1\}, \{\}) = (m', I')$
$clock = \theta_1{+}\theta_2 < max \& clock \geq min\ S \xrightarrow{\lambda_3} (finished, ok, clock)$
$\Pi((finished, ok, clock)) = (m', I')$

– *les autres valeurs possibles de* S *sont traduites dans le réseau de Petri avec aucune transition applicable.*

3. *Cas initial. De manière triviale* $(m, I) = \Pi(S_0)$ *est défini et satisfait la propriété.*

Lemme 2 *Soit l'état d'un modèle de procédé* $S \in$ MS *avec un nombre fini* n *d'activités, avec des règles de dépendance entre elles, tel que* S *et* $\Pi(S)$ *sont faiblement bisimilaires. Soit l'état d'un modèle de procédé* $S' \supseteq S \in$ MS *défini comme l'état* S *avec une activité* A *de plus sans dépendance. Donc* S' *et* $\Pi(S')$ *sont faiblement bisimilaires.*

Preuve 4 *Si il n'existe pas de lien entre* S *et* A *dans* S' *alors*
– $S \rightarrow X \implies S \cup A \rightarrow X \cup A$

– *De même, dans les réseau de Petri, puisque aucun lien de dépendance existe entre* A *et* S *alors* $\Pi(S') = \Pi(S) \cup \Pi(A)$ *et* $\Pi(S) \rightarrow \Pi(X) \implies \Pi(S \cup A) = \Pi(S) \cup \Pi(A) \rightarrow \Pi(X) \cup \Pi(A)$. *La transition ne peut pas ajouter de lien alors* $\Pi(X) \cup \Pi(A) = \Pi(X \cup A)$.

Un raisonnement similaire s'applique à la transition A *avec la présence de* S *avec aucun lien entre* A *et* S.

– $A \rightarrow A' \implies S \cup A \rightarrow S \cup A'$
– $\Pi(A) \rightarrow \Pi(A') \implies \Pi(S \cup A) = \Pi(S) \cup \Pi(A) \rightarrow \Pi(S) \cup \Pi(A') = \Pi(S \cup A')$.

Comme S *et* $\Pi(S)$ *sont faiblement bisimilaire (par hypothèse d'induction) et en utilisant le lemme 1 :*

– *si une transition* λ *se produit sur* $S \subseteq S'$ *alors* $S \rightarrow^{\lambda} X \implies \Pi(S') \rightarrow^{\lambda} \Pi(X \cup A)$ *;*
– *si une transition* λ *se produit sur* $A \subseteq S'$ *alors* $A \rightarrow^{\lambda} A' \implies \Pi(S') \rightarrow^{\lambda} \Pi(S \cup A')$ *;*
– *si une transition* λ *se produit sur* $\Pi(S) \subseteq \Pi(S')$ *alors* $\Pi(S) \rightarrow \Pi(X) \implies \Pi(S') \rightarrow^{\lambda} \Pi(X \cup A)$
– *si une transition* λ *se produit sur* $\Pi(A) \subseteq \Pi(S')$ *alors* $\Pi(A) \rightarrow \Pi(A') \implies \Pi(S') \rightarrow^{\lambda} \Pi(S \cup A')$

Donc S' *et* $\Pi(S')$ *sont faiblement bisimilaire.*

Lemme 3 *Soit l'état d'un modèle de procédé* $S \in$ MS *avec un nombre fini* n *d'activités, avec des règles de dépendance entre elles, tel que* S *et* $\Pi(S)$ *sont faiblement bisimilaires. Soit l'état d'un modèle de procédé* $S' \supseteq S \in MS$ *défini comme l'état* S *avec en plus un lien de dépendance entre les deux activités* A_1 *et* $A_2 \in S$. *Donc* S' *et* $\Pi(S')$ *sont faiblement bisimilaires.*

Preuve 5 *Le nouveau lien de dépendance contraint l'activité* A_2 *par l'activité* A_1. *Pour toute les transitions* λ *applicables à toute activité* $A \in S \setminus A_2$, *la transition peut être franchie dans* $\Pi(S')$ *car* S *et* $\Pi(S)$ *sont faiblement bisimilaire et l'activité* A *n'est pas contrainte par le nouveau lien. Et réciproquement, si la transition peut se produire dans* $\Pi(A) \subseteq \Pi(S)$ *alors elle peut se produire dans* $\Pi(S')$.

Nous considérons maintenant que les transitions applicables sur A_2 *dans* S' *dépendent du nouveau lien ajouté. Nous pouvons ajouter comme remarque préliminaire que si une transition peut se produire sur* A_2 *dans* S', *elle peut également être franchie dans* S *car le nouveau lien de dépendance n'existe pas.*

– *un lien de type* startToStart
  Alors, selon la définition de la figure 5.4, tous les liens ciblant cette activité A_2 *et étiqueté* startToStart, *resp.* finishToStart, *doivent avoir leur activité source dans l'état commencé, resp. dans l'état fini, afin de franchir la transition start sur* A_2.
  Nous montrons maintenant que si la transition est franchie dans S' *alors elle l'est dans* $\Pi(S')$.

Si la transition est franchie dans S', alors tous les liens au dessus dans S contraignant le démarrage de A_2 satisfont leurs propres contraintes (soit démarré, soit fini). $\Pi(S)$ *est tel que pour chacun de ces liens il existe un* read-arc *dans le réseau de Petri obtenu, de $ax\_started$ ou $ax\_finished$, en fonction du type de lien, vers $a2\_start$. Chacun de ces* read-arc *à pour source une place remplie avec un jeton (cf. remarque préliminaire).*
De plus, dans $\Pi(S')$*, le nouveau lien de A_1 vers A_2 est aussi traduit en un* read-arc *de la place $a1\_started$ vers la transition $a2\_start$. La transition est franchie dans S' alors l'activité A_1 doit être démarrée.*
Furthermore, in $\Pi(S')$*, the new link from A_1 to A_2 is also mapped to a read-arc from the place $a1\_started$ to transition $a2\_start$. The transition is computable in S' then the activity A_1 must be started. Si c'est vrai, sa place $a1\_started$ a un jeton.*
La transition peut alors est franchie dans $\Pi(S')$.

– *un raisonnement similaire s'applique pour les dépendances de type* finishToStart, startToFinish *et* finishToFinish.

Réciproquement, dans les réseaux de Petri,

– *l'image d'un lien de type $startToStart$*
$\Pi(S') = \Pi(S) \cup \{$ *un nouveau* read-arc *de $a1\_started$ vers $a2\_started$*$\}$.
Alors si $\Pi(S') \rightarrow Y$ utilisant la transition $a2\_start$, alors il doit y avoir au moins un jeton dans chaque place liée à $a2\_start$ par soit un arc soit un read-arc. *Alors par définition de* Π *et utilisant la remarque préliminaire, S' est tel que toutes les activités contraignant le démarrage de $A2$ satisfont leurs propres contraintes, incluant le nouveau lien. Alors la transition peut être franchie dans S'.*

– *un raisonnement similaire s'applique pour les dépendances de types* finishToStart, startToFinish *and* finishToFinish.

Parce que S et $\Pi(S)$ *sont faiblement bisimilaire (par hypothèse d'induction), nous avons S' et* $\Pi(S')$ *qui sont également en bisimulation faible.*

Théorème 4 (Bisimulation faible) *Pour tout état de modèle* $S \in$ MS *et pour toute séquence* $u \in T*$ *tel que* $S_0 \rightarrow^{u*} S$ *:*

1. $\forall \lambda \in T, S' \in \text{MS}, S \rightarrow^{\lambda} S' \implies \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} \Pi(S')$
2. $\forall \lambda \in T, P \in \text{PNS}, \Pi(S) \rightarrow^{\epsilon*} \rightarrow^{\lambda} \rightarrow^{\epsilon*} P \implies \exists S' \in \text{MS}\, t.\, q. \begin{cases} S \rightarrow^{\lambda} S' \\ \Pi(S') \equiv P \end{cases}$

Preuve 6 *Par induction sur la structure du modèle de procédé :*

– *L'état initial est prouvé grâce au lemme 1 ;*
– *L'ajout d'une activité préserve la propriété (lemme 2) ;*
– *L'ajout d'un lien de dépendence préserve la propriété (lemme 3) ;*

La décomposition hiérarchique des activités est traduite par des liens de dépendance et préserve donc la propriété de bisimulation.

Annexe B

Outils pour éditer, simuler et vérifier des modèles xSPEM

Dans le contexte de cette thèse, nous avons développé différents outils permettant de construire des modèles de procédé en xSPEM, de les simuler et de les vérifier formellement par *model-checking*. La description et le code source de ces outils sont disponibles à l'adresse http://combemale.perso.enseeiht.fr/xSPEM/.

A cette même adresse nous présentons également les outils issus d'autres applications de l'approche proposée dans cette thèse. Plus particulièrement :

- un simulateur de machine à états UML2.0 qui est intégré dans l'atelier TOPCASED à partir de la version 2, synchronisée avec Eclipse Ganymede en juillet 2008.
- une chaîne de vérification formelle pour des programmes Ladder, prenant plus particulièrement en compte la détection de *race* condition.

Résumé

Nous proposons dans cette thèse une démarche permettant de décrire un DSML (*Domain Specific Modeling Language*) et les outils nécessaires à l'exécution, la vérification et la validation des modèles. La démarche que nous proposons offre une architecture générique de la syntaxe abstraite du DSML pour capturer les informations nécessaires à l'exécution d'un modèle et définir les propriétés temporelles qui doivent être vérifiées. Nous nous appuyons sur cette architecture pour expliciter la sémantique de référence et l'implanter. Plus particulièrement, nous étudions les moyens :

- d'exprimer et de valider la définition d'une traduction vers un domaine formel dans le but de réutiliser des outils de model-checking.
- de compléter la syntaxe abstraite par le comportement ; et profiter d'outils génériques pour pouvoir simuler les modèles construits.

Enfin, de manière à valider les différentes sémantiques implantées vis-à-vis de la sémantique de référence, nous proposons un cadre formel de métamodélisation.

Abstract

We propose in this thesis a specific taxonomy of the mechanisms allowing to express an execution semantics for *Domain Specific Modeling Languages* (DSMLs). Then, we integrate these different mechanisms within a comprehensive approach describing DSMLs and tools required for model execution, verification and validation. The proposed approach provides a rigorous and generic architecture for DSML abstract syntax in order to capture the information required for model execution. We rely on this generic architecture to make the reference semantics explicit and implement it. More specifically, we study the means :

- to express and validate the definition of a translation into a formal domain in order to re-use model-checking techniques.
- to enrich the abstract syntax with the definition of the DSML behaviour and take advantage of generic tools so to simulate the built models.

Finally, for the purpose of validating the equivalence of different semantics implemented according to the reference semantics, we also propose a formal metamodeling framework.