Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS

Stéphane Garredu

UNIVERSITE DE CORSE-PASCAL PAOLI

ECOLE DOCTORALE ENVIRONNEMENT ET SOCIETE
UMR CNRS 6134 S.P.E.

Thèse présentée pour l'obtention du grade de
DOCTEUR DE L'UNIVERSITÉ DE CORSE
en Informatique

Soutenue publiquement par

Stéphane GARREDU

le 16 Juillet 2013

Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS

Directeur(s) :

M. Jean-François SANTUCCI, Professeur, Università di Corsica
Mme Evelyne VITTORI, Maître de Conférences, Università di Corsica

Rapporteurs :

M. Mourad OUSSALAH, Professeur, Université de Nantes
M. Hans VANGHELUWE, Professeur, Université d'Antwerp

Jury

M. Mourad OUSSALAH, Professeur, Université de Nantes
M. Hans VANGHELUWE, Professeur, Université d'Antwerp (Anvers) et McGill
M. Jean-Jacques CHABRIER, Professeur Émérite, Université de Bourgogne
M. Paul BISGAMBIGLIA, Professeur, Università di Corsica
M. Jean-François SANTUCCI, Professeur, Università di Corsica
Mme Evelyne VITTORI, Maître de Conférences, Università di Corsica

UNIVERSITE DE CORSE-PASCAL PAOLI

ECOLE DOCTORALE ENVIRONNEMENT ET SOCIETE
UMR CNRS 6134 S.P.E.

Thèse présentée pour l'obtention du grade de
DOCTEUR DE L'UNIVERSITÉ DE CORSE
en Informatique

Soutenue publiquement par

Stéphane GARREDU

le 16 Juillet 2013

Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS

Directeur(s) :
M. Jean-François SANTUCCI, Professeur, Università di Corsica
Mme Evelyne VITTORI, Maître de Conférences, Università di Corsica

Rapporteurs :
M. Mourad OUSSALAH, Professeur, Université de Nantes
M. Hans VANGHELUWE, Professeur, Université d'Antwerp

Jury

M. Mourad OUSSALAH, Professeur, Université de Nantes
M. Hans VANGHELUWE, Professeur, Université d'Antwerp (Anvers) et McGill
M. Jean-Jacques CHABRIER, Professeur Émérite, Université de Bourgogne
M. Paul BISGAMBIGLIA, Professeur, Università di Corsica
M. Jean-François SANTUCCI, Professeur, Università di Corsica
Mme Evelyne VITTORI, Maître de Conférences, Università di Corsica

Pà Nanò è Pépé,

Pà Papy è Mamy,

Pà a mè famiglia,

É pà Anne-Laure

À voi tutti ch'aveti fattu cio ch'eiu sò oghji

Remerciements

Un travail de thèse ne s'accomplit jamais de manière individuelle mais grâce à l'aide précieuse de plusieurs personnes, qui, chacune dans son domaine, chacune à sa manière, contribue à l'aboutissement de ce projet. Que ce soit par leur savoir-faire, leur aide technique, leurs conseils, leur expérience, leur vision de la recherche, ou, tout simplement, leur soutien moral, leur amitié ou leur amour, toutes ces personnes méritent d'être chaleureusement remerciées, et c'est ce que je vais tenter de faire en écrivant ces quelques lignes.

Ce travail de thèse s'est déroulé au sein de l'équipe du projet Technologies de l'Information et de la Communication (T.I.C.) de l'Unité Mixte de Recherche 6134 Systèmes Pour l'Environnement (S.P.E.) de l'Università di Corsica. Il a été réalisé grâce au soutien technique et financier conjoint du Ministère de l'Enseignement Supérieur et de la Recherche, et de l'Università di Corsica et n'aurait pas été possible sans eux.

Je voudrais en premier lieu remercier M. Jean-François SANTUCCI, professeur à l'Università di Corsica, qui a réussi à me faire partager sa passion pour la recherche lorsque j'étais en Master et qui, depuis lors, a dirigé mes travaux. Ses connaissances du monde de la recherche et de la modélisation et simulation, son expérience, son ouverture d'esprit, ses conseils et sa gentillesse m'ont énormément aidé pour accomplir ce travail.

Mes travaux sont, depuis mon Master, co-encadrés par Evelyne VITTORI, maître de conférences à l'Université de Corse, qui s'est toujours énormément impliquée dans son rôle, toujours avec beaucoup de patience, de pédagogie, et de bonne humeur. Ses idées pertinentes, nos discussions enrichissantes, sa disponibilité sans faille, ses connaissances techniques, ont permis de cadrer mes travaux, de les étoffer, et de leur donner la rigueur dont ils pouvaient parfois manquer. Pour tout cela, je la remercie chaleureusement.

Ma reconnaissance va également à M. Paul BISGAMBIGLIA, professeur à l'Università di Corsica et directeur de l'UMR 6134, pour ses conseils, et pour l'intérêt qu'il porte à mes travaux en acceptant de participer à ce jury. Il m'a accueilli au sein de ce laboratoire et fait en sorte que mes années d'enseignement et de recherche se déroulent du mieux possible.

À M. Hans VANGHELUWE, professeur à l'Université McGill de Montréal et à l'Université d'Anvers, et à M. Mourad Chabane OUSSALAH, professeur à l'Université de Nantes, qui ont tous deux manifesté de l'intérêt pour cette thèse en acceptant d'en être les rapporteurs, j'exprime ma profonde gratitude.

M. Jean-Jacques CHABRIER, professeur émérite de l'Université de Bourgogne, a accepté de participer à ce jury et d'en être le président : qu'il trouve ici l'expression de ma profonde reconnaissance.

Merci à toute l'équipe informatique du projet TIC, stagiaires, doctorants et enseignants, pour leur soutien, et en particulier à mes amis et anciens compagnons de bureau Paul-Antoine BISGAMBIGLIA, maître de conférences à l'Université de Corse, qui a relu ce manuscrit, et m'a prodigué des conseils avisés tout au long de mon cursus, et à Jean-Sébastien GUALTIERI, qui m'a apporté son soutien et sa sempiternelle jovialité. Leurs encouragements ont été très importants.

Je tiens à témoigner toute ma gratitude à M. Dominique URBANI, docteur en informatique, et à M. Dominique CASANOVA, amateur éclairé, pour leur relecture attentive de ce manuscrit et leurs conseils.

Merci, d'une manière plus générale, au personnel universitaire avec qui j'ai été en contact, comme certains professeurs et la scolarité de la FST, les doctorants, la comptabilité, et avec une pensée toute particulière pour l'équipe du Centre de Ressources Informatiques, notamment mon ami M. Antò MARTINETTI.

Mes amis m'ont énormént apporté durant ces années, certains se sont même déplacés pour ma soutenance et je les en remercie vivement.

Enfin, je tiens à terminer ces longs remerciements en rendant hommage à ma compagne et à ma famille, car sans eux ce document n'aurait jamais existé : c'est à eux que je veux le dédier, car cette thèse est aussi la leur.

Anne-Laure m'a supporté au quotidien comme elle a supporté les contraintes impliquées par ce travail, elle m'a toujours énormément soutenu, et je lui témoigne ici toute ma reconnaissance et tout mon amour.

Mes parents, ma sœur, mes oncles et tantes, mes cousines, ma fantastique filleule, et plus généralement toute ma famille, m'ont encouragé dans absolument tout ce que j'entreprenais et se sont tous intéressés avec enthousiasme à mes travaux.

Mes grands-parents occupent une place particulière dans ces remerciements. J'ai eu la chance de les connaître tous les quatre et ils ont toujours été, chacun à sa manière, mes plus fervents supporters. Je remercie tout d'abord mes grands-parents maternels pour tout ce qu'ils ont fait et font pour moi, et pour l'amour immense qu'ils me portent. Ils se sont toujours intéressés au plus haut point à l'évolution de mes études. Ce document tient tellement à cœur à mon grand-père qu'il dit, non sans humour, qu'il est très content de « passer SA thèse ». La voici !

J'ai, enfin, une pensée très particulière pour mes grand-parents paternels qui ne sont plus parmi nous. Je garde d'eux un souvenir merveilleux. Ma grand-mère, qui est malheureusement partie quelques semaines avant la date soutenance, était très fière de ce manuscrit. Ses yeux brillaient lorsqu'elle parlait de moi, les miens brillent lorsque je parle d'elle. Par ces quelques lignes, ce manuscrit sera à jamais associé à sa mémoire.

Résumé

Cette thèse s'inscrit au carrefour du monde de la modélisation et simulation de systèmes d'une part et du monde de l'ingénierie logicielle d'autre part. Elle vise à faire bénéficier un formalisme de spécification de systèmes à évènements discrets (DEVS) des apports de l'ingénierie dirigée par les modèles (IDM) avec l'une de ses incarnations les plus populaires : MDA (Model Driven Architecture). Le formalisme DEVS de par son adaptabilité et son extensibilité permet l'expression et la simulation de modèles dans des domaines très variés, mais l'existence de plusieurs plateformes dédiées à ce langage nuit fortement à l'interopérabilité de ces modèles. Ces difficultés, si elles ne sont pas nouvelles, représentent cependant un défi d'autant plus important que les modèles considérés sont complexes (i.e composés en général de nombreux sous modèles et interagissant fortement entre eux).

L'objectif de la thèse est de proposer une réponse à la problématique de l'interopérabilité des modèles DEVS, vis-à-vis d'autres formalismes voisins de DEVS et également vis-à-vis des différents simulateurs existants. Le cœur de notre travail est constitué par MetaDEVS, méta-modèle offrant une représentation des modèles DEVS indépendante des plateformes. MetaDEVS est également le nom donné à l'approche globale qui vise à fournir des passerelles génériques entre différents formalismes et DEVS («Model-To-Model»). Cette approche montre également comment, à partir de modèles DEVS spécifiés selon MetaDEVS, du code orienté-objet, simulable, peut être automatiquement généré («Model-To-Text»).

Les formalismes choisis pour faire l'objet d'une transformation vers DEVS sont BasicDEVS, un petit formalisme pédagogique créé pour l'occasion, ainsi que les automates à états finis (FSM). La plateforme de destination choisie pour la génération de code est la plateforme éducative PyDEVS, compatible avec la plateforme DEVSimPy, utilisée par les chercheurs du projet TIC de l'Università di Corsica.

Abstract

This thesis takes place at the intersection between the world of modeling and simulation, and the world of software engineering. Its provides a contribution to a discrete-event specification formalism (DEVS) using techniques of Model-Driven Engineering, with one of its most popular incarnations: MDA (Model Driven Architecture). The DEVS formalism, thanks to its adaptability and its extensibility, is able to express and simulate models in various domains. However, the existence of many dedicated platforms damages the interoperability of those models. Those difficulties, even if they are not new, are a challenge which is all the greater as the studied models are complex (i.e. usually composed of several submodels with a strong interaction).

The main purpose of this thesis is to tackle the problem of the DEVS models interoperability, with respect to other formalisms close to DEVS, and also with respect to the different existing simulators. The core of our work is constituted by MetaDEVS, a metamodel that offers a platform-independent representation of DEVS models. MetaDEVS is also the name given to the global approach which aims to provide generic bridges between different formalisms and DEVS ("Model-To-Model"). This approach also shows how, starting from DEVS models specified with MetaDEVS, object-oriented code can be automatically generated ("Model-To-Text").

The formalisms chosen to be transformed into DEVS are BasicDEVS, a small pedagogical formalism create for our needs, and the finite state machines (FSM). The chosen target platform for the code generation is the educative framework PyDEVS, compliant with the DEVSimPy framework used by the researchers of the TIC project of the University of Corsica.

Riassuntu

Quista tesa si scrivi à a crucciata di u mondu di a mudillisazzioni è simulazioni d'i sistemi d'una parti, è di u mondu di l'inghjinirìa d'i prugrammi di l'altra parti. U so scopu hè di parmetta ch'un furmalisimu di specificazioni di sistemi à avvinimenti scuntinii (DEVS) pudissi prufittà d'i tecnichi di l'inghjinirìa capitanata da i mudelli (IDM), grazia à u pupulari attrizzu MDA (Model Driven Architecture). U furmalisimu DEVS, cù i so capacità à essa apprupiatu è stesu, hè capaci à sprimà è simulà mudelli in parechji campi. Tandu, ch'elli fussini cusì numarosi i rimpianati didicati à DEVS nuci assà à l'interuparabilità d'issi mudelli. Quisti anciampi, ancu s'elli ùn sò novi, sò una sfida chi cresci cù a cumplessità d'i mudelli studiati (par chi, spessu, sò cumposti da mudelli numarosi à livelli più bassi, ligati trà elli di manera forti).

U scopu principali di 'ssu travaghju hè di prupona una risposta à u prublema di l'interuparabilità d'i mudelli DEVS, cù d'altri furmalisimi vicini à DEVS, mà dinò cù i sfarenti rimpianati ch'esistini par avà. U cori di u nostru travaghju hè MetaDEVS, meta-mudellu chi pruponi una riprisintazioni d'i mudelli DEVS di manera indipendenti da i rimpianati. MetaDEVS hè dinò u nomi datu à a dimarchja ghjinarali, di a quali u scopu hè di custruiscia puntili ghjinerichi trà sfarenti furmalismi è DEVS («Model-To-Model»). 'Ssa dimarchja mostra dinò comu, partendu di mudelli DEVS spicificati cù MetaDEVS, codici « ughjettu », pudendu essa simulatu, si pò producia di manera automatica (« Model-To-Text »).

I furmalisimi scelti pà essa tracambiati versu DEVS sò BasicDEVS, un chjucu furmalisimu pidagoghjicu fattu pà l'uccasioni, è ancu l'automati à stati finiti (FSM). A rimpianata scelta pà a ghjinirazioni di codici hè a rimpianata educativa PyDEVS, cumpatibili incù DEVSimPy, chi hè a rimpianata adduprata da i circatori di u prughjettu TIC di l'Università di Corsica.

Sommaire

Introduction Générale

L'ETRE humain a, depuis la nuit des temps, toujours cherché à comprendre le monde qui l'entourait, à transmettre ses connaissances et à prévoir, voire maîtriser, des phénomènes naturels. Issu de l'approche systémique, elle-même née peu après la cybernétique, le domaine de la modélisation et simulation (M&S) apporte un cadre formel pour la spécification de systèmes, continus ou discrets, naturels ou artificiels. La spécification d'un système revient à définir celui-ci à l'aide d'un ou de plusieurs modèles, selon un formalisme de modélisation. Utilisée conjointement avec l'outil informatique, la M&S se révèle être un ensemble d'outils indispensable pour l'aide à la décision, l'amélioration des connaissances scientifiques sur divers systèmes, la mise en commun d'informations entre experts.

Parallèlement à l'essor de la M&S, d'énormes progrès ont été faits en génie logiciel, notamment, depuis une douzaine d'années, en termes de réutilisabilité, d'interopérabilité et d'évolutivité, grâce notamment aux apports capitaux de l'Ingénierie Dirigée par les Modèles (IDM). Dans une philosophie IDM [Bézivin et al. 2002a] [Bézivin 2004], et contrairement aux pratiques classiques de développement, le code n'est plus vu comme l'aboutissement final d'un projet, mais comme un simple aspect de celui-ci. Les projets sont désormais centrés sur un élément fondamental : le modèle. Tout comme en M&S, celui-ci est spécifié au moyen d'un formalisme de modélisation, mais de manière indépendante de toute plateforme et à haut niveau d'abstraction : on se focalise sur les concepts plutôt que sur l'implémentation. Ceci permet, lors de changements de plateformes par exemple, de conserver le modèle, de le transporter, puis de le transformer afin d'obtenir du code. On cherche à générer ce code par raffinements successifs des modèles, de la manière la plus automatisée qui soit. On parle de passage du stade « contemplatif » au stade « productif », stade auquel le modèle est, très souvent, disponible sous forme de code.

Dans ce monde où « tout est modèle », il existe à l'heure actuelle un outillage conséquent permettant de spécifier des modèles de modèles : des méta-modèles. Un méta-modèle est un moyen de décrire un langage ou un formalisme de modélisation, et donc de fournir la syntaxe abstraite nécessaire à la spécification de modèles. Actuellement, beaucoup de méta-modèles sont disponibles, certains, très généraux, ne sont pas liés à un domaine particulier, et c'est le cas d'un des plus célèbres d'entre eux, UML (*Unified Modeling Language*) [Rumbaugh et al. 2005], d'autres, au contraire, sont plus spécifiques : on parle alors de DSL (*Domain Specific Languages*) [Lédeczi et al. 2001].

Tout comme chaque modèle se conforme à son méta-modèle, chaque méta-modèle se conforme lui-même à un modèle d'abstraction plus élevé, appelé méta-méta-modèle, qui décrit un méta-formalisme. Pour éviter la prolifération des méta-formalismes, plusieurs initiatives en IDM ont vu le jour : une des plus déterminantes a été la *Model Driven Architecture* (MDA) [Kleppe et al. 2004] de l'*Object Management Group* (OMG)[1] et son méta-formalisme *Meta-Object Facility* (MOF).

[1] http://www.omg.org

Dans le monde de la M&S, le formalisme DEVS (*Discrete EVent system Specification*) [Zeigler 1976] [Zeigler 1984], est dédié à la spécification de systèmes évoluant en fonction d'évènements discrets (i.e. non continus dans le temps). Il apparaît pour nous comme un outil fondamental : il permet des travaux interdisciplinaires car sa souplesse et son adaptabilité, son extensibilité, font qu'il peut convenir à l'expression et à la simulation de modèles dans des domaines très variés. Il a été implémenté sur plusieurs plateformes, dans plusieurs environnements de modélisation et simulation dédiés. Il offre la possibilité de représenter des systèmes de façon comportementale et structurelle, ce qui permet de créer des interconnections de modèles et d'inclure des modèles dans d'autres modèles (hiérarchie). Une fois qu'un modèle a été défini, il doit, pour être productif, être implémenté dans un langage de programmation (en général, orienté objet) : le simulateur associé à ce modèle est automatiquement créé, par un processus d'instanciation. Les algorithmes de simulation de DEVS sont réputés robustes, et plusieurs extensions sont venues compléter ce formalisme (Cellulaire, Parallèle…).

Cependant, DEVS est en quelque sorte victime de son succès, car l'existence de plusieurs plateformes dédiées à ce langage nuit à l'interopérabilité des modèles et freine les collaborations entre scientifiques. Le groupe chargé de la standardisation de DEVS [SISO 2008] n'a d'ailleurs pas arrêté de choix définitif quant à la manière de représenter des modèles DEVS. Plusieurs travaux visent également à transformer vers DEVS des modèles, exprimés dans d'autres formalismes, basés notamment sur les états-transitions, afin de faire bénéficier ces modèles de la puissance de simulation de DEVS. Dans ce contexte, l'IDM apparaît comme une approche très prometteuse pour faciliter la création, la modification, et la transformation de modèles DEVS.

Objectifs de la thèse

Le but de ce travail est, après avoir étudié les solutions existantes, de proposer une réponse à la problématique de l'interopérabilité des modèles DEVS, en faisant bénéficier ces derniers des techniques issues de l'IDM afin de les rendre interopérables tout en préservant leur productivité, c'est-à-dire leur capacité à être simulés. Nous apportons une contribution, à plusieurs niveaux, au formalisme DEVS. Cette dernière se compose de trois idées directrices et complémentaires :

- **L'étape centrale et fondamentale : la création d'un méta-modèle que nous avons baptisé MetaDEVS.** Le rôle de MetaDEVS (introduit dans [Garredu et al. 2012a]) est de proposer une représentation des modèles DEVS indépendante des plateformes de simulation. Pour définir ce méta-modèle, nous nous basons sur la définition formelle de DEVS établie par le Pr. Zeigler, en y ajoutant un certain nombre de contraintes et restrictions. Nous introduisons en particulier une approche de définition des fonctions DEVS, elles-mêmes indépendantes de toute plateforme, et évolutives. C'est ce qui

nous différencie des approches existantes visant à méta-modéliser DEVS, présentées dans le chapitre III de ce document.

- **MetaDEVS peut être la cible de transformations « Model To Model » (M2M) à partir d'autres formalismes**, qui par exemple ne bénéficient pas de simulateurs. Nous présentons une démarche générale, une méthodologie, pour créer des correspondances entre d'autres formalismes (que nous caractérisons) et DEVS en facilitant la définition de règles de transformation M2M. Nous appliquons ensuite concrètement cette démarche à travers deux exemples de transformations.

  Pour ce faire, nous utilisons comme premier exemple un formalisme inspiré de DEVS, baptisé BasicDEVS (introduit pour la première fois dans [Garredu et al. 2009]), dont nous définissons le méta-modèle et la transformation vers DEVS, grâce à une approche privilégiant une implémentation dite « hybride » (combinaison de règles de transformation déclaratives et impératives).

  Nous illustrons de surcroît que des formalismes un peu plus éloignés de DEVS que BasicDEVS, tels que les automates à états finis, peuvent être eux aussi transformés aisément vers DEVS en vertu de leur appartenance à la famille des modèles à états-transitions. Pour ce faire, nous définissons également leur méta-modèle, de la même manière que nous l'avons défini pour DEVS et BasicDEVS et nous utilisons encore une fois des règles de transformation hybrides. Nous montrons ainsi qu'une partie d'entre elles peuvent être réutilisées telles quelles.

- **MetaDEVS peut être implémenté grâce à des transformations « Model To Text » (M2T)**. Un seul modèle exprimé en MetaDEVS peut avoir autant d'équivalents implémentés que de règles de transformation auront été spécifiées entre MetaDEVS et des plateformes cibles (vers PyDEVS, vers DEVSJAVA, vers PowerDEVS, etc…). Pour chaque modèle DEVS exprimé à l'aide de MetaDEVS nous générons en sortie, et de manière totalement automatique, des modèles DEVS en code objet (nous avons choisi l'exemple de Python à travers le simulateur PyDEVS). Une telle transformation ne se définit qu'une fois par plateforme de simulation. Elle permet de rendre productifs (et donc simulables) les modèles définis avec MetaDEVS. Nous nous basons sur une approche de transformation de modèle à texte dite « par templates ». Nous présentons ici aussi des règles génériques pouvant être réutilisées avec un minimum de modifications, notamment grâce à des mécanismes d'héritage et de polymorphisme des templates.

Structure du document

Ce manuscrit s'organise en deux grandes parties.

Dans la première, composée de trois chapitres, nous dressons un état de l'art sur la M&S, sur l'IDM, et enfin sur les apports existants de l'IDM au formalisme DEVS.

La seconde, composée également de trois chapitres, développe quant à elle nos contributions.

Partie 1 : Au carrefour de l'IDM et de la M&S

Le premier chapitre de cette partie est consacré à l'étude des fondamentaux de la modélisation et simulation. Système, modèle et simulation sont des concepts qui n'existent que depuis récemment sous leur forme actuelle, et s'avèrent très utiles pour l'étude et l'amélioration de réalisations humaines (électronique, mécanique, réalisation complexes…) ou l'analyse de systèmes naturels. Ils restent difficiles à définir formellement, chaque domaine scientifique les adaptant à ses besoins particuliers. Nous tentons néanmoins de caractériser le plus précisément possible ces notions importantes qui sont maintes fois évoquées et utilisées au cours de nos travaux. Un croquis, une recette de cuisine, sont des modèles que nous utilisons presque quotidiennement. Mais, pour pouvoir effectuer des observations sous certaines conditions, et obtenir des résultats, il faut agir sur le modèle : cette action s'appelle la simulation. Par analogie avec l'expérience, la simulation se rapporte à tout ce qui touche au système et au cadre expérimental. La systémique a apporté un socle commun pour toutes les disciplines : un système pouvait enfin être un concept commun. Nous terminons en expliquant de quelle manière le monde de la modélisation et simulation a utilisé ces concepts pour donner naissance à des formalismes tels que DEVS.

Dans le second chapitre, nous montrons comment l'ingénierie dirigée par les modèles) a développé des approches qui placent les modèles au centre du cycle de vie des logiciels. Ces modèles décrivent les concepts essentiels d'un domaine, et ils sont l'élément incontournable de l'IDM. L'emploi de modèles est préconisé dans un souci d'interopérabilité et de réutilisabilité, et surtout dans un souci d'économie de code, car un modèle productif implique une génération automatisée, totale ou partielle, de code. Nous débutons ce chapitre par l'introduction de concepts de base de l'IDM, puis nous donnons une vue d'ensemble de la philosophie de cette approche, des acteurs qui la font évoluer et des environnements qui l'ont implémentée. Nous présentons également des approches plus spécifiques telles que le *Model Integrated Computing* (MIC), la *Generative Programming*, les *Softwares Factories*, et l'approche-standard de l'OMG : MDA. Nous nous concentrons plus particulièrement sur l'environnement Eclipse EMF [Steinberg et al. 2003], sur son méta-formalisme Ecore, et nous étudions le fonctionnement des plugins Acceleo et ATL (*ATLas Transformation Language*) [Jouault et al. 2006b] [Jouault et al. 2008].

Le troisième chapitre est consacré à l'état de l'art du domaine de l'IDM et de ses applications au formalisme DEVS : nous y verrons comment les équipes de chercheurs ont tenté de rendre compatibles des composants DEVS séparés par les plateformes et les

langages. Nous nous concentrons sur les approches dans lesquelles un méta-modèle DEVS a été défini et qui proposent éventuellement des transformations de modèles vers ou depuis DEVS. Nous étudions quels formalismes ont déjà été transformés vers DEVS, et comment ils l'ont été. Ce chapitre, qui vient clore la première partie de ce document, sera conclu par une comparaison entre les approches existantes et notre approche MetaDEVS, qui sera présentée tout au long de la seconde partie de ce manuscrit.

Partie 2 : Approche MetaDEVS

Cette partie débute avec le quatrième chapitre, entièrement dédié à notre principale contribution : la création de MetaDEVS, qui désigne tout d'abord un méta-modèle pour le formalisme DEVS mais aussi, par extension, l'ensemble de notre approche. Nous expliquons pas à pas comment nous avons créé une architecture de méta-classes permettant de décrire un modèle DEVS, qu'il soit atomique ou couplé. Nous insistons particulièrement sur la notion d'état DEVS, et nous proposons une manière de gérer ces derniers dans MetaDEVS Une fois défini le concept d'état, nous l'utilisons pour proposer un formalisme de définition de fonctions, intégré pleinement au méta-modèle. Ces fonctions sont indépendantes de toute plateforme. Elles permettent de déclarer des tests sur des variables d'état et de les modifier de manière basique. La syntaxe abstraite définie par ce méta-modèle est ensuite enrichie d'une sémantique statique, grâce à des contraintes OCL[1] (*Object Constraint Language*) [Richters et al. 2002] [Garredu et al. 2012b]. Nous montrons comment implémenter les concepts de ce méta-modèle dans Eclipse EMF et comment instancier un modèle DEVS.

Le cinquième chapitre se place en amont de DEVS, et considère le méta-modèle MetaDEVS, dont nous disposons à présent, comme un pivot : il nous sert de point d'entrée vers le formalisme DEVS. Notre but est de montrer qu'il est facile de transformer des modèles à évènements discrets vers DEVS, pour peu que l'on dispose de leur méta-modèle. Nous expliquons comment nous avons conçu ces méta-modèles, puis nous présentons une démarche générale pour l'écriture de règles de transformation d'un modèle source *x* vers un modèle DEVS. Nous appliquons cette démarche à deux formalismes : un formalisme inspiré de DEVS que nous avons défini afin de créer des modèles pédagogiques, et les automates à états finis. Les règles de transformation sont déclarées en langage ATL, nous en donnons les principales caractéristiques ainsi que des fragments de code.

Le sixième chapitre se situe quant à lui en aval de DEVS et couvre un autre aspect très important de l'IDM : la génération de code. Grâce au plug-in Acceleo d'Eclipse, nous expliquons comment créer des règles de transformation M2T pour rendre les modèles DEVS productifs (indifféremment, ceux créés grâce à MetaDEVS ou ceux transformés vers ce dernier). Nous expliquons notre démarche de manière générale, en mettant en lumière les éléments requis pour une telle transformation : ces éléments peuvent être des exemples de code objet, ou des templates fournis par les auteurs du formalisme, comme dans le cas de PythonDEVS. Nous procédons selon une approche basée sur les templates. Nous montrons

[1] http://www.omg.org/spec/OCL/2.3.1

ainsi comment créer une chaîne de génération complète, depuis un modèle exprimé dans un formalisme différent de DEVS et indépendant de toute plateforme, vers un modèle DEVS codé en langage objet et prêt à être simulé. Ces modèles pédagogiques, spécifiés dans d'autres formalismes que MetaDEVS, transformés vers ce dernier et enfin transformés en code objet permettent ainsi de valider notre approche.

PREMIERE PARTIE

Au carrefour de l'IDM et de la M&S

Chapitre I. MODELES ET FORMALISMES EN M&S

DANS le monde scientifique actuel, que ce soit pour comprendre ou prévoir les phénomènes naturels, améliorer les calculs au niveau des machines, des processeurs, ou plus généralement étudier des systèmes réels ou artificiels, les chercheurs et les ingénieurs ont recours à une discipline qui se situe à la confluence de plusieurs domaines de la science : la Modélisation et Simulation (M&S).

Elle-même constituée de deux sous-disciplines, la M&S vise d'une part à créer des modèles qui soient le plus conformes possibles à la réalité, ou du moins à une partie de cette réalité, et ensuite les traduire en code informatique dans le but de les faire évoluer dans le temps et observer leur « comportement » virtuel.

La création de ces modèles, la modélisation, fait appel à plusieurs techniques, et requiert l'utilisation d'un formalisme de modélisation. Un formalisme de modélisation n'est rien d'autre qu'un langage de description de modèles. Ces modèles, une fois définis, constituent des entités statiques : il est nécessaire de leur donner vie, c'est le but de la simulation. L'idée est de doter ces modèles de simulateurs, c'est-à-dire de programmes capables de créer un comportement, au moyen des descriptions contenues dans les modèles. Parmi les formalismes de modélisation, le formalisme DEVS (*Discrete EVent system Specification*) [Zeigler 1976] [Zeigler et al. 2000], un formalisme de bas niveau d'abstraction dédié à la spécification de modèles basés sur les évènements discrets, apparaît comme un des formalismes les plus populaires, de par, notamment, les approches interdisciplinaires qu'il permet, son adaptabilité et sa généricité.

Ce chapitre débute par une section dédiée aux processus d'acquisition et de restitution des connaissances.

La seconde section de ce chapitre est consacrée aux notions de modélisation et de simulation, nous y définissions les notions de modèle, de simulateur, notions que nous lions par la suite au concept de système.

La troisième section s'intéresse aux formalismes de modélisation « classiques » employés en M&S : ils ont pour point commun d'être de bas niveau d'abstraction.

Pour finir, nous présentons dans une quatrième et dernière section le formalisme de modélisation et simulation à évènements discrets DEVS, qui se trouve au cœur de nos travaux.

1.1. Origine et concepts fondateurs de la M&S

Chaque espèce est en proie à une lutte perpétuelle et sans merci avec les éléments et les autres espèces, dans le but de préserver son territoire, s'y acclimater, s'y nourrir, le façonner, s'y reproduire et, ainsi, prospérer.

La seule espèce d'hominidés connue encore présente à ce jour sur Terre, *homo sapiens*, n'échappe pas au mécanisme de lutte qui touche toutes les espèces, mais à ces

besoins primaires vient s'ajouter le besoin irrépressible de comprendre et, avec une créativité sans limites, ponctuée parfois de cuisants échecs, *homo sapiens* utilise au profit mais également au détriment de lui-même les connaissances qu'il amasse, faisant souvent passer les autres espèces et la planète qu'il habite au second plan.

Mu par sa curiosité naturelle, l'homme amasse depuis des millénaires une somme gigantesque de connaissances, les étudie, les confronte, les utilise pour produire d'autres connaissances encore plus précises, et les transmet, de manière orale, à l'aide de dessins, mais aussi, depuis environ 5400 ans, de manière écrite. Une caractéristique propre à l'espèce humaine est donc de chercher à produire et accumuler des connaissances, afin d'acquérir un savoir.

L'anecdote qui suit pourrait prêter à sourire, mais elle est révélatrice de cette nature. En effet, dans la Rome antique, aucun mot latin du langage soutenu n'existait pour exprimer le verbe « savoir » : on utilisait alors le temps parfait (un équivalent du passé) du verbe *noscere* qui signifiait « apprendre à connaître ». Ce mot, qui s'écrivait *novi*, signifiait littéralement : « j'ai appris à connaître », et par extension : « je sais ».

1.1.1. **Acquisition et restitution de connaissances scientifiques**

Nos connaissances scientifiques nous viennent principalement de nos observations et de nos expériences, dont les résultats sont traités par notre cerveau, assimilés, et réutilisés à l'infini. L'observation et l'expérimentation, qui sont des notions complexes à décrire, pouvant être appréhendées selon différents points de vue, accompagnent l'espèce humaine dans son évolution depuis des millénaires, mais aussi l'être humain lors de son développement individuel, depuis l'apprentissage de la vie lors de sa petite enfance jusqu'à sa mort. Observation et expérimentation sont deux notions liées, la première donnant souvent lieu à une volonté de mener la seconde, cette dernière conduisant d'ailleurs à un certain nombre d'observations !

Les données que nous acquérons via l'expérience et l'observation sont ensuite traitées par notre cerveau, lors d'un processus cognitif complexe appelé induction, et sont conceptualisées afin d'être réutilisées. La réciproque de ce processus, la déduction, procédé cognitif tout aussi complexe, consiste à confronter nos connaissances générales à un problème particulier, et d'en tirer des conclusions.

Nous nous efforçons ici de définir ce que sont l'information, la connaissance, l'observation et l'expérience, l'induction et la déduction, tout en nous appuyant sur des travaux divers et pluridisciplinaires menés sur le sujet. Bien définir ces concepts est selon nous essentiel pour aborder par la suite la notion complexe de système, qui elle-même nous ouvre les portes du monde de la modélisation et simulation.

a) ***Information ou connaissance ?***

Il est important de bien distinguer deux termes qui de prime abord peuvent sembler équivalents : information et connaissance. Une information est une donnée reçue telle quelle par notre cerveau (par l'intermédiaire d'un ou de plusieurs de nos cinq sens), c'est une donnée brute, non interprétée. Au contraire, une connaissance est un ensemble d'informations traitées par les hommes [Prax, 2003].

Mais information et connaissance sont intimement liées, car une connaissance mobilise des informations en vue de produire ou d'influencer un raisonnement, une action, un résultat. Le passage de l'information à la connaissance fait donc intervenir des mécanismes de cognition plus ou moins complexes, et la frontière est parfois mince entre ces deux concepts.

b) ***L'observation***

On peut définir l'observation comme l'action de surveiller un phénomène de manière contrôlée, attentive, sans pour autant intervenir sur celui-ci. Initialement limitée par nos cinq sens, et aux phénomènes naturels, elle a au fil des siècles revêtu une nature beaucoup plus complexe, tant par la nature des phénomènes observés, que par les moyens et instruments mis en œuvre pour la mener à bien.

Encore de nos jours, l'observation reste parfois la seule source d'information dont nous disposions pour améliorer nos connaissances scientifiques car certains phénomènes ne permettent pas l'expérimentation, par exemple dans le domaine de l'astronomie, malgré les instruments de plus en plus performants destinés à l'observation du ciel (radiotélescopes…) et en dépit de l'importance des moyens mis en œuvre pour se rapprocher des astres (missions spatiales…). De manière plus large, le fait de collecter les résultats d'une expérience s'assimile également à de l'observation.

c) ***L'expérience***

Etroitement liée à l'observation, l'expérience est une pratique scientifique dont le but est de reproduire, sous certaines conditions et suivant un protocole défini, un phénomène que l'on veut comprendre, voire mettre en évidence, de sorte que n'importe quelle autre personne ou groupe de personnes puisse, en utilisant le même matériel, en disposant du même cadre expérimental, en suivant le même protocole, parvenir aux mêmes résultats (ces résultats expérimentaux n'étant d'ailleurs rien de plus que les produit de nos observations).

Précurseur de la médecine expérimentale, le physiologiste Claude Bernard écrit en 1865 à propos de l'homme [Bernard, 1865] :

« […] pense et veut connaître la signification des phénomènes dont l'observation lui a révélé l'existence. Pour cela il raisonne, compare les faits, les interroge, et, par les réponses qu'il en tire, les contrôle les uns par les autres. C'est ce genre de contrôle, au moyen du raisonnement et des faits, qui constitue, à proprement parler, l'expérience, et c'est le seul procédé que nous ayons pour nous instruire de la nature des choses qui sont en dehors de nous. »

Une expérience scientifique se compose de deux « entités » distinctes et complémentaires :

- **le sujet que l'on étudie** (le mot sujet doit ici être pris au sens large : matériau, organisme, phénomène, capteur ou même système informatique…)

- **le cadre expérimental**, qui détermine le déroulement de l'expérience. Un cadre expérimental peut par exemple définir l'ensemble des expérimentations effectuées, leur durée, les paramètres qui varient, ceux que l'on désire maintenir constants, ainsi que les résultats obtenus (via l'observation) et le moyen d'obtention (mesure…). Il existe deux manières équivalentes de définir ce qu'est le cadre expérimental : la première le considère comme la

définition de tous les éléments, de toutes les données, qui vont alimenter la « base de données » du système (et donc influencer son évolution), la seconde le définit comme un système à part entière, interagissant avec le système étudié, cette interaction produisant des données en fonction des conditions spécifiées

Prenons, pour illustrer ces notions, le cas d'une expérience concrète et simple : la réaction de combustion des vapeurs d'essence dans l'oxygène de l'air, avec pour objectif la recherche du meilleur rendement. Dans ce cas, le sujet de l'expérience est facilement définissable : il s'agit du mélange air-essence et plus précisément le phénomène de combustion de ce dernier. Le cadre expérimental pourra par exemple être fixé par les différentes proportions air-essence testées, ainsi que d'autres paramètres éventuels : pression atmosphérique, température de l'air, nature de la source produisant l'étincelle ou la flamme, nature de l'essence, système de mesure des composants de l'air avant et après la combustion, etc. Le cadre expérimental est donc une série de variations de la même expérience, ces variations s'appliquant aux paramètres cités ci-dessus.

Les observations effectuées permettront d'améliorer nos connaissances sur le sujet. Nous reprenons par la suite l'exemple de l'expérience ci-dessus pour préciser la notion de cadre expérimental.

d) ***Induction, déduction et savoir***

À partir d'observations et d'expériences peuvent être tirés des enseignements plus généraux, via l'émission d'un certain nombre d'hypothèses : ce procédé s'appelle l'induction. Le procédé d'induction fait appel à notre capacité d'abstraction, c'est à dire notre capacité à, en partant de faits concrets, d'informations, les généraliser, les regrouper selon leurs caractéristiques communes, les conceptualiser, afin de simplifier, entre autres, leur représentation et leur manipulation : en d'autres termes, l'induction est l'acquisition du, ou plutôt d'un, savoir. L'expérience présentée ci-dessus nous a permis, par induction, de déterminer les proportions idéales air-essence pour que la combustion offre un rendement optimal dans des conditions de pression et de température données.

Il est également possible de raisonner à partir d'un fait général, d'un savoir, pour le restreindre à un cas particulier, et commencer à en tirer des déductions, toujours à l'aide d'hypothèses. Ce processus s'appelle la déduction. Les déductions obtenues sont ensuite confrontées au phénomène d'origine, lorsque cela est possible, et leur validité pourra être confirmée.

Les rapports entre induction et déduction sont très étroits dans la mesure où l'on produit des déductions (faits, conséquences, nouvelles idées…) à partir de nos connaissances, qui elles-mêmes ont été emmagasinées dans notre savoir en utilisant le processus d'induction.

Un raisonnement inductif erroné (à cause d'une expérience mal définie, d'un cadre expérimental inapproprié) conduira inévitablement à une représentation de la réalité tout aussi erronée. De ce fait, les déductions qui en découleront par la suite seront, elles aussi altérées. Ceci n'est pas forcément négatif, c'est même très courant dans l'histoire des sciences, car cette dissonance peut nous interpeller, et nous amener tout d'abord, par comparaison, à observer que nos déductions sont fausses, et nous encourager ensuite à critiquer notre savoir (et la manière dont nous l'avons acquis). Un autre type de raisonnement, légèrement différent

mais mettant en scène les mêmes procédés, est bien connu des mathématiciens qui l'emploient couramment : le raisonnement par l'absurde.

Si les résultats des observations de l'expérience de combustion air-essence sont suffisamment précis, nous pourrons les utiliser pour déduire par exemple quelle composition air-essence utiliser à 4000 mètres d'altitude, avec une température de -10°, pour obtenir une combustion au rendement acceptable. Nous employons le terme de déduction, car c'est bien ce phénomène qui entre en jeu dans ce cas : il y a en effet de faibles chances pour que les conditions d'altitude (donc de raréfaction de l'oxygène dans l'air), de faiblesse de température et de pression qu'elles impliquent, aient été testées simultanément lors de la phase d'expérimentation (et d'induction, grâce à l'observation des résultats). C'est ici que la mise en place du cadre expérimental revêt tout son sens.

L'induction est donc le mécanisme par lequel nous acquérons nos connaissances, tandis que leur mise en application, leur enseignement, leur transmission, sont assurés par la méthode déductive. Ces concepts sont schématisés dans la figure Figure I-1.

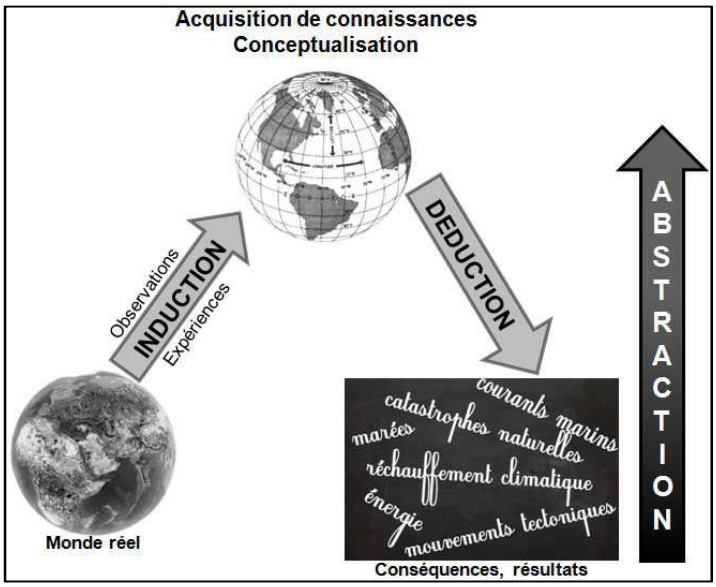


Figure I-1 : Processus d'acquisition et de restitution des connaissances

1.1.2. La notion de système

Nous avons jusqu'à présent employé indifféremment des termes tels que « problème », « sujet/objet de l'expérience », « phénomène » pour désigner, au final, le même concept. Ce dernier, duquel on tire un certain nombre de données, par le biais d'expériences et/ou d'observations, sera à présent désigné par le terme plus approprié de *système*. Le terme système nous vient du mot grec ancien *συστεμα* (« *systema* ») qui signifie « ensemble organisé ». Pour en revenir à l'expérience, nous sommes maintenant en mesure de dire que les résultats obtenus sont presque intégralement conditionnés par deux éléments : le système (objet de l'étude) et le cadre expérimental. Ces deux notions dépassent le cadre de l'expérience concrète elle-même : elles sont également partie intégrante du monde de la modélisation et simulation.

a) ***L'approche systémique***

Différents chercheurs ont travaillé sur la notion de système, notamment au XXème siècle : chacun, selon la discipline dont il était issu, s'est efforcé de définir ce qu'était, selon lui, un système, en mettant en avant telle ou telle caractéristique notable. Leurs travaux constituent différents aspects de ce que l'on appelle communément « l'approche systémique ». Cette approche est fondamentale dans le monde de la modélisation et simulation.

Parmi les chercheurs évoqués ci-dessus, le biologiste autrichien L.Von Bertallanfy a été parmi les premiers à se pencher sur la question des systèmes : il introduit au milieu des années 1930 le concept de « système ouvert » : c'est « un système qui, à travers ses échanges de matière, d'énergie et d'information manifeste la capacité de s'auto-organiser ».

Von Bertallanfy travaille ensuite pendant plus de trois décennies à sa *General System Theory*, ou théorie générale des systèmes. Cette théorie est issue de la recherche pluridisciplinaire, dont la biologie, la cybernétique, au gré de sa collaboration avec d'autres scientifiques. Pour Von Bertallanfy, un système est un « complexe d'éléments en interaction », formant un tout.

Selon lui, un système peut être décrit d'un point de vue comportemental (ou fonctionnel) et structurel.

Du point de point de vue de son comportement, un système comprend :

- des flux d'informations qui empruntent les réseaux de relations et passent par les entrées et les sorties du système,
- des centres de décision qui organisent les réseaux de relations (coordonnent les flux),
- des boucles de rétroaction qui informent les centres de décision, dès l'entrée des flux, sur leur sortie,
- certains ajustements comportementaux, effectués par les centres de décision en fonction des boucles de rétroaction et des délais de réponse.

Structurellement, un système comprend :

- des sous-systèmes, que l'on peut dénombrer et dont on connaît la nature,
- une limite (voir b)),
- des relations entre les différents éléments (réseaux de relations),
- des stocks d'information qui contiennent ce qui doit être transmis ou réceptionné par le système.

On représente parfois un système comme une « boîte noire » (ou *black box*) comportant des entrées et des sorties. On utilise ce type de représentation lorsque l'on ne connaît pas la structure interne d'un système mais qu'il est possible de prédire les sorties en fonction des entrées.

b) ***Limites d'un système***

L'approche systémique admet qu'un système peut également être décrit par sa structure, constituée d'éléments dénombrables et identifiés, et de réseaux de liens entre ces éléments, transportant des informations. En théorie, du fait que le monde, et plus largement l'Univers, peut être considéré comme une imbrication complexe de systèmes, un système peut toujours en contenir d'autres et être contenu par des systèmes plus larges, hormis l'Univers lui-même (qui n'est contenu par aucun autre système) et les quarks (qui sont les plus petits constituants connus de la matière, et supposés ne contenir aucun sous-système…pour le moment). En pratique, et particulièrement en modélisation, nous verrons que l'on donne une limite au système : ceci permet de simplifier de façon notable le modèle, et également sa simulation, en réduisant les paramètres à prendre en compte lors des calculs.

c) ***Entrées et sorties du système***

Un système possède forcément au moins une sortie. Sans cette dernière, il serait en effet impossible de détecter ce système (si toutefois il existait) car il ne pourrait pas communiquer avec son environnement. En revanche, un système peut ne pas posséder d'entrée, ce qui revient à dire que ce système évolue indépendamment de toute perturbation extérieure à lui-même : un tel système est dit autonome. Un générateur de valeurs est un exemple de système autonome couramment rencontré en modélisation et simulation.

d) ***Systèmes dynamiques et complexes***

Un système est qualifié de dynamique lorsqu'il évolue au cours du temps, de manière déterministe et liée à des causes dépendant du présent ou du passé (stimuli extérieurs par exemple).

Un système est dit complexe lorsque son comportement est très difficile voire impossible à prévoir par le calcul manuel. Sa complexité est due en général au grand nombre d'éléments qui le composent et à la nature des relations qui les lient. Lorsque l'on désire étudier un système complexe, on procède en deux étapes, centrales en M&S :

- on identifie ses composants et leur comportement (règles d'évolution), puis on modélise le système ;
- on effectue une simulation afin d'obtenir des résultats.

1.1.3. Fondements et principes de la modélisation

L'expérimentation permet de collecter des données, d'observer des comportements, mais elle se heurte à ses propres limites quand il s'agit de *décrire* le système lui-même.

Les scientifiques ont donc naturellement tendance à essayer de s'abstraire de la réalité dans le but de créer une représentation simplifiée d'un système, manipulable à volonté, même une fois l'expérience terminée. Une telle représentation s'appelle un *modèle* et le processus mental par lequel il est créé s'appelle la *modélisation*. La description d'un modèle se fait grâce à un *formalisme de modélisation* : c'est un langage formel, auquel s'ajoute une sémantique particulière, utilisé pour modéliser des systèmes. Nous serons confrontés, tout au long de ce mémoire, à plusieurs formalismes de modélisation différents.

Dans tous les cas, un formalisme de modélisation doit être adapté au contexte scientifique, au domaine d'étude et aux connaissances des utilisateurs. Il est par exemple relativement inapproprié de décrire une recette de cuisine grâce à des équations différentielles.

Certains formalismes de modélisation sont aptes à représenter un système sous un aspect uniquement comportemental, tandis que d'autres combinent les aspects comportementaux et structurels.

Revenons un instant à l'approche systémique vue en 1.1.2.a), pour dire qu'actuellement encore, elle est très étroitement liée à la majorité des processus de modélisation. Elle constitue la manière privilégiée d'appréhender un système afin d'en réaliser des modèles pertinents, et elle a inspiré la logique qui a conduit à l'élaboration de plusieurs formalismes de modélisation.

Étudiée dans cette partie en fonction de son rapport avec le système et le simulateur, la notion de modèle le sera encore de manière récurrente dans la suite de ce travail, et ce sous différents angles : elle sera tout d'abord approfondie en 1.2 lorsque nous aborderons en détail les formalismes de modélisation, puis en 1.3 avec l'étude du formalisme DEVS, et enfin lorsque nous présenterons les grands principes fondateurs de l'Ingénierie Dirigée par les Modèles (IDM) dans le chapitre II. Par conséquent, le lien que nous nous proposons de tisser entre le monde de la modélisation et simulation d'une part, et le monde de l'IDM d'autre part, sera lui-même centré sur la notion fondamentale de modèle.

Attardons-nous pour le moment sur les raisons qui nous conduisent à devoir utiliser des modèles, et tentons de donner quelques éclaircissements sur les notions fondamentales de modèle et de modélisation.

1.1.4. **Limites de l'expérimentation**

Avec un minimum de moyens et de prudence, il est possible de mener à bien l'expérience de combustion du mélange air-essence. Cependant, en pratique, les choses ne sont pas aussi simples, et la majorité des problèmes que nous étudions sont bien plus complexes que la combustion des vapeurs d'essence dans l'oxygène de l'air.

En effet, plusieurs raisons peuvent s'opposer à la mise en place d'expériences, les plus courantes étant liées au coût, à la dangerosité, à l'éthique, au fait que l'objet de l'étude est inaccessible de par sa distance (par exemple, comme nous l'avons mentionné plus haut, l'astronomie permet l'observation mais pas l'expérimentation, étant donné qu'il est plutôt difficile de manipuler un objet céleste !), inaccessible du fait qu'il n'existe pour l'instant que virtuellement (architecture, industrie…), ou enfin inaccessible de par sa taille, très éloignée de l'échelle humaine (les plaques tectoniques sont en théorie accessibles, par contre nous sommes dans l'impossibilité d'influencer leur mouvement).

Pourtant, ces expériences seraient nécessaires pour combler notre soif de savoir, et améliorer nos connaissances.

Mais, de quelle manière nos connaissances nouvellement acquises (via un procédé inductif…) sur ce phénomène chimique sont-elles organisées ? Certes, nos expériences nous ont démontré que cette réaction ne se produit que sous certaines conditions optimales, c'est-à-

dire lorsque la proportion air-essence se situe aux alentours de 15:1, soit 1g d'essence pour 15g d'air, mais comment transmettre cette connaissance de manière précise ?

Nous pouvons répondre à ces deux questions simultanément. Pour cela, nous avons besoin de décrire le système, tout en faisant abstraction d'un certain nombre de paramètres du protocole expérimental que nous jugeons négligeables (température de la pièce, nature et forme des récipients utilisés, nature de la source de chaleur provoquant la combustion du mélange…). Nous venons d'exprimer le besoin de recourir à un modèle.

1.1.5. **Le modèle, une représentation du système**

Des techniques permettant de contourner les obstacles qui s'opposent à l'expérimentation ont été développées au cours des siècles : l'une d'entre elles consiste à faire appel à notre capacité d'abstraction pour s'éloigner du problème lui-même (et des contraintes matérielles qui rendent impossibles des expériences sur celui-ci), afin de raisonner sur une version simplifiée de ce problème. Cette technique fondamentale dans le monde scientifique en général l'est aussi en informatique : il s'agit de la modélisation (i.e. le fait de créer un modèle).

a) ***Notion de modèle***

Le célèbre scientifique américain Marvin Lee Minsky, co-fondateur du MIT a proposé une définition très simple de ce qu'est un modèle [Minsky, 1968] :

"To an observer B, an object A is a model of an object A to the extent that B can use A* to answer questions that interest him about A"*

(«Pour un observateur B, un objet A* est un modèle d'un objet A dans la mesure où B peut utiliser A* pour répondre aux questions qu'il se pose sur A»).

Revenons sur l'exemple auquel nous faisons référence depuis le début de ce chapitre : le système que constitue la combustion de l'essence dans l'air. Nos expériences nous ont permis de définir que la combustion de l'essence était pratiquement totale et dégageait le plus d'énergie lorsque le mélange air-essence était d'environ 15 :1.

Simplifions ce système :

- dans la réalité, ce que nous désignons par « essence » est en fait un mélange d'hydrocarbures, notamment des alcanes et des alcènes. Considérons pour notre part que l'on peut assimiler ce mélange à de l'octane pur, de formule chimique C_8H_{18} ;
- considérons également que cette essence est entièrement vaporisée dans l'air du récipient dans lequel nous la faisons brûler : le mélange air-essence est homogène. Dans ce cas, la combustion est considérée comme totale. Nous négligeons donc les produits de la réaction autres que H_2O_2 (eau, sous forme de vapeur) et CO_2 (dioxyde de carbone) ;
- considérons enfin que la proportion de dioxygène dans l'air est constante, aux alentours de 20% : nous négligeons l'altitude et la pression atmosphérique, nous considérons l'air qui va être utilisé comme « pur » et ne contenant aucune particule, ni vapeur d'eau.

Nous voici maintenant en mesure de décrire ce système par une équation-bilan classique :

$C_8H_{18} + O_2 => CO_2 + H_2O$, sa forme équilibrée étant :

$$\mathbf{1}\ C_8H_{18} + \mathbf{12,5}\ O_2 => \mathbf{8}\ CO_2 + \mathbf{9}\ H_2O$$

Un rapide calcul en combinant cette équation-bilan avec les masses molaires des réactifs (octane : 114g, air : 1716g) nous donne un mélange optimal à 15,1 :1 ce qui est conforme à nos résultats expérimentaux.

b) ***Pluralité des modèles***

Nous venons d'extraire un modèle de notre système : c'est la modélisation. Ce modèle, statique, nous permet de transmettre notre connaissance de ce phénomène chimique, en particulier des proportions à utiliser, tout en négligeant certains aspects liés au cadre expérimental (essence non pas constituée uniquement d'octane mais par un mélange d'hydrocarbures, non-homogénéité du mélange air-essence due à la difficulté de vaporiser l'essence, air jamais vraiment pur, proportions de dioxygène dans l'air pouvant varier selon le temps, le lieu). D'autres modèles de ce système pourraient nous renseigner sur la vitesse flamme, le temps de combustion…etc.

En fait, pour un même système, plusieurs modèles sont possibles. De plus, il est fréquent qu'un système complexe soit décrit par non pas un seul, mais plusieurs modèles complémentaires, pouvant être de natures diverses.

Les architectes utilisent fréquemment deux types de modèles dans leur travail. Les premiers, les plans, sont des modèles bidimensionnels qui illustrent les propriétés remarquables de la construction (distances, angles), et seront réalisés à une certaine échelle (conservation des proportions, détails, mais taille réduite). En revanche, les seconds, les maquettes, construites, tout comme le plan, à échelle réduite, permettent cette fois-ci d'offrir une vue tridimensionnelle de l'édifice, d'insister sur l'impact visuel d'un point de vue extérieur.

Un autre exemple de modèles complémentaires peut être apporté avec le génie logiciel et l'outil, ou plutôt la boîte à outils, UML 2.4.1[1] [Rumbaugh et al. 2005]. Cette dernière propose en effet une panoplie de 13 diagrammes statiques et dynamiques, répartis en 5 grandes familles de vues. Ces diagrammes se complètent, et leur utilisation est laissée au modeleur, auquel incombe ainsi le choix de sélectionner quel(s) diagramme(s) il utilisera pour décrire le système. Par exemple, un système « distributeur de billets » pourra être modélisé grâce à :

- un diagramme des cas d'utilisation (diagramme comportemental), décrivant à un haut niveau d'abstraction les acteurs amenés à intervenir sur le distributeur : clients, convoyeurs de fonds, techniciens…etc ;
- un diagramme de classes décrivant l'architecture logicielle contrôlant le système, et un diagramme d'objets représentant les instances de ces classes (diagrammes structurels) ;

[1] http://www.omg.org/spec/UML/2.4.1

- des diagrammes de séquence (diagramme dynamique) montrant les interactions : entre les acteurs et les éléments du système, et entre les éléments du système lui-même.

En guise de dernier exemple, considérons un modèle DEVS quelconque (nous présentons exhaustivement ce formalisme en 1.3) : ce dernier sera tout d'abord exprimé de manière informelle, écrite, en se basant sur le formalisme de base et, au besoin, en enrichissant le modèle avec des algorithmes écrits en pseudo-langage. Mais, afin d'être simulé, ce modèle devra être intégralement codé dans un langage de programmation, ce qui constitue de ce fait un modèle différent : écrit dans un langage de programmation, respectant les contraintes de ce langage, mais écrit aussi selon les spécifications de DEVS. Ces modèles sont complémentaires : le premier a plus de chances d'être compréhensible au bout de quelques secondes par une tierce personne connaissant DEVS que le second, alourdi par les instructions propres au langage de programmation utilisé.

À la lumière des quatre exemples précédents, nous pouvons conclure que :

- décrire un système à l'aide d'un modèle (le modéliser) présente comme principal avantage la conservation des propriétés considérées comme importantes du système étudié, tout en faisant l'impasse sur d'autres paramètres considérés comme secondaires : cela permet notamment de contourner les contraintes pratiques qui pourraient gêner la compréhension ;
- un modèle est lui-même toujours décrit dans un formalisme de modélisation, plus ou moins complexe, pouvant être « implicite » (basé sur des connaissances que l'on suppose partagées par le plus grand nombre) et pas vraiment « défini » (recette de cuisine, croquis…) ou au contraire reposant sur des bases solides, définies formellement (formalisme DEVS, UML, langages de programmation…) ;
- un formalisme de modélisation est lui-même décrit dans un formalisme, plus exactement un méta-formalisme : le chapitre II précise cette imbrication entre modèle, formalisme et méta-formalisme en introduisant des notions fondamentales comme la méta-modélisation.

Il est toutefois important d'insister une nouvelle fois sur le fait qu'à partir d'un même système peuvent être (et sont souvent) tirés un ou plusieurs modèles de nature et de complexité différentes en fonction de la ou des caractéristiques du système qui nous intéressent, du formalisme de modélisation choisi, et même de l'appréciation personnelle des scientifiques qui établissent ces modèles.

c) *La hiérarchie dans les modèes*

Certains formalismes de modélisation autorisent la description hiérarchique de modèles, ce qui permet d'introduire dans ces derniers la notion de hiérarche de description d'abstraction. Le terme « hiérarchie de modèles » indique donc une structure composée de modèles imbriqués et/ou interconnectés avec d'autres modèles (pouvant en contenir d'autres, et/ou être contenus par d'autres). Disposer d'une hiérarchie de modèles implique que le simulateur associé au formalisme soit en mesure de prendre en compte l'aspect structurel des modèles qu'il va simuler.

d) ***Les variables dans les modèles***

Avant d'aborder la sous-section dédiée à la simulation, intéressons-nous aux différents types de variables qui peuvent caractériser les modèles (nous verrons plus tard qu'elles sont étroitement liées à la notion d'état) et donc être gérées par les simulateurs.

Ces variables correspondent généralement aux types de base usuels ou à des types représentant des structures plus complexes (listes, tableaux…). Ces types de base ou complexes sont manipulés par les ordinateurs via les langages de programmation.

Dans tous les cas, les variables peuvent être qualifiées de discrètes ou continues, de qualitatives ou quantitatives (ces deux dernières notions étant directement empruntées à la statistique).

Les variables ne représentant pas une quantité sont qualifiées de qualitatives. Deux variables qualitatives ne peuvent être comparées, sauf pour vérifier qu'elles sont égales ou différentes. Les variables qualitatives peuvent être par exemple des chaînes de caractères (noms), des suites de nombres (numéro de téléphone)…etc.

Inversement, sont qualifiées de quantitatives des variables numériques, issues de mesures, exprimées dans une unité, pouvant être comparées entre elles (supériorité, infériorité, égalité). Une variable possédant une valeur numérique ne représente donc pas forcément une donnée quantitative.

Soit un modèle décrivant par exemple les 4 temps d'un moteur à explosion. Un tel modèle décrit un processus connu, comportant des étapes dénombrables, les valeurs possibles de la variable V_1 (de type *chaîne de caractères*) associée à ce modèle seront des mots choisis dans l'ensemble $S_1 = \{admission ; compression ; explosion ; échappement\}$. Ici, $Card(S_1) = 4$ et V_1 est une variable qualitative. L'ensemble désigné par une variable qualitative possède toujours une cardinalité finie.

Soit à présent un modèle comptant les heures d'une horloge classique, donc effectuant une opération *modulo 12*. La variable V_2 (de type *entier*) associé à ce modèle prendra ses valeurs entières dans $S_2 = \{0…11\}$. Ici, $Card(S_2) = 12$ et V_2 est une variable cette fois-ci quantitative, les heures pouvant être comparées entre elles, et reflétant une grandeur : le temps (divisé en heures).

Enfin, soit un modèle décrivant le remplissage d'un réservoir de 50 litres d'essence : la variable V_3 (de type *réel*) associée prendra ses valeurs réelles dans $S_3 = \{0…50\}$.

Ici, $Card(S_3) = \infty$. V_3 est également une variable quantitative, issue d'une mesure et indiquant une grandeur.

Les deux derniers exemples montrent toutefois des variables quantitatives de types différents : V_2 est qualifiée de discrète, car elle prend ses valeurs dans un ensemble dénombrable et possède une valeur finie. En revanche, V_3 est dite continue, elle peut théoriquement prendre une infinité de valeurs, au fur et à mesure de l'évolution du temps (lui-même continu) lesquelles forment un ensemble continu. Pour conclure, ajoutons que dans le cas d'une variable qualitative, il serait absurde, de par la nature même de cette variable, de la qualifier de discrète ou de continue.

e) ***La notion d'état et de dimension***

Le mot « état » sera abondamment employé dans ce document, car il constitue l'élément fondamental d'un système, et donc du ou des modèles qui s'y rapportent (i.e. le décrivent). Un modèle peut être décrit par une seule variable, comme le moteur à explosion ci-dessus : l'état du modèle est donc caractérisé par la valeur V_1 à un instant donné. V_1 est une variable d'état.

À travers cet énoncé se dessine une définition simple de l'état : c'est l'ensemble des variables (en dehors des entres et des sorties) qu'il faut connaître pour pouvoir caractériser un système. Le fait d'avoir connaissance à un instant *t* de ces variables permet de prévoir l'état du système à l'instant *t+1*.

Le modèle de moteur possède une seule variable d'état, son état est donc unidimensionnel. En revanche, si on ajoutait une variable quantitative V_p à ce modèle de moteur, représentant la pression moyenne à l'intérieur des cylindres à un instant donné, on aurait besoin, pour caractériser ce modèle à un instant donné, de connaître V_1 et V_p. Nous venons d'ajouter une dimension à l'état. L'intérêt de modèles multidimensionnels est qu'ils permettent une description très précise des systèmes, au prix toutefois d'une perte de simplicité, notamment dans leur représentation graphique.

Souvent, lorsqu'un système possède un état unidimensionnel, décrit par une variable qualitative, on dit que ses états sont représentés « en extension », c'est-à-dire énumérés et nommés explicitement dans la description du système. Par exemple, des formalismes comme les automates à états finis (1.2.2.b)) permettent seulement la spécification d'états unidimensionnels, en extension.

f) ***Techniques et processus de modélisation***

Sans déborder sur les formalismes de modélisation, qui seront largement étudiés plus loin dans ce document, il est néanmoins intéressant d'élargir quelque peu nos propos pour se faire une idée des différentes techniques et processus utilisés en modélisation. Le lecteur intéressé par ces techniques et ces processus pourra consulter l'annexe 1 de ce manuscrit.

Disposer d'un modèle est certes utile, mais on a souvent besoin d'étudier son comportement : la simulation nous fournit le cadre nécessaire à une telle étude.

1.1.6. Simulation de modèles

Le fait de produire une évolution possible d'un système (plus exactement, du ou des modèle(s) de ce système) au cours du temps, sous certaines conditions, et de générer des résultats, s'appelle la simulation. En d'autres termes, l'intérêt de la simulation est de chercher à reproduire le comportement du système, en utilisant des modèles.

Autrefois réalisée manuellement (et donc nécessairement avec un nombre fini de valeurs et de données), souvent de manière empirique, elle est aujourd'hui traitée avec l'outil informatique, qui offre des possibilités gigantesques dont les seules limites, elles-mêmes sans cesse repoussées, se trouvent dans la nature du matériel utilisé (système d'exploitation utilisé, fréquence d'horloge du ou des processeurs, mémoire vive disponible, mémoire graphique, qualité du réseau dans le cadre d'une simulation distribuée…) et dans la nature des algorithmes de simulation. La simulation permet donc à moindres frais de mener des

observations sur le comportement d'un système, recréé artificiellement, plus ou moins conforme à la réalité selon la nature des modèles. Elle conduit à la production de données, d'observations, qui serviront à nous renseigner sur le système étudié, voire à prévoir certaines de ses évolutions.

a) ***Le simulateur, « moteur » des modèles***

Le simulateur est l'ensemble des structures matérielles et logicielles (algorithmes notamment) qui contrôlent le processus de simulation, son rôle est de faire évoluer le modèle dans le temps. On peut dire en ce sens qu'il est l'équivalent du cadre expérimental puisqu'il a le pouvoir de faire varier des paramètres, de simuler des stimuli sur le modèle, ce qui impacte directement le comportement de ce dernier, et qu'il fournit, au cours et/ou au terme du processus de simulation, un ensemble de résultats.

C'est donc l'adjonction d'un simulateur aux modèles qui leur confère un intérêt certain : ceux-ci ne constituent plus seulement des descriptions (comportementales, structurelles…) d'un système mais sont de surcroît des composants pouvant être utilisés, dynamiquement, par un simulateur. Le simulateur a forcément « connaissance » du ou des modèles qu'il doit simuler. C'est pourquoi la simulation est indissociable de la modélisation.

b) ***Le temps de la simulation***

Faire évoluer un ou des modèles au cours du temps implique que l'on définisse ce qu'est *le temps*. Ce dernier peut être abordé de deux manières : soit on considère qu'il s'écoule de la même manière que le temps du monde réel, on l'appelle alors *temps réel*, soit on considère que cet écoulement est différent (plus rapide ou plus lent) et on l'appelle *temps de simulation*. Nous nous intéresserons ici uniquement au temps de simulation. Ce dernier peut être considéré comme évoluant de manière continue, ou discrète.

Le temps de simulation continu (Figure I-2) est utilisé pour simuler des systèmes dont l'évolution (changements d'états) dans le temps se fait continuellement, par exemple un système décrit par une équation différentielle linéaire.

Cependant, utiliser un temps de simulation continu n'est possible que théoriquement, étant donné que les machines sur lesquelles se déroulent la simulation sont toutes numériques et non pas analogiques. Le temps est échantillonné plusieurs milliards de fois par seconde (en fonction du processeur utilisé) et ne peut pas être « capturé » dans son intégralité.

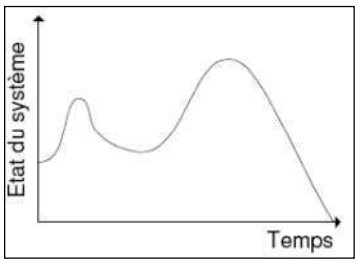


Figure I-2 : Temps de simulation continu

C'est exactement la même chose qui se produit lorsque l'on cherche à calculer de manière approximative l'intégrale d'une fonction par la méthode des rectangles (le pas correspondant à l'échantillonnage).

Le temps de simulation discret peut être géré soit par l'horloge (synchrone), soit par des évènements (asynchrone). Il est utilisé lorsque l'évolution du système se fait à des instants précis dans le temps, ces instants étant dénombrables.

Dans le premier cas (figure Figure I-3), la simulation évolue de manière constante et le temps est incrémenté régulièrement d'un pas Δt. Les calculs inhérents à des évènements se produisant sur l'intervalle]t ; t+Δt] ne sont effectués qu'à la date t+Δt. Ce type de simulation est particulièrement adapté aux systèmes synchrones (par exemple, tout ce qui concerne les systèmes séquentiels).

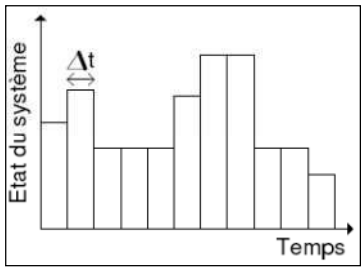


Figure I-3 : Temps de simulation discret dirigé par l'horloge

En effet, ces derniers ne voient leur état modifié qu'à des instants précis déterminés par les signaux d'horloge. Ce sont d'ailleurs ces signaux d'horloge, ou « tops d'horloge », du processeur qui vont fixer le pas de temps minimal Δt que l'on peut utiliser.

Dans le second cas (Figure I-4), la simulation évolue en fonctions des évènements se produisant dans le système : le pas de temps n'est pas fixe et peut varier entre chaque date d'évènement. Le système n'évolue non plus selon une horloge mais selon un calendrier, que l'on appelle souvent échéancier, comprenant tous les évènements qui se produiront au cours de la simulation, et sans cesse remis à jour au cours de celle-ci. Une simulation en fonction des évènements est une simulation de type asynchrone, très utilisée pour simuler des modèles à évènements discrets. C'est elle que nous désignons dans la suite de ce document par le terme simulation. C'est aussi elle qui va de pair avec les modèles à évènements discrets.

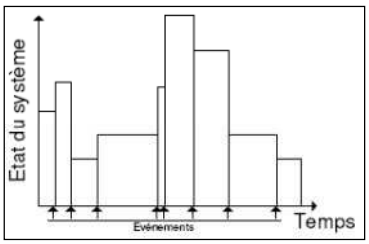


Figure I-4 : Temps de simulation discret dirigé par les évènements

Modèles à évènements discrets et simulation asynchrone dirigée par les évènements constituent le socle du type de modélisation et simulation que nous étudions dans ce document.

Remarque : Il existe aussi des systèmes hybrides : ces derniers sont des systèmes composés de modèles dont le comportement évolue, pour certains, de manière discrète, et, pour d'autres, continue.

1.1.7. Vérification et validation

Une fois la simulation effectuée, il est nécessaire de confronter les résultats obtenus aux résultats attendus et donc aux résultats issus d'observations ou d'expériences. Mais on peut aussi se rendre compte, bien avant la phase de simulation, qu'il existe des incohérences au niveau du modèle. Ces vérifications s'inscrivent dans un processus appelé processus de vérification et de validation. Bien qu'on le situe souvent, dans un souci de simplification, à la fin de la simulation, ce processus existe, de manière parfois implicite, dans toutes les phases du cycle de vie du modèle (par exemple, au moment où l'on remarque les incohérences citées ci-dessus).

Son but est de valider, en se plaçant à plusieurs niveaux d'abstraction, le modèle et le simulateur qui lui est associé : globalement, on doit valider le modèle en s'assurant qu'il remplit la définition énoncée en 1.1.5 et vérifier que le simulateur génère un comportement cohérent de ce modèle. Dans cette sous-section, nous analysons tout d'abord les concepts de validation et de vérification, puis nous montrons comment ils s'articulent en pratique. Un modèle est dit valide lorsque le système et le modèle se confondent dans le cadre expérimental : en d'autres termes, lorsqu'un modèle fournit une description correcte du monde réel, et que son comportement correspond aux spécifications du système étudié, il est valide.

La vérification désigne, quant à elle, le fait de s'assurer que le simulateur produit un comportement conforme à celui décrit dans le modèle.

Le lecteur qui s'intéresse à la vérification et validation pourra consulter l'annexe 2 de ce document qui décrit ces dernières de manière plus approfondie.

1.1.8. Synthèse

Précisons les liens évidents qui se dessinent désormais entre les différentes notions abordées depuis 1.1.2.

L'étude de tout système, qu'elle se fasse dans le monde réel ou dans un environnement approprié (laboratoire) conduit à l'observation d'un comportement de ce dernier, qui génère des données expérimentales. La modélisation de ce système conduit à l'élaboration d'un ou de plusieurs modèles qui sont des vues simplifiées du système. Dans le cas où plusieurs modèles complémentaires cohabitent, ils peuvent être situés à différents niveaux d'abstraction et/ou appartenir à différentes vues et sont interconnectés.

La simulation consiste à faire évoluer ces modèles en fonction du temps, et, dans le cas qui nous intéresse, du temps de simulation dirigé par les évènements, en utilisant un simulateur (à l'aide de l'outil informatique). Cette simulation conduit également à l'observation d'un comportement des modèles de ce système, ainsi qu'à la collecte des données générées.

Une fois ce comportement et ces données acquises, le scientifique doit procéder à la validation de ses modèles et à la vérification du simulateur. Pour ce faire, il compare les résultats de la simulation au système original : si le comportement des modèles durant la simulation et les données générées sont conformes à ses attentes, il considère les modèles

comme valides. Mais, bien souvent, du fait de la perte d'information durant le processus de modélisation, ces modèles doivent être modifiés pour être encore plus conformes à la réalité.

Modélisation, simulation et validation sont étroitement liées dans un cycle de vie représenté dans la figure Figure I-5.

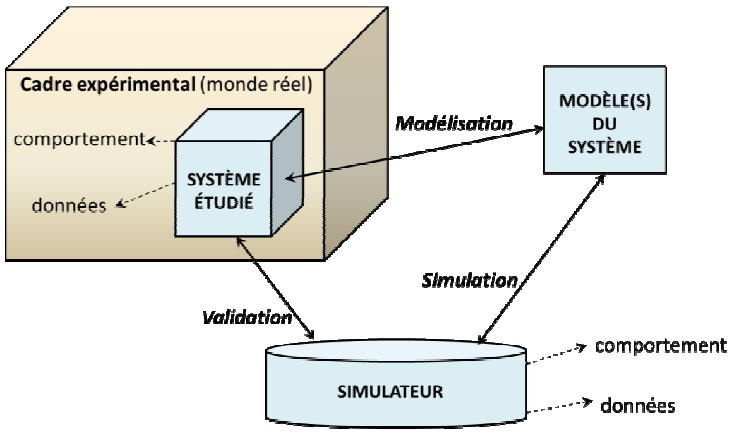


Figure I-5 : Liens entre système, modèle et simulateur

1.2. Formalismes de modélisation classiques

Un modèle est, nous l'avons vu, toujours décrit dans un formalisme, ou langage de modélisation. Ces formalismes sont très nombreux à cohabiter dans le monde scientifique, et en particulier dans celui de la modélisation et simulation. Qu'est-ce qu'un formalisme, et à quoi ressemble-t-il ?

Le but de cette partie est de répondre à cette question, en introduisant notamment les notions fondamentales de langue, langage, syntaxe, sémantique, et en présentant certains formalismes couramment utilisés en modélisation et simulation. Pour ce faire, nous nous inspirons entre autres de l'état de l'art de [Agrawal, 2004].

Nous nous restreignons à certains formalismes considérés comme des « classiques » et employés en M&S, et ce pour une double raison : restreindre le cadre de notre étude au domaine de la M&S, et mettre en lumière le fait que certains formalismes de modélisation, du fait de leur haut niveau d'abstraction, sont aussi très proches de ceux qui sont utilisés par l'IDM. Ils se situent à la frontière commune des formalismes employés à la fois en M&S et en génie logiciel. C'est le cas du formalisme Statecharts [Harel 1987], formalisme assez ancien mais désormais faisant partie intégrante d'UML (diagramme de « machines d'état »).

1.2.1. Notions de base

a) *Langue ou langage ?*

Dans les sciences de la communication, le langage et les langues sont deux choses bien distinctes : le langage désigne une faculté, un ensemble d'aptitudes qui forment une fonction d'expression de la pensée et sa mise en œuvre au moyen de signes vocaux (parole) et éventuellement de signes graphiques (écriture), et ne s'emploie qu'au singulier. La langue, quant à elle, désigne la façon de s'exprimer par le langage. En d'autres termes, lorsque le langage est parlé ou écrit, il devient une langue.

Le mot *langage*, par extension, s'emploie également au pluriel pour désigner des langages de programmation, dans un contexte informatique. Nous utiliserons par la suite le terme de *langages* pour désigner des mots, caractères, symboles, signes, qui, en respectant une syntaxe et une sémantique bien précises, se combinent pour former un ensemble interprétable par une machine. Cette combinaison décrit une suite d'instructions que la machine doit effectuer dans le but de produire un résultat (après interprétation et/ou compilation du code).

Dans le cas d'un langage de modélisation, on distingue deux types différents : les langages graphiques (diagrammes par exemple) et les langages textuels. Certains langages sont une combinaison de ces deux types. Le but d'un langage de modélisation est de décrire les concepts, les composants, la structure d'un système.

b) ***Syntaxe***

La syntaxe est une partie de la grammaire qui a pour but de fixer des règles permettant d'énoncer correctement une langue, sans pour autant se préoccuper du sens. La syntaxe étudie aussi la manière dont des mots se combinent pour former des propositions, des énoncés. En informatique, la notion de syntaxe est rattachée à la définition de séquences de symboles valides, et donc au respect ou au non-respect d'une grammaire d'un langage, toujours sans se préoccuper du « sens ».

Il existe deux types de syntaxe : la syntaxe concrète et la syntaxe abstraite. Dans un contexte de programmation, la syntaxe concrète décrit précisément comment nous devons placer les caractères, les symboles, la ponctuation, dans une expression ; la syntaxe abstraite s'attache davantage à la définition d'une structure correcte du langage, par exemple en représentant une expression à l'aide d'un arbre de syntaxe.

On voit bien que plusieurs langages peuvent ainsi avoir la même syntaxe abstraite alors que leurs syntaxes concrètes sont différentes : une affectation a=2 en C, et une affectation a :=2 en Maple sont différentes au niveau de la syntaxe concrète mais rigoureusement identiques au niveau de la syntaxe abstraite.

c) ***Sémantique***

Contrairement à la syntaxe, la sémantique comprend tout ce qui est relatif à l'étude du sens, suite à une combinaison de mots. Informatiquement parlant, elle est le résultat de l'évaluation des expressions (correctes) d'un langage.

Lors de la conception d'un langage de programmation, on utilise trois types de sémantiques de niveaux d'abstraction différents : opérationnelle (bas niveau), axiomatique (niveau moyen), dénotationnelle (haut niveau).

Le but de la sémantique opérationnelle est de donner une signification à un langage, de manière formelle, en s'appuyant sur un cadre mathématique. La sémantique axiomatique, quant à elle, est une approximation, une abstraction, de la sémantique opérationnelle. Cette sémantique fait appel à la logique des prédicats pour fournir un ensemble de preuves sur la validité d'un programme. La sémantique axiomatique s'appuie essentiellement sur la logique de Hoare [Hoare, 1968] et utilise des règles d'inférence, des pré-conditions et des post-conditions. La sémantique dénotationnelle, de plus haut niveau, peut être utilisée comme

point de départ lors de la conception d'un langage. Elle fait appel à une branche spécifique des mathématiques, la théorie des domaines, forme particulière de la théorie des ensembles. Cette théorie a été élaborée par Dana Scott et Christopher Strachey [Stoy, 1977] dans le but de fournit un cadre formel au λ-calcul. Elle est donc basée sur la programmation fonctionnelle.

d) ***Syntaxe et sémantique en M&S***

Dans un contexte de modélisation et simulation, on peut dire que la syntaxe abstraite d'un formalisme décrit les éléments, les concepts, ainsi que leurs liens (nous verrons dans le prochain chapitre que c'est exactement le rôle d'un *méta-modèle*), tandis que la syntaxe concrète peut prendre la forme d'un formalisme graphique correspondant aux éléments de la syntaxe abstraite, et régissant leur aspect visuel. La sémantique, quant à elle, s'emploie tout d'abord de manière statique pour enrichir les formalismes de description de modèles (sémantique statique), mais aussi de manière dynamique pour définir des règles d'exécution : dans ce cas, elle peut agir directement sur l'état d'un système (sémantique d'exécution).

1.2.2. La grande famille des MOC

Une des manières les plus évidentes de classifier des langages de modélisation est de les « trier » selon les détails d'un système qu'ils sont capables de représenter, et en d'autres termes le niveau d'abstraction auquel ils se situent. Schématiquement, on va distinguer les formalismes *high-level* des formalismes *low-level*. Un formalisme de type *low-level* est également appelé un *model of computation* que nous abrégeons par le terme MOC. Bien que nous puissions le traduire par « modèle de calcul » nous préférons garder le terme anglo-saxon, car il rappelle la racine du mot *computer*. Un MOC se définit comme :

> *« Une définition formelle, abstraite, d'un ordinateur. En utilisant un tel modèle, on peut analyser le temps d'exécution intrinsèque ou l'espace occupé en mémoire d'un algorithme, tout en ignorant de nombreux problèmes liés à l'implémentation. Il existe beaucoup de MOC, qui diffèrent par leur puissance de calcul (c'est-à-dire que certains modèles pourront effectuer des calculs alors que cela est impossible pour d'autres modèles) et par le coût algorithmique de leurs diverses opérations »* (définition du National Institute of Standards and Technology[1], traduction personnelle)

Un MOC constitue donc l'abstraction, indépendante de toute plateforme, d'un ordinateur, et plus précisément du composant de ce dernier qui effectue les calculs : le microprocesseur. Un MOC peut aussi décrire d'autres types de calculs ou de processus comme nous allons le voir ci-après. Nous donnons ici, de manière chronologique, un bref aperçu des MOC les plus utilisés.

a) ***Machine de Turing***

Le plus ancien type de MOC formellement défini est certainement la machine de Turing [Turing 1936]. Une machine de Turing est une représentation abstraite d'un dispositif de calcul. Elle est constituée par :

---

[1] http://xlinux.nist.gov/dads/HTML/modelOfComputation.html

- une bande (ou ruban) unidimensionnelle, supposée infinie, divisée en cases contenant chacune un symbole (les symboles appartiennent, eux, à un ensemble fini),
- une tête de lecture/écriture qui parcourt de manière bidirectionnelle (soit à gauche, soit à droite) et séquentielle (ne peut pas sauter de cases). En d'autres termes, elle peut lire et modifier le caractère courant, et se déplacer d'une case à gauche ou à droite,
- un état interne qui garde en mémoire l'état courant de la machine. Avant que la machine ne démarre le calcul, cet état interne a pour valeur l'état initial,
- une table d'instructions, associant pour chaque couple état interne/caractère lu un triplet nouvel état / nouveau caractère / déplacement (cette table est en fait une table de transitions).

b) ***Automates à états finis***

Note : Les automates que nous présentons dans nos schémas ont été dessinés à l'aide de l'environnement en ligne Finite State Machine Designer[1]

Les automates ou machines à états finis (*Finite State Machine* ou FSM) [Glushkov 1961] [Hopcroft et al. 1976] sont certainement les plus connus des MOC, et ils portent en eux des concepts que l'on va retrouver dans tous les autres types de MOC, dont le formalisme DEVS. Par conséquent nous pensons qu'il est nécessaire de les développer ici.

Les FSM sont largement utilisés dans la modélisation des protocoles, des processus, mais aussi en compilation, voire en linguistique car ils permettent notamment de décrire les grammaires régulières se rapportant aux langages rationnels. Un automate à états finis A est décrit par le quintuplet suivant :

$$A = < S, \Sigma, \delta, I, F >$$

Avec :

- S est un ensemble fini d'états
- Σ est un alphabet fini (et ε en est le mot vide)
- δ est l'ensemble des transitions : $\delta \subseteq (Q \times (\Sigma \cup \{\varepsilon\}) \times Q)$
- I est l'ensemble des états initiaux : $I \subseteq Q$
- F est l'ensemble des états finaux : $T \subseteq Q$ (on parle aussi d'états terminaux, ou d'états acceptants)

Les deux notions importantes ici sont les notions d'*état* et de *transition*, car elles régissent tous les (nombreux) formalismes dérivés des FSM. L'état peut être défini par la « situation dans laquelle se trouve, à un moment particulier, un dispositif matériel ou logiciel » (définition Grand Dictionnaire Terminologique[2]). Les états possibles d'un

[1] http://madebyevan.com/fsm/
[2] www.granddictionnaire.com

interrupteur sont par exemple 0 (éteint) ou 1 (allumé). Une transition désigne le passage d'un état à un autre état, soumis à condition (on appelle souvent cette condition une étiquette).

Il est courant de travailler avec des FSM déterministes. Ces derniers sont une restriction des FSM car :

- ils ne possèdent qu'un seul état initial : $Card(I) = 1$
- pour tout état s et pour toute lettre a, il existe au plus une transition partant de s portant l'étiquette a : $\forall\, s \in S, \forall\, a \in \Sigma, Card(\delta(s,a)) \leq 1$

Les automates possèdent naturellement une grande capacité à « reconnaître » des motifs textuels, et sont de ce fait très utilisés en théorie des langages (pour l'anecdote, la célèbre commande de recherche textuelle *grep* sous UNIX a été implémentée à l'aide d'automates).

On parle de *calcul* (et également de chemin ou de trace) dans un automate pour désigner une suite de transitions successives. Un *calcul* est dit *réussi* si son état de départ se fait sur un des états initiaux (et dans le cas d'un FSM déterministe, de son état initial) et son état d'arrivée coïncide avec un état final. Un mot est *accepté* (ou reconnu) par un automate s'il est l'étiquette d'un calcul réussi. Enfin, l'ensemble des mots acceptés par un automate constitue le *langage* reconnu par l'automate.

Sur la est représenté un automate déterministe à états finis, capable de reconnaître certains mots formés par les lettres *m*, *i* et *u*. Il représente de ce fait un clin d'œil à l'énigme MU posée par D.Hofstadter dans son célèbre ouvrage Gödel, Escher, Bach [Hofstadter 1979]. De manière formelle, cet automate s'écrit de la manière suivante :

$S = \{ 1, 2, 3 \}$

$\Sigma = \{ m, i, u \}$

$\delta = \{ (1,m,2), (2,i,3), (3,u,2), (3,m,3) \}$

$I = \{ 1 \}$

$F = \{ 3 \}$

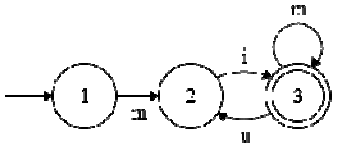


Figure I-6 : Automate à états finis « MIU »

Cet automate est capable de reconnaître, à raison d'une lettre à la fois, le mot *mi* (le plus petit mot pouvant être reconnu par l'automate) ainsi que tous les mots commençant par *mi* et finissant par *m*, ou par *ui* tels que : *mim, mimm, mimmm,* ou bien *miui, miuiui, miuiuiui,* ou encore *mimui, miuim, mimuim, miuimui*…etc.

On peut, à partir de la définition de l'automate, déduire l'expression régulière permettant de retrouver les mots Σ^* reconnus par cet automate : `mi[[ui]*[m]*]*`

Dans le cas où l'automate sert à modéliser un élément d'un système complexe, on trouve parfois dans la littérature une notation particulière, utilisée pour les transitions :

– ? : réception de message ou action externe ;

– ! : envoi de message ou action interne.

La Figure I-7 schématise d'un point de vue serveur l'établissement d'une connexion TCP : le serveur reste dans l'état 1 (assimilé à un état de type « *listen* ») en attendant un message *syn*, passe dans l'état 2 dans le cas où un client envoie un tel message. Le serveur renvoie un message de type *syn/acc* et, à la reception d'un message *ack*, passe dans l'état 4 : la connexion est établie. Ce schéma à but pédagogique ne prend pas en compte l'armement des *flags*, l'aspect synchrone de l'établissement d'une connexion TCP (le client n'est pas modélisé ici) et néglige le phénomène de *timeout*.

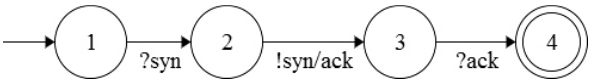


Figure I-7 : Automate émettant et recevant des messages

Les FSM ont bénéficié d'une évolution majeure dans les années 90 : la possibilité de temporiser les transitions. Les automates temporisés ou *Timed Automata* ont été formalisés dans [Alur et al. 1994].

c) ***Réseaux de Petri***

Les réseaux de Petri (*Petri-Nets*, abrégé en *P-N*) ont été inventés au début des années 60 par le mathématicien Carl Adam Petri [Petri 1962] et leur intérêt pour la modélisation de systèmes a notamment été montré dans [Peterson 1981].

Les RdP sont des graphes orientés : leurs éléments de base sont les sommets, ou nœuds (constitués par des places et des transitions) et les relations entre ces nœuds appelées arcs. Il existe plusieurs définitions formelles d'un RdP basique dans la littérature scientifique, mais nous faisons le choix d'en donner une définition personnelle, que nous jugeons plus simple. Il s'agit du triplet :

$$PN = \{ P, T, F \}$$

Avec :

– $P = \{ p_1, p_2, p_3, \ldots, p_m \}$ est un ensemble fini de places (ronds blancs),

– $T = \{ t_1, t_2, t_3, \ldots\ldots, t_m \}$ est un ensemble fini de transitions (traits perpendiculaires aux arcs),

– F est un ensemble d'arcs (flèches). $F \subseteq (P \times T) \cup (T \times P)$, en d'autres termes un arc ne peut pas connecter deux places ou deux transitions.

On ajoute souvent un marquage initial M_0 au RdP tel que :

$$M_0 : P \rightarrow \{0, 1, 2, \ldots.\}$$

Nous entendons par « marquage » un nombre (positif ou nul) de jetons (*tokens*), également appelés marques, contenus dans les places au début de la simulation. Le franchissement d'une transition est régi par des règles, la principale règle pouvant être énoncée comme suit : "*Une transition est dite franchissable si toutes les places en entrée comportent au moins un jeton*". Les jetons représentent le fait que dans un système existent

des ressources, pouvant être consommées si elles sont disponibles, et des ressources produites.

Une extension des réseaux de Petri propose de prendre en compte l'aspect hiérarchique, absent du formalisme de base : elle repose sur un cadre mathématique très formel, et est présentée dans [Fehling 1993].

Plusieurs autres extensions proposent également de tenir compte des aspects temporels, en d'autres termes de rendre synchrones les RdP. Les plus simples des extensions temporelles ajoutées aux RdP ont été présentées à l'occasion de travaux de deux thèses de doctorat dans les années 70. Il s'agit des Time Petri Nets [Merlin 1971] et des Timed Petri Nets [Ramchandani 1974]. Les Time Petri Nets sont très proches des Timed Automata [Bérard et al. 2005]. Plus récemment ont été introduits des RdP temporisés de manière encore plus précise : outre les conditions habituelles de franchissement d'une transition, et la temporisation éventuelle de celle-ci, sont ajoutées des contraintes liées à l'âge des jetons présents dans les places en amont [de Frutos Escrig et al. 2000] [Abdulla et al. 2001].

Concluons notre étude consacrée aux RdP en mentionnant une de leurs plus populaires extensions : les RdP colorés [Jensen 1992], qui permettent la manipulation de variables liées aux jetons.

d) ***Systèmes à évènements discrets***

Ils font eux aussi partie de la famille des MOC, leur particularité est d'évoluer en fonction d'évènements. Ils sont présentés dans [Fishman 1978], [Zeigler 1976] et [Zeigler et al. 2000]. Le plus populaire de ces formalismes est DEVS. Il est au centre de nos travaux et nous le présentons de manière exhaustive en 1.3.

e) ***Cas des formalismes « high-level »***

À l'inverse de tous les formalismes vus depuis b), un formalisme de type *high-level* contient peu ou pas de détails sur l'implémentation. Les langages de modélisation de type « *system level* », utilisés pour modéliser des systèmes, sont par définition situés à un haut niveau d'abstraction. On trouve par exemple dans cette catégorie le formalisme UML, que l'on peut qualifier de standard, et son sous-ensemble SysML (chapitre II).

Il existe enfin des formalismes, descendants directs des MOC mais capables de modéliser des concepts plus abstraits que le seul calcul. Les diagrammes de machines d'états en sont un parfait exemple : intégrés à UML, amélioration des Statecharts présentés dans [Harel 1987], eux-mêmes extension des automates. Les diagrammes de machines d'états sont capables de modéliser des systèmes complexes, des activités parallèles, de décrire des hiérarchies de composition. Ces diagrammes sont donc considérés comme des formalismes *high-level* à part entière, même s'ils utilisent des éléments de plus bas niveau.

Qu'ils soient de haut niveau ou de bas niveau, les formalismes que nous venons de citer restent très génériques, voire universels, et englobent bon nombre de domaines. Ils ont parfois été spécialisés dans le but de faire bénéficier de leur puissance de modélisation un ou des domaines particuliers. Dans ce cas, ils sont utilisés dans le cadre d'outils spécifiques. La *portée* d'un formalisme peut donc constituer un autre critère de classification : est-ce un

formalisme générique, couvrant plusieurs domaines, ou au contraire est-il adapté à un domaine particulier ?

Le fait qu'un formalisme soit de haut niveau ou de bas niveau d'abstraction ne détermine en rien le niveau d'abstraction qu'il est capable de décrire, en d'autres termes un formalisme de bas niveau d'abstraction est tout à fait à même de décrire des systèmes complexes, à un niveau d'abstraction élevé.

1.2.3. Interopérabilité des formalismes

Concluons à présent cette sous-section en introduisant le problème d'interopérabilité au sein de ces formalismes.

Bien que ce problème soit traité de manière approfondie dans le chapitre III, et appliqué à DEVS en particulier, il est important de l'évoquer ici, après avoir eu un aperçu de la diversité des formalismes de modélisation existant.

Le terme interopérabilité désigne de manière générale la « capacité que possèdent des systèmes informatiques hétérogènes à fonctionner conjointement, grâce à l'utilisation de langages et de protocoles communs, et à donner accès à leurs ressources de façon réciproque » (définition Grand Dictionnaire Terminologique[1]). Nous allons voir ici qu'il n'existe pratiquement aucune interopérabilité entre les formalismes présentés ci-dessus.

a) *L'interopérabilité : dans quel but ?*

L'existence de plusieurs formalismes de modélisation nous amène naturellement à nous poser la question suivante : peut-on combiner les aspects des uns et des autres pour modéliser des systèmes, au sein de modèles hybrides ? Cette question est légitime : un système est souvent composé de sous-systèmes, et ces derniers sont parfois de nature différente. Or, il existe plusieurs formalismes, chacun étant adapté pour décrire un type de problème en particulier. N'est-il donc pas possible de modéliser un système en utilisant au moins deux formalismes différents pour modéliser ses sous-systèmes, et de pouvoir cependant simuler ces modèles ? Nous venons d'exprimer un besoin d'interopérabilité.

En restreignant ce raisonnement à un seul formalisme, implémenté sur plusieurs environnements de modélisation et simulation différents (pour peu que ce formalisme en dispose), apparaissent d'autres problématiques, par exemple : existe-t-il des passerelles entre des modèles issus du même formalisme mais implémentés dans des environnements différents ? Nous venons d'exprimer un autre besoin d'interopérabilité.

L'intérêt de telles passerelles est évident : favoriser les échanges entre scientifiques utilisant le même formalisme mais sous des plateformes différentes, améliorer le stockage et la réutilisabilité des modèles…etc. L'interopérabilité ne porte alors plus sur des formalismes différents mais sur les différentes implémentations d'un même formalisme. Pourtant, nous verrons dans le chapitre III que l'interopérabilité entre deux modèles créés avec deux plateformes différentes dédiées au même formalisme (ex : DEVS) n'est absolument pas garantie.

[1] www.granddictionnaire.com

b) ***Plusieurs types d'interopérabilité***

Les questions précédentes nous ont permis d'entrevoir qu'il existait plusieurs types d'interopérabilité, et donc plusieurs types de besoins :

- Décrire un système en combinant plusieurs modèles employant un même formalisme mais implémentés dans des environnements différents ;
- Simuler, dans un environnement *b*, un modèle, spécifié (programmé) dans un environnement *a* ;
- Décrire, et si possible simuler, un système en utilisant plusieurs modèles provenant de plusieurs formalismes ;
- Décrire, et si possible simuler, un système à l'aide de modèles appartenant à des vues (niveaux d'abstraction) différentes.

c) ***Solutions proposées***

Notre discussion nous amène à considérer l'existence de deux types d'interopérabilité distincts : l'interopérabilité « interne » à un formalisme (qui concerne les modèles propres à un formalisme que l'on désire porter vers d'autres plateformes, par exemple), et l'interopérabilité « externe » (qui concerne les modèles issus de formalismes différents).

Actuellement, les solutions proposées pour l'interopérabilité des formalismes de bas niveau ne sont pas universelles et font rarement l'objet de standards. Plusieurs équipes de chercheurs peuvent par exemple travailler sur l'interopérabilité entre deux formalismes identiques et proposer pourtant des solutions différentes : ces solutions sont parfois juste des transformations formelles et ne sont donc pas implémentées. Le chapitre III de ce document traite en profondeur de l'interopérabilité autour de DEVS, qu'elle lui soit interne ou externe. La section qui suit est quant à elle dédiée à ce formalisme.

1.3. Modélisation et Simulation à évènements discrets avec DEVS

Intéressons-nous à présent au formalisme central de ce travail : le formalisme DEVS. En tant que MOC, il hérite des concepts de base qui se rapportent à ces derniers, à savoir les états, et les transitions, auxquels se rajoute la notion d'évènement.

Le formalisme DEVS (Discrete EVent system Specification) a été introduit au milieu des années 1960 par le Professeur Bernard P. Zeigler [Zeigler 1976] [Zeigler 1984]. Il est de nos jours utilisé par une large communauté de scientifiques, qui l'ont étendu afin de l'adapter à certains domaines spécifiques. Il hérite des concepts de la théorie générale des systèmes, ainsi que des formalismes basés sur les états et les transitions, la gestion du temps via les évènements et la notion d'échéancier, et repose sur une base mathématique inspirée de la théorie des ensembles.

Ce formalisme est un formalisme abstrait, orienté vers la modélisation et la simulation de systèmes à évènements discrets ; il est ensuite implémenté, dans la plupart des cas, au moyen de langages orientés-objet [Zeigler 1987]. La popularité de DEVS dans le monde de la recherche académique vient principalement du fait qu'il permet d'appréhender un système de manière comportementale et structurelle : DEVS est modulaire et hiérarchique.

De plus, une propriété importante de DEVS est qu'il fournit automatiquement un simulateur pour chaque modèle : chaque modèle défini et implémenté selon DEVS peut être simulé directement. Il y a donc une séparation explicite entre modélisation et simulation.

Tout système, dont les différents états possibles sont connus et finis, et dans lequel les transitions (conditions de passage d'un état à un autre) sont définies, peut être modélisé en utilisant DEVS. Les transitions peuvent être déclenchées suite à l'expiration d'une horloge interne au modèle, ou bien sous l'action de stimuli extérieurs. Les modèles DEVS peuvent être de deux sortes : modèles atomiques ou modèles couplés.

Les différentes incarnations de DEVS dans des simulateurs ou des environnements de M&S différents conduisent toutefois à une absence d'interopérabilité entre les différents modèles. Du fait de ses implémentations orientées-objet, DEVS nécessite pour être utilisé comme formalisme M&S par une équipe que cette dernière compte au moins une personne maîtrisant parfaitement la programmation orientée-objet. C'est la raison pour laquelle, malgré l'extensibilité et l'adaptabilité de DEVS, il reste surtout populaire dans le monde de la recherche académique et bien moins dans le monde industriel.

1.3.1. **Modèle atomique**

a) ***Définition***

Les modèles atomiques sont les plus petits éléments constitutifs de DEVS. Ils décrivent le comportement, ou une partie du comportement, du système. Ils sont définis comme suit :

$$AM = < X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda >$$

où :

- $X = \{(p,v)|p \in InputPorts, v \in Xp\}$ est l'ensemble des entrées, par lequel les évènements externes sont réceptionnés. InputPorts est l'ensemble des ports d'entrée, et Xp est l'ensemble des valeurs possibles pour ces entrées,
- $Y = \{(p,v)|p \in OutputPorts, v \in Yp\}$ est l'ensemble des évènements de sortie, par lequel les évènements externes sont réceptionnés. OutputPorts est l'ensemble des ports de sortie, et Yp est l'ensemble des valeurs possibles pour ces entrées,
- S est l'ensemble des états du système,
- $ta : S \rightarrow R_0+ \cup +\infty$ est la fonction d'avancement du temps (correspondant à la durée de vie des états),
- $\delta_{int} : S \rightarrow S$ est la fonction de transition interne,
- $\delta_{ext} : Q \times X \rightarrow S$ avec $Q = \{(s,e)/s \in S, e \in [0,ta(s)]\}$ est la fonction de transition externe,
- $\lambda : S \rightarrow Y$, avec $Y = \{(p,v)|p \in OutputPorts, v \in Yp\}$ est la fonction de sortie.

b) ***Exécution de la simulation***

La transition la plus simple (la transition interne) se comporte de la manière suivante : à un instant donné, un système est dans l'état s∈S. À moins qu'un évènement externe arrive sur un des ports d'entrée, le système reste dans l'état s pendant une durée d définie par d=*ta*(s). Lorsque *ta*(s) arrive à expiration, le modèle envoie sur un des ports de sortie la valeur donnée par λ(s), et évolue vers un nouvel état s'∈S défini par δ_{int}(s). Une telle transition, déclenchée par l'expiration de *ta*(s), est une transition interne. Si un évènement externe x∈X se produit, sur un des ports d'entrée, avant l'expiration de d, cela déclenche une transition externe. Dans ce cas, c'est la fonction δ_{ext}(s,e,x) qui définit quel état est le prochain état s' (avec s l'état courant, e le temps écoulé depuis la dernière transition, et x∈X l'évènement reçu). Dans les deux cas, le système se retrouve donc dans un état s' pour une nouvelle durée d' = *ta*(s') et l'algorithme recommence.

c) ***Valeurs particulières renvoyées par* ta**

La fonction d'avancement du temps ta peut avoir des valeurs particulières. Si sa valeur est +∞, l'état s est appelé « état passif » : le système va rester indéfiniment dans cet état jusqu'à ce qu'un évènement externe se produise. Bien entendu, lors de l'implémentation de l'algorithme, +∞ devra être traduit en un mot clef, ou une valeur particulière, afin d'être interprété par le compilateur du langage de programmation utilisé.

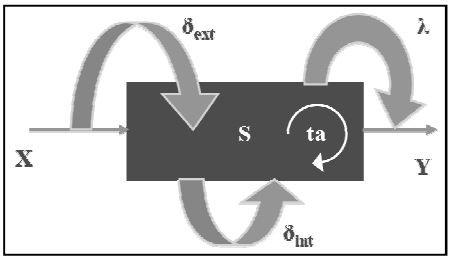


Figure I-8 : Modèle atomique DEVS

En revanche, si la valeur renvoyée par la fonction *ta* est 0, l'état est qualifié de transitoire : il déclenche instantanément λ(s) et effectue une transition vers l'état s'= δ_{int}(s). La Figure I-8 montre une représentation possible d'un modèle atomique DEVS.

1.3.2. Modèle couplé

a) ***Définition***

Un modèle couplé DEVS se compose d'au moins un sous-modèle (atomique ou couplé). Un modèle couplé peut être vu comme un modèle parent, décrivant une hiérarchie (i.e. une liste de sous-modèles ainsi que les liens qui les unissent). Formellement, un modèle couplé est défini par :

$$MC = <X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, select>$$

où

- X = {(p,v)|p∈InputPorts, v∈Xp} est l'ensemble des entrées, par lequel les évènements externes sont réceptionnés. InputPorts est l'ensemble des ports d'entrée, et Xp est l'ensemble des valeurs possibles pour ces entrées,

- Y = {(p,v)|p∈OutputPorts, v∈Yp} est l'ensemble des évènements de sortie, par lequel les évènements externes sont réceptionnés. OutputPorts est l'ensemble des ports de sortie, et Yp est l'ensemble des valeurs possibles pour ces entrées,
- D est l'ensemble des noms des composants, d∈D,
- M_d est un modèle DEVS (soit atomique, soit couplé),
- *EIC* est l'ensemble des couplages des entrées externes : ce lien existe entre le port d'entrée du modèle couplé et le port d'entrée de l'un de ses sous-modèles,
- *EOC* est l'ensemble des couplages des sorties externes : ce lien existe entre le port de sortie du modèle couplé et le port de sortie de l'un de ses sous-modèles,
- *IC* est l'ensemble des couplages internes ; un couplage interne est un lien impliquant le port de sortie d'un sous-modèle du modèle couplé, et le port d'entrée d'un autre sous-modèle,
- *select* est la fonction de sélection : elle permet de lever les ambiguïtés dans le cas où, à la même date, plus d'un modèle atomique doit effectuer une transition interne : pour ce faire, elle définit des priorités entre les modèles de D, sous forme de liste ordonnée.

La Figure I-9 illustre l'exemple d'un modèle couplé contenant 2 sous-modèles : un modèle atomique, et un modèle couplé, contenant lui-même deux autres sous-modèles.

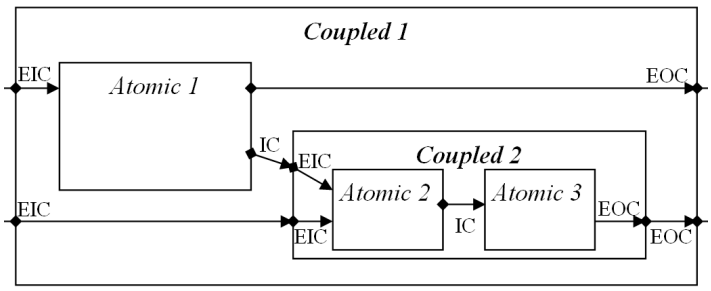


Figure I-9 : Exemple de modèle couplé DEVS

C'est un exemple simple des niveaux de hiérarchie que DEVS est capable de décrire. Les fonctions de couplage (EOC, EIC et IC) sont marquées, et les ports sont représentés par des losanges noirs. Par exemple, le modèle couplé 1 possède deux ports d'entrée, et deux ports de sortie. Le modèle atomique 2 possède un port d'entrée, et un port de sortie. Il a été prouvé que DEVS était « fermé sous couplage », ce qui signifie qu'un modèle couplé (quel que soit le nombre de ses sous modèles) peut être transformé en modèle atomique unique, qui en est l'exact l'équivalent.

1.3.3. **Le simulateur**

a) ***Structure***

La manière la plus simple de simuler un modèle DEVS est d'écrire un programme dont la structure hiérarchique est équivalente à celle du modèle devant être simulé. C'est cette

méthode qui est développée dans [Zeigler et al. 2000]. Une routine appelée *simulateur* est associée à chaque modèle atomique, tandis qu'une routine appelée *coordinateur* est associée à chaque modèle couplé. Le rôle d'un coordinateur est de transmettre les messages à ses enfants, et le coordinateur racine (ou *root*) a en plus la charge d'interroger ses enfants afin, notamment, de connaître la date du prochain évènement, d'entrée ou de sortie. La Figure I-10 représente l'arbre de simulation correspondant au modèle couplé que nous avons présenté sur la Figure I-9.

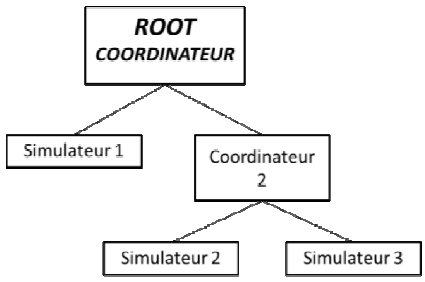


Figure I-10 : Arbre de simulation du modèle présenté sur la Figure I-9

b) ***Séparation entre modèle et simulateur***

DEVS sépare modèles et simulateur de manière explicite : le dernier est « automatiquement » construit à partir des informations (fonctions, variables, hiérarchies) contenues dans les premiers. Cette construction fait appel à un mécanisme d'héritage qui fait hériter de classes DEVS générales les classes des modèles atomiques et couplés composant le modèle à simuler, et leur associe un moteur de simulation, souvent appelé *simulation engine* (abrégé en SE) dans le monde de DEVS, qui implémente pour chaque plateforme de simulation les algorithmes des simulateurs abstraits DEVS décrits dans [Zeigler et al. 2000]. Le SE interagit, lors de la simulation, avec les instances des modèles DEVS : il les interroge et les fait évoluer dans le temps. Il a été formellement démontré [Wainer 2009] que l'entité « simulateur » est capable d'exécuter correctement le comportement décrit par l'entité « modèle ». On considère donc que, théoriquement, il n'y a pas de perte de précision au moment de la phase de simulation, car les simulateurs DEVS sont supposés corrects.

1.3.4. Évolutivité et implémentations de DEVS

a) ***Évolutivité***

Le formalisme DEVS que nous venons de présenter est qualifié de classique. Il a bénéficié au fil du temps de nombreuses extensions, chacune permettant de résoudre un type de problème particulier ou de mieux s'adapter à des domaines spécifiques. Nous en citons quelques-unes ici, même si nous ne les utilisons pas directement dans nos travaux, car notre objectif, sur le long terme, est de pouvoir les intégrer au méta-modèle de DEVS que nous proposons dans le chapitre IV.

Une des extensions majeures de DEVS a été d'améliorer la gestion des évènements simultanés grâce à Parallel DEVS (ou PDEVS) [Chow et al. 1994]. Ce formalisme permet une plus grande souplesse au niveau de la gestion des priorités en les traitant non plus au niveau du modèle couplé mais au niveau du modèle atomique. PDEVS ajoute dans le modèle atomique une fonction de confluence, $\delta_{con} : Q \times X^b \rightarrow S$. Cette fonction permet de gérer les collisions pour des évènements programmés simultanément. Les ensembles X et Y deviennent

X^b et Y^b. En pratique, cela revient à collecter non pas un seul évènement d'entrée et de sortie à la fois, mais un ensemble d'évènements, pouvant se produire au même moment. La fonction de sélection *select* disparaît du modèle couplé.

Une autre évolution importante du formalisme DEVS concerne les modèles à structure dynamique, c'est-à-dire les modèles dont la structure est susceptible d'évoluer durant la simulation. Ceci est particulièrement utilisé pour modéliser des systèmes qui s'adaptent à leur environnement. Deux formalismes distincts implémentent cette évolution. Il s'agit de DSDE (Dynamic Structure Discrete Event) [Barros 1996] [Barros 1997], dont les algorithmes de simulation ont été détaillés dans [Barros 1998], et dynDEVS [Uhrmacher 2001]. Certains systèmes peuvent se définir dans un espace, que l'on peut assimiler à une grille de cellules. Le formalisme Cell-DEVS [Wainer 1998] [Wainer et al. 2002], qui combine DEVS et les automates cellulaires, permet de recréer une telle grille, dans laquelle chaque case (ou cellule) est un modèle atomique, qui, en fonction des évènements reçus, peut être actif ou inactif.

Tous ces formalismes peuvent être vus comme *surclassant* le formalisme DEVS. À l'inverse, il est possible aussi de *sous-classer* DEVS : c'est le cas par exemple du formalisme FD-DEVS (*Finite Deterministic* DEVS) [Hwang 2005]. L'idée principale de ce sous-ensemble de DEVS réside en trois points :

- Un nombre fini d'évènements et d'états ;
- La durée de vie d'un état peut soit être un nombre rationnel soit l'infini ;
- L'échéancier peut soit être préservé soit mis à jour par un évènement interne (i.e. un évènement d'entrée ne peut pas le mettre à jour).

b) ***Implémentations de DEVS***

Environ une vingtaine de simulateurs pour DEVS[1] ont été implémentés à ce jour. Certains se présentent uniquement sous la forme de code objet (librairies) et d'autres s'intègrent dans des environnements de modélisation et simulation avec GUI (*Graphical User Interface*). Ces simulateurs ont été programmés à l'aide de différents langages.

Un test comparatif de différents simulateurs DEVS est proposé notamment dans [Wainer et al. 2011] : il se base sur un *benchmark* (outil d'analyse comparative) créé spécialement pour DEVS, DEVStone [Glinsky et al. 2005]. Nous proposons ici de classer ces différentes implémentations en fonction du langage de programmation employé.

- **Outils programmés en C++**

L'environnement DEVS-C++[2] [Zeigler et al. 1996] [Zeigler et al. 1997] permet la spéfication de modèles avec l'extension Parallel DEVS. La librairie ADEVS[3] [Nutaro 2010] implémentée à la fin des années 90 en C++ (avec des passerelles vers Java) permet la spécification de modèles DEVS parallèles (Parallel DEVS) ou dynamiques (DSDEVS). Ces deux outils ont été développés à l'université d'Arizona.

[1] http://cell-devs.sce.carleton.ca/devsgroup/?q=node/8
[2] http://www.acims.arizona.edu/SOFTWARE/software.shtml
[3] http://freecode.com/projects/adevs

L'environnement-outil CD++[1] [Wainer 2002] permet la spécification graphique de modèles DEVS classiques et cellulaires. Un cas complet d'utilisation de cet environnement dans le domaine de la biologie est donné dans [Djafarzadeh et al. 2005]. CD++ a été développé grâce à une collaboration entre l'université de Carleton et celle de Buenos Aires.

L'environnement PowerDEVS[2] [Kofman et al. 2003] [Bergero et al. 2011] est un environnement dédié à DEVS et plus particulièrement à la simulation de systèmes hybrides (1.1.6.b)). Il est développé à l'université de Rosario.

Il existe également la plateforme VLE, une plateforme de multimodélisation et de simulation basée sur DEVS. Cette plateforme est détaillée en 3.1.2.c).

- **Outils programmés en Java**

Dans cette catégorie on trouve l'environnement JDEVS [Filippi, 2003] qui utilise une interface graphique pour la génération des modèles atomiques et couplés. Un point intéressant au niveau de la réutilisabilité des modèles, ou du moins au niveau de l'indépendance par rapport à un langage, est que cet environnement utilise XML pour spécifier les modèles couplés, au moyen d'une DTD (*Document Type Definition*). Il existe également l'environnement DEVSJAVA[3] qui permet de créer des modèles DEVS à structure variable, et, plus récemment, a été proposé l'environnement graphique DEVS-Suite[4] [Zengin et al. 2010].

Enfin, très récemment, a été développé l'environnement MS4ME[5], sous licence commerciale cette fois-ci. MS4ME a été créé au sein d'EMF, puis adapté aux besoins du formalisme DEVS (et de son extension Parallel DEVS). C'est le Pr. Ziegler, le « père » de DEVS, qui en est à l'origine.

- **Outils programmés en Python**

PythonDEVS (ou PyDEVS)[6] est un simulateur pour DEVS en Python, développé au sein du laboratoire MSDL de l'université de McGill par [Bolduc et al. 2001]. Son but est essentiellement éducatif. Il a été créé par des chercheurs ayant également travaillé sur l'environnement de méta-modélisation AToM^3 : ce dernier repose sur l'API PyDEVS. C'est pourquoi il existe une passerelle entre cet environnement et le simulateur PythonDEVS, car il est possible de transformer directement en modèles interprétables par PythonDEVS des modèles spécifiés dans AToM^3. Avec ce simulateur est proposée une documentation simple et exhaustive, avec notamment des fichiers « patrons » qui donnent la forme générale que doivent avoir des modèles DEVS spécifiés en PythonDEVS. Nous les utiliserons dans le dernier chapitre de ce travail pour montrer comment générer du code vers cette plateforme de simulation. Récemment, des chercheurs de l'équipe TIC de la Faculté des Sciences et Techniques de l'Università di Corsica ont développé un environnement de modélisation et simulation collaboratif DEVS basé sur Python : DEVSimPy[7][Capocchi et al. 2011]. Ce dernier est open source, sous licence GPL. Il s'appuie à la fois sur la bibliothèque graphique

[1] http://sourceforge.net/projects/cdpptoolkit
[2] http://sourceforge.net/projects/powerdevs
[3] http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA
[4] http://devs-suitesim.sourceforge.net
[5] http://www.ms4systems.com
[6] http://msdl.cs.mcgill.ca/projects/projects/DEVS
[7] http://code.google.com/p/devsimpy

wxPython et sur l'API PyDEVS, écrites toutes deux au moyen du langage Python. Un des objectifs de cet environnement est de faciliter le travail et l'échange entre chercheurs en proposant une mise en commun (au sein de bibliothèques) de leurs modèles DEVS provenant de différents domaines.

1.3.5. **Représenter DEVS ?**

Il n'existe pas de manière normalisée de représenter, définir, des modèles DEVS, malgré plusieurs efforts fournis en ce sens (évoqués dans le chapitre III de ce travail). Les solutions les plus employées dans la littérature consistent à :

- décrire les modèles atomiques et couplés en énumérant les tuples qui les composent, en employant du pseudo-code pour définir les différentes fonctions atomiques, puis d'ajouter en cas de couplage un schéma qui représente les imbrications et les liens des modèles (de même type que le schéma présenté en Figure I-9) ;
- donner le code objet correspondant aux modèles.

Dans le premier cas, les modèles ne sont pas extrêmement compréhensibles, mais le pseudo-code et la représentation graphique des couplages permet de donner une idée de leur structure et de leur comportement. Ils ont de plus le mérite de ne pas être liés à une plateforme. Ils restent cependant des modèles contemplatifs.

A contrario, dans le second cas, les modèles sont productifs, du moins dans la plateforme pour laquelle ils ont été implémentés. En revanche, leur compréhension est encore moins rapide et beaucoup plus difficile que pour les modèles du premier cas.

Des travaux ont été effectués dans le but de proposer une représentation pour DEVS. Leur point commun est de combiner une notation graphique enrichie si besoin est avec du texte. Citons par exemple [Traoré 2009] et [Wainer et al. 2009] : l'auteur propose d'implémenter, dans la boîte à outils CD++, un formalisme graphique pour DEVS, et va plus loin en explorant les pistes de la visualisation des sorties des modèles, notamment en couplant DEVS avec des logiciels d'images de synthèse payants (Maya[1]) ou gratuits (Blender[2]). Citons également une tentative pour exprimer des modèles DEVS en langage naturel, adossée à XFD-DEVS, intitulée « *Natural Language For* XFD-DEVS[3] ». Cette approche reste en pratique assez peu utilisée, car elle ne permet que la spécification de modèles assez simples.

Nous pouvons à présent récapituler les concepts présentés dans la Figure I-5 pour les appliquer explicitement à DEVS. La représentation abstraite (i.e. le(s) modèle(s)) du système source est donnée par son comportement et par les données collectées à travers le cadre expérimental, situé dans ce que nous pourrions appeler le « monde réel ». Le but de la simulation DEVS est de reproduire le comportement du système sous des conditions expérimentales particulières et d'observer son évolution au cours du temps : l'entité capable de reproduire ce comportement est, comme nous l'avons déjà vu, le simulateur. Un modèle DEVS, dans le but d'être simulé, doit être implémenté dans le cadre d'un environnement

[1] http://usa.autodesk.com/maya
[2] http://www.blender.org
[3] http://www.acims.arizona.edu/EDUCATION/XFDDEVS/UpdatedXFDDEVS.htm

orienté-DEVS, qui repose très souvent sur un langage orienté objet. Cet environnement fournit les algorithmes de simulation requis pour faire évoluer les modèles dans le temps. Rappelons également que le fait de modéliser un système consiste à faire une description simplifiée de ce système, avec une certaine abstraction.

La modélisation conduit inévitablement à une perte de précision, à cause de plusieurs facteurs comme : la nature même de l'action de modéliser un système qui bien souvent consiste à des simplifications, les limitations des langages de programmation, la précision des variables utilisées, le type d'architecture matérielle utilisée.

Enfin, le scientifique doit vérifier que le comportement du modèle et les données générées sont en accord avec le comportement du système : c'est l'étape de validation [Labiche et al. 2005].

Conclusion du chapitre

Ce chapitre nous a permis d'appréhender la M&S comme le prolongement presque naturel de l'expérience, puisqu'elle nous permet d'améliorer nos connaissances sur des systèmes qu'il est trop difficile, ou trop dangereux, de soumettre à des expériences réelles.

La M&S ne serait rien sans la notion de modèle, abstraction de la réalité d'un ou plusieurs aspects du système. Créer ces modèles suppose un recours à des formalismes de modélisation, dont on a donné un aperçu tout au long de ce chapitre, ces formalismes sont en général de bas niveau d'abstraction. Parmi eux, le formalisme DEVS possède de nombreux avantages comme la séparation des modèles et de leurs simulateurs, la modularité, la capacité à décrire simplement des hiérarchies de modèles. Ce formalisme, qui repose sur de solides bases théoriques issues des mathématiques, s'implémente la plupart du temps au moyen de langages orientés objet. Ces langages sont difficiles à manipuler pour des non-informaticiens, ce qui implique que pour modéliser un système d'un domaine particulier avec DEVS, il faut, en plus du spécialiste du domaine qui décrit le modèle, qu'un informaticien soit présent pour le transcrire en code objet.

DEVS a fait l'objet de multiples extensions, et il a été implémenté dans de nombreux environnements de modélisation et simulation dédiés. La plupart des modèles DEVS créés existent sous forme de code objet, et ne peuvent pas être simulés sur des plateformes autres que la leur (i.e. leur portabilité est impossible). L'absence de formalisation standard des modèles DEVS vient creuser ce manque d'interopérabilité et de réutilisabilité, freinant ainsi la collaboration entre scientifiques. DEVS reste cependant un formalisme incontournable pour la spécification de modèles à évènements discrets. Comment améliorer l'interopérabilité des modèles DEVS ?

Le chapitre suivant prépare la réponse à cette question, en introduisant l'Ingénierie Dirigée par les Modèles. Cette discipline récente du génie logiciel possède des outils, des techniques, et des standards dont le but est d'améliorer grandement le cycle de vie logiciel en considérant le modèle comme la base de ce cycle : l'IDM permet ensuite de transformer ces modèles en d'autres modèles, et même en code. Notre but, tout au long de ce travail, sera de montrer comment l'IDM peut apporter des réponses aux problèmes soulevés dans ce chapitre quant à l'interopérabilité des modèles et donc des formalismes, et de la réutilisabilité de ces derniers sur d'autres plateformes.

Chapitre II. INGENIERIE DIRIGEE PAR LES MODELES

APRES avoir abordé le monde de la modélisation et simulation avec des concepts, des formalismes de bas niveau d'abstraction, dont le formalisme DEVS, et introduit la notion d'interopérabilité entre ces formalismes, abordons maintenant un autre aspect des modèles à travers ce qui est devenu en quelques années une discipline incontournable du génie logiciel : l'Ingénierie Dirigée par les Modèles (ou IDM).

L'IDM (en anglais *Model Driven Engineering*, abrégé en MDE, parfois *Model Driven Development*, abrégé en MDD) est l'appellation générique pour une discipline particulière du génie logiciel qui regroupe plusieurs familles d'approches partageant des pratiques, des méthodes, des techniques communes, souvent implémentées au sein d'outils de développement.

L'émergence de l'IDM s'inscrit dans l'évolution générale de l'informatique qui vise à se détacher de la machine et du code pour se concentrer sur des concepts plus généraux. L'IDM est née d'un besoin simple : répondre aux exigences grandissantes du génie logiciel en termes de qualité des systèmes, de portabilité et d'interopérabilité. Pour cela, l'IDM propose comme solution de regrouper des concepts au sein d'abstractions de systèmes : les modèles. Ceci n'est certes pas un fait nouveau en génie logiciel et encore moins en modélisation et simulation mais, contrairement aux approches précédentes ou similaires centrées sur les modèles, l'IDM fait systématiquement appel à des langages de description de modèles : les méta-modèles. Ces méta-modèles décrivent comment spécifier des modèles qui peuvent être généralistes, ou bien spécifiques à un domaine, un « métier ». Une fois spécifiés, ces modèles peuvent ensuite faire l'objet de transformations pour obtenir d'autres artefacts, dont du code. Souvent, ces spécifications et ces transformations ont lieu dans le même environnement/outil orienté IDM.

Les deux étapes sur lesquelles repose un processus IDM « complet » sont donc la spécification (i.e. la définition) de modèles dans un premier temps, à l'aide de formalismes de modélisation reposant eux-mêmes sur des méta-formalismes, et dans un second temps, leur transformation en d'autres modèles, en code, ou en d'autres artefacts, au moyen de langages de spécification dédiés aux transformations.

La finalité de l'IDM est double : d'une part améliorer la qualité des systèmes, et d'autre part simplifier leur mise en œuvre et leur maintien sur une ou plusieurs plateformes.

Raisonner exclusivement sur les modèles implique *de facto* de ne plus considérer le code comme l'élément clef du processus de développement mais comme un résultat, une conséquence, généré à la suite d'une ou plusieurs transformations de modèles. Cette idée centrale de génération fait que, sans pour autant remettre en cause le sigle IDM, on qualifie souvent cette dernière, dans la littérature scientifique francophone, d'ingénierie générative dirigée par les modèles [Thomas 2008], ou de *Generative Model Driven Engineering* dans la littérature anglophone.

Grâce aux progrès effectués dans le domaine des langages de spécification et environnements orientés IDM, le modèle est passé du stade « contemplatif » au stade

« productif », au sein d'outils spécifiques [Bézivin et al. 2002b]. En d'autres termes, autrefois utilisé uniquement comme une représentation figée d'un système, sans interaction directe possible avec le cycle du développement logiciel, le modèle est envisagé désormais comme l'élément central, actif, du processus de développement. Une des conséquences les plus « spectaculaires » de cette productivité des modèles est, nous venons de l'évoquer, la génération semi-automatique, voire automatique, de code.

L'IDM présente un intérêt pour deux mondes différents :

- **Le monde de la recherche académique.** Citons tout d'abord l'Action Spécifique IDM au sein du CNRS[1], menée de 2003 à 2004 qui a débouché sur le rapport [Estublier et al. 2005], sur l'organisation d'évènements liés à l'IDM, et sur l'ouvrage [Favre et al. 2006]. Il existe également des conférences spécifiques telles que la célèbre MoDELS[2], ou la toute nouvelle MODELSAWARD[3]. Enfin, des équipes de recherche comme Triskell[4], qui a défini KerMeta [Muller et al. 2005] [Fleurey 2006] ou l'équipe ATLAS[5], avec ATL [Jouault et al. 2006b] [Jouault et al. 2008], langage de transformation de modèles, contribuent à l'essor et à la démocratisation de l'IDM ;
- **Le monde du génie logiciel au sens large**, que ces organisations soient à but lucratif ou pas. Citons par exemple le consortium *Object Management Group*[6] (ou OMG), comprenant plus de 1000 membres, qui se concentre sur la définition de standards au sein de l'IDM regroupés au sein de la désormais célèbre approche MDA ; la société Softeam[7] (atelier Modelio et environnement Objecteering), la société IBM [Booch et al. 2004] avec son « Manifeste MDA » ; la société Microsoft aves ses *Software Factories* [Greenfield et al. 2004b] ; la fondation Eclipse[8] (initiée par IBM, regroupant actuellement de nombreux acteurs du génie logiciel), qui propose l'environnement open-source *Eclipse Modeling Framework* dans lequel sont proposés de nombreux plugins (gratuits ou pas) orientés IDM. Softeam et Eclipse sont d'ailleurs elles-mêmes membres de l'OMG, depuis 1994 pour la première et 2007 pour la seconde, ce qui explique que de plus en plus de plugins Eclipse soient orientés MDA.

La *Model Driven Architecture* (MDA, « Architecture Dirigée par les Modèles ») est certainement l'incarnation la plus célèbre de l'IDM, à tel point que la confusion entre le terme générique MDD (*Model Driven Developpement*) et le terme MDA est très fréquente dans les publications scientifiques. MDA est un ensemble de spécifications qui fournit, outre des méta-

[1] http://www.actionidm.org
[2] http://models2012.info
[3] http://www.modelsward.org
[4] http://www.irisa.fr/triskell/home_html
[5] http://www.inria.fr/equipes/atlas/(section)/publications
[6] http://www.omg.org
[7] http://www.softeam.fr
[8] http://www.eclipse.org

modèles « standards », un méta-méta-modèle ou méta-formalisme, destiné à en spécifier d'autres : MOF.

L'objectif de ce chapitre est de dresser un état de l'art de l'IDM, tout en montrant que cette dernière, loin d'être figée, est en constante mutation du fait du travail effectué par les chercheurs et les industriels. L'IDM n'étant pas une discipline monolithique, il n'est pas simple de la décrire car certains concepts sont partagés par plusieurs méthodes, certaines méthodes se retrouvent dans plusieurs outils ou environnements dédiés, certains outils utilisent les mêmes formalismes…etc.

Devant l'engouement du monde académique, industriel, mais aussi des organismes gouvernementaux pour l'IDM, et le nombre sans cesse croissant de publications scientifiques s'y rapportant, devant la quantité de problèmes abordés et des solutions proposées, faisant appel à des technologies, des supports, des environnements, parfois très différents, nous avons dû faire le choix de limiter le nombre de références, tout en nous efforçant de donner un panorama de ces divers travaux aussi fidèle et représentatif que possible.

La première partie est dédiée à la philosophie générale de l'IDM. L'IDM se positionne à la fois dans la continuité des technologies existantes (utilisation fréquente de la technologie objet) mais aussi en rupture avec ces dernières (notion de composition d'objet remplacée par la notion de transformation de modèle) [Bézivin 2004]. Nous définissons ensuite quelques concepts clef de l'IDM et/ou de MDA, comme par exemple la notion de modèle, de méta-modèle, de méta-formalisme, le tout lié par l'idée de « niveaux méta ». Nous présentons aussi la notion d'espace technique en donnant notamment l'exemple de l'espace technique de l'OMG.

La seconde partie est consacrée aux transformations de modèles, nous y verrons de quoi se compose une transformation, et quelles sont les caractéristiques des différentes approches de transformations possibles en IDM.

Enfin, la troisième partie présente un panorama des grandes approches assimilables à de l'IDM, et donne quelques uns des outils qui les implémentent.

2.1. « Tout est modèle »

Pour être en mesure d'appréhender l'IDM à travers les différentes familles d'approches qui la composent, de comprendre quels sont leurs points communs et leurs différences, quelques concepts doivent être préalablement explicités.

Concernant ces concepts, il est bon de rappeler que l'IDM n'en a pas véritablement *créé* de nouveaux mais a plutôt contribué à expliciter, formaliser, harmoniser, et faire évoluer, ce que l'on pourrait appeler les *best practises*, ou « meilleures pratiques », qui étaient employées par différents acteurs du génie logiciel de par le monde. L'IDM a agi comme un levier qui a démultiplié la puissance des *design patterns* (patrons de conception) [Gamma et al. 1995], des aspects, utilisés jusque lors en génie logiciel, dans le cadre de la programmation orientée objet. Ce contexte particulier permet d'ailleurs d'expliquer, voire de prévenir, certaines confusions qui sont faites dans la littérature dédiée, notamment l'application « abusive » à l'IDM de concepts pourtant bien spécifiques au monde de l'objet (« *instance de…* »).

Par conséquent, même si l'IDM reprend des concepts et des méthodes existants en les mettant en synergie, en les standardisant, afin de faciliter leur mise en œuvre, il reste tout à fait possible, mais plus long, et plus compliqué, de transformer des modèles ou de générer du code dans un cadre hors IDM (tout comme il est possible d'afficher une chaîne de caractères à l'écran en utilisant un langage de type assembleur plutôt qu'un langage objet). Il en résulterait néanmoins une perte énorme de l'interopérabilité que permet l'IDM.

Dans cette première partie, nous nous attachons à définir précisément les concepts de l'IDM, en nous inspirant notamment de [Bézivin et al. 2002b] [Bézivin 2004] [Blanc 2005] dont nous partageons la vision sur la caractérisation des relations entre modèles, méta-modèles...etc.

2.1.1. Définitions autour du modèle

La notion de modèle en IDM reste étroitement liée à la notion de modèle, plus générale, que nous avons présentée dans le chapitre précédent. Ce qui caractérise l'IDM n'est pas tant l'emploi de modèles, que l'utilisation de formalismes et de langages pour décrire et transformer ces derniers. Les notions de modèle, de méta-modèle et de méta-méta-modèle se situent clairement à trois niveaux d'abstraction différents. Il est nécessaire de formaliser ces notions ainsi que les liens qui les unissent, car de cette formalisation dépend la formalisation d'éventuelles transformations entre modèles (effectuées sur une machine) qui rendront ces derniers productifs.

a) *Le modèle : une représentation du monde réel*

Le concept de modèle est relativement semblable en IDM et en M&S. Le modèle, rappelons-le, constitue une abstraction de la réalité et s'utilise pour représenter un système existant ou virtuel, naturel ou artificiel. En IDM plus qu'en M&S, les systèmes représentés sont souvent artificiels, créés par la main de l'homme, il s'agit la plupart du temps de systèmes d'information.

Une définition plus précise de la notion de modèle, complétant celle de [Minsky, 1968] vue dans le chapitre I, a été proposée par [Bézivin et al. 2001] :

> *"A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system."*

La notion d'objectif est selon nous importante dans cette définition, et on peut même ajouter que dans un contexte IDM les objectifs d'un modèle sont multiples : d'une part décrire le système (rôle contemplatif), et d'autre part pouvoir être utilisé de manière dynamique pour créer d'autres modèles ou d'autres artefacts tels que le code (rôle productif). [Kleppe et al. 2004] donnent une définition différente mais tout aussi intéressante du concept de modèle :

> *"A model is a description of (part of) a system written in a well-defined language"*

Cette autre définition met en lumière le fait qu'un modèle est décrit dans un langage, un formalisme : le méta-modèle.

b) ***Le méta-modèle : le langage de description des modèles***

En IDM, un formalisme, ou langage de modélisation, dans lequel est exprimé un modèle, est décrit par un méta-modèle. Le méta-modèle a la particularité de contenir tous les concepts nécessaires pour créer des modèles dans un domaine, un contexte particulier : le méta-modèle est au cœur de l'IDM.

Plus précisément, le rôle d'un méta-modèle est de définir, au minimum, la syntaxe abstraite d'un formalisme, en définissant des concepts et des relations entre ces derniers. Nous verrons que, souvent, les langages de méta-modélisation partagent des concepts hérités de la technologie objet : des classes (pouvant être utilisées dans des héritages) et des propriétés, ces dernières étant soit des références lorsqu'elles sont typées par une autre classe, soit des attributs si elles sont typées par des types de base (ex : entier, booléen…).

Par exemple, le méta-modèle d'un langage de programmation représente sa grammaire, le méta-modèle d'un fichier XML représente sa DTD (*Document Type Definition*). Il existe à ce jour de nombreux langages de modélisation nés d'un processus IDM et donc décrits par des méta-modèles. Il s'agit de méta-modèles représentant des *Domain Specific Languages* [van Deursen et al. 1998] (DSL) ou bien des langages plus universels, couvrant un large spectre d'applications (par exemple le méta-modèle d'UML).

En IDM tout est modèle, un méta-modèle est donc lui aussi un modèle, décrit selon un certain formalisme : le méta-méta-modèle.

Remarque : La frontière entre langage et méta-modèle est mince : les deux sont souvent confondus, par abus de langage, ou souci de simplification, mais il existe toutefois une différence entre ces concepts. En effet, un langage, un formalisme, est un ensemble de concepts, alors que le méta-modèle en est le moyen de définition. Cette différence se retrouve entre une langue, et sa grammaire. Une langue est un ensemble riche et complexe constitué de noms, de pronoms, de verbes, d'adverbes, d'adjectifs, etc. Un traité grammatical à propos de cette langue a pour but de décrire la grammaire, la syntaxe de cette langue : les éléments de syntaxe donnés dans ce traité serviront à créer des phrases, des textes, conformes à la langue. Ce traité *décrit* donc la langue, sans pour autant *être* cette langue : c'est uniquement un outil de description. Dans le même ordre d'idées, dans nos travaux nous proposons un méta-modèle du formalisme de DEVS, ce méta-modèle est un outil dont le but est de créer des modèles DEVS : mais ce méta-modèle *n'est pas* le formalisme DEVS, c'est juste un moyen de *définir* ce formalisme. Ces modèles DEVS appartiennent au formalisme DEVS, et se conforment au méta-modèle de DEVS.

Par analogie avec 2.1.1.a), si on considère le formalisme comme un système, alors le méta-modèle, outil de description, est son modèle. Nous reviendrons sur ces liens entre système, modèle, méta-modèle et méta-formalisme en g).

c) ***Le méta-méta-modèle : description des formalismes de modélisation***

Le méta-modèle employé pour pouvoir décrire des formalismes de modélisation (qui serviront à créer des modèles…) s'appelle un méta-méta-modèle. Par analogie avec les méta-modèles utilisés pour décrire des formalismes de modélisation, un méta-méta-modèle sert à décrire un méta-formalisme (un formalisme de description de formalismes de modélisation). En IDM, on retrouve des méta-formalismes reposant sur des concepts (ou espaces

technologiques) différents (objet, XML, entités-relations…). Par exemple, un des méta-formalismes les plus utilisés en IDM provient de MDA : il s'agit de MOF (2.1.2.b)).

Pour éviter d'avoir recours à un « méta-méta-méta-modèle » (et ainsi de suite…), ce qui conduirait à une infinité de niveaux d'abstraction, un méta-méta-modèle, en IDM, est toujours conçu de manière auto-descriptive, c'est-à-dire qu'il contient en son sein tous les concepts nécessaires à sa propre définition (voir g)).

Raisonnons à présent sur les relations existant entre les différents niveaux d'abstraction vus en 2.1.1. Nous nous basons notamment sur [Estublier et al. 2005] et [Bézivin 2004] qui décrivent ces relations et en proposent une interprétation dans un contexte IDM. Nous sommes en accord avec cette interprétation.

d) ***« Tout est objet » ?***

Dans la philosophie « tout est objet », les deux notions centrales sont l'*héritage* (le fait qu'une classe puisse hériter d'une autre classe) et l'*instanciation* (le fait qu'un objet soit l'instance d'une classe) (Figure II-1).

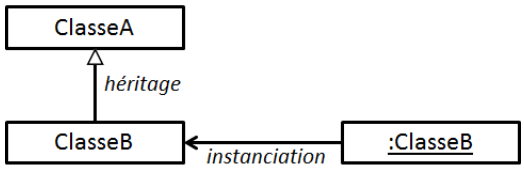


Figure II-1 : Concepts de la technologie objet

De prime abord, on serait tenté d'utiliser, ou plutôt de réutiliser, ces relations, pour caractériser les liens entre les niveaux d'abstraction séparant le modèle, le méta-modèle et le méta-méta-modèle. Par exemple, on lit souvent dans la littérature dédiée à l'IDM qu'un modèle est l'*instance de* son méta-modèle. Or, ceci signifierait que les modèles et les méta-modèles sont décrits au moyen d'une technologie objet : l'IDM n'impose pourtant rien à ce sujet.

Il s'agit donc d'un abus de langage, pouvant s'expliquer par deux facteurs principaux :

- Le contexte de la naissance de l'IDM : comme nous l'avons dit dans l'introduction à cette partie, l'IDM a vu le jour dans un monde fortement influencé par les pratiques de programmation orientée objet ;
- Les tendances actuelles de l'IDM : on assimile souvent cette dernière à l'approche MDA (orientée-objet, proposée par l'OMG). Or, si MDA fait partie des approches IDM les plus populaires, elle n'en reste pas moins une approche IDM parmi d'autres.

e) ***Espace technique***

En fait, tout dépend de l'espace technique (ou technologique) dans lequel on se place. Cette notion d'espace technique [Kurtev et al. 2002] [Bézivin et al. 2005a] [Combemale 2008] nous permet de considérer l'IDM sous un autre jour, et l'articulation des niveaux d'abstraction prend ainsi tout son sens. Selon [Favre et al. 2006] :

« un espace technique est l'ensemble des outils et techniques issus d'une pyramide de méta-modèles dont le sommet est occupé par une famille de (méta)méta-modèles similaires »

Des données identiques pourront être différemment représentées selon l'espace technique dans lequel on se situe. Ce dernier est en pratique déterminé par le méta-formalisme employé. Par exemple, un programme Java est conforme à la grammaire du langage Java et existe sous forme de code (espace technologique de programmation) mais il peut tout aussi bien être décrit au moyen d'un document JavaML conforme à la DTD de JavaML (espace technologique XML).

Ajoutons à cela que le but de l'IDM est d'intégrer plusieurs modèles, exprimés dans plusieurs formalismes, appartenant donc à des espaces technologiques différents.

Comme autres exemples d'espaces techniques citons le *Grammarware*, espace technique des grammaires de langages, qui fait appel aux notations de type BNF ou EBNF, le *BDware*, espace technique des bases de données, qui utilise les entités-relations, ainsi que l'espace technologique des documents XML et assimilés (HTML, XMI, XSLT…), qui repose sur le XML Schema.

Aucun de ces trois espaces techniques n'utilise la technologie objet, bien que certains d'entre eux soient utilisés dans l'IDM (2.3). En revanche, il est vrai qu'un autre espace technologique, celui de MDA, est clairement orienté objet, de par l'utilisation du formalisme UML (entre autres), et du méta-formalisme MOF : ceci résulte également d'un choix fait par l'OMG. Mais ce choix technologique propre à l'OMG n'est pas une généralité en IDM et ne doit pas être un frein à l'utilisation de cette dernière.

Il faut donc veiller à distinguer la philosophie et les concepts généraux de l'IDM d'une part, des choix technologiques spécifiques à des variantes particulières de l'IDM d'autre part. Nous nous permettrons parfois d'utiliser le terme « *instance de* » dans la partie de ce document consacrée à notre approche, orientée MDA et donc partageant avec elle le même espace technique.

f) ***Représentation et conformité***

Afin de bénéficier de l'adaptabilité de l'IDM à des espaces technologiques différents, il est nécessaire de dégager des concepts adaptables à tous les cas particuliers. Comment dès lors qualifier le lien entre un modèle et le monde réel, et celui entre un modèle et son méta-modèle ?

Nous l'avons vu en 2.1.1.a), et aussi dans le chapitre premier de ce travail, le modèle constitue une représentation du monde réel. En conséquent, malgré l'absence de définition formelle et unique de modèle, de système, il est communément admis qu'entre un système et le modèle qui le décrit existe une relation de *représentation*. Cette relation n'est toutefois pas formalisée.

La relation entre un modèle et son méta-modèle peut, quant à elle, se concevoir en fonction de ce que nous avons dit en 2.1.1.b)2.1.1.b) : dans un esprit « tout est modèle », on sait qu'un modèle est décrit dans un langage de modélisation. De plus, l'IDM a pour vocation de rendre les modèles productifs : il faut donc que ces derniers soient interprétables à un moment donné par une machine, qui devra les manipuler. Ceci suppose que leur définition est rigoureuse et obéit à des règles précises contenues au sein de leurs méta-modèles, car les modèles pourront faire l'objet de transformations exécutées sur la machine. Cette notion de méta-modèle conduit à proposer une relation basée sur la *conformité* d'un modèle à son méta-

modèle [Bézivin 2004]. Un modèle est *conforme* à son méta-modèle (on dit aussi qu'un modèle est *typé* par son méta-modèle). Plus que sur la relation de *représentation*, déjà largement explorée au sein du monde de la M&S, c'est bien cette relation de conformation entre un modèle et son méta-modèle qui constitue un point clef de l'IDM.

La Figure II-2 montre les deux relations qui constituent des concepts-clef de l'IDM : la conformation et la représentation.

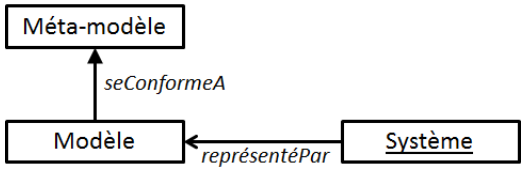


Figure II-2 : Concepts de la technologie IDM

Le lien de conformité, tout en restant générique, peut, selon l'espace technologique concerné (voir 2.1.1.d)), prendre des formes différentes :

- un document XML est *conforme* au XML Schema (ou à sa DTD), on dit aussi qu'il est *valide* ;
- un objet est *conforme* à une classe, on dit aussi qu'il en est une *instance*.

Cette relation de conformité est d'autant plus pertinente qu'elle peut s'appliquer de la même manière à un méta-modèle vis-à-vis de son méta-méta-modèle : un méta-modèle se conforme à son méta-méta-modèle (exactement comme un modèle se conforme à son méta-modèle). Tout méta-modèle représente donc un langage de modélisation. Enfin, un méta-méta-modèle, de par sa nature auto-descriptive, est conforme à lui-même.

g) ***Synthèse : les « niveaux méta » en IDM***

Résumons à présent tout ce que nous avons vu jusqu'à présent : pour clarifier les choses, nous introduisons la notion de « niveaux méta ». Cette notion, applicable à tous les espaces techniques de l'IDM, correspond à quatre niveaux distincts, classés par ordre croissant d'abstraction :

- M_0 est le niveau qui correspond au monde réel,
- M_1 est le niveau des modèles,
- M_2 est le niveau des méta-modèles,
- M_3 est le niveau des méta-méta-modèles.

Nous aurons l'occasion de revenir plus en détail sur ces « niveaux méta » lorsque nous aborderons l'approche MDA de l'OMG. En IDM, pour un espace technologique donné, on utilise un seul méta-méta-modèle. En revanche, il est possible d'utiliser plusieurs méta-modèles, complémentaires, illustrant chacun un aspect du problème considéré, ce qui entraîne par conséquent l'existence de plusieurs modèles.

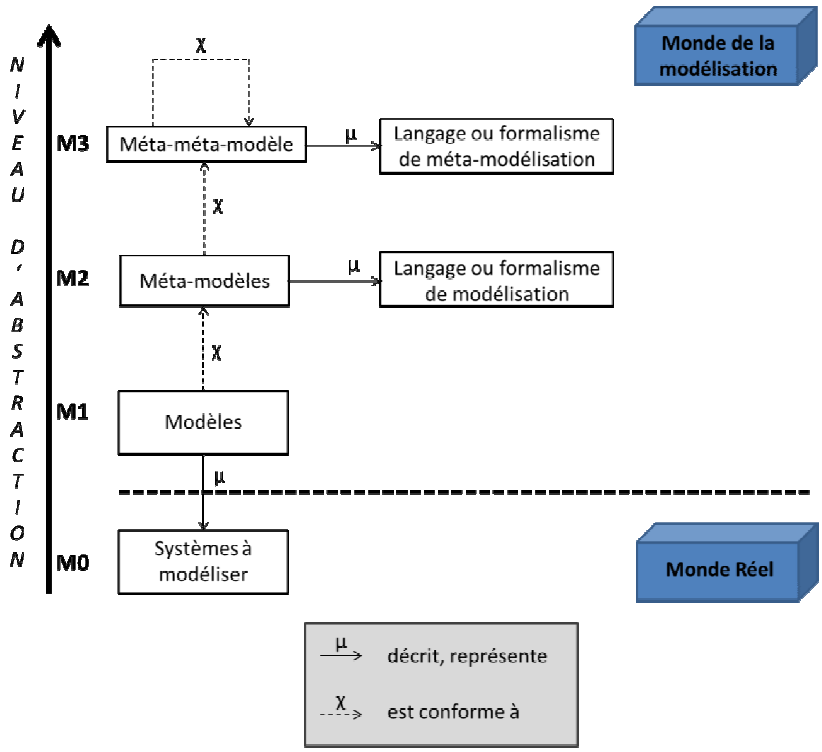


Figure II-3 : Niveaux « méta » et relations en IDM, pour un espace technologique donné

Désignons par χ la relation de conformation, et par μ la relation de représentation selon [Estublier et al. 2005]. Nous les utilisons dans la Figure II-3, qui constitue un résumé de cette première section.

2.1.2. Un espace technologique « moteur » de l'IDM : MDA

Il y a une douzaine d'années, l'intérêt pour l'IDM s'est d'autant plus accru que l'Object Management Group (OMG)[1] a amorcé un changement dans le monde du génie logiciel [Bézivin et al. 2002a] [Bézivin et al. 2002b] [Blanc 2005] [Soley 2004] [Frankel 2003] en structurant et formalisant, notamment par le biais de sa *Model Driven Architecture* (MDA[2]) [Blanc 2005] [Kleppe et al. 2004], de nombreuses techniques et pratiques issues de l'IDM. Certains auteurs comme [Gitzel et al. 2004] voient même MDA comme « la plus importante réalisation de l'IDM ».

Par le biais de RFP (*Request For Proposal*), ou « appel à propositions », auxquelles de nombreux acteurs industriels, académiques et gouvernementaux ont répondu, l'OMG a peu à peu créé un certain nombre de standards et proposé des solutions d'interopérabilité entre ces derniers, ce qui a fortement marqué et influencé les orientations de l'IDM.

Comme nous l'avons déjà mentionné précédemment, l'espace technologique de MDA est clairement orienté-objet. Ceci est dû au fait que MDA, lancée en l'an 2000, succède à l'OMA (*Object Management Architecture*) (en se plaçant certes à un niveau d'abstraction supérieur). Mais MDA n'a pas abandonné le concept d'objet pour autant, afin entre autres de rester en adéquation avec les pratiques techniques actuelles du génie logiciel (implémentation objet, utilisation massive d'UML) et aussi d'intégrer dans sa nouvelle architecture des

[1] http://www.omg.org
[2] http://www.omg.org/mda

standards nés dans l'OMA, par exemple CORBA (*Common Object Request Broker Architecture*) [Siegel 1996].

a) ***La « pyramide » de MDA***

La Figure II-4 montre une pyramide à quatre niveaux d'abstraction qui représente les concepts spécifiques à l'approche MDA de l'OMG : elle est une variante particulière de la Figure II-3.

Utiliser une forme pyramidale permet de donner une indication sur le nombre d'entités pouvant exister sur chaque niveau : le plus bas niveau, M_0, comporte une infinité de systèmes, certains d'entre eux ont fait l'objet de processus de modélisation et sont représentés sous forme de modèles (M_1). Ces modèles sont spécifiés au moyen de méta-modèles généraux situés en (M_2) pouvant être des standards de l'OMG (UML en est l'exemple le plus connu) ou créés pour l'occasion (méta-modèles représentant des *Domain Specific Modeling Languages* ou DSMLs). La sous-section 2.1.2.d) traite du rôle de ces méta-modèles. Enfin, le méta-formalisme MOF (*Meta Object Facility*), au niveau M_3, joue le rôle de méta-méta-modèle. Il est défini avec ses propres concepts. Même si elles ne sont pas indiquées, les relations de conformation et de représentation entre les niveaux sont analogues à celles de la Figure II-3.

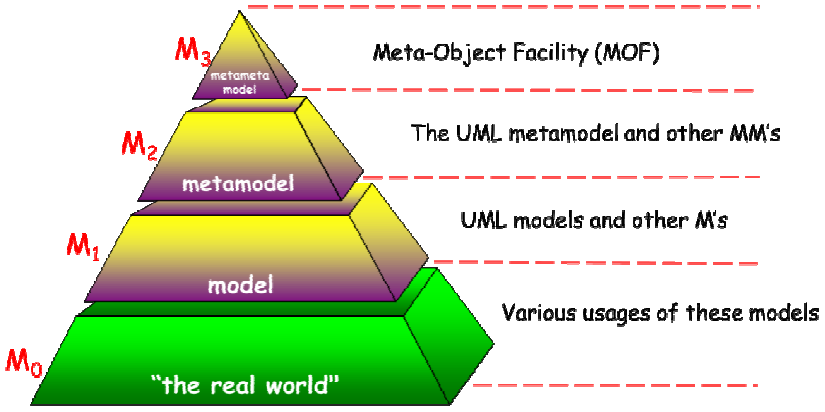


Figure II-4 : La « pyramide » des « niveaux méta » de MDA (extraite d'un document de Richard Paige présent sur le site de la fondation Eclipse[1])

b) ***MOF : sommet de l'architecture MDA***

Le standard le plus important de l'approche MDA est certainement le méta-formalisme MOF[2] : il se situe de par sa nature au sommet de l'architecture à « niveaux méta », le niveau M_3.

La totalité des méta-modèles standardisés par l'OMG se conforment à MOF, et tout méta-modèle créé dans une optique MDA doit s'y conformer aussi. La version de MOF actuellement en cours est la version 2.4.1. Nous verrons également que tout langage de transformation de modèles se réclamant d'une approche MDA doit se conformer à MOF. Ce méta-formalisme est auto-descriptif, en vertu des principes de l'IDM. Il se subdivise en deux sous-formalismes, EMOF (*Essential MOF*) et CMOF (*Complete MOF*). Le premier est une restriction, un sous-ensemble, du second. MOF contient tous les concepts nécessaires à la description de la syntaxe abstraite de méta-modèles au niveau M_2, notamment les notions de

[1] http://www.eclipse.org/gmt/omcw/resources/chapter04/downloads/MOFLecture.York.ppt
[2] http://www.omg.org/spec/MOF/2.4.1

classes, d'attributs, d'héritages, d'associations. On retrouve ces concepts dans la spécification du méta-modèle d'UML par exemple, défini dans l'UML 2.4.1 « *infrastructure* »[1], qui décrit le méta-modèle UML.

Le lien entre ce document décrivant UML d'une part et EMOF et CMOF d'autre part est très étroit : en fait, ces derniers sont issus de la fusion des packages de l'*Infrastructure Library*, la bibliothèque dans laquelle sont contenus tous les packages utilisés pour la spécification d'UML dans l'UML *infrastructure*. En d'autres termes, UML et MOF partagent un cœur commun au niveau des concepts du niveau M_2 dont nous parlons un peu plus haut : UML est partiellement défini en termes de MOF, MOF est défini en termes d'UML (via les diagrammes de classes). Pour l'OMG, UML est un standard de l'IDM.

c) ***Modèles de MDA***

Il existe trois types de modèles au centre de l'architecture MDA, liés entre eux par des transformations de modèles :

- le CIM (*Computation Independent Model*) : un CIM est un modèle indépendant de tout ce qui touche à l'implémentation et même à l'informatique. Les CIM sont des modèles métier (de haut niveau) qui n'ont pas été raffinés (par exemple, des diagrammes UML de cas d'utilisation) ;
- le PIM (*Platform Independent Model*) : un PIM est un modèle de conception indépendant des plateformes et des technologies (ce qui lui confère une grande durée de vie), il représente la logique métier à un niveau plus bas que les CIM. Un PIM est spécifié, usuellement, au moyen de diagrammes de classes UML enrichis par des contraintes OCL ;
- le PSM (*Platform Specific Model*) : un PSM est le modèle le plus raffiné en MDA. Il est utilisé pour générer le code qui sera exécuté sur une ou plusieurs plateformes spécifiques. Souvent, on associe la notion de PSM à la notion de code (le code est assimilé à un PSM particulier).

d) ***UML, méta-modèle universel ?***

Outre ses liens étroits avec l'OMG et MDA (notamment avec MOF) comme nous l'avons vu en b), UML 2.4.1 [Rumbaugh et al. 2005] est aussi et avant tout un formalisme graphique et générique de modélisation : c'est le langage standard pour modéliser tout ce qui touche au monde de l'objet.

UML hérite de trois langages de modélisation célèbres des années 90 : Booch [Booch 1993], OMT (*Object Modeling Technique*) [Ebert et al. 1994] et OOSE [Jacobson et al. 1999]. Il se situe à un haut niveau d'abstraction et est composé de 13 diagrammes (divisés en 5 catégories) permettant de représenter un système de manière aussi bien comportementale que structurelle. Il est principalement utilisé pour spécifier et documenter des systèmes. Son méta-modèle est, comme nous l'avons dit, fourni par l'OMG.

[1] http://www.omg.org/spec/UML/2.4.1/

Cependant, sa grande généricité peut parfois être un inconvénient, surtout lorsque l'on désire exprimer des concepts très spécifiques à des domaines particuliers. L'OMG préconise deux types de solutions pour résoudre ce problème :

- Utiliser les mécanismes de spécialisation d'UML, en ayant recours à la création de profils (« *UML profiles* »). Un profil UML respecte toujours le méta-modèle UML, en d'autres termes il ne permet que de restreindre ce dernier, en ajoutant par exemple des contraintes entre des éléments, en ajoutant une représentation graphique pour certains éléments…etc. Il est également possible d'étendre UML, mais toujours en utilisant son méta-modèle comme point de départ ;
- Définir un nouveau langage (et donc un nouveau méta-modèle) à utiliser à la place d'UML, tout en utilisant les mécanismes de définition de langages préconisés par l'OMG (donc en utilisant le méta-formalisme MOF).

La première solution a par exemple été appliquée pour le langage de modélisation open-source et graphique SysML (*Systems Modeling Language*) proposé par l'OMG[1]. Destiné à la spécification, l'analyse, la conception, la vérification et la validation de systèmes, SysML constitue un sous-ensemble d'UML, et il est d'ailleurs souvent défini comme « l'extension d'un sous-ensemble d'UML ». SysML est en fait un profil d'UML 2.0. Il en conserve certains diagrammes, en adapte d'autres, et en abandonne plusieurs. Au total, SysML réutilise 7 diagrammes UML, sur les 13 originellement présents.

La seconde solution est celle que nous avons par exemple utilisée pour définir MetaDEVS (chapitre IV) : elle présente comme avantage de pouvoir redéfinir complètement la syntaxe abstraite (et la sémantique statique avec OCL) d'un formalisme.

Il n'existe pas à proprement parler de solution « type », d'ailleurs en IDM ces deux solutions sont autant employées l'une que l'autre : tout dépend en fait du problème considéré. Nous verrons par exemple dans le chapitre III que la représentation des modèles DEVS peut, selon les approches, se faire au moyen de diagrammes UML ou au moyen d'un formalisme créé spécialement pour cela.

e) ***OCL***

Lors du processus de modélisation, il peut être utile d'avoir la possibilité d'exprimer des contraintes sur les modèles, c'est-à-dire imposer certaines règles sur les attributs ou les relations. Le langage OCL (*Object Constraint Language*)[2] [Richters et al. 2002], dont la version actuellement en cours est la 2.3.1, permet de spécifier des contraintes, notamment sur les modèles UML. Ces contraintes prennent usuellement la forme de préconditions et de post-conditions. Ce langage est sans effet de bord, il permet seulement l'enrichissement des éléments déjà existants. Il s'agit donc d'un « pur » langage de modélisation et non pas d'un langage de programmation.

En tant que spécification officielle de l'OMG, OCL est très lié à UML et MOF : OCL contient un sous-ensemble basé sur le cœur commun à UML et MOF, ce qui permet à ce sous-

[1] http://www.omgSysML.org
[2] http://www.omg.org/spec/OCL/2.3.1

ensemble d'être utilisé avec UML et MOF, alors que la spécification complète d'OCL ne peut être utilisée qu'avec UML. OCL est également compatible avec XMI.

OCL peut-être également utilisé comme un puissant langage de requête (sur des instances de méta-modèles) afin, par exemple, de récupérer, trier, des éléments précis. C'est de surcroît un des rares langages de contraintes qui ne soit pas attaché à une plateforme particulière. Il est de ce fait très populaire dans le monde de l'IDM et en particulier du MDA.

Nous verrons dans la seconde partie de ce travail qu'OCL nous sera utile dans toutes les phases de notre approche, que ce soit au moment de la spécification du méta-modèle MetaDEVS, ou bien pour enrichir des règles lors des transformations M2M et M2T.

2.1.3. Méta-formalismes disponibles en IDM

Rappelons que si MDA impose l'utilisation de méta-modèles et donc d'un méta-formalisme provenant de la technologie objet, l'IDM dans son ensemble n'impose pas de contraintes particulières sur le choix d'un méta-formalisme plutôt qu'un autre, et donc pas de contraintes sur le choix d'un espace technologique.

Rappelons également que le méta-formalisme MOF reste une spécification, et non un méta-formalisme implémenté. Il a par conséquent fait l'objet d'adaptations concrètes (souvent d'EMOF) afin de démocratiser son utilisation au sein d'environnements IDM : nous décrivons en c) le cas particulier d'Eclipse EMF.

Outre MOF et ses incarnations, il existe bien entendu plusieurs autre méta-formalismes, appartenant à des espaces technologiques différents, employés pour décrire des méta-modèles : certains sont génériques et intégrés dans des standards, d'autres sont plus spécifiques à certains *frameworks*. Nous présentons ici quelques-uns des méta-formalismes les plus populaires en IDM, nous les retrouverons par la suite dans la partie consacrée aux différents environnements orientés-IDM disponibles (2.3).

a) *Espace technologique des bases de données*

On trouve dans l'espace technologique lié aux bases de données (*BD ware*) le méta-formalisme employé par l'environnement AToM^3 (présenté en 2.3.2.d)) : les diagrammes d'entités-relations (*ER diagrams*). En comparant les diagrammes d'entités-relations au méta-formalisme provenant de l'espace technologique objet, MOF, les analogies sont évidentes : les entités jouent le rôle des méta-classes, tandis que les relations jouent le rôle des méta-associations.

b) *Espace technologique XML*

L'espace technologique XML est couramment employé en IDM, et à travers lui le XML Schema, et, dans une moindre mesure, les DTD. On peut citer plusieurs raisons à cet engouement :

- l'essor de XMI (*XML Metadata Interchange*) qui permet de sérialiser des modèles (provenant par exemple de l'espace technologique objet) sous forme de modèles textuels et balisés XML,
- l'existence de méta-modèles XML pour des langages de programmation, notamment JavML [Badros 2000],

– l'existence de langages de transformation basés sur XML (voir 2.2.3).

c) ***Ecore et EMF***

L'espace technologique objet est, nous l'avons vu en 2.1.2, majoritairement porté par l'OMG avec l'approche MDA et le méta-formalisme MOF. Une des incarnations les plus populaires de MDA est EMF.

Le projet EMF (*Eclipse Modeling Framework*) [Steinberg et al. 2003], et c'est l'une des raisons de sa popularité, permet de réduire le fossé existant entre la modélisation de systèmes, d'applications, de haut niveau d'abstraction, et leur implémentation, de plus bas niveau (programmation). Ceci se fait notamment par l' « unification », au sein de l'environnement Eclipse, de Java, de XML et d'UML autour d'EMF. Eclipse et EMF peut-être vu comme un lieu où convergent les chercheurs et les programeurs, une sorte de « juste milieu » entre modélisation et programmation.

Au cœur de tout cela se trouve Ecore, le méta-formalisme sous-jacent d'EMF. Ecore est une implémentation concrète de MOF 1.4 (et particulièrement de son sous-ensemble EMOF) qui permet de représenter les modèles EMF. Ecore est lui-même à la fois un modèle et un méta-modèle EMF, et c'est aussi son propre méta-modèle.

Par conséquent, Ecore est lui-aussi étroitement lié à UML : n'oublions pas que MOF et UML partagent un cœur commun (2.1.2.b)). De la même manière qu'il est possible avec EMF de définir un modèle Java, et d'en obtenir automatiquement une représentation en XMI, il est possible de définir un modèle Ecore et d'en obtenir une représentation Java.

EMF accepte pleinement (depuis peu) le langage de contraintes OCL, qui permet d'enrichir considérablement des méta-modèles Ecore, et d'effectuer des tests de contraintes sur des instances de ces derniers.

La Figure II-5 présente une vue simplifiée du méta-modèle Ecore. On y voit que les éléments principaux sont les classes, et qu'elles peuvent être composées d'attributs (possédant un type) d'une part, et /ou de références vers d'autres classes d'autre part. En cas de l'utilisation d'Ecore comme méta-formalisme (donc pour créer des méta-modèles), on parlera alors de méta-classes, de méta-attributs…etc.

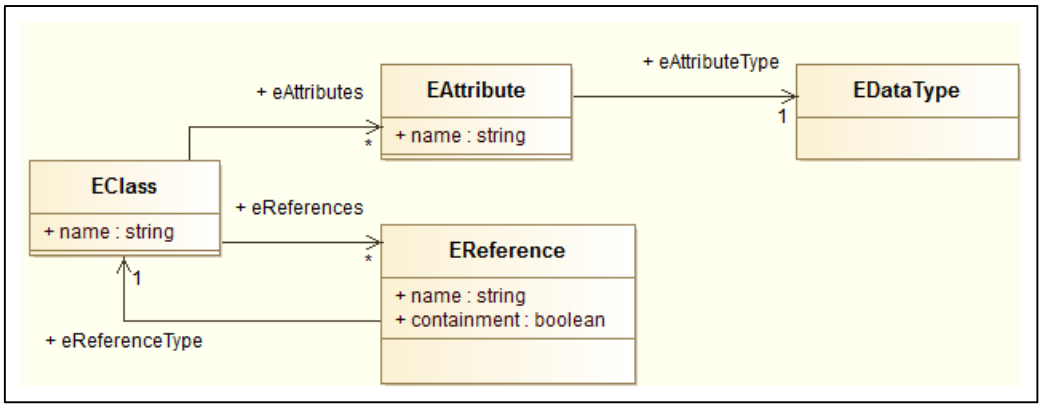


Figure II-5 : Méta-modèle simplifié Ecore inspiré de [Steinberg et al. 2003]

d) ***Autres méta-formalismes de l'espace technologique objet***

Il existe à ce jour plusieurs autres méta-formalismes situés dans l'espace technique des objets, citons par exemple :

- Le méta-formalisme utilisé par l'environnement GME (*Generic Modeling Environment*) [Sztipanovits et al. 1997] présenté plus loin dans ce document (2.3.2.d)). Ce dernier est très proche d'UML, il se situe donc lui aussi dans l'espace technique objet ;
- Le méta-formalisme open-source Kermeta [Muller et al. 2005] [Fleurey 2006], basé sur EMOF et disponible sous forme de *plugin* dans l'environnement EMF. Kermeta est capable à la fois de créer la syntaxe abstraite d'un langage de modélisation (donc un méta-modèle) mais aussi de lui donner une sémantique opérationnelle (le comportement des modèles qui en sont issus). En d'autres termes, Kermeta est ce qu'on appelle un langage de méta-modélisation exécutable (ou langage d'action, ou langage de méta-programmation objet) : il permet de décrire des méta-modèles dont les modèles sont exécutables. Nous ferons référence à nouveau à Kermeta dans la suite de ce document lorsque nous aborderons les langages de transformation basés sur l'approche relationnelle ;
- Le méta-formalisme KM3 (Kernel Meta Meta Model) [Jouault et al. 2006a], disponible lui aussi pour l'environnement Eclipse. La syntaxe abstraite de KM3 est, comme celle de Kermeta, exprimée en termes de Ecore (et donc de EMOF) ;
- Le méta-formalisme XCORE [Clark et al. 2008], basé sur MOF, utilisé par XMF-Mosaic [Clark et al. 2008] conjointement avec l'outil de spécification de contraintes XOCL.

2.2. **Transformation de modèles**

Un modèle ne peut passer du stade contemplatif au stade productif qu'à condition qu'on lui applique une ou plusieurs transformations : la transformation de modèles est un élément crucial de l'IDM [Sendall et al. 2003].

Dans un contexte IDM, on qualifie de transformation de modèles tout programme dont les entrées et les sorties sont des modèles. On parle également de « modèle source » et de « modèle cible ». Selon qu'une transformation produise en sortie un modèle ou du code, elle sera qualifiée de « Model To Model » (« M2M ») ou « Model To Text » (« M2T »). Nuançons cependant cette définition, car d'un point de vue rigoureux, en IDM, « tout est modèle », le texte d'un code est donc quand même un modèle. On utilisera toutefois ces termes dans ce manuscrit, car ils sont aujourd'hui largement employés dans la littérature dédiée.

Abordée sous un angle IDM, une transformation de modèles est elle-même un modèle, certes particulier, mais qui se conforme quand même à un méta-modèle de langage de transformation, ce dernier se conformant au méta-formalisme de l'architecture choisie. Par exemple, une transformation XSLT se conformera à un schéma XML, une transformation QVT se conformera à MOF.

Les modèles source et cible peuvent appartenir au même espace technique, mais ceci n'est absolument pas une obligation : dans le cas où les modèles proviendraient d'espaces techniques différents, il suffit de disposer d'outils permettant de prendre en compte les deux espaces technologiques. La transformation peut, quant à elle, être indifféremment définie et exécutée dans l'espace technologique du modèle source, ou dans celui du modèle cible.

L'immense majorité des langages de transformations de modèles disponibles à ce jour sont des langages textuels : ceci constitue un des grands paradoxes de l'IDM, qui d'une part encourage largement l'emploi de modèles dont une partie est spécifiée graphiquement, alors que cette même IDM utilise principalement des langages textuels pour transformer ces mêmes modèles.

Il existe de très nombreux outils IDM liés à la transformation de modèles, c'est-à-dire autorisant l'emploi d'un langage de transformation de modèles : ces outils sont généralement proposés dans des *frameworks* IDM qui incluent d'autres types d'outils. Nous nous concentrons dans cette partie sur la nature des transformations (vis-à-vis du niveau d'abstraction, de l'espace technologique), sur leurs caractéristiques, sur la technique de transformation sous-jacente (transformation de graphes, d'arbres, manipulation directe…), et sur les artefacts qu'elles produisent (« ModelToModel » ou « ModelToText »).

2.2.1. Caractérisation des transformations de modèles

Pour ces définitions, nous nous inspirons notamment de la taxonomie des transformations de modèles proposée dans [Mens et al. 2006]. Dans la littérature anglo-saxonne, on parle souvent de *transformation mapping* pour désigner l'action d'établir des correspondances entre modèles sources et cible.

a) *Contexte historique*

La question de la transformation a toujours occupé une grande place dans l'histoire de l'informatique, le processus de calcul informatique pouvant lui-même être considéré comme une transformation.

La transformation de modèles, sous-discipline relativement jeune de l'IDM, hérite du savoir-faire des approches existantes, en particulier la transformation de programmes [Partsch et al. 1983]. Les deux approches sont tellement voisines qu'elles se confondent parfois. Ce qui les différencie réside plus dans leurs exigences respectives, et dans le type d'objets transformés que dans leur fonctionnement. Nous verrons du reste que bien des approches de transformation de modèles se basent sur l'établissement de correspondances, de schémas : on peut considérer ceci comme l'équivalent « modélisation », de plus haut niveau d'abstraction, des transformations de bas niveau basées sur les expressions régulières (par exemple la célèbre commande *awk*).

La frontière entre transformation de programmes et de modèles reste donc très mince. Il y a consensus pour dire que lorsque l'objet de la transformation porte sur des métadonnées (par exemple des modèles) plutôt que sur des données brutes, on entre alors dans le domaine de la transformation de modèles d'un point de vue IDM.

Nous préférons, dans un souci d'homogénéité, employer systématiquement le terme de « transformation de modèles » dans nos travaux, tout en gardant à l'esprit que ce terme peut parfois, dans un contexte IDM, concerner du code des deux côtés de la transformation.

b) ***Définition de base***

Dans [Kleppe et al. 2004] se trouve une définition précise de la transformation de modèles, dans laquelle chacun des termes importants est à son tour défini.

*"A **transformation** is the automatic generation of a target model from a source model, according to a **transformation definition**. A **transformation definition** is a set of **transformation rules** that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language".*

(« Une **transformation** est la génération automatique d'un modèle cible à partir d'un modèle source, suivant une **définition de transformation**. Une **définition de transformation** est un assortiment de **règles de transformation** qui décrivent ensemble comment un modèle du langage source peut être transformé en un modèle dans le langage cible. Une **règle de transformation** est la description de comment un ou plusieurs concepts du langage source peuvent être transformés en un ou plusieurs concepts du langage cible » - traduction personnelle)

Cette définition montre bien la différence entre la *définition* d'une transformation (ensemble statique de règles de transformation) et l'*exécution* de cette transformation qui *génère*, *produit*, un artefact (un modèle).

[Mens et al. 2006] suggèrent également que cette définition soit étendue et généralisée, dans le sens où une transformation de modèles devrait aussi être possible avec plusieurs modèles source et/ou plusieurs modèles cibles. Un exemple du premier cas (relation de type « *many to one* ») serait une fusion de modèles (comme lorsqu'on veut combiner plusieurs modèles développés parallèlement pour produire un seul modèle cible). Un exemple du second cas (relation de type « *one to many* ») serait une transformation qui créerait, à partir d'un modèle indépendant de toute plateforme (PIM), plusieurs modèles spécifiques à certaines plateformes (PSM), voire plusieurs artefacts de code.

Nous sommes d'accord avec cette proposition mais pensons également que doit apparaître la notion d'*espace technique* pour une transformation. Nous proposons finalement le changement suivant dans le début de la définition ci-dessus :

« Une **transformation** est la génération automatique, d'un *ou plusieurs* modèle*(s)* cible à partir d'un *ou plusieurs* modèle*(s)* source, appartenant ou pas au même espace technologique, suivant une **définition de transformation** […] »

c) ***Transformation type***

Pour effectuer une transformation de modèles, il est nécessaire de disposer des méta-modèles source et cible, ainsi que d'un modèle typé dans le méta-modèle source. Les modèles source et cible (et donc leurs méta-modèles) ne sont pas obligatoirement situés dans le même espace technique (et ne partagent donc pas le même méta-formalisme) : la définition de la

transformation se fait alors dans un langage dont le méta-modèle se conforme soit au méta-formalisme source, soit au méta-formalisme cible.

Une transformation est définie (spécifiée) au niveau M2 : celui des méta-modèles. En revanche, son exécution, impacte le niveau des modèles M1 puisque l'on applique les règles définies en M2 sur une « instance » située en M1 du méta-modèle source : c'est également à ce niveau-là que se situe le ou les modèle(s) cible généré(s). Le terme général « transformation » désigne le fait de définir (au niveau M2) puis d'appliquer (au niveau M1) ces règles.

Une transformation peut être vue d'un point de vue plus formel comme une fonction, une application, qui, à des éléments appartenant un ensemble de départ, associe des éléments d'un ensemble d'arrivée.

Nous montrons sur la Figure II-6 un exemple de transformation type dans laquelle est généré un modèle *My* (conforme à son méta-modèle *MMY*) à partir d'un modèle *Mx* (conforme à son méta-modèle *MMX*) et de règles de transformations établies au niveau M2 et exécutées en M1. Tout comme sur la Figure II-3 la lettre χ désigne la relation de conformation.

Une transformation est, elle aussi, conforme à un méta-formalisme, qui peut être celui auquel se conforme le méta-modèle source ou le méta-modèle cible. Par exemple, transformer un document XML en un diagramme UML peut se faire soit dans l'espace technologique XML (il faut alors utiliser XQuery ou XSLT pour décrire la transformation) ou bien dans l'espace technologique MDA, avec un langage implémentant MOF QVT, par exemple. Il est par contre nécessaire que l'outil que l'on utilise supporte les deux espaces technologiques avec lesquels on travaille (mécanisme d'import et export de fichiers).

Note : On dit en outre qu'une transformation est *réversible* s'il existe une transformation permettant de retrouver le modèle source à partir du modèle cible. La relation qui lie une transformation et la transformation inverse qui lui correspond est analogue à celle qui lie, en mathématiques, une fonction à sa réciproque.

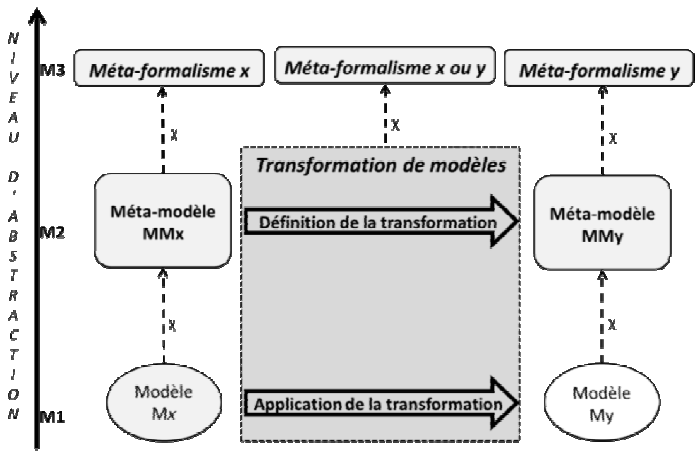


Figure II-6 : Transformation de modèles type

d) ***Endogène vs. exogène***

On qualifie une transformation de modèles d'*exogène* lorsque les méta-modèles source et cible sont différents, en d'autres termes lorsque l'on veut produire un modèle cible exprimé

dans un formalisme différent que celui dans lequel est exprimé le modèle source. Par exemple, la Figure II-6 montre une transformation exogène.

Inversement, une transformation dans laquelle le modèle source et le modèle cible possèdent le même méta-modèle est qualifiée d'*endogène*. Les règles de transformation sont définies sur un unique méta-modèle. La Figure II-7 montre une transformation endogène : un modèle *Mx* est transformé en un modèle *Mx'*, tous deux sont conformes au méta-modèle *MMx*.

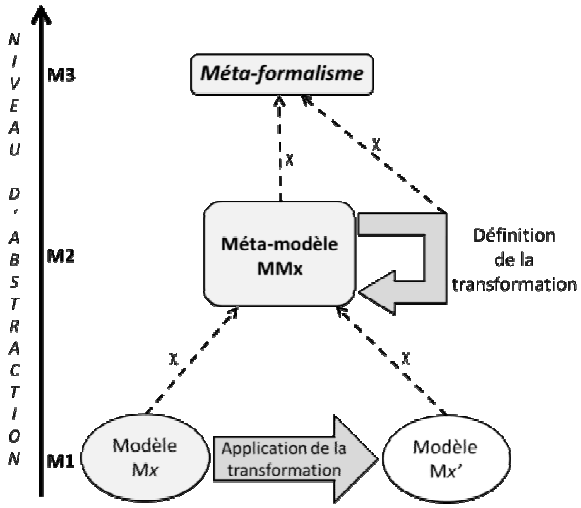


Figure II-7 : Transformation de modèles endogène

On a recours à des transformations endogènes dans le cadre, par exemple, d'une optimisation de modèles, ou bien d'une simplification, ou encore de *refactoring*.

On utilise des transformations exogènes pour toute transformation de modèle impliquant des méta-modèles différents, l'exemple le plus significatif étant sans aucun doute la génération de code. On fait également appel aux transformations exogènes pour effectuer des migrations de modèles (depuis une plateforme vers une autre) tout en restant au même niveau d'abstraction, ou encore dans opérations de *reverse engineering* (rétro-ingénierie ou rétro-conception).

Une variante de transformation endogène est une transformation « sur place » ou « de mise à jour » : dans ce cas, le modèle source et le modèle cible sont identiques (et se partagent le même méta-modèle). Il n'y a donc pas de création de modèle cible. La transformation appartient à un langage dont le méta-modèle se conforme au même méta-formalisme, et par conséquent le même espace technique, que le méta-modèle *MMx*.

Note : En transformation de programmes, on parle plutôt de *rephrasing* (« reformulation ») pour désigner une transformation endogène, ou de *translation* pour désigner une transformation exogène [Visser 2004].

e) ***Verticalité vs. horizontalité***

Une transformation de modèles, outre le fait qu'elle puisse intervenir entre deux méta-modèles distincts, peut également mettre en jeu des niveaux d'abstraction différents. Si c'est le cas, on parlera de transformation *verticale*. En l'absence de changement de niveau d'abstraction, on qualifiera au contraire la transformation *d'horizontale*.

Cette notion de niveaux d'abstraction ne doit surtout pas être confondue avec celle des « niveaux méta » de l'IDM : une transformation de modèles verticale est toujours définie en M2 et est exécutée sur un modèle de M1 pour produire un autre modèle de M1 ! Mais deux modèles situés au niveau M1 et impliqués dans une transformation peuvent appartenir à des niveaux d'abstraction très différents, d'un point de vue de processus de développement. Par exemple, le modèle source est un modèle générique, le modèle de destination est un modèle de code spécifique à la plateforme Java : la génération de code est une transformation verticale.

Le Tableau II-1 résume les utilisations possibles des transformations selon leur aspect vertical/horizontal et endogène/exogène.

	Verticale	Horizontale
Endogène	Raffinement	Simplification Restructuration Normalisation
Exogène	Génération de code Rétro-ingénierie PIM vers PSM	Fusion de modèles Migration

Tableau II-1 : Synthèse des transformations possibles en IDM

2.2.2. Règles de transformation

Après avoir considéré les transformations dans leur ensemble, « descendons » à présent d'un niveau d'abstraction pour considérer la structure même de ces transformations. Une transformation se compose de règles : chacune d'elles établit une correspondance entre un élément du modèle source et un élément concept du modèle cible. La somme de ces correspondances constitue la définition de la transformation (que nous avons vue en 2.2.1.b)).

a) *Structure interne*

Chaque règle de transformation comporte un membre gauche et un membre droit que l'on abrège souvent en « LHS » (*Left-Hand Side*) et « RHS » (*Right-Hand Side)*. Schématiquement, le LHS accède au modèle source, tandis que le RHS agit dans le modèle cible.

La nature de ces membres est variable [Czarnecki et al. 2003], elle contient usuellement une combinaison des trois éléments suivants :

- **Variables**. Elles peuvent dépendre du modèle source, du modèle cible, voire d'éléments intermédiaires éventuels ;
- **Patterns** (ou « patrons »). Ce sont des fragments de modèles, avec ou sans variables. Il existe plusieurs sortes de patrons, pouvant porter par exemple sur des chaînes de caractères (dans le cas de *templates* textuels, i.e. de gabarits de conception), des termes ou des graphes (ces deux derniers étant souvent utilisés pour des transformations « M2M »). Ces patrons sont définis au moyen de la syntaxe concrète ou abstraite des modèles source et destination et peuvent être textuels et/ou graphiques ;

– **Expressions logiques**. C'est le fait d'exprimer des contraintes ou de décrire des calculs dans les modèles source et cible. Les expressions logiques peuvent être exécutables de manière déclarative, (avec par exemple des expressions OCL chargées de récupérer des éléments source), ou impérative, (au moyen de code informatique, d'une API, qui manipule les modèles directement, par l'ajout, la suppression et la modification d'éléments) ou non exécutables (décrit simplement une relation entre les modèles). Certains langages comme ATL permettent la spécification d'expressions de manière déclarative et/ou impérative.

La Figure II-8 résume le contenu d'une règle de transformation au moyen d'un diagramme de caractéristiques [Kang et al. 1990].

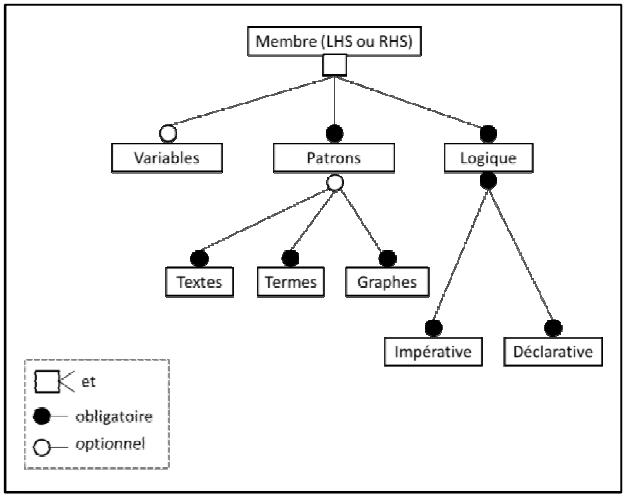


Figure II-8 : Structure d'une règle de transformation

b) ***Aspects remarquables***

Il existe plusieurs caractéristiques importantes pour les règles de transformation.

L'aspect déclaratif/impératif. L'aspect déclaratif ou impératif d'une approche de transformation est un élément clef lorsque l'on veut la comparer à d'autres approches. Une approche déclarative a pour but de traiter l'aspect « Quoi ? Que ? » c'est-à-dire répondre à la question « Qu'a-t-on besoin de transformer en quoi ? ».

L'approche impérative, quant à elle, traite l'aspect « Comment ? » c'est-à-dire de la manière dont la transformation elle-même doit être faite. L'approche déclarative est plus intuitive et moins « lourde » à gérer, elle est syntaxiquement simple, et repose sur des fondements mathématiques.

L'ordre d'application des règles d'une transformation déclarative est implicite, et le parcours du modèle source ainsi que la création dans le modèle cible le sont également. L'approche impérative devient pourtant très utile pour certaines approches de type « many to many » impliquant des modèles complexes. C'est pourquoi l'OMG préconise des langages « hybrides » (voir 2.2.4) pouvant supporter des combinaisons de règles déclaratives et de règles impératives [Gardner et al. 2003] : QVT est l'exemple-type d'un tel langage hybride. Une telle combinaison peut par exemple prendre la forme de règles impératives encadrées par

des pré-conditions et des post-conditions déclaratives [Jézéquel 2008]. Nous faisons nous-même le choix de l'approche hybride pour implémenter nos transformations M2M.

La directionnalité. À la base, une règle de transformation est unidirectionnelle : sa source et sa cible sont deux éléments distincts. Une règle bidirectionnelle peut, quant à elle, être exécutée dans les deux sens, c'est-à-dire que chaque modèle joue à la fois le rôle de source et de cible. Une transformation bidirectionnelle [Akehurst et al. 2002] peut se définir au moyen d'une seule règle bidirectionnelle ou de deux règles unidirectionnelles. Ce type de transformation est très utilisé pour la synchronisation de modèles. La bidirectionnalité ne doit pas être confondue avec la réversibilité (voir 2.2.1.c).

Le paramétrage. Les règles de transformation peuvent posséder des paramètres de contrôle additionnels permettant une configuration plus précise.

L'appel aux structures intermédiaires. La construction d'une structure intermédiaire se révèle particulièrement utile lors d'une transformation de mise à jour (absence de modèle cible). Des approches par transformation de graphes comme GreAT [Agrawal et al. 2003] et VIATRA [Varro et al. 2002] (*VIsual Automated model TRAnsformations*), utilisent ces structures intermédiaires.

La séparation syntaxique. Les membres gauches et droits d'une règle peuvent être désignés explicitement ou pas dans la syntaxe de la règle.

Les liens de traçabilité. Cela consiste à enregistrer les liens entre les éléments source et cible. Ces liens sont utiles en pour analyser l'impact de la transformation (comment un changement dans un modèle affecte d'autres modèles liés), pour effectuer des synchronisations entre modèles ou du débogage.

La stratégie d'application. Une règle s'exécute habituellement en fonction de son champ d'application source. Parfois, il peut exister plus d'une correspondance au niveau de la source d'une règle : il est alors nécessaire de disposer d'une stratégie d'application. Cette stratégie peut être déterministe, non-déterministe, ou même interactive (environnement XDE, voir 0263). On parle aussi d'application implicite ou explicite.

La planification (ordonnancement) des règles. Elle définit à quel moment une règle est exécutée, et donc l'ordre d'exécution pour une suite de règles. Cette planification peut :

- être exprimée implicitement ou explicitement. Une planification explicite peut être interne ou externe. Elle est externe lorsqu'il existe une séparation claire entre les règles de transformation et la logique de planification (par exemple dans VIATRA, qui utilise pour ce faire des automates à états finis). Elle est interne lorsqu'il existe, au sein d'une règle, un mécanisme permettant d'invoquer d'autres règles, ce qui est souvent le cas dans une approche par *templates* (par exemple, dans MOF QVT[1] ou dans Jamda, voir 0) ;
- proposer la sélection de règles, et ce au moyen d'une condition explicite (Jamda), d'un choix non-déterministe (comme le permet BOTL [Braun et al. 2003] [Marschall et al. 2003])). Cette sélection peut être interactive (XDE, voir ci-dessus « stratégie d'application »). Une variante de la sélection de

[1] http://www.omg.org/spec/QVT/1.1/PDF

règles est de proposer l'itération sur les règles via des mécanismes de récursivité, de boucles, et de structures de type *do…while* ;

- Correspondre à un processus organisé en phases. Cela signifie que certaines règles ne peuvent être invoquées que dans certaines phases, chaque phase ayant un but spécifique dans la transformation. Par exemple, des approches orientées-structure comme OptimalJ (voir 0) séparent la phase de création de hiérarchie dans le modèle cible, et celle de son peuplement (attributs, références).

La composition de règles. Il existe plusieurs moyens de créer des compositions de règles, c'est-à-dire créer des sortes de structures de règles. Elles peuvent bénéficier par exemple de mécanismes de modularité comme dans VIATRA, qui permettent de grouper des règles à l'intérieur de modules. Des mécanismes plus avancés de réutilisabilité permettent de définir une règle en la basant sur une ou plusieurs règles : héritage, spécialisation, héritage de modules, composition logique. Enfin, il est possible d'organiser les règles selon la structure de la source, de la cible, ou de les organiser de manière complètement indépendante.

2.2.3. **Approches pour la transformation de modèles**

Nous classons ces approches en deux grandes familles : les approches de modèle à modèle ou M2M, et les approches de modèle à texte ou M2T.

Selon l'IDM, une approche M2T ne devrait pas se distinguer d'une approche M2M, le code étant assimilé, dans l'absolu, à un modèle. D'ailleurs, une transformation vers du code implique théoriquement que l'on fournisse comme cible le méta-modèle d'un langage de programmation.

Pourtant, dans la pratique, il est courant de s'affranchir de cette contrainte technique. Il est dans bien des cas plus simple et plus rapide de travailler sur un squelette de code composé de parties fixes et variables, que de spécifier des règles en fonction d'un méta-modèle cible de langage. En effet, il faudrait pour cela disposer d'un tel méta-modèle, et de surcroît qu'il soit exprimé dans le même méta-formalisme que le méta-modèle source. Dans le cas contraire, il serait nécessaire de « recréer » ce méta-modèle à partir de la grammaire du langage. Très souvent donc, en M2T le code est tout simplement généré sous forme de texte, dans un fichier, à partir d'un modèle et d'un méta-modèle source et de règles de transformation. Ce fichier de code généré est ensuite directement utilisable par un compilateur ou un interpréteur. C'est ainsi que nous avons procédé pour transformer des modèles DEVS en code dans le chapitre VI. C'est pourquoi ce genre de transformation est appelée « Model To Text ».

Les notions abordées précédemment concernant les aspects remarquables des transformations vont nous servir à présent pour illustrer cet état de l'art sur les approches de transformation de modèles existantes.

Note : Certaines approches présentées ici sont intégrées dans des outils présentés dans la section consacrée au panorama des approches IDM (2.3).

a) ***Vue d'ensemble***

Le Tableau II-2 résume les différentes approches que nous allons explorer dans cette partie.

Approches de transformation	
Transformation M2M	**Transformation M2T**
dirigée par la structure	parcours de modèles (programmation)
manipulation directe	
approche relationnelle	
transformation de graphes	parseurs existants (XSLT)
template	
opérationnelle	template
parseurs existants (XSLT)	
hybride	

Tableau II-2 : Les familles M2M et M2T

Dans la famille M2T, on trouve les approches par parcours de modèle au moyen de langages de programmation classiques (on les rencontre parfois dans la littérature sous l'appellation *visitor-based*, littéralement « basées sur le principe du visiteur »), les approches *template-based* [Cleaveland 2001] (littéralement « basées sur les gabarits/patrons », mais nous préférons utiliser le terme « par template » car c'est le plus couramment employé), et les approches par basées sur des parseurs existants (typiquement, XSLT), qui sont également présentes dans les approches M2M. Dans la famille M2M, on trouve les approches : par manipulation directe, relationnelles, dirigées par la structure, basées sur la transformation de graphes, par template, opérationnelles et enfin hybrides (mêlant aspects déclaratifs et impératifs). Nous présentons à présent les transformations utilisant les parseurs, car elles sont communes au monde du M2M et du M2T.

Nous considérons cette méthode de transformation de modèles comme faisant partie de l'IDM car même si elle repose sur l'analyse syntaxique, cette dernière se fait à un niveau d'abstraction plus élevé que le simple texte : le niveau des modèles. La transformation de modèles avec les parseurs est utilisée dans l'espace technologique de XML : elle est réalisée au moyen du langage XSLT, ou XQuery, qui se conforment au méta-formalisme XML Schema. Un modèle XML, ou XMI, passé en entrée est transformé en un autre document XML ou XMI : ce type de transformation est donc à la fois M2M et M2T, car un modèle XML est aussi un fichier de texte. La génération de code avec des parseurs connaît toutefois plusieurs limitations : fable capacité d'adaptation, mauvaise lisibilité des transformations (beaucoup d'informations textuelles)…

b) ***M2T***

- ***Génération de code par parcours de modèle.***

Il s'agit d'une des approches de génération de code les plus basiques : elle consiste à parcourir la représentation interne du modèle donné en entrée et à écrire du texte (code) sur un flux de sortie. Un des rares exemples (à notre connaissance) d'implémentation d'une telle approche est l'environnement orienté-objet Jamda, dans lequel les modèles UML sont représentés par un ensemble de classes, une API manipule les modèles, et un outil génère le code.

- ***Génération de code par template : solution préconisée par l'OMG***

Il s'agit de l'approche M2T la plus employée. Elle consiste à utiliser comme RHS un texte fixe dans lequel certaines parties sont variables : elles sont renseignées en fonction des

informations récupérées dans le modèle source (LHS), et peuvent notamment faire l'objet d'itérations (enchaînement de structures de contrôle de type *if…else* par exemple). Il n'y a pas de séparation syntaxique entre LHS et RHS. La plupart de langages de transformation par template offrent la possibilité d'appeler un template à partir d'un autre, voire de créer des liens d'héritage entre templates (ce qui permet une grande réutilisabilité des règles).

Comme le membre droit d'un template concerne du texte, il n'y a pas de notion de typage, ce qui peut présenter un inconvénient (erreurs de typage lors de l'interprétation, etc…). L'avantage d'une telle approche découle de cet inconvénient : l'approche par template est extrêmement souple et facilite la génération de n'importe quel artefact textuel. Même si le code est l'artefact phare des approches IDM, nous avons vu que suite à des transformations d'autres artefacts pouvaient être générés : par exemple, l'approche M2T par template permet de générer de la documentation sur les modèles.

La spécification de l'OMG nommée « MOFM2T » (*MOF Model To Text Transformation Language*)[1] décrit un standard aligné avec UML 2.0, MOF 2.0 et OCL 2.0 permettant de générer du code à partir de templates. Comme MOFM2T n'est qu'une spécification, un standard, et non un langage concret, elle a été implémentée au sein de divers outils IDM/MDA : une majorité d'entre eux supporte désormais la génération de code par template. La plupart de ces implémentations sont disponibles sous forme de plugins sur la plateforme EMF, et font partie du projet Eclipse « M2T »[2].

Parmi les plus utilisées actuellement, citons :

- JET,
- Xpand,
- Acceleo.

Ces trois implémentations sont pratiquement équivalentes, et il existe à ce jour très peu de retours comparatifs les concernant. Il semblerait néanmoins qu'Acceleo soit légèrement supérieur à ses « concurrents » en termes de facilité de prise en main, de performances[3], de niveau d'activité de la communauté d'utilisateurs et surtout de compatibilité avec les standards de l'OMG (OCL notamment).

Pour compléter cette liste, citons un des « ancêtres » de ces langages : MOFScript. Il n'appartient pas au projet M2T mais au projet GMT (*Generative Modeling Technologies*)[4] et son utilisation semble décliner depuis quelques années.

La Figure II-9 montre une capture d'écran faite en deux temps, montrant une partie d'un template créé sous Eclipse avec le plugin Acceleo (dans sa version 3.2). Ce template prévoit de générer un fichier et l'en-tête de celui-ci. Dans un premier temps (1) le template (première ligne surlignée) est défini, son exécution est soumise à une condition : le modèle passé en LHS doit être le modèle racine. L'exécution de ce template conduit à la création d'un fichier composé du nom du modèle auquel est concaténée la chaîne « Model.py » (ce sera

[1] http://www.omg.org/spec/MOFM2T/1.0
[2] http://www.eclipse.org/modeling/m2t
[3] http://eclipsemde.blogspot.fr/2010/12/acceleo-vs-acceleo-vs-xpand.html
[4] http://www.eclipse.org/gmt

donc un fichier Python), et d'un en-tête de fichier comportant le même nom (2 dernières lignes surlignées). La partie (2) montre le résultat de l'exécution de ce template : le fichier qui a été généré comporte bien la chaîne « Model.py », l'en-tête également. La majeure partie du « squelette » du texte est identique, (1) et (2) ne diffèrent que par quelques caractères : une des caractéristiques du template est qu'il ressemble beaucoup au code généré.

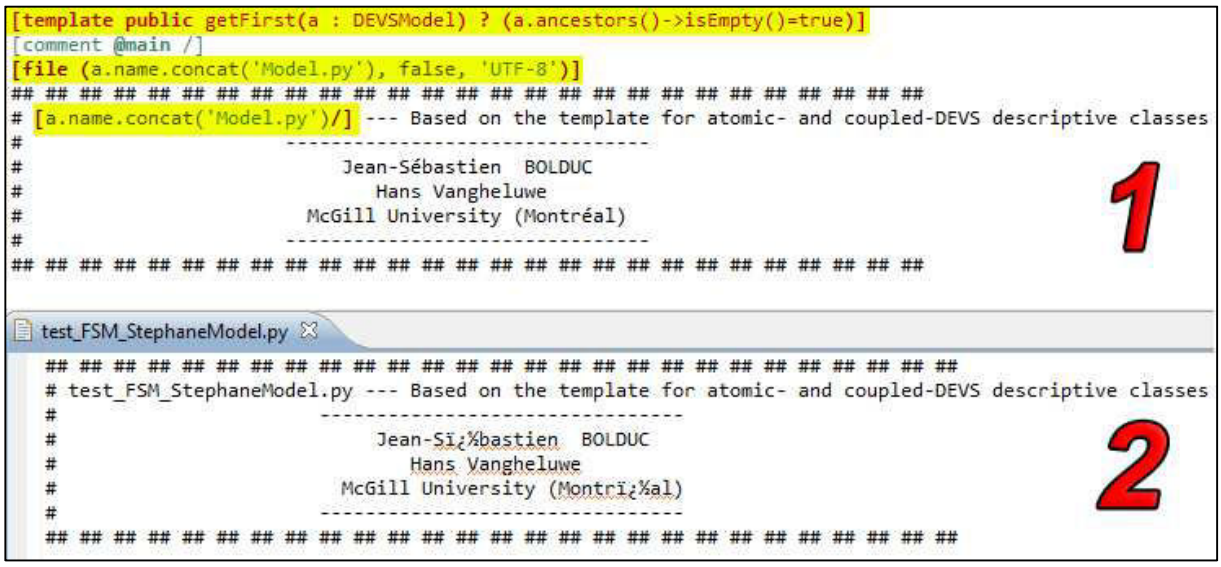


Figure II-9 : Approche par template avec Acceleo: définition et exécution

c) ***M2M***

- ***Approche par manipulation directe***

Ce type d'approche de transformation de modèle offre une représentation interne d'un modèle et des API pour le manipuler (par exemple JMI). L'implémentation se fait usuellement dans un environnement orienté-objet, ce qui fournit une petite infrastructure pour organiser les transformations (classes abstraites pour spécifier des transformations par exemple). Très souvent il incombe aux utilisateurs d'implémenter eux-mêmes leurs règles, très souvent aussi à partir de zéro, en utilisant un langage de programmation classique. Jamda utilise cette approche (qui est similaire dans son principe à l'approche M2T par parcours de modèle). L'approche par manipulation directe est de type impératif : elle est située à un bas niveau d'abstraction.

- ***Approche opérationnelle***

Cette approche, également d'aspect impératif, est une variante de la manipulation directe, mais elle se situe à un niveau d'abstraction un peu plus élevé. Une solution courante dans ce type d'approche est d'étendre le méta-formalisme avec un langage d'expression de requêtes : par exemple, utiliser un langage de type OCL, étendu de manière à être capable de spécifier une sémantique opérationnelle au moyen de la programmation impérative, couplé au méta-formalisme MOF. Cette approche se retrouve dans des langages tels que *Model Transformation Language* (MTL) [Vojtisek et al. 2005] et QVT *Operationnal*. Le méta-formalisme exécutable Kermeta [Fleurey 2006] s'inscrit lui aussi en partie dans cette approche opérationnelle, il constitue une extension d'Ecore (et donc de EMOF) d'une part et intègre des fonctionnalités de transformation d'autre part (prise en compte des aspects et mécanismes de parcours de modèle pour les transformations M2M).

- ***Approche par template***

Cette approche, très utilisée en M2T, existe également dans une moindre mesure au niveau du M2M. Un template de modèle est un modèle comportant du méta-code intégré dont le but est d'évaluer, de calculer, les parties variables des instances correspondant à ce modèle. Souvent, ces templates sont exprimés dans la syntaxe concrète du langage cible, et le méta-code utilisé est en OCL. Un exemple de cette approche appliqué à UML est donné dans [Czarnecki et al. 2005].

- ***Approche relationnelle***

Peuvent être qualifiées de relationnelles toutes les transformations de type déclaratif reposant sur les relations (au sens mathématique du terme). Citons parmi elles MOF QVT *Relations*, *Model Transformation Framework* (MTF, open-source) [Griffin 2004], *Kent Model Transformation Language* (KMTF) [Akehurst et al. 2005], le plugin Eclipse Tefkat[1] [Lawley et al. 2005], et l'outil XMF-Mosaic.

L'idée directrice d'une transformation par approche relationnelle est de spécifier à l'aide de contraintes des liens entre la source et la cible d'une relation. Par définition, une telle transformation n'est pas exécutable telle quelle, mais son exécution est possible par l'adjonction d'une sémantique exécutable, par exemple au moyen de la programmation logique. Il n'y a pas d'effets de bord avec l'approche relationnelle, contrairement à l'approche par manipulation directe vue en c). La plupart des transformations basées sur l'approche relationnelle permettent la multi-directionnalité des règles, et fournissent des liens de traçabilité. Elles n'autorisent pas la transformation sur place (source et cible doivent être distinctes).

Dans [Gerber et al. 2002] sont expérimentées des transformations par approche relationnelle dont la sémantique exécutable est exprimée au moyen de Mercury (un dialecte de Prolog) [Somogyi et al. 1996] et F-logic (un paradigme logique orienté-objet) [Kifer et al. 1995].

- ***Approche dirigée par la structure***

Les approches dirigées par la structure se décomposent en deux phases distinctes :

  - La première consiste à créer la structure du modèle cible,
  - La seconde va renseigner les attributs et les références ;

L'utilisateur doit uniquement fournir les règles de transformation. Ensuite, les stratégies d'application et d'ordonnancement des règles sont déterminées par l'environnement de transformation. Une des particularités de cette approche est que l'organisation des règles se fait en fonction de la cible : pour chaque type d'élément cible il existe une règle, et l'imbrication des règles correspond à la structure du méta-modèle cible.

OptimalJ (0), implémenté en Java, est un exemple d'un environnement de transformation par structure : la transformation s'obtient en créant des sous-classes qui héritent de classes de transformation génériques, et les règles de transformation proprement

[1] http://tefkat.sourceforge.net

dites prennent la forme de méthodes. Elles n'ont pas d'effets de bord. L'environnement gère automatiquement l'application des règles (ordre, etc…).

Un autre exemple d'approche dirigée par la structure a été proposé en réponse à la RFP de l'OMG pour QVT [RFP-QVT] par la société *Interactive Objects and Project Technology*. Les documents concernant cette approche (fichier PDF « ad/03-08-11 », fichier UML « ad/03-08-12 », archive ZIP « ad/03-08-13 ») ne sont hélas plus disponibles à ce jour dans la base de documents publics de l'OMG.

- ***Approche par transformation de graphes***

Cette approche très populaire repose sur la théorie des graphes : elle a été expérimentée dans de nombreux environnements essentiellement issus du monde de la recherche académique. Une approche par transformation de graphes utilise usuellement des graphes typés, orientés et étiquetés [Andries et al. 1996].

Il existe de nombreuses approches M2M basées sur la transformation de graphes, certaines sont intégrées dans des environnements-outils MIC (voir 2.3.2). Les plus connues sont : **AGG** (*Attributed Graph Grammar*) [Taentzer 2003], l'environnement **VMTS** (*Visual Modeling and Transformation System*) [Levendovszky et al. 2004], l'environnement **AToM^3** (voir 2.3.2.d) [Lara et al. 2002a] [Lara et al. 2002c], **BOTL** (*Bidirectional Object-oriented Transformation Language*) [Braun et al. 2003] [Marschall et al. 2003], la suite d'outils **Fujaba**[1] (*From UML to Java And Back Again*), **GReAT** (*Graph REwriting And Transformation*) [Agrawal et al. 2003], **MOLA** (*MOdel transformation LAnguage*) [Kalnins et al. 2004], **VIATRA** [Varro et al. 2002] (*VIsual Automated model TRAnsformations*) et le langage de transformations graphique **UMLX** [Willink 2003].

Les règles de transformation de graphes possèdent un schéma LHS et un schéma RHS : le schéma LHS est repéré dans le modèle source et remplacé par le schéma RHS dans le modèle cible. GReAT permet d'enrichir ces schémas en y ajoutant des multiplicités.

Les schémas de graphes peuvent être exprimés dans la syntaxe concrète de leurs modèles source ou cible (VIATRA) ou dans la syntaxe abstraite MOF (AGG, BOTL). Utiliser la syntaxe concrète est plus facile (les développeurs travaillant avec un langage de modélisation particulier sont déjà habitués à cette dernière) et souvent plus « concis ». Utiliser la syntaxe abstraite permet en revanche d'exprimer des concepts plus généraux qui pourront s'adapter à plusieurs méta-modèles. L'ordonnancement des règles est automatique (implicite) dans AGG et AToM^3 (les règles sont unidirectionnelles et les transformations sur place) tandis qu'il est explicite (possède une représentation) dans VIATRA (comme nous l'avons vu, avec des automates à états finis), MOLA, Fujaba (qui utilisent tous deux des graphes de flots de contrôle) et GReAT (graphes de flots de données).

Il existe également des approches relationnelles basées sur les transformations de graphes, ce sujet est évoqué entre autres dans [Königs 2005]. Des études comparatives exhaustives de certaines de ces approches par transformation de graphes ont été menées dans [Agrawal 2004]. Dans [Ehrig et al. 2005], les auteurs vont plus loin effectuent une

[1] http://www.fujaba.de

comparaison entre plusieurs de ces environnements et QVT *Core* : ce dernier apparaît comme étant proche des transformations basées sur les graphes.

2.2.4. **Des approches M2M particulières : les approches hybrides**

Ces approches de transformation font partie des plus prometteuses, notamment grâce à leur puissance d'expression. Elles permettent, selon le contexte, de spécifier des règles de manière impérative, déclarative, ou bien de mélanger les deux. Par conséquent, la stratégie d'application de ces règles peut être soit implicite (gérée par l'environnement de transformation) soit explicite (gérée par l'utilisateur). Les approches hybrides se situent *de facto* à cheval sur plusieurs des approches que nous venons de voir et permettent de combiner leurs avantages.

a) ***La spécification QVT de l'OMG***

La spécification standardisée par l'OMG, MOF QVT 1.1 ou plus simplement QVT, est l'exemple le plus connu et le plus théoriquement abouti d'approche hybride [OMG-QVT-2011]. QVT est né suite à une RFP faite en 2002, dans le but de mettre en place un standard compatible avec les recommandations MDA (UML, OCL, MOF..).

QVT agit sur des méta-modèles qui se conforment à MOF 2.0. Chaque transformation QVT est elle-même un modèle, elle se conforme donc au méta-modèle (plus exactement aux méta-modèles) de QVT, qui lui-même se conforme au méta-modèle de MOF 2.0. On dit que QVT est « aligné » sur MOF 2.0.

Tout au long de cet état de l'art consacré aux transformations de modèles, nous avons mentionné à maintes reprises QVT parmi les réponses apportées à la RFP de l'OMG. Nous avons déjà eu un aperçu des caractéristiques qui constituent l'essence même de son caractère hybride. Plus précisément, QVT se compose d'une partie d'une partie déclarative sur deux niveaux qui forme le cadre sur lequel repose la sémantique opérationnelle de la partie impérative (voir Figure II-10). Chaque partie est un méta-modèle à part entière :

- QVT *Core* (proche de l'approche par transformation de graphes) : langage déclaratif simple, créé à la base pour être utilisé comme cible d'une transformation décrite par QVT *Relations*, ou comme la sémantique de cette dernière. QVT prévoit d'ailleurs un passage de QVT *Core* à QVT *Relations* via une transformation : « *Relations to Core Transformation* ». La seule implémentation (approximative) de ce langage a, à notre connaissance, été faite dans l'environnement OptimalJ ;
- QVT *Relations* (approche relationnelle) : langage déclaratif permettant des transformations uni et bidirectionnelles. La sémantique de ce langage est un mélange de langue anglaise et de prédicats de la logique du premier ordre ;
- QVT *Operationnal* (approche opérationnelle) : langage impératif pour décrire des transformations unidirectionnelles.

Nous renvoyons le lecteur vers le document officiel [OMG-QVT-2011] qui décrit le standard QVT de manière très exhaustive.

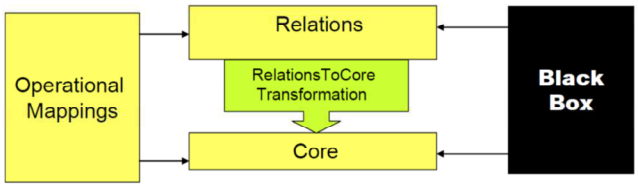


Figure II-10 : Relations entre les méta-modèles de QVT

Note : Il est important de se rappeler que QVT constitue un standard : en tant que tel, ce n'est pas un langage à proprement parler mais un ensemble de spécifications, de contraintes, de caractéristiques que se doit de respecter tout langage se réclamant du standard QVT.

2.3. Panorama des approches et outils IDM

Durant les années 80 et l'essor des langages objet, il existait déjà des approches visant à regrouper des concepts dans des modèles plus ou moins indépendants de toute plateforme. Parmi ces approches, ces méthodes, ancêtres de l'IDM, on peut citer l'*Information Engineering Methodology* ou *Information Engineering* (IE) [Macdonald 1986] [Finkelstein 1992], *Structured Systems Analysis and Design Method* (SSADM) [Robinson et al. 1994], et la méthode séquentielle Merise [Tardieu et al. 1989].

La majorité de ces méthodes, bien que centrées sur les modèles, ne donnaient à ces derniers qu'une dimension contemplative, et ne constituent pas des approches IDM *stricto sensu* car, selon nous, pour qu'une approche puisse être assimilée à de l'IDM, elle doit en plus de manipuler des modèles, définir un cadre précis de création pour ces derniers (méta-modèles) et pouvoir les manipuler (transformations de modèles).

De nos jours, plusieurs approches, méthodes, outils peuvent être assimilés à de l'IDM et il n'est pas chose aisée de les différencier.

En effet, certaines sont des approches assez « anciennes » (programmation générative par exemple) mais sont toujours employées. Elles se sont tout de même peu à peu rapprochées d'autres méthodes, pour, finalement, être aujourd'hui utilisées de manière conjointe, au sein d'outils de développement. On parle parfois d'outils CASE (*Computer-Aided Software Engineering*) pour désigner les outils de génie logiciel qui permettent le contrôle voire l'automatisation d'une ou plusieurs phases du cycle de vie du logiciel. Par exemple, FUJABA et TOPCASED sont des outils CASE.

Le fait que les différentes approches de l'IDM partagent des concepts communs implique aussi que certaines utilisent les mêmes outils (UML…). Il en résulte une sorte de « chevauchement » entre les différentes sous-familles de l'IDM [Agrawal 2004]. Ainsi, *Generative Programming* et *Model Integrated Computing* (MIC) sont, à notre sens, aujourd'hui complémentaires. Elles sont d'ailleurs souvent imbriquées. En effet, plusieurs environnements orientés MIC incluent des outils de génération semi-automatique voire automatique de code. Pourtant, historiquement, il s'agit de deux paradigmes distincts. De même, les approches de type MIC souvent employées pour créer des DSL et/ou intégrées dans des *Software Factories*.

Selon nous, il est important de comprendre que le grand changement de paradigme en génie logiciel [Bézivin et al. 2002a] initié il y a environ douze ans a beaucoup rapproché plusieurs de ces techniques et a contribué à les standardiser.

Il existe un rapport étroit entre les méta-modèles décrivant des DSL et des méta-modèles généraux comme UML : beaucoup d'outils dédiés au génie logiciel permettent de créer des profils UML que l'on peut voir comme des DSL à part entière. C'est ainsi qu'un formalisme tel qu'UML, formalisme *high level* générique par excellence, multi-domaines, est à l'origine de plusieurs DSL, grâce aux profils qui permettent de spécialiser des éléments de son méta-modèle (voir 2.1.2.d)).

L'objectif de cette section est de présenter un état de l'art sur l'IDM d'un point de vue pratique : nous mentionnons ses débuts mais aussi et surtout ses orientations et défis actuels. Pour ce faire, nous nous efforçons de présenter dans leur diversité les outils sur lesquels repose cette discipline du génie logiciel. Ces outils implémentent totalement ou partiellement les notions de modèle et de méta-modèle, vues en 2.1, et les approches de transformation de modèles, vues en 2.2.

Cet état de l'art sera conclu par un tableau récapitulatif de différents environnements, langages et approches de transformation disponibles, classés selon différents critères. Cette classification sera ensuite utilisée dans ce travail pour justifier certains de nos choix.

Outre les travaux déjà cités dans cette introduction, notre état de l'art s'appuie en particulier sur les travaux de [Diaw et al. 2010] [Czarnecki et al. 2006] [Mens et al. 2006] ainsi que sur la documentation du site de l'OMG.

2.3.1. Generative Programming

La *Generative Programming* (ou programmation générative) [Agrawal 2004] [Czarnecki et al 1999] est dédiée à l'automatisation totale ou partielle du processus de codage.

a) *Principe*

À l'origine, la programmation générative regroupait des techniques de programmation relatives à la génération automatique (totale ou partielle) de code, à partir de modèles, de *templates*, et bien entendu de générateurs de code. Par exemple, la programmation générative permet d'automatiser le codage pour des logiciels appartenant à une même famille. Le parallèle peut être établi avec une usine fabriquant plusieurs produits d'une même gamme : certaines pièces sont communes à tous ces produits et leur montage peut être automatisé. Ce type de programmation peut s'implémenter de plusieurs manières, couvrant un grand nombre d'aspects : au niveau du code (programmation générique, méta-programmation, programmation orientée-aspect…) ou à un niveau d'abstraction plus élevé (*Domain Specific Languages*, voir 2.3.2.b)).

La méta-programmation peut être vue comme un cas particulier de programmation générique, dans laquelle les programmes sont écrits avec un grand nombre d'éléments variables, ces éléments pouvant être configurés à différents stade du cycle de vie du développement.

La programmation orientée-aspect permet, quant à elle, de s'abstraire de certains aspects d'un programme. Par exemple, pour un programme donné, les éléments relatifs à la

sécurité peuvent être isolés, de sorte qu'il sera alors facile de créer un système avec ou sans sécurité (encore une fois, selon le stade du développement auquel on se trouve).

b) ***Travaux associés***

Citons comme exemples de méthodologies dédiées à la programmation générative DRACO [Neighbors 1980] [Neighbors 1983], initié dans les années 80, et l'outil-méthode GenVoca [Batory et al. 2003]. Plus récemment, on peut assimiler à de la *Generative Programming* le projet Eclipse GMT (*Generative Modeling Tools Project*)[1]. La génération de code que nous proposons dans le dernier chapitre de ce manuscrit s'apparente également à de la programmation générative.

2.3.2. *Model Integrated Computing* et *Domain Specific Languages*

Le MIC est une approche du génie logiciel par méta-modélisation qui se concentre sur des modèles spécifique à un domaine. C'est pourquoi la notion de création de *Domain Specific Modeling Language* (DSML) est très souvent associée au MIC, car les artefacts de base de ce dernier sont des modèles spécifiques au domaine étudié. Nous faisons le choix de les présenter conjointement.

a) ***Principe du MIC***

À l'origine, le MIC [Sztipanovits et al. 1997] a surtout été employé pour la conception de systèmes embarqués. De nos jours, le MIC se retrouve dans plusieurs environnements dédiés. Le MIC préconise un cycle de développement type, reposant sur un processus en 3 étapes, ou niveaux :

1) **La méta-modélisation.** Les concepts du domaine sont identifiés, formalisés, liés entre eux par des attributs ou des références. En fait ce processus permet de créer des méta-modèles qui définissent pour les éléments du domaine : la syntaxe abstraite, la sémantique statique et également des règles de visualisation qui décrivent l'aspect graphique (comment les modèles sont représentés visuellement et comment ils interagissent avec l'environnement graphique). Une fois cette étape terminée, le méta-modèle est utilisé pour générer automatiquement un *Domain Specific Design Environment* ou DSDE. Cette génération automatique est qualifiée de *Meta-Level Translation* car elle consiste en un changement de niveau d'abstraction.
2) **Le DSDE.** Il correspond à un niveau d'abstraction plus bas, équivalent au niveau modèle. Le DSDE (environnement de modélisation spécifique à un domaine) généré est utilisé pour créer des modèles conformes aux règles du méta-modèle de la première étape. Ces deux premières étapes se situent à deux niveaux d'abstraction différents, la première étant en amont de la seconde.
3) **Application.** L'idée est de rendre productifs les modèles de l'étape précédente, par exemple en les convertissant en un autre format comme du code exécutable, ou un autre modèle. Le passage de ces modèles source en modèles cible plus « utiles » n'est rien d'autre qu'une transformation de modèles, effectuée par des transformateurs de modèles (*model transformers*) également appelés interpréteurs de modèles (*model*

[1] http://www.eclipse.org/gmt

interpreters). Ces transformations peuvent être effectuées dans le cadre du même domaine (ex : un modèle transformé en code) ou dans le cadre de deux domaines différents (ex : un modèle source transformé en un modèle cible d'un autre domaine). Dans le dernier cas, le modèle obtenu se conformera non plus au méta-modèle source mais au méta-modèle de destination.

Remarque : On peut considérer les approches de type MIC comme une incarnation particulière de MDA, tronquée, car restreinte à la spécification de systèmes au moyen de DSML.

b) ***Principe du DSL***

Un formalisme de modélisation spécifique à un domaine (on parle de DSL, *Domain Specific Language*, par opposition à *Domain Independent*, ou de DSML (*Domain Specific Modeling Language*) intègre les notions et concepts propres à un domaine [Kelly et al. 2007], que les utilisateurs ont l'habitude de manipuler : l'utilisation de ces concepts familiers permet un gain de temps précieux lors de la modélisation et donc un gain en puissance de modélisation.

En revanche, leur spécificité fait que certains DSL ne sont utilisés que par de petites communautés de personnes, ce qui entraîne un coût de production plus important, et surtout représente un frein au dynamisme de la communauté d'utilisateurs, qui ne bénéficient pas de la même qualité de code, de la même plateforme d'entraide et de support, des mêmes mises à jour que des utilisateurs d'un formalisme sous licence GPL par exemple.

Certes les DSL sont spécifiques, mais la tendance actuelle dans l'IDM veut qu'ils soient créés à partir d'outils et d'environnements plus génériques, capables de créer ces DSL. Il s'agit le plus souvent d'outils de méta-modélisation et de transformation de modèles de type MIC.

Pour [van Deursen et al. 1998] :

"*A Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*"

« Un DSL est un langage de programmation ou un langage de spécification exécutable qui offre, à travers des notations appropriées et des abstractions, une puissance d'expression concentrée (et habituellement restreinte) sur un domaine de problème particulier » – traduction personnelle

c) ***Approches basées sur les DSL***

Nous donnons ici une rapide vue d'ensemble des outils qui peuvent être considérés comme des « DSL purs ».

Ptolemy

Le projet Ptolemy, débuté en 1990 à l'Université de Californie de Berkeley, est un exemple de DSL permettant la spécification orientée-objet de systèmes hétérogènes à l'aide de plusieurs formalismes *low-level* différents [Lee 2001].

Le couple MATLAB/Simulink

MATLAB est un environnement (sous licence commerciale) de programmation pour le développement d'algorithmes, l'analyse des données, leur visualisation et le calcul numérique. Il est très souvent couplé à Simulink, qui est un environnement de simulation multi-domaine et de conception par modélisation (*Model-Based Design*) destiné aux systèmes dynamiques et embarqués [Simulink 2002]. MATLAB/Simulink est en partie graphique et connaît un succès grandissant dans le monde scientifique.

Idef3

La méthode *Integrated DEFinition for Process Description Capture Method* ou Idef3 [Mayer et al. 1995] fournit un mécanisme pour la description des processus. Elle est basée sur la causalité et la précédence entre les situations et les évènements, et on l'utilise pour exprimer des connaissances sur le fonctionnement d'un processus, d'un système, ou d'une organisation.

d) ***Evironnements de type MIC/DSML***

Le point commun à chacune des implémentations MIC suivantes est qu'elles reposent sur la même architecture : une couche « méta-modélisation » qui permet la spécification d'un DSL (dans un méta-formalisme donné, une couche « modélisation » qui permet de créer et de manipuler des modèles se conformant à ce DSL [Lédeczi et al. 2001], le tout dans un environnement dédié). La diversité des environnements créés d'une part, et celle des méta-formalismes, des langages de transformation qu'ils emploient d'autre part, témoignent de l'intérêt grandissant porté par la communauté scientifique et les acteurs économiques au MIC. Nous détaillons un environnement orienté MIC : AToM^3. Il existe de nombreux autres environnements MIC, plusieurs d'entre eux sont décrits dans l'annexe 3 de ce document. Certains utilisent des méta-formalismes vus dans la section 2.1.3.

AToM^3

AToM^3 est présenté comme suit sur la page du groupe de standardisation de DEVS:

"*AToM^3 is a tool for multi-paradigm modeling, by Juan De Lara (Autonomous University of Madrid, UAM, Spain) and Hans Vangheluwe (Mc. Gill University, Canada). AToM^3-DEVS is a tool for constructing DEVS models and generating Python code for the PyDEVS simulator by Jean-Sebastien Bolduc, developed in AToM^3*"

C'est un environnement multi-formalismes (les modèles peuvent être spécifiés en combinant plusieurs formalismes) de méta-modélisation, graphique, développé par le laboratoire MSDL (*Modelling, Simulation and Design Lab*) du département informatique de l'Université de McGill (Canada) [Lara et al. 2002a] [Lara et al. 2002c]. À l'instar des autres environnements MIC, il autorise la méta-modélisation et la modélisation ainsi que la transformation de modèles (endogène ou exogène). L'environnement est codé en Python et fonctionne sur plusieurs plateformes (Windows, Mac, Linux).

Le méta-formalisme de base utilisé par AToM^3 est basé sur des diagrammes entité-relation (*ER Diagrams*), enrichis par des contraintes exprimées en Python. À partir d'un méta-modèle défini visuellement à l'aide de l'éditeur graphique intégré, AToM^3 est capable de générer un éditeur graphique permettant de manipuler visuellement (création, édition) des modèles décrits selon ce méta-modèle. Formalismes et modèles sont décrits sous forme de

graphes. Les transformations de modèles sont également effectuées par réécriture de graphes (*graph rewriting*). Elles sont considérées comme des modèles à part entière.

La sémantique statique des méta-modèles est par défaut spécifiée au moyen de fragments de code Python. Toutefois, d'autres approches de définition des contraintes ont été proposées, notamment avec OCL[1] (contraintes associées soit aux entités soit aux relations), ou avec Prolog [Sen et al. 2008].

Parmi les méta-modèles disponibles nativement dans AToM^3 on trouve : les entités-relations, les diagrammes de flots de données, les automates finis (déterministes et non déterministes), les réseaux de Petri [Lara et al. 2002b]. Pour chacun de ces méta-modèles est fournie la génération de code vers des simulateurs qui respectent la sémantique opérationnelle des formalismes source. Cette dernière est décrite au moyen de transformations basées sur la grammaire des graphes. AToM^3 est considéré dans la littérature comme outil de transformation de modèles à part entière, par une approche basée sur la transformation de graphes.

Nous reviendrons par la suite sur AToM^3 car il rejoint directement le sujet de nos travaux (chapitre III) : il a en effet été utilisé dans un contexte DEVS, d'une part pour générer, à partir d'un méta-modèle de DEVS, un environnement permettant de spécifier des modèles atomiques et couplés, mais aussi pour transformer ces modèles en code exécutable vers le simulateur PythonDEVS.

2.3.3. Software Factories

Les *Software Factories*[2] ou usines logicielles, dans leur sens large, sont à relier aux approches vues précédemment : elles désignent des outils de type MIC. Mais le terme *Software Factory* désigne aussi une incarnation particulière de l'IDM vue par Microsoft [Greenfield et al. 2004a] [Greenfield et al. 2004b]. Nous présentons séparément cette approche car, à la différence des approches de types MIC que nous venons de voir, elle est exclusivement utilisée dans le monde industriel par des acteurs utilisant les produits Microsoft. C'est d'ailleurs sur la plateforme .NET qu'est implémentée cette suite d'outils d'aide à la création de DSL [Microsoft 2012].

Une définition de *software factory* selon Microsoft est donnée dans [Greenfield et al. 2004a] :

"*a software product line that configures extensive tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components*"

(« une ligne de production logicielle qui met en place des outils extensifs, des processus, et du contenu en utilisant un template d'usine basé sur un schema d'usine logicielle pour automatiser le développement et la maintenance des différentes variantes d'un modèle de produit type en adaptant, assemblant, et configurant des composants logiciels » - traduction personnelle)

[1] http://msdl.cs.mcgill.ca/people/wbliang/NSERC_2002/report.html
[2] http://www.softwarefactories.com

Les principales usines logicielles diffusées par Microsoft sont : Web Service Software Factory : Modeling Edition (ou Service Factory), SharePoint Software Factory, Prism Software Factory. Ces dernières favorisent clairement une interopérabilité « propriétaire » entre les standards et langages promus par la société (C#, Silverlight…) et intégrés à .NET.

Conclusion du chapitre

Ce chapitre nous a permis de comprendre la philosophie de l'IDM, à savoir la place qu'occupe le modèle dans cette discipline, les différentes approches de transformation de modèles existantes, les différents méta-formalismes et méta-modèles que l'on peut utiliser, et enfin les différents environnements orientés IDM existants.

Le standard MDA de l'OMG a occupé une place particulière dans ce chapitre, il est une incarnation particulière de l'IDM qui bénéficie des méta-modèles et des méta-formalismes déjà standardisés par l'OMG, et propose un langage de transformation de modèles hybride (QVT) : c'est ce standard qui fonde la base technique des implémentations proposées tout au long de la seconde partie de ce document.

Revenons à présent sur le formalisme DEVS afin de l'étudier sous un angle orienté-IDM : nous aurons l'occasion de discuter, entre autres, des problématiques d'interopérabilité liés à ce formalisme : vis-à-vis d'autres modèles DEVS écrits pour des simulateurs différents, vis-à-vis de ses extensions ou restrictions, vis-à-vis d'autres formalismes de bas niveau (de type MOC). Le chapitre que nous abordons maintenant constitue donc un état de l'art comparatif autour des solutions proposées pour favoriser la création, l'échange, la réutilisation et l'interopérabilité de modèles DEVS, ces solutions faisant entrer en jeu, à des niveaux plus ou moins importants, les techniques et concepts de l'IDM dont nous avons discuté tout au long de ce chapitre.

Chapitre III. TRAVAUX CONNEXES IDM & DEVS

DANS les chapitres précédents, nous nous sommes intéressés d'une part aux formalismes de modélisation et simulation, en particulier au formalisme DEVS, et d'autre part aux notions d'IDM et aux grands standards qui portent cette discipline. Dans ce chapitre qui vient conclure la première partie de ce document, nous analysons les travaux existants visant à faire bénéficier les modèles DEVS des apports de l'IDM en les situant dans la problématique plus générale de l'interopérabilité des composants[1] DEVS. L'objectif final de ce chapitre est de positionner nos travaux par rapport à cette analyse en introduisant nos principales contributions.

Comme nous l'avons vu précédemment, un modèle DEVS peut exister de manière purement textuelle : il suffit de partir de la définition mathématique de base et de décrire les tuples qui le composent, ou même de les spécifier explicitement à l'aide par exemple d'un pseudo-langage. Or, la finalité étant la phase de simulation, l'intérêt de tels modèles « statiques » reste très limité. Ces derniers doivent donc être implémentés. Différentes équipes de chercheurs ont, en fonction de leurs besoins, créé plusieurs environnements DEVS afin d'y coder leurs modèles en langage orienté objet (C++, Java, Python…) (voir 1.3.4).

L'implémentation de modèles DEVS en vue de leur simulation conduit automatiquement à une perte de généricité relativement à leur spécification formelle. En effet, qu'ils soient codés ou non au moyen du même langage, chacun des environnements orientés DEVS définit ses propres contraintes techniques, règles et conventions de programmation pour traduire et implémenter le formalisme DEVS et les algorithmes de simulations associés. Ceci caractérise le problème de l'interopérabilité « interne » de composants DEVS : la diffusion et la réutilisation de modèles DEVS sur différentes plateformes sont particulièrement difficiles alors que le partage, l'échange et la réutilisation des modèles par la communauté scientifique mondiale devraient se faire de la manière la plus large possible.

Comme le montrent les travaux de recherche que nous évoquons dans ce chapitre, l'IDM peut proposer des solutions plus ou moins complètes à ce problème : dans leur forme la plus aboutie, ces solutions impliquent à la fois un processus de méta-modélisation, des transformations de modèles et de la génération de code. Notre approche MetaDEVS se situe dans cette tendance.

Par ailleurs, il existe aussi un problème d'interopérabilité, que l'on peut qualifier d'« externe » dans la mesure où il fait référence aux correspondances possibles entre DEVS et d'autres formalismes : à évènements discrets (DESS, DTSS), ou bien basés sur les états et les transitions, ou bien de nature autre. Plusieurs travaux s'inscrivent dans cette optique et visent à transformer des modèles provenant d'autres formalismes en modèles DEVS, dans le but de faire bénéficier ces modèles de la puissance de simulation de DEVS. Nous montrons ici que l'IDM permet ici aussi d'apporter des réponses à cet autre besoin d'interopérabilité.

Ce chapitre s'organise de la manière suivante :

[1] Terme générique employé pour désigner les modèles et simulateurs DEVS.

- Dans une première section, nous introduisons la problématique générale de l'interopérabilité autour de DEVS. Elle peut être localisée tant au niveau des composants DEVS qu'à un niveau « externe » entre DEVS et d'autres formalismes (via les simulateurs, via les modèles, et hybride). Nous situons les travaux orientés IDM par rapport à l'ensemble de cette problématique et nous montrons en quoi ils proposent des solutions, tant au niveau de la conception des modèles DEVS que de leur implémentation, ce qui impacte la totalité du cycle de M&S ;
- La seconde section est consacrée à l'étude détaillée des approches qui utilisent les outils de l'IDM dans un contexte DEVS. Ces dernières sont centrées sur la définition d'un méta-modèle de DEVS et certaines proposent en outre des transformations de modèles M2M et/ou M2T ;
- Une dernière partie conclut ce chapitre en présentant deux tableaux comparatifs des principales approches abordées. Cette synthèse nous permet de situer notre approche parmi les approches existantes en mettant en évidence nos contributions.

3.1. De la problématique de l'interopérabilité des composants DEVS à l'IDM

Nous avons dit dans le chapitre précédent que l'interopérabilité désignait la capacité que possèdent des systèmes informatiques hétérogènes à fonctionner conjointement.

Affinons cette définition et appliquons-la à DEVS, en définissant simplement l'interopérabilité des composants DEVS comme la « capacité que possèdent des composants DEVS hétérogènes à être utilisés ensemble ». Nous pouvons aussi élargir cette définition à d'autres formalismes en considérant cette fois-ci une compatibilité que nous qualifions d' « externe » : elle concerne « la capacité qu'ont des modèles DEVS et non-DEVS à être utilisés ensemble ». Ceci peut se faire soit en transformant les modèles non-DEVS en modèles DEVS avant de les simuler, soit de simuler des modèles hétérogènes sur la même plateforme, ce qui implique le recours à l'encapsulation.

Dans cette section, nous caractérisons le problème de l'interopérabilité des composants DEVS d'une part et le problème de l'interopérabilité externe d'autre part. Nous mettons en évidence que le problème de l'interopérabilité des composants DEVS peut être abordé soit d'un point de vue centré sur les simulateurs, soit d'un point de vue centré sur les modèles. Dans les deux cas, l'approche IDM peut offrir des éléments de solution qui sont détaillés dans les sections suivantes. Toutefois, nous nous intéressons de manière plus approfondie aux approches d'interopérabilité basées sur les modèles, dans lesquelles s'inscrivent nos travaux : nous pensons en effet que c'est dans cette optique que les techniques de l'IDM (méta-modélisation, transformations de modèles), et les outils associés, sont en mesure de fournir les apports les plus significatifs, tant au niveau de l'interopérabilité des composants DEVS qu'au niveau de l'interopérabilité « externe », grâce aux transformations de modèles exogènes qu'elles permettent.

3.1.1. Le problème de l'interopérabilité DEVS

a) *Facteurs limitatifs*

Plusieurs facteurs limitent fortement l'interopérabilité des composants DEVS, nous en avons identifié trois principaux :

1) Bien que les modèles DEVS puissent être décrits en pseudo-langage algorithmique, ils ont avant tout vocation à être simulés, et donc à être implémentés (modèles productifs). Il existe différents langages informatiques orientés objet utilisés pour la description de modèles DEVS, et de ce fait pour un seul et même système étudié, il existera différents modèles implémentés (i.e. exécutables) équivalents ;
2) De la même manière, plusieurs plateformes de simulation DEVS sont disponibles, mais elles reposent sur des architectures et/ou des algorithmes, des méthodes, différents. En effet, même si les algorithmes abstraits de simulation DEVS sont (par définition) génériques, il n'empêche que les simulateurs les implémentent avec des langages de programmation divers. L'export d'un modèle implémenté dans un langage différent que celui de son simulateur est donc limité, ainsi que la simulation de modèles DEVS hétérogènes (évoquée ci-dessus). De plus, même si un même langage est utilisé pour implémenter deux simulateurs, ces implémentations seront différentes ;
3) De légères disparités au niveau de l'interprétation du formalisme DEVS lui-même, des pratiques de codage, font que, même dans le cadre du même simulateur, on peut se trouver en présence de modèles dont les fonctions, la structure, peuvent varier.

Le dernier facteur a des conséquences sur les modèles provenant d'une même plateforme, qui ne respecteront pas les mêmes contraintes et n'auront pas tout à fait la même forme s'ils ont été définis par des personnes différentes, même si initialement leur description textuelle (sous forme d'expressions mathématiques) peut être commune à tout le monde.

Les deux premiers facteurs, quant à eux, se combinent dans le cas où l'on considère que le processus de modélisation et simulation se fait dans un *framework* DEVS particulier et que l'on veut passer de celui-ci à un autre : cet inconvénient majeur résulte en grande partie de la vision monolithique que les équipes de scientifiques avaient lorsqu'ils concevaient des applications. Cette vision, qui a prévalu jusqu'à il y a quelques années, tend actuellement à laisser la place à des pratiques centrées sur les modèles (IDM) et/ou les services (de type SOA, *Service-Oriented Architecture*) facilitant la portabilité des composants DEVS. Ceci s'est fait en parallèle avec le monde du génie logiciel, qui de l'ère « tout est objet » est passé peu à peu, depuis la fin des années 90, dans la vision du « tout est modèle ».

b) *Vers une interopérabilité des composants DEVS*

Le terme interopérabilité suggère qu'à un moment donné il y ait une standardisation soit sur la manière de représenter les modèles, soit sur leur simulation.

Il existe à ce jour quelques projets de standardisation du formalisme DEVS, notamment l'initiative du SISO (*Simulation Interoperability Standards Organisation*) [SISO 2008] qui a conduit à la rédaction d'un court rapport préconisant certaines « pratiques » afin d'harmoniser le développement des environnements de modélisation et simulation DEVS. Les initiatives favorisant l'interopérabilité DEVS sont dans leur majorité des initiatives

provenant de chercheurs ou d'équipes de chercheurs du monde académique. Une page web[1] de l'université de Carleton, régulièrement mise à jour, recense une bonne partie de ces initiatives. Certaines de ces initiatives sont détaillées dans [Touraille et al. 2009a] qui aborde l'interopérabilité des modèles DEVS, et cite plusieurs travaux significatifs de ce domaine, ou également dans [Wainer et al. 2010]. Il existe également, sur le site dédié à l'environnement VLE (abordé en 3.1.2.c)), le compte rendu d'une réunion intitulée « Standardisation DEVS[2] » à laquelle ont participé des chercheurs français.

Enfin, un site commercial, vitrine d'une société fondée par des chercheurs du domaine de la M&S DEVS, est dédié aux productions scientifiques allant dans le sens de la recherche de représentations de DEVS indépendantes des plateformes : le site DUNIP Technologies[3].

Deux grandes familles de solutions ayant pour but de résoudre la problématique de l'interopérabilité DEVS ont vu le jour. Elles abordent le problème :

- D'un point de vue centré sur les simulateurs (aucune modification des modèles) ;
- D'un point de vue centré sur les modèles (aucune modification des simulateurs). C'est dans cette approche que nous nous inscrivons.

Notons qu'un certain nombre d'approches très intéressantes se situent à mi-chemin entre ces deux familles, nous en traitons quelques-unes dans cet état de l'art. Signalons par ailleurs que toutes ces démarches peuvent être orientées vers une utilisation de type offline, mais aussi online, soit sur un réseau local soit, et cela va de pair avec l'essor des architectures distribuées, particulièrement des SOA, via le Web.

Dans la section suivante, nous abordons brièvement les approches d'interopérabilité via les simulateurs. Ces dernières ne rentrent pas directement dans le cadre de nos travaux, mais nous jugeons intéressant de les évoquer ici afin de mieux situer notre approche par rapport à l'ensemble des travaux relatifs à l'interopérabilité des composants DEVS.

3.1.2. Interopérabilité via les simulateurs

Comme évoqué précédemment, les algorithmes de simulation DEVS, une fois implémentés dans un environnement de simulation DEVS selon le cycle de simulation basique défini avec le formalisme, deviennent des protocoles de simulation DEVS : ces protocoles décrivent comment DEVS utilise les simulateurs et de quelle manière ils interagissent avec les modèles. Appréhender l'interopérabilité des modèles DEVS d'un point de vue orienté simulation consiste à s'appuyer sur une propriété essentielle de DEVS : il y a une séparation explicite entre un modèle et son simulateur.

Cette philosophie part du principe qu'un modèle écrit dans un langage particulier doit pouvoir être simulé sur n'importe quelle plateforme, elle s'appuie donc clairement sur la finalité d'un modèle : être simulé afin d'obtenir des résultats. À noter que dans toutes les démarches que nous allons évoquer, il n'y a pas à proprement parler de modifications apportées aux simulateurs eux-mêmes, mais plutôt la mise en place d'une architecture leur permettant de communiquer de manière normalisée.

[1] http://cell-devs.sce.carleton.ca/devsgroup/
[2] http://www.vle-project.org/wiki/CR1_Standardisation_DEVS
[3] http://duniptechnologies.com/jm/

Nous présentons tout d'abord une vue d'ensemble des solutions existantes, puis nous choisissons d'examiner plus attentivement deux approches qui nous semblent illustrer les tendances dans la recherche d'interopérabilité des composants via les simulateurs

a) ***Vue d'ensemble des solutions existantes***

Une des solutions privilégiées est de créer une collection de services de simulation, existant soit de manière locale, sur la même machine, soit distribuée sur le réseau (Intranet, Internet…). Dans ce cas, l'environnement de simulation n'est donc plus situé physiquement sur la même machine, mais constitué par un ensemble de services de simulation (couplés à des simulateurs DEVS) répartis sur le réseau, la finalité étant de permettre l'envoi de messages homogènes entre logiciels DEVS hétérogènes. C'est cette dernière possibilité qui est choisie dans l'immense majorité des cas, utilisant des architectures de type CORBA, SOA ou HLA (*High Level Architecture*). Le principe de base est le suivant : un simulateur DEVS est vu comme un service et utilisé comme tel dans une architecture appropriée. Ce principe est ensuite adapté, décliné, suivant différents intergiciels (*middlewares*).

La communication entre simulateurs DEVS dans le cadre d'une architecture CORBA a fait l'objet de travaux, notamment [Zeigler et al. 1999a] et [Cho et al. 2001].

De même, l'architecture Peer-to-Peer (Pair à Pair) a été utilisée pour permettre une simulation partagée [Cheon et al. 2004], et [Zhang et al. 2005] a créé un environnement de simulation basé sur l'API Java RMI (Remote Method Invocation).

Une intégration de simulateurs dans une architecture « grid computing » (grille informatique) a été proposée par [Seo et al. 2004] (avant de s'orienter vers les SOA, voir 3.2.2). Ce type particulier d'architecture autorise la mise en commun des ressources inexploitées de machines distantes (notamment pour des calculs complexes).

Nous pouvons également citer l'existence de l'architecture HLA qui décrit un cadre et des règles pour normaliser les communications entre simulateurs hétérogènes distants [HLA 2000] : cette architecture est toutefois relativement compliquée à mettre en œuvre. Elle est utilisée et promue principalement par le *Department of Defense* (Ministère de la Défense) ou DoD, des U.S.A., plus précisément par son *Defense Modelling and Simulation Office* ou DMSO. HLA a donné lieu à des partenariats qui ont débouché sur plusieurs travaux, menés avec des scientifiques spécialistes de DEVS. Ces travaux ont intégré DEVS dans l'architecture HLA, dans [Zeigler et al. 1998] et [Zeigler et al. 1999b] qui décrivent l'environnement DEVS/HLA, ainsi que dans [Sarjoughian et al. 2000].

Cependant, depuis quelques années, la tendance s'oriente clairement vers des architectures SOA, leurs principaux avantages étant la souplesse de leur mise en œuvre, la simplicité du langage sous-jacent, SOAP[1], encapsulé dans le protocole HTTP. Par exemple, l'environnement CD++ a été modifié afin de pouvoir être utilisé de manière distribuée [Wainer et al. 2008].

Nous présentons de manière détaillée en b) une approche qui nous semble particulièrement représentative de cette tendance.

[1] http://www.w3.org/TR/soap12-part0

b) ***Une solution particulière : interopérabilité via SOA et espace de noms DEVS***

Les travaux de [Seo 2009] proposent une solution d'interopérabilité basée sur une architecture SOA et un espace de noms DEVS. Orientés « simulation pure », faisant abstraction de tout ce qui touche à la modélisation DEVS, et utilisant des solutions efficaces basées sur les Web Services, ces travaux sont relativement récents par rapports aux travaux cités en a) et figurent parmi les plus aboutis dans ce domaine.

L'architecture proposée repose sur différents simulateurs situés sur des serveurs Internet, sous forme de Web Services. Le point d'entrée est un serveur principal, sur lequel s'établit la connexion avec le client qui transmet son modèle. La simulation s'effectue ensuite de manière distribuée entre le serveur principal et les autres serveurs, via des messages transitant sur le réseau. Pour mener à bien ce processus, Seo a tout d'abord uniformisé les messages susceptibles d'être transmis en les répertoriant puis en les décrivant en XML sous forme d'espace de noms DEVS afin de lever toute ambiguïté sémantique. L'étape suivante consiste à répertorier les messages spécifiques à chaque simulateur (messages d'initialisation, récupération de la date de l'évènement suivant…). Ces simulateurs sont au nombre de deux dans cet exemple : ADEVS (écrit en C++) et DEVSJAVA (implémenté comme son nom l'indique en JAVA), chacun ayant par définition un protocole de simulation DEVS qui lui est propre. Ces messages sont ensuite traduits en XML en accord avec l'espace de noms créé (qui, lui, rappelons-le, est commun à tous les simulateurs) via des algorithmes : chaque fichier XML cible stockera en fait une instance d'un message DEVS. L'opération inverse, à savoir l'extraction de données d'un fichier XML récupéré via le réseau, afin de la traduire dans un protocole de simulation DEVS particulier, est également codée. Une fois en mesure d'encapsuler et de décapsuler des messages, les Web Services implémentant ces opérations sont mis en place sur les serveurs et couplés aux simulateurs, créant ainsi une architecture de type SOA. Les messages transitent ensuite sur le réseau entre les serveurs via le protocole SOAP, permettant de simuler un modèle sur des plateformes hétérogènes. Au moment de lancer la simulation, il suffit alors de déclarer les serveurs DEVS sur lesquels on veut la distribuer.

Il est à noter que, comme souvent lorsque des difficultés à simuler un modèle couplé se présentent, celui-ci sera considéré comme un modèle atomique, en vertu de la propriété de fermeture sous couplage du formalisme DEVS.

Pour conclure, on peut pour ces approches basées sur une architecture SOA dégager les avantages et inconvénients suivants :

- il est facile de mettre en œuvre des services ;
- les messages entre simulateurs sont standardisés ;
- la simulation est efficace, le fait que le code « existe » déjà limitant considérablement les risques d'erreurs, mais dépendante de l'architecture réseau utilisée : internet ou intranet, avec ce que cela comporte de risques d'erreurs lors de la transmission de messages séquentiels ;
- certaines difficultés d'ordre technique apparaissent pour ce qui touche à l'encapsulation et à la décapsulation de ces messages : pour écrire les algorithmes relatifs à ces dernières, il faut à la fois une excellente connaissance du langage

dans lequel est programmée la plateforme, ainsi que du protocole de simulation DEVS qui lui est spécifique ;

- il faut créer un service associé pour chaque simulateur.

Intéressons-nous à présent à un autre type d'interopérabilité via les simulateurs, représenté par le concept DEVS bus.

c) ***DESS, DTSS, DEVS bus et VLE***

Il existe une interopérabilité forte entre DEVS et d'autres formalismes employés en modélisation et simulation : DESS (*Differential Equation Specified System*) et DTSS (*Discrete Time Specified System)*. Celle-ci se manifeste au niveau des simulateurs tout d'abord, avec le concept de DEVS *bus*, qui permet d'implémenter un environnement de simulation hétérogène, mais aussi de manière beaucoup plus étroite puisque DEVS et DESS coexistent au sein d'un multi-formalisme appelé DEV&DESS : *Discrete Event and Differential Equation Specified System*.

DEVS bus [Kim et al. 1998] [Kim et al. 2003] est avant tout un concept : il se compose d'un bus de données virtuel sur lequel sont connectés divers simulateurs DEVS et non-DEVS (DESS, DTSS, voire d'autres formalismes). DEVS bus résulte en fait de l'application d'un *pattern* courant en informatique : le bus logiciel. La finalité est simple : plutôt que d'implémenter des simulateurs spécifiques de modèles à temps continu comme DESS, à temps discret comme DTSS, voire hybrides comme DEV&DESS, on a recours à l'encapsulation pour décrire ces formalismes comme si ceux-ci étaient des modèles DEVS. DEVS joue de ce fait le rôle de formalisme fédérateur : tous les messages sont encapsulés sous forme de messages DEVS, ce qui implique l'utilisation de convertisseurs dans certains cas. Une architecture de type DEVS bus permet donc l'interopérabilité des simulateurs grâce à l'encapsulation.

L'environnement VLE[1] (*Virtual Laboratory Environment*) [Quesnel et al. 2001] constitue une implémentation de DEVS bus. Il s'agit d'un environnement graphique, possédant une GUI, composé de différentes bibliothèques réutilisables sous licence GPL. Les algorithmes de simulation sont basés sur PDEVS [Chow et al. 1994]. Le but de cet environnement est analogue à la finalité de DEVS bus : permettre la simulation de multi-modèles.

VLE prend en charge les équations différentielles, ce qui permet la simulation de modèles de type DESS. Au niveau de DEVS, VLE permet, outre des modèles PDEVS, de spécifier et simuler des modèles DS-DEVS et FD-DEVS. Il est également possible, et cela est intéressant pour l'interopérabilité des formalismes situés au même niveau que DEVS, de modéliser et simuler des MOC dont : les FSM, les réseaux de Petri, les Statecharts.

La section que nous abordons à présent est destinée à étudier les différentes approches visant à améliorer l'interopérabilité DEVS via les modèles, et servira par la suite de référentiel permettant de situer précisément nos travaux.

[1] http://www.vle-project.org

3.1.3. Interopérabilité via les modèles : solutions proposées par l'IDM

À l'inverse de l'interopérabilité basée sur les simulateurs, l'interopérabilité via les modèles consiste à utiliser des modèles DEVS indépendants, dans la mesure du possible, de toute plateforme, donc indépendants de tout ce qui concerne les simulateurs. Cela implique de ne pas utiliser des modèles de simulation mais plutôt des modèles plus généraux capable de décrire les concepts de DEVS. Cette philosophie peut être appliquée de manière statique (offline) ou dynamique (online), mais repose dans les deux cas sur les modèles.

La recherche d'une représentation standardisée des modèles DEVS, évoquée en 3.1.1.b), est le point commun d'une grande partie des travaux se rapportant à l'interopérabilité DEVS via les modèles, y compris les nôtres.

Cette représentation standardisée peut prendre 2 formes :

- soit on utilise un formalisme connu dont le méta-modèle est disponible (exemple : UML) pour décrire des modèles DEVS (voir a)),
- soit on crée un méta-modèle pour DEVS (voir b)). C'est cette solution que nous avons choisie, et elle a débouché sur la création de MetaDEVS (chapitre IV). Nous étudierons particulièrement les travaux proposant eux-aussi un méta-modèle de DEVS.

Ceci vient illustrer notre discussion du chapitre précédent sur la nécessité, selon les cas, de créer un nouveau-méta-modèle plutôt que d'en utiliser un existant (voir 2.1.2.d)).

Concernant les transformations « externes » de DEVS, certaines ne s'inspirent absolument pas de l'IDM, et restent très formelles, comme par exemple [Giambiasi et al. 2003], qui décrivent formellement une transformation depuis les *Timed Automata* vers DEVS.

D'autres au contraire sont totalement IDM, dans le sens où elles se font via des définitions de transformations spécifiées au niveau des méta-modèles, et exécutées au sein d'outils IDM.

a) *Démarche de ré-utilisation des méta-modèles existants (UML)*

Cette démarche, dont tous les travaux s'y rappportant ont recours au formalisme UML, a tout d'abord été initiée par [Schulz et al. 2000] qui utilise le formalisme Statecharts pour créer des modèles équivalents aux modèles DEVS. Dans le même esprit et plus récemment, nous pouvons citer [Risco-Martín et al. 2007b], que nous retrouverons dans la section 3.2, qui utilise XML pour transformer le formalisme Statecharts en DEVS (plus précisément, DEVS State Machine). Ceci revient à spécifier des modèles DEVS au moyen de Statecharts.

Parfois, certains travaux ont recours à plusieurs des 13 diagrammes UML. Par exemple, dans [Zinoviev 2005], l'auteur part du constat que DEVS souffre d'un défaut de représentation graphique et propose de s'appuyer sur UML pour représenter tous les types de modèles DEVS. Les modèles atomiques sont spécifiés au moyen de diagrammes de machines d'états, les modèles couplés grâce à des diagrammes de composants. Cette approche ne possède toutefois pas d'implémentation concrète. En revanche, dans [Mooney et al. 2009], un environnement complet est proposé, il permet l'exécution de modèles DEVS spécifiés en

UML : l'essentiel de cette approche porte sur la gestion des aspects temporels, notion centrale à DEVS mais assez peu présente en UML.

Les mécanismes d'extensibilité d'UML peuvent aussi être sollicités : soit se servir d'un profil UML existant, c'est le cas par exemple de [Nikolaidou et al. 2008] qui propose de représenter graphiquement, au moyen de SysML, des modèles DEVS stockés en DEVSML (voir 3.2.1.a)), soit en créant un profil UML comme dans [Nikolaidou et al. 2007], qui prévoit en outre de générer partiellement du code (squelette de code).

Citons également, pour être complet, [Hong et al. 2004] et [Huang et al. 2005] qui emploient eux aussi UML pour spécifier des modèles DEVS.

Selon nous, cette démarche de ré-utilisation des méta-modèles existants a pour principal avantage de pouvoir ré-utiliser un méta-modèle (syntaxe abstraite) connu et populaire, UML, et, *de facto*, de bénéficier aussi de sa représentation graphique (syntaxe concrète). Les modèles DEVS écrits en UML sont donc globalement bien représentés et bien documentés, même si UML ne permet pas une gestion du temps très évoluée. L'inconvénient de cette démarche est la difficulté à produire du code à partir de ces modèles, excepté pour [Risco-Martín et al. 2007b] qui bénéficient comme nous le verrons plus loin des outils propres à l'espace technique XML.

b) ***Démarche de création d'un méta-modèle de DEVS***

En effet, la mise en place d'un méta-modèle pour DEVS permet de réduire le fossé existant entre la représentation formelle de DEVS et son implémentation dans un langage orienté-objet d'une part, et facilite la définition de transformations de modèles d'autre part.

En interopérabilité DEVS via les modèles et l'IDM, on peut mettre en évidence le processus suivant :

- **Définir un méta-modèle de DEVS**, exprimé dans un méta-formalisme, et ce afin de pouvoir définir un modèle DEVS, atomique ou couplé. Tous les modèles que l'on désire rendre interopérables devront se conformer à ce méta-modèle. Pour créer un tel méta-modèle, il est nécessaire de prendre comme point de départ les définitions de base des modèles atomiques et couplés. On répertorie ensuite les éléments propres à chacun d'eux et les éléments qu'ils possèdent en commun, en liant le tout de manière cohérente, et en choisissant un formalisme de méta-modélisation approprié. Nous détaillerons notre vision de cette méthode dans le chapitre suivant, qui présente notre démarche de méta-modélisation qui nous a conduits à définir le méta-modèle MetaDEVS.
- **Établir des passerelles entre ces modèles et différents simulateurs**, l'aboutissement idéal étant la génération automatique de code conforme à celui attendu par l'environnement de simulation (productivité) : les modèles sont ainsi prêts à être simulés. Ces passerelles sont implémentées sous la forme de transformations M2T entre des modèles se conformant au méta-modèle de DEVS et leur implémentation vers des plateformes de simulation spécifiques ;
- **D'autres passerelles peuvent être mises en place, à partir de formalismes différents vers DEVS, ou en provenance de DEVS vers d'autres formalismes**. Ces

passerelles permettent de résoudre le problème de l'interopérabilité « externe » à DEVS évoqué dans l'introduction de ce chapitre. En pratique, elles sont implémentées sous la forme de transformations M2M, de ou vers DEVS.

Ce processus caractérise parfaitement la philosophie centrée sur les modèles. La problématique principale est plutôt de savoir comment *créer* des modèles unifiés, y accéder, les modifier et les partager, que comment les *simuler* : l'aspect simulation n'est considéré qu'ensuite. Plus précisément, initialement, abstraction est faite de tout ce qui touche aux simulateurs, en partant du principe qu'il est facile d'installer et d'utiliser un simulateur DEVS quelconque sur une machine quelconque, vers lequel on exportera les modèles. Précisons que, dans tous les cas, il n'y a jamais aucune modification au niveau des simulateurs.

Il est important de garder à l'esprit que, quelle que soit la technique employée, l'élément qui influence directement la viabilité, la puissance d'expression, des modèles DEVS est la qualité du méta-modèle DEVS : plus ce dernier est précis, plus la capacité de modélisation sera étendue, et par conséquent plus on pourra simuler de systèmes complexes différents.

Par la suite, nous mettrons en évidence que le principal problème qui se pose lors de la création d'un modèle DEVS indépendant de toute plateforme n'est pas la définition de sa structure (dans le cas d'un modèle couplé), mais surtout la spécification de son comportement (modèle atomique), à travers les fonctions qui le composent. En effet, le modèle atomique DEVS reste ce qu'il y a de plus difficile à modéliser lorsqu'on veut rester en-dehors de toute plateforme et donc de tout langage. Nous donnons à présent un panorama des différentes approches de méta-modélisation DEVS qui sont apparues ces dernières années.

3.2. **Méta-modélisation DEVS et transformations associées**

L'IDM a apporté des améliorations notables dans la vision qu'avaient jusque lors les scientifiques des modèles DEVS. Dans un processus IDM, le code final d'une application n'est qu'un élément parmi d'autres, résultant d'une ou de plusieurs transformations de modèles, un modèle DEVS cesse d'être considéré sous sa forme finale de code simulable mais plutôt sous une forme plus abstraite : le modèle.

Ce modèle se conforme à un méta-modèle de DEVS et ne dépend plus d'une plateforme particulière, mais reste parfaitement à même de représenter tous les concepts du formalisme DEVS. Nous répertorions ici les principales approches visant à créer des méta-modèles de DEVS, au moyen de divers méta-formalismes, et nous étudions les transformations qui leurs sont éventuellement associées. Ces transformations interviennent à plusieurs niveaux : elles peuvent concerner exclusivement les modèles, et dans ce cas elles sont définies depuis le méta-modèle de DEVS vers d'autres méta-modèles (ou vice-versa), ou concerner les modèles DEVS et leurs équivalents simulables (code objet), elles sont alors définies entre le méta-modèle de DEVS et une plate-forme de simulation donnée.

Nous verrons dans cette partie que l'automatisation des transformations de modèles et de la génération de code est étroitement liée aux méta-modèles DEVS qui regroupent les concepts de ce formalisme.

Nous choisissons de présenter les principaux méta-modèles de DEVS existants en les classant selon le méta-formalisme qu'ils emploient et donc l'espace technique dans lequel ils se situent :

- XML Schema (espace XML) ;
- E/R Diagrams (espace BDD) ;
- EBNF (*grammarware*) ;
- MOF/Ecore (espace objet).

Pour chacun d'eux, nous examinerons particulièrement la spécification des modèles atomiques, plus précisément la manière de représenter les états et les fonctions permettant de décrire le comportement (fonctions de transition interne et externe, fonction d'avancement du temps et fonction de sortie).

3.2.1. **Méta-modèles DEVS basés sur XML**

Une première méthode consiste à créer des modèles DEVS sous formes de variantes de XML, langage balisé permettant le stockage de modèles. Tout modèle XML peut être :

- bien formé : il respecte la syntaxe préconisée par le W3C ;
- valide : il est bien formé, et de plus il est conforme à son méta-modèle, exprimé sous forme de DTD, ou bien de XML Schema. C'est ce méta-modèle qui est en fait celui du formalisme DEVS.

La caractéristique principale des méta-modèles de DEVS basés sur XML est qu'ils ne permettent pas d'offrir la possibilité de représenter des modèles DEVS autrement que textuellement.

Dans de nombreuses approches, telles que [Filippi, 2003], XML est utilisé pour représenter des modèles couplés DEVS (la nature de ce langage balisé fait qu'il est souvent utilisé pour spécifier des hiérarchies), mais les modèles atomiques restent spécifiés en langage objet. XML est donc utilisé dans ce cas-là pour une représentation partielle des modèles DEVS, et surtout pour permettre leur stockage.

D'autres approches, en revanche, prennent en charge les modèles atomiques et couplés, même si les fonctions propres aux modèles atomiques sont encapsulées dans un langage orienté-objet : ce sont ces dernières que nous détaillons ici.

a) ***DEVSML***

La méthode qui vise à représenter tous les modèles DEVS en XML a été initiée dans [Janousek et al. 2006] : elle introduit l'approche DEVS *Modeling Language* ou DEVSML. Elle permet de décrire des modèles atomiques et couplés DEVS, et est fortement influencée par JavaML (langage basé sur XML décrivant des modèles de code JAVA) [Badros, 2000]. La description sous forme textuelle d'un modèle étant efficace mais peu pratique, les auteurs proposent un petit outil graphique permettant de créer le squelette des modèles. Ces travaux posent la question de la modélisation des fonctions : langage simple, pseudo code…etc.

Les auteurs proposent provisoirement d'utiliser un petit langage fonctionnel proche de LISP. Les transformations vers le code de la plateforme de destination se font ensuite via des spécifications de type XSL (XSLT, XPath). Parallèlement à ces travaux, [Risco-Martín et al. 2007a] ont publié un schéma XML appelé DEVSXML exposant des concepts très proches de ceux de DEVSML, et ajoutant un petit simulateur à l'architecture de base : xDEVS.

Ces deux approches similaires ont par la suite fusionné et [Mittal et al. 2007a] constitue l'aboutissement de cette fusion. Le langage balisé DEVSML permet de définir le comportement de modèles atomiques en offrant la possibilité de définir des fonctions et aussi de décrire la structure du système via les modèles couplés. A également été prévue une fonctionnalité permettant à DEVSML de spécifier des modèles atomiques avec des fonctions proches de C++, mais à notre connaissance elle n'a pas été implémentée.

DEVSML est employé pour modéliser des modèles DEVS, au sein d'une architecture SOA. De ce fait, ces travaux peuvent être considérés comme des travaux hybrides : ils s'intéressent à l'interopérabilité via les simulateurs ainsi que via les modèles. Mais le plus intéressant ici est l'aspect modélisation/méta-modélisation, car l'export vers différentes plateformes est assez limité.

La DTD d'un modèle atomique a été définie comme le montre la Figure III-1). Les éléments simples sont indiqués sous formes d'attributs, ils suivent plus ou moins la définition de base d'un modèle DEVS atomique. On peut noter l'ajout des éléments suivants :

- un champ de métadonnées *simulator* : tout modèle en DEVSML porte donc en lui une indication concernant la plateforme sur laquelle il est destiné être simulé ;
- des éléments *service* : chaque service est en fait un container, il est implémenté dans le langage cible en tant que méthode. Cette fonctionnalité n'est pas encore implémentée ;
- un élément spécifique à Java concernant les imports, les constructeurs et les méthodes ;
- de même, comme DEVSML repose sur JavaML, les fonctions *deltint*, *deltext*…etc. sont proches de JavaML et seront ainsi transformées en code JAVA au moment de la simulation.

Les limitations de cette approche sont donc claires : malgré le fait que les modèles DEVSML puissent être multiplateformes, ils ne sont pour l'instant pas compatibles avec des simulateurs autres que ceux basés sur JAVA (ici, DEVSJAVA et xDEVS).

```
<!-- DEVS ATOMIC MODEL -->
<!ENTITY % variable-info
       "name CDATA #REQUIRED
        type CDATA #REQUIRED">
<!ELEMENT atomic
(inputs,outputs,states,ta,deltint,deltext,delt
con,lambda,services?,java-specific?)>
<!ATTLIST atomic
       name ID #REQUIRED
       simulator (devsjava|xdevs) #REQUIRED
       host CDATA #REQUIRED>
<!ELEMENT inputs (port*)>
<!ELEMENT port EMPTY>
<!ATTLIST port
       name CDATA #REQUIRED>
<!ELEMENT states (state*)>
<!ELEMENT state EMPTY>
<!ATTLIST state
       %variable-info;>
<!ELEMENT outputs (port*)>
<!ELEMENT ta (block?)>
<!ELEMENT deltint (block?)>
<!ELEMENT deltext (block?)>
<!ELEMENT deltcon (block?)>
<!ELEMENT lambda (block?)>
<!ELEMENT services (service*)>
<!ELEMENT service (method)>
<!ATTLIST service
       name ID #REQUIRED
       port CDATA #REQUIRED>
<!ELEMENT java-specific (package-
decl,import*,constructor*,method*)>
<!ELEMENT import EMPTY>
```

Figure III-1 : DTD d'un modèle atomique DEVSML [Mittal et al. 2007a]

Une nouvelle version du « framework DEVSML » a été définie dans [Mittal et al. 2012]. Elle propose une approche offrant la possibilité à des experts d'un domaine de définir des DSLs (*Domain Specific Language*) et de leur associer des transformations vers le langage DEVSML et vers du code exécutable DEVSJAVA. Cette dernière version s'inscrit par conséquent dans une optique plus clairement IDM et utilise les outils de l'environnement EMF. Le langage DEVSML est décrit sous la forme d'une grammaire EBNF en utilisant le framework XText[1]. Des DSLs tels que *NLDEVS* Natural Language DEVS sont également définis en suivant la même démarche (définition d'une grammaire EBNF). Des transformations de type M2M, vers DEVSM et DEVS sont ensuite spécifiées en utilisant le langage Xtend[2].

La Figure III-2 montre l'exemple d'un modèle atomique de générateur dans le langage DEVSML version 2 (vision arborescente du modèle générée par Eclipse-EMF).

La notion d'état est représentée sous la forme d'une énumération (passive, active, finishing) intégrée dans la description de la fonction timeAdvance elle-même ainsi définie également en extension (une valeur est associée à chaque valeur de l'état). Les fonctions de transitions sont représentées par une machine d'état *(DEVS state Machine)* définissant en énumération les différents changements d'état associés à chacune d'elle (internal_Transitions et external_Transitions). Il en est de même pour la fonction output qui est décrite en affectant un évènement de sortie à chaque valeur d'état concernée. On note également la possibilité de définir des variables globales typées au niveau d'un modèle atomique (ici arrTime et count).

[1] http://www.eclipse.org/Xtext
[2] http://www.eclipse.org/xtend

Le code DEVSJAVA est généré par l'exécution automatique d'une transformation spécifiée à l'aide du langage Xtend.

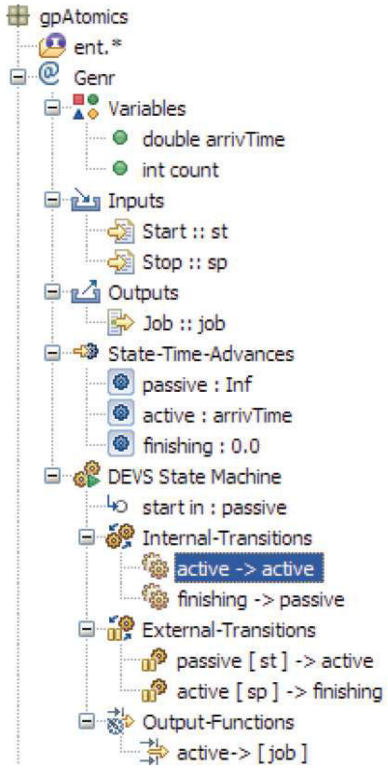


Figure III-2 : Exemple de modèle atomique DEVSML V2 [Mittal et al. 2012]

b) ***SimStudio et DML***

Une approche similaire à DEVSML a été menée par une équipe de l'ISIMA [Touraille et al. 2010] [Touraille et al. 2011]. Les auteurs proposent un méta-modèle unificateur pour DEVS baptisé DEVS *Markup Language* ou DML [Touraille et al. 2009b], exprimé en XML, et se conformant au XML *Schema*.

Ils décrivent également un environnement de modélisation et simulation DEVS : SimStudio. Cet environnement possède son propre noyau de simulation (programmé en C++) : DEVS Meta-Simulator (DEVS-MS). Sur celui-ci doivent venir se greffer différents composants logiciels, notamment pour la modélisation, la visualisation des modèles et leur stockage. A ce jour, seul un composant de modélisation basique semble avoir été implémenté, pourtant cette approche semble prometteuse de par sa philosophie originale. A terme, les auteurs annoncent l'intégration de plusieurs sortes de modèles dans l'environnement, notamment :

- Modèles DML modifiables, (ré)utilisables tels quels ou intégrables dans une hiérarchie de modèles couplés
- Modèles de simulateurs avec transformations (depuis DML) associées

L'intérêt de l'emploi d'un autre méta-formalisme est toutefois souligné et présenté comme une évolution possible de l'approche. Le méta-formalisme candidat est MOF, à cause notamment de son association avec le standard XMI.

DML propose une gestion originale des algorithmes, qui offrent un pouvoir d'expression relativement puissant. Pour décrire des fonctions DEVS il faut utiliser un

mélange de code générique, possédant une correspondance directe dans les langages impératifs existants, et de fragments de code abstraits, qui doivent être implémentés pour chaque langage cible. Nous donnons ci-dessous (Figure III-3) un exemple de code hybride ou semi-générique : dans ce fragment, deux implémentations sont possibles pour l'opération de mise d'un élément dans une file : pour C++ et pour Java.

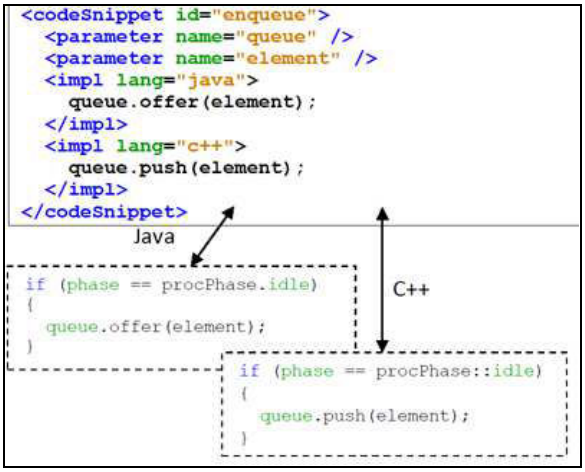


Figure III-3 : Fragment de code hybride C++/Java avec DML [Touraille et al. 2009a]

Seuls les types et les fragments de code abstraits nécessitent d'être concrétisés en une implémentation spécifique. Les auteurs, en vue d'une simplification de cette étape, proposent la construction de « catalogues » de types et de fragments accompagnés de leurs implémentations dans les langages les plus populaires. Ils offrent par ailleurs aux fournisseurs de *frameworks* DEVS de publier le modèle de leur plate-forme pour permettre l'intégration de celle-ci à SimStudio.

Trois niveaux d'indépendance sont identifiés : par rapport à la plateforme, par rapport au langage cible, par rapport aux bibliothèques.

Le problème délicat de la gestion des fonctions DEVS va de pair avec la spécification des algorithmes : code générique, code tiré d'un langage fonctionnel, ou fragments de code orientés-objet. Le fait d'utiliser un méta-modèle balisé complexifie notablement la définition d'une fonction, fut-elle très simple. La figure suivante tirée de [Touraille et al. 2010] est la description en DML d'une fonction testant la variable *sigma* : si cette dernière est nulle, un élément est ajouté dans un tampon. D'un fonctionnement pourtant très simple d'un point de vue algorithmique, cette fonction n'en est pas moins fastidieuse à implémenter manuellement (Figure III-4).

Des fragments de code objet spécifique, non pas au langage cible, mais à la plateforme de destination, sont là-aussi ajoutés.

L'intégration de DML dans le cadre d'une architecture orientée services a été évoquée dans [Touraille et al. 2009a]. Chaque modèle DML est interprété puis, grâce à WSDL, implémenté en Web Service. Cela permet de regrouper des modèles en ligne de manière à ce que les modèles locaux ou distants soient indifféremment utilisés dans la simulation, de manière totalement transparente. Les messages transitant sur le réseau ne sont plus cette fois-ci des messages entre simulateurs comme vu en 3.2) mais des invocations d'opérations propres au modèle (la fonction ta, δint, δext…). L'avantage d'une telle approche aurait été de

pousser au maximum l'interopérabilité des modèles en considérant indifféremment les modèles situés en local et les modèles distants. Toutefois, ce projet n'a visiblement pas été implémenté et n'est plus évoqué dans les articles présentant SimStudio [Touraille et al. 2010] [Touraille et al. 2011].

```
<if>
  <test>
    <binaryExpr op="equality">
      <lhs><var>sigma</var></lhs>
      <rhs><literal>0</literal></rhs>
    </binaryExpr>
  </test>
  <then>
    <codeSnippet kind="libDependent"
                 id=  "enqueue">
      <param id="queue">buffer</param>
      <param id="element">
        <codeSnippet kind="simDependent"
                     id=  "getValueOnPort">
          <param id="port">in</param>
        </codeSnippet>
      </param>
    </codeSnippet>
  </then>
</if>
```

Figure III-4 : Fonction DML implémentée en code semi-générique [Touraille et al. 2010]

Pour conclure cette sous-section traitant des méta-modèles DEVS basés sur XML, il faut souligner que malgré le fait que ce formalisme soit particulièrement bien adapté au stockage et donc au partage des données, il reste, lorsqu'il n'est pas associé à une représentation graphique, très lourd à modifier, dès que la hiérarchie des modèles ou la structure des fonctions se complexifient un tant soit peu.

c) ***Transformations associées***

[Risco-Martín et al. 2007b] proposent une transformation du formalisme Statecharts vers DEVS. Ce travail s'inscrit dans la continuité de

- [Mittal 2006] qui montre comment la plupart des diagrammes UML peuvent être transformés en composants DEVS orientés XML : DEVSML ;
- [Risco-Martín et al. 2007a] qui proposent une manière de créer des PIM DEVS (le terme PIM est employé ici en référence à MDA, même si les auteurs ne se placent pas dans le même espace technologique) avec DEVSXML (qui deviendra par la suite DEVSML).

Les auteurs se placent donc dans l'espace technologique XML et proposent une transformation en trois étapes, les deux premières étapes sont M2M et la dernière étape est M2T. La première étape consiste à définir des modèles exprimés en UML *State Machine Diagrams*. Ces modèles sont ensuite transformés en modèles SCXML[1]. Les fichiers SCXML représentant le modèle Statecharts sont ensuite transformés en FD-DEVS SM[2], un formalisme balisé représentant des machines DEVS finies et déterministes (restriction FD-DEVS vue dans le premier chapitre de ce travail) [Mittal 2007].

[1] http://www.w3.org/TR/scxml
[2] http://www.acims.arizona.edu/EDUCATION/XFDDEVS/UpdatedXFDDEVS.htm

Enfin, la dernière étape consiste à transformer les modèles FD-DEVS en des modèles de simulation interprétables par des simulateurs DEVS.

Ce travail se limite à la transformation de modèles comportementaux, l'auteur partant du principe que le comportement reste l'aspect d'un modèle DEVS le plus difficile à spécifier, contrairement à la structure (modèles couplés) dont les concepts peuvent plus facilement être compris par des chercheurs provenant d'horizons scientifiques différents.

En ce qui concerne SimStudio, les transformations associées se font via XSLT. Dans un premier temps, chaque modèle DML est transformé vers un modèle DML-Lang. La différence entre le méta-modèle de DML-Lang et celui de DML est minime : elle concerne le code générique. Les portions de code « génériques » en DML sont remplacées en DML-Lang par leurs équivalents dans le langage de destination, mais les fragments de code dépendants des bibliothèques et des simulateurs n'ont toujours pas été générés. Ensuite, chaque modèle DML-Lang est transformé en modèle DML-Sim, au moyen d'un méta-modèle appelé « fragments de code » établissant des correspondances entre la syntaxe abstraite exprimée avec DML et la syntaxe concrète propre aux plateformes et/ou au langage cible.

3.2.2. Méta-modèles DEVS basés sur les diagrammes entités-relations avec AToM³

L'environnement de méta-modélisation AToM³ [Lara et al. 2002a] (vu dans le chapitre II) a été utilisé dans [Posse et al. 2003] puis dans [Levytskyy et al. 2003a] pour créer un méta-modèle de DEVS, avec des diagrammes d'entités-relations comme méta-formalisme. Le diagramme ER représentant le méta-modèle de DEVS est montré sur la Figure III-5.

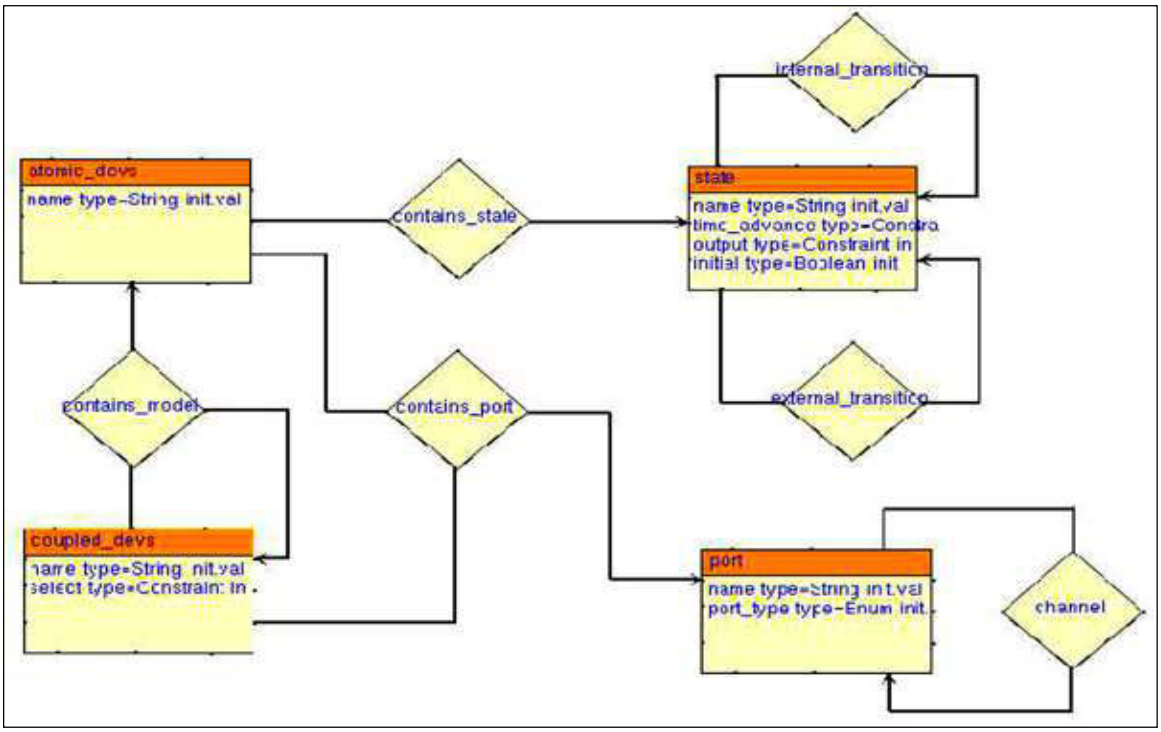


Figure III-5 : Diagramme ER décrivant DEVS dans AToM³ (version V1) [Posse et al. 2003]

Cette approche a pour caractéristiques de s'appuyer sur les possibilités d'AToM³ : outil MIC à part entière, il permet de générer un environnement de modélisation graphique

associé à un méta-modèle préalablement créé, ce qui a été fait pour DEVS dans les articles cités en exemple ci-dessus.

Dans cette première version, le méta-modèle DEVS permet la spécification d'états de type énuméré (états séquentiels) grâce à l'entité *State*. Les fonctions de transition sont représentées par des relations réflexives sur l'entité *State*. Cela signifie que dans un modèle, conforme à ce méta-modèle, l'ensemble des états et l'ensemble des transitions associées doivent être définis en extension.

Une seconde version de ce méta-modèle[1] a été définie par Hongyan Song de l'Université de McGill en 2005 (Figure III-6) [Song 2006].

Les états peuvent être définis non seulement de manière énumérée (entité *stateDevsV2*) comme dans la première version du méta-modèle, mais aussi en spécifiant des variables d'état sous la forme d'attributs de l'entité *atomicDevsV2*. L'entité *stateDevsV2* représentant les états séquentiels a été enrichie par l'ajout de deux attributs de type *Text* (*intAction* et *extAction*) spécifiant les actions devant être réalisées avant chaque changement d'état.

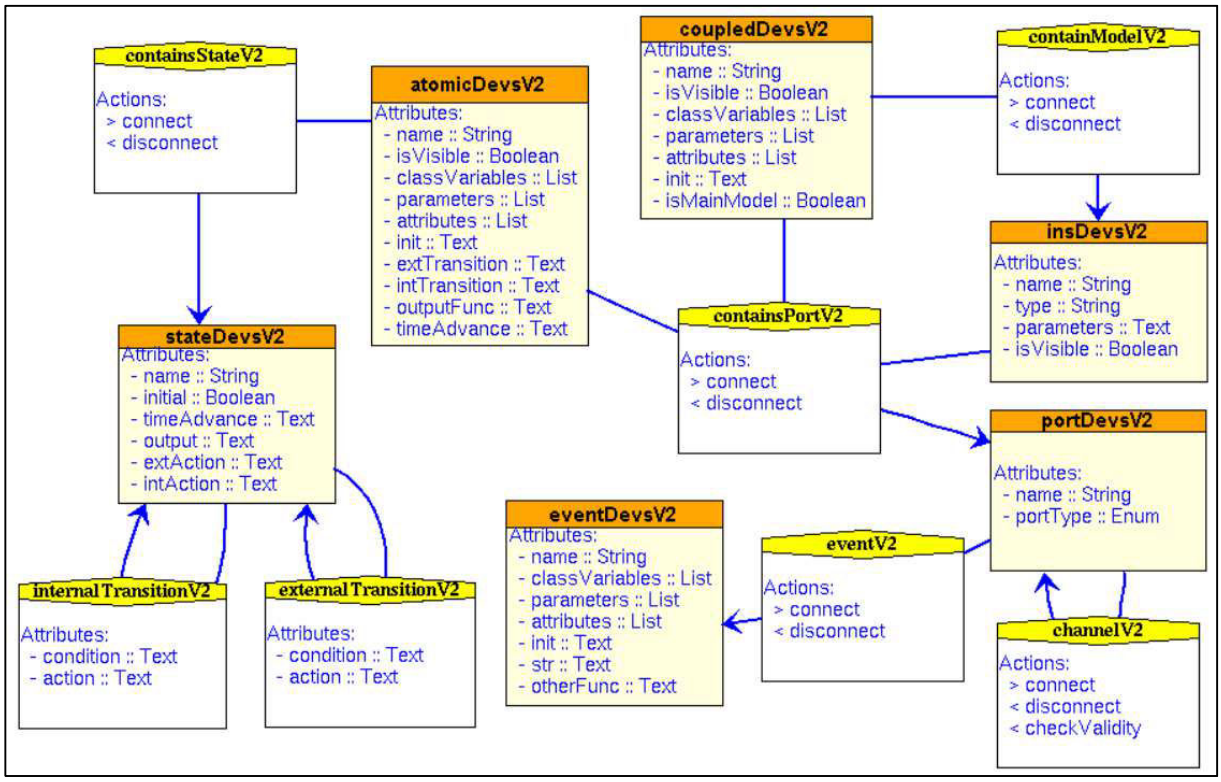


Figure III-6 : Diagramme ER décrivant DEVS dans AToM^3 (version V2) [Song 2006]

Les relations représentant les fonctions de transition interne et externe ont été enrichies par la définition de deux attributs de type *Text* (*condition* et *action*) dans les entités *internalTransitionV2* et *externalTransitionV2*. Des attributs de type *Text* associés à chaque fonction DEVS (*extTansition*, *intTransition*, *outputFunc*, *timeAdvance*) ont de surcroît été ajoutés dans l'entité *AtomicDevsV2*. Cependant, il n'est pas précisé comment les définir concrètement. Globalement, il semble que les valeurs des attributs de type Text associés à la représentation du comportement soient définies via l'inclusion en dur de code Python.

[1] http://msdl.cs.mcgill.ca/people/bill/devsenv/summerpresentation.pdf

Le langage Python sert aussi à exprimer des contraintes sur ce méta-modèle. Ces contraintes sont équivalentes à une sémantique statique pour le méta-modèle, elles viennent préciser, enrichir, les relations de base existantes entre les différentes entités. Elles jouent exactement le rôle que jouerait le langage OCL sur un méta-modèle exprimé en MOF.

À partir de ces spécifications, des environnements de modélisation DEVS ont été automatiquement générés. Les modèles DEVS créés dans le cadre de ces environnements peuvent être transformés en modèles de simulation programmés en Python et être simulés avec PyDEVS, via des processus utilisant les transformations de graphes. Ces processus suivent le principe du parcours de modèles que nous avons décrit dans la section du chapitre précédent dédiée aux approches pour la transformation de modèles. Pour pouvoir effectuer une génération de code, il faut d'abord parcourir le modèle passé en entrée. Il est nécessaire d'enrichir le méta-modèle avec ce que les auteurs appellent des « marqueurs », dont le but sera de désigner quels sous-modèles ont déjà été traités.

Le langage Python est utilisé à la fois pour les transformations M2M (vers DEVS ou depuis DEVS) et M2T (modèles DEVS vers code de simulation PyDEVS).

Plusieurs transformations ont été implémentées en suivant cette approche. On peut citer la transformation M2M à partir de Statecharts vers DEVS qui a été établie dans [Borland, 2003], la transformation M2T de DEVS vers PyDEVS, et la transformation M2T de SCD (*Simplified Class Diagrams*) vers la plateforme web Zope (code python) [Levytskyy et al. 2003b].

3.2.3. **Méta-modèles DEVS basés sur MOF**

Plus récemment, plusieurs approches de méta-modélisation DEVS au moyen de MOF ont vu jour. Le but est de décrire les concepts de DEVS au moyen de ce méta-formalisme, concepts implémentés généralement avec le langage Ecore, dans l'environnement de modélisation Eclipse EMF. Ces méta-modèles sont utilisés ensuite pour favoriser l'interopérabilité soit via les modèles, soit via les simulateurs.

Certaines approches que nous présentons ici ont pour objectif central la seule création d'un méta-modèle de DEVS, d'autres en revanche ont recours à un méta-modèle DEVS en vue de définir des transformations de modèles DEVS vers un langage donné. Il en résulte quelques différences de conception.

a) ***Approche « DEVS to SMP2 »***

Cette première approche présentée dans [Lei et al. 2009], est employée pour transformer DEVS en SMP2 (*Simulation Model Portability* 2) et, dans ce cadre, propose un méta-modèle MOF pour DEVS. Le but est de reconstruire un environnement de modélisation et simulation DEVS en termes de spécification SMP2, pour effectuer une simulation sur des environnements de type Sim2000. Cette approche s'inscrit dans une approche MDA. Le méta-modèle proposé a été conçu à la base dans une optique de transformation vers SDML : dans un souci de simplification de la transformation, les auteurs ont essayé autant que possible de le faire correspondre avec le méta-modèle de SDML. Ce dernier est composé de deux packages distincts, le premier (*Assembly*) contenant les composants utilisés pour la modélisation, inclus dans le second package (*Schedule*), qui permet de décrire comment les

instances du package *Assembly* sont utilisées au cours du temps (planification). Le méta-modèle de DEVS proposé suit cette philosophie.

Il est constitué par un package *ModelSystem* (Figure III-7) et un package *ModelComposite* (Figure III-8), qui décrit le déroulement des simulations DEVS, et utilise les instances des classes du package *ModelSystem*.

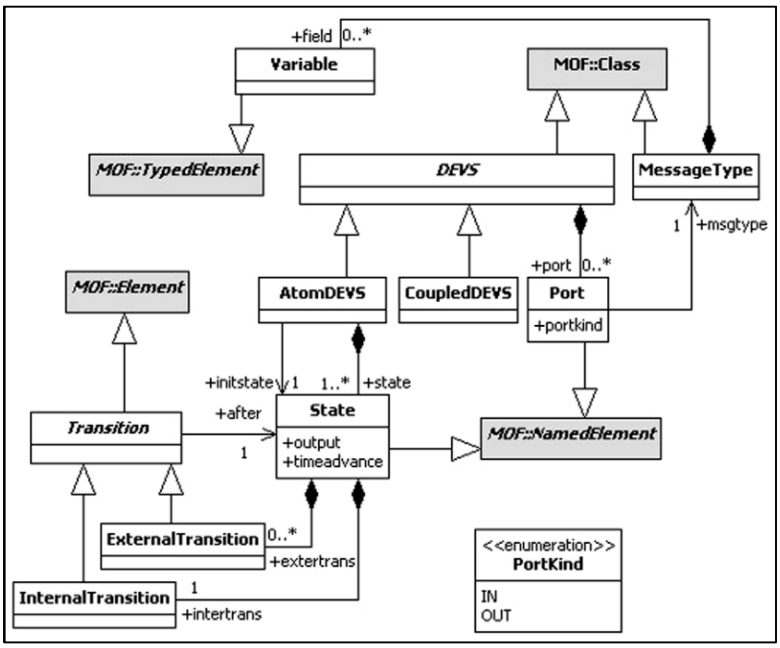


Figure III-7 : Le package *ModelSystem* [Lei et al. 2009]

L'implémentation se fait dans l'environnement EMF, plus précisément dans une architecture baptisée GMSE (*Generative Multiple formalisms modeling and Simulation Environment*).

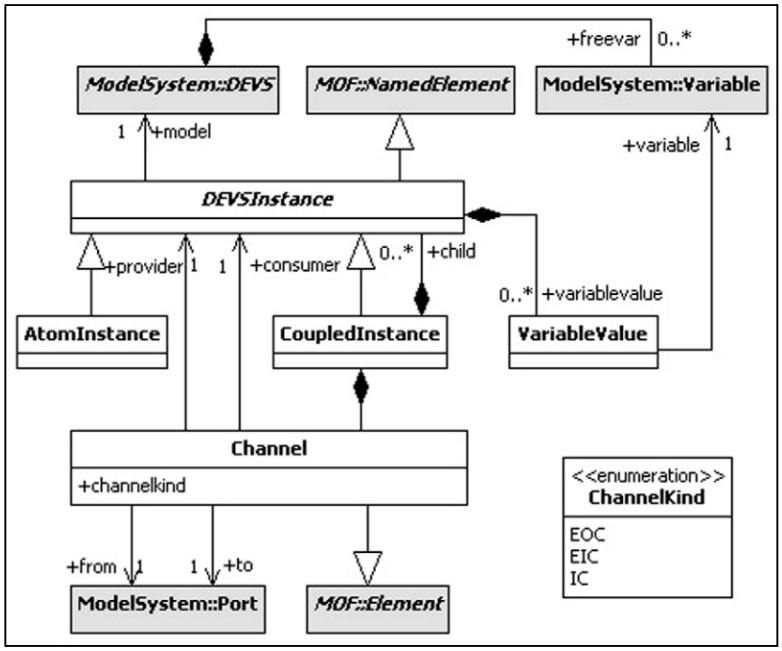


Figure III-8 : Le package *ModelComposite* [Lei et al. 2009]

Dans le package qui nous intéresse (*ModelSystem*), qui constitue la partie du méta-modèle chargée de décrire les modèles DEVS, les fonctions sont prises en compte (i.e. il est

possible de les déclarer) mais laissées vides. Il incombe au programmeur de les remplir, soit avant de transformer le modèle DEVS en un modèle SMP2, soit dans le modèle cible SMP2, après transformation. La spécification des états se fait par énumération explicite, à chaque modèle atomique sont attachés des états, l'un d'entre eux est l'état initial. Les états sont spécifiés sous forme textuelle, et énumérés.

b) ***EMF-DEVS [Sarjoughian et al. 2012]***

Contrairement à la précédente, cette approche, très récente, est avant tout centrée sur le seul formalisme DEVS, plus exactement *Parallel* DEVS, et l'accent est mis sur l'environnement EMF. Le méta-modèle de DEVS (Figure III-9) est directement pensé en termes Ecore : l'environnement décrit dans cette approche se nomme EMF-DEVS.

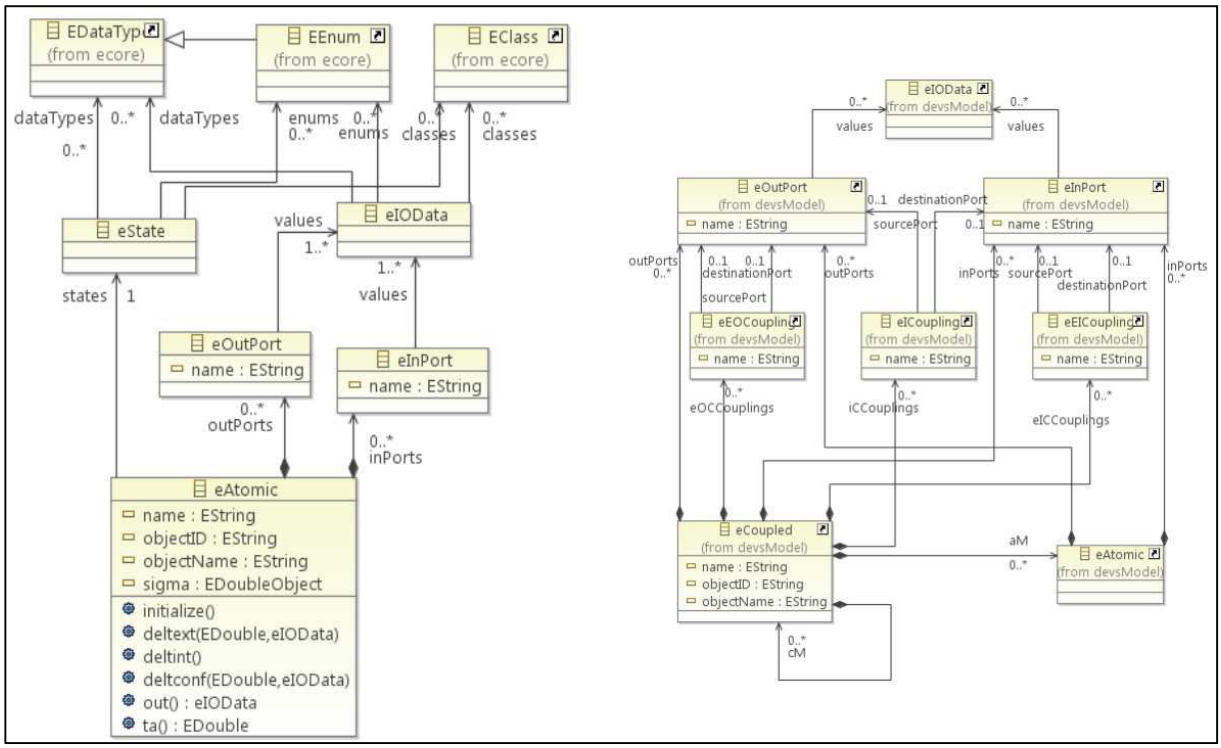


Figure III-9 : EMF-DEVS avec à gauche *eAtomic* et à droite *eCoupled* [Sarjoughian et al. 2012]

La génération de code se fait avec l'outil Eclipse Java EMF, le simulateur cible est DEVS-Suite [Zengin et al. 2010].

Le méta-modèle de DEVS proposé, baptisé EMF-DEVS, sépare explicitement les vues structurelle (modèle atomique) et comportementale (modèle couplé) en deux méta-modèles distincts (*eAtomic* et *eCoupled*). Dans [Sarjoughian et al. 2012] le méta-modèle *eAtomic* n'est cependant que partiellement représenté. Les états sont attachés au modèle atomique et peuvent être énumérés : une instance de modèle *eAtomic* possède un attribut *eState* contenant une collection de types de données de base (de type Ecore *EDataType*), une collection de types énumérés (*EEnum*) et une collection de classes (de type Ecore *EClass*).

Dans une analyse visant à caractériser les modèles DEVS, précédant la phase de méta-modélisation, l'auteur considère les fonctions atomiques comme abstraites, constituées d'une partie structurelle, et d'une partie comportementale, auxquelles sont associées une syntaxe et une sémantique. En pratique, les fonctions correspondant au comportement du modèle

doivent être implémentées manuellement. En revanche, les autres fonctions propres aux modèles couplés (fonctions de couplage) sont générées automatiquement.

c) ***MDD4MS***

Cette approche, très récente également, nous semble parmi les plus abouties. Son fondement théorique provient de [Cetinkaya et al. 2011b] dans lequel les aspects méta-modélisation et transformation de modèles en modélisation et simulation (dans une optique d'application à DEVS) sont explorés.

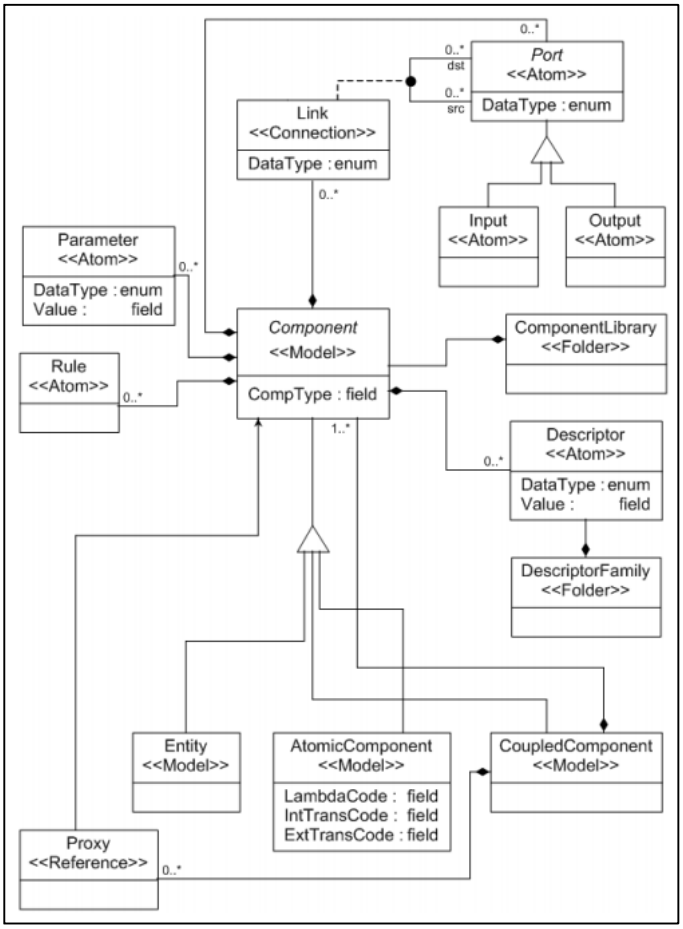


Figure III-10 : Méta-modèle GME de *Simulation Modeling* [Cetinkaya et al. 2010]

À l'origine de l'approche MDD4MS proprement dite, les auteurs ont proposé un méta-modèle d'environnement de simulation (Figure III-10) DEVS spécifié au moyen de l'environnement GME dans [Cetinkaya et al. 2010], selon une approche MIC.

On parle de *simulation modeling*, domaine qui s'intéresse à la création de modèles de simulation. Ces travaux proposent une technique de méta-modélisation et modélisation conceptuelle basée sur les composants DEVS, permettant de résoudre certains problèmes spécifiques à la simulation hiérarchique, et de générer automatiquement un environnement de modélisation graphique.

Le but de cet environnement graphique est de faciliter la création visuelle de modèles couplés à partir de modèles atomiques récupérés dans une bibliothèque. Enfin, par un mécanisme de génération de code, les modèles ainsi créés deviennent simulables. La simulation se fait dans le cadre de DSOL (*Distributed Simulation Object Library*), une

bibliothèque de simulation open source et multi-formalisme, reposant sur le langage Java, souvent utilisée comme SE [Jacobs et al. 2002]. Le méta-modèle proposé ne permet pas de spécifier le code des fonctions atomiques autrement que sous forme de texte « en dur », au moyen du langage Java. Les autres éléments des modèles sont obtenus par génération automatique de code. En 2011, l'environnement baptisé MMD4MS (*a Model Driven Development framework FOR Modeling and Simulation*) voit le jour : son prototype a été implémenté sous Eclipse EMF [Cetinkaya et al. 2011a].

Un méta-modèle de DEVS sommaire est présenté (voir Figure III-11), et la possibilité d'une génération de code partielle est évoquée (le corps des fonctions reste à écrire par le programmeur).

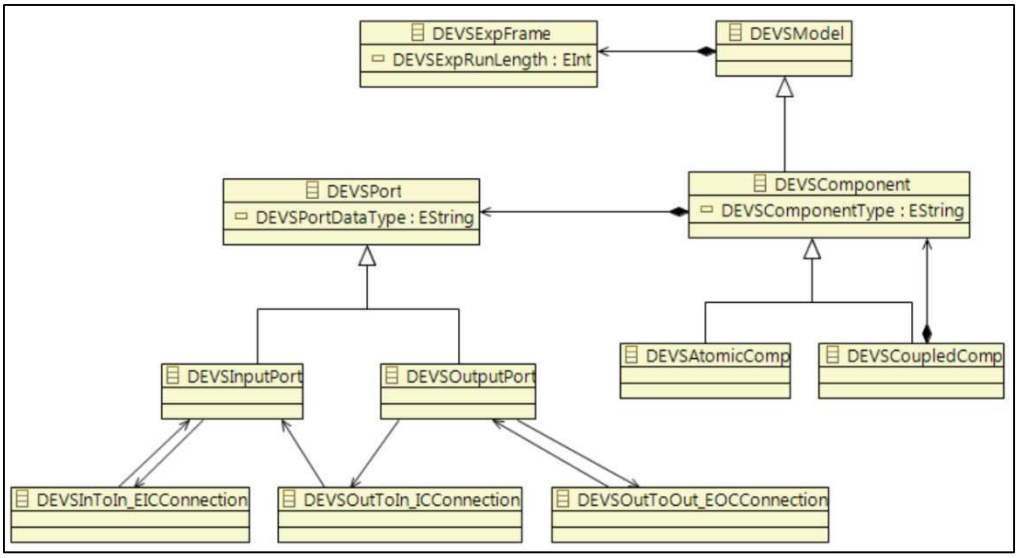


Figure III-11 : Méta-modèle de DEVS « simple » dans MDD4MS [Cetinkaya et al. 2011a]

Puis, très récemment, dans [Cetinkaya et al. 2012], ce méta-modèle est affiné et décrit dans son intégralité (voir Figure III-12) dans l'optique d'une transformation, au sein de l'environnement MDD4MS, de BPMN (*Business Process Model and Notation*) vers DEVS.

Le méta-modèle suit une approche MDA, avec les concepts de SCM (*Simulation Conceptual Model*), équivalent du CIM, de PISM (*Platform Independent Simulation Model*) et PSSM (*Platform Specific Simulation Model*).

Les cardinalités sont en revanche totalement absentes du diagramme. L'originalité de ce méta-modèle est que, contrairement au méta-modèle de la Figure III-11, les fonctions atomiques sont cette fois-ci décrites, via la variable *Expression*, en pseudo-code, indépendant de toute plateforme. Une expression peut être un appel de fonction, un bloc conditionnel, ou une affectation. Les auteurs ne donnent toutefois pas plus d'indications sur le pseudo-langage utilisé : ils signalent simplement qu'il existe un méta-modèle de ce langage, lié au méta-modèle de DEVS via la variable *Expression*.

La gestion des états se fait via une méta-classe *State* à laquelle sont attachées des variables d'état, pouvant être de différents types de base, et possédant une valeur initiale. La génération de code M2T n'est que très peu abordée dans cet article, les auteurs annoncent

cependant qu'elle est totale (à la fois pour les modèles BPMN et les modèles DEVS) et qu'elle est effectuée en direction de Java.

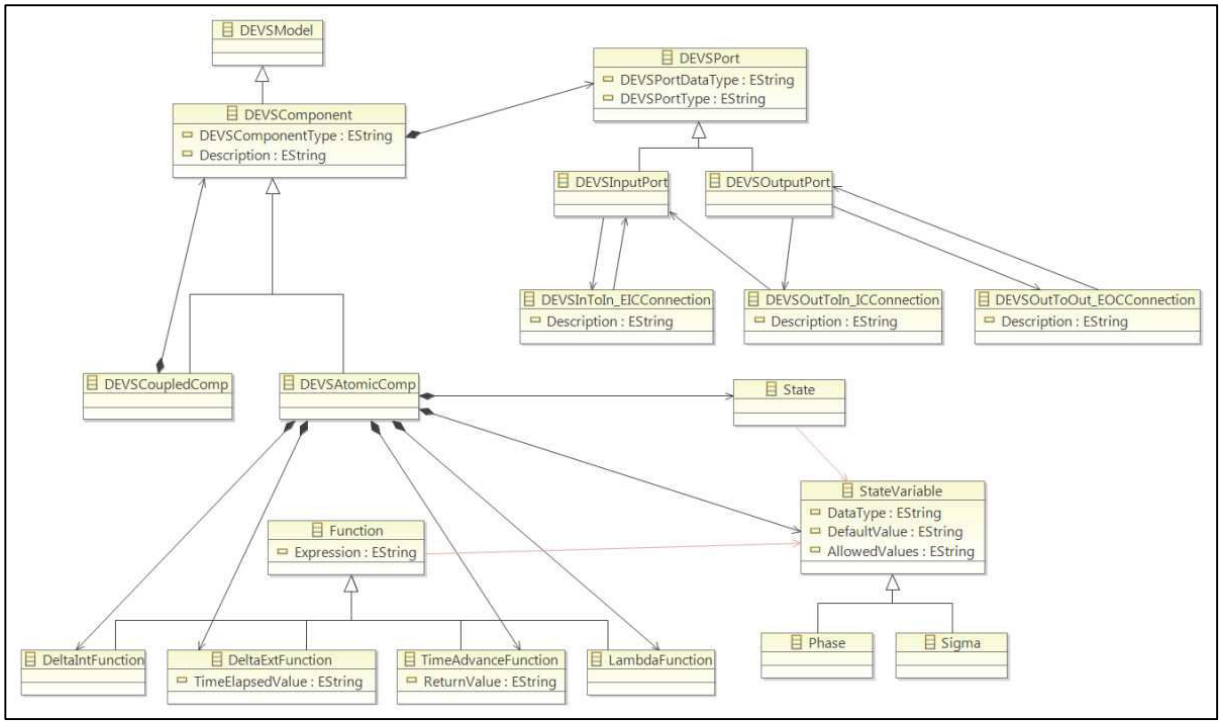


Figure III-12 : Méta-modèle de DEVS « étendu » dans MDD4MS [Cetinkaya et al. 2012]

3.3. **Synthèse**

Ce chapitre nous a tout d'abord permis de mettre en évidence l'existence d'un double problème d'interopérabilité liée à DEVS :

- entre d'autres formalismes et DEVS (interopérabilité externe) ;
- au niveau des composants DEVS eux-mêmes (via les simulateurs et via les modèles).

Nous avons vu à travers les exemples précédents que l'interopérabilité « externe » des modèles DEVS « classiques » pouvait concerner comme autres formalismes cible des formalismes :

- à évènements discrets ;
- de type MOC (basés sur les états-transitions) ;
- de type « extension de DEVS » ou « restriction de DEVS » ;
- autres (à déterminer au cas par cas).

Suivre une approche IDM permet de résoudre une partie de ce problème d'interopérabilité car si l'on dispose du méta-modèle de DEVS vers lequel ou à partir duquel on veut transformer des modèles conformes au méta-modèle d'un autre formalisme (dont on dispose également), il est possible d'établir des règles de transformation pour convertir un modèle (source) en un autre (cible). Ces règles de transformation sont de type M2M.

On peut aussi, à partir d'un modèle de simulation DEVS (i.e. un modèle implémenté sur une plateforme de simulation spécifique), se poser la question de sa portabilité vers une autre plateforme, et exprimer ainsi le besoin d'interopérabilité des composants DEVS. Nous avons vu qu'une première solution pouvait consister à employer une architecture de type DEVS-Bus, en d'autres termes privilégier l'interopérabilité via les simulateurs DEVS.

Nous avons ensuite mis en évidence que suivre une démarche IDM permettait d'offrir des solutions au problème de l'interopérabilité des composants DEVS via les modèles.

Disposer d'une représentation standardisée des modèles DEVS facilite non seulement le stockage, mais aussi l'échange, la modification et l'implémentation de ces modèles sous forme de code exécutable sur une plateforme de simulation donnée. Le modèle DEVS, indépendant de toute plateforme, placé au centre de cette approche, se trouve affranchi des contraintes spécifiques à telle ou telle architecture, tel ou tel langage et telle ou telle plateforme. Sa représentation ne varie plus. Grâce à la définition de règles de transformation, ce modèle peut au besoin être traduit en plusieurs modèles de simulation propres à des plateformes données, grâce à des transformations de type M2T. Pour un même modèle de base, il peut exister plusieurs implémentations. Ceci illustre les apports de la méta-modélisation, dans un cadre IDM, qui permet de combler le fossé qui sépare habituellement la représentation formelle des modèles DEVS et leur implémentation.

Le Tableau III-1 résume nos propos, en montrant qu'une interopérabilité entre un modèle DEVS indépendant de toute plateforme et une extension/restriction de DEVS, ou un autre formalisme (sous réserve de la possibilité d'établir des correspondances avec les concepts du formalisme DEVS) est possible en faisant appel à des transformations de type M2M, de manière bidirectionnelle (transformation depuis DEVS ou vers DEVS). Le tableau montre aussi qu'il est possible d'obtenir du code de simulation à partir de ce modèle DEVS : toutefois, cette transformation M2T se fait pour l'instant principalement à sens unique (sauf pour [Mittal et al. 2007a] qui paraît proposer des transformations réversibles de Java vers DEVSML). La réversibilité d'une transformation M2T demeure un thème de recherche lié au *reverse engineering*.

<table>
<tr><th colspan="3">Eléments candidats à l'interopérabilité</th><th>Solution IDM</th></tr>
<tr><td rowspan="3">Modèle DEVS indépendant de toute plateforme (assimilable à un PIM)</td><td>→</td><td>Code(s) de simulation DEVS</td><td>M2T</td></tr>
<tr><td>↔</td><td>Modèle d'un autre formalisme</td><td rowspan="2">M2M (interopérabilité externe)</td></tr>
<tr><td>↔</td><td>Modèle d'une extension ou une restriction de DEVS</td></tr>
</table>

Tableau III-1 : L'IDM, une solution pour l'interopérabilité DEVS

Avant de conclure ce chapitre et cette partie, résumons les principales approches de méta-modélisation DEVS que nous venons de voir, au moyen de deux tableaux comparatifs. Ces tableaux comparatifs ont pour but, pour les principales approches vues précédemment, d'illustrer :

- les caractéristiques des méta-modèles de DEVS proposés : nous nous intéressons plus particulièrement à la représentation des états et des fonctions comportementales DEVS, qui constituent les éléments les plus difficiles à décrire indépendamment des plateformes ;

- la nature des transformations associées : ces dernières ont permis d'une part d'effectuer des transformations M2M en provenance ou à destination de DEVS et d'autre part de générer automatiquement des modèles de simulation DEVS avec des transformations M2T. Nous évoquons ici ces différents types de transformations à travers les approches et langages de description utilisés, le sens des transformations M2M, et la plateforme de destination éventuelle (M2T).

La dernière ligne de chaque tableau, introduit notre approche MetaDEVS en la positionnant par rapport aux autres approches étudiées. Cette approche a été introduite dans [Garredu et al. 2012a] et est présentée en détail dans le chapitre IV de ce travail.

MetaDEVS est issu d'une approche IDM, orientée MDA, que nous proposons pour le formalisme DEVS. MetaDEVS est plus concrètement un méta-modèle permettant de spécifier des modèles DEVS indépendamment de toute plateforme, tout en prenant en charge la gestion des fonctions, sans utiliser de code. Plus largement, lorsque nous parlerons par la suite d'approche MetaDEVS, nous ferons référence non seulement au méta-modèle situé au centre de cette approche mais aussi aux techniques de transformation M2M et M2T que nous employons autour de ce méta-modèle.

Nous considérons en effet le méta-modèle MetaDEVS comme un pivot, vers lequel il est possible de transformer d'autres formalismes (interopérabilité « externe) via des transformations M2M. Ces modèles spécifiés en MetaDEVS sont stockables, échangeables, modifiables, et représentent fidèlement les concepts de DEVS. Leur intérêt ne serait pourtant que limité s'ils ne pouvaient pas être traduits en modèles de simulation (i.e. code objet). Nous montrerons comment un modèle typé par MetaDEVS peut être ensuite transformé en code interprétable par un simulateur DEVS (transformations M2T).

a) *Comparaison des méta-modèles DEVS*

Si l'on considère le méta-formalisme utilisé, les méta-modèles DEVS peuvent se classer en quatre catégories, correspondant à quatre espaces techniques distincts :

- Méta-formalisme XML : DEVSML [Risco-Martín et al. 2007b] et SimStudio [Touraille et al. 2010]
- Méta-formalisme Entité-Relation : AToM^3 DEVS metamodel V1 [Posse et al. 2003], AToM^3 DEVS metamodel V2 [Song 2006]
- Méta-formalisme textuel EBNF : [Mittal et al. 2012]
- Méta-formalisme MOF-Ecore : DEVS to SMP2 [Lei et al. 2009], MDD4MS [Cetinkaya et al. 2012] et EMF-DEVS [Sarjoughian et al. 2012]

Notre méta-modèle MetaDEVS se classe dans cette dernière catégorie. Le Tableau III-2 nous permet d'identifier globalement deux solutions au niveau de la représentation des états : une représentation orientée « états finis » et une représentation orientée «variables d'états ». Ces deux solutions apparaissent comme complémentaires et plusieurs méta-modèles DEVS proposent de les intégrer.

Approche	Méta-formalisme	Représentation des Etats enumérés	Variables d'état
DEVS to SMP2 [Lei et al. 2009]	MOF - Ecore	méta-classe *State*	méta-attribut free variable dans la métaclasse DEVS
MDD4MS [Cetinkaya et al. 2012]	MOF-Ecore	méta-classe *State*	méta-classe *StateVariable* associée à la métaclasse *State* et à la métaclasse *AtomDEVS*
AToM3 DEVS metamodel V1 [Posse et al. 2003]	ER Diagrams étendus	méta-entité *State*	-
AToM3 DEVS metamodel V2 [Song 2006]	ER Diagrams étendus	méta-Entité *State*	méta-attribut ClassVariables (List) dans la métaEntité AtomicDevsV2
DEVSML V1 [Risco-Martín et al. 2007b]	XML Schema	balise *State*	-
DEVSML V2[Mittal et al. 2012]	Grammaire EBNF	énumérés dans le méta-attribut *state-Time-Advances*	métaattribut *Variables* du méta-élément *Atomic*
SimStudio - DML [Touraille et al. 2010]	XML Schema	balise *State*	balises *Variable (name, type)* intégrées dans la balise *State*
EMF-DEVS [Sarjoughian et al. 2012]	MOF-Ecore	meta-classe *eState*	-
MetaDEVS	**MOF-Ecore**	**méta-classe *StateVar (name, type)***	**méta-classe *StateVar (name, type)***

Tableau III-2 : Tableau comparatif des principaux méta-modèles DEVS

- **Etats finis énumérés : Méta-Elément *STATE***

Le concept d'état est représenté par un méta-élément généralement nommé « State » (méta-classe, méta-entité, ou balise selon le métaformalisme utilisé) et permettant de représenter uniquement des états unidimensionnels. Cette solution est proposée dans la plupart des méta-modèles DEVS : on peut citer en particulier les méta-modèles DEVS to SMP2 [Lei et al. 2009], [Posse et al. 2003] (DEVS avec AToM3 version V1) et [Song 2006] (DEVS avec AToM3 version V2). Les états d'un modèle basé sur un tel méta-modèle seront ainsi définis en extension (et donc finis), sous la forme d'une énumération de valeurs littérales.

- **Etats caractérisés par un ensemble de variables typées : Méta-Eléments de type variables d'état**

Cette solution consiste à caractériser le concept d'état par des méta-éléments de type variables d'état autorisant la spécification d'états multidimensionnels. Les approches utilisant le méta formalisme XML proposent une solution de ce type. Elle consiste à définir des balises <variable>, intégrées dans une balise plus générale <state>. Dans les méta-modèles MOF-Ecore, la solution consiste à définir des variables intégrées comme des attributs situés dans le méta-élément correspondant au modèle atomique ([Lei et al. 2009] et [Song 2006]).

[Cetinkaya et al. 2012] propose l'introduction d'une méta-classe StateVariable venant compléter la méta-classe State.

Par ailleurs, le formalisme que nous avons décrit dans [Garredu et al. 2009] et que nous introduisons dans le chapitre V de ce travail, sous le nom de BasicDEVS, permet lui aussi de représenter les états en extension : ils sont énumérés explicitement lors de la définition de modèles.

Approche	Fonctions de transition	Avancement du temps	Fonction de sortie
DEVS to SMP2 [Lei et al. 2009]	métaclasses *ExternalTrans* et *InternalTrans*	métaattribut *timeadvance* de la métaclasse *State*	métaattribut *output* de la métaclasse *State*
MDD4MS [Cetinkaya et al. 2012]	métaclasses Spécifiques avec attribut *expression* de type String		
AToM3 DEVS metamodel V1 [Posse et al. 2003]	métaAssociations *External_transition* et *Internal_transition)*	attribut *time_advance* de la méta Entité *State*	métaattribut *output* de la méta Entité *State*
AToM3 DEVS metamodel V2 [Song 2006]	métaAssociations + méta-attributs de type text de la méta Entité *atomicDevsV2*	méta-attribut de type text de la méta Entité *atomicDevsV2*	méta-attribut de type text de la méta Entité *atomicDevsV2*
DEVSML V1[Risco-Martín et al. 2007b]	*Element* spécifiques ("blocks" à compléter)		
DEVSML V2 [Mittal et al. 2012]	méta attribut *DEVS state machines*	méta attribut *state-Time-Advances* du méta-élément *Atomic*	méta attribut *outputFunctions* du méta-élément *Atomic*
SimStudio - DML [Touraille et al. 2010]	Code hybride semi-générique (à compléter selon la plateforme)		
EMF-DEVS [Sarjoughian et al. 2012]	méthodes abstraites *deltaext* et *deltaint* de la métaclasse *eAtomic*	méthode abstraite *ta* de la classe *eAtomic*	méthode abstraite *out* de la classe *eAtomic*
MetaDEVS	**hiérarchie de métaclasses Spécifiques (notion de *DevsRule*)**		

Tableau III-3 : Spécification des fonctions comportementales dans méta-modèles DEVS

En revanche, la solution que nous proposons dans le méta-modèle MetaDEVS, qui a été notamment présentée en [Garredu et al. 2012a], consiste à utiliser exclusivement le concept de *StateVariable* pour encapsuler les états possibles d'un modèle atomique, que ces derniers soient énumérés ou pas. Nous sommes en ce sens proches de la solution proposée par [Cetinkaya et al. 2012] à laquelle manquent toutefois les cardinalités dans la spécification du méta-modèle, pourtant nécessaires à une meilleure compréhension des liens entre les méta-classes *State* et *StateVariable*. Nous rejoignons également les solutions proposées dans les méta-modèles XML mais l'utilisation du méta-formalisme MOF nous permet de bénéficier d'un environnement IDM plus riche notamment au niveau de la spécification des transformations (existence de nombreuses implémentations de QVT…).

Le Tableau III-3 présente les différentes approches de spécifications des fonctions comportementales dans les méta-modèles DEVS étudiés.

La méta-modélisation des fonctions comportementales (DeltaExt, DeltaInt, ta et Lambda) consiste à définir des méta-éléments permettant de décrire la logique associée : en

d'autres termes, il s'agit de décrire le corps des fonctions. Selon l'approche choisie à ce niveau, la génération du code associé aux fonctions lors de la mise en œuvre de transformations M2T pourra être limitée aux entêtes des fonctions, partielle ou totale.

On distingue globalement trois types d'approches en fonction de la solution choisie pour représenter les états et du degré d'indépendance vis-à-vis des plateformes d'implémentation : méta-modélisation des fonctions limitées aux états finis (énumérés), méta-modélisation partiellement dépendante des plateformes et méta-modélisation indépendante des plateformes.

- **Méta-modélisation des fonctions basées sur des états finis**

Ce premier type de solution est proposée par les méta-modèles ayant opté pour une méta-modélisation des états finis énumérés évoquée précédemment. Les fonctions de transition sont représentées par des méta-éléments Transition tels que les méta-classes InternalTransition et ExternalTransition dans [Lei et al. 2009], ou les méta-entités (InternalTransitionV2 et externalTransitionV2) dans AToM^3 V2 [Song 2006], doublement liées à un méta-élément State (source, destination) ou encore des méta-associations reflexives sur une méta-entité State (Internal_Transition et external_Transition) dans AToM^3 V1. Lors de la définition d'un modèle, les fonctions seront représentées en extension par l'énumération de liens entre états (ensemble de couples d'états).

Dans ces approches, les fonctions ta et Lambda sont en général représentées par des attributs du méta-élément State. Lors de la définition d'un modèle, chaque instance d'une méta-classe State et donc chaque état possèdera ainsi une durée de vie intrinsèque, et un message de sortie. C'est par exemple la solution choisie dans [Song 2006] et [Lei et al. 2009]. Cependant, la nature de l'attribut lié à Lambda ne semble pas définie de manière très précise.

Le principal avantage de ces solutions est qu'elles permettent une génération complète du code à partir de transformations M2T. En revanche, elles sont limitées à des transitions élémentaires et ne permettent pas de définir des logiques plus complexes.

Nous proposons également cette solution au niveau de notre méta-modèle BasicDEVS [Garredu et al. 2009] présenté dans le chapitre V pour décrire une restriction du formalisme DEVS. Ce formalisme simplifié nous sert à illustrer les transformations M2M à destination de DEVS.

- **Méta-modélisation partiellement dépendante des plateformes**

Ce type d'approche ne propose pas véritablement de solution complète au niveau du méta-modèle lui-même car elle se limite à identifier les éléments qui devront être entièrement définis ou complétés par du code lié à la plateforme choisie. Elle peut venir compléter une méta-modélisation basée sur des états finis pour introduire une logique comportementale plus complexe définie par programmation.

Les fonctions peuvent par exemple être représentées sous la forme de méthodes abstraites comme dans [Sarjoughian et al. 2012]. Une autre variante consiste à utiliser des méta-attributs textuels, comme dans [Song 2006] ou des balises de blocs de code, comme dans [Mittal et al. 2007a]. Lors de la définition d'un modèle, ces attributs resteront vides. Le

corps des fonctions devra être entièrement défini manuellement par le programmeur en fonction de la plateforme de destination choisie.

Dans une optique similaire, [Touraille et al. 2009a] propose une solution plus élaborée qui consiste à décrire partiellement le corps des fonctions. Il suggère l'utilisation d'un pseudo-langage indépendant pour définir les portions de code non liées aux plateformes. Le code généré à partir d'un modèle conforme à ce méta-modèle sera ainsi incomplet mais comportera néanmoins des éléments programmés générés contrairement aux approches évoquées précédemment. La tâche du programmeur se limitera ainsi à compléter le code généré au niveau des instructions spécifiques à la plateforme choisie.

Pour que la génération complète soit possible, il est envisageable de définir plusieurs fois la même fonction au moyen de différents langages (à choisir selon la plateforme de destination) mais dans tous les cas ces approches ne proposent pas de définition de la logique comportementale indépendante des plateformes.

- **Méta-modélisation indépendante des plateformes**

Cette dernière solution consiste à se doter de méta-éléments spécifiques permettant la création intégrale de règles comportementales définissant les fonctions indépendamment de toute plateforme. L'objectif final est de permettre une génération de code sans intervention du programmeur ou en limitant celle-ci au maximum. Les travaux introduits dans [Cetinkaya et al. 2012] se situent dans cette optique. Ils proposent d'employer un pseudo-langage de spécification de règles via le méta-attribut textuel Expression (de type Ecore EString). Ce pseudo-langage est décrit par un méta-modèle spécifique. Cependant, les caractéristiques de ce méta-modèle n'ont, à notre connaissance, pas encore été publiées et nous n'avons donc pas pu en étudier les détails.

Notre approche MetaDEVS se positionne clairement dans une optique similaire et notre contribution consiste à proposer une méta-modélisation de la logique comportementale indépendamment des plateformes à travers la définition de méta-éléments spécifiques. Nous introduisons ainsi la notion de DEVSRule pour décrire précisément les changements d'états, l'avancement du temps et les sorties d'un modèle.

Contrairement aux approches précédemment évoquées (méta-modélisation dépendante des plateformes) et bien que les objectifs soient assez similaires, notre approche MetaDEVS et celle Cetinkaya, nous semblent se placer dans un démarche IDM plus stricte. Nous considérons en effet que même si une génération complète semble encore difficile à obtenir, c'est néanmoins vers cela qu'il faut tendre en définissant des méta-modèles les plus riches et les plus extensibles possibles au niveau de méta-modélisation de la logique comportementale.

C'est dans cette optique que l'apport des outils et techniques de l'IDM pourra s'avérer le plus significatif dans le contexte de la modélisation et simulation DEVS.

b) *Transformations associées aux méta-modèles DEVS*

Le Tableau III-4 replace dans le contexte des transformations de modèles les méta-modèles présentés dans le Tableau III-2.

	Trasformation M2M		Transformation M2T vers une plateforme de simulation DEVS	
Approche	**Sens**	**Approche de transformation**	**Plateforme de destination**	**Approche de transformation**
DEVS to SMP2 [Lei et al. 2009]	DEVS → SDML	Hybride (QVT - ATL)	x	x
MDD4MS [Cetinkaya et al. 2012]	BPMN → DEVS	Hybride (QVT - ATL)	DEVSJAVA	Parcours de modèles (EMF Java)
AToM3 DEVS metamodel V1 [Posse et	SC → DEVS	Transformation de graphes (Python)	PyDEVS	Parcours de modèles (Python)
DEVSML [Mittal et al. 2012]	DSLs->DEVSML	Xtend	DEVSJAVA	Xpand
DEVSML [Risco-Martín et al. 2007b]	SC->DEVS-SM	Parseur XML (XSLT)	DEVS-XML	Parseur XML
SimStudio - DML [Touraille et al. 2010]	x	x	DML-Lang & DML-Sim	Parseur XML (XSLT)
EMF-DEVS [Sarjoughian et al. 2012]	x	x	DEVS-Suite	Parcours de modèles (EMF Java)
MetaDEVS	**FSM, BasicDEVS, etc… → DEVS**	**ATL**	**PyDEVS (et autres plateformes)**	**Template (Acceleo)**

Tableau III-4 : Comparaison des transformations associées aux méta-modèles DEVS

Au niveau des transformations M2M, nous nous situons dans une approche basée sur QVT et donc hybride, implémentée avec le langage ATL. Nous rejoignons ainsi les solutions proposées dans « DEVS to SMP2 » [Lei et al. 2009] et MDD4MS [Cetinkaya et al. 2012]. Cette approche suit une démarche totalement IDM et même MDA puisque la description des transformations se fait au niveau des modèles au moyen de méta-modèles alignés sur le méta-formalisme MOF (OCL et QVT), implémentés dans le langage hybride ATL.

D'autres approches situés dans des espaces technologiques différents que celui de MDA préfèrent employer XSLT (transformations de documents XML ou assimilés) [Risco-Martín et al. 2007b], dans l'espace technologique XML, ou le langage de programmation Python, dans l'espace technologique ER Diagrams – AToM^3 [Borland, 2003]. Enfin, certaines approches ne paraissent pas pour l'heure inclure des transformations de modèles (elles sont exclusivement M2T).

Concernant les transformations M2T depuis DEVS vers des plateformes particulières, la plupart des approches étudiées sont basées sur le parcours de modèles par programmation, en utilisant des langages de programmation orientés-objet : ces langages peuvent être Java, pour MDD4MS [Cetinkaya et al. 2012] et EMF-DEVS [Sarjoughian et al. 2012] ou Python [Borland, 2003]. En revanche, SimStudio [Touraille et al. 2010] et DEVSML [Risco-Martín et al. 2007b], situés dans l'espace technologique XML, utilisent pour le M2T, comme pour le M2M, l'analyse syntaxique des modèles afin de générer du code.

Pour notre part, dans MetaDEVS, nous avons fait le choix de générer du code avec une approche par Template, au moyen du plugin Acceleo pour Eclipse, qui implémente la spécification *Mof2Text*. Tout comme le langage M2M ATL, le langage M2T Acceleo autorise l'inclusion d'expressions OCL permettant de naviguer dans le modèle source et permet de rester dans une approche IDM et MDA.

SECONDE PARTIE

L’approche MetaDEVS

Chapitre IV. LE META-MODELE METADEVS

Ce chapitre ouvre la seconde partie de ce document, consacrée exclusivement à nos apports IDM pour le formalisme DEVS. Nous introduisons ici MetaDEVS, un méta-modèle pour le formalisme DEVS permettant de créer des modèles en utilisant tous les concepts de DEVS sans pour autant dépendre des plateformes de simulation.

L'architecture orientée modèle que nous proposons est basée sur l'utilisation de standards, techniques et outils issus de l'IDM, et en particulier de MDA, avec d'une part tout ce qui concerne la modélisation et la méta-modélisation, et d'autre part ce qui implique la mise en œuvre de transformations de modèles jusqu'à la génération de code. Son objectif est de faciliter l'interopérabilité des modèles DEVS à deux niveaux :

- transformation en modèles DEVS de modèles définis à l'aide d'autres formalismes de modélisation jouant le rôle de « formalismes d'interface »;
- implémentation des modèles DEVS dans différents environnements de simulation.

La définition d'un méta-modèle associé au formalisme DEVS constitue le point central de cette architecture. Il introduit une approche de représentation à la fois syntaxique et sémantique de modèles DEVS indépendante de toute plateforme, permettant la mise en œuvre des deux niveaux de transformations, modèle d'interface vers modèle DEVS d'une part et modèle DEVS vers code de simulation d'autre part.

Bien que des méta-modèles pour DEVS aient été déjà proposés, nous avons vu en 3.3 que, parmi ces derniers, seuls certains permettaient de manipuler tous les concepts propres aux fonctions DEVS. Le programmeur a dès lors la charge d'implémenter les fonctions (avec du code « pur », ou du code « hybride »). Le problème est que cela conduit à une perte d'indépendance vis-à-vis des plateformes de simulation, et nuit par conséquent à l'interopérabilité des modèles DEVS. Une solution vue en 3.2.3.c) propose l'utilisation de pseudo-code, sans toutefois détailler la nature de ce code.

Notre principale contribution, au regard des travaux existants, consiste donc à introduire, au sein de notre méta-modèle MetaDEVS, et donc sous forme de méta-classes, tous les concepts associés aux fonctions DEVS, et ce, indépendamment de tout langage et de toute plateforme de simulation DEVS. Ceci confère aux modèles issus de MetaDEVS une durée de vie qui n'est pas soumise à évolution des plateformes et des langages.

Ce chapitre s'organise de la manière suivante :

Nous évoquons tout d'abord les apports de l'IDM vis-à-vis de notre démarche et nous présentons l'approche que nous avons privilégiée ainsi que le formalisme de méta-méta-modélisation choisi.

Ensuite, en nous basant sur les définitions du formalisme DEVS, telles qu'elles ont été énoncées par le Pr. Zeigler, nous présentons les éléments principaux du méta-modèle. Nous détaillons ensuite ces éléments en procédant de manière progressive.

Nous décrivons comment nous avons choisi de représenter les états, les valeurs littérales, et comment nous contrôlons les types.

La définition de fonctions fait l'objet d'une section spécifique. Nous exposons les différentes options qui étaient possibles, et nous nous attachons à expliquer pourquoi l'une d'entre elles a eu notre préférence.

Enfin, dans une optique de validation, nous affinons notre méta-modèle MetaDEVS en lui associant des contraintes, dans le but d'éviter les erreurs d'interprétation et de limiter au maximum les facteurs susceptibles de nuire à l'interopérabilité.

Nous terminons ce chapitre par une présentation synthétique du méta modèle dans sa globalité sous la forme d'un diagramme de packages puis nous décrivons à quel niveau le langage de contraintes OCL peut intervenir pour affiner MetaDEVS.

4.1. Problématique de la méta-modélisation DEVS

4.1.1. Quels apports pour la modélisation DEVS ?

Comme nous l'avons vu tout au long du chapitre II, l'approche orientée modèle définie par l'IDM est un paradigme de développement logiciel centré sur la définition de modèles à différents niveaux d'abstraction et sur la mise en œuvre de transformations entre ces modèles. L'objectif principal est de parvenir à l'automatisation, même partielle, du processus de développement et de faciliter la portabilité des applications.

Dans le contexte de la modélisation et simulation DEVS, l'utilisation des concepts, techniques et outils issus de l'IDM offre des solutions aux problèmes de la réutilisabilité et l'interopérabilité de modèles.

La définition d'une architecture centrée sur les modèles, ayant pour élément central un méta-modèle du formalisme DEVS, permet de :

- proposer un format de représentation, à la fois syntaxique et sémantique, standardisé, fiable des modèles DEVS (fonctions y compris : fonctions de transition interne, externe, de sortie et d'avancement du temps) en faisant totalement abstraction de tout langage de programmation et de toute plateforme de simulation ;
- faciliter le stockage de modèles DEVS, leur édition, leur visualisation, leur modification, leur réutilisabilité ;
- rendre les modèles DEVS « productifs » grâce aux techniques de transformations de modèles (modèles vers modèles, modèles vers code) et assurer ainsi leur interopérabilité.

Dans le cadre d'une telle architecture, chaque modèle DEVS créé sera conforme au méta-modèle MetaDEVS et indépendant de tout environnement de modélisation et simulation. On pourra désormais parler de PIM (*Platform Independent Model*) DEVS pour désigner les modèles DEVS typés par ce méta-modèle.

4.1.2. Méta-méta-formalisme

Afin de pouvoir rendre, à terme, nos modèles productifs en réalisant des transformations, nous avons retenu l'environnement de méta-modélisation Eclipse EMF

(2.1.3.c)). Sans pour autant revenir en détail sur les caractéristiques de ce *framework*, rappelons tout de même celles qui ont conditionné notre choix :

- ce *framework* supporte à la fois la définition de méta-modèles et la génération de code,
- il possède pour cela deux méta-méta-modèles : le méta-méta-modèle Ecore qui est basé sur MOF 1.4 (2.1.3.c)),
- il est possible d'y créer des méta-modèles selon plusieurs vues : sous forme d'Ecore diagrams, ou alors avec l'éditeur classique (*Sample Ecore Model Editor*) qui offre lui aussi une interface graphique moins précise, mais donnant une bonne vue d'ensemble des éléments du projet, et même avec la vue de l'éditeur *OCLinEcore* pour ajouter directement des contraintes OCL au méta-modèle,
- on peut également en fonction de nos besoins créer un méta-modèle avec Ecore mais aussi Java et XMI qui sont supportés nativement par Eclipse EMF, voire au moyen d'autres méta-formalismes tels que Kermeta,
- il est facile de créer une instance dynamique à partir d'un méta-modèle, de la valider (établir qu'elle est conforme à ce dernier), de la modifier, de la stocker, ce qui facilite la mise au point d'un méta-modèle lors de la phase expérimentale,
- le format d'échange des modèles est par défaut XMI : le format d'échange de métadonnées de type UML basé sur XML.

En ce qui concerne le méta-méta-formalisme, notre choix s'est naturellement porté sur MOF et plus précisément sur Ecore, son implémentation dans l'environnement EMF-Eclipse. Dans la mesure où MOF ne bénéficie pas d'une syntaxe concrète spécifique, nous utilisons, pour donner une représentation graphique des concepts du méta-modèle, les diagrammes de classe UML. La Figure IV-1 montre comment le méta-modèle DEVS et les modèles DEVS s'inscrivent dans la hiérarchie de référence issue de l'IDM. Un modèle DEVS est vu comme « une instance » du méta-modèle DEVS qui est lui-même vu comme « une instance » du méta-méta-modèle MOF(ECORE).

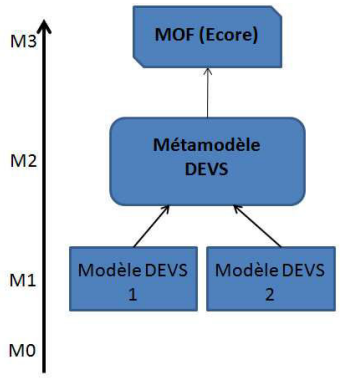


Figure IV-1 : Méta-modèle de DEVS et niveaux méta

4.1.3. Le méta-modèle DEVS : le pivot de notre approche

Disposer d'un méta-modèle de DEVS indépendant de toute plateforme conçu selon l'architecture IDM permet en premier lieu la spécification directe de modèles conformes à celui-ci, et donc homogènes, et améliore de ce fait leur interopérabilité. Mais en raisonnant

toujours dans un esprit IDM, nous allons voir que ce méta-modèle occupe également une place centrale dans notre approche car il peut être vu comme un carrefour vers lequel « convergent » des modèles hétérogènes et duquel « repartent » des modèles DEVS qui lui sont conformes et qui deviennent par conséquent homogènes eux aussi. Ce chapitre traite spécifiquement de la mise en place de ce pivot, mais pour mieux situer notre approche dans sa globalité, nous donnons ici des éléments que nous approfondirons dans le chapitre suivant.

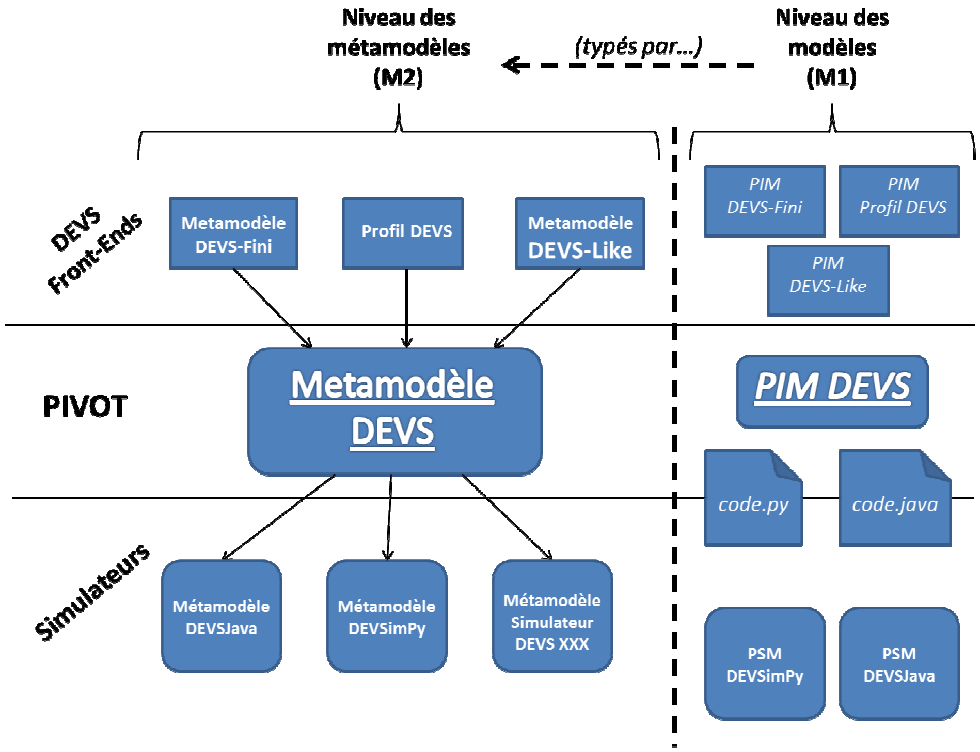


Figure IV-2 : Le méta-modèle de DEVS au centre de notre approche

La Figure IV-2 ci-dessus complète donc la figure présentée en début de chapitre, en illustrant le rôle de pivot du méta-modèle que nous allons définir par la suite. La partie gauche montre le niveau M2 et le rôle « unificateur » qu'occupe MetaDEVS. La partie de droite montre des modèles, instances de leurs méta-modèles respectifs.

La zone contenant les modèles PSM (*Platform Specific Model*) et le code est particulière : en effet, si en théorie, l'IDM fournit le concept de PSM, ce dernier n'est toutefois pas indispensable à la génération de code, tout dépend en fait du niveau de complexité du modèle que l'on désire transformer. Ce qui signifie en d'autres termes que, malgré le fait qu'un modèle de code soit de préférence généré à partir d'un PSM, il est tout à fait possible de générer un tel modèle, propre à une plateforme de simulation DEVS, directement à partir d'un PIM, pour peu que ce dernier soit « simple », et ne fasse pas appel à des fonctions ou des mécanismes trop spécifiques à un langage de programmation donné. Nous discutons plus en détail de ces possibilités au début du chapitre suivant consacré aux transformations de modèles.

Les modèles hétérogènes évoqués plus haut peuvent être exprimés :

- soit dans des langages de modélisation de type états-transitions par exemple, car nous avons vu que tout formalisme ou langage basé sur les états-transitions et à indications temporelles pouvait être transformé en DEVS

- soit des langages dits « d'interface », c'est-à-dire des front-ends de DEVS.

Dans tous les cas, il est bien entendu indispensable de disposer de leur méta-modèle pour pouvoir effectuer de telles transformations. Elles sont rendues possibles par l'approche IDM que nous suivons, qui permet de décrire des règles de transformation établissant des correspondances entre chaque méta-modèle source et le méta-modèle cible (qui sera toujours notre pivot), permettant de transformer un modèle « quelconque » en un PIM DEVS.

De la même manière, pour un modèle DEVS donné (peu importe sa provenance, qu'il ait été défini directement en respectant les règles de notre méta-modèle ou transformé selon celui-ci) l'approche IDM permet de le rendre productif en le transformant en PSM, et/ou en code objet directement utilisable dans un simulateur DEVS donné. Là encore, ce sont des transformations de modèles qui sont mises en oeuvre, faisant correspondre le méta-modèle de DEVS à un méta-modèle de simulateur, et en transformant un modèle DEVS en fichier texte contenant du code.

Toutes ces transformations seront largement détaillées dans les chapitres V et VI. Par ailleurs, le code complet de MetaDEVS, tel qu'il est utilisé pour toutes les transformations effectuées dans le cadre ce mémoire, figure en annexe 4 de ce document.

4.2. Architecture de base de MetaDEVS

En faisant abstraction pour le moment des éléments qui conditionnent le comportement et la structure détaillée d'un modèle DEVS, nous cherchons ici à représenter très simplement le fait que les modèles DEVS peuvent émettre et recevoir des informations avec le monde extérieur, nous dotant ainsi d'une base de travail pour la suite de la phase de méta-modélisation.

Nous donnons donc premièrement une représentation, provisoire, de l'architecture des modèles DEVS. Pour établir un méta-modèle de DEVS, il est absolument nécessaire de se baser le plus possible sur sa définition mathématique, de se laisser guider par elle, en essayant de faire preuve de l'abstraction nécessaire pour pourvoir la formaliser sous forme de diagrammes de classes, mais aussi en « redescendant » d'un niveau d'abstraction lorsque cela est nécessaire, pour bien vérifier que le maximum de cas peuvent être traités (dans le respect de l'adaptabilité et de la généricité de DEVS). C'est la raison pour laquelle nous « oscillerons » très souvent entre les niveaux M2 et M3 dans nos raisonnements.

Gardons aussi à l'esprit que DEVS s'implémente en langage orienté objet : MetaDEVS quoi que destiné à créer des PIM, pourra néanmoins contenir des éléments génériques communs à ces langages, ceci facilitera la phase de génération de code.

4.2.1. Contraintes globales

Fixons-nous dès à présent des contraintes générales qui faciliteront notre démarche de méta-modélisation. Certaines sont évidentes au vu de ce que nous avons évoqué jusque-là, d'autres en revanche le sont un peu moins et se rapportent plutôt à l'aspect « structurel » du méta-modèle que nous désirons obtenir.

Tout d'abord, rappelons que la vocation première de ce méta-modèle est de permettre la spécification intégrale de modèles DEVS tout en s'abstrayant de toute plateforme, il faut donc qu'il :

- respecte le plus possible le formalisme DEVS, nous devons donc trouver une manière de représenter le plus fidèlement possible les concepts de ce formalisme ;
- puisse exprimer des modèles DEVS dont la simulation doit être possible sur n'importe quelle plateforme (PIM) ;
- soit évolutif (ajout de fonctionnalités correspondant à des extensions de DEVS) ;
- soit modulaire, dans un souci de réutilisabilité et de partage.

Dans cette optique, nous choisissons d'utiliser des patterns de conception dès que cela est possible. Le méta-modèle étant situé au niveau d'abstraction M2, nous employons par souci d'exactitude le terme « méta-classe » plutôt que celui de « classe ».

Dans cette partie, les figures représentant les méta-classes de MetaDEVS seront de deux types :

- les méta-classes (et les hiérarchies dont elles peuvent faire partie) dont nous avons besoin pour illustrer nos propos, mais qui seront amenées à être modifiées par la suite
- les méta-classes « définitives »

Les premières seront présentées telles quelles. En revanche, dans un souci de clarté et de modularité, nous structurons systématiquement les secondes à l'intérieur de packages qui contiendront donc les hiérarchies de classes « définitives » de MetaDEVS. Nous les présentons par conséquent telles qu'elles sont organisées dans le méta-modèle créé dans l'environnement EMF Ecore.

4.2.2. **Hiérarchie des modèles**

Pour commencer, partons de la définition de base des modèles DEVS, à savoir :

$$\text{MA} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

$$\text{et MC} = \langle Y, Y, D, \{Md \mid d \in D\}, EIC, EOC, IC, select \rangle$$

Cette définition sera notre base de travail pour la construction de MetaDEVS.

La première étape est de chercher à représenter simplement les modèles DEVS en tant qu'éléments : pour ce faire, le choix de faire correspondre une méta-classe à un modèle DEVS semble le plus approprié. Nous déduisons de la définition de DEVS qu'un modèle basique peut être soit un modèle atomique (le plus « petit » modèle DEVS car ne contenant aucun autre modèle), soit un modèle couplé contenant quant à lui *plusieurs* sous-modèles. Aucune précision supplémentaire n'étant donnée concernant le contenu d'un modèle couplé, nous supposerons qu'un tel modèle contient *au moins un* sous-modèle basique.

Le cas où un modèle couplé contiendrait un seul sous-modèle ne peut être écarté, même si cela n'aurait d'influence que sur les couplages EIC et EOC (un modèle de type multiplexeur/démultiplexeur par exemple).

De plus, *D* informe sur le fait que tout modèle doit avoir un nom. Ainsi, un modèle couplé contenant 2 modèles atomiques *Message* et *Traduction* aura notamment pour valeurs : D = { Message, Traduction } et M_d = { $M_{Message}$,$M_{Traduction}$ }.

La contrainte résultante pourrait être exprimée comme suit : « tout modèle DEVS est soit un modèle atomique soit un modèle couplé et possède obligatoirement un nom. De plus, un modèle couplé contient au minimum un modèle DEVS ». Nous donnons ici une solution respectant la contrainte ci-dessus (Figure IV-3).

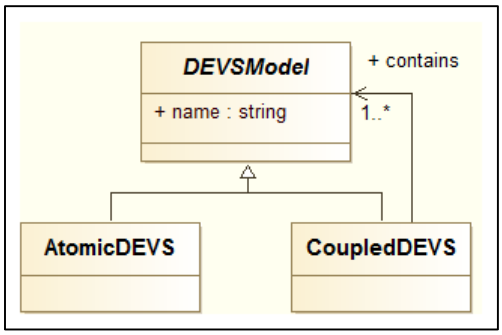


Figure IV-3 : Hiérarchie globale basique des modèles DEVS

Le fait d'utiliser des relations d'héritage est préférable dans ce cas (application du pattern de conception dit « composite »). Le nom du modèle est stocké dans un attribut de type Ecore String de la classe mère, qui est une classe abstraite. Nous disposons à présent du socle de MetaDEVS

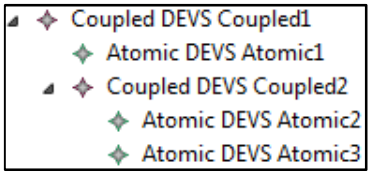


Figure IV-4 : Exemple de modèle couplé décrivant la structure vu en 1.3.2

À titre d'exemple, reprenons le modèle couplé présenté dans le premier chapitre (1.3.2) et créons-le avec MetaDEVS dans EMF (Figure IV-4).

4.2.3. Méta-classe *AtomicDEVS*

Nous présentons ici de manière sommaire les métaclasses intervenant dans la définition d'un modèle atomique DEVS afin de représenter plus précisément le but que nous désirons atteindre.

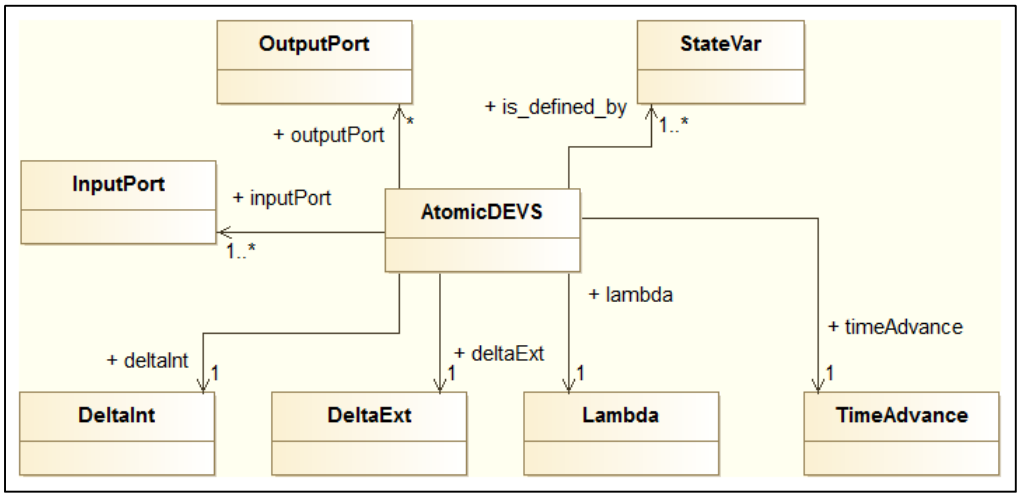


Figure IV-5 : Métaclasses d'un modèle atomique DEVS

Certes incomplet pour le moment, ce diagramme sera progressivement enrichi dans les pages suivantes. Il peut donc être appréhendé comme une sorte de « grille de lecture » ou de fil conducteur de notre démarche de méta-modélisation. Cette première présentation d'un modèle atomique avec ports, fonctions et variables d'état est directement issue de la définition formelle de DEVS (Figure IV-5).

Précisons que comme un modèle atomique ne possède pas obligatoirement de port d'entrée, la fonction δ_{ext} est certes nécessaire mais peut posséder un contenu nul.

4.3. Représentation des états

À partir du moment où on veut qu'il puisse permettre de générer des modèles effectuant des affectations, des comparaisons, des gestions d'évènements, le méta-modèle MetaDEVS devra être à même de gérer des valeurs littérales de types différents, une valeur littérale étant la «représentation explicite de la valeur d'un article qui ne doit être modifiée par aucune traduction du programme d'origine où elle figure» (définition Grand Dictionnaire Terminologique[1]).

Définir ces valeurs littérales et ces types de base nous permettra ensuite de proposer une manière de représenter les états DEVS qui soit aussi générique et efficace que possible, et même de gérer des expressions plus complexes, le tout s'inscrivant dans un ensemble plus large : la DEVSXpression.

4.3.1. Valeurs littérales et typage des données

Le méta-modèle que nous sommes en train de construire ayant pour vocation première de décrire des PIM DEVS, il est indispensable de pouvoir bénéficier d'un typage des données, indépendamment de toute plateforme. Ceci a pour conséquence que toute valeur littérale, toute variable d'état, tout élément manipulé dans une expression, en plus de sa valeur intrinsèque, doit être explicitement typé.

Dans MetaDEVS, ces derniers héritent d'une classe abstraite Type et sont au nombre de quatre, mais on peut facilement en rajouter grâce à la conception modulaire de

[1] www.granddictionnaire.com

l'architecture. Les types peuvent être représentés comme indiqué sur la Figure IV-6 et inclus dans le package Type sous leur forme définitive car ils ne seront plus remaniés par la suite.

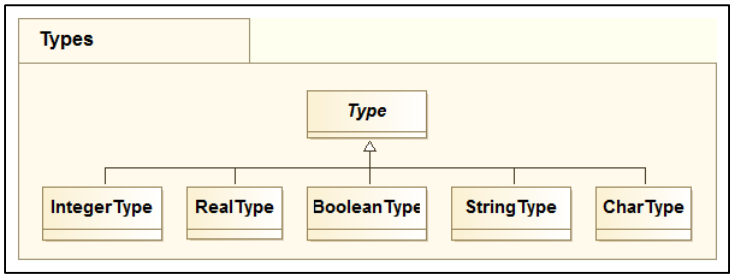


Figure IV-6 : Package des types

Ces classes ont vocation à être instanciées dans un modèle DEVS, au plus une fois chacune, en fonction des besoins de l'utilisateur : ce sont en fait des singletons qui serviront à décrire explicitement le type des valeurs littérales, mais aussi, plus généralement, de toute expression DEVS (exemple sur la Figure IV-7).

Nous avons choisi de limiter ce méta-modèle à la création de cinq sortes de types, et donc cinq sortes de valeurs littérales possibles, mais ceci n'est pas définitif et est fourni à titre d'exemple.

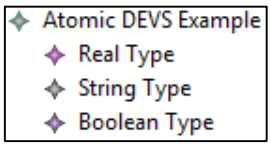


Figure IV-7 : Singletons de type *Real*, *String*, *Boolean*

La classe *LitteralBasicValue* est une classe abstraite de laquelle héritent les types littéraux, chacun possédant une valeur initiale sous forme d'attribut de base EMF Ecore. Ne perdons pas de vue que le méta-modèle que nous élaborons doit à tout moment pouvoir contrôler les types selon la hiérarchie présentée sur la Figure IV-6.

Tous les éléments susceptibles d'être manipulés dans le cadre d'un modèle conforme à ce méta-modèle devront donc être typés de la même manière. Il est par conséquent nécessaire de fournir, au moment de l'instanciation de chaque nouvelle valeur littérale dans le cadre d'un modèle DEVS, une référence vers un élément de notre hiérarchie de types. Cette référence se trouvant dans la classe mère abstraite *LitteralBasicValue*, elle est de ce fait transmise à toutes ses classes filles.

Cette référence (*is_always_typed*) vers un type nous sera également très utile lorsque ces littéraux participeront à des opérations plus complexes (comparaisons par exemple). La gestion des valeurs littérales se fait selon le diagramme de classes de la Figure IV-8.

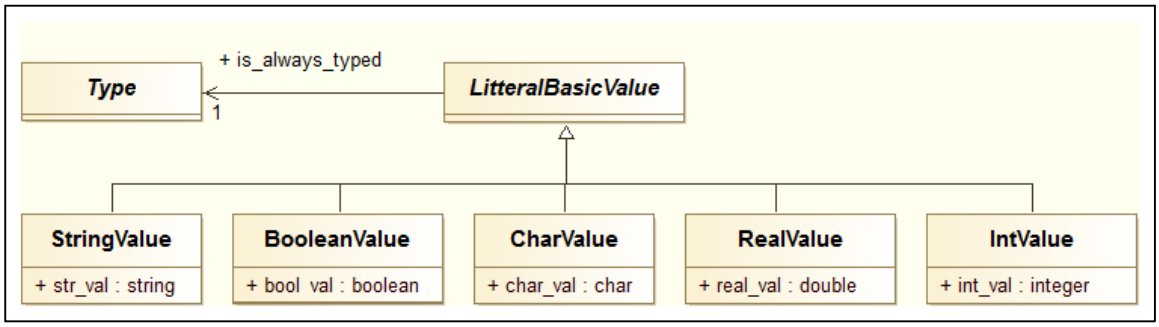


Figure IV-8 : Valeurs littérales de base

4.3.2. La notion d'état dans DEVS

D'un point de vue formel, l'état d'un modèle atomique DEVS à un instant donné est caractérisé par l'ensemble (fini ou pas) des valeurs de ses états possibles S (liés entre eux par des transitions déterministes). Ces états sont des valeurs biens distinctes, ce qui entraîne que chaque modification d'un état crée de facto un nouvel état.

Un état peut être caractérisé par une ou plusieurs variables discrètes :

- à valeurs dans un sous-ensemble fini de $E \in \mathbb{N}$ tel que $card(E) \leq card(\mathbb{N})$, par exemple le résultat d'un modulo,
- à valeurs dans n'importe quel autre ensemble dont les éléments sont finis, par exemple une énumération ;

Mais aussi par une ou plusieurs variables à valeurs dans $\mathbb{N}$, par exemple, comme le nombre d'individus d'une population, ou dans $\mathbb{R}$, qui sont des variables continues (1.1.5.d))

Ceci ne pose pas de problème particulier pour des états énumérables à valeurs discrètes :

- à valeurs textuelles prédéfinies, à valeurs numériques peu nombreuses…ect ;
- représentés par des valeurs booléennes ;
- dont l'évolution est prévisible, par exemple un modèle atomique d'horloge comptant les heures (modulo 12).

En effet, tous ces états sont connus et ne résultent pas de calculs plus complexes devant être effectués lors de la simulation : ils peuvent donc être explicitement énumérés durant la phase de modélisation (au sein de types énumérés par exemple, décrivant l'ensemble des valeurs possibles).

Pourtant, dans la pratique, il peut être très difficile, voire impossible, de lister les états possibles dans le cas où ceux-ci auraient des valeurs numériques appartenant à des ensembles infinis (par exemple des états à valeurs dans $[0;1] \in \mathbb{R}$).

Illustrons nos propos en examinant le cas d'un modèle atomique de feu de signalisation : on peut distinguer 3 états représentés par des chaînes de caractères : « vert », « orange », « rouge ». Ces états sont finis, peu nombreux et connus d'avance. Mais comment faire avec un modèle atomique représentant un réservoir contenant 50 litres de carburant se vidant au fur et à mesure de l'évolution du temps de simulation selon une formule complexe basée sur une équation différentielle ? En d'autres termes, comment gérer un modèle possédant un ensemble infini d'états ?

Pour contourner cette difficulté, nous avons fait le choix de représenter un état par une variable d'état pouvant prendre plusieurs valeurs suivant l'évolution du modèle. Chaque changement d'état se traduira par un changement de valeur de cette variable. Autrement dit, pour un même « type d'état », un modèle atomique ne manipulera pas autant d'états que de valeurs possibles, mais une seule variable, prenant ses valeurs dans l'ensemble des états possibles. Une telle variable doit par conséquent être nommée, et typée.

Concernant l'exemple du réservoir, nous aurons besoin d'une unique variable appelée « quantité », de type réel, initialisée à 50, et qui évoluera au fil du temps. En effet, tous les états possibles du système dépendent uniquement de la quantité de carburant.

Pour revenir à l'exemple du feu de signalisation, on pourrait décrire les états du système par une variable unique « couleur » de type chaîne de caractères, admettant les valeurs littérales textuelles suivantes « vert », « orange » et « rouge ». Une variable d'état, que nous nommerons StateVar, possède donc un identifiant, un type donné lors de l'instanciation, et un attribut correspondant à une valeur littérale, qui sera modifié au cours de la simulation. Cet attribut devra correspondre au type de la variable d'état (ceci peut être vérifié par une contrainte).

En utilisant les informations fournies par les diagrammes Figure IV-6 et Figure IV-8, nous possédons désormais tous les outils nécessaires pour décrire les états DEVS (Figure IV-9).

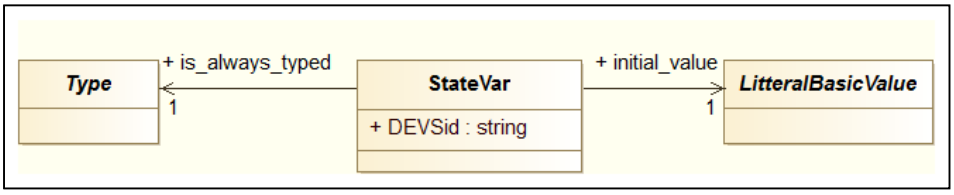


Figure IV-9 : Variable d'état

Nous pouvons à présent montrer à travers un exemple simple de capture d'écran (d'une partie d'un modèle atomique) comment les singletons *Type* et les valeurs littérales permettent de décrire une variable d'état (Figure IV-10).

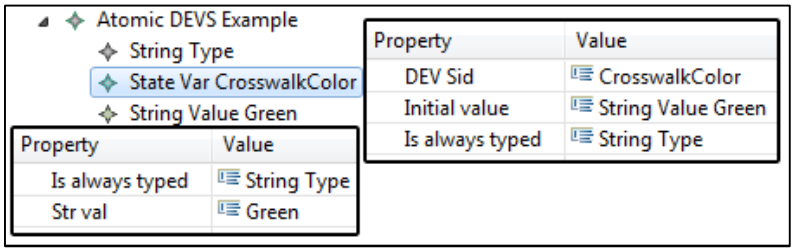


Figure IV-10 : *StateVar* décrivant la couleur d'un feu de signalisation

Ici, la *StateVar* (dont les propriétés sont représentées dans la fenêtre de droite) se nomme *CrosswalkColor*, elle est de type *String*, et on suppose qu'elle possède des états finis, énumérés, également de type *String*. Son état initial est la *StringValue* « *Green* » (dont les propriétés sont représentées dans la fenêtre en bas à gauche).

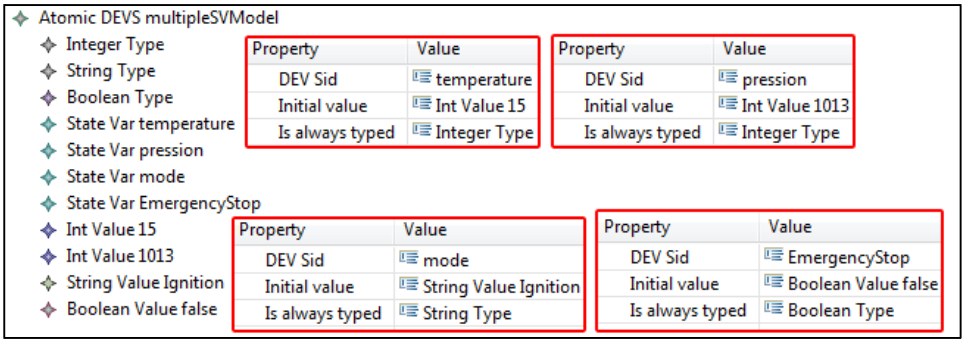


Figure IV-11 : *StateVar* multiples, décrivant un état multidimensionnel

MetaDEVS est également à même de gérer les états multidimensionnels, pouvant mélanger variables qualitatives et quantitatives. L'exemple suivant () montre une capture d'écran partielle d'un modèle comportant 4 variables d'état (de la gauche vers la droite et du haut vers le bas) : « température », « pression », « mode », « emergencyStop ». Les deux premières sont de type entier, la troisième de type chaîne de caractères, la dernière de type booléen.

Mais le fait d'être en présence d'états non dénombrables, donc d'une instance de StateVar admettant un grand nombre de valeurs possibles, implique lors de la définition de fonctions de pouvoir raisonner non plus sur une valeur mais sur des ensembles de valeurs afin de couvrir tous les cas possibles. En d'autres termes, il s'agit de borner un intervalle continu. Nous expliquons comment procéder en 4.5.4.

4.3.3. Variables d'état et expressions DEVS

Si on examine avec attention les diagrammes de la Figure IV-8 et de la Figure IV-9, on observe une redondance : en effet, une StateVar possède entre autres un type et une référence vers une LitteralBasicValue qui elle aussi possède un type.

Mise à part cette redondance, la relation entre une StateVar et une LitteralBasicValue est double : la première contient obligatoirement une référence vers la seconde, mais la seconde peut exister sans la première. Ces classes sont par conséquent étroitement liées mais nécessitent néanmoins d'avoir chacune une référence vers un type, et cela doit apparaître au niveau du méta-modèle. La solution la plus logique qui se dessine est de créer une super-classe dont elles hériteraient, ce qui permettrait de faire « migrer » l'attribut de type vers cette classe mère.

De plus, les variables d'état et les valeurs littérales ne sont pas les seuls éléments manipulés par un modèle DEVS. Pour que ce méta-modèle soit le plus riche possible, nous avons prévu la possibilité de gérer des expressions complexes. De telles expressions devront, pour pouvoir s'intégrer au méta-modèle, être elles aussi forcément typées par une classe fille de la classe abstraite Type. Une expression complexe devra hériter de la classe DEVSComplex. Nous entendons par expression complexe une expression impliquant par exemple plusieurs opérateurs, mais dont l'évaluation renvoie systématiquement une valeur typée. Il faut en outre qu'une expression complexe respecte l'esprit IDM en restant parfaitement indépendante de tout langage.

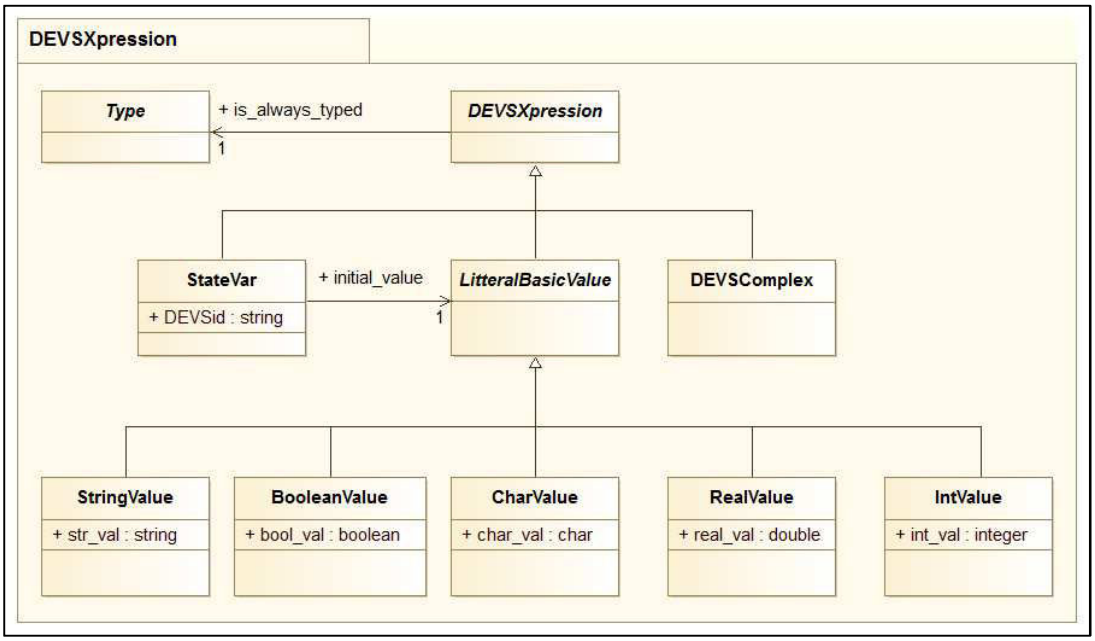


Figure IV-12 : Le package des expressions DEVS

Une valeur littérale, une variable d'état, ou une expression complexe sont trois sortes d'expressions différentes, mais font au final partie du même ensemble que nous avons choisi de qualifier d'expression DEVS : c'est tout simplement la super-classe que nous évoquions ci-dessus. La Figure IV-12 représente le package des expressions DEVS, ayant pour classe mère la classe DEVSXpression. Le concept de port, bien que faisant l'objet d'un traitement séparé (en 4.4.1) est également lié, par l'héritage, à la notion de DEVSXpression.

4.4. Représentation des ports et des couplages

4.4.1. Le package Port

X et Y désignent respectivement l'ensemble des évènements d'entrée (déclenchant δ_{ext}) et de sortie (contenus dans la fonction λ). Plus exactement, un évènement est un couple de type « port-valeur » soit :

- $X = \{(p,v) | p \in InputPorts \ , \ v \in X_p\}$ avec InputPorts désignant l'ensemble des ports d'entrée et X_p correspondant à l'ensemble des valeurs possibles sur ces ports
- $Y = \{(p,v) | p \in OutputPorts \ , \ v \in Y_p\}$ OutputPorts désignant l'ensemble des ports de sortie et Y_p correspondant à l'ensemble des valeurs possibles sur ces ports

En laissant pour l'instant de côté les valeurs possibles de ces ports, on peut exprimer la contrainte suivante : « tout modèle DEVS peut posséder zéro ou plusieurs ports d'entrée, et zéro ou plusieurs ports de sortie ». En effet, un modèle atomique qui ne possède pas de port d'entrée peut exister : la conséquence directe est que sa fonction δ_{ext} est nulle, donc ce modèle serait autonome et ne pourrait évoluer qu'en fonction de la durée de vie de ses états. De tels modèles existent pourtant, on les utilise par exemple comme « générateurs » chargés d'envoyer des informations sur leur(s) port(s) de sortie.

Tout comme un modèle, un port doit pouvoir être identifié, il doit donc posséder obligatoirement un champ *portID* de type String. Nous faisons le choix de distinguer les ports

selon qu'ils soient d'entrée ou de sortie. Les ports seront liés par la suite à la classe DEVSModel sous forme de références. Bien que nous fassions le choix de présenter leur package dans la partie dédiée aux modèles atomiques, les ports sont bien entendu des éléments communs aux modèles atomiques et couplés.

Même si, pour des raisons de clarté et de sémantique, de respect du formalisme DEVS, un port est lié à un modèle DEVS (atomique ou couplé) d'une manière distincte d'une DEVSXpression (propre aux modèles atomiques), et plus particulièrement d'une StateVar, il n'en reste pas moins évident que ce dernier possède un fonctionnement très similaire : il est amené selon son type à émettre ou recevoir des informations, ce qui se traduit par le fait qu'il change de valeur, et donc il se comporte comme une variable. On remarque d'ailleurs l'analogie « port=valeur » et « variable d'état = valeur ».

Aussi, à l'instar des autres expressions DEVS, il est nécessaire que les ports soient eux aussi typés. Nous choisissons donc de considérer un port comme une DEVSXpression, au même titre qu'une StateVar ou qu'une LitteralBasicValue. Les packages Port et DEVSXpression restent toutefois distincts.

La Figure IV-13 représente le package Port de MetaDEVS. Tout comme les autres DEVSXpression, un port doit être typé lors de son instanciation, ce qui permettra de le comparer (port d'entrée, dans le cas de la fonction DeltaExt), ou de lui affecter (port de sortie, dans le cas de la fonction Lambda), une DEVSXpression.

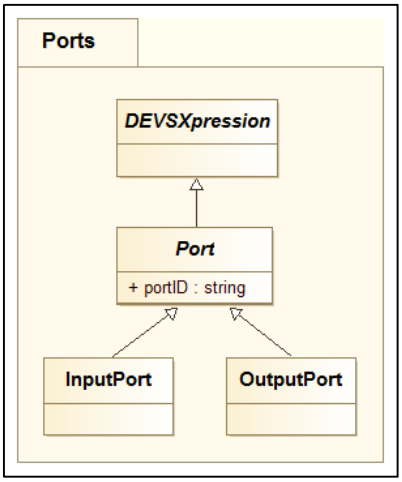


Figure IV-13 : Le package Ports

Toujours avec l'exemple du modèle couplé présenté dans le premier chapitre de ce document, nous modélisons *Atomic1*, en faisant apparaître ses ports (Figure IV-14).

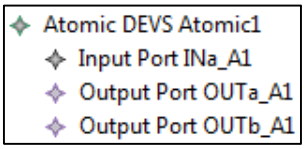


Figure IV-14 : Exemple de modèle AtomicDEVS avec ports

4.4.2. Le package Coupling

Examinons à présent de plus près les fonctions de couplage possibles d'un modèle couplé :

- Un couplage EIC met en relation deux ports d'entrée : l'un appartenant au modèle couplé lui-même, l'autre appartenant à l'un de ses sous-modèles $Md|d \in D$
- Un couplage EOC met en relation deux ports de sortie : l'un appartenant au modèle couplé lui-même, l'autre appartenant à l'un de ses sous-modèles $Md|d \in D$
- Un couplage IC met en relation un port de sortie et un port d'entrée, appartenant chacun à un des sous-modèles $Md|d \in D$ du modèle couplé

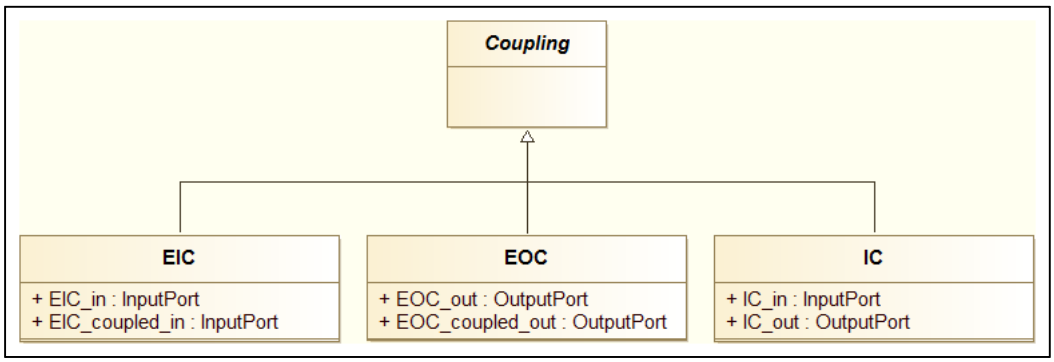


Figure IV-15 : Package des fonctions de couplage

En accord avec ce que nous avons défini pour les ports, seules les fonctions EOC et IC ne doivent pas être nulles, en vertu du fait qu'un modèle ne possède pas forcément de port d'entrée.

Nous ajoutons en 4.7.3 des contraintes OCL aux fonctions de couplage afin de les rendre plus précises et totalement conformes à l'énoncé ci-dessus, mais nous pouvons d'ores et déjà proposer la hiérarchie définitive suivante, en l'incluant dans un package, conteneur qui nous permettra d'avoir une vision plus claire de MetaDEVS (Figure IV-15).

Les trois fonctions de couplage sont référencées sous forme d'attributs dans la classe CoupledDEVS, en fonction des contraintes exprimées précédemment.

Pour conclure cette section, reprenons à nouveau l'exemple de modèle couplé *Coupled1* donné en 1.3.2. Nous le représentons cette fois-ci plus en détail sur la Figure IV-16, grâce à une capture d'écran recomposée, sur laquelle figure une représentation graphique du modèle en guise de rappel en haut à droite).

Nous montrons les sous-modèles qui le composent, tous les ports et toutes les fonctions de couplage (partie gauche). Pour le modèle Coupled2, nous donnons de plus le détail de ses quatre fonctions de couplage EIC, EIC, EOC, IC (voir les quatre encadrés en bas à droite).

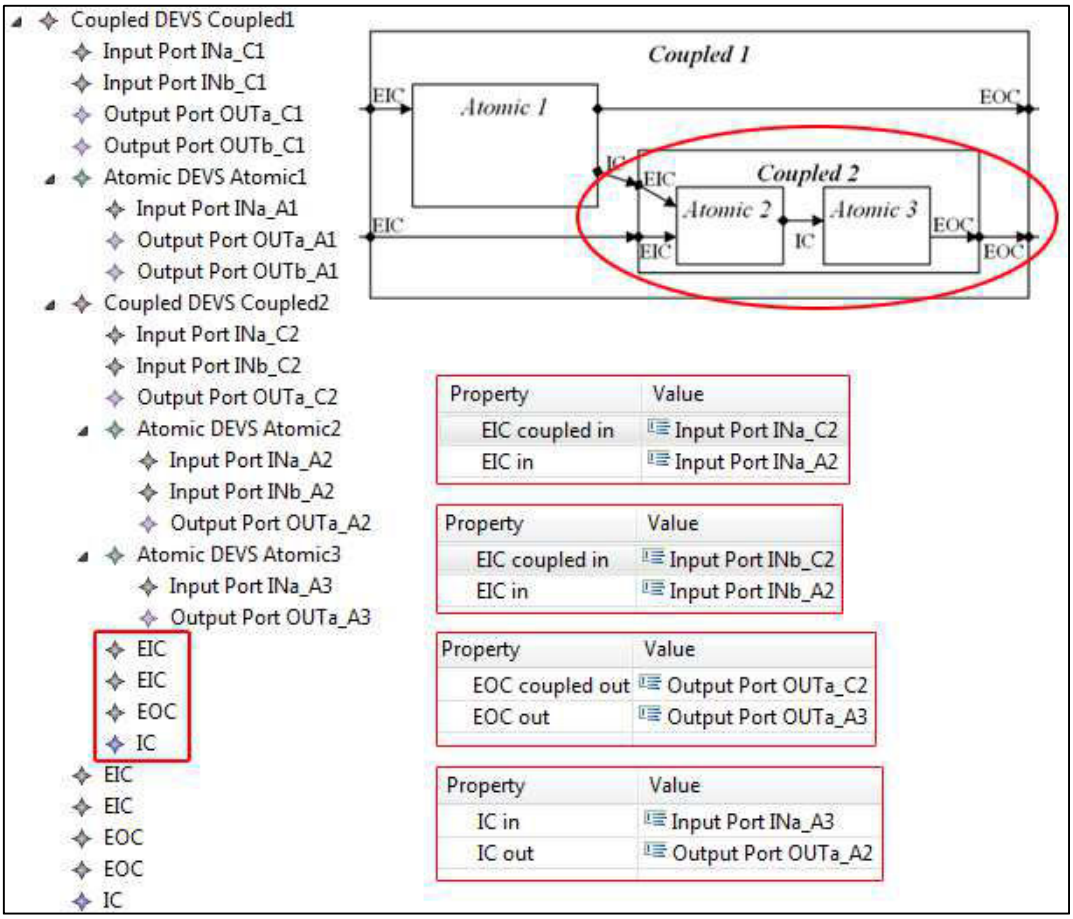


Figure IV-16 : Capture d'écran EMF recomposée de Coupled1

4.5. **Fonctions atomiques DEVS**

Maintenant que nous avons défini dans MetaDEVS les notions d'état et de port DEVS, que nous avons montré comment ces derniers étaient typés, et que nous avons structuré le tout grâce à la notion d'expression DEVS, il nous reste à décrire la dynamique d'un modèle dont le comportement est décrit à l'aide de fonctions DEVS. La gestion des fonctions constitue selon nous la plus grande difficulté lors de la méta-modélisation DEVS.

Dans la discussion suivante, nous énumérons les différentes possibilités que nous avons pour représenter des fonctions, puis nous présenterons notre solution : elle consiste à décrire un modèle atomique DEVS contenant obligatoirement chacune des quatre fonctions (transition interne et externe, avancement du temps, sortie) elles-mêmes composées d'une série de ce que nous appelons les « règles DEVS ». Nous verrons enfin qu'une règle DEVS, quelle que soit la fonction de base à laquelle elle se rapporte, se définit toujours de la même manière : avec des conditions et des actions (plus exactement, des descriptions d'actions).

Nous pouvons proposer dès à présent une hiérarchie de méta-classes correspondant aux différentes fonctions d'un modèle atomique DEVS (Figure IV-17).

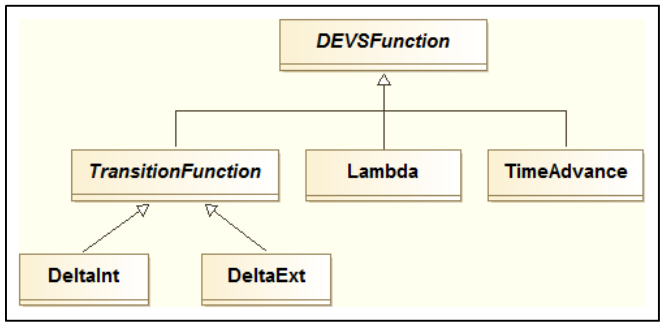


Figure IV-17 : Hiérarchie des fonctions atomiques DEVS

Chaque fonction est instanciée une et une seule fois dans chaque modèle atomique (Figure IV-18), elle peut par contre être vide (i.e. ne pas contenir de règles).

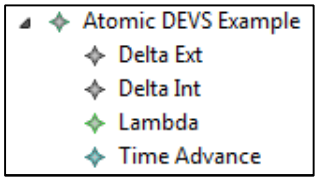


Figure IV-18 : Fonctions atomiques DEVS

4.5.1. Discussion sur les fonctions DEVS

Un modèle atomique décrit un comportement (changements d'états, interactions entre modèles via des ports d'entrée et de sortie…), et la description de ce comportement passe obligatoirement par la définition de fonctions :

- Une fonction de transition interne δ_{int} et une fonction de transition externe δ_{ext} provoquant toutes deux des changements d'état mais provoquées par des causes différentes,
- Une fonction de sortie λ un évènement (valeur) sur un port de sortie,
- Une fonction d'avancement du temps *ta*.

Dans le but de représenter les quatre fonctions de base d'un modèle atomique DEVS, quelles possibilités pouvons-nous envisager ? Commençons par lister ces dernières, par ordre de difficulté de mise en œuvre croissant :

- Codage des fonctions « en dur », sous formes de chaînes de caractères, en utilisant un langage de programmation et des méthodes, fonctions…compatibles avec la plateforme de destination,
- Mêler fragments de code objet et fonctions génériques,
- Inclure du code objet (issu d'un langage dont on dispose du méta-modèle) sous forme de modèles de code,
- Utiliser un langage de règles simple, souple, de compréhension facile,
- Définir un pseudo-langage complet, structuré, de type impératif.

Les deux premières possibilités correspondent aux solutions que nous avons présentées au chapitre précédent (partie 3) : malgré de légères différences, elles reposent sur le même principe. Elles sont définies par l'introduction de code « en dur » (fragments de code

ou code complet) dans les modèles DEVS (à l'aide d'un code semi-générique). Bien que relativement simples à appliquer, surtout la première, elles ne semblent pas parfaitement adaptées à une approche se voulant totalement MDA, et n'offrent que trop peu de contrôle sur les modèles.

La troisième possibilité est quant à elle plus séduisante : combiner dans un environnement de méta-modélisation MetaDEVS avec celui d'un langage de programmation orienté objet. Ceci permettrait d'avoir plus de contrôle sur les types, les algorithmes des fonctions…et surtout de bénéficier en partie des mécanismes MDA : le code n'étant plus écrit sous forme de chaînes de caractères mais introduit selon des modèles typés au niveau supérieur par le méta-modèle du langage, il deviendrait alors possible, mais difficile, d'effectuer des transformations vers d'autres langages et d'autres plateformes. Pourtant, ceci reste lourd à mettre en œuvre, de par les spécificités de chaque langage orienté objet, et nuirait tout de même fortement à la capacité de tels modèles à être compris (ceci impliquant de bien connaître le langage dans lequel les fonctions ont été programmées), mais aussi modifiés, réutilisés, échangés.

L'utilisation des solutions ci-dessus a pour conséquence la perte partielle voire totale d'indépendance des modèles par rapport à un langage de programmation, ou une plateforme.

Les deux dernières solutions semblent donc être les plus pertinentes : elles supposent toutes deux la création d'un méta-modèle spécifique, ou bien l'enrichissement de celui que nous possédons déjà. Pourtant, si la pénultième peut paraître particulièrement opportune en étant privilégiée dans le cadre d'un formalisme d'interface simplifié, elle reste cependant peu adaptée au formalisme DEVS et limiterait considérablement sa puissance d'expression. Nous privilégierons les idées émises dans la dernière solution.

4.5.2. **La notion de règle DEVS**

Poursuivons maintenant la discussion précédente, en tenant compte de ce que nous avons dit et de ce qui est illustré par la figure Figure IV-17.

Un point de départ intéressant est de déterminer les points communs des différentes fonctions (δ_{int},δ_{ext},λ et *ta*) d'un modèle atomique DEVS : lorsque l'on examine la description d'un tel modèle, qu'elle soit mathématique, ou en pseudo-code, on remarque que chaque fonction peut décrire : des tests, des envois de messages, des actions sur des variables,…etc.

Malgré plusieurs différences entre les fonctions, ces descriptions peuvent tout de même être regroupées au sein d'ensembles plus larges, des sortes de schémas, qui sont en fait des énumérations. Par exemple, la fonction d'avancement du temps *ta* énumère une liste de durées correspondant à un état, la fonction δ_{int} énumère des passages d'un état à un autre, etc…nous appellerons dorénavant de telles énumérations des « règles DEVS ».

Une règle a pour but de décrire, suivant la fonction qui la porte, un ensemble d'opérations se rapportant à des éléments précis. Chaque fonction DEVS est donc constituée d'une ou plusieurs règles, à l'exception de la fonction de transition externe qui peut ne pas en contenir (car un modèle atomique ne possède pas forcément de port d'entrée). Avant d'examiner en détail la composition d'une règle, nous pouvons d'ores et déjà présenter le package des fonctions des modèles atomiques DEVS (Figure IV-19), améliorant le

diagramme Figure IV-17, puisque prenant cette fois-ci en compte le lien entre fonctions et règles DEVS.

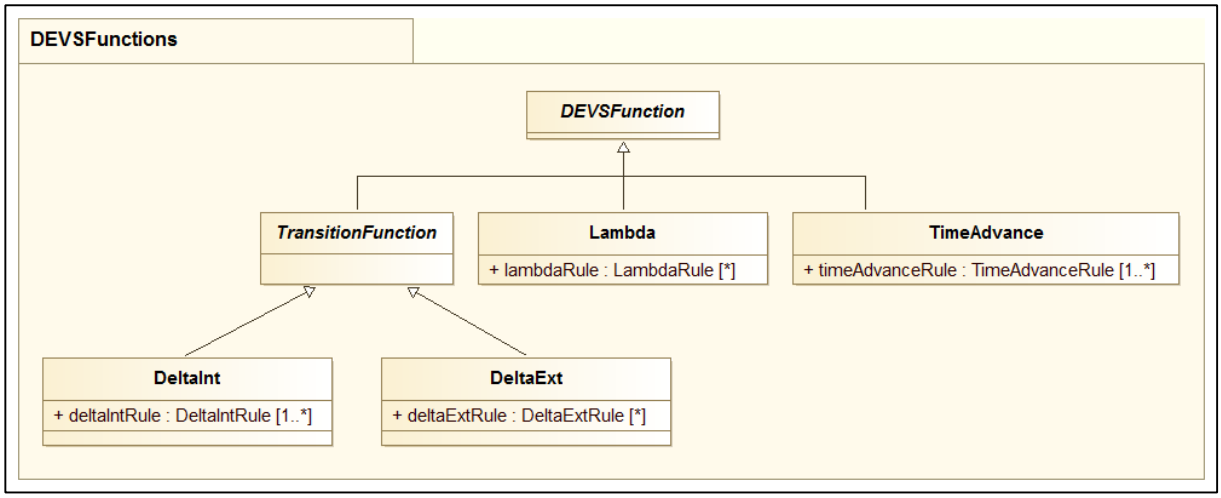


Figure IV-19 : Package des fonctions DEVS

4.5.3. Comment se construit une règle DEVS ?

Nous savons qu'un modèle atomique DEVS contient une et une seule fonction de chaque type (δ_{int},δ_{ext},λ et ta), chaque fonction contenant elle-même un certain nombre de règles, déclinées sous forme d'énumérations. Comment devons-nous organiser ces énumérations ? Toujours en suivant notre philosophie qui consiste à utiliser comme point de départ la définition mathématique des modèles DEVS, essayons de trouver les points communs et les différences des règles des quatre fonctions d'un modèle atomique.

- $ta: S \rightarrow \mathbb{R}_0^+ \cup +\infty$, apparaît comme la fonction contenant les règles les plus simples. L'ensemble de départ est les états, et à chaque état une TimeAdvanceRule doit associer un réel positif symbolisant la durée de vie, par exemple : $ta(S_1)=66$. Il y aura donc au maximum autant de règles que d'états.

- $\lambda: S \rightarrow Y$, avec $Y = \{(p,v) | p \in OutputPorts\ ,\ v \in Y_p\}$ n'est guère plus compliquée, elle associe à chaque état un évènement de sortie (port + valeur). Cet évènement associe un port de sortie à une valeur littérale, pouvant être le résultat d'un calcul. Ici encore, il y aura au maximum autant d'instances de LambdaRule que d'états possibles.

- $\delta_{int}: S \rightarrow S$, associe à chaque état un autre état (pouvant être lui-même). δ_{int} résultant de l'expiration de la durée de vie d'un état, elle possède au plus, à l'instar des fonctions précédentes, autant de règles, ou transitions, que d'états.

- $\delta_{ext}: Q \times X \rightarrow S$ avec $Q = \{(s,e)/s \in S,\ e \in [0, ta(s)]\}$, associe à un couple état-évènement un autre état. Un tel évènement est caractérisé par un couple composé d'un port d'entrée et d'une valeur littérale. Le nombre de règles peut être nul (si le modèle ne possède pas de port d'entrée), mais n'est pas lié au nombre d'états (il peut y avoir de nombreux évènements différents sur les ports d'entrée d'un modèle atomique, représentant chacun une règle différente).

Le point commun de toutes ces fonctions est que leurs règles tiennent compte de l'état dans lequel se trouve le système, ce qui est d'ailleurs logique pour les fonctions d'un modèle

dont la vocation est de décrire un comportement. Chaque règle sera par conséquent composée, entre autres, d'un test.

Le point commun aux deux dernières fonctions est que leur vocation est de définir des transitions, autrement dit des changements d'états : cela peut se traduire par la création d'une nouvelle valeur littérale liée à une StateVar.

λ va quant à elle, après un test d'état, affecter une valeur littérale à un port de sortie, du même type que ce dernier, tandis que δ_{ext} va, en plus d'un test d'état, tester si un port d'entrée a pour valeur un certain littéral (de même type). Pour clarifier les choses, consignons les remarques ci-dessus dans un tableau (Tableau IV-1) qui récapitule, pour chaque fonction, de quoi sont composées les règles qui lui sont associées.

	Test (condition)		***Description d'action***		
	Test d'état	*Test de port*	*Renvoi de valeur*	*Changement d'état*	*Envoi de message*
δ_{int}	*OUI*			*OUI*	
δ_{ext}	*OUI*	*OUI*		*OUI*	
λ	*OUI*				*OUI*
ta	*OUI*		*OUI*		

Tableau IV-1: Fonctions DEVS et opérations associées (tableau simplifié)

Nous connaissons désormais la composition d'une règle DEVS : ce qui est mis en évidence par ce tableau est qu'une règle se compose toujours d'au moins une condition et d'une action. Ce tableau est également en accord avec la définition mathématique du formalisme en termes d'ensembles de départ et d'arrivée et valide ainsi notre raisonnement : il nous servira de trame pour la définition des actions et des tests.

4.5.4. Les conditions et les actions

Une condition, dans son expression la plus simple, n'est rien d'autre qu'un test effectué sur une variable d'état ou un port d'entrée alors qu'une action simple se traduit quant à elle par la modification ou le renvoi d'une valeur. Bien entendu, nous devons garder à l'esprit que comme un modèle peut être décrit par plus d'une StateVar, les conditions et les actions peuvent porter non pas sur une mais sur plusieurs StateVar.

De plus, ces conditions et ces actions peuvent être plus compliquées, une condition pouvant par exemple inclure plusieurs autres conditions imbriquées, et une action pouvant appeler des fonctions plus élaborées, mais, quoi qu'il en soit, elles seront toujours utilisées de la même manière dans une règle.

Une condition doit comporter un membre gauche, un opérateur binaire de comparaison, et un membre droit.

Dans le cas d'une condition simple, le membre gauche aura pour valeur la StateVar ou l'instance de l'InputPort (il nous faudra créer deux classes de test différentes).

Le membre droit pourra être une valeur littérale de type LitteralBasicValue, ou bien un autre état (donc une DEVSXpression) devant être de même type que le membre gauche (voir

4.7.4). Une comparaison sur un port ou sur un état ne diffèreront donc que par leur membre gauche (elles auront la même classe mère).

Pour résumer :

Un test simple (ou condition simple), sur une variable d'état, est de la forme suivante : `[StateVar] <opérateur> [DEVSXpression]` tandis qu'un test simple (ou condition simple), sur un port d'entrée, est de la forme suivante : `[InputPort] <opérateur> [DEVSXpression]`.

Le premier opérateur binaire de comparaison dont nous avons besoin est naturellement l'opérateur d'égalité =, il sera utilisé pour sélectionner un état précisément lors de la création de la règle. Par exemple, dans le modèle représentant un feu de circulation, on pourra écrire les 3 fonctions *ta* de la sorte : *ta*(« rouge »)=30, *ta*(« orange »)=5, *ta*(« vert »)=35.

Or nous nous trouverons parfois en présence d'états nombreux voire d'états non finis. Comment faire pour les utiliser lors de nos tests ?

Nous proposons comme solution de permettre de désigner un état dans une fonction DEVS non plus par une valeur littérale exacte associée à son identifiant (et l'opérateur d'égalité) mais par des ensembles de valeurs littérales, ces ensembles que l'on appellera E étant des sous-ensembles de S dont l'union couvre la totalité des états possibles. Ces sous-ensembles sont définis tels que $E_1 \cup E_2 \cup \ldots \cup E_n = S$.

Pour obtenir de tels sous-ensembles, la solution la plus simple consiste à enrichir notre liste d'opérateurs binaires de comparaison avec l'opérateur de supériorité stricte >, d'infériorité stricte <, de supériorité ≥, d'infériorité ≤.

Ces opérateurs jouent un rôle crucial dans la gestion des fonctions d'un modèle, en particulier pour un modèle à états non finis : ils permettent de comparer (supériorité, infériorité, test d'intervalle…etc.) la valeur littérale d'une instance de StateVar (ou d'InputPort) à une ou plusieurs autres valeurs manipulées par le modèle. Bien entendu ils ne peuvent être utilisés qu'avec des valeurs numériques.

Afin de conserver le déterminisme de DEVS, dans une fonction, il ne peut exister plus d'une règle possédant exactement la même condition et conduisant à des actions différentes. Par exemple, pour le feu de signalisation, il est impossible d'avoir : δ_{int}(« vert »)= « rouge » et δ_{int}(« vert »)= « orange ». En d'autres termes, les sous-ensembles E de S ne doivent pas avoir d'éléments commun et donc : $E_1 \cap E_2 \cap \ldots \cap E_n = \emptyset$.

Nous avons discuté en 4.5.3 du nombre maximum de règles possibles pour chaque fonction, explicitions ici nos propos :

- le modèle représentant une horloge « modulo 12 » possède 12 états finis numérotés de 0 à 11, décrits par une StateVar nommée « heure », de type entier, pouvant contenir une référence vers une LitteralBasicValue. On peut subdiviser l'intervalle des valeurs possibles en deux sous-intervalles : E_1=[0 ;5] E_2=[6 ;11] (donc $E_1 \cup E_2 = S$ et : $E_1 \cap E_2 = \emptyset$). Ainsi, écrire dans une fonction d'une part : heure=2 ou heure=4, etc…et d'autre part heure≤6 ou heure>6 aura une portée différente : utiliser les intervalles permet en fait de sélectionner un ensemble d'états pour leur attribuer un comportement commun ;

– le modèle représentant le réservoir de 50 litres possède une variable d'état appelée « quantité » possédant une référence vers une LitteralBasicValue de type réel appartenant à l'intervalle [0;50]. On pourrait par exemple subdiviser cet intervalle en E_1=[0;5[, E_2=[5;45] , E_3=]45 ;50] ce qui se traduirait dans les règles d'une fonction par les tests suivants : 0≤quantité<5, 5≤quantité≤45, 45<quantité≤50. Procéder ainsi équivaut à ramener ce modèle à états infinis à un modèle à états finis, inclus dans des sous-ensembles de S. Ces trois intervalles sont utilisés exactement de la même manière que s'ils désignaient des états, puisque leur union contient tous les *s* possibles de S car $E_1 \cup E_2 \cup E_3 = S$ et leur intersection est vide : $E_1 \cap E_2 \cap E_3 = \emptyset$. Un tel encadrement dépasse un peu le cadre d'une condition simple (de type opérande-opérateur-opérande) et sera réservé à la description de conditions complexes.

Nous avons fait le choix de représenter ces 5 opérateurs par un type énuméré (Figure IV-20).

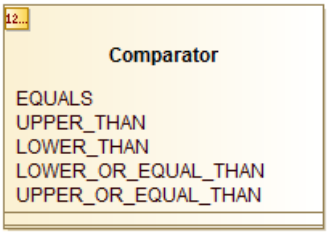


Figure IV-20 : Type énuméré pour les comparaisons

La Figure IV-21 présente le package Condition permettant de créer un bloc conditionnel simple et incluant les classes *ComplexStateVarComparison* et *ComplexInputPortComparison* (grisées) prêtes à accueillir des attributs et des références permettant de gérer les conditions plus complexes (nous en avons donné un aperçu avec l'exemple du réservoir ci-dessus) ou même des imbrications de bloc conditionnels. Manipuler ces classes ne changera rien aux transformations déjà présentes, ce qui confère à MetaDEVS une bonne extensibilité.

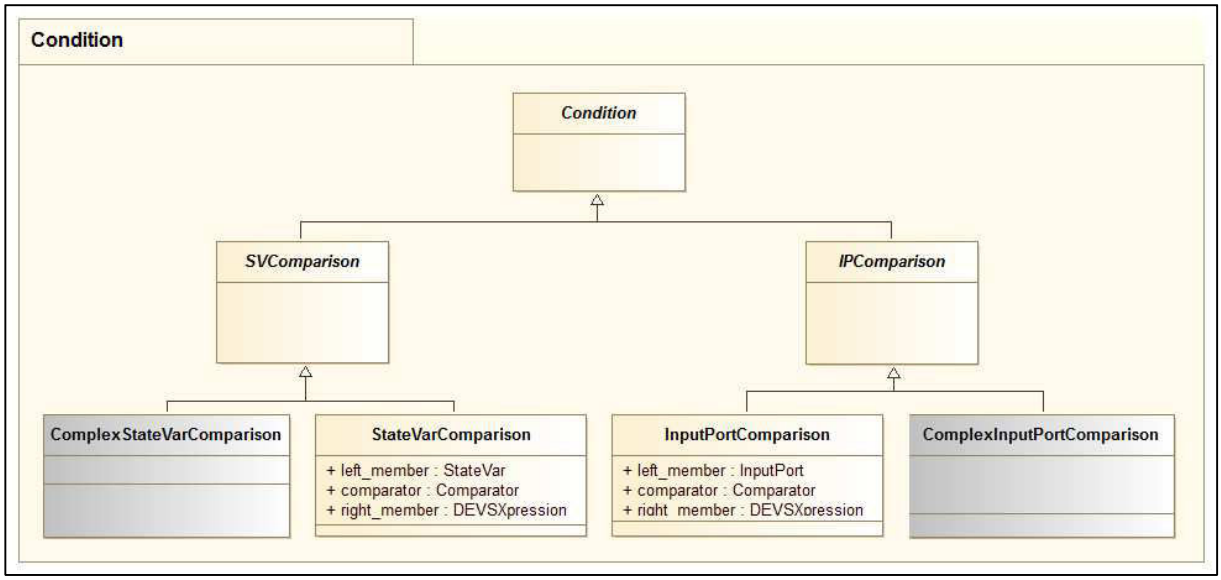


Figure IV-21 : Package des conditions

Cette possibilité d'extensibilité existe également, comme nous allons le voir, au niveau des actions.

La Figure IV-22 montre les propriétés d'une condition simple (*StateVarComparison*) qui teste si la *StateVar* « Color » a pour valeur la *StringValue* « geen ».

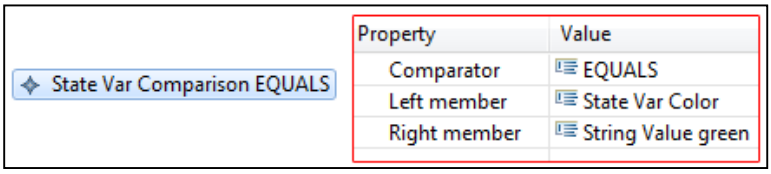


Figure IV-22 : Une condition simple d'égalité

Autre exemple de condition simple, une *InputPortComparison* qui teste si la valeur d'un port d'entrée est inférieure ou égale à 25. Cet exemple est montré sur la Figure IV-23.

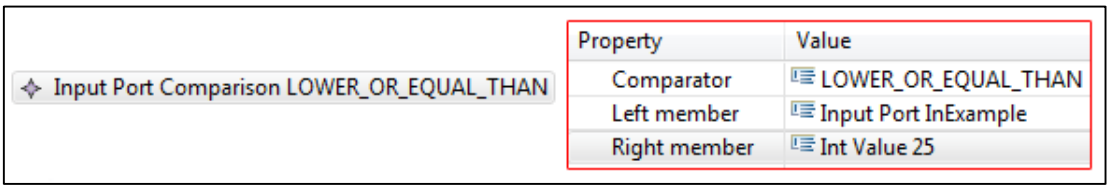


Figure IV-23 : Une condition simple portant sur un port

Une règle peut contenir plusieurs conditions, portant par exemple chacune sur une StateVar différente, ou sur la même StateVar (si on veut l' « encadrer » entre deux valeurs par exemple) : elles sont liées par défaut entre elles au moyen de l'opérateur logique ET.

Toujours selon le Tableau IV-1, une action se traduit par la spécification d'une modification d'au moins une valeur (excepté pour la fonction *ta*) d'une variable d'état ou d'un port de sortie, l'opérateur implicitement utilisé étant l'opérateur d'affectation.

Cette modification, sous sa forme la plus simple (Figure IV-24), concerne d'une part une StateVar ou un OutputPort et d'autre part une DEVSXpression (de même type). Une règle peut en contenir plusieurs, portant sur des variables d'état différentes.

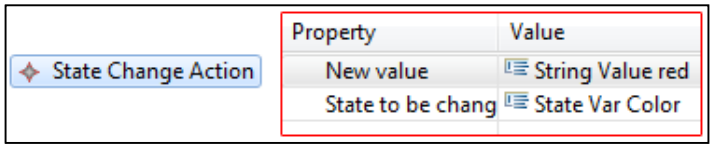


Figure IV-24 : Une action simple (modification textuelle)

Elle pourra aussi, à terme, impliquer une opération un peu plus élaborée faisant référence de part et d'autre de l'opérateur à une variable d'état ou un port, par exemple : `heure=heure+1`. Elle pourra enfin déclencher des calculs faisant intervenir d'autres éléments du modèle, utilisant des structures conditionnelles…etc.

Il est nécessaire, même si l'état ou le port sont explicitement mentionnés dans le test de la règle, de rappeler lors de l'action quel élément devra être affecté par une modification de valeur : ce dernier n'est pas forcément le même que celui utilisé lors du test.

Nous avons fait le choix de simplifier le tableau III.21 afin de séparer clairement les tests des actions, mais il faut garder à l'esprit que la fonction d'avancement du temps *ta* ne génère pas d'action à proprement parler, car elle n'agit pas sur une valeur, elle ne fait que la

renvoyer. Nous l'excluons volontairement du package Action pour marquer cette différence, et nous verrons plus loin comment la gérer.

La Figure IV-25 présente le package des actions, avec les classes abstraites *ComplexSCA* et *ComplexOA* qui sont prévues pour gérer des actions complexes : elles constituent un des points d'extensibilité de MetaDEVS.

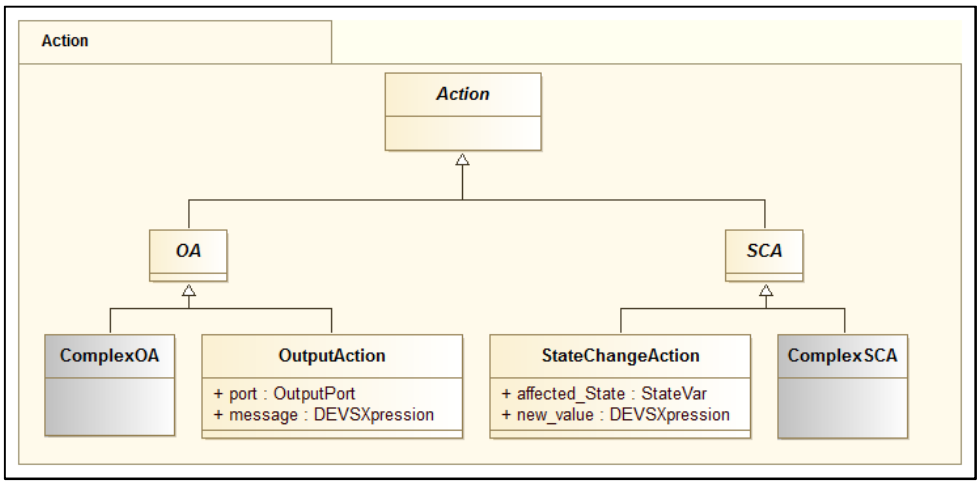


Figure IV-25 : Package des Actions

4.5.5. Le package des règles

Comme nous le savons déjà, une règle se compose toujours d'au moins une condition et presque toujours d'une action (au moins), et selon la Figure IV-19 une fonction est constituée par un ensemble de règles.

Afin de pouvoir gérer le cas où un modèle DEVS serait décrit par plus d'une *StateVar*, ou qu'il possède plus d'un port d'entrée et/ou plus d'un port de sortie, il faut prévoir qu'une règle puisse porter sur plusieurs *StateVarComparison* (et dans le cas d'une *DeltaExtRule*, plusieurs *InputPortComparison*), et, de la même manière, sur plusieurs *StateChangeAction* (dans le cas d'une *TransitionFunctionRule*). Une fonction DEVS se compose donc d'un ensemble de règles, et chacune de ses règles possède au moins une condition et au moins une action. Implicitement, s'il y a plusieurs conditions, elles sont liées entre elles par un « et » logique.

Concernant le cas de la fonction d'avancement du temps *ta*, nous pensons qu'il est plus pertinent d'inclure directement, dans chaque règle définissant cette fonction, un attribut Ecore de type EInt qui correspond à la valeur renvoyée par la fonction. Les règles de type *TimeAdvanceRule* sont ainsi constituées de l'attribut *ta_value* d'une part et d'au moins une *StateVarComparison* définissant l'état ou les états concernés, dont *ta_value* symbolise la durée de vie. La Figure IV-26 représente un exemple de fonction d'avancement du temps composée deux règles de type *StateVarComparison*.

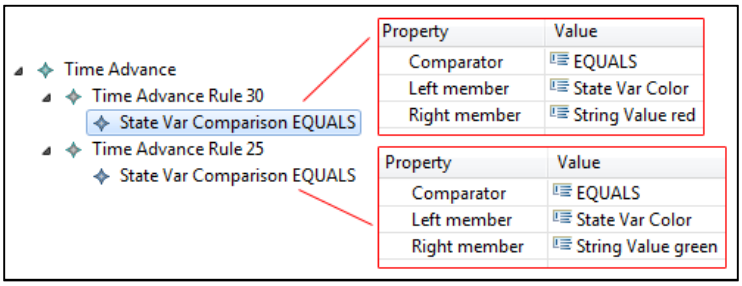


Figure IV-26 : Exemple de *TimeAdvanceRule*

La première teste si la variable *Color* contient la StringValue « *red* », la durée de vie qui lui est associée est de 30 unités de temps. La seconde teste si la variable *Color* contient la StringValue « *green* », la durée de vie qui lui est associée est de 25 unités de temps.

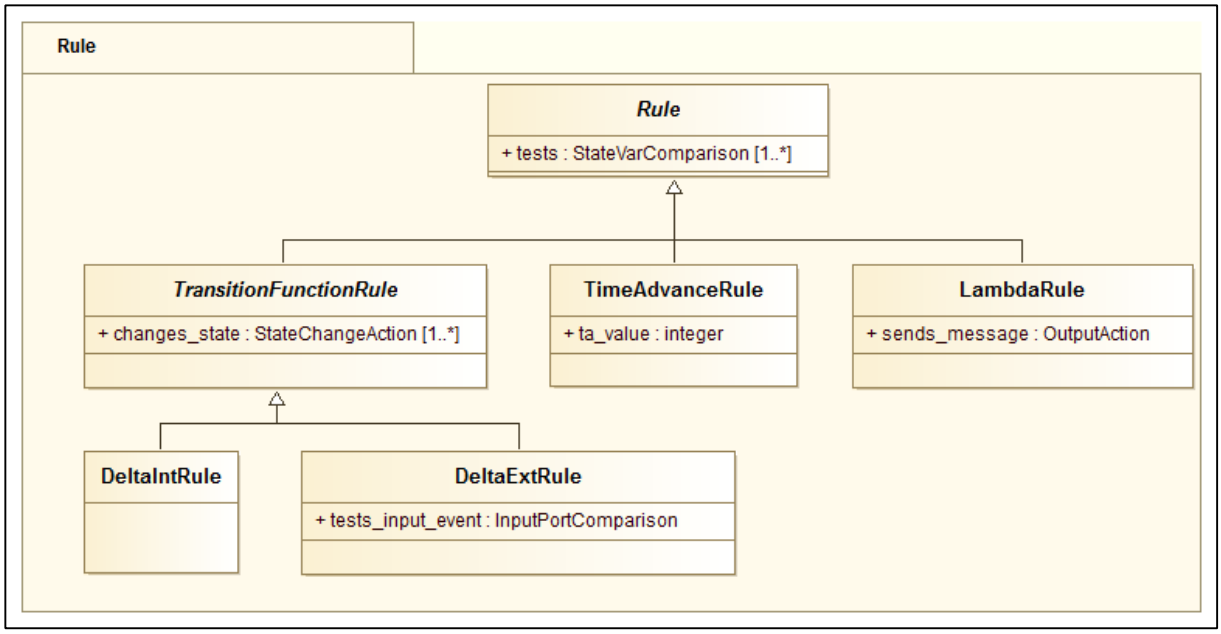


Figure IV-27 : Package des règles

Nous disposons maintenant de tous les éléments nécessaires pour présenter le dernier package de MetaDEVS : le package des règles (Figure IV-27). La classe mère (abstraite) de ce dernier contient une référence vers un test de StateVar (car ce test est commun à toutes les fonctions).

Prenons comme exemple de règle complète un feu tricolore qui fonctionnerait en mode « manuel » : le message reçu sur un port d'entrée est une StringValue indiquant la couleur à laquelle le feu doit passer (on suppose que la couleur actuelle et la couleur passée en entrée sont différentes). La règle est donc composée de trois éléments : une condition sur l'état, une condition sur un port et une action sur l'état.

La Figure IV-28 est une capture d'écran EMF qui montre les propriétés des trois éléments de cette règle.

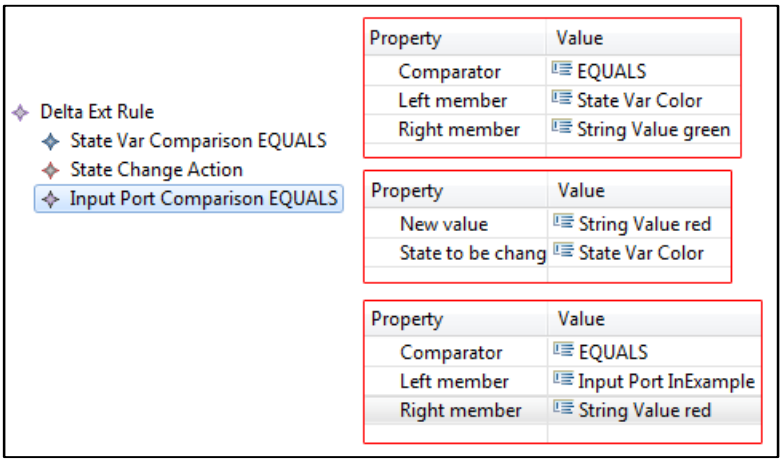


Figure IV-28 : Exemple de *DeltaExtRule*

4.6. **Le package DEVSModel**

Nous sommes maintenant pratiquement à même d'affiner encore plus les figures Figure IV-3 et Figure IV-5 et de leur faire prendre leur forme définitive sous forme de package.

Nous avons juste besoin de poursuivre notre raisonnement sur la gestion des différentes expressions DEVS. Comme nous l'avons dit, pour pouvoir bénéficier d'un contrôle de types dans nos modèles, il faut instancier chaque fille de la classe type voulue sous forme de singleton (une seule instanciation par classe).

Par exemple, pour un modèle gérant du texte et des entiers, il est nécessaire d'instancier la méta-classe StringType et IntegerType. Les fonctions susceptibles de manipuler des entiers ne se trouvant que dans les modèles atomiques, il suffit de placer dans la classe AtomicDEVS une référence vers la classe abstraite Type, en précisant dans les cardinalités qu'au moins un type doit être instancié.

De même, tout modèle DEVS atomique doit être capable de gérer des expressions DEVS, notamment des valeurs littérales utilisées pour les comparaisons simples et les affectations : l'ajout d'une référence vers la classe abstraite DEVSXpression est par conséquent nécessaire.

Un état doit être créé au sein d'un modèle atomique, car il n'a pas vocation à être partagé par les modèles de niveau supérieur. En d'autres termes, de par la séparation explicite faite par le formalisme DEVS entre structure et comportement, un modèle couplé n'a pas à avoir connaissance des états des modèles atomiques qu'il contient.

Le package DEVSModel est l'élément central du méta-modèle MetaDEVS, il est présenté ci-dessous sur la Figure IV-29.

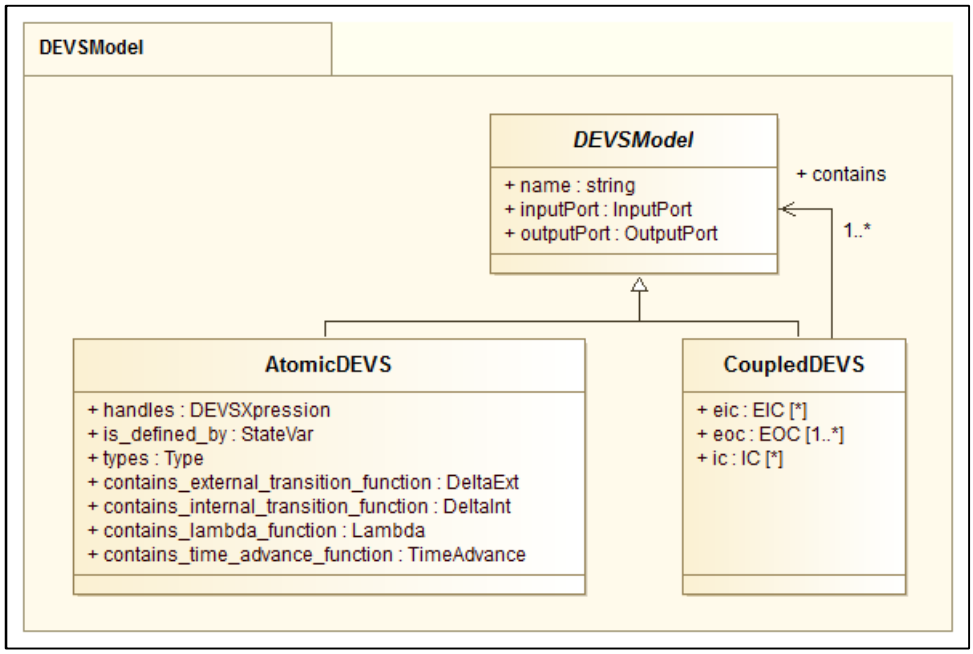


Figure IV-29 : Package des modèles DEVS

4.7. Enrichissement du méta-modèle avec des contraintes OCL

Même si MetaDEVS est conforme au formalisme DEVS, nous n'avons pas pour l'instant spécifié des contraintes autrement qu'en utilisant des classes, des héritages et des cardinalités.

Or, nous avons besoin, et nous l'avons mentionné tout au long de la présentation de MetaDEVS, d'avoir un contrôle précis sur les attributs et les relations entre classes, un contrôle qui aille bien plus loin au niveau de la nature des relations : cela passe obligatoirement par l'expression d'une sémantique statique, que nous nous proposons à présent de définir. Plus précis sera le méta-modèle, moins les modèles qui en seront des instances comporteront d'erreurs, et donc leur transformation vers des PSM (et donc du code) en sera grandement facilitée.

OCL (*Object Constraint Language*) est, dans notre cas, particulièrement adapté à la spécification de contraintes indépendamment des plateformes, c'est par conséquent ce dernier que nous allons utiliser. Son autre avantage est qu'il est intégré à Eclipse EMF et peut s'utiliser directement dans un modèle ou un méta-modèle.

Nous exprimerons chaque contrainte en langage naturel, en précisant son contexte d'application, puis nous donnerons son équivalent en OCL tel que nous l'avons spécifié dans le méta-modèle à l'aide du langage OCLinEcore. La version l'OCL que nous avons utilisée est OCL 3.1.0. Cette approche a été validée dans [Garredu et al. 2012b].

4.7.1. Contraintes sur StateVar

Nous avions évoqué une redondance qui apparaissait lorsqu'on donnait une référence vers une LitteralBasicValue, typée, à une instance de StateVar, typée elle aussi. Bien que nécessaire du fait de l'architecture que nous avons choisie, cette redondance peut conduire à

des absurdités, par exemple une StateVar de type String contenant une référence vers une IntValue

La contrainte la plus simple que nous puissions mettre en place est donc celle qui contrôle que ces deux types sont bien identiques : cette contrainte exprime un invariant de classe. Dans le contexte de la classe StateVar, cette contrainte s'énonce en OCL :

```
invariant StateVarTypeConstraint:
self.type = self.initial_value.type;
```

Nous devons aussi nous assurer qu'une StateVar puisse être identifiée : elle doit pour cela avoir un attribut DEVSid différent de la chaîne vide :

```
invariant idNotEmpty: self.DEVSid.size()>0;
```

Enfin, il ne faut pas que deux StateVar puissent avoir le même *DEVSid*. Pour ce faire, il suffit de collecter les identifiants de toutes les StateVar manipulées par le modèle, de supprimer les doublons grâce à la foction OCL asSet() puis de compter les éléments de cette liste. Si leur nombre est différent au nombre d'identifiants collectés sans supression des doublons, alors au moins deux StateVar portent le même *DEVSid*. Les deux comptage des deux différentes collectes doivent donc renvoyerle même nombre d'éléments, ce que spécifie la contrainte suivante :

```
invariant noIdenticalStateVars:
self.is_defined_by->collect(DEVSid)->asSet()->size()
= self.is_defined_by->collect(DEVSid)->size();
```

4.7.2. Contraintes sur LitteralBasicValue

En tant que fille de DEVSXpression, une valeur littérale peut se voir attribuer un type différent de la valeur qu'elle véhicule. Par exemple, il est théoriquement possible de créer une StringValue de type IntegerType, ce qui entraîne que cette valeur littérale sera considérée dans MetaDEVS comme un entier : ceci est à éviter absolument et il est par conséquent nécessaire que chaque valeur littérale nouvellement créée corresponde à un et un seul type. De plus, une StringValue ne doit pas être vide :

- IntValue doit être de type IntegerType

```
invariant intIsInt:
self.type.oclType().name = 'IntegerType';
```

- CharValue doit être de type CharType

```
invariant charIsChar:
self.is.type.oclType().name = 'CharType';
```

- BooleanValue doit être de type BooleanType

```
invariant boolIsBool:
self.type.oclType().name = 'BooleanType';
```

- StringValue doit être de type StringType et non vide

```
invariant stringIsString:
self.type.oclType().name = 'StringType';
invariant nameNotEmpty : self.str_val.size()>0
```

4.7.3. **Contraintes sur Coupling**

Dans l'état actuel des choses, une instance de EIC lie deux ports d'entrée, une instance de EOC lie deux ports de sortie, une instance de IC lie un port d'entrée à un port de sortie. Or, ceci est très insuffisant et ne correspond que vaguement à la définition de base d'une fonction de couplage. Dans le but d'être le plus précis possible et de respecter les conditions définies en 4.4.2, nous avons donc besoin d'ajouter des contraintes sur les couplages.

Nous présentons ici trois séries de contraintes. Chaque série, comportant elle-même trois contraintes, est définie dans le cadre d'une des trois fonctions de couplage. Ces contraintes sont très importantes car elles empêchent de coupler n'importe quelle instance de port avec n'importe quelle autre. Notons que la dernière contrainte de chaque série a pour but de vérifier que deux ports liés sont bien du même type.

- Il faut que EIC contienne d'une part une référence (EIC_coupled_in) vers un port d'entrée du modèle couplé qui la contient et d'autre part une référence (EIC_in) vers le port d'entrée d'un sous-modèle $Md|d \in D$

```
invariant EICcurrentModelInputPort :
self.oclContainer()=self.EIC_coupled_in.oclContainer();

invariant EICsubmodelInputPort :
self.oclContainer()=self.EIC_in.oclContainer().oclContainer();

invariant EICtypes:
self.EIC_coupled_in.type = self.EIC_in.type;
```

- Il en va de même pour EOC, dont le EIC_coupled_out doit désigner un port de sortie du modèle couplé courant, tandis que EOC_out doit désigner le port de sortie de l'un de ses sous-modèles

```
invariant EOCcurrentModelOutputPort :
self.oclContainer()=self.EOC_coupled_out.oclContainer();

invariant EOCsubmodelOutputPort :
self.oclContainer()=self.EOC_out.oclContainer().oclContainer();

invariant EOCtypes:
self.EOC_coupled_out.type=self.EOC_out.type;
```

- IC doit lier deux ports (un d'entrée, un de sortie) appartenant tous deux à des sous-modèles du modèle courant, il faut là aussi le vérifier séparément pour chacune des deux références

```
invariant ICsubmodelInputPort :
self.oclContainer()=self.IC_in.oclContainer().oclContainer();

invariant ICsubmodelOutputPort :
self.oclContainer()=self.IC_out.oclContainer().oclContainer();

invariant ICtypes:
self.IC_out.is_typed=self.IC_out.is_typed;
```

4.7.4. Contraintes sur Condition et Action

Le but ici est de s'assurer que, dans une condition, l'élément testé, en l'occurrence une instance de StateVar, ou instance d'Input Port, soit du même type que l'élément avec lequel il est comparé. Ceci se fait très simplement :

- dans la définition de la classe StateVarComparison :

```
invariant svcTypeConstraint:
self.left_member.type=self.right_member.type;
```

- dans la définition de la classe InputPortComparison :

```
invariant ipcTypeConstraint:
 self.left_member.type=self.right_member.type;
```

De même, il faut s'assurer que dans une Action il y ait le même type de concordance s'il y a un changement d'état ou l'envoi d'une valeur sur un port de sortie, et dans le cas d'une OutputAction il faut également vérifier que le port de sortie passé en paramètre appartienne bien au modèle atomique :

- pour une StateChangeAction :

```
invariant stateChangeTypeConstraint:
 self.affected_state.type=self.new_value.type;
```

- pour une OutputAction :

```
invariant outputActionTypeConstraint:
self.port.type=self.message.type;

invariant portBelongsToCurrentAtomic :
self.port.oclContainer()=self.oclContainer();
```

Conclusion du chapitre

Dans ce chapitre, qui constitue le cœur de nos travaux, nous avons tout d'abord justifié de la nécessité d'un méta-modèle de DEVS permettant de spécifier des modèles DEVS (y compris les fonctions comportementales qui s'y rappportent) indépendamment de toute plateforme, puis nous avons décrit notre démarche de création de méta-modèle. Nous nous sommes servis pour cela de diagrammes de classe UML (équivalents au méta-formalisme MOF) et nous avons implémenté le méta-modèle au moyen d'EMF et du méta-formalisme Ecore (proche d'EMOF). Ceci nous a permis de définir une syntaxe abstraite pour DEVS. Nous laissons l'utilisateur libre d'implémenter la syntaxe concrète de son choix (i.e. créer un formalise graphique associé à MetaDEVS).

Nous avons ensuite doté ce méta-modèle de contraintes précises au niveau de ses attributs et de ses relations, chose que ne peuvent pas décrire les seuls diagrammes UML. Nous avons employé pour cela un langage de modélisation simple, standardisé par l'OMG, sans effets de bord, ayant pour but de spécifier des contraintes : OCL. Ceci nous a permis de doter le méta-modèle d'une sémantique statique (sous forme notamment d'invariants de classes). L'annexe 4 contient l'intégralité du code de MetaDEVS, y compris les contraintes qui lui sont associées.

Ce méta-modèle se veut aussi modulaire que possible, et il est facile d'y ajouter des extensions, grâce à des héritages entre les méta-classes. Nous sommes donc maintenant en mesure de créer des modèles DEVS de type PIM (indépendants de la plateforme). Ces modèles ont donc un caractère général et universel. Cependant, ils ne sont pas utilisables tels quels. Nous devons pour cela les transformer en code orienté-objet, spécifique à telle ou telle plateforme particulière. Nous pouvons aussi, à partir de modèles non-DEVS, mais proches de par leur caractéristiques, montrer comment ces derniers peuvent être transformés en des modèles DEVS conformes à MetaDEVS. C'est le but des deux prochains chapitres.

Chapitre V. TRANSFORMATION DE MODELES M2M

Le chapitre précédent était consacré à l'utilisation de la méta-modélisation afin de proposer un méta-modèle de DEVS indépendant de toute plateforme : MetaDevs. Nous avons également évoqué la place de ce dernier en tant que pivot central de notre approche.

C'est à présent ce rôle de pivot central que nous allons aborder en détail. Dans ce chapitre, nous nous intéresserons tout d'abord aux transformations « Model To Model », pour aborder ensuite les transformations « Model To Text » dans le chapitre suivant. Nous montrerons que ces deux types de transformations apparaissent comme complémentaires dans le cadre de notre étude, dans la mesure où l'une se concentre sur l'interopérabilité entre certains formalismes et DEVS (« interopérabilité externe »), tandis que l'autre a trait à la portabilité et la productivité des modèles.

Une transformation « M2M », comme nous l'avons vu dans le chapitre 2, décrit comment transformer un modèle en un autre modèle exprimé dans un formalisme différent de celui du modèle de départ. Donc, l'exécution d'une transformation « M2M » ayant DEVS comme objectif n'est rien d'autre que l'application, sur un modèle source, de règles de transformation définies entre un méta-modèle source et le méta-modèle MetaDEVS : une telle transformation produira un modèle DEVS.

Mais pourquoi vouloir transformer des formalismes vers DEVS ? Nous avons montré dans le chapitre précédent qu'un certain nombre de travaux avaient pour but de transformer des formalismes couramment utilisés en M&S vers DEVS. Nous avons également montré, que l'interopérabilité entre formalismes, notamment l'interopérabilité externe de DEVS, était un souci majeur et un enjeu important pour le monde de la M&S. Disposant à présent d'un méta-modèle de DEVS permettant de spécifier des modèles indépendamment de toute plateforme, MetaDEVS, il est donc naturel d'accroître le rayon d'action de ce méta-modèle en créant des passerelles entre d'autres formalismes et ce dernier.

S'il est établi que le formalisme DEVS est la destination de cette transformation « M2M », qu'en est-il de la nature des formalismes source ? D'après les formalismes que nous avons évoqués dans ce travail, en particulier dans les deux premiers chapitres, nous pouvons envisager trois *scenarii* de transformation. En effet, le formalisme en entrée peut :

- soit n'avoir aucun point commun avec DEVS,
- soit partager avec DEVS les concepts de *modèle*, d'*état* et de *transition*,
- soit constituer une simple restriction des concepts de DEVS.

Chacune de ces possibilités correspond à une famille de formalismes. Nous les caractériserons plus précisément dans la suite de ce chapitre.

Comment exprimer avec des concepts DEVS des modèles appartenant à d'autres formalismes ? La solution que nous proposons est de définir des règles de transformation selon des concepts et outils issus de l'IDM et en particulier de l'approche MDA.

Dans un esprit IDM, une transformation est aussi un modèle : elle est donc forcément une instance d'un méta-modèle, et elle est aussi potentiellement réutilisable. Nous nous efforçons donc de montrer qu'il est possible d'identifier des sortes de patrons de transformation spécifiques à certains formalismes, qui, combinés à une conception modulaire des transformations, permettent à ces dernières d'être partiellement réutilisées.

Dans ce chapitre, tout comme nous l'avons fait dans le précédent, nous avons recours à la méta-modélisation, afin de décrire les formalismes que nous allons transformer vers MetaDEVS. Ces formalismes sont BasicDEVS, un formalisme simple proche de DEVS, et les automates à états finis. Nous avons, pour créer les méta-modèles de ces formalismes, suivi une démarche analogue à celle qui a abouti à la création de MetaDEVS. Ces formalismes étant plus simples que DEVS, nous ne montrons ici que le résultat du processus de méta-modélisation, en présentant pour chacun, succinctement, son méta-modèle et les contraintes associées.

Concernant l'implémentation des transformations, notre choix s'est porté sur l'environnement Eclipse, pour plusieurs raisons :

- Assurer une continuité dans la démarche : c'est également dans le cadre d'Eclipse EMF que nous avions défini le méta-modèle MetaDEVS ;
- Il existe à ce jour plusieurs outils, utilisables sous forme de plugins Eclipse permettant d'effectuer des transformations de modèles, aussi bien de type M2M que M2T, ces plugins respectant par ailleurs les standards préconisés par l'OMG.

Plus précisément, nous avons choisi d'utiliser pour effectuer nos transformations M2M le langage hybride (déclaratif et impératif) ATLAS Transformation Language (ou ATL) car il est basé sur le standard QVT de l'OMG, est disponible sous licence EPL, et surtout il permet une grande puissance d'expression notamment au niveau des fonctions et des arguments de ces dernières.

Les règles implémentées en ATL diffèrent assez peu des règles exprimées dans le pseudo-langage que nous proposons dans ce chapitre.

Ce chapitre débute par une vue d'ensemble. Nous définissons de manière générale ce qu'est une transformation M2M vers DEVS en termes d'IDM/MDA et caractérisons au mieux les trois familles de formalismes évoqués précédemment ainsi que les transformations dans lesquelles elles peuvent être impliquées. Nous discutons de l'intérêt de transformer chacune d'entre elles vers DEVS. C'est dans cette discussion que nous introduirons le but central du chapitre : proposer, pour les langages issus de la seconde famille, une méthode générale de transformation « Model-To-Model » vers MetaDEVS, pivot de notre approche.

Nous proposons un schéma de transformation générique pouvant s'appliquer à certains formalismes retrouvés en entrée, que nous caractérisons. Ce schéma est présenté de manière synthétique sous forme de tableau, puis sous forme détaillée en pseudo-code.

Nous prenons enfin deux exemples de formalismes différents, futurs candidats à la transformation, appartenant à la famille des formalismes à états-transitions et nous présentons leur méta-modèle (BasicDEVS et FSM). Pour chacun d'entre eux, nous définissions une transformation vers MetaDEVS, toujours avec des règles génériques. Enfin, nous donnons des

exemples de transformation complets, montrant comment un modèle source FSM ou BasicDEVS est transformé automatiquement en un modèle MetaDEVS.

5.1. Vers le formalisme DEVS : approche générale

5.1.1. Caractéristiques d'une transformation

La définition d'une transformation M2M vers DEVS, dans un esprit IDM/MDA, exige toujours que l'on dispose au préalable :

- du méta-modèle source
- du méta-modèle cible : dans le cadre de nos travaux, ce sera toujours le méta-modèle de DEVS

Les deux méta-modèles devront être typés par le même méta-méta-modèle (ou méta-formalisme) : ici, il s'agit de MOF ou d'un de ses équivalents.

En outre, pour qu'une transformation soit exécutable et donc produise un résultat, il est naturellement nécessaire de disposer d'un modèle source, instance du méta-modèle source.

Par contre, et ceci est très important, rappelons que notre approche pour une meilleure interopérabilité se situe exclusivement au niveau des modèles. Cela ne signifie pas que les modèles ne doivent pas être simulés, bien au contraire, mais que les mécanismes favorisant leur interopérabilité sont situés au niveau des modèles et non au niveau du code.

Nous ne nous occupons pas de la productivité des modèles source (il n'est donc pas important que ces derniers appartiennent à des formalismes qui ont été implémentés et qui possèdent des algorithmes de simulation) car c'est en tant que modèles DEVS qu'on va les simuler. Le chapitre suivant montre de quelle manière.

Une transformation vers DEVS met en œuvre les mêmes mécanismes que toute transformation de modèle effectuée dans un cadre IDM. Chronologiquement, on peut la décomposer en deux étapes, située chacune à un niveau d'abstraction différent :

- **Niveau M2 :** Identification des correspondances entre le méta-modèle source et le méta-modèle de DEVS. Cela suppose la mise en place d'une fonction de transformation qui puisse être appliquée sur n'importe quelle instance du méta-modèle source (i.e. un modèle) afin de produire un modèle DEVS
- **Niveau M1 :** Mise en œuvre de la transformation. Elle se traduit par l'utilisation d'un moteur de transformation qui va exécuter la transformation et produire un modèle DEVS à partir du modèle source.

5.1.2. Principe des transformations vers DEVS

La figure suivante montre les entités présentes avant et après l'exécution d'une transformation type d'un modèle M*x* se conformant à un méta-modèle MM*x* vers DEVS (Figure V-1). Les règles de transformation se situent au même niveau d'abstraction que les méta-modèles. Sur cette figure, les éléments (modèles, méta-modèles, méta-formalisme, règles de transformation) présents avant l'exécution des transformations sont représentés en nuances de gris. Modèles et méta-modèles en gris clair, règles en gris foncé. Les modèles sont

des ellipses, les méta-modèles des rectangles arrondis. Les niveaux « méta » (niveau des modèles, des méta-modèles, et du méta-formalisme qui leur est commun) sont représentés par la flèche verticale pleine à gauche. Le typage des modèles par leurs méta-modèles du niveau supérieur est représenté par des flèches en points tillés.

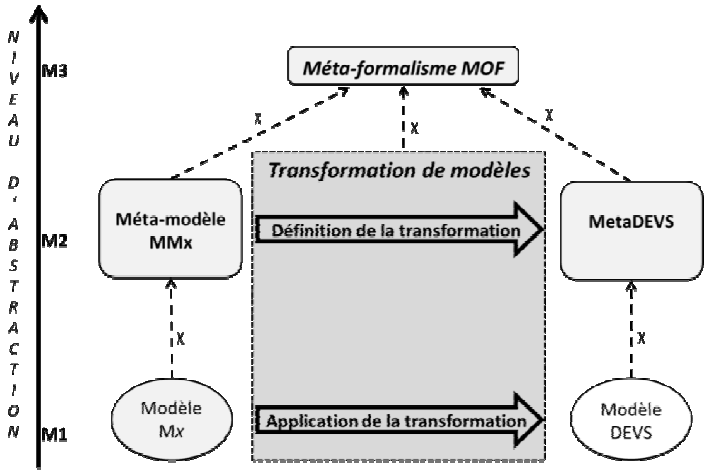


Figure V-1 : Une transformation type vers DEVS suivant une approche IDM/MDA

Les transformations (vers DEVS), en tant que modèles, se conforment elles-aussi à MOF. Une transformation se situe à deux niveaux :

- elle est définie au niveau des méta-modèles puisqu'elle constitue la mise en correspondance d'un ou plusieurs éléments du méta-modèle source avec un ou plusieurs éléments du méta-modèle cible,
- elle exécutée au niveau des modèles.

Enfin, les modèles, qui sont tous des modèles indépendants des plateformes (PIM) sont classés en deux catégories : les modèles présents avant la transformation sont coloriés en gris clair, tout comme les méta-modèles auxquels ils se conforment, tandis que les modèles issus de l'exécution d'une transformation sont blancs. Nous faisons en outre apparaître sur cette figure la relation de conformation abordée dans le second chapitre (2.1.1.f)).

5.1.3. Formalismes sources et classification des transformations

Avant de chercher à définir un processus de transformation, il est selon nous nécessaire de chercher à préciser, caractériser, la nature de ces transformations de modèle à modèle, en fonction de la famille à laquelle elles appartiennent. Cette réflexion préalable nous permettra de nous situer par rapport aux notions abordées dans le chapitre II dédié à l'IDM et de garder une vision claire de notre approche.

a) *Classification en familles des formalismes candidats à une transformation vers DEVS*

Nous proposons de classer en trois familles les formalismes utilisés en M&S que nous avons évoqués dans l'introduction. Ces familles sont :

- F_3 : tous les formalismes de nature différente de DEVS au niveau des concepts (par exemple les SMA…)

- F_2 : famille des formalismes pouvant être qualifiés de type « états-transitions ». Ces derniers peuvent être de bas niveau (typiquement, des MOC, tels que les automates à états finis ou les réseaux de Petri) ou de plus haut niveau mais manipulant les mêmes concepts que les MOC. Un exemple de ces derniers est celui des diagrammes de machines d'états en UML.
- F_1 : famille des formalismes constituant une restriction de DEVS, voire DEVS lui-même (ce qui serait possible dans le cas, par exemple, d'une transformation dite « de mise à jour » ou « sur place »)

La figure suivante (Figure V-2) montre un tableau qui regroupe les trois transformations possibles et les caractérise en fonction de leur type.

Famille source	Niveau	Cible	Niveau	Type de transformation
F_1 : DEVS ou restriction de DEVS	MOC	**DEVS**	**MOC**	Horizontale/Endogène
F_2 : Formalisme à états-transitions	MOC ou assimilé			Horizontale/Exogène
F_3 : Autre formalisme	?			Horizontale/Exogène

Figure V-2 : Tableau résumant les principales transformations possibles vers DEVS

De ce tableau, nous pouvons extraire deux types de transformations différentes : l'une, endogène, se fait entre un méta-modèle et un autre qui l'inclut (famille source F1), tandis que l'autre, exogène, se fait entre deux méta-modèles distincts (familles sources F2 et F3).

b) ***Transformations à partir de DEVS ou d'une de ses restrictions***

La transformation la plus simple consiste à transformer des modèles DEVS créés suivant les spécifications de MetaDEVS en d'autres modèles MetaDEVS équivalents. Le but d'une telle transformation est d'améliorer la lisibilité des modèles en les réorganisant.

Une transformation d'un modèle DEVS vers un modèle DEVS suppose que les méta-modèles source et cible soient confondus : on parle alors de transformation « sur place ». Ce type de transformation est par nature toujours endogène (à lier aux notions de *refactoring*, d'optimisation). Un exemple de transformation sur place, dans notre cas, serait de supprimer les doublons des instances de *LitteralBasicValue* d'un modèle DEVS. Nous reviendrons par la suite sur les améliorations pouvant être apportées aux transformations notamment en faisant appel à une transformation « sur place ».

Concernant les formalismes qui sont des restrictions de DEVS, même si techniquement ils possèdent leur propre méta-modèle, celui-ci est un dérivé de celui de DEVS caractérisé par un sous-ensemble des concepts DEVS (exemple : un méta-modèle permettant uniquement la spécification de modèles de type AtomicDEVS). Une transformation d'un formalisme décrivant une restriction de DEVS vers le formalisme DEVS peut être considérée comme endogène, même si, dans la pratique, on peut se trouver en présence de méta-modèles source et cible distincts.

La Figure V-3 illustre une telle transformation. « DEVS-Rest » est un méta-modèle dérivé du méta-modèle de DEVS, constituant une restriction de celui-ci. Nous l'avons donc représenté de sorte qu'il soit inclus dans le méta-modèle de DEVS. Ici, l'exécution des règles de transformation sur le modèle DEVS-Rest a créé un modèle DEVS *m'*.

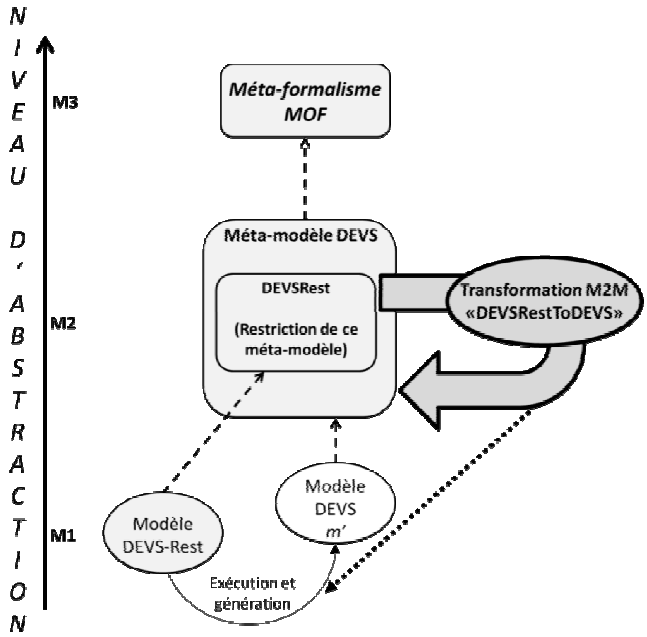


Figure V-3 : Transformation d'une restriction de DEVS vers DEVS

c) ***Transformation à partir d'un formalisme à états-transitions***

Comme nous l'avons évoqué dans la partie consacrée à l'état de l'art des transformations vers DEVS, beaucoup de formalismes basés sur les états et les transitions peuvent subir une transformation vers le formalisme DEVS.

Un début de justification peut-être avancé de manière intuitive : pour peu qu'un formalisme permette de décrire des états et des liens simples (déterministes ou pas) entre ceux-ci, il possède par conséquent des concepts communs avec le formalisme DEVS.

Dans [Vangheluwe 2000], il a été montré que les formalismes basés sur les états-transitions de bas niveau (MOC), mais aussi les formalismes de plus haut niveau tels que les digrammes de machines d'états, ou d'autres formalismes basés sur les évènements discrets, peuvent être transformés vers DEVS.

Pratiquement tous les langages basés sur les états et les transitions sont des MOC, mais ils peuvent être de plus haut niveau d'abstraction, comme les diagrammes de machines d'états UML. Comme ces derniers possèdent une grande partie de concepts communs avec les MOC, nous les considèrerons, pour simplifier, comme étant d'un niveau d'abstraction plus ou moins équivalent à celui de DEVS. Chaque transformation sera donc de type horizontal (pas de changement de niveau d'abstraction).

On se trouve ici en présence de transformations nécessairement exogènes (i.e. méta-modèle source ≠ méta-modèle cible).

d) ***Transformation à partir d'un autre formalisme non états-transitions (famille F_3)***

La famille F_3 ne se définit pas par les formalismes qu'elle contient mais plutôt par ceux qu'elle ne contient pas : F_3 regroupe en effet tous les formalismes n'appartenant pas F_2 ou F_1. Il est par conséquent impossible de préjuger ou non du degré de complexité de leur transformation vers DEVS, car ils ne partagent pas de concepts avec ce formalisme, et ne partagent pas forcément non plus de concepts entre eux. On ne peut donc pas être sûr du fait

qu'une telle transformation soit possible. La seule solution consiste à travailler au cas par cas et par abstraction, en essayant de faire correspondre des éléments du formalisme source avec tel ou tel concept de DEVS.

Des transformations depuis un formalisme appartenant à F_3 vers DEVS peuvent donc exister, mais il n'est pas possible, à notre sens, de définir démarche générique pouvant guider leur mise en place, du fait même de leurs natures très différentes.

5.1.4. Recherche d'une approche générique de transformation

a) *Restriction à certains formalismes source*

La nécessité d'une restriction à certains formalismes apparaît clairement lorsque l'on veut suivre une approche qui se veut la plus générique possible. Cette restriction permet de dégager des points communs, invariants, entre différents formalismes source.

Tout modèle candidat à la transformation vers DEVS doit être exprimé dans un formalisme qui possède au minimum les caractéristiques suivantes :

- permettre de définir des états et donc de les différencier (et autoriser la déclaration d'un état initial) ;
- décrire le passage d'un état *s* à un état *s'* au moyen d'une fonction de transition, basée sur le temps ou sur des évènements ;
- manipuler des types de base communs à tous les langages de programmation (entiers, booléens, caractères, chaînes de caractères, réels…) et toutes les plateformes (nombres entiers ou réels, chaînes de caractères, booléens).

Les formalismes des familles F_2 et F_1 remplissent tous ces critères. Optionnellement, un modèle exprimé dans ce formalisme pourra également :

- évoluer de manière autonome ou recevoir des stimuli extérieurs, ce qui suppose la possibilité de définir des ports d'entrée ;
- envoyer des informations à l'extérieur ou être lié à d'autres modèles, ce qui suppose la possibilité de définir des ports de sortie ;
- contenir lui-même des modèles (i.e. hiérarchie).

Si un formalisme est en adéquation avec les critères ci-dessus, il est donc transformable en modèle DEVS, et nous allons montrer comment. La seule restriction est imposée par la nature de notre méta-modèle MetaDEVS qui possède une expressivité restreinte au niveau des fonctions (conditions et actions). Bien entendu, un formalisme constituant une restriction de DEVS représente un sous-ensemble de ce dernier : il est donc transformable vers DEVS.

b) *Méthodologie*

En résumé, nous avons tout d'abord sélectionné un panel de formalismes candidats à une transformation vers DEVS : ce panel, forcément restreint, concerne les formalismes appartenant à la famille des formalismes à états-transitions.

Notre objectif est maintenant de définir, sous forme de tableaux, des correspondances génériques des concepts que partagent forcément, de par leur nature, DEVS et ces formalismes candidats. Ces correspondances génériques seront affinées ensuite sous forme de règles génériques, implémentées dans un pseudo-langage inspiré du standard de l'OMG QVT (*Query/View/Transformation*).

Enfin, ces règles de transformation seront implémentées au moyen du langage de transformation ATL (*ATL*as *T*ransformation *L*anguage). Nous verrons alors que certaines règles peuvent être réutilisées telles quelles.

5.2. Correspondances génériques de concepts

Nous présentons ici une approche générique pour une transformation d'un formalisme remplissant les critères vus en 5.1.4 vers DEVS : ceux constituant soit une restriction de DEVS, soit étant basés sur les états et les transitions. Cette approche s'appuie sur les contraintes inhérentes au méta-modèle MetaDEVS.

Elle se compose d'un tableau synthétique décrivant la transformation, assorti de courtes remarques, suivi de plusieurs notes concernant des points essentiels de la transformation. Cette approche peut être appréhendée comme une marche à suivre, un fil conducteur, pour décrire de manière efficace des règles de transformation. Nous l'utiliserons d'ailleurs dans la suite de ce document pour définir chacune des trois transformations qui nous serviront d'exemples concrets.

5.2.1. Description des correspondances de concepts

Départ	*Arrivée*	*Interprétation*
1	1	À l'unique élément forcément présent au départ correspond un unique élement d'arrivée
0..* |1..*| 0..1	0..* |1..*| 0..1	Créer dans l'ensemble cible autant d'éléments qu'il en existe dans l'ensemble source
0..1	1	Que l'élément source existe ou pas, créer quand même un (unique) élément à l'arrivée (ex. nom d'un modèle, ID d'un port…)
0..*	1..*	Si pas d'élément source, créer au moins un élément de destination Si n ($n > 0$) éléments source, créer n éléments destination
	1	Élément de conception spécifique au méta-modèle de DEVS, créé systématiquement à chaque instanciation de modèle DEVS

Tableau V-1 : Les différentes cardinalités possibles

Pour décrire au mieux les correspondances entre les concepts des méta-modèles source et cible, nous utilisons différentes cardinalités au sein d'un tableau. Elles sont issues d'une part de nos hypothèses quant à la nature du formalisme source, et d'autre part des contraintes de cardinalités fixées dans le chapitre IV consacré à MetaDEVS. Nous les résumons dans le Tableau V-1).

5.2.2. Correspondances de concepts

L'objectif de cette sous-section est d'établir des correspondances entre les formalismes candidats à une transformations vers DEVS d'une part, et le formalisme DEVS

d'autre part. Nous donnons tout d'abord une vue schématique, puis une vue détaillée, de ces correspondances.

a) ***Vue schématique***

Au vu des caractéristiques des formalismes que nous nous proposons de transformer vers DEVS, il est possible de définir un schéma global de transformation montrant quels éléments du formalisme source vont être transformés en éléments du formalisme de destination (Tableau V-2).

SOURCE	*DESTINATION*
Modèle	AtomicDEVS ou CoupledDEVS
État	LitteralBasicValues énumérées dans une/des StateVar
Port	InputPort ou OutputPort
Transition	DeltaInt ou DeltaExt

Tableau V-2 : Vue schématique d'une transformation vers DEVS

b) ***Vue détaillée***

Nous présentons à présent un tableau subdivisé en trois sous-tableaux montrant une vue détaillée de la transformation d'un formalisme correspondant à nos critères (BasicDEVS) vers le formalisme DEVS.

Nous distinguons trois grands groupes de correspondances de concepts :

- Les correspondances communes aux modèles atomiques et couplés et donc se rapportant à la méta-classe abstraite *DEVSModel* (Tableau V-3) ;
- Les correspondances relatives aux seuls modèles couplés : méta-classe *CoupledDEVS* (Tableau V-4) ;
- Les correspondances relatives aux concepts propres aux modèles atomiques : méta-classe *AtomicDEVS*. Il s'agit du groupe comportant le plus de correspondances en raison du nombre important de concepts qu'il contient. (tableau 3).

Chacun de ces groupes correspond à un sous-tableau distinct.

Pour chaque élément ou groupes d'éléments « principaux » du modèle source, correspondent un ou plusieurs éléments dans le modèle DEVS. Chaque élément de destination (ou élément cible) peut contenir deux types d'information différents :

- soit des propriétés, qui doivent être renseignées,
- soit des références vers des objets créés (ou pas) contenant eux-mêmes des propriétés et/ou des références.

Pour chaque élément cible, on décrit une action. Si cet élément correspond à un concept de DEVS (donc en général à une classe), l'action correspondante est l'instanciation (I), par analogie avec le vocabulaire des langages orientés-objet. Si l'élément cible correspond plutôt à une propriété d'un concept plutôt qu'à un concept, elle doit être renseignée (R). Bien

entendu, pour chaque référence à une classe ces actions sont consécutives et complémentaires : on instancie d'abord une classe, on en renseigne les champs ensuite. Par exemple, pour la partie de la règle qui établit les correspondances entre modèles, on instancie d'abord un modèle DEVS et on lui donne ensuite un nom. L'action (R) ou (I) figure dans la colonne « A » (Action) du tableau.

SOURCE		DESTINATION			REMARQUES
Élément du méta-modèle	**Car.**	**Méta-modèle DEVS**	**A**	**Car.**	
Modèle	**1..***	***DEVSModel***	**I**	**1..***	**Si le modèle d'entrée contient une hiérarchie *CoupledDEVS* (conteneur), sinon *AtomicDEVS***
Nom du modèle	0..1	*this*.name	R	1	Pour chaque modèle, renseigner nom
		StringType	I	1	Instancier chaque type de base une seule fois
		IntegerType	I	1	
		CharType	I	1	
		BooleanType	I	1	
Port d'entrée	**0..***	**InputPort**	**I**	**0..***	**Facultatif**
Nom du port	0..1	this.portID	R	1	Si le port d'entrée est nommé, recopier nom, sinon donner nom par défaut incrémentable
		this.isAlwaysTyped	R	1	Comparer avec le type de valeurs manipulées
Port de sortie	**0..***	**OutputPort**	**I**	**1..***	**Obligatoire**
Nom du port	0..1	this.portID	R	1	Si le port de sortie est nommé, recopier nom, sinon donner nom par défaut incrémentable
		this.isAlwaysTyped	R	1	Comparer avec le type de valeurs manipulées

Tableau V-3 : Correspondances patagées par les modèles atomiques et couplés

SOURCE		DESTINATION			REMARQUES
Élément du méta-modèle	**Car.**	**Méta-modèle DEVS**	**A**	**Car.**	
Modèle conteneur	**1**	**DEVSCoupled**	**R**	**1**	**Correspond à un modèle couplé**
modèles contenus	1..*	this.contains	R	1..*	Liste de sous-modèles (A ou C)
liens entre modèles	1..*	*Coupling*	I+R	1..*	Instancier selon le type de lien EIC, EOC, ou IC et renseigner avec les ports créés ci-dessus

Tableau V-4 : Correspondances se rapportant à CoupledDEVS

Dans tous les cas, chaque concept devant être instancié, et comportant lui-même des références et/ou des propriétés, est mis en caractères gras, les références/propriétés qu'il contient figurent en-dessous, le tout est encadré par un trait épais. Il se peut qu'un élément dans le modèle DEVS cible doive être créé, et ce alors qu'il n'existe aucun « antécédent » dans le modèle source.

Dans ce cas, aucun élément n'apparait dans la partie « source ». Les cardinalités sont indiquées dans la colonne « Car. ».

Nous nous sommes efforcés de décrire ce processus générique d'un point de vue pratique. Ne perdons pas de vue que MetaDEVS prévoit pour certaines propriétés des valeurs

par défaut, il n'est donc pas nécessaire de les renseigner, c'est pourquoi nous n'en faisons pas mention. Enfin, nous supposons que le comparateur par défaut pour les tests est le comparateur d'égalité.

SOURCE		DESTINATION			REMARQUES
Élément du méta-modèle	**Card.**	**Méta-modèle DEVS**	**A**	**Car.**	
Enchaînement états	**1..***	**DEVSAtomic**	**I**	**1..***	**Créer un modèle atomique**
		DeltaInt	I	1	Instancier chaque fonction une seule fois
		DeltaExt	I	1	
		Lambda	I	1	
		TimeAdvance	I	1	
Valeur manipulée	**1..***	**LitteralBasicValue**	**I**	**1..***	**Répertorier toutes les valeurs manipulées**
		this.isAlwaysTyped	R	1	avec un des types ci-dessus
	1	this."valeur"	R	1	d'après la valeur source
État	**1..***	**StateVar**	**I**	**?**	**Dépend des "types d'états" (paramètres)**
		this.isAlwaysTyped	R	1	d'après les types manipulés
		this.DEVSid	R	1	donner un nom à la variable d'état
valeur initiale	0..1	this.initial_value	R	1	donner une valeur initiale
Changement d'état	**1..***	**DeltaIntRule ou DeltaExtRule**	**I**	**1..***	**à créer selon le type de déclencheur et à lier à la fonction (int ou ext) correspondante**
état d'arrivée	1	this.changesState	I	1	Instancier la StateChangeAction
état de départ	1	this.tests	I	1	instancier une StateVarComparison
évènement ext.	0..1	this.testsInputEvent	I	0..1	instancier une InputPortComparison
Sortie	**0..***	**LambdaRule**	**I**	**1..***	**quand ta expire, une sortie doit être prévue**
état de sortie	0..1	this.tests	I	1	instancier une StateVarComparison
message de sortie	0..1	this.sendsMessage	I	1	instancier une OutputAction
Durée de vie état	**0..***	**TimeAdvanceRule**	**I**	**1..***	**Instancier au moins une règle TA**
		this.tests	I	1	instancier une StateVarComparison
		this.TA_value	R	0..1	durée de vie (par défaut 9999999)
		StateChangeAction	R	1..*	Renseigner chaque StateChangeAction créée dans le cadre des fonctions de transition
		this.StateToBeChanged	R	1	Renseigner la StateVar à changer
		this.new_value	R	1	Renseigner la DEVSExpression
		StateVarComparison	R	1..*	Renseigner chaque StateChangeAction créée dans le cadre des fonctions de transition
		this.left_member	R	1	Renseigner la StateVar
		this.right_member	R	1	Renseigner la DEVSExpression
		OutputAction	R	1..*	Renseigner chaque StateChangeAction créée dans le cadre des fonctions de transition
		this.port	R	1	Renseigner le port de sortie
		this.value	R	1	Renseigner la valeur de retour (LitteralBasicValue)
		InputPortComparison	R	1..*	Renseigner chaque StateChangeAction créée dans le cadre des fonctions de transition
		this.left_member	R	1	Renseigner le port d'entrée
		this.value	R	1	Renseigner la valeur à tester (LitteralBasicValue)

Tableau V-5 : Correspondances se rapportant à AtomicDEVS

5.3. Règles génériques de transformation

L'objectif de cette section est de proposer, en fonction des correspondances de concepts vues précédemment, des règles génériques de transformation M2M.

5.3.1. Discussion

a) *Intérêt d'utiliser des règles génériques*

Passer du tableau de correspondance de concepts à une implémentation directe est certes possible, mais on se priverait alors de l'atout essentiel de notre approche : la réutilisabilité des règles. Pour pouvoir bénéficier de cette dernière, il faut, tout comme lorsque l'on crée un méta-modèle, rester le plus longtemps possible en-dehors de toute implémentation. Nous avons donc besoin de cette étape intermédiaire.

Ceci permet de repérer les points communs que l'on retrouvera entre certaines règles et, lors de l'implémentation, nous nous en servirons pour écrire le minimum de règles possible tout en les réutilisant au maximum, au sein de la transformation, mais aussi pour d'autres transformations M2M vers DEVS. Cela se traduit par un recours à l'appel de règles, en d'autres termes, certaines règles qui seront « immuables » pour chaque cas de transformation, seront appelées par des règles qui, elles, changeront selon le contexte de chaque transformation, c'est-à-dire selon la nature du formalisme source. Les règles « immuables » (création dans le modèle cible de *LitteralBasicValue*, de *StateVarComparison*…etc.) pourront donc être réutilisées telles quelles, quel que soit le formalisme source.

b) *Justification de la technique de transformation employée*

Nous pensons que le standard QVT est le plus à même de répondre à nos besoins, notamment grâce au fait qu'il permette de créer des règles de manière impérative et déclarative. Des méta-modèles relativement complexes comme celui de DEVS ont selon nous besoin de cette souplesse au niveau des règles pour pouvoir être l'objet de transformations. De plus, le fait que le langage OCL soit aligné avec QVT permet de démultiplier la puissance d'expression des règles en autorisant l'utilisateur à exprimer des contraintes, formuler des requêtes, sur les modèles.

5.3.2. Pseudo-langage de description des règles de transformation

Nous avons fait le choix de décrire les transformations qui vont suivre par un ensemble de règles écrites en pseudo-code, mêlant approche impérative et déclarative, et ce dans un souci de lisibilité. Le nom de chaque règle est choisi afin d'être le plus explicite possible. Nous détaillons les actions à mener sur le modèle cible en fonction des éléments source, ces actions étant précédées par des commentaires détaillant la démarche suivie.

Cependant, afin de rester dans un esprit IDM et en particulier MDA, et donc de pouvoir traduire le plus facilement possible ces règles dans un langage de transformation de l'IDM, ce pseudo-langage s'inspire de la syntaxe décrite dans les spécifications de l'OMG de *MOF Query/View/Transformation Specification*, d'OCL, et aussi d'ATL, qui en constitue une implémentation. Nous renvoyons le lecteur au chapitre II de ce document ainsi qu'à la documentation officielle en ligne sur le site de l'OMG pour plus d'informations sur QVT et MOF.

De cette manière, tout en se plaçant hors de toute implémentation spécifique, nous pouvons décrire assez précisément cette transformation.

Chaque formalisme candidat à la transformation est unique : c'est pourquoi ces règles utilisent souvent une « navigation » rudimentaire dans les modèles source, nous nous appuyons simplement sur les hypothèses faites en 5.1.3 et 5.1.4 pour désigner certains éléments importants comme les états ou les transitions.

Notre objectif n'est pas de définir de manière exhaustive l'intégralité des règles envisageables mais de nous concentrer sur la démarche de création des règles. Nous focalisons donc notre présentation sur les points que nous jugeons importants dans une optique de réutilisation.

Nous employons souvent des couples de règles, dont certaines sont « immuables » (voir 5.3.1.a)), pour nous aider à créer un élément dans le modèle cible : une règle effectue les tests nécessaires, et passe en argument les résultats de ces tests à une seconde règle qui crée effectivement les éléments cibles. Cette technique est couramment employée pour les transformations M2M, elle permet d'alléger l'implémentation en rendant les règles aussi modulaires que possible. L'analogie avec la programmation est évidente : au lieu de coder une grosse fonction en un seul bloc, on préfère souvent la fragmenter en fonctions plus simples et plus petites que l'on appellera au fur et à mesure depuis une fonction principale. Nous nous aidons aussi, quand cela est possible, de la récursivité.

a) ***Transformation***

Une transformation se compose d'un ensemble de règles (voir **b)**). Chaque transformation possède une signature composée du mot-clef **`transformation`** suivi du nom de la transformation, et des mots-clefs **`in`** et **`out`** qui donnent les noms des modèles de départ (source) et d'arrivée (cible), ainsi que les méta-modèles auxquels il se conforment. Le mot-clef transformation n'est utilisé qu'une fois par transformation.

b) ***Règle***

Une règle de transformation nommée **`rule`** se décrit syntaxiquement par une signature et un corps, et comporte éventuellement une garde (entre crochets) et/ou une post-condition. De manière générale, les équivalences entre éléments source et cible sont décrits par une relation d'égalité, dont le membre gauche fait référence à la cible et le membre droit à la source. Une règle accepte des arguments en entrée (ces arguments peuvent être des paramètres, des objets, des éléments des modèles). Enfin, les règles peuvent s'appeler entre elles, à l'instar des fonctions en programmation classique. Ceci permet leur réutilisabilité (une même règle peut être utilisée dans des cas différents).

c) ***Autres mots-clefs et symboles employés***

Le mots-clef **`to`** fait référence aux éléments cibles, le mot-clef **`from`** désignant quant à lui un ou des éléments du modèle source.

Le mot-clef **`foreach()`** est utilisé pour effectuer une action sur chaque élément de la collection d'objets entre parenthèses.

Le symbole ! placé après un méta-modèle et suivi d'un nom de classe désigne une classe spécifique à ce méta-modèle (ex : `DEVS!AtomicDEVS`).

La mention **`renseigné manuellement`** indique que selon le type de transformation, plus exactement selon le type du méta-modèle source, il faut remplir le champ correspondant manuellement, ou alors implémenter une fonction de vérification à l'aide d'OCL par exemple.

Nous nous sommes attachés à utiliser des termes explicites pour désigner les propriétés des objets, par exemple, même si on parle d'un modèle source générique `s`, on comprendra sans peine que `s.Ports` indique la référence vers les ports de `s`.

5.3.3. Règles de base : modèles atomiques et couplés

Ici, `SourceMM` désigne le méta-modèle source, et `s` désigne l'instance qui s'y conforme. Il suffira, pour adapter une règle à un contexte particulier, de remplacer `SourceMM` et `s` par les éléments qui correspondent au méta-modèle source spécifique (ex : `BasicDEVS` et son instance `bd`).

Le choix de créer un modèle atomique ou couplé à partir d'un modèle en entrée dépend la plupart du temps de la nature du modèle source : s'il contient lui-même d'autres modèles, ou contient ce qui s'apparente à des états.

Nous présentons tout d'abord les deux règles basiques de création de modèles atomiques et couplés, nous les combinons dans une troisième qui sera celle à privilégier lors de transformations, et nous décrivons enfin les fonctions de création de ports et d'instanciation des types.

a) *Fonctions auxiliaires*

Nous utilisons ici deux fonctions auxiliaires simples.

`containsModels() : Boolean` Appelée sur un modèle en entrée renvoie **`true`** si le modèle contient des sous-modèles, **`false`** sinon. Cette fonction est spécifique à chaque formalisme source.

`subModel() : List` appelée sur un modèle en entrée renvoie une liste de sous-modèles. Si la fonction renvoie 0, la liste est vide donc le modèle ne contient pas de sous-modèle.

b) *Déclaration de la transformation*

```
transformation unknownFormalismToDEVS(in m:unknownFormalism, out dev:DEVS) {
        main() {
                [règles]
        }
}
```

c) *Création d'un modèle atomique*

Cette règle accepte en entrée un modèle ne contenant pas de sous-modèles et crée le modèle AtomicDEVS correspondant, en renseignant son nom puis en instanciant une seule fois chacune des 4 fonctions DEVS.

```
rule createAtomicDEVS(s : SourceMM!Model)
to dev : DEVS!AtomicDEVS
      dev.name=s.name;
```

```
DEVS!DeltaInt;
DEVS!DeltaExt;
DEVS!Lambda;
DEVS!TA;
dev.Types=createTypes()
dev.inputPort = collectInputPorts (s)
dev.outputPort = collectOutputPorts(s)
```

La fonction précédente présente un inconvénient : elle n'accepte en entrée que les modèles équivalents des modèles atomiques, et le contrôle du modèle en entrée est laissé à la charge du programmeur. C'est pourquoi il est préférable, selon nous, de ne pas l'utiliser telle quelle mais plutôt de l'inclure dans une règle de création de hiérarchie, comme nous le montrons en e).

En plus des fonctions grisées, nous enrichissons cette règle en y ajoutant une référence vers les LitteralBasicValues et les StateVar manipulées, via les relations *handles* et *is_defined_by* de la méta-classe *AtomicDEVS* (voir 5.3.4.a).

d) ***Création d'un modèle couplé***

Cette règle récursive accepte en entrée un modèle contenant des sous-modèles et crée le modèle *CoupledDEVS* correspondant, en renseignant son nom. Ensuite, elle teste si le modèle reçu ne possède pas de sous-modèles : dans ce cas elle appelle la règle `createAtomicDEVS()`. Sinon, elle se rappelle elle-même sur chaque sous-modèle de `s`.

```
rule createCoupledDEVS(s : SourceMM!Model)
 to dev : DEVS!CoupledDEVS
 dev.name=s.name;
dev.Types=createTypes()
dev.inputPort = collectInputPorts (s)
dev.outputPort = collectOutputPorts(s)
      dev.contains = foreach (s.subModel) {
      if s.subModel.containsModels()=true
            createCoupledDEVS(s.subModel)
     else createAtomicDEVS(s.subModel)
      }
```

Le défaut de cette règle est qu'elle est limitée aux modèles source ne comportant pas de sous-modèles. Il est donc nécessaire de la combiner avec la précédente dans une règle qui tient compte de cette possibilité : la règle de création d'une hiérarchie.

e) ***Création d'une hiérarchie***

Cette règle ne comporte pas de champ **`to`**. Ce sont les règles qui sont appelées à l'intérieur qui se chargent de créer les éléments cibles.

```
rule createHierarchy(s : SourceMM!Model)
if s.containsModels()
      createCoupledDEVS(s);
else
      createAtomicDEVS(s);
```

C'est cette règle qu'il faut utiliser lors de transformations mettant en œuvre des modèles sources pouvant en contenir d'autres. Implémentons à présent les règles dont le but est de créer les ports et d'instancier les types.

f) ***Types***

L'instanciation des types ne pose pas de problème particulier, elle ne nécessite pas de modèle source puisque nous n'avons besoin d'aucune information concernant ce dernier. Dans le cas où certains modèles source possèdent des types, la définition des règles de transformation restera, quoi qu'il en soit, aisée.

```
rule createTypes() {
 to dev : DEVS!DEVSModel
      DEVS!StringType
      DEVS!IntegerType
      DEVS!CharType
      DEVS!BooleanType
}
```

g) ***Ports de sortie***

Nous distinguons les ports d'entrée et de sortie. La règle collectOutputPorts() demande en entrée l'instance du modèle source. Si le nombre de ports de sortie du modèle source n'est pas nul, la règle teste pour chacun d'eux s'il possède un nom. Si c'est le cas, elle appelle la règle createOutputPort() en lui passant le nom du port en paramètre, « default » sinon. Danstous les cas, collectOutputPorts() va appeler au moins une fois createOutputPort() en lui passant comme argument « default » car un modèle DEVS ne peut pas ne pas contenir de port de sortie.

Le type du port est renseigné manuellement mais peut également être automatisé avec l'aide d'une fonction (dont le but est de tester le type du port passé en paramètre). La description algorithmique d'une telle fonction et son implémentation ne présentent aucune difficulté particulière. Cependant, nous choisissons de ne pas détailler ici cette fonction car cela alourdirait nos règles et rendrait leur lecture moins aisée.

```
rule collectOutputPorts(s : SourceMM!Model) {
     if s.Ports.OutPut<>0
      foreach (s.Ports.OutPut){
            if s.Ports.OutPut.name<>0
            createOutputPort(s.Ports.OutPut.name)
                  else
                  createOutputPort("default")
      }
     else
      createOutputPort("default")
}

rule createOutputPort(n:String) {
 to outP : DEVS!OutputPort
                  outP.portID=n
                  outP.isAlwaysTyped=renseigné manuellement
}
```

h) ***Ports d'entrée***

Cette règle est légèrement plus simple que les précédentes (un modèle DEVS peut ne pas avoir de port d'entrée). Son fonctionnement est le même que la règle concernant les ports de sortie.

```
rule collectInputPorts(s : SourceMM!Model)
      foreach (s.Ports.InPut){
```

```
                if s.Ports.InPut.name<>0
                createOutputPort(s.Ports.InPut.name)
                     else
                     createOutputPort("default")

rule createInputPort(n:String)
 to inP : DEVS!InputPort
                      inP.portID=n
                     inP.isAlwaysTyped=renseigné manuellement
```

i) ***Note sur les fonctions de couplage***

Les fonctions de couplage ne seront pas abordées exhaustivement ici. Voici toutefois une description de la marche à suivre pour transformer des relations à plusieurs niveaux entre modèles en couplages DEVS. Cette dernière est simple :

- Si le modèle d'entrée est un modèle en contenant d'autres, collecter les ports de ce dernier, les ports de ses sous-modèles ;
- Trouver l'équivalent des fonctions de couplage DEVS : si deux de ses sous modèles sont interconnectés, il s'agit d'un IC, si un de ses ports d'entrée est relié au port d'entrée d'un de ses sous-modèles, il s'agit d'EIC, si un port de sortie d'un de ses sous-modèles est lié à un de ses propres ports de sortie, il s'agit d'EOC.

5.3.4. Règles spécifiques à AtomicDEVS

a) ***Répertorier les valeurs qui deviendront LitteralBasicValues***

À présent, nous nous proposons de collecter l'intégralité des valeurs manipulées par le modèle source et de les instancier sous forme de LBV dans le modèle DEVS cible avec une seule règle. Nous restons très généraux, l'implémentation étant trop liée au type de méta-modèle source. On suppose donc ici que l'intégralité des valeurs a été collectée suite à l'exécution de ces règles.

```
rule collectLBV(s : SourceMM!Model) {
      foreach (s.valeursManipulees){
      //selon le formalisme d'entrée, la collecte des valeurs est
     différente, nous supposons ici qu'elle est déjà faite grâce à
     « valeursManipulees » et que le type est vérifié
      createLBV(s.valeursManipulées)
      } }

rule createLBV(m : valeur) { //le type de la valeur n'est pas connu, on le
//renseignera donc manuellement, mais on peut aussi écrire une fonction qui le
fait automatiquement
 to lbv : DEVS!LitteralBasicValue
                     lbv.isAlwaysTyped=renseigné manuellement
                     lbv.(int/str/char/bool)val=m.valeur
}
```

Pour finir, il faut simplement créer un lien entre la référence *handles* de AtomicDEVS et ces LBV, en appelant `collectLBV()` sur l'instance `s` :

```
dev.handles = collectLBV(s)
```

Cette ligne doit figurer dans la règle vue en 5.3.3.c).

b) ***Création d'une ou plusieurs StateVar***

L'état d'un modèle atomique DEVS à un instant *t* est caractérisé par les valeurs des différentes StateVar qui le composent (notion de dimension). La tâche qui consiste à créer ces StateVar dans le modèle cible en fonction du modèle source dépend de la représentation des états dans le formalisme source.

Au vu de la caractérisation des formalismes d'entrée que nous avons effectuée, nous pouvons supposer ici que les différents types d'états du modèle source sont connus et que leurs valeurs ont été créées sous forme de LitteralBasicValues dans le modèle cible. Nous supposons par ailleurs que la valeur initiale de chaque famille d'états (qui correspondront à des StateVar) du modèle source est connue. Il ne faut pas oublier que les StateVars ne contiennent pas toutes des types énumérés et que les valeurs littérales qu'elles sont susceptibles d'avoir ne sont pas forcément connues à l'avance. L'essentiel est que chaque StateVar créée possède une valeur initiale.

Nous proposons ici le squelette de cette règle dont le but est de créer une StateVar et de renseigner sa valeur initiale avec une LBV déjà créée. Cette règle doit ensuite être adaptée au cas par cas. Elle est appelée pour chaque nouvelle StateVar que l'on désire créer.

```
        rule createStateVar() {
        to sv : DEVS!StateVar
             sv.DEVSid= 'stateVar1' //à incrémenter automatiquement
             sv.isAlwaysTyped=renseigné manuellement
             sv.initial_value=<LBV créée précédemment correspondant à l'état
initial>
       }
```

c) ***Création des règles correspondant aux* Conditions**

Avant de pouvoir exprimer des fonctions DEVS à partir notamment des transitions présentes dans le modèle source, nous devons disposer des éléments qui composent nos règles DEVS (pour mémoire, composées de *Conditions* et d'*Actions*).

- fonctions permettant de réaliser des tests (sur les états, le port d'entrée…),
- fonctions permettant de décrire des actions (changement d'état, envoi de valeur sur une sortie).

Les règles de transformation dont nous fournissons la description ci-après ne seront jamais utilisées de manière isolée mais génèreront les briques des *DEVSRules* composant chaque fonction DEVS.

La règle qui nous sera la plus utile est celle qui permet la création d'une *StateVarComparison*. Le membre gauche de cette règle porte sur une StateVar, que l'on supposera unique pour simplifier, et le membre droit contiendra une LBV choisie parmi les LBV déjà créées dans le modèle de destination, telle que cette LBV corresponde à la valeur de l'état source de la transition passée en paramètre. Si la StateVar n'était pas unique dans le modèle DEVS cible, il faudrait alors implémenter, en amont, une fonction qui sélectionne celle qui est concernée par la transition.

```
       rule createSVC(s: SourceMM!State, sv: DEVS!StateVar) {
       to sv : DEVS!StateVarComparison
```

```
                left_member <- sv, // on suppose que la //SVC se fait toujours
                sur stateVar1
          right_member <- selectionner(DEVS!LBV, s.valeur)
    }
```

Dans le même ordre d'idées, on peut implémenter simplement la règle `createIPC()` qui se charge de créer une *InputPortComparison* à partir de la transition (supposée évènementielle) passée en paramètre, ou à partir d'un évènement d'entrée. Le membre gauche de cette règle est un port d'entrée, choisi de la même manière que la fonction `selectionner()` ci-dessus a choisi une LBV dont la valeur correspondait avec la valeur de l'état source. Le membre droit est une DEVSXpression, en général une LBV, choisie aussi de la même manière.

```
rule createIPC(t: SourceMM!Transition) {  //t est supposée évènementielle
     left_member <- selectionner(DEVS!InputPort, t.valeurPort)
     right_member <- selectionner(DEVS!LBV, t.valeurEvenementEntree)
}
```

d) *Création des règles correspondant aux* Actions

Commençons par décrire la règle la plus simple, celle qui à partir d'une transition du modèle source décrit en termes DEVS un changement d'état (*StateChangeAction*). Elle se base également sur une transition, et son membre gauche est `stateVar1`. Cette règle ne présente aucune difficulté particulière.

```
rule createSCA(t: SourceMM!Transition, sv: DEVS!StateVar) {
      to sv : DEVS!StateChangeAction
                state_to_be_changed <- sv
          new_value <- selectionner(DEVS!LBV, t.valeurEtatCible)
      }
```

Enfin, définissons la règle qui décrit l'action d'envoyer une valeur sur un port de sortie (*OutputAction*). Le modèle source n'a pas forcément de port de sortie ni de règle associée. Mais, comme DEVS a toujours besoin d'une fonction de sortie, il faudra pourtant en créer une.

Dans ce cas, la fonction de sortie sera associée à un état (en fait, à la valeur d'une StateVar), et il incombe au méta-modeleur de choisir de quelle manière : arbitrairement si le formalisme source ne se prête pas à la création de fonctions de sortie dans DEVS, ou pas si au contraire certaines transitions du formalisme source peuvent correspondre à des sorties DEVS. Un bon exemple est celui des états terminaux d'un automate à états finis passé en entrée.

Ici, on supposera que si elle existe dans le modèle source, l'action de renvoyer une valeur sur un port de sortie est contenue dans le modèle. Le port est choisi parmi la liste de ports du modèle DEVS dont le *portID* correspond au port de sortie concerné dans le modèle d'entrée. Le message est choisi parmi les LBV créées dans le modèle DEVS.

```
rule createOA(m: SourceMM!Model) {
      to sv : DEVS!OutputAction
            if (m.output<>0) {
                message <- selectionner(DEVS!LBV, m.output.message)
                port <- selectionner(DEVS!OutputPort, m.output.idPort)}
}
```

e) ***Fonctions DEVS***

À partir des règles ci-dessus, nous pouvons construire facilement dans le modèle cible toutes les règles DEVS dont nous avons besoin.

Commençons par la plus simple, la règle qui crée des *DeltaIntRule*. On suppose que la transition passée en paramètre est une transition temporelle, et pas une transition déclenchée par une perturbation extérieure survenant sur un port d'entrée.

```
rule createDeltaIntRule(t: SourceMM!Transition, sv: DEVS!StateVar) {    //t est
temporelle
      to sv : DEVS!DeltaIntRule
                  tests <- createSVC(t.source,sv)
                  changes_state <- createSCA(t,sv)
      }
```

Créer des TARules n'est guère plus complexe. Il suffit de récupérer dans le modèle source la durée de vie de chaque état.

```
rule createTARule(s: SourceMM!State, sv: DEVS!StateVar) {    //t est temporelle
      to sv : DEVS!TimeAdvanceRule
                  tests <- createSVC(s,sv)
            ta_value <- t.time
      }
```

Créer une *DeltaExtRule* se fait de manière très similaire à la création d'une *DeltaIntRule*. Il suffit seulement de renseigner en plus le type d'évènement qui surviendra. Ici la transition t est supposée évènementielle.

```
rule createDeltaExtRule(t: SourceMM!Transition, sv: DEVS!StateVar) {    //t est
évènementielle
      to sv : DEVS!DeltaExtRule
                  tests <- createSVC(t.source,sv)
                  tests_input_event <- createIPC(t)
                  changes_state <- createSCA(t,sv)
      }
```

Enfin, une *LambdaRule* dispose, en plus d'une *StateVarComparison* basique, du message que le modèle devra envoyer sur le port de sortie. Elle est, selon la définition de base du formalisme DEVS, associée à un état, et théoriquement prend donc en paramètre un état, non une transition. Cependant, nous faisons le choix de passer en paramètre de cette règle de transformation une transition, pour une raison pratique : il arrive souvent que la sortie soit associée à une transition, dans les formalismes de la famille F_2 (d'où l'appel à *t.message*)

```
rule createLambdaRule(t: SourceMM!Transition, sv:DEVS!StateVar){
//t est temporelle
      to sv : DEVS!LambdaRule
                  tests <- createSVC(s,sv)
                  sends_message <- createOA(t.message)
      }
```

f) ***Synthèse***

Nous venons d'énoncer des règles qui, pour une famille donnée, caractérisée en 5.1.4, permettent de transformer tout modèle source en un modèle DEVS équivalent, typé par le méta-modèle MetaDEVS. Nous allons maintenant aborder concrètement deux cas de transformations vers DEVS, en adaptant, toujours à l'aide du pseudo-code que nous avons

présenté, les fonctions génériques ci-dessus. Nous ne ferons pas figurer ici les collectes des éléments dans les modèles sources, car elles alourdiraient considérablement notre description. De plus, la manière de collecter des éléments dans les modèles sources dépend en grande partie de l'implémentation choisie (i.e. du langage M2M employé). Nous nous contenterons donc de décrire brièvement ces collectes et nous nous concentrerons uniquement sur l'élément essentiel des règles de transformation : le corps des règles de transformation proprement dites. Nous renvoyons le lecteur aux annexes de ce mémoire pour consulter le code complet, implémenté, des transformations M2M avec ATL : l'annexe 5 donne le code de la transformation BasicDEVS vers DEVS, l'annexe 6 le code de la transformation FSM vers DEVS.

5.4. Transformation M2M : de BasicDEVS vers DEVS

Nous débutons par ce cas de transformation car c'est selon nous le plus simple à appréhender. Il s'agit d'une transformation horizontale et exogène depuis un formalisme que nous avons créé, qui constitue un sous-ensemble de DEVS, vers le formalisme DEVS lui-même (incarné par le méta-modèle MetaDEVS présenté au chapitre précédent). Ce formalisme baptisé BasicDEVS a été présenté dans [Garredu et al. 2009].

5.4.1. Présentation du méta-modèle BasicDEVS et contraintes associées

Ce formalisme est très simple : il permet de créer des petits modèles proches de DEVS dans lesquels certains concepts de DEVS sont simplifiés. Contrairement à MetaDEVS qui permet la spécification d'états multidimensionnels grâce au concept de StaveVar (représentant une variable d'état), les états dans BasicDEVS sont unidimensionnels, spécifiés en extension lors de la création d'un modèle, et représentés par des chaînes de caractères. Les messages émis ou reçus sont également des chaînes de caractères.

La création de modèles interconnectés est impossible, mais ce formalisme permet toutefois de spécifier des évènements d'entrée et de sortie, ce qui permettra, une fois les modèles transformés en modèles atomiques DEVS, de les coupler entre eux.

Nous avons recours ici à une capture d'écran de l'éditeur EMF pour présenter ce méta-modèle (Figure V-4), car il est simple et ne comporte pas beaucoup de méta-classes. Bien entendu, pour le présenter, nous aurions tout aussi bien pu utiliser le formalisme UML (comme nous l'avons fait pour le méta-modèle de DEVS), ou même le formalisme XMI (quoi que moins « lisible »), ces trois formalismes étant pratiquement équivalents au niveau des concepts décrits.

Insistons à nouveau sur le fait que tous les méta-modèles présentés dans ce travail n'offrent qu'une syntaxe abstraite d'un formalisme : nous employons systématiquement OCL pour affiner ces méta-modèles avec une sémantique statique, lorsque les seules cardinalités ne permettent pas une précision suffisante. Du fait de l'inclusion de ce DEVS-Like (baptisé « BasicDEVS » pour l'occasion) dans DEVS, ces deux formalismes possèdent beaucoup de concepts en commun, ce qui rend cette transformation particulièrement aisée à écrire et à implémenter.

Note : BasicDEVS étant une vue simplifiée du formalisme DEVS, il est plus intuitif mais possède une puissance de modélisation très réduite.

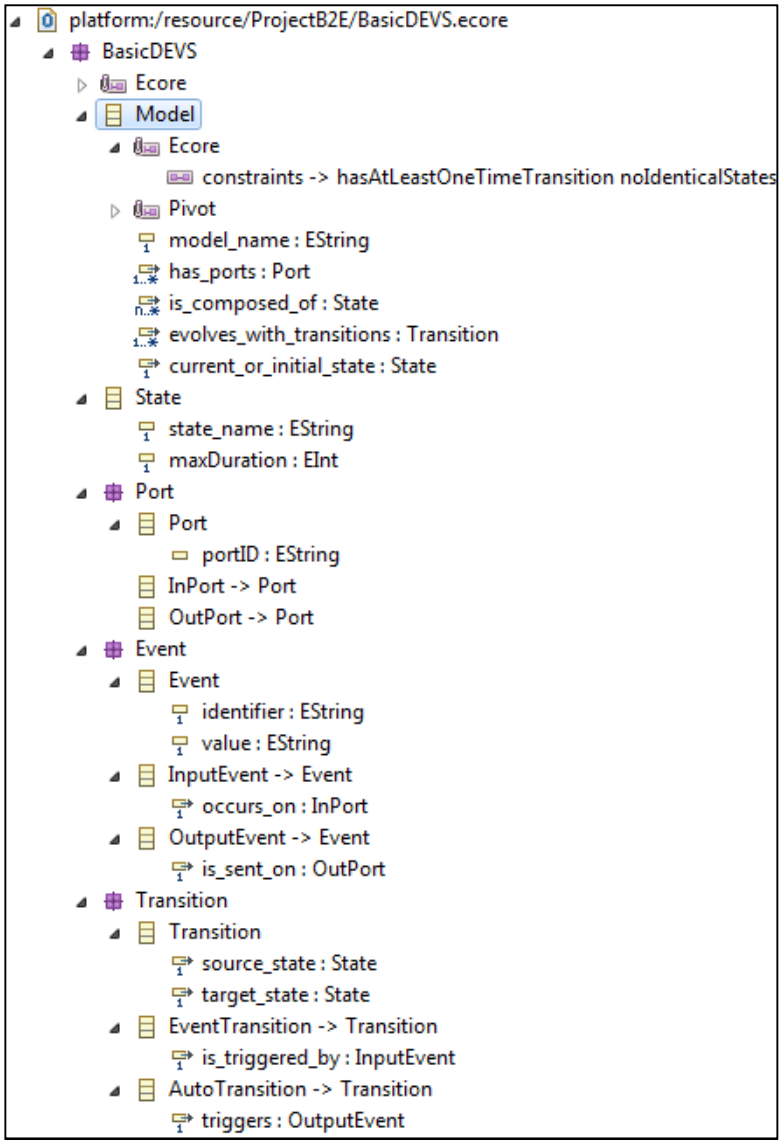


Figure V-4 : Méta-modèle Ecore de BasicDEVS vu dans l'éditeur

Le modèle, pour être valide, doit comporter au moins :

- un nom (cardinalité 1..1 sur le méta-modèle),
- deux états, nommés et possédant une durée de vie, dont la valeur est 0 par défaut, ce qui correspond à un état transitoire (cardinalité 2..* sur le méta-modèle),
- un port,
- une transition interne (*AutoTransition*) avec une sortie associée.

De plus, les états doivent être distincts : deux états ne peuvent donc avoir le même libellé. Ceci ne peut pas se spécifier à l'aide de cardinalités, mais peut l'être en ayant recours à la puissance d'expression du langage OCL.

a) ***Comment créer un modèle BasicDEVS ?***

On crée un modèle BasicDEVS en suivant les étapes suivantes :

1. **Identifier le modèle :** donner un nom au modèle, déclarer son ou ses ports et les nommer. Les ports sont de type « entrée » (*Inport*) ou « sortie » (*Outport*) ;
2. **Énumérer ses états :** à partir des états qui ont été identifiés, les nommer, et affecter à chacun d'eux une durée de vie (via la propriété maxDuration).

   Remarque : de par la nature même du méta-modèle, on voit ici que tous les états du système seront du même type (chaîne de caractères) et font référence à la même variable qualitative. On peut d'ores et déjà voir qu'une seule *StateVar* devra être créée dans le modèle de sortie ;
3. **Identifier les transitions et leur déclencheur :** il ne doit pas y avoir d'état orphelin, i.e. chaque état doit être soit un état source, soit un état cible d'une transition. Pour une transition temporelle, qui permet au système d'évoluer de manière autonome, il n'est pas nécessaire de renseigner autre chose que les états source et cible. Pour une transition évènementielle, résultant d'un stimulus provenant du monde extérieur sur le port d'entrée du modèle, il est en revanche obligatoire de spécifier quel est l'évènement d'entrée déclencheur (identifiant, port, valeur). Dans tous les cas, le déterminisme doit être respecté, i.e. 2 transitions temporelles ne peuvent pas avoir le même état source, et 2 transitions évènementielles ne peuvent pas avoir le même état source et le même évènement déclencheur.

5.4.2. Définition des règles génériques de transformation

Pour mieux comprendre la méthode employée, il est préférable d'avoir sous les yeux le méta-modèle de BasicDEVS, le méta-modèle de DEVS, les tableaux de correspondances de concepts spécifiques aux modèles atomiques présenté en 5.2.2.b) ainsi que les règles de transformation génériques que nous venons d'énoncer.

Nous ne faisons figurer ici que les règles importantes, en employant le même pseudo-langage que pour la description générique des règles.

a) **Helpers *(fonctions auxiliaires)***

Nous utilisons 3 *helpers* ou fonctions pour nous aider dans cette transformation. Leur code complet figure en annexe, nous ne donnons ici que leur signature.

`isInPort()` : renvoie *true* si le port de *BasicDEVS* testé est de type *InPort*, *false* sinon

`isAutoTransition()` : renvoie *true* si la transition de *BasicDEVS* testée est de type *AutoTransition*, *false* sinon

`isEventTransition()` : renvoie *true* si la transition de *BasicDEVS* testée est de type *EventTransition*, *false* sinon

b) ***Vue d'ensemble de la transformation BasicDEVS vers DEVS***

Nous proposons ici une vue d'ensemble des correspondances de concepts pour cette transformation, ce qui constitue une adaptation du Tableau V-2 aux spécificités d'une transformation de BasicDEVS vers DEVS. Certaines règles de cette transformation sont ensuite détaillées au moyen du pseudo-langage que nous avons proposé.

c) **Déclaration *de la transformation***

Cette transformation met en œuvre une instance de *BasicDEVS* que nous nommerons pour la suite *bd*, et vise à créer une instance d'un *DEVSModel*.

```
transformation BasicDEVSToDEVS(in bd:BasicDEVS, out dev:DEVS) {
        main() {
                    [règles]
                }
}
```

SOURCE	*DESTINATION*
Model	AtomicDEVS
Liste de **State**	Plusieurs **StringValue** dans une seule StateVar (DEVSid="StateSet")
Liste de **State**	TimeAdvanceRule basée sur la variable *maxDuration* de chaque état
InPort (**OutPort**)	InputPort (OutputPort)
AutoTransition	DeltaIntRule basée sur le nom de l'état de départ et de celui d'arrivée
EventTransition	DeltaExtRule basée sur le nom de l'état et sur l'évènement d'entrée
OutputEvent	LambdaRule basée sur le nom de l'état et sur l'évènement de sortie

Tableau V-6 : Vue d'ensemble de la transformation BasicDEVS vers DEVS

d) ***Création du container***

Nous devons en premier lieu instancier l'objet qui contiendra tous les autres : ici, c'est tout simplement le modèle atomique AtomicDEVS. Il doit posséder le même nom que le modèle BasicDEVS.

```
rule instanciationAtomicDEVS()
  from bd : BasicDEVS!Model
   to dev : DEVS!AtomicDEVS
     dev.name=model.model_name
```

Nous considérons à présent l'instance *dev* créée, et nous la renseignerons au fur et à mesure, parfois sans utiliser de **from**, ce qui correspond au fait qu'un élément cible n'a pas forcément d'antécédent dans le modèle source (par exemple, les types de base).

e) ***Types de base et fonctions DEVS***

Nous devons à présent créer les quatre fonctions DEVS (vides pour le moment). Cette règle s'exécute sans tenir compte d'un élément en entrée. Nous n'instancions ici que le singleton *StringType* car le modèle ne manipulera que des chaînes de caractères.

```
rule fonctionEtTypes()
       to dev : DEVS!AtomicDEVS
              DEVS!StringType
              DEVS!DeltaInt
              DEVS!DeltaExt
              DEVS!Lambda
              DEVS!TA
```

f) ***Créer des LitteralBasicValue***

```
rule collecteCreationLBV()
    from bd : BasicDEVS!Model
     to dev : DEVS!AtomicDEVS
        [foreach état, entrée ou sortie]
        Dev.handles=createLBV(valeur)
```

g) ***Créer l'unique StateVar du modèle***

```
rule createStateVar()
     from bd : BasicDEVS!Model
     to sv : DEVS!StateVar
            sv.DEVSid= 'stateVar1' //à incrémenter automatiquement
            sv.isAlwaysTyped=StringType
            sv.initial_value=<LBV créée précédemment correspondant à l'état
initial>
```

h) ***Créer les ports***

Cette règle est très similaire à la règle générique. La seule différence est que, comme pour les autres DEVSXpression qui seront manipulées par le modèle, le type utilisé pour les valeurs littérales est StringType.

```
inP.isAlwaysTyped=StringType
outP.isAlwaysTyped=StringType
```

i) ***Les fonctions atomiques***

Le formalisme d'entrée BasicDEVS manipule des concepts que l'on peut assimiler à des fonctions DEVS. L'analogie qui lie les *EventTransition* aux règles composant DeltaExt est la plus évidente. En effet, une *EventTransition* possède un état source et un état cible, et est conditionnée par la réception d'un message sur un port d'entrée (évènement d'entrée).

Une *AutoTransition* quant à elle, est à rapprocher des règles composant DeltaInt : elle contient uniquement un état source et un état cible, elle sera déclenchée en fonction l'attribut *maxDuration* que possède chaque état d'un modèle BasicDEVS. Rappelons que cet attribut contient la durée de vie de l'état. Chaque couple état-maxDuration est à relier à la notion de *TimeAdvanceRule*.

Un *OutputEvent* est quant à lui très proche d'une règle *LambdaRule*. Il est déclenché par l'exécution d'une *AutoTransition*. Pour chaque exécution, un message est envoyé sur un port de sortie.

Concernant les règles qui vont être utilisées par les 4 fonctions DEVS, il n'y a pas de grande différence avec les règles génériques énoncées en 5.3.4.c) et 5.3.4.d) (correspondant aux *Conditions* et aux *Actions*). Il faut juste remplacer le libellé de certains éléments par leur

nom exact dans BasicDEVS. Il en va de même pour les fonctions DEVS, dont une bonne partie est identique aux règles génériques, mis à part les arguments qu'elles attendent. Nous ne ferons figurer ici que les règles permettant de générer des fonctions DEVS.

```
rule createTARule(s: BasicDEVS!State, sv: DEVS!StateVar) {    //t est temporelle
      to sv : DEVS!TimeAdvanceRule
            tests <- createSVC(s.state_name,sv)
            ta_value <- t. source_state.maxDuration
      }
rule createDeltaIntRule(t: BasicDEVS!AutoTransition, sv: DEVS!StateVar) {    //t
est une AutoTransition
      to sv : DEVS!DeltaIntRule
                  tests <- createSVC(t.source_state,sv)
                  changes_state <- createSCA(t,sv)
      }
rule createDeltaExtRule(t: BasicDEVS!EventTransition, sv: DEVS!StateVar) {    //t
est une EventTransition
      to sv : DEVS!DeltaExtRule
                  tests <- createSVC(t.source_state,sv)
                  tests_input_event <- createIPC(t)
                  changes_state <- createSCA(t,sv)
      }
```

5.4.3. **Exemple de transformation n°1**

a) ***Modèle de feu tricolore BasicDEVS***

Créons à présent un modèle BasicDEVS, conforme au méta-modèle présenté en 5.4.1. Notre choix se porte sur un modèle autonome : un feu tricolore. Simplifié au maximum, ce modèle n'évolue pas en fonction des données extérieures mais suivant le temps. Le formalisme BasicDEVS nous permet de modéliser rapidement un tel système, qui sera composé d'un modèle comportant trois états énumérés, correspondant chacun à une des trois couleurs du feu : « vert », « rouge », « orange ».

Nous associons à chacun de ces trois états une durée de vie via l'attribut de durée maximale de vie : le feu rouge durera 30 secondes, le vert 25 secondes, l'orange 5 secondes. Nous créons ensuite trois changements d'état : de « vert » à « orange », de « orange » à « rouge », de « rouge » à « vert ».

Enfin nous associons à chaque changement d'état une sortie sur un port préalablement créé : cette sortie matérialise le passage d'un état à un autre en envoyant sur le port une chaîne de caractères.

La Figure V-5 représente une capture d'écran (sous Eclipse) d'une instance dynamique du système de feu tricolore que nous avons décrit. Le modèle est conforme au méta-modèle BasicDEVS. Pour des raisons de lisibilité, les propriétés des éléments figurent en encadré, et nous n'avons fait figurer que la dernière *AutoTransition*, les autres étant toutes construites sur le même principe, comme nous l'avons énoncé ci-dessus.

Nous pouvons voir sur cette figure que trois états ont bien été créés et nommés, leurs de vies respectives ont été renseignées. Le modèle possède un port de sortie nommé « outputCrosswalk » et l'état initial du système est « green ».

Au bas de la figure se trouvent les propriétés de la dernière *AutoTransition* qui spécifie que de l'état « red » le modèle évolue vers l'état « green ». L'évènement de sortie associé envoie le message « redToGreen ! » sur le port de sortie du modèle.

Note : Lors de la transformation vers MetaDEVS, l'information portée par l'attribut « identifier » de l'*OutputEvent* ne sera pas utilisée.

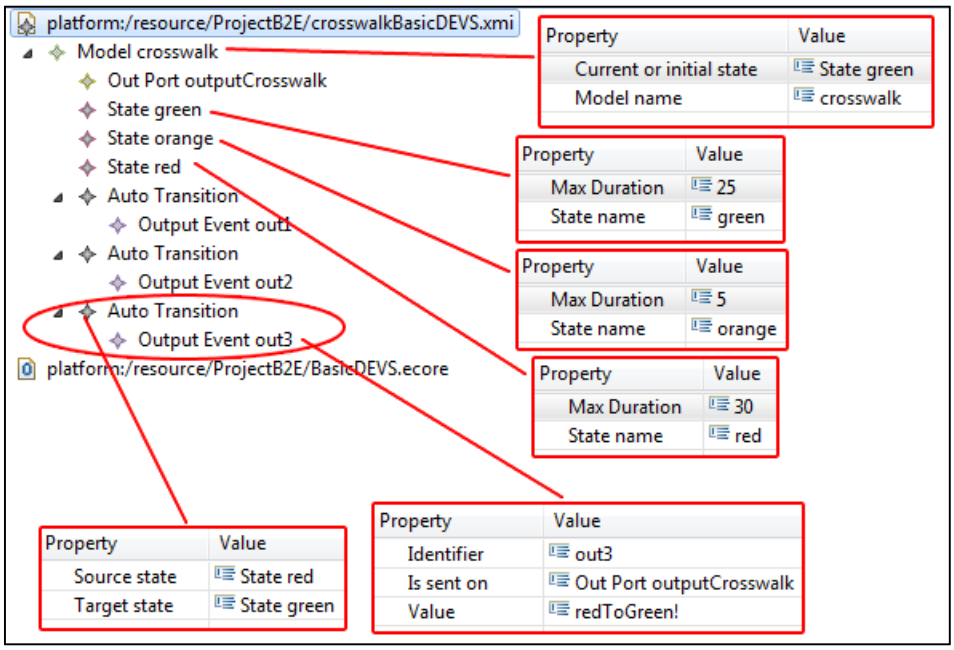


Figure V-5 : Capture d'écran reconstituée d'un modèle autonome de feu tricolore BasicDEVS

b) ***Transformation vers MetaDEVS***

Nous avons implémenté en ATL les règles génériques énoncées en 5.4.2. Cette implémentation est très proche du pseudo-lange que nous avons employé. La définition de cette transformation de BasicDEVS vers MetaDEVS a été appliquée sur l'instance de BasicDEVS présentée en a). L'exécution de la transformation est totalement automatique. Pour des raisons de lisibilité, nous préférons montrer le résultat de cette transformation (un fichier .xmi) sous une forme graphique, certes basique, grâce à des captures d'écran Eclipse, plutôt que de donner le fichier .xmi brut qu'a généré la transformation. Nous procèderons de même pour les autres transformations de ce chapitre.

La Figure V-6 montre une vue d'ensemble du résultat généré par la transformation : un modèle MetaDEVS. On y voit que ce modèle est composé d'un unique modèle atomique appelé « crosswalk », et que ce modèle contient lui-même des éléments propres à MetaDEVS et donc au formalisme DEVS :

- Un singleton StringType qui est utilisé pour typer chaque DEVSXpression ;
- Un port de sortie « outputCrosswalk » récupéré sur le modèle source, et de type StringType (n'apparaît pas sur la figure) ;
- Les quatre fonctions comportementales MetaDEVS : DeltaInt, DeltaExt, Lambda, TimeAdvance (avec DeltaExt laissée vide, ce qui est normal car le

modèle passé en entrée est autonome). Ces fonctions sont détaillées dans les captures d'écran suivantes ;

– Des LitteralBasicValue de type chaîne de caractères : trois d'entre elles contiennent les noms des états du système, trois autres les messages à envoyer sur le port de sortie ;

– Une variable d'état, nommée par défaut « StateSet », dont les propriétés figurent en encadré : une valeur initiale à « green », récupérée sur le modèle initial, et un typage StringType.

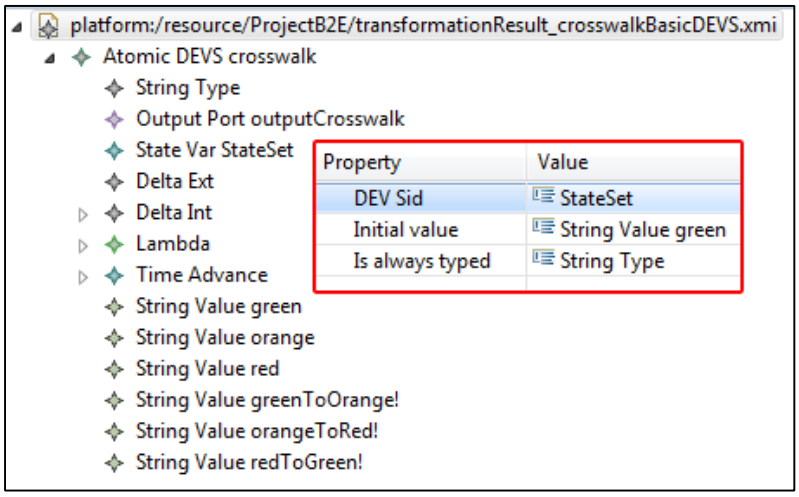


Figure V-6 : Capture d'écran globale du modèle de feu tricolore généré par BasicDEVSToDEVS

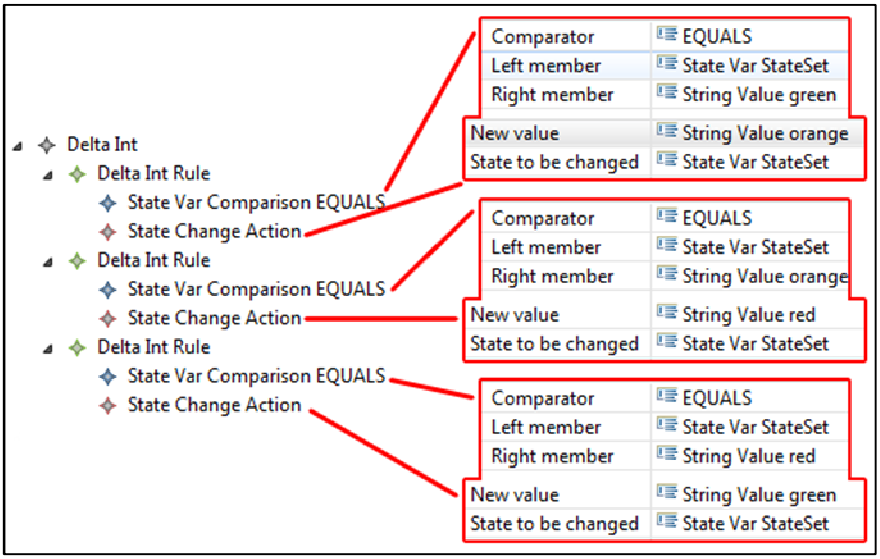


Figure V-7 : Résultat de la génération de la fonction DeltaInt

Examinons à présent les fonctions comportementales. La fonction de transition interne δ_{int} est montrée sur la capture d'écran de la Figure V-7.

Elle est composée de trois règles, et le contenu de chaque règle se trouve dans les blocs situés sur la moitié droite de la figure, ils sont liés par des traits aux règles auxquels ils se rattachent. Les trois premières lignes du sommet des blocs contiennent la définition d'une *StateVarComparison*, les deux lignes de la base contiennent la description d'une

StateChangeAction. Conformément à nos attentes, les *AutoTransition* du modèle source BasicDEVS ont été correctement transformées en *DeltaIntRule* MetaDEVS : lorsque la variable « StateSet » est dans l'état « green », elle doit (après expiration de la durée de vie de l'état) passer à l'état « orange », puis passer de « orange » à « red », pour enfin passer de « red » à « green ».

En respectant la même convention graphique, nous montrons dans la Figure V-8 la fonction de sortie *Lambda*. Elle se compose de trois règles *LambdaRule*, générées à partir de chaque *OutputEvent* déclenché par chaque *AutoTransition* du modèle de feu rouge BasicDEVS. Lorsque l'état « green » arrive à expiration, le modèle produira une sortie de type chaîne de caractère, sur le port « outputCrosswalk », signifiant qu'il passe du vert à l'orange, et ainsi de suite pour les deux autres changements d'états suivants.

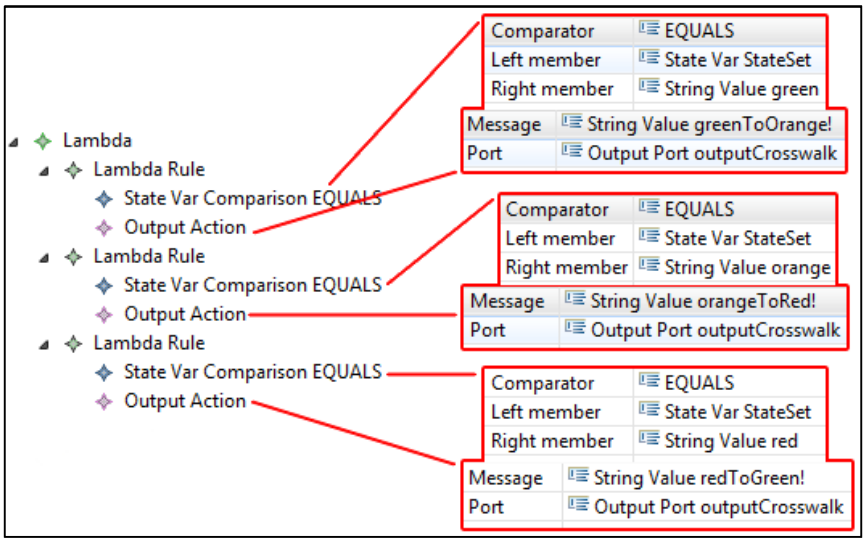


Figure V-8 : Résultat de la génération de la fonction Lambda

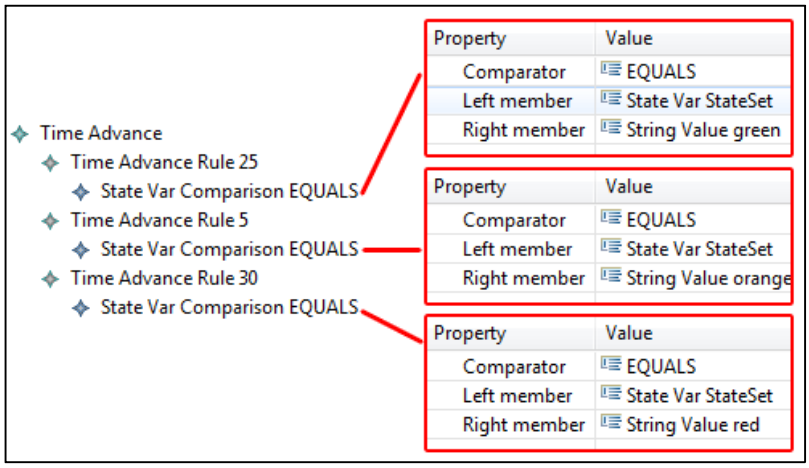


Figure V-9 : Résultat de la génération de la fonction TimeAdvance

Enfin, la Figure V-9 montre le résultat de la génération de la fonction *TimeAdvance*.

Elle est composée de trois *TimeAdvanceRule* : la première donne pour la valeur « green » de la variable d'état « StateSet » une durée de vie de 25 unités de temps, la seconde donne pour l'état « orange » une durée de vie de 5 unités de temps, et la dernière donne pour l'état « red » une durée de vie de 30 unités de temps. Le modèle que nous avons passé en

entrée étant un des plus simples qui soient, nous allons donc maintenant le compliquer légèrement en lui permettant de réagir à des évènements extérieurs.

c) *Amélioration du modèle et transformation*

Améliorons maintenant le modèle BasicDEVS précédent en lui donnant la possibilité de réagir aux évènements externes. Ceci passe par l'ajout d'un port d'entrée de type *InPort* et de transitions évènementielles de type *EventTransition*.

Si un piéton se trouvant à un carrefour appuie sur un bouton STOP, le feu tricolore devra, s'il se trouve dans l'état « green » ou « orange », effectuer une transition vers l'état « orange » à la suite de la réception du message « STOP ». En revanche, s'il se trouve dans l'état « red », il devra effectuer une transition vers ce même état. Le système doit ensuite continuer son évolution de manière autonome : rester 5 unités de temps dans l'état « orange » puis passer à l'état « red (s'il était dans l'état « green » ou « orange » au moment de la réception du message), ou bien rester dans l'état « red » pendant 30 unités de temps (s'il était dans l'état « red » au moment de la réception du message). Le piéton pourra ainsi traverser.

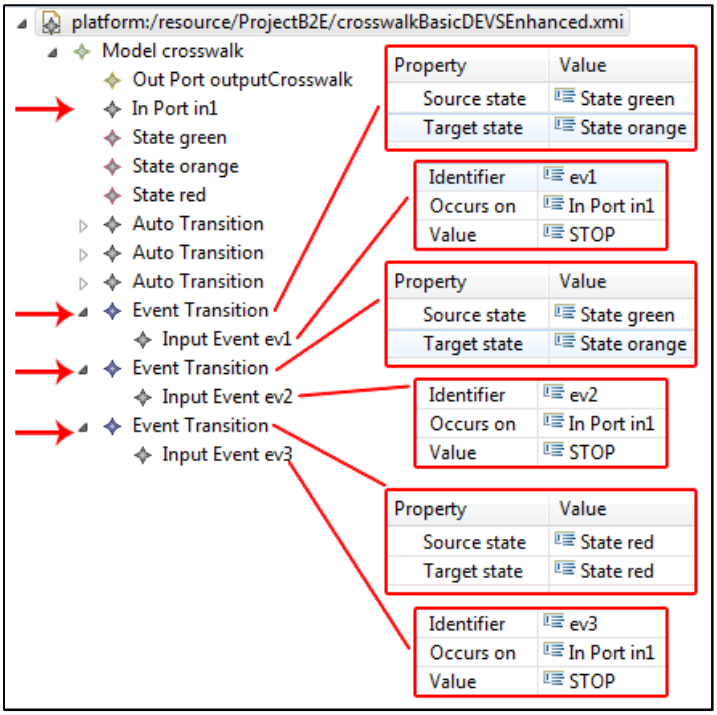


Figure V-10 : Modèle de feu tricolore amélioré

La Figure V-10 montre ce modèle de feu tricolore amélioré : les flèches sur la partie gauche de la figure montrent les éléments qui ont été rajoutés, à savoir un port d'entrée nommé « in1 » ainsi que trois transitions évènementielles *EventTransition*. La partie droite de la figure montre le détail de ces transitions.

Note : Lors de la transformation vers MetaDEVS, l'information portée par l'attribut « identifier » de l'*EventTransition* ne sera pas utilisée.

Encore une fois, nous appliquons la transformation BasicDEVSToDEVS à cette instance de BasicDEVS et, conformément à la définition de cette transformation, la fonction *DeltaExt* du modèle cible n'est désormais plus vide. Il y a eu également création d'une StringValue contenant la valeur textuelle « STOP » à la racine du modèle atomique cible.

La Figure V-11 montre le détail de la fonction DeltaExt générée. Sur le quart gauche de la figure se trouvent les trois règles créées, avec, pour chacune d'elle (sur la même ligne, en encadré) : la *StateVarComparison*, la *StateChangeAction*, et l'*InputPortComparison* (toujours la même, conformément au modèle BasicDEVS source) correspondantes.

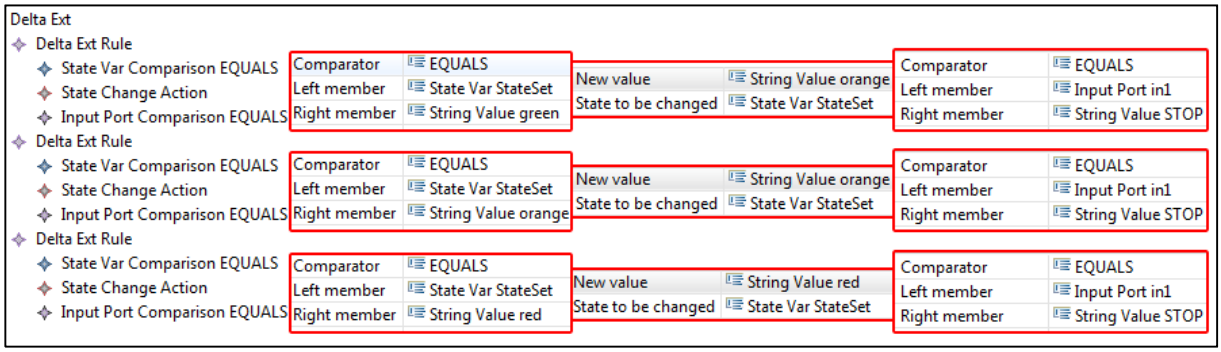


Figure V-11 : Détail de la fonction DeltaExt générée dans le modèle cible

5.4.4. Exemple de transformation n°2

a) *Modèle BasicDEVS de générateur de lettres*

Nous venons de voir que BasicDEVS pouvait être utilisé comme un générateur simple.

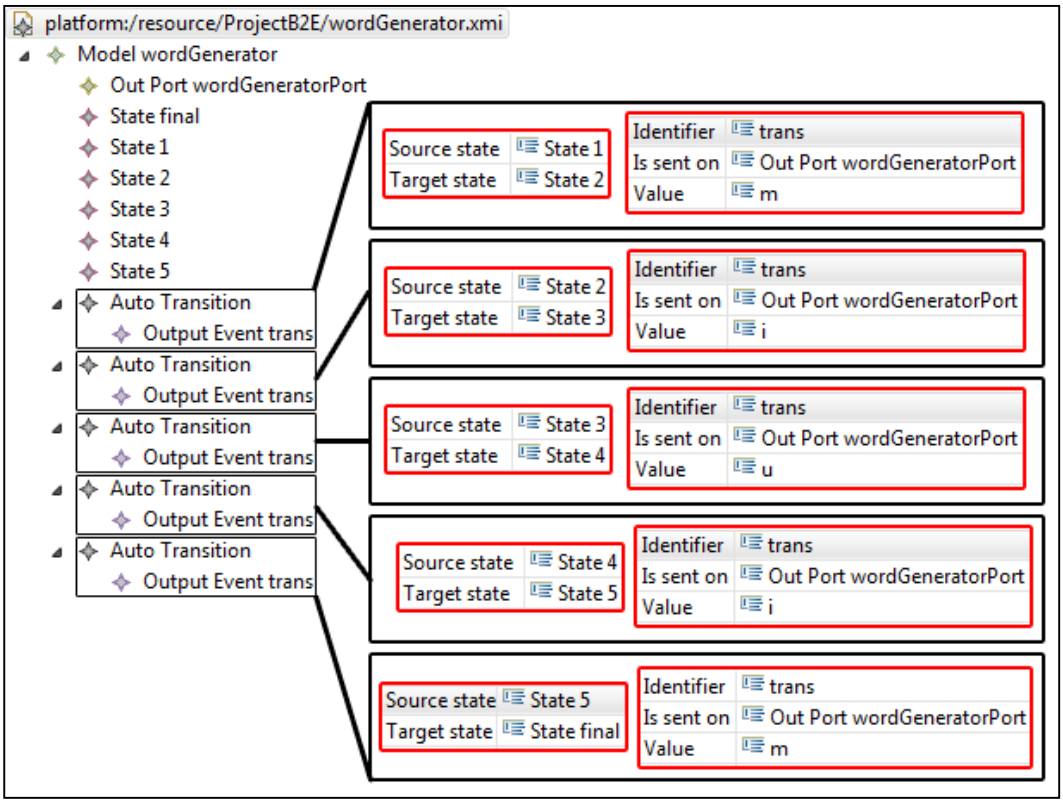


Figure V-12 : Modèle BasicDEVS de générateur de lettres

Voici un autre exemple d'utilisation : supposons que l'on désire créer un générateur de lettres, qui, à intervalles réguliers, envoie une lettre sur son port de sortie. Il est possible de créer simplement un tel générateur avec BasicDEVS. La capture d'écran partielle de la Figure V-12 montre un générateur de lettres, correspondant au mot « *miuim* ».

Il est composé de 6 états : les états numérotés de 1 à 5 ont une durée de vie de 1, l'état nommé « final » a une durée de vie de 9999999. Son état initial est 1. Il se compose également d'un port de sortie « wordGeneratorPort ». La figure montre aussi les transitions entre états et les sorties associées : à partir de l'état 1, le modèle évolue vers l'état 2 au bout d'une unité de temps, et envoie « m » sur le port de sortie. Il procède ainsi pour chaque lettre du mot « *miuim* » pour finir ensuite dans l'état « final » qui a une durée de vie assimilable à l'infini.

b) ***Transformation vers DEVS***

Ce type de modèle est similaire au modèle de feu tricolore présenté en 5.4.3.a). De plus, le fait qu'il possède 6 états implique que le modèle MetaDEVS équivalent possède autant de fonctions de transition interne, de fonctions de sortie et de fonctions d'avancement du temps. Pour ces raisons, nous choisissons de ne pas présenter ici le résultat de la transformation de ce générateur de lettres vers DEVS, car cela ne présente que peu d'intérêt. Nous emploierons ce modèle de générateur dans le prochain chapitre, au sein d'un modèle couplé.

5.5. Transformation M2M : d'un FSM vers DEVS

Nous nous proposons ici de montrer comment transformer un autre formalisme à états-transitions très populaire, les automates à états finis.

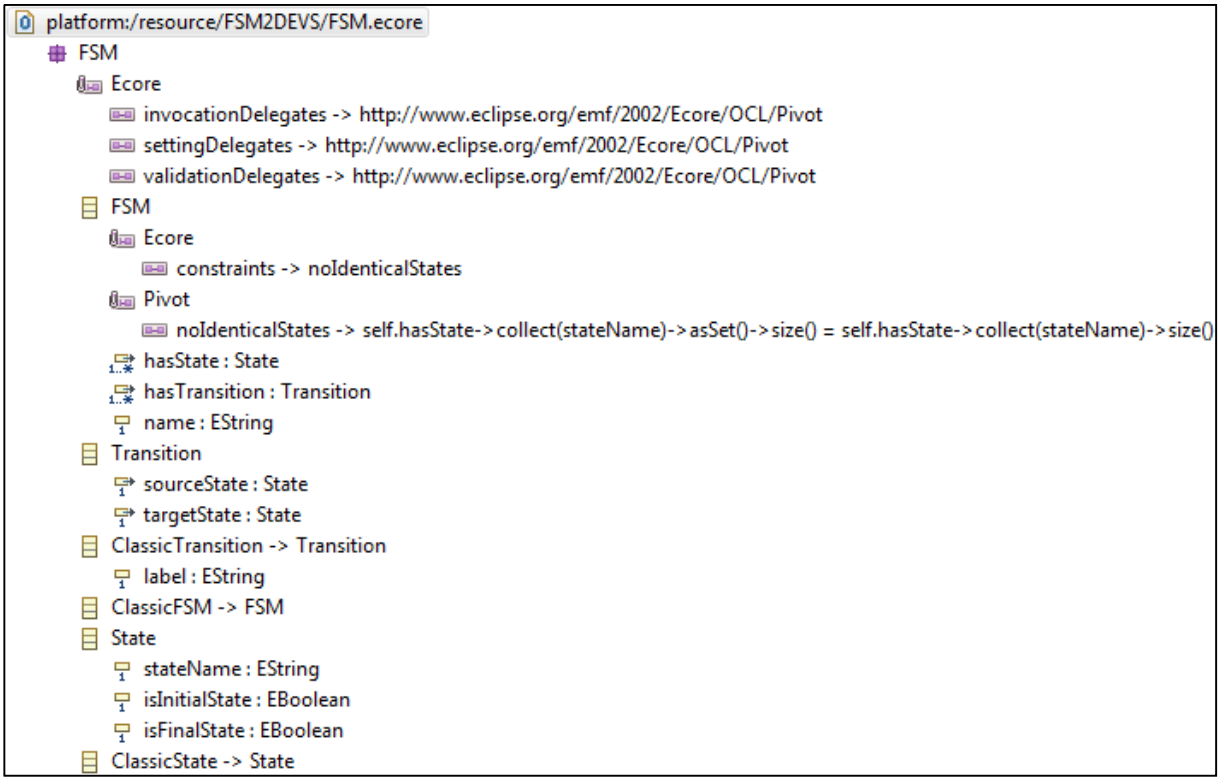


Figure V-13 : Méta-modèle Ecore des FSM vu dans l'éditeur

Bien que plus éloignés de DEVS que le formalisme que nous venons de voir, ces derniers appartiennent néanmoins à la famille F_2 et en particulier aux MOC. Nous présentons ici un méta-modèle possible des FSM, que nous avons élaboré grâce au méta-formalisme

Ecore de l'environnement EMF puis nous montrons comment on peut les transformer vers DEVS.

Ce formalisme simple gère uniquement des chaînes de caractères, que ça soit pour les étiquettes ou pour les états. Il n'est pas possible de spécifier une hiérarchie de modèles. À l'instar de la précédente, cette transformation est horizontale et exogène.

5.5.1. Présentation du méta-modèle FSM et contraintes associées

La Figure V-13 montre une capteure d'écran Ecore du méta-modèle FSM que nous avons défini.

Ce modèle d'automate doit comporter un seul état initial, et au moins un état final.

Deux états ne peuvent pas avoir le même libellé. Une transition se fait entre un et un seul état source et un et un seul état cible. Deux transitions ayant le même état source et le même état cible ne peuvent pas avoir le même label. Un état ne peut être initial et final en même temps.

a) *Comment créer un modèle FSM ?*

Nous proposons de suivre les quelques étapes suivantes pour y parvenir. Créer un FSM est très simple.

1. **Créer le modèle.** Il possède un nom.
2. **Créer les états du modèle.** Chaque état possède un identifiant sous forme de chaîne de caractères. Il peut être initial ou final. Par défaut, les valeurs des attributs *isInitialState* et *isFinalState* sont fixées à *false*.
3. **Créer transitions entre les états.** On doit créer les transitions qui se composent d'un état-source, d'un état-cible, et d'une étiquette, ou label. Ce label correspond par exemple à une lettre « lue » par l'automate.

5.5.2. Définition des règles génériques de transformation

a) *Discussion préalable*

À l'inverse de la transformation précédente, le formalisme source ne contient pas certains concepts qui sont pourtant essentiels à DEVS. Ceci rend la transformation moins évidente à appréhender. Le fait de connaître à la fois les FSM et DEVS (rôle du méta-modeleur) est ici très utile pour trouver un moyen de créer des règles qui combleront ce vide. Nous préconisons de raisonner comme suit.

1. **La première question à se poser est : qu'est-ce qui appartient à DEVS et qui ne figure pas dans le méta-modèle source ?** Un simple examen des deux formalismes nous permet de dire que les FSM ne connaissent ni la fonction de sortie, ni la fonction de transition externe, ni la fonction de transition interne, ni donc la fonction d'avancement du temps. Les ports ne sont donc pas connus non plus.
2. **Comment introduire ces concepts sans qu'ils faussent le modèle en entrée ?** Le fait que l'on soit dans la famille F_2 nous aide beaucoup. Car

même si à priori aucune des fonctions DEVS n'est présente dans le formalisme de base des FSM, ce formalisme se base quand même sur des états et des transitions, et c'est ce qui est le plus important.

3. **Comment doit se comporter le modèle DEVS résultant de la transformation ?** Il s'agit ici de réfléchir à l'exécution d'un automate à états finis lisant un mot en entrée, dans un contexte DEVS : le but est de déterminer quand un mot est reconnu ou pas.

Prenons tout d'abord le cas de l'absence de port d'entrée dans les FSM. Bien que ce port soit facultatif dans DEVS, on sait intuitivement qu'on en aura besoin : en effet, un automate « lit » des mots (composés de lettres), et c'est la lecture de ces lettres qui le fait passer d'un état à un autre. Il est donc clair que créer un port d'entrée sur le modèle DEVS correspondant à l'automate, qui récupèrera les lettres des mots lus en entrée, est indispensable. Nous avons donc besoin d'une règle qui crée d'office un port d'entrée, nous avons déjà présenté une règle similaire (générique) en 5.3.3.h).

Si on pousse un peu plus loin la réflexion, il apparaît immédiatement que le fait de lire une lettre sur le port d'entrée dans le but d'effectuer un changement d'état s'apparente à la définition d'une transition externe : la fonction DeltaExt du modèle de destination sera composée de règles, dans lesquelles les conditions testeront si le modèle est dans un état particulier, et si une valeur spécifique arrive sur le port d'entrée, et comme action un changement d'état. Tant que le modèle ne lit rien en entrée, il reste indéfiniment dans un état. L'évolution de ce modèle sera donc essentiellement conditionnée par les règles définies dans la fonction de transition externe, en fonction des lettre lues.

Il n'y a pas de transition interne dans les FSM de base, pas plus que d'états temporisés. Pourtant, selon les spécifications de MetaDEVS, *DeltaInt* ne doit pas être laissé vide. De plus, cette fonction est liée à la fonction de sortie, qui peut nous nous être utile dans certains cas. (*Note :* Rappelons toutefois qu'une fonction de sortie n'est pas obligatoire dans DEVS. Un modèle passif en est l'exemple parfait)

Il faut donc chercher quelles transitions particulières pourraient être le fait de *DeltaInt* sans que cela ne fausse pour autant l'automate décrit par le modèle source, et réfléchir sur la temporisation des états. Puisque les transitions dans les FSM sont uniquement le fait de mots lus, le fait d'assigner une durée de vie égale à l'infini aux états du FSM en entrée ne changerait rien : ceci est d'ailleurs couramment utilisé quand on veut modéliser un système qui ne réagit qu'aux évènements extérieurs. Admettons pour le moment que la durée de vie de tous les états du FSM soit infinie. Dans ce cas, une fois toutes les lettres passées en entrée lues, il n'y aurait aucun moyen de préciser que le mot a bien été reconnu.

Une solution possible consiste donc à utiliser la fonction de sortie comme un indicateur de reconnaissance du mot, par exemple en envoyant un message sur le port de sortie signifiant que le mot a été reconnu. Rappelons qu'un mot est reconnu par un automate si, à la fin de sa lecture, on se trouve dans un état final. De par la nature de DEVS, et de ses sémantiques d'exécution pendant la simulation, la fonction de sortie est toujours associée à la fonction de transition interne, laquelle ne se déclenche que lorsqu'un état arrive en fin de vie.

Nous pouvons donc, pour représenter le fait qu'un mot soit reconnu par l'automate, associer à chaque état final :

- une durée de vie inférieure à l'infini, mais supérieure ou égale à la fréquence d'arrivée des lettres sur le port d'entrée de l'automate
- une fonction de transition interne, qui par exemple fait « boucler » l'état sur lui-même, ou se calque sur la fonction de transition externe (même état source, même état cible). Nous choisissons cette dernière option. Cette fonction n'a d'importance que pour déclencher la fonction de sortie, donc pour matérialiser le fait que l'automate a reconnu le mot en entrée.
- une fonction de sortie, qui envoie, à expiration de la durée de vie de l'état, et avant le déclenchement de la fonction de transition interne, un message sur le port de sortie, disant que le mot a été reconnu

b) **Helpers *(fonctions auxiliaires)***

Encore une fois, nous nous servons de fonctions auxiliaires appelées *helpers* dont nous ne donnons ici que la signature. Elles sont au nombre de deux :

`isInitial()` : renvoie *true* si l'état *FSM* passé en paramètre possède son attribut *isInitial* à *true*, *false* sinon

`isFinal()` : renvoie *true* si l'état *FSM* passé en paramètre possède son attribut *isFinal* à *true*, *false* sinon

c) ***Vue d'ensemble de la transformation FSM vers DEVS***

Nous pouvons à présent proposer une vue d'ensemble pour cette transformation (Tableau V-7).

SOURCE	*DESTINATION*
FSM	AtomicDEVS
Liste de **State**	Plusieurs **StringValue** dans une seule StateVar (DEVSid="FSMState")
Liste de **State**	durée de TARule >"fréquence d'arrivée" si état final, égale à l'infini sinon
-	1 InputPort et 1 OutputPort
Transition	DeltaExtRule basée sur le nom de l'état et sur la lettre lue en entrée
Transition	Créer autant de LambdaRule qu'il y a de transitions vers des états finaux
-	Créer autant de DeltaIntRule qu'il existe d'étatsfinaux

Tableau V-7 : Vue d'ensemble de la transformation FSM vers DEVS

Elle constitue, tout comme la transformation précédente, une adaptation du Tableau V-2 aux spécificités d'une transformation des FSM vers DEVS. Elle est montrée dans le Tableau V-7. Certaines règles de cette transformation sont ensuite détaillées au moyen du pseudo-langage que nous avons proposé.

d) ***Déclaration de la transformation***

Cette transformation met en œuvre une instance de *FSM* que nous nommerons pour la suite *fsm*, et vise à créer une instance d'un *DEVSModel*.

```
transformation FSMToDEVS(in fsm:FSM, out dev:DEVS) {
    main() {
                [règles]
            }
}
```

e) ***Création du container***

Nous devons en premier lieu instancier l'objet qui contiendra tous les autres : ici, c'est tout simplement le modèle atomique AtomicDEVS. Il doit posséder le même nom que le modèle FSM.

```
rule instanciationAtomicDEVS()
   from fsm : FSM!ClassicFSM
    to dev : DEVS!AtomicDEVS
        dev.name=fsm.name
```

Nous considérons à présent l'instance *fsm* créée, et nous la renseignerons au fur et à mesure, parfois sans utiliser de **from**, ce qui correspond au fait qu'un élément cible n'a pas forcément d'antécédent dans le modèle source (par exemple, les types de base).

f) ***Types de base et fonctions DEVS***

Ici nous procédons de la même façon que nous l'avons fait pour la transformation de BasicDEVS vers DEVS en 5.4.2.e). Seule la cible change, il s'agit ici de l'instance *fsm*.

g) ***Créer des LitteralBasicValue***

```
rule collecteCreationLBV()
  from fsm : FSM!ClassicFSM
  to dev : DEVS!AtomicDEVS
[foreach état et entrée]
dev.handles=createLBV(valeur)
```

Il est nécessaire de créer, en outre, une LBV supplémentaire contenant le texte du message que nous désirons envoyer sur le port de sortie. Ceci se fait par un appel direct à la fonction `CreateLBV()`, qui est donc réutilisée telle quelle, quelle que soit la transformation :

```
dev.handles=createLBV ('__the word has been recognized__', sT)
```

h) ***Créer l'unique StateVar du modèle***

```
 rule createStateVar()
 from fsm : FSM!ClassicFSM
 to sv : DEVS!StateVar
 sv.DEVSid= 'FSMState'
sv.isAlwaysTyped=StringType
sv.initial_value=<LBV créée précédemment correspondant à l'état initial>
```

i) ***Créer les ports***

Nous forçons ici la création de deux ports : un port d'entrée, un port de sortie.

```
iPort : ExtendedDEVS!InputPort (
  portID <- 'FSMReadInputPort'
  is_always_typed <- sT
  )
```

```
    oPort : ExtendedDEVS!OutputPort (
        portID <-'FSMReadOutputPort'
        is_always_typed <- sT
                    )
```

j) ***Les fonctions atomiques***

Encore une fois, les règles correspondant aux fonctions atomiques font appel aux règles réutilisables que nous avons énoncées en 5.3.4.c) et 5.3.4.d) (*Condition* et *Action*).

Comme nous en avons discuté en 5.5.2.a), nous créons pour chaque transition spécifiée dans le modèle FSM une règle de type *DeltaExtRule*. Ces règles seront placées dans la fonction de transition externe.

```
rule createDeltaExtRule(t: FSM!Transition, sv: DEVS!StateVar)
//t est une Transition
      to sv : DEVS!DeltaExtRule
                  tests <- createSVC(t.sourceState,sv)
                  tests_input_event <- createIPC(t)
                  changes_state <- createSCA(t,sv)
```

Il faut ensuite donner une durée de vie infinie aux états non finaux de l'automate, et une durée de vie finie aux états finaux : par défaut, 4000 unités de temps. Nous employons pour cela les *helpers* définis plus haut, que nous utilisons pour modifier la règle de création des *TimeAdvanceRule* présentée en 5.3.4.e). La fonction suivante est appelée sur chaque état du *FSM* source.

```
rule createTARule (s: FSM!State, sv: ExtendedDEVS!StateVar)
     to tarule: ExtendedDEVS!TimeAdvanceRule (     )
            tarule.tests<-thisModule.createSVC(s, sv);
            if (s.isFinal()=true) {
            tarule.ta_value<-4000;
            }
            else {
            tarule.ta_value<-9999999;
            }
            tarule;
```

Pour la fonction de transition interne, nous choisissons comme nous l'avons expliqué dans la discussion ci-dessus de la remplir avec des règles portant sur les états finaux (les seuls qui ont une durée de vie finie). L'état cible doit être final. En fait, ces règles seront donc identiques, en termes d'états sources et cibles, aux règles *DeltaExtRule* qui possèdent un état final comme cible. Cette fois-ci, nous ne modifions pas le code de la règle *DeltaIntRule* comme nous l'avons fait pour `createTARule()` car cela n'est pas nécessaire. Il suffit simplement de collecter les états de l'automate, et de n'appeler cette règle que sur les états cibles qui sont finaux, au moyen du *helper* `isFinal()`.

```
rule createDeltaIntRule(t: FSM!Transition, sv: DEVS!StateVar)
//t est une Transition dont l'état cible est un état final
      to sv : DEVS!DeltaIntRule
                  tests <- createSVC(t.sourceState,sv)
                  changes_state <- createSCA(t,sv)
```

Enfin, nous devons décrire la règle qui remplira la fonction de sortie avec des *LambdaRule*. Tout comme nous venons de le faire pour la fonction de transition interne, nous devons collecter les états de l'automate, et de n'appliquer cette règle qu'aux seuls états cibles qui sont finaux. Ceci se fait une fois de plus en employant le *helper* `isFinal()`.

```
rule createLambdaRule (tr:FSM!Transition, sv: ExtendedDEVS!StateVar, m:)
     to
      lambdarule: ExtendedDEVS!LambdaRule
            tests <- createSVC(tr.sourceState, sv, m),
            sends_message <- createOutputAction(tr,m)
```

5.5.3. Exemple de transformation

a) *Création d'un modèle FSM*

Créons à présent un modèle de FSM, conforme au méta-modèle présenté en 5.5.1. Nous choisissons de décrire l'automate présenté dans le chapitre I de ce document : l'automate que nous avions baptisé « MIU ». Pour mémoire, cet automate reconnaît l'alphabet { *m*, *i*, *u* } et se compose de trois états, le premier état est initial, le troisième est final.

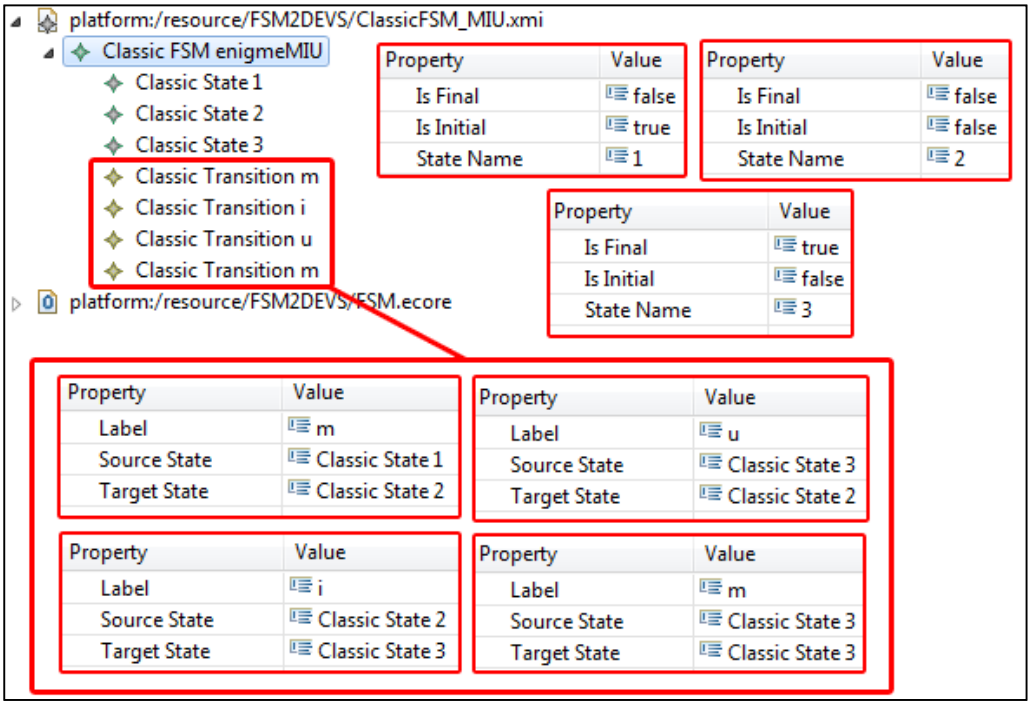


Figure V-14 : Capture d'écran reconstituée d'un modèle de FSM

La Figure V-14 représente une capture d'écran (sous Eclipse) d'une instance dynamique de l'automate ci-dessus. L'automate possède un nom : « enigmeMIU ». La partie supérieure droite de la figure montre les attributs des trois états qui composent cet automate : on y voit comment les attributs booléens *isInitial()* et *isFinal()* permettent de caractériser simplement un état (état 1 initial, état 3 final). L'encadré situé dans la moitié inférieure de cette figure montre les propriétés des quatre transitions entre états qui décrivent le comportement de cet automate.

b) *Transformation vers MetaDEVS*

Comme nous l'avons fait en 5.4.3 et 5.4.4 pour transformer des modèles BasicDEVS en modèles MetaDEVS, nous avons ici aussi implémenté en ATL les règles génériques

énoncées en 5.5.2. La transformation est ici aussi entièrement automatique, et une fois cette dernière exécutée sur l'instance présentée sur la Figure V-14, nous obtenons un modèle MetaDEVS qui est présenté dans sa globalité sur la Figure V-15.

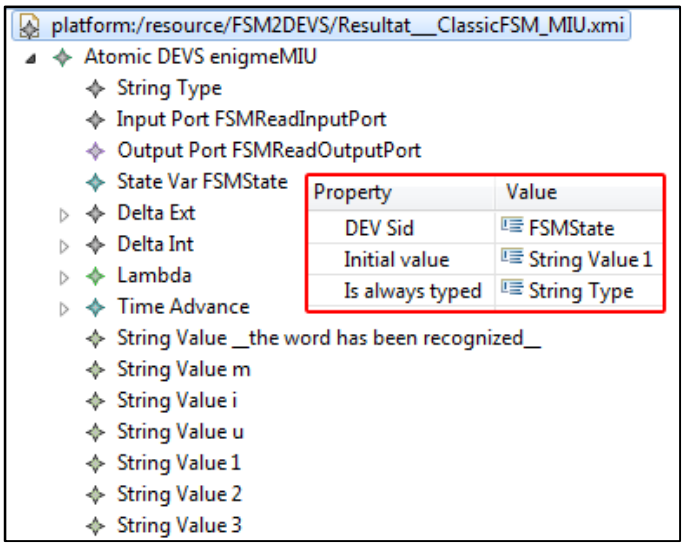


Figure V-15 : Capture d'écran globale du modèle d'automate généré par FSMToDEVS

Cette figure montre que le modèle résultant de la transformation est composé d'un unique modèle atomique MetaDEVS se nommant « enigmeMIU », tout comme le modèle d'automate FSM passé en entrée. Ce modèle atomique contient :

- Un singleton *StringType* qui est utilisé pour typer chaque *DEVSXpression* ;
- Un port d'entrée « *FSMReadInputPort* » et un port de sortie « *FSMReadOutputPort* » qui ont été instanciés d'office par la transformation (le méta-modèle FSM ne connaissant pas la notion de port) ;
- Les quatre fonctions comportementales MetaDEVS : DeltaInt, DeltaExt, Lambda, TimeAdvance. Ces fonctions sont détaillées dans les captures d'écran suivantes ;
- Des *LitteralBasicValue* de type chaînes de caractères : trois d'entre elles contiennent les numéros des états du système, trois autres les lettres qui sont manipulées par l'automate, et une dernière contient la chaîne « __the word has been recognized__ », elle sera envoyée sur le port de sortie en cas de succès lors de la lecture du mot ;
- Une variable d'état, nommée par défaut « FSMState », dont les propriétés figurent en encadré : une valeur initiale à « 1 », récupérée sur le modèle initial (l'attribut « isInitial » de l'état 1 de l'automate contenait la valeur *true*), et un typage *StringType*.

Examinons à présent les fonctions comportementales. La fonction *ta* est montrée sur la Figure V-16. Elle se compose de trois règles *TimeAdvanceRule* dont les deux premières (les règles correspondant aux états 1 et 2) ont une durée de vie assimilable à l'infini et la dernière (la règle correspondant à l'état final de l'automate) une durée de vie de 4000 unités de temps. Ceci est donc conforme aux règles énoncées dans la définition de la transformation FSMToDEVS, que nous avons créées à partir du raisonnement tenu en 1735.5.2.a).

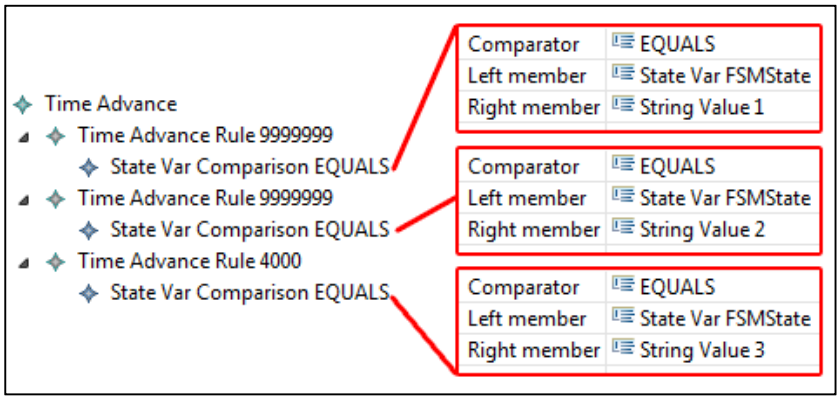


Figure V-16 : Résultat de la génération de la fonction TimeAdvance

La fonction Lambda est illustrée sur la capture d'écran de la Figure V-17. Elle est composée de deux règles, et en face de chacune d'entre elles figure la *StateVarComparison* et l'*OutputAction* qui lui sont associées. Toujours en accord avec notre définition de transformation, nous pouvons voir que le modèle cible a bien créé deux fonctions de sortie qui seront déclenchées soit lors d'un passage de l'état 2 vers l'état 3, soit lors d'un passage de l'état 3 vers l'état 3 (voir fonction de transition interne ci-après) : le message de type *StringValue* « __the word has been recognized__ » sera alors envoyé sur le port de sortie « *FSMReadOutputPort* » du modèle atomique.

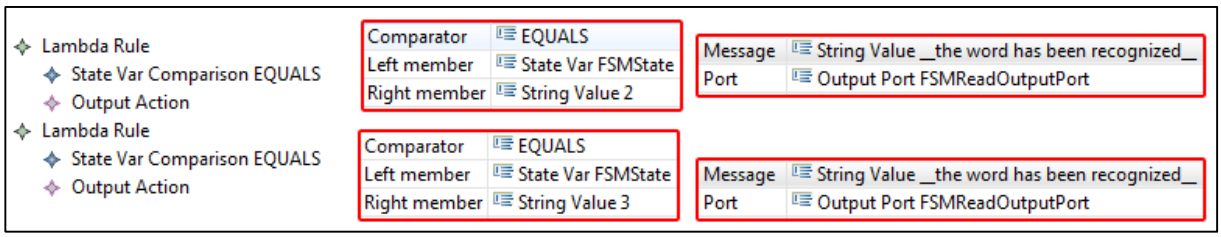


Figure V-17 : Résultat de la génération de la fonction Lambda

La fonction DeltaInt est présentée sur la capture d'écran de la Figure V-18.

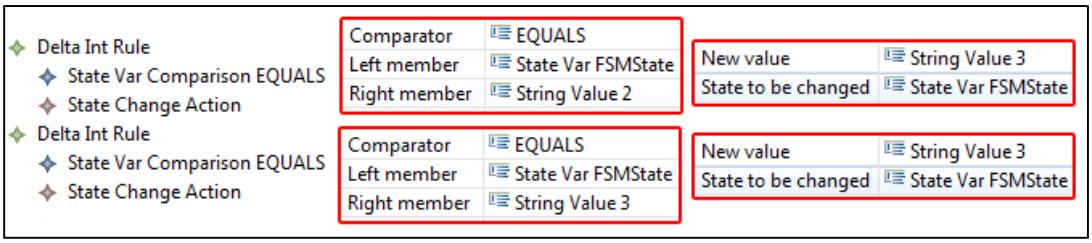


Figure V-18 : Résultat de la génération de la fonction DeltaInt

Elle ne concerne que les états source finaux, conformément encore une fois à notre réflexion sur le déroulement de la simulation, qui a abouti à la définition des règles FSMToDEVS. Les états destination sont quant à eux calqués sur ceux décrits par les transitions du FSM source (de l'état 2 à 'létat 3, de l'état 3 à l'état 3).

Enfin, la fonction de transition externe est montrée sur la capture d'écran reconstituée de la Figure V-19. Elle se compose de quatre DeltaExtRule, qui ont été générées à partir des quatre transitions contenues dans le modèle source FSM. La partie droite de la figure illustre le contenu de chaque DeltaExtRule dans des rectangles. Ils sont composés, en haut à gauche, des propriétés de la StateVarComparison, puis, en bas à gauche, des propriétés de

l'InputPortComparison, et enfin, sur la droite, de la StateChangeAction. La fonction DeltaExt générée est, elle aussi, en tous points conforme au modèle source et aux règles de transformation que nous avons définies.

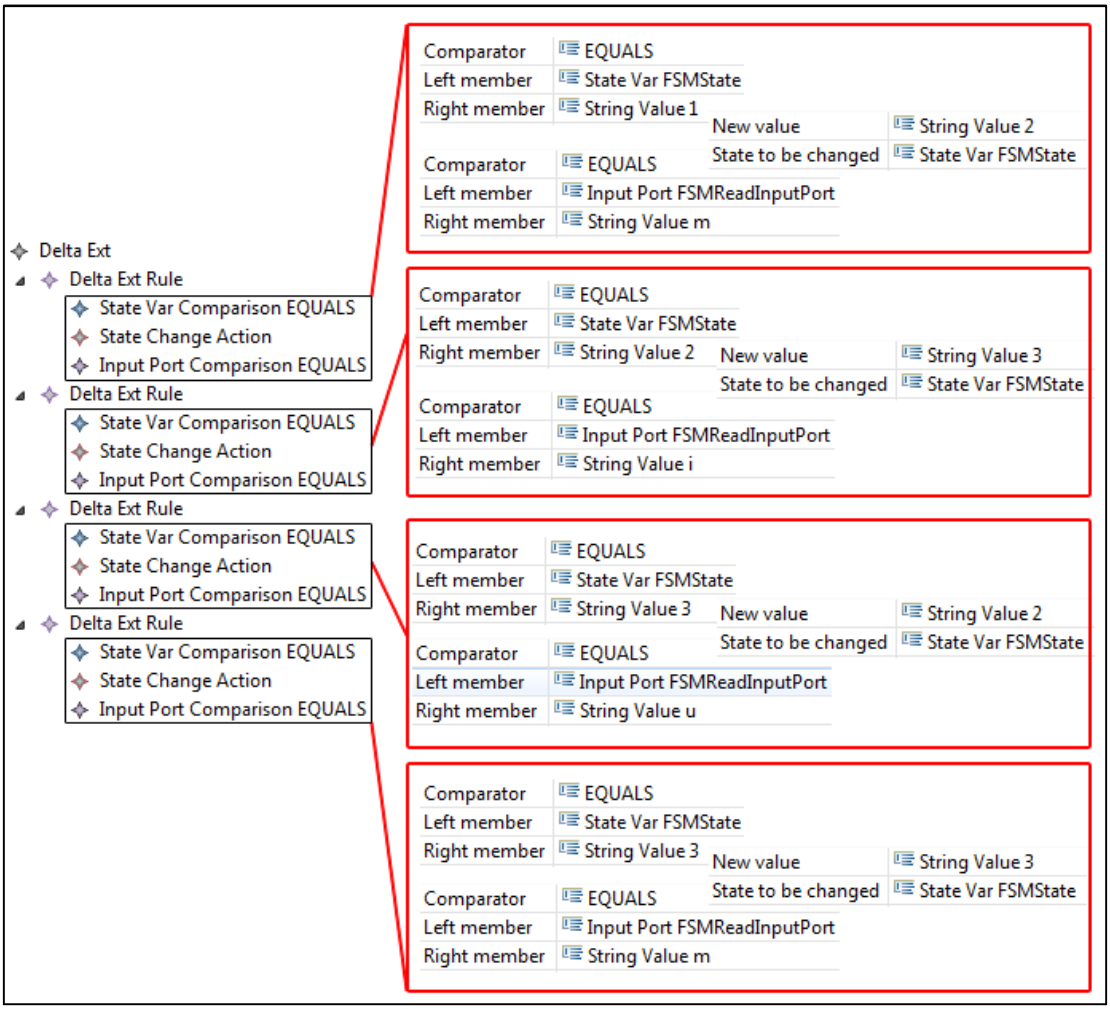


Figure V-19 : Résultat de la génération de la fonction DeltaExt

Conclusion du chapitre

Nous avons, dans ce chapitre, identifié en premier lieu quelles pouvaient être les familles de formalismes candidats à une transformation vers DEVS, et avons établi des règles génériques pour décrire ces transformations. La description de ces règles s'est faite au moyen d'un pseudo langage proche de QVT que nous avons défini pour les besoins de nos travaux.

Nous avons ensuite appliqué ces règles génériques à deux exemples : un langage créé de toutes pièces permettant de spécifier des modèles équivalents à des modèles DEVS atomiques, aux états unidimensionnels, et un langage permettant de décrire des automates à états finis, dont nous avons conçu le méta-modèle de la même manière que nous l'avons fait pour le méta-modèle de DEVS. Notre but était de transformer ces deux langages vers MetaDEVS, dans une optique de générer ensuite du code de simulation DEVS .

Nous avons en outre privilégié la réutilisabilité des règles : par exemple, les règles concernant la création des StateVarComparison, des StateChangeAction, des InputPortComparison et des OutputAction sont les mêmes dans la transformation

BasicDEVSToDEVS que dans la transformation FSMToDEVS. Ce sont seulement les règles appelantes qui changent.

Ces règles génériques ont enfin été implémentées concrètement dans l'environnement Eclipse au moyen du langage de transformation hybride ATL. Nous avons programmé une définition de transformation entre les FSM et MetaDEVS, et entre BasicDEVS et DEVS. Ces implémentations restent très proches, syntaxiquement et sémantiquement, des règles génériques que nous avons établies. Ces transformations vers MetaDEVS sont de surcroît totalement automatiques : à chaque fois que l'on exécute l'une d'entre elles, elle dispose à ce moment-là de tous les paramètres nécessaires à son bon déroulement. La seule condition nécessaire est que le modèle source soit conforme à son méta-modèle (i.e. en respecte les cardinalités, les contraintes…).

La modularité de MetaDEVS, que nous évoquions dans le chapitre précédent, entre désormais en jeu : si on choisit d'étendre ce méta-modèle, en y ajoutant des méta-classes héritant des méta-classes existantes par exemple, cela ne remettrait aucunement en cause les transformations M2M que nous avons implémentées. Par conséquent, aucune règle ne devrait être changée, puisque ces changements ne concernent que l'ajout de nouvelles classes, et ne modifient en rien l'architecture de notre méta-modèle MetaDEVS et laissent inchangée la composition des règles de transformations écrites précédemment.

Nous sommes maintenant en mesure d'aborder la dernière partie de notre travail, consacrée à la génération automatique de code objet à partir de modèles DEVS se conformant à MetaDEVS (et ce, indifféremment du fait qu'ils aient été créés en tant que modèles DEVS, ou créés en tant qu'autres modèles, FSM par exemple, transformés ultérieurement en modèles DEVS) : nous illustrons cet exemple en choisissant PyDEVS comme plateforme de destination.

Chapitre VI. TRANSFORMATIONS M2T : PIM METADEVS VERS CODE

À PRESENT, nous arrivons à l'aboutissement concret de notre démarche guidée par les modèles et orientée IDM/MDA : l'obtention d'un code simulable pour nos modèles MetaDEVS, par génération automatique. C'est ici que l'une des devises les plus populaires de l'IDM prend tout son sens « *Model once, generate everywhere* » (« Modéliser une fois, générer partout »).

Chaque modèle DEVS se conformant à MetaDEVS est par définition un *Platform Independent Model*, un PIM. Notre but est de transformer ce PIM en un voire plusieurs modèles de codes, qui pourront être directement simulés sur leurs plateformes respectives. Nous faisons appel pour cela à des transformations de type M2T que nous appliquons à des modèles DEVS afin de générer du code compatible avec le simulateur PyDEVS [Bolduc et al. 2001].

Note : Il est important de rappeler que notre approche se basant exclusivement sur la modélisation et la méta-modélisation, nous n'apportons à aucun moment de modification aux simulateurs DEVS, que ce soit à niveau des algorithmes des simulateurs abstraits ou du code DEVS.

Une transformation M2T en vue d'obtenir du code requiert, comme toute transformation de modèles :

- un méta-modèle cible ;
- un méta-modèle source ;
- des règles de transformation pour générer un modèle cible à partir du modèle source

Pour tester cette transformation, il est de surcroît nécessaire de disposer d'un modèle source, se conformant au méta-modèle source.

Nous avons vu dans le chapitre II que, si ceci était en théorie valable pour toutes les transformations de modèles, il est très courant dans la pratique de ne pas utiliser de méta-modèle cible lors d'une transformation M2T. En théorie également, selon l'approche MDA, le passage d'un PIM au code se fait via un PSM. Mais en pratique, on confond souvent la notion de PSM et de code : on passe alors d'un PIM au code, directement.

Même si les transformations M2T sont utilisées la plupart du temps pour générer du code, nous verrons dans ce chapitre qu'il est possible de générer d'autres artefacts textuels, par exemple un fichier .html de documentation sur les modèles. C'est en l'occurrence par ces fichiers de documentation que nous abordons les transformations de modèles DEVS vers du texte, car cet exemple est plus simple que la génération de code et permet d'illustrer les potentialités du M2T avec une approche de transformation par template.

Pour toutes les transformations de ce chapitre, nous utilisons le plugin Acceleo au sein de l'environnement Eclipse, qui, à l'instar d'autres langages de transformation M2T, permet à

partir d'un modèle typé par son méta-modèle (dans notre cas, un fichier XMI typé par un méta-modèle Ecore représentant MetaDEVS) de produire du code objet, via des templates de code. L'utilisation de cet outil se fait conjointement avec l'emploi du langage de description de contraintes OCL, standardisé par l'OMG. Nous réutilisons également des éléments du pseudo-langage introduit dans le chapitre précédent dans le cadre de nos transformations M2M.

Nous montrons que suivre une approche M2T par template revient à utiliser une combinaison de deux types d'expressions : les expressions statiques, qui sont en fait des chaînes de caractères qui ne seront jamais manipulées, et les expressions du langage de transformation (qui elles-mêmes peuvent contenir d'autres expressions du langage, mêlées à des expressions statiques), qui vont utiliser des éléments du modèle source pour calculer le texte qui sera produit en sortie, dans le fichier cible. Ces deux types d'expressions viennent préciser nos propos tenus dans le chapitre II, lorsque nous évoquions un « squelette » de texte composé de parties fixes et variables. Il est très courant de combiner, au sein même d'une expression du langage, des expressions statiques et d'autres expressions du langage, par exemple pour générer du code correspondant à une succession de structures de contrôle, avec des mots-clefs « fixes » (de type *if*) et des variables provenant du modèle.

Il peut être aussi utile, dans la pratique, d'utiliser des « patrons de code », c'est-à-dire des squelettes décrivant la forme finale d'un fichier de code (dans notre cas un fichier PyDEVS) parfois fournis par les équipes qui ont implémenté les simulateurs DEVS.

Pour chaque exemple de génération de code, nous raisonnerons autant que possible de manière générique, en montrant comment récupérer les éléments qui nous intéressent dans le modèle source, puis en appliquant de manière concrète les templates, c'est-à-dire en implémentant cette récupération, et en la combinant avec du code Python.

Nous donnerons quand cela est nécessaire un aperçu de nos templates grâce à des captures d'écran de fichiers Acceleo (fichiers de type *.mtl*), contenant les définitions des transformations. Nous choisissons d'utiliser ces captures d'écran dans le but de bénéficier de la coloration syntaxique du langage, ce qui améliore considérablement la lecture et la compréhension des transformations. Le texte de couleur noire correspond à des parties de code (HTML, Python…) correspondant à des commentaires, des mots-clefs du langage…etc. qui sont générées une seule fois. Si ce texte se trouve à l'intérieur d'un template, et si ce template fait appel à un mécanisme itératif, il sera généré dans le fichier cible autant de fois que le template (ou le mécanisme itératif qu'il contient) est exécuté.

La première section de ce chapitre illustre les fonctionnalités de l'approche M2T à travers un exemple simple de génération d'artefact : un fichier HTML décrivant la structure du modèle DEVS passé en entrée. Elle est conclue par un petit exemple.

La seconde section est quant à elle dédiée à la définition de règles de transformation de modèles DEVS, conformes à notre méta-modèle MetaDEVS, en code Python, respectant les exigences du simulateur PyDEVS et donc directement simulables. Nous structurons cette description en nous attachant d'abord à décrire la création d'éléments communs aux modèles atomiques et couplés, nous montrons ensuite comment reproduire la structure d'un modèle DEVS couplé, et enfin comment définir le comportement de nos modèles atomiques. Nous concluons cette partie par trois exemples complets de génération de code : nous montrons

ainsi que l'approche MetaDEVS permet de simuler toutes les combinaisons des types de modèles suivants :

- modèles « purement DEVS » (c'est-à-dire créés selon le méta-modèle MetaDEVS) ;
- soit des modèles qui à l'origine étaient des modèles non-DEVS, mais qui grâce à une approche M2M semblable à celle présentée dans le chapitre V, sont devenus des modèles MetaDEVS.

Par exemple, notre exemple d'un modèle couplé FSM+générateur de lettres contient deux modèles provenant de deux formalismes différents (FSM et BasicDEVS), tous deux non-DEVS à l'origine, mais transformés vers MetaDEVS (chapitre V).

6.1. Générer la documentation d'un modèle : « DEVS2HTMLDoc »

Les modèles DEVS que nous avons créés l'ont été dans l'éditeur de modèles EMF : cela nous permet donc de disposer de modèles conformes au méta-modèle MetaDEVS que nous avons proposé dans le chapitre 4 et stockés sous forme de fichiers XMI, format adapté à la sérialisation et l'échange de modèles. Il peut être utile dans un premier temps de pouvoir extraire certaines informations textuelles de ces modèles, notamment des informations concernant leur structure. Le but de cette partie est d'appréhender les transformations M2T à travers l'exemple d'une génération de fichier de documentation au format HTML. Dans l'annexe 8 figure le code utilisé.

6.1.1. Identifications des besoins

a) *Besoins généraux*

Pour faciliter la lecture de la structure d'un modèle DEVS, nous avons identifié un certain nombre de besoins :

- Mettre en évidence les relations contenant-contenu
- Identifier rapidement les noms de modèles par rapport au reste du texte
- Nommer le fichier contenant les informations d'après le nom du modèle couplé contenant tous les autres modèles, ou de l'unique modèle atomique qui constitue le modèle
- Distinguer les modèles atomiques des modèles couplés
- Utiliser un format lisible sur toutes les plateformes et offrant des possibilités au niveau de la mise en forme : le format HTML nous paraît adapté, d'autant plus qu'il est lisible sur n'importe quel type de navigateur Web

b) *Besoins spécifiques au fichier HTML*

Soit le modèle couplé de la figure Figure I-9. Au vu des besoins énoncés en a), nous voudrions obtenir un fichier de documentation de ce modèle qui, affiché dans un navigateur Web, ressemblerait à ce qui se trouve sur la Figure VI-1.

Bien entendu, il existe une infinie combinaison de possibilités, tant au niveau de la mise en forme du document que des renseignements qui y sont fournis. Nous nous limiterons à cet exemple très simple afin de montrer quels raisonnements employer pour manipuler les modèles DEVS et en extraire des informations textuelles basiques, que nous mettons en forme ensuite.

Ce fichier de documentation se nomme **Coupled 1_Doc.html**

Le modèle DEVS est **couplé**.

Modèle racine : ***Coupled 1***

Coupled 1 contient **2** sous-modèles :

Atomic 5 ***Coupled 2***

Coupled 2 contient **2** sous-modèles:

Atomic 2 Atomic 3

Figure VI-1 : Exemple de patron de documentation d'un modèle DEVS

6.1.2. Etapes pour la génération du fichier : raisonnement général

a) *Création et nommage du fichier*

La première étape consiste à définir les conditions de création du fichier, notamment en choisissant son nom son type. Le fichier doit être nommé selon le nom du modèle. On peut éventuellement y concaténer la chaîne de caractères '_Doc'. L'extension du fichier doit être également rajoutée manuellement, car il ne faut pas oublier qu'à la base on génère du texte brut, donc le fichier sera composé du nom du modèle (ou du nom du modèle de plus haut niveau dans le cas d'un modèle couplé, c'est-à-dire le nom du premier modèle trouvé), auquel on concatène la chaîne '_Doc.html' :

```
aDEVSModel.name.concat('_Doc.html')
```

(avec 'aDEVSModel' désignant une instance de *DEVSModel*)

b) *Squelette*

Il se compose de parties qui seront écrites une seule fois pendant la génération, incluant des balises HTML et du texte brut et de parties qui seront écrites à plusieurs reprises, en fonction du parcours du modèle :

```
<html>
      « Ce fichier de documentation se nomme »
      « Le modèle DEVS est »
      « … contient … sous-modèles : »
</html>
```

c) *Identification du modèle*

Il est nécessaire d'identifier le type de modèle DEVS dont on veut générer la documentation : atomique, ou couplé, ou plus précisément *AtomicDEVS* ou *CoupledDEVS*. Dans notre exemple, la génération de la documentation correspondant à un modèle atomique n'aurait pas un grand intérêt, à part donner le nom du modèle, mais il faut néanmoins tenir compte de ce cas.

Pour effectuer un tel test, nous utilisons le langage OCL avec une expression de type :

```
aDEVSModel.oclIsTypeOf(AtomicDEVS)
```

qui renverra `'true'` si le modèle testé est de type *AtomicDEVS* `'false'` sinon.

d) ***Affichage des sous-modèles d'un modèle couplé***

Comme montré dans l'exemple en b), l'intérêt de cette génération de texte réside dans la représentation de la structure d'un modèle couplé. Pour ce faire, il faut en premier lieu trouver un moyen d'afficher les noms des sous-modèles contenus dans un modèle couplé. L'expression suivante constitue l'équivalent implicite d'une boucle itérative qui, à partir de la liste de sous-modèles (un « Set » en termes de langage OCL) affiche le nom de chacun d'eux.

```
aDEVSModel.contains.name
```

Toujours pour rester dans l'esprit de l'exemple, il faut veiller à ce que les modèles atomiques soient affichés avant les modèles couplés. Pour ce faire, il existe plusieurs solutions possibles : nous choisissons de réutiliser `oclIsTypeOf()` utilisé ci-dessus. L'idée est ici de tester le type de chaque modèle, le résultat renvoyé sera une liste de modèles à trier selon l'ordre alphabétique du résultat de l'évaluation de `oclIsTypeOf()`. En d'autres termes, si l'on teste les sous-modèles selon le critère *AtomicDEVS*, puis que l'on trie par ordre alphabétique le résultat, la liste sera remaniée et les modèles couplés apparaîtront avant les modèles atomiques. En effet, si on applique `oclIsTypeOf(AtomicDEVS)` à la liste suivante : {Coupled1, Atomic2, Coupled2}, on obtiendra {*false*, *true*, *false*}. Un tri alphabétique donnera {*false*, *false*, *true*} et la liste finale sera donc {Coupled1, Coupled2, Atomic2}. En testant par rapport à `CoupledDEVS` et non par rapport à `AtomicDEVS` on obtiendra donc une liste de sous-modèles triée, composée des modèles atomiques en premier, suivis des modèles couplés. L'expression suivante permet un tel résultat :

```
afficher_sous_modèles(CoupledDEVS) :
aDEVSCoupled.contains->sortedBy(oclIsTypeOf(CoupledDEVS)).name
```

e) ***Parcours d'un modèle couplé***

L'idée de base de l'algorithme à mettre en œuvre est la suivante : on réutilise l'affichage des sous modèles vus ci-dessus, et on l'utilise conjointement avec une itération qui concerne uniquement les sous-modèles couplés. On utilise la récursivité.

```
parcours (CoupledDEVS) :
afficher_sous_modèles() ;
foreach (sous_modèle_couplé)
      sous_modèle_couplé.parcours() ;
```

Cela permet à chaque fois d'afficher d'une part tous les sous-modèles, puis de rappeler `parcours()` sur chaque sous-modèle couplé. Deux solutions s'offrent à nous pour l'implémentation de *foreach* : soit profiter de la structure itérative [for] présente dans Acceleo, et parcourir tous les sous-modèles en testant chacun d'entre eux pour savoir s'il est couplé ou non, ou bien utiliser de nouveau le langage OCL.

C'est ce choix-là que nous avons fait : l'idée n'est pas de tester un à un les sous-modèles pour voir s'ils sont atomiques ou couplés, ni ne sélectionner que les sous-modèles couplés, mais de « surclasser » la fonction `parcours()` ce qui permettra de l'appeler sur

n'importe quel modèle *DEVSModel*, mais qui ne produira de résultat que si on est présence d'un modèle *CoupledDEVS*. Surclasser cette fonction est très simple, il suffit de la déclarer avec le même nom et la même signature, et en paramètre il faut préciser la classe abstraite *DEVSModel*. En M2M avec Acceleo, cela revient à créer un héritage de templates (voir 6.1.3.a)) pouvant se baser sur les éventuels héritages du méta-modèle, comme dans le cas présent. Ceci se fait de la manière suivante :

```
parcours (DEVSModel)
```

Le nombre de sous-modèles d'un modèle couplé s'obtient en appelant l'opération OCL `size()` sur la relation *contains* du modèle couplé.

6.1.3. Implémentation à l'aide de Accceleo

Ce que nous avons jusqu'à présent appelé « fonction » désigne en fait l'élément central d'Acceleo : le template. Les templates sont très utilisés en M2T, notamment pour leur grande flexibilité et le fait qu'ils supportent les mécanismes d'héritage. Nous montrons dans cette sous-section comment créer des templates simples à partir de la réflexion que nous avons menée précédemment, au sein d'un fichier de définition de transformation Acceleo (fichier de type *.mtl*). Ce fichier est utilisé au sein de l'environnement Eclipse.

a) *Créer du texte mis en forme : héritage de templates*

Le langage Acceleo offre la possibilité de redéfinir la méthode, ou plutôt le template, `toString()`, et ce pour n'importe quel objet. Par exemple, le template suivant

```
[template public toString(aDEVSModel : DEVSModel)]
[aDEVSModel.name]
[/template]
```

appelé sur un modèle DEVS *m* (de la même manière qu'on appellerait une fonction) :
`m.toString()`

aura le même effet que l'opération : `m.name`

Un autre aspect du template est qu'il supporte l'héritage : cela peut par exemple être utilisé pour gérer l'affichage HTML en employant du gras-italique (balises HTML <b> et <i>) pour les modèles couplés, et de l'italique seul (balise HTML <i>) pour les modèles atomiques. La Figure VI-2 montre l'héritage de templates que nous avons utilisé, le template « super » étant laissé vide.

Ainsi, chaque modèle sur lequel on appellera `toString()` s'affichera différemment en fonction de sa nature. Pour changer la manière d'afficher les modèles atomiques ou couplés, il suffit de changer une seule fois le corps de ces templates.

```
35 [template public toString(aDEVSModel : DEVSModel)]
36 [/template]
37
38 [template public toString(aDEVSModel : CoupledDEVS)]
39 <b><i>[aDEVSModel.name/]</i></b>
40 [/template]
41
42 [template public toString(aDEVSModel : AtomicDEVS)]
43 <i>[aDEVSModel.name/]</i>
44 [/template]
```

Figure VI-2 : Héritage de templates `toString()`

b) ***Le template `parcours()`***

De la même manière, il est nécessaire de construire ce template de sorte qu'il puisse être appelé sur n'importe quel modèle DEVS, mais n'avoir d'effet que sur un modèle couplé (voir 6.1.2.e)). Nous voulons ici dans un premier temps afficher le nom du modèle couplé qui joue le rôle de conteneur et la liste de ses sous modèles, sans oublier de leur appliquer le template `toString()`, et dans un second temps effectuer la même opération pour chaque sous-modèle couplé (donc rappeler `parcours()` grâce à la récursivité).

Pour l'affichage de la liste des sous-modèles, nous concaténons un espace afin de séparer les noms de modèles. La Figure VI-3 montre l'héritage de templates, l'affichage du nom du modèle couplé, du nombre des sous-modèles qui le composent (ligne 27). La ligne 29 est consacrée à l'affichage, sur la même ligne, des noms de ces sous-modèles (modèles atomiques d'abord et couplés ensuite). La ligne 31 rappelle le template de parcours sur tous les sous-modèles (via la relation *contains* présente dans le méta-modèle de DEVS), et n'aura d'effet, en pratique, que sur les sous-modèles couplés (voir note ci-après).

```
22 [template public parcours(aDEVSModel : DEVSModel)]
23 [/template]
24
25 [template public parcours(aDEVSCoupled : CoupledDEVS)]
26 <br>
27 [aDEVSCoupled.toString()/] contient <b>[aDEVSCoupled.contains->size()/]</b> sous-modèles :
28 <br>
29 [aDEVSCoupled.contains->sortedBy(oclIsTypeOf(CoupledDEVS)).toString().concat(' ')/]
30 <br>
31 [aDEVSCoupled.contains.parcours()/]
32 [/template]
```

Figure VI-3 : Héritage de templates `parcours()`

Note : Lorsque ce template sera rappelé sur un modèle atomique, ce sera le template « super » qui sera exécuté par défaut, car nous n'avons pas créé de template de parcours spécifique à *AtomicDEVS*. Cette exécution du « super » template ne donnera rien, ce dernier étant vide. Ce mécanisme permet d'éviter d'utiliser des structures conditionnelles ou d'employer des opérations sur les collections en spécifiant des contraintes. Les balises HTML
 (*break line*) ont été rajoutées pour produire des passages à la ligne et augmenter le confort de lecture du fichier.

c) ***Template principal et création du fichier***

Pour franchir cette dernière étape nous allons nous aider des points a), b) et 0 de la sous-section 6.1.2. La Figure VI-4 montre l'intégralité du template principal qui débute avec la mention `@main`.

Note : Cette mention est un commentaire, car la spécification MOF M2T qu'implémente Acceleo ne précise pas comment débuter la transformation : les programmeurs d'Acceleo ont donc décidé d'indiquer le point d'entrée de la transformation sous forme de commentaire (mais interprété tout de même par le moteur de transformation).

La ligne 5 correspond à ce que nous avons défini en 6.1.2.a). Tout ce qui va être généré dans le fichier doit se situer entre les balises `[file]…[/file]`. Les lignes 10 à 13 correspondent à l'identification du modèle, et prévoient le cas où le modèle testé serait atomique (voir 0). Dans le cas où le modèle est couplé, le nom du modèle racine est affiché

(ligne 15) via un appel au template montré sur la Figure VI-2 puis le template parcours (montré sur la Figure VI-3) est appelé sur ce même modèle. Le dernier point d'intérêt de ce template se situe dans sa définition.

```
3  [template public gerFirst (aDEVSModel : DEVSModel) ? (aDEVSModel.ancestors()->isEmpty()=true)]
4  [comment @main /]
5  [file (aDEVSModel.name.concat('_Doc.html'), false, 'ISO-8859-1')]
6  <html>
7  Ce fichier de documentation se nomme <b>[aDEVSModel.name.concat('_Doc.html')/]</b>
8  et il a été généré automatiquement
9  <br><br>
10 [if aDEVSModel.oclIsTypeOf(AtomicDEVS)]
11 Ce modèle DEVS est atomique : il ne contient pas de sous-modèles
12 [else]Ce modèle DEVS est couplé, il s'appelle [aDEVSModel.name/]
13 et voici son arborescence complète
14 <br><br>
15 Modèle racine : [aDEVSModel.toString()/]<br>
16 [parcours(aDEVSModel)/]
17 [/if]
18 </html>
19 [/file]
20
21 [/template]
```

Figure VI-4 : Template principal : création du fichier

En effet, Acceleo dispose de mécanismes d'itération implicites, ce qui fait qu'en invoquant sur une instance de notre-méta modèle le fichier de transformation, le moteur va générer autant de fichier HTML que de modèles et sous-modèles DEVS trouvés dans cette instance. En clair, si l'instance de DEVS sur laquelle on invoque la transformation est un modèle couplé, le moteur d'Acceleo créera autant de fichiers en sortie que de modèles présents dans l'arborescence, et ce en utilisant un mécanisme d'itération implicite qui appelle le template « main » non seulement sur le modèle conteneur, mais aussi sur chaque sous-modèle de son arborescence.

Pour éviter cela et n'avoir qu'un seul fichier en sortie, il faut appliquer une condition à ce template principal. Nous nommons tout d'abord notre template `getFirst()` : son but sera de n'être appelé qu'une seule fois, et donc détecter que le modèle passé en paramètre est bien la racine de l'arborescence. Pour cela nous utilisons la possibilité offerte par le langage Acceleo d'exprimer des gardes (préconditions) sur les templates au moyen d'expressions en langage OCL, introduites par le signe `?` : le template sera exécuté si et seulement si la précondition est vraie, dans notre cas si le modèle passé en paramètre n'a pas d'« ancêtres » (i.e. contient tous les autres). Cette condition est exprimée sur la ligne 3.

L'annexe 8 présente l'intégralité du code Acceleo utilisé.

Note : Nous avons choisi comme encodage ISO-8859-1 afin de pouvoir afficher correctement les éventuels accents. Par défaut, l'encodage le plus employé en M2T est UTF-8.

6.1.4. Exemple d'application

Exécutons maintenant cette définition de transformation .mtl sur une instance de modèle DEVS, par exemple le modèle du chapitre I cité en 6.1.1.b).

Cette application est illustrée par la Figure VI-5, dont la partie 1 est une capture d'écran d'une instance dynamique DEVS décrivant le modèle-exemple, la partie 2 est le code

source HTML généré par l'exécution de cette transformation, et la partie 3 le résultat obtenu dans un navigateur Web classique.

Figure VI-5 : Application de la transformation « DEVS2HTMLDoc.mtl »

6.2. Générer automatiquement du code objet

Après avoir montré un premier exemple de génération d'artefact textuel à partir d'un modèle DEVS passé en entrée, et avoir donné un aperçu concret des possibilités offertes par l'approche M2T via les templates, nous abordons ici un point clef de notre travail puisqu'il s'agit de transformer automatiquement nos modèles PIM DEVS en code objet. Cela permet leur simulation sur une plateforme donnée. Nous avons choisi comme exemple

d'implémentation la plateforme PyDEVS qui utilise le langage Python, pour trois raisons principales :

- La souplesse du langage Python, langage orienté objet
- L'existence d'une documentation conséquente sur le site de l'université qui propose ce simulateur, ce qui nous aidera à créer notre modèle de code
- La compatibilité de PyDEVS avec l'environnement utilisé par les chercheurs en informatique du projet T.I.C. : DEVSimPy

Notre exemple de génération de code vers la plateforme PyDEVS ne doit pas occulter le fait qu'un même PIM DEVS peut être simulé sur plusieurs autres plateformes différentes, pour peu que l'on crée la définition de la transformation vers chaque plateforme.

Nous montrons dans cette partie comment nous pouvons raisonner de manière générique sur des règles de transformation M2T, ce qui permet parfois de réutiliser certaines de ces règles. Ce type de raisonnement générique est analogue à ce que nous avons déjà montré dans le chapitre précédent, pour les transformations M2M. Dans ce cas, c'était le méta-modèle source qui n'était pas connu, mais dont on connaissait cependant à l'avance certaines caractéristiques, et le méta-modèle cible qui était parfaitement connu : DEVS. Dans le cas actuel, c'est le contraire : on connait parfaitement le méta-modèle source (DEVS) mais on ne connaît pas le méta-modèle cible. On peut néanmoins, en se limitant aux plateformes DEVS orientées-objet, exprimer un certain nombre d'invariants concernant ces plateformes de simulation d'une part, et la manière d'interroger le modèle DEVS source d'autre part.

En effet, l'avantage de posséder un méta-modèle DEVS permettant de spécifier des modèles DEVS indépendants de toute plateforme est que, quelle que soit l'implémentation ultérieure choisie, la manière de récupérer des éléments dans le modèle DEVS source sera toujours la même. Seule diffèrera la manière de les inclure dans du code, et ceci dépend de la plateforme cible (mots clefs du langage, etc…). Le fait que les parties de code que nous avons qualifiées de variables puissent devenir des invariants si on considère non pas une mais plusieurs transformations peut constituer un paradoxe.

Mais cela s'explique simplement : si l'on considère un programme type attendu par un simulateur DEVS, on se rend compte que les parties variables concernent toujours des éléments extraits du modèle DEVS. Si l'on considère maintenant plusieurs patrons de code attendus par plusieurs simulateurs DEVS, et que l'on réfléchit sur les transformations permettant d'extraire des éléments des modèles DEVS, on s'aperçoit vite que la manière de récupérer ces éléments sera toujours la même quelle que soit la plateforme (pour peu que l'on utilise toujours le même langage de transformation).

L'implémentation en code objet de la définition d'un modèle ne remet donc pas en cause la nécessité, exprimée plusieurs fois dans ce travail, de disposer de modèles DEVS indépendants de toute plateforme, mais, bien au contraire, la renforce. Disposer de ces modèles indépendants permet leur stockage, leur échange, leur modification, et surtout leur réutilisabilité, sans tenir compte de contraintes liées à leur environnement. Certes, ces modèles ne seraient d'aucune utilité s'ils n'étaient pas simulés, et la génération de code permet de les rendre pleinement productifs, mais il est de notre point de vue essentiel de les conserver le plus longtemps possible sous leur forme de PIM, ce qui permet, au gré des

besoins et des collaborations scientifiques, de les transformer en code Python, puis en code Java par exemple, obtenus tous deux à partir du même PIM.

Pour illustrer nos propos, nous nous appuyons sur des captures d'écran des points les plus significatifs de la transformation de DEVS vers PyDEVS. Nous omettons volontairement les parties les moins importantes (déclaration des fonctions, etc…), le code complet des transformations M2T figurant dans l'annexe 7 de ce mémoire. Nous avons largement recours aux héritages de templates, permettant une grande modularité, pour profiter au maximum de l'architecture de notre méta-modèle DEVS et gagner en simplicité, en efficacité et en réutilisabilité au niveau des transformations.

6.2.1. **Travail préparatoire**

Même si l'absence de méta-modèle cible nous empêche de créer des règles génériques précises de nos transformations, nous proposons ici d'identifier certaines caractéristiques communes aux simulateurs DEVS orientés objet, qui pourront faire office de guide pour la création de règles et nous permettront d'en réutiliser certaines.

a) ***Similitudes et différences dans les simulateurs DEVS orientés objet***

Lors de la présentation du formalisme DEVS, nous avons souligné qu'une de ses particularités remarquables était de séparer la modélisation et la simulation, dans le sens où une fois qu'un modèle est écrit, son simulateur est (presque) automatiquement fourni, grâce à un mécanisme d'instanciation. Ce mécanisme est commun à tous les simulateurs DEVS orientés objet.

Nous renvoyons le lecteur au premier chapitre de ce document, et plus particulièrement la sous-section dédiée à la simulation DEVS, pour plus d'information sur cette dernière, et aussi à [Zeigler et al. 2000] qui présente exhaustivement les algorithmes génériques de simulation DEVS et l'architecture de classes communément employée, sur lesquels nous nous basons pour établir nos règles.

Les simulateurs DEVS disposent, au minimum, d'un fichier de définition globale contenant une hiérarchie de classes de base associée aux concepts DEVS (modèles atomiques et couplés, ports…). Les modèles définis par le programmeur doivent hériter de ces classes. Au moment de la simulation, les modèles définis sont instanciés et interrogés par le SE (*Simulation Engine*).

Les deux classes principales sont toujours celles qui correspondent aux modèles atomiques et couplés : les premières décrivent un comportement et se composent de fonctions DEVS, de ports, tandis que les secondes décrivent une structure et sont composées, entre autres, de modèles.

Les fonctions DEVS sont exprimées sous forme de combinaisons de conditions sur les variables d'état, et de retours de valeurs, agissant directement sur l'évolution du le modèle, ou bien renseignant sur l'avancement du temps, ou encore envoyant un message sur un port de sortie.

En revanche, tout ce qui se rapporte aux mots-clefs du langage, à l'affichage des valeurs, aux opérateurs, et plus généralement aux mécanismes d'instanciation, d'héritage et

d'affectation est spécifique à un langage. Bien entendu, il existe parfois des similitudes syntaxiques entre langages.

D'autres différences existent au niveau de la structure des modèles : certaines plateformes attendent un fichier différent pour le modèle et chacun de ses sous-modèles, d'autres utilisent le modèle DEVS et tous ses sous-modèles au sein du même fichier.

b) ***Patrons de code***

Pour pallier l'absence de méta-modèle cible, il est préférable de disposer d'un patron de code cible pour le simulateur DEVS vers lequel on veut implémenter nos modèles : certains simulateurs sont bien documentés et fournissent un tel patron, qui se compose d'un ou de plusieurs fichiers textuels de la forme que doit avoir le code final.

Toutes proportions gardées, ces fichiers peuvent être considérés comme des « modèles » de modèles DEVS codés dans un langage particulier, ce qui revient à dire que ce sont des sortes de méta-modèles. L'analogie s'arrête ici, car ces patrons de code ne sont pas exprimés dans un méta-formalisme bien défini, et leur utilisation se fait de manière intuitive. Néanmoins, leur utilisation facilite grandement la définition d'une transformation M2T. Ils peuvent constituer un point de départ pour la création des transformations. De tels patrons existent pour PyDEVS, et sont disponibles sur la page du MSDL dédiée à DEVS (université de Mac Gill)[1]. Un simple copier-coller de ce dernier dans l'interface de programmation choisie (ici, le plug-in Acceleo) permet de disposer du squelette du code final. Il faut ensuite identifier les parties variables et les remplacer par des templates.

c) ***Fonctionnement du simulateur PyDEVS***

Le simulateur PyDEVS fonctionne grâce à deux fichiers.

Fichier Simulation	*importe le*	**Fichier Principal**
instancie le modèle racine instancie la classe simulator		modèles atomiques modèles couplés fonctions atomiques fonctions de couplage états

Tableau VI-1 : Les deux fichiers utilisés par PyDEVS pour simuler un modèle

Ils sont montrés sur le Tableau VI-1: un fichier principal contenant toutes les informations qui concernent les modèles (fonctions de couplage, fonctions atomiques…) et un fichier plus « léger » servant à lancer la simulation, qui contient notamment des informations sur cette dernière et qui est lié au fichier principal. Les sections de 6.2.2 à 6.2.6 traitent de la génération de code du fichier principal, tandis que la section 6.2.7 montre comment générer le fichier qui servira à appeler la simulation.

6.2.2. Structure globale du fichier cible PyDEVS principal

Commençons la définition de notre transformation par la structure même du fichier de sortie principal d'une part, et par la déclaration de la structure éventuelle des modèles DEVS qu'il contient d'autre part.

[1] http://msdl.cs.mcgill.ca/projects/projects/DEVS/pydevs.zip

La lecture du patron de code de PyDEVS nous informe sur la structure attendue du modèle DEVS : celui-ci doit se trouver dans le même fichier Python *.py*. En clair, en cas de modèle couplé, le modèle et ses sous-modèles doivent tous être définis dans le même fichier (que nous avons déjà évoqué, sous le nom de « fichier principal »).

Nous adoptons exactement le même raisonnement que pour la transformation décrite en 6.1.3 avec :

- un template *getFirst()*, chargé de détecter au moyen d'une garde (analogue à celle présentée en 6.1.3.c)) que le modèle passé en paramètre n'est le sous-modèle d'aucun modèle (i.e. il joue le rôle de racine de la structure) : c'est le point d'entrée de la transformation, il ne sera exécuté qu'une seule fois ;
- un héritage de templates *generateModel()* appelés par le template principal : ils correspondent à l'architecture de base *DEVSModel* (classe abstraite *DEVSModel*, classes filles *AtomicDEVS* et *CoupledDEVS*) ;
- un fichier de sortie *.py* ayant pour nom le premier modèle trouvé (qui sera donc forcément un modèle atomique unique ou un modèle couplé contenant d'autres sous-modèles) auquel on concatène la chaîne « AutoGen ». ;
- un en-tête spécifique au simulateur PyDEVS, récupéré sur le patron de code du simulateur.

```
4  [template public getFirst(a : DEVSModel) ? (a.ancestors()->isEmpty()=true)]
5  [comment @main /]
6  [file (a.name.concat('AutoGen.py'), false, 'ISO-8859-1')]
7  ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
8  # [a.name.concat('Model.py')/] --- Based on the template for atomic- and coupled
9  #                          --------------------------------
10 #                               Jean-Sébastien  BOLDUC
11 #                                  Hans Vangheluwe
12 #                            McGill University (Montréal)
13 #                          --------------------------------
14 ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
15 [generateModel(a)/]
16 [/file]
17 [/template]
18
19 [template public generateModel(a : DEVSModel)]
20 [/template]
21
22 [template public generateModel(a : AtomicDEVS)]
23 ...
24 [/template]
25
26 [template public generateModel(c : CoupledDEVS)]
27 [for (sub: DEVSModel | c.contains) separator('\n')]
28 [generateModel(sub)/]
29 [/for]
30 ...
31 [/template]
```

Figure VI-6 : Création de la structure d'un fichier *.py*

Ici aussi, le template *generateModel()* concernant la classe abstraite *DEVSModel* est laissé vide. L'intégralité du code (excepté l'en-tête contenu entre les balises `[file]`) se situera dans les deux templates *generateModel()* correspondant respectivement à *AtomicDEVS* et *CoupledDEVS*. *generateModel()* est rappelé récursivement sur chaque sous-modèle d'un modèle couplé.

Cette solution est présentée sur la Figure VI-6 : elle est générique et s'adapte à tous les simulateurs DEVS qui ne requièrent qu'un seul fichier pour contenir tous les modèles. Dans le cas où un fichier par modèle serait requis, l'unique manipulation à effectuer consiste à supprimer la garde sur *getFirst()* (ligne 4), afin de générer autant de fichiers cible que de modèles source contenus dans l'arborescence (via une itération implicite), chacun nommé en fonction du nom du modèle auquel il se rapporte.

6.2.3. Eléments communs aux modèles atomiques et couplés

Il existe quelques éléments communs aux modèles atomiques et couplés, que nous retrouvons dans MetaDEVS, dans la spécification formelle du formalisme et également dans PyDEVS. Il s'agit du nom du modèle, et de ses ports.

a) *Nommage du modèle*

Tout modèle DEVS se conformant à MetaDEVS doit être explicitement nommé, il en va de même pour les modèles DEVS créés dans des simulateurs orientés objet, qui sont des classes et qui donc doivent posséder un nom. Nous pouvons récupérer le nom du modèle source, grâce à l'attribut « name » de la méta-classe DEVSModel puis générer le code de la classe correspondante en employant un héritage. La Figure VI-7 montre cet héritage de templates utilisé pour nommer les classes.

```
[template public generateModel(a : DEVSModel)]
[/template]

[template public generateModel(a : AtomicDEVS)]
class [a.name/](AtomicDEVS):
[/template]
[template public generateModel(c : CoupledDEVS)]
class [c.name/](CoupledDEVS):
[/template]
```

Figure VI-7 : Création des noms de classe

b) *Ports*

Nous créons dans un premier temps l'équivalent d'une méthode Acceleo: `toStringPort()` chargée d'afficher le nom du port en majuscule, puis nous créons un template `declarePort()` pour tous les modèles DEVS. Ce template récupère chaque port contenu par le modèle, en distinguant les ports d'entrée et de sortie, et l'ajoute (après l'avoir affiché correctement) au modèle DEVS, ce qui, dans le code final, est représenté par des appels aux méthodes PyDEVS *addInPort*() et *addOutPort*().

```
[template public declarePort(aDEVSModel : DEVSModel)]
    [for (pin: InputPort | aDEVSModel.inputPort)]
    self.[pin.toStringPort()/] = self.addInPort(name="[pin.toStringPort()/]");
    [/for]
    [for (pout: OutputPort | aDEVSModel.outputPort)]
    self.[pout.toStringPort()/] = self.addOutPort(name="[pout.toStringPort()/]");
    [/for]
[/template]
[template public toStringPort(p : Port)]
[p.portID.toUpper()/]
[/template]
```

Figure VI-8 : Déclaration des ports

6.2.4. Eléments spécifiques aux modèles couplés : création de la hiérarchie

Les éléments les plus significatifs des modèles couplés sont, en plus de leur nom et leurs ports, traités précédemment, la liste des modèles qu'ils contiennent et les liens qui les unissent : ceci constitue la hiérarchie de modèles de DEVS, c'est tout simplement une description structurelle de modèles.

a) *Sous-modèles*

La démarche est analogue à celle qui effectue l'ajout de ports. Le but est de générer l'appel d'une méthode Python pour créer la liste d'enfants du modèle couplé. On utilise la référence *contains* du *CoupledDEVS* concerné, ce qui nous donne une liste d'enfants. Il n'est pas nécessaire de créer un nouveau template pour cela, on insère donc cette expression directement dans le template du modèle couplé : la Figure VI-9 montre cette expression. Elle génère une liste d'appels à la méthode *addSubModel()* de PyDEVS.

```
[for (sub: DEVSModel | c.contains) separator('\n')]
self.[sub.name.toLower()/] = self.addSubModel([sub.name/](name="[sub.name.toLower()/]"));
[/for]
```

Figure VI-9 : Ajout des enfants

b) *Fonctions de couplage*

De la même manière qu'il a été nécessaire de fournir une liste de sous-modèles, nous devons, pour chaque modèle couplé, fournir la liste des différents couplages qui le composent, ce qui se traduira dans le code final par une suite d'appels à la méthode PyDEVS *connectPorts*() prenant en paramètre les ports concernés. Le passage du méta-modèle DEVS au code Python est pratiquement transparent : il suffit de récupérer pour chaque modèle couplé tous ses IC, tous ses EIC et tous ses EOC. Si un port appartient non pas au modèle couplé courant mais à un sous-modèle de celui-ci, il est nécessaire de spécifier le nom du sous-modèle : ici, on utilise OCL pour récupérer l'objet contenant le port (modèle atomique ou couplé, donc de type *DEVSModel*), cet objet fait ensuite l'objet d'un *cast* (forçage de type) et on accède enfin à son nom, que l'on met en minuscules (tout comme on l'a fait dans la fonction précédente d'ajout d'enfants) avec l'invocation de *toLower()*.

La requête OCL passée sur une instance de port appartenant à un sous-modèle (atomique ou couplé) du modèle courant a la forme suivante :

```
eContainer().oclAsType(DEVSModel).name.toLower()
```

```
[template public declareCouplage(aCoupledDEVS : CoupledDEVS)]
[for (ic: IC | aCoupledDEVS.IC)]
self.connectPorts(self.[ic.IC_out.eContainer().oclAsType(DEVSModel).name.toLower()/].[ic.IC_out.toStringPort()/]
, self.[ic.IC_in.eContainer().oclAsType(DEVSModel).name.toLower()/].[ic.IC_in.toStringPort()/])
[/for]

[for (eic: EIC | aCoupledDEVS.EIC)]
self.connectPorts(self.[eic.EIC_coupled_in.toStringPort()/]
, self.[eic.EIC_in.eContainer().oclAsType(DEVSModel).name.toLower()/].[eic.EIC_in.toStringPort()/])
[/for]

[for (eoc: EOC | aCoupledDEVS.EOC)]
self.connectPorts(self.[eoc.EOC_out.eContainer().oclAsType(DEVSModel).name.toLower()/].[eoc.EOC_out.toStringPort()/]
, self.[eoc.EOC_coupled_out.toStringPort()/])
[/for]
[/template]
```

Figure VI-10 : Déclaration des couplages

Encore une fois, l'appel à la méthode `toStringPort()` permet un affichage correct (conforme aux conventions de PyDEVS) des ports. Le template `declareCouplage()` est montré dans la Figure VI-10. Il suffit, pour l'exécuter, de l'appeler en lui passant le modèle couplé courant en paramètre, depuis le template `generateModel()` du modèle couplé. Ensuite, `declareCouplage()` va récupérer et générer le code de des couplages IC, puis les EIC et enfin EOC. Pour des raisons de lisibilité, des retours chariot ont été insérés dans le code du template.

La fonction de sélection est laissée vierge, elle sera gérée par le simulateur PyDEVS qui par défaut choisira le premier élément de la liste parmi ceux qui doivent gérer un évènement simultanément. Les ports ont quant à eux été décrits en 6.2.3.b), nous venons donc de terminer notre définition de transformation pour les modèles couplés DEVS.

6.2.5. **Réécriture des valeurs manipulées par les modèle DEVS**

Avant de présenter la transformation d'un modèle *AtomicDEVS* en code PyDEVS il est nécessaire de se pencher sur le cas des valeurs que manipule un modèle DEVS et comment elles doivent être affichées en Python. Nous définissons dans cette sous-section des méthodes de type `toString()` que nous emploierons pour la transformation proprement dite. Non seulement le fait de procéder de la sorte clarifie l'écriture des transformations, mais cela permet aussi, en fonction de la plateforme cible, de modifier le moins possible nos templates.

a) ***Définition de l'affichage des* DEVSXpression *en PyDEVS***

Il nous faut tout d'abord gérer l'affichage de chaque *DEVSXpression*. L'utilisation d'un héritage de templates constitue selon nous la meilleure solution possible, et permet selon le langage et la plateforme cible une modification aisée et rapide de cet affichage. Cet héritage de templates complet est montré sur la Figure VI-11.

Concernant les *StateVar* (qui sont des *DEVSXpression* spéciales) il n'y a pas de difficulté particulière : on se contente de renvoyer leur nom *DEVSid* (ligne 5). Concernant les chaînes de caractères, elles doivent être mises entre « double quotes », les caractères seuls aussi (ils sont considérés en Python comme des chaînes de caractères) (lignes 12 et 21). Les entiers et les réels sont affichés tels quels (lignes 15 et 18).

```
1  [template public toString(devsexp : DEVSXpression)]
2  [/template]
3
4  [template public toString(sv : StateVar)]
5  [sv.DEVSid/]
6  [/template]
7
8  [template public toString(lbv : LitteralBasicValue)]
9  [/template]
10
11 [template public toString(lbv : StringValue)]
12 "[lbv.str_val/]"
13 [/template]
14 [template public toString(lbv : IntValue)]
15 [lbv.int_val/]
16 [/template]
17 [template public toString(rv : RealValue)]
18 [rv.real_val/]
19 [/template]
20 [template public toString(lbv : CharValue)]
21 "[lbv.char_val/]"
22 [/template]
23 [template public toString(lbv : BooleanValue) post(trim())]
24 [if (lbv.bool_val=true)]True
25 [else]False
26 [/if]
27 [/template]
```

Figure VI-11 : Template général `toString()`

Les booléens doivent être affichés sous forme de `True` ou `False` (lignes 23 à 27).

b) ***Affichage des comparateurs***

Les comparateurs que nous avions définis sous forme d'énumération dans notre méta-modèle MetaDEVS doivent être « traduits » en une symbolique interprétable par le langage sur lequel repose la plateforme cible. Nous utilisons un template `rewriteComp()` pour les afficher correctement (voir Figure VI-11). Selon le langage choisi, ce template pourra être modifié facilement, mais l'écriture des opérateurs en Python est similaire à beaucoup d'autres langages orientés objet.

```
[template public rewriteComp(comp : Comparator) post(trim())]
[if self.toString()='EQUALS']==
[elseif self.toString()='UPPER_THAN']>
[elseif self.toString()='LOWER_THAN']<
[elseif self.toString()='UPPER_OR_EQUAL_THAN']>=
[elseif self.toString()='LOWER_OR_EQUAL_THAN']<=
[/if]
[/template]
```

Figure VI-12 : Template `rewriteComp()`

c) ***Cas d'une durée infinie***

Il nous reste un dernier template de mise en forme à écrire : celui qui va gérer le fait que la fonction *ta* renvoie une valeur infinie, ce qui signifie que le système reste dans cet état jusqu'à ce qu'un évènement externe vienne le perturber.

```
[template public toStringTA(tar : TimeAdvanceRule)]
[if (tar.ta_value >= 999999)]
 INFINITY
[else]
[tar.ta_value/]
[/if]
[/template]
```

Figure VI-13 : Template `toStringTA()`

Ceci se traduit en PyDEVS par le mot clef `INFINITY`. Nous faisons le choix (arbitraire) de qualifier d'infinie toute valeur *ta_value* d'une *TimeAdvanceRule* supérieure à l'entier 999999. Le template `toStringTA()` montré sur la Figure VI-13 teste cette condition sur la *TimeAdanceRule*. Nous pouvons désormais aborder la génération de code pour un modèle atomique DEVS.

6.2.6. Générer le code des fonctions atomiques DEVS

Nous présentons ici quelques points clefs de la transformation d'un modèle *AtomicDEVS* en code pour PyDEVS. La création des ports ayant été résolue précédemment (en 6.2.3.b)), nous ne l'abordons pas ici. Il nous reste donc à traiter la gestion des états et les fonctions atomiques DEVS.

Quelle que soit la plateforme de destination, il y faudra toujours définir ces 4 fonctions atomiques DEVS. Ceci se fait en parcourant le modèle source avec les *DEVSRule* contenues dans les fonctions atomiques DEVS, et en récupérant pour chaque règle la ou les conditions d'une part, et la ou les actions d'autre part. En pratique, les plateformes DEVS orientées objet implémentent ces conditions sous forme de tests, et les actions sous forme de modification des valeurs des variables d'état appartenant au modèle, ou au retour de valeur dans le cas de l'avancement du temps.

Nous ne présentons pas l'intégralité du corps des fonctions (déclaration, commentaire), mais uniquement leurs parties variables générées grâce à des templates.

PyDEVS prévoit de surcroît, pour chaque modèle atomique, une classe permettant de créer des évènements, et une classe (optionnelle mais fortement recommandée) encapsulant l'état du système. Nous n'utilisons pas la première mais, en revanche, nous avons recours à la seconde car elle permet de gérer simplement les états multidimensionnels.

a) *Représentation de l'état*

Comme nous l'avons dit, les concepteurs de PyDEVS ont prévu d'encapsuler l'état dans une classe distincte de la classe correspondant au modèle atomique.

Le code permettant de la générer se trouve dans le template `generateModel()` qui concerne les AtomicDEVS.

La classe d'encapsulation des états accepte autant de paramètres, et contient autant d'attributs, que le modèle comporte de variables d'états. Nous choisissons pour la nommer de récupérer le nom du modèle atomique correspondant et de lui concaténer la chaîne de caractères « EncapState ». Enfin, toujours selon les spécifications PyDEVS, la classe « EncapState » dispose d'une méthode *get()* et d'une méthode *str()*, la première étant destinée à récupérer l'état courant du système et la seconde ayant pour but de produire un affichage clair de l'état, lorsque l'utilisateur demande une simulation détaillée.

```
# TOTAL STATE:
#  Define 'state' attribute (initial sate):
self.state = [a.name.concat('EncapState')/]([for (v:StateVar|a.is_defined_by)separator(',')]
[v.initial_value.toString()/]
[/for])
```

Figure VI-14 : Instanciation de la classe d'encapsulation de l'état

Cette classe d'encapsulation est instanciée dans la classe correspondant au modèle atomique et cette instance se retrouve contenue dans la variable (Figure VI-14). Lors de l'instanciation, on passe en argument de cette fonction d'encapsulation la valeur initiale de chaque variable d'état du modèle.

À l'origine, en PyDEVS, chaque fonction atomique provoquant un changement d'état (transition externe et interne) ré-instanciait la classe « EncapSate », mais nous avons pour notre part opté pour une solution permettant à la fois un simple changement de valeur du ou des attributs concernés plutôt qu'une ré-instanciation. La raison est simple : si l'utilisateur spécifie un modèle MetaDEVS avec plus d'une variable d'état, et ne crée une StateChangeAction pour chacune d'elles, les autre variables ne seront pas modifiées et resteront néanmoins dans leur état courant, alors qu'une ré-instanciation aurait eu pour but de les ramener à leurs valeurs initiales.

Enfin, pour faciliter les tests sur les états en Python, nous choisissons de faire renvoyer à la méthode *get()* une énumération des états sous forme de dictionnaire. Cette structure de données Python est couramment utilisée et permet d'associer des clefs et des valeurs.

```
1  class [a.name.concat('EncapState')/]:
2
3  def __init__(self, [for (v:StateVar|a.is_defined_by) separator(', ')][v.toString()/][/for]):
4  """Constructor (parameterizable).
5  """
6
7      [for (v:StateVar|a.is_defined_by)]
8      self.[v.toString()/]=[v.toString()/]
9      [/for]
10
11 def get(self):
12     return {[for (v:StateVar|a.is_defined_by) separator(', ')]
13     '[v.toString()/]': self.[v.toString()/]
14     [/for]}
15
16 def __str__(self):
17     return [for (v:StateVar|a.is_defined_by) separator(' + " and " + ')]
18     " [v.toString()/] = "+ str(self.[v.toString()/])
19     [/for]
```

Figure VI-15 : Génération du code de la classe d'encapsulation de l'état

La Figure VI-15 est une capture d'écran (sur laquelle des retours chariot ont été appliqués, pour des raisons de lisibilité) montrant le code Acceleo qui génère la classe « EncapState ». Les éléments importants sont :

- la définition du constructeur (ligne 3) dans lequel sont collectés puis affichés les noms des StateVar du modèle MetaDEVS ;
- le corps du constructeur lui-même (lignes 7 à 9) qui affecte à chaque variable d'état du modèle la valeur correspondante passée en entrée ;
- la méthode *get()* qui renvoie, sous forme de dictionnaire Python (reconnaissable grâce aux accolades) un couple StateVar(clef)/valeur ;
- la méthode *str()* qui concatène l'affichage des noms StateVar et de leurs valeurs. Cette méthode n'est utilisée lors de la simulation que pour des raisons de lisibilité.

Comme nous séparons explicitement la récupération de l'état d'un modèle, via un dictionnaire, pour faciliter les tests, et la récupération de l'état courant afin de le modifier, nous créons en outre un petit template `retrieveState()` (Figure VI-16) qui sera appelé dans chaque fonction atomique.

```
1 [template public retrieveState(a : AtomicDEVS)]
2 statedico = self.state.get()
3 state = self.state
4 [/template]
```

Figure VI-16 : Le template `retrieveState()`

Ce template montre bien la séparation que nous préconisons entre la variable baptisée *statedico* (qui contient le dictionnaire de données, issu de la méthode *get()* décrite précédemment) et la variable *state* qui contient l'instance courante de la classe d'encapsulation des états (Figure VI-14). L'une sera utilisée pour effectuer des tests, tandis que l'autre sera utilisée pour agir directement sur l'état du modèle.

Le recours à un dictionnaire est selon nous très important : il permet de gérer beaucoup plus facilement le *return* en cas de variables d'état multiples (états multidimensionnels).

Note : Nous avons choisi de passer un modèle atomique en paramètre de ce template pour bien illustrer le fait que ce dernier ne concerne que les modèles atomiques, mais nous aurions tout aussi bien pu l'omettre, car il n'est pas utilisé dans le corps du template.

b) ***Traduire correctement une Condition et une Action***

Comme nous l'avons vu dans le chapitre IV, l'état d'un modèle atomique DEVS à un instant donné est donné par les valeurs littérales de l'ensemble des StateVar qui le composent, à ce même instant. De plus une fonction DEVS est composée d'un ensemble de règles. Chaque règle est composée d'une ou plusieurs conditions (portant sur les StateVar) et d'une ou plusieurs actions (portant également sur les *StateVar*, ou les *OutputPort*…).

Notre but ici est de transcrire correctement ces fonctions dans un code objet cohérent. Pour cela, l'idée générale est, pour chaque fonction DEVS, de créer une boucle de recherche qui récupère toutes les règles qu'elle contient, et pour chaque règle de créer une boucle qui récupère la ou les conditions, une boucle qui récupère la ou les actions, sans oublier de récupérer le comparateur, et d'afficher correctement le tout.

Encore une fois, nous aurons recours à un héritage de templates basé sur la hiérarchie de méta-classes de MetaDEVS. Commençons par l'affichage des éléments de plus « bas niveau » : les conditions et les actions.

Il existe deux sortes de conditions selon notre méta-modèle DEVS, les conditions sur les ports (utilisées pour la fonction de transition externe) et les conditions sur la ou les variables d'état, utilisées pour toutes les fonctions atomiques DEVS. Notre objectif est, pour chaque condition, d'afficher son membre gauche (avec le template `toStringPort()` s'il s'agit d'un port ou `toString()` s'il s'agit d'une *StateVar*), d'invoquer `rewriteComp()` pour afficher le comparateur, puis d'afficher son membre droit (avec la méthode `toString()`). Nous pouvons y parvenir grâce au template `toStringCond()` qui affichera de manière correcte chacune des conditions possibles (voir Figure VI-17).

```
[template public toStringCond(cond : Condition)]
[/template]
[template public toStringCond(svc : StateVarComparison)]
statedico['['/]'[svc.left_member.toString()/]']'/] [svc.comparator.rewriteComp()/] [svc.right_member.toString()/]
[/template]
[template public toStringCond(ipc : InputPortComparison)]
[ipc.left_member.toStringPort()/] [ipc.comparator.rewriteComp()/] [ipc.right_member.toString()/]
[/template]
```

Figure VI-17 : Le template `toStringCond()`

Ce template s'appuie sur le fait que l'état du modèle est contenu, sous forme de dictionnaire, par la variable *statedico* (mise à jour au début de chaque fonction atomique DEVS, via le code que génère le template `retrieveState()`).

Raisonnons maintenant de manière analogue sur les actions. Nous savons qu'il existe deux types d'actions : celles qui provoquent un changement d'état, et celles qui provoquent l'envoi d'un message sur un port de sortie. Le seul cas particulier est la fonction d'avancement du temps, qui renvoie juste une valeur, c'est en fait la fonction atomique la plus simple.

Il faut donc, selon le cas, afficher la nouvelle valeur d'un état ou le contenu d'un message : encore une fois, cela se fait simplement, en invoquant le template `toString()`.

En ce qui concerne les actions à mener lors d'un changement d'état, l'idée est de générer une liste d'instructions qui concernent la ou la StateVar à modifier (toutes contenues dans *state*, qui représente l'instance de la classe d'encapsulation de l'état). Ces instructions sont donc des affectations. On termine en renvoyant *state*. Ce template nommé `toStringAct()` est montré sur la Figure VI-18.

```
1  [template public toStringAct(rule : Rule)]
2  [/template]
3  [template public toStringAct(tfRule : TransitionFunctionRule)]
4  [for (sca : StateChangeAction | tfRule.changes_state)separator('\n')]
5  state.[sca.state_to_be_changed.DEVSid/]=[sca.new_value.toString()/]
6  [/for]
7  return state
8  [/template]
9  [template public toStringAct(lambdaRule : LambdaRule)]
10 [lambdaRule.sends_message.message.toString()/]
11 [/template]
```

Figure VI-18 : Le template `toStringAct()`

Le cas où de multiples actions seraient spécifiées dans une règle DeltaIntRule ou DeltaExtRule est donc résolu, mais il nous reste à prévoir le cas où plusieurs conditions existeraient, car `toStringCond()` n'est pour l'instant capable que de générer le code correspondant à une seule et unique condition. Nous avons besoin pour cela d'un template supplémentaire, dont le rôle est de lier les conditions entre elles par des appels successifs à `toStringCond()`. Nous avons nommé ce template `toStringMultipleCond()` (Figure VI-19). Il doit être systématiquement appelé, que la règle contienne une ou plusieurs conditions.

Si la règle ne comporte qu'une seule condition, son invocation revient à appeler simplement `toStringCond()`. S'il en contient plusieurs, `toStringCond()` sera appelé sur chacune d'elles, et les conditions seront séparées par le mot-clef Python and. En effet, nous considérons que dans le cas de conditions simples, le fait d'en spécifier plusieurs pour la même règle revient à, implicitement, les lier avec l'opérateur logique « ET ». Revenons à

présent au corps du `generateModel()`, et utilisons tous les templates que nous venons de voir pour générer le code Python nos fonctions atomiques DEVS.

```
[template public toStringMultipleCond(rule : Rule)]
[for (cond : Condition | rule.tests) separator (' and ')][cond.toStringCond()/][/for]
[/template]
```

Figure VI-19 : Le template `toStringMultipleCond()`

Rappelons qu'au début de chacune d'entre elles, nous faisons un appel au template `retrieveState()`.

c) *Générer la fonction d'avancement du temps*

Il s'agit, comme nous l'avons dit précédemment, de la fonction la plus simple : son but est de renvoyer une valeur correspondant à chaque état. Il nous faut donc parcourir la fonction *TimeAdvance* du modèle source et, pour chaque règle qu'elle contient, afficher les conditions sous forme d'enchaînement de conditions Python, puis renvoyer la valeur contenue dans la variable *ta_value*, à laquelle on applique `toStringTA()` pour gérer le cas où l'état serait passif (valeur infinie).

Nous employons comme séparateur le mot-clef Python `elif` (Figure VI-20).

```
 if [for (taRule: TimeAdvanceRule | a.contains_time_advance_function.is_defined_by_ta_rules)
separator('        elif ')] [taRule.toStringMultipleCond()/]:
    return [taRule.toStringTA()/]
 [/for]
```

Figure VI-20 : Génération du code de la fonction *TimeAdvance*

d) *Générer la fonction de transition interne*

Ici, nous procédons exactement de la même manière que pour la fonction d'avancement du temps au niveau des conditions (séparateur identique, parcours et affichage identiques), mais nous devons afficher en plus la ou les actions (changements portant sur une seule ou plusieurs StateVar) à effectuer. Cela se traduit par un appel à `toStringAct()`. Le changement d'état s'effectue sous la forme d'une mise à jour de la variable *state*.

```
 if [for (dint: DeltaIntRule | a.contains_internal_transition_function.is_defined_by_deltaint_rules)
separator ('        elif ')] [dint.toStringMultipleCond()/]:
    [dint.toStringAct()/]
 [/for]
```

Figure VI-21 : Génération du code de la fonction *DeltaInt*

e) *Générer la fonction de transition externe*

Cette fonction est en partie identique à la précédente, et se compose d'un test d'état associé à une action de changement d'état. La seule différence réside dans l'adjonction d'une condition supplémentaire qui teste la valeur d'un port (i.e. un évènement d'entrée, d'un point de vue simulation). Nous avons également soumis la génération du code de cette fonction à une condition (qui ne figure pas sur les captures d'écran mais se trouve sur le code en annexe 7) : si la fonction DeltaExt du modèle atomique courant est vide (i.e. ne contient pas de règles), nous ne générons aucun code.

Pour ce faire, PyDEVS prévoit d'abord de récupérer les valeurs présentes sur tous les ports d'entrée du modèle atomique (Figure VI-22).

```
[for (pin: InputPort | a.inputPort)]
input[pin.toStringPort()/] = self.peek(self."[pin.toStringPort()/]");
[/for]
```

Figure VI-22 : Récupération des valeurs présentes sur chaque *InputPort*

Ces valeurs sont susceptibles de faire l'objet de tests. Pour chaque port est créée une variable avec composée du mot « input » suivi du nom du port, cette variable reçoit le résultat de la méthode *peek* appelée sur le port.

Ensuite, le template de génération de la fonction de transition externe peut être créé, il suffit juste de récupérer, pour chaque règle, la valeur présente sur le ou les ports d'entrée et de l'inclure dans les tests, en invoquant le template `toStringCond()` qui, grâce au polymorphisme, sera utilisé sous sa forme destinée aux InputPortComparison. On suppose donc que pour chaque DeltaExtRule, il ne peut y avoir qu'une seule InputPortComparison. Ce template est montré sur la Figure VI-23.

```
[for (dext: DeltaExtRule | a.contains_external_transition_function.is_defined_by_deltaext_rules)]
if input[dext.tests_input_event.toStringCond()/]:
    if [dext.toStringMultipleCond()/]:
        [dext.toStringAct()/]
[/for]
```

Figure VI-23 : Génération du code de la fonction *DeltaExt*

f) *Générer la fonction de sortie*

Enfin, nous présentons le template destiné à afficher la fonction de sortie. Encore une fois, il faudra effectuer un test sur une ou plusieurs variables d'état. L'action associée à ce test a été traitée dans le template `toStringAct()` : il s'agit d'envoyer un message sur un port de sortie, renseigné dans le modèle source au niveau des *OutputAction*.

Ce template est créé de manière analogue aux templates précédents.

```
 if [for (lambda: LambdaRule | a.contains_lambda_function.is_defined_by_lambda_rules)
separator ('        elif ')] [lambda.toStringMultipleCond()/]:
     self.poke(self.[lambda.sends_message.port.toStringPort()/],[lambda.toStringAct()/])
 [/for]
 [/if]
```

Figure VI-24 : Génération du code de la fonction *Lambda*

6.2.7. Instanciation des modèles et appel au SE PyDEVS

```
[template public generateExperiment(aDEVSModel : DEVSModel)]
[comment ce template génère un fichier .py à part, destiné à lancer la simulation/]
[file (aDEVSModel.name.concat('ExperimentAutoGen.py'), false, 'ISO-8859-1')]
# -*- coding: iso-8859-15 -*-
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
# [aDEVSModel.name.concat('ExperimentAutoGen.py')/].py --- Template for PythonDEVS experiments
#                      --------------------------------
#                               November 2005
#                              Hans Vangheluwe
#                        McGill University (Montréal)
#                      --------------------------------
#       created: 20/11/05
# last modified: 20/11/05
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##

# Add the directory where pydevs lives to Python's import path
import sys
import os.path
sys.path.append(os.path.expanduser('~/src/projects/PythonDEVS/'))

# Import code for model simulation:
import pydevs
from pydevs.infinity import *
from pydevs.simulator import *

#    ======================================================================    #

# Import the model to be simulated
from [aDEVSModel.name.concat('AutoGen')/] import *

#    ======================================================================    #

# 1. Instantiate the (Coupled or Atomic) DEVS at the root of the
#  hierarchical model. This effectively instantiates the whole model
#  thanks to the recursion in the DEVS model constructors (__init__).
#
[aDEVSModel.name.toLower()/] = [aDEVSModel.name/](name="[aDEVSModel.name.toLower()/]")

#    ======================================================================    #

# 2. Link the model to a DEVS Simulator:
#  i.e., create an instance of the 'Simulator' class,
#  using the model as a parameter.
sim = Simulator([aDEVSModel.name.toLower()/])
```

Figure VI-25 : Génération du fichier *ExperimentAutoGen.py*

Le fichier *.py* que permettent de générer les templates précédents constitue le fichier principal que nous évoquions en 6.2.1.b). Il contient toutes les informations concernant les modèles, les couplages, les fonctions comportementales. Toutefois, il n'est pas utilisable tel quel avec PyDEVS : il nous faut pour cela générer un autre fichier *.py* lié au fichier principal, et contenant quelques informations à propos de la simulation. C'est ce fichier qui devra être exécuté par la console IDLE de Python.

La génération du code correspondant à ce fichier ne pose pas de problèmes particuliers. Le code complet du template (avec en l'occurrence ce qui concerne les conditions d'arrêt de la simulation) figure en annexe 7. Il est contenu dans le même fichier .mtl que pour le fichier principal, et nous en donnons ici un bref aperçu, contenant seulement les informations textuelles variables (Figure VI-25).

Le but est simple, nous devons :

- donner un nom à ce fichier : nous concaténons la chaîne « ExperimentAutoGen » au nom du modèle racine, c'est à dire le modèle qui n'a pas d'ancêtres, et qui a donné son nom au fichier principal (ligne 3),
- générer le code correspondant à un import du fichier *.py* principal créé précédemment (ligne 29),
- instancier le modèle DEVS racine de ce même fichier (ligne 37),
- instancier la classe Simulator en lui passant ce modèle racine en paramètre (ligne 44).

6.3. Exemples de génération de code et validation par la simulation

Cette section est dédiée à la mise en œuvre des templates vus précédemment génération de code et à la validation des simulations associées aux modèles, à partir du fichier *.mtl* décrit précédemment, et illustre le fait que l'on rend nos modèles pleinement productifs, et ce de manière totalement automatique.

Nous montrons ici que n'importe quel modèle conforme à MetaDEVS peut être simulé, cela inclut donc les modèles qui :

- ont été créés selon les spécifications du méta-modèle MetaDEVS,
- étaient définis, à l'origine, dans d'autres formalismes et qui ont été transformés vers MetaDEVS (comme nous l'avons montré dans le chapitre V),
- sont des modèles couplés contenant des sous-modèles constituant une combinaison des deux possibilités précédentes.

Le rôle de pivot de MetaDEVS prend donc ici tout son sens.

Pour chacun des exemples suivants, nous considérons que le fichier de simulation (6.2.7) a toujours été correctement généré, et nous nous concentrons exclusivement sur le code Python du fichier principal (notamment les fonctions atomiques) ainsi que sur le résultat de la simulation. Nous faisons de même pour l'en-tête des fichiers. Le code Python (volontairement débarrassé de la plupart des commentaires) sera donné sous forme de captures d'écran prises sous Notepad++ afin de bénéficier, entre autres de la coloration syntaxique, et le résultat des simulations sera donné sous forme de captures d'écran de la console Python.

6.3.1. Génération du code d'un modèle atomique : feu tricolore

Reprenons en guise de premier exemple le modèle de feu tricolore BasicDEVS dont nous avons donné la définition dans le chapitre V. Ce modèle avait été ensuite transformé en un modèle se conformant à MetaDEVS. Nous pouvons donc lui appliquer une transformation M2T respectant la définition de transformation donnée tout au long de ce chapitre, pour le rendre productif.

Le résultat de cette transformation est la création de deux fichiers : ***crosswalkExperimentAutoGen.py*** et ***crosswalkAutoGen.py***. Examinons le contenu de ce dernier.

a) *Classe d'encapsulation de l'état*

Cette classe est montrée sur la Figure VI-26.

```
class crosswalkEncapState:

    def __init__(self, StateSet):
        """Constructor (parameterizable).
        """

        self.StateSet=StateSet

    def get(self):
        return {'StateSet': self.StateSet}

    def __str__(self):
        return " StateSet = "+ str(self.StateSet)
```

Figure VI-26 : Classe d'encapsulation de l'état du feu tricolore

On peut voir que son unique variable d'état est nommée correctement, que la méthode *get()* renvoie un dictionnaire composé d'une unique clef (correspondant à l'unique variable d'état) à laquelle est associée une valeur. Enfin, la fonction d'affichage *str()* est également correcte : elle renvoie une chaîne de caractères composée du nom de la variable d'état et de sa valeur courante. Cette classe d'encapsulation de l'état étant relativement facile à générer par rapport à d'autres parties de code, notamment les fonctions comportementales, que pour les exemples suivants qui ne contiennent qu'une seule variable d'état, nous considèrerons qu'elle a toujours été correctement générée.

b) *Constructeur de la classe principale (modèle atomique)*

La classe principale encapsulant le modèle atomique se compose d'un constructeur (contenant des variables) et de fonctions comportementales, vues en c).

D'après le code du constructeur (Figure VI-27), chaque instance de *crosswalk* contient :

- une variable *elapsed* qui sera utilisée par le simulateur PyDEVS pour symboliser un décalage dans le temps écoulé (par défaut, elle sera toujours égale à 0),
- un port de sortie (déclaré par un appel à la fonction PyDEVS *addOutPort()*),
- une variable *state* qui contient une instance de la classe qui encapsule l'état (appelée ici *crosswalkEncapState*) . Cette instance est créée à partir de la valeur initiale de la variable d'état, passée en paramètre.

```
class crosswalk(AtomicDEVS):

    def __init__(self, name=None):
        """Constructor (parameterizable).
        Giving a name (string) to a model is not mandatory
        (a globally unique name "A<i>" is assigned by default).
        If a name is given, the simulation trace becomes far more
        readable.
        """

        AtomicDEVS.__init__(self, name)

        self.state = crosswalkEncapState("green")

        self.elapsed = 0

        self.OUTPUTCROSSWALK = self.addOutPort(name="OUTPUTCROSSWALK")
```

Figure VI-27 : Constructeur de la classe *crosswalk*

Conformément à notre template, l'état initial est correctement spécifié, ainsi que les ports du modèle (ici, uniquement un port de sortie). Chaque instance de *crosswalk* contient en outre des fonctions comportementales, que nous détaillons à présent.

c) ***Fonctions de la classe principale (fonctions comportementales)***

Penchons-nous, en premier lieu, sur le code des fonctions de transition. Ce dernier est montré sur la capture d'écran de la Figure VI-28.

La fonction de transition externe est générée mais laissée vide, ce qui est normal car le modèle ne comporte pas de port d'entrée et ne peut donc pas réagir aux évènements extérieurs.

La fonction de transition interne se compose d'une double récupération de l'état (résultat de l'appel du template `retrieveState()`), avec la variable *statedico* qui sert à effectuer les tests, et la variable *state* qui sert à modifier l'état lui-même. Le corps de cette fonction est conforme aux spécifications du modèle MetaDEVS (issu de la transformation du modèle initial BasicDEVS vers MetaDEVS) : de l'état « green » on passe à l'état « orange », de l'état « orange » on passe à l'état « red », de l'état « red » on passe à l'état « green ». Dans tous les cas, après modification, l'état est retourné.

La fonction de sortie est présentée sur la Figure VI-29. L'état est encore une fois récupéré. Pour chacune des trois valeurs de variable d'état possibles, cette fonction envoie sur le port de sortie « OUTPUTCROSSWALK » le message initialement défini dans le modèle BasicDEVS. Ce message annonce passage d'un état à l'autre. Le code de cette fonction est lui aussi conforme aux spécifications données dans le modèle original.

```
def extTransition(self):
    """External Transition Function."""

def intTransition(self):
    """Internal Transition Function.
    """

    statedico = self.state.get()
    state = self.state

    if statedico['StateSet'] == "green":
        state.StateSet="orange"
        return state
    elif statedico['StateSet'] == "orange":
        state.StateSet="red"
        return state
    elif statedico['StateSet'] == "red":
        state.StateSet="green"
        return state
    else :
        raise DEVSException(\
        "unknown state <%s> in crosswalk internal transition function"\
        % state)
```

Figure VI-28 : Fonctions de transition de la classe *crosswalk*

```
def outputFnc(self):
    """Output Function.
    """

    statedico = self.state.get()
    state = self.state

    if statedico['StateSet'] == "green":
        self.poke(self.OUTPUTCROSSWALK,"greenToOrange!")
    elif statedico['StateSet'] == "orange":
        self.poke(self.OUTPUTCROSSWALK,"orangeToRed!")
    elif statedico['StateSet'] == "red":
        self.poke(self.OUTPUTCROSSWALK,"redToGreen!")
```

Figure VI-29 : Fonction de sortie de la classe *crosswalk*

Enfin, nous présentons le code qui a été généré pour la fonction d'avancement du temps (Figure VI-30).

Après la récupération de l'état courant, cette fonction teste ce dernier : s'il est à « green », la fonction renvoie la valeur 25, s'il est à « orange », la fonction renvoie la valeur 5 et enfin, s'il est à « red », la fonction renvoie la valeur 30. Ces valeurs sont en accord avec les valeurs des *TARule* du modèle passé en entrée, elles-mêmes en accord avec les valeurs des attributs *max_duration* du modèle original BasicDEVS.

```
def timeAdvance(self):
    """Time-Advance Function.
    """

    statedico = self.state.get()
    state = self.state

    if statedico['StateSet'] == "green":
        return 25

    elif statedico['StateSet'] == "orange":
        return 5

    elif statedico['StateSet'] == "red":
        return 30

    else :
        raise DEVSException(\
        "unknown state <%s> in crosswalk time advance function"\
        % state)
```

Figure VI-30 : Fonction d'avancement du temps de la classe *crosswalk*

d) ***Simulation***

```
Current Time:       0.00 _____________________________


    INITIAL CONDITIONS in model <crosswalk>
      Initial State:  StateSet = green
      Next scheduled internal transition at time 25

Current Time:      25.00 _____________________________


    INTERNAL TRANSITION in model <crosswalk>
      New State:  StateSet = orange
      Output Port Configuration:
        port <OUTPUTCROSSWALK>: greenToOrange!
      Next scheduled internal transition at time 30

    ROOT DEVS <crosswalk> Output Port Configuration:
        port <OUTPUTCROSSWALK>: greenToOrange!

Current Time:      30.00 _____________________________


    INTERNAL TRANSITION in model <crosswalk>
      New State:  StateSet = red
      Output Port Configuration:
        port <OUTPUTCROSSWALK>: orangeToRed!
      Next scheduled internal transition at time 60

    ROOT DEVS <crosswalk> Output Port Configuration:
        port <OUTPUTCROSSWALK>: orangeToRed!

Current Time:      60.00 _____________________________


    INTERNAL TRANSITION in model <crosswalk>
      New State:  StateSet = green
      Output Port Configuration:
        port <OUTPUTCROSSWALK>: redToGreen!
      Next scheduled internal transition at time 85

    ROOT DEVS <crosswalk> Output Port Configuration:
        port <OUTPUTCROSSWALK>: redToGreen!
```

Figure VI-31 : Résultat partiel de la simulation du modèle de feu tricolore sous PyDEVS

Il nous reste maintenant à vérifier la validité de notre modèle MetaDEVS, provenant d'un modèle BasicDEVS, en examinant les résultats que nous fournit sa simulation. Voici ce que l'on obtient (Figure VI-31) lorsque l'on exécute dans la console IDLE le fichier ***crosswalkExperimentAutoGen.py***.

Nous ne faisons figurer que les premiers changements d'état, les autres se répétant de la même manière jusqu'à la fin de la simulation. À t=0, le système se trouve dans l'état initial « green », et la prochaine transition est programmée à t=25 (conformément aux données fournies par la fonction d'avancement du temps).

À t=25, l'état « green » arrive à expiration, une transition interne va donc être déclenchée, vers le prochain état : « orange ». Une sortie est envoyée sur le port « OUTPURCROSSWALK », elle contient le message « greenToOrange ! ».

La prochaine transition est programmée à t=30, soit 5 unités de temps plus tard (ce qui correspond à la durée de vie de l'état « orange »), et fera passer le modèle de l'état « orange » à l'état « red », toujours en envoyant sur le port de sortie un message indiquant que le changement d'état s'effectue.

La transition suivante est prévue à t=60, soit 30 unités de temps plus tard (durée de vie de l'état « red »). Un message est envoyé sur le port de sortie, et la transition de l'état « red » vers l'état « green » est effectuée. La simulation continue ainsi de suite.

e) *Génération du code du modèle de feu tricolore amélioré*

Le modèle précédent, très simple, avait fait l'objet d'une amélioration dans le chapitre V, ce qui lui permettait, grâce à l'ajout d'un port d'entrée, de pouvoir réagir aux évènements provenant du monde extérieur. Nous avions alors créé dans le modèle BasicDEVS des *EventTransition* qui, lors de la transformation M2M vers MetaDEVS, étaient devenues des *DeltaExtRule*.

Générons le code correspondant à ce modèle (Figure VI-32). Il est évidemment très proche du code du modèle précédent, il nous faut juste vérifier que le port d'entrée est bien créé et que la fonction *DeltaExt* a cette fois-ci été renseignée avec des *Rule*. La capture d'écran de la Figure VI-32 montre :

- la dernière partie du code du constructeur de la classe *crosswalk*, sur laquelle se trouvent les instructions de création de ports, on voit donc bien qu'au port de sortie initial s'est ajoutée l'instruction de création d'un port d'entrée « IN1 »;
- le corps de la fonction DeltaExt. À la réception du message « STOP » sur le port d'entrée « IN1 », le modèle passera à l'état « orange » s'il se trouve dans l'état « green » ou l'état « orange », et à l'état « red » s'il se trouve déjà dans l'état « red ».

Bien entendu, il serait totalement inutile de simuler ce modèle tel quel, car, en l'absence d'évènements externes, nous obtiendrions exactement la même trace de simulation que pour le modèle précédent. Nous effectuerons par contre une simulation utilisant ce modèle dans la section suivante, après l'avoir intégré dans un modèle couplé.

```
    self.IN1 = self.addInPort(name="IN1")
    self.OUTPUTCROSSWALK = self.addOutPort(name="OUTPUTCROSSWALK")


def extTransition(self):
    """External Transition Function."""

    inputIN1 = self.peek(self.IN1)

    statedico = self.state.get()
    state = self.state

    if inputIN1 == "STOP":
        if statedico['StateSet'] == "green":
            state.StateSet="orange"
            return state
    if inputIN1 == "STOP":
        if statedico['StateSet'] == "orange":
            state.StateSet="orange"
            return state
    if inputIN1 == "STOP":
        if statedico['StateSet'] == "red":
            state.StateSet="red"
            return state
    else :
        raise DEVSException(\
        "unknown state <%s> in crosswalk external transition function"\
        % state)
```

Figure VI-32 : Fonction de transition externe et création des ports du modèle *crosswalk* amélioré

6.3.2. Génération du code d'un modèle couplé : carrefour

Afin d'utiliser au mieux le modèle de feu rouge créé précédemment, nous nous proposons de modéliser le comportement d'un piéton désirant traverser, et appuyant sur un bouton qui commande le feu tricolore. Nous intégrons ce modèle de piéton avec le modèle *crosswalk* précédent, au sein d'un modèle couplé MetaDEVS, dont nous effectuons la simulation.

Cet exemple illustre la possibilité de simuler conjointement un modèle créé directement en MetaDEVS et un modèle provenant à l'origine de BasicDEVS, préalablement transformé en MetaDEVS via un procédé M2M. C'est donc une démonstration de la compatibilité externe de DEVS qui est rendue possible grâce à une interopérabilité au niveau des modèles, ainsi que de la réutilisabilité des modèles permise par MetaDEVS.

a) *Description MetaDEVS du modèle de bouton*

Ce modèle est en fait un modèle atomique MetaDEVS décrivant un générateur très simple, il modélise l'appui d'un piéton sur un bouton STOP, à intervalles réguliers. Comme ce modèle est semblable aux générateurs précédemment étudiés, nous le décrivons uniquement de manière textuelle, et renvoyons le lecteur aux annexes de ce document afin qu'il puisse en examiner la définition complète MetaDEVS.

Ce modèle se compose d'un bouton, pouvant être à l'état « IDLE » ou à l'état « PUSHED ». La variable d'état est nommée « StateSet » (par analogie avec le nommage de

la variable d'état dans BasicDEVS). Le modèle ne possède pas de port d'entrée mais dispose d'un port de sortie « outbutton ». Initialement, le bouton est dans l'état « IDLE ». La durée de vie de cet état est de 87 unités de temps. De cet état, le bouton passe à l'état « PUSHED » qui est un état transitoire (ou *transient state*) : sa durée de vie est égale à 0. La fonction de sortie envoie le message « STOP » sur le port de sortie du modèle, et le modèle revient à l'état « IDLE » pour une durée de 87 unités de temps. Nous avons nommé ce modèle atomique *MetaDEVS_button*.

b) ***Génération du code de « MetaDEVS_button »***

Nous montrons ici le code des fonctions DeltaInt, TimeAdvance et Lambda du modèle de bouton MetaDEVS que nous venons de décrire (Figure VI-33).

```
def intTransition(self):
    """Internal Transition Function.
    """
    if statedico['StateSet'] == "IDLE":
        state.StateSet="PUSHED"
        return state
    elif statedico['StateSet'] == "PUSHED":
        state.StateSet="IDLE"
        return state
    else :
        raise DEVSException(\
        "unknown state <%s> in button internal transition function"\
        % state)

def outputFnc(self):
    """Output Function.
    """
    if statedico['StateSet'] == "IDLE":
        self.poke(self.OUTBUTTON,"STOP")

def timeAdvance(self):
    """Time-Advance Function.
    """
    if statedico['StateSet'] == "IDLE":
        return 87

    elif statedico['StateSet'] == "PUSHED":
        return 0

    else :
        raise DEVSException(\
        "unknown state <%s> in button time advance function"\
        % state)
```

Figure VI-33 : Fonctions comportementales du modèle *MetaDEVS_button*

Ce modèle étant un générateur, il possède une fonction de transition externe vide. Les deux instructions générées par le template `retrieveState()`, qui sont `statedico = self.state.get()` et `state = self.state` sont bien entendu présentes dans le code originalement généré, au début de chaque fonction comportementale, mais nous les avons juste supprimées pour la capture d'écran, par souci de simplification. Le code généré est en adéquation avec notre description textuelle faite en a). Nous allons maintenant combiner le modèle de bouton avec le modèle de feu tricolore afin de créer un modèle couplé.

c) ***Création d'un modèle couplé de carrefour***

Notre but ici est donc de modéliser un carrefour, incarné d'une part par un modèle représentant le feu tricolore capable de réagir à des évènements, et par un modèle représentant un bouton sur lequel appuie un piéton d'autre part.

Pour bénéficier d'une génération de code automatique et complète, il est impossible de réutiliser le code précédent. Nous devons en effet rester dans un esprit MDA, c'est-à-dire utiliser le plus longtemps possible des représentations des modèles indépendantes des plateformes, la génération de code n'intervenant qu'à la fin de la modélisation.

Le code de simulation des modèles atomiques donné 6.3.1 et 6.3.1.e) d'une part, et en b) d'autre part, devra par conséquent être complétement régénéré et intégré dans un fichier *.py* tenant compte de l'existence d'un modèle couplé.

Bien évidemment, les modèles atomiques qui se trouveront dans le fichier *.py* que nous allons générer possèderont exactement le même code de simulation que celui que nous avons présenté dans les diverses captures d'écran précédentes.

Les étapes pour la création d'un modèle couplé, à partir de modèles existants, sous Eclipse EMF, sont très simples :

1. **Créer un modèle couplé MetaDEVS vide, et lui donner un nom.** Ici « crosswalkSystem »
2. **Importer les modèles (atomiques ou couplés) nécessaires au fonctionnement de ce modèle couplé.** Ici il s'agit du modèle atomique *MetaDEVS_button*, créé directement avec MetaDEVS, et du modèle *crosswalk*, modèle MetaDEVS spécifié à l'origine sous forme de modèle BasicDEVS.
3. **Lui créer éventuellement des ports.** Ici, ce n'est pas nécessaire.
4. **Spécifier le ou les couplages.** Ici, il faut lier le port de sortie du bouton au port d'entrée du feu tricolore, il s'agit d'un couplage interne (IC)

La Figure VI-34 montre le résultat de la création de ce modèle couplé MetaDEVS.

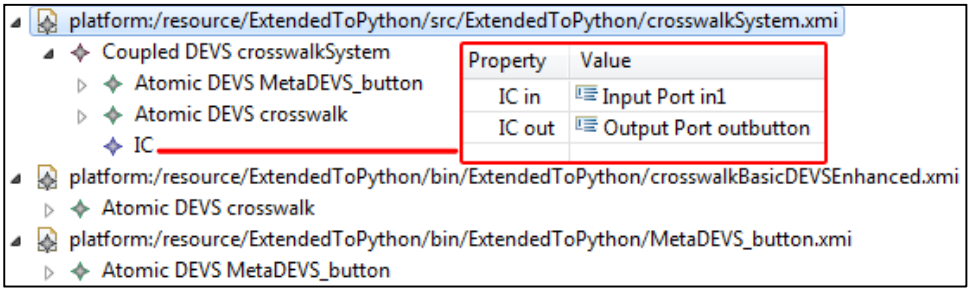


Figure VI-34 : Capture d'écran EMF reconstituée du modèle couplé de carrefour MetaDEVS

On y voit le modèle *crosswalkSystem* composé de ses deux sous-modèles atomiques *MetaDEVS_button* et *crosswalk*, ainsi que d'une fonction de couplage interne IC dont le détail est donné dans l'encadré : nous avons fait correspondre le port de sortie du bouton « outbutton » avec le port d'entrée du feu tricolore « in1 ». Nous voyons également sur cette

capture d'écran les modèles utilisés pour créer le modèle couplé, avec leur chemin d'accès complet.

d) ***Génération de code***

Nous pouvons maintenant passer à la génération du code de simulation correspondant à ce modèle couplé. Le résultat de cette génération est la création de deux fichiers, ***crosswalkSystemExperimentAutoGen.py*** et ***crosswalkSystemAutoGen.py***. Encore une fois, nous nous concentrons sur ce dernier. Comme nous l'avons établi précédemment, le code des modèles atomiques sera exactement le même que celui que nous avons déjà présenté, ce qui est tout à fait normal. Nous ne faisons donc figurer ici que la partie du code du fichier ***crosswalkSystemAutoGen.py*** qui concerne l'instance de la classe contenant décrivant un modèle couplé (Figure VI-35).

Ce modèle couplé ne possède pas de ports, conformément à notre spécification. Les sous-modèles sont correctement déclarés, ainsi que l'unique fonction de couplage (IC), générée grâce au template `declareCouplage()` présenté en 6.2.4.b).

```
class crosswalkSystem(CoupledDEVS):
    """Sample Coupled-DEVS descriptive class: compulsory material."""

    def __init__(self, name=None):
        """Constructor """

        # Always call parent class' constructor FIRST:
        CoupledDEVS.__init__(self, name)

    # PORTS:

    # SUB-MODELS:
        self.metadevs_button = self.addSubModel(MetaDEVS_button(name="metadevs_button"))
        self.crosswalk = self.addSubModel(crosswalk(name="crosswalk"))

    # COUPLINGS:
        self.connectPorts(self.metadevs_button.OUTBUTTON, self.crosswalk.IN1) # INTERNAL COUPLING
```

Figure VI-35 : Génération du code décrivant les couplages

e) ***Simulation***

Nous pouvons maintenant simuler notre modèle couplé et voir s'il se comporte conformément à nos attentes. La différence avec 6.3.1.d) sera que lorsque le piéton appuiera sur le bouton à t=87, le feu tricolore réagira à cet évènement et passera soit à l'état « orange » s'il est dans les états « orange » ou « green », soit repassera à l'état « red » s'il se trouve déjà dans cet état. La trace de simulation présentée sur la Figure VI-36 montre l'état initial du système au début de la simulation : on y voit en effet que le prochain évènement prévu pour le sous–modèle du feu tricolore est à t=25 et que celui prévu pour le bouton est à t=87.

```
Current Time:       0.00

    INITIAL CONDITIONS in model <crosswalksystem.metadevs_button>
      Initial State:  StateSet = IDLE
      Next scheduled internal transition at time 87

    INITIAL CONDITIONS in model <crosswalksystem.crosswalk>
      Initial State:  StateSet = green
      Next scheduled internal transition at time 25
```

Figure VI-36 : Début de la simulation : modèle « *crosswalk* »

```
Current Time:      85.00 _____________________________________________

    INTERNAL TRANSITION in model <crosswalksystem.crosswalk>
      New State:  StateSet = orange
      Output Port Configuration:
        port <OUTPUTCROSSWALK>: greenToOrange!
      Next scheduled internal transition at time 90

Current Time:      87.00 _____________________________________________

    INTERNAL TRANSITION in model <crosswalksystem.metadevs_button>
      New State:  StateSet = PUSHED
      Output Port Configuration:
        port <OUTBUTTON>: STOP
      Next scheduled internal transition at time 87

    EXTERNAL TRANSITION in model <crosswalksystem.crosswalk>
      Input Port Configuration:
        port <IN1>: STOP
      New State:  StateSet = orange
      Next scheduled internal transition at time 92

Current Time:      87.00 _____________________________________________

    INTERNAL TRANSITION in model <crosswalksystem.metadevs_button>
      New State:  StateSet = IDLE
      Output Port Configuration:
        port0: NoEvent
      Next scheduled internal transition at time 174
```

Figure VI-37 : Réception d'un évènement extérieur à *t*=87 par le modèle « *crosswalk* »

L'examen, un peu plus loin dans le journal de simulation, de ce qui se produit à *t*=87 est en totale adéquation avec le comportement attendu du système (Figure VI-37). Le feu tricolore est depuis *t*=85 dans l'état « orange » et aurait dû passer à l'état « red ». La réception de l'évènement d'appui sur le bouton par le piéton a pour effet de faire « redémarrer » le système dans l'état « orange » pour une durée de 5 unités de temps.

6.3.3. Génération du code d'un modèle couplé : générateur de lettres et automate

Nous allons maintenant montrer que l'on peut simuler au sein d'un modèle couplé MetaDEVS des modèles provenant chacun d'un formalisme différent, autre que MetaDEVS. L'idée ici est de combiner le générateur de lettres et l'automate présentés dans le chapitre précédent et de fournir une simulation de ce système, donc de tester si l'automate reconnaît, ou pas, un mot donné. Ceci montrera que des formalismes externes à DEVS, ne disposant pas de simulateur, peuvent être simulés en tant que modèles DEVS. Nous allons tout d'abord créer un modèle couplé comme spécifié en 6.3.2.c), générer le code PyDEVS correspondant, puis effectuer une simulation. Le résultat de cette génération est la création de deux fichiers :

- ***coupledGeneratorandFSMExperimentAutoGen.py***
- ***coupledGeneratorandFSMAutoGen.py***

Comme nous l'avons fait dans les sous-sections suivantes, nous considérons que le fichier de simulation a correctement été généré, et nous ne présentons que le fichier correspondant à la définition des modèles DEVS.

Nous présentons ce modèle couplé différemment par rapport au modèle couplé précédent, en donnant d'abord une vue d'ensemble de ce dernier, puis son code correspondant et ensuite le code des deux modèles atomiques qui le composent, et, enfin, en simulant le tout.

a) ***Vue d'ensemble du modèle couplé***

Ce modèle MetaDEVS couplé, que nous avons nommé « coupledGeneratorandFSM », se compose de deux modèles issus eux-mêmes de modèles provenant de deux formalismes différents : le modèle atomique MetaDEVS « *wordGenerator* », venant d'un modèle BasicDEVS, et le modèle atomique MetaDEVS « *enigmeMIU* », venant d'un modèle FSM. En encadré figure le détail de la fonction de couplage interne IC que nous avons rajoutée, elle lie le port de sortie « wordGeneratorPort » du générateur au port d'entrée « FSMReadInputPort » de l'automate.

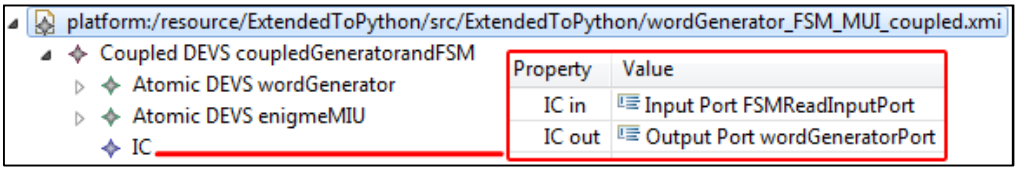


Figure VI-38 : Le modèle couplé générateur de lettres+automate

Nous donnons maintenant dans les sous-sections b), c) et d) un aperçu du code contenu dans *coupledGeneratorandFSMAutoGen.py*.

b) ***Génération de code : le modèle couplé***

Le code généré est montré par la capture d'écran de la Figure VI-39. On y voit que les sous-modèles sont correctement déclarés et nommés, et que le couplage interne est cohérent. Ce code PyDEVS correspondant au modèle couplé est conforme aux spécifications du modèle MetaDEVS.

```
class coupledGeneratorandFSM(CoupledDEVS):
    """Sample Coupled-DEVS descriptive class: compulsory material.
    """
    def __init__(self, name=None):
        """Constructor """

        # Always call parent class' constructor FIRST:
        CoupledDEVS.__init__(self, name)

    # PORTS:
    #  Note: at the root level, the Coupled DEVS will often
    #  not have any InPorts nor OutPorts (i.e., be "autonomous").

    # SUB-MODELS:
        self.wordgenerator = self.addSubModel(wordGenerator(name="wordgenerator"))
        self.enigmemiu = self.addSubModel(enigmeMIU(name="enigmemiu"))

    # COUPLINGS:

    # INTERNAL COUPLING
        self.connectPorts(self.wordgenerator.WORDGENERATORPORT, self.enigmemiu.FSMREADINPUTPORT)
```

Figure VI-39 : Génération du code décrivant les couplages

Nous traitons dans cette sous-section du code du modèle atomique « *wordGenerator* ». La Figure VI-40 montre le code généré de ce modèle atomique. Sur la moitié supérieure de la

figure se trouve le code de la classe d'encapsulation de l'état. Comme le modèle provient d'un modèle BasicDEVS, la variable d'état se nomme « StateSet ».

c) ***Génération de code : le modèle atomique* « wordGenerator »**

```
class wordGeneratorEncapState:

    def __init__(self, StateSet):
        """Constructor"""

        self.StateSet=StateSet

    def get(self):
        return {'StateSet': self.StateSet}

    def __str__(self):
        return " StateSet = "+ str(self.StateSet)

class wordGenerator(AtomicDEVS):

    def __init__(self, name=None):
        """Constructor"""

        AtomicDEVS.__init__(self, name)

        self.state = wordGeneratorEncapState("1")

        self.elapsed = 0

        self.WORDGENERATORPORT = self.addOutPort(name="WORDGENERATORPORT")
```

Figure VI-40 : Classe d'encapsulation et constructeur du modèle « *wordGenerator* »

La seconde moitié de la figure montre le code du constructeur de la classe du modèle. L'état initial est spécifié correctement (état « 1 ») et le port de sortie « WORDGENERATORPORT » a bien été ajouté en tant qu'attribut.

Le code généré de la fonction de transition interne DeltaInt ne présente pas un grand intérêt. Cette fonction contient les règles de changement d'état, conformément au modèle d'origine BasicDEVS et donc au modèle source de type MetaDEVS. Elle décrit le passage de l'état « 1 » à l'état « 2 », du « 2 » au « 3 », et ainsi de suite jusqu'à l'état « 5 » qui va dans l'état « final ». DeltaExt est, quant à elle, laissée vide, car ce modèle est un modèle autonome de générateur.

La fonction d'avancement du temps (Figure VI-41) montre que la durée de vie des états de « 1 » à « 5 » est de 1 unité de temps. La durée de vie de l'état final, originalement fixée à 99999999 dans le modèle BasicDEVS et dans sa version MetaDEVS, a été remplacée par le mot clef reconnu par PyDEVS : `INFINITY`. Ceci a été fait au moyen du template `toStringTA()` présenté en 6.2.5.c). La fonction d'avancement du temps a donc été, elle aussi, correctement générée.

La Figure VI-42 représente une capture d'écran du code généré pour la fonction de sortie. Pour chaque état le modèle envoie une lettre sur son port de sortie « WORDGENERATORPORT », et ceci est conforme à la description du modèle faite avec BasicDEVS d'une part, et d'autre part avec le modèle MetaDEVS à partir duquel nous avons

généré ce code. Lors du passage de l'état « 1 » à l'état « 2 », le modèle enverra la lettre « m » en sortie, lors du passage de l'état « 2 » à l'état « 3 » la lettre « i », et ainsi de suite jusqu'au moment où il quittera l'état « 5 » pour l'état « final » en envoyant la lettre « m ». Le mot généré est donc « *miuim* » (c'est d'ailleurs un palindrome).

```
def timeAdvance(self):
    """Time-Advance Function."""

    if statedico['StateSet'] == "final":
        return INFINITY
    elif statedico['StateSet'] == "1":
        return 1
    elif statedico['StateSet'] == "2":
        return 1
    elif statedico['StateSet'] == "3":
        return 1
    elif statedico['StateSet'] == "4":
        return 1
    elif statedico['StateSet'] == "5":
        return 1
    else :
        raise DEVSException(\
        "unknown state <%s> in wordGenerator time advance function"\
        % state)
```

Figure VI-41 : Fonction d'avancement du temps du modèle « *wordgenerator* »

```
def outputFnc(self):
    """Output Function."""

    if statedico['StateSet'] == "1":
        self.poke(self.WORDGENERATORPORT,"m")
    elif statedico['StateSet'] == "2":
        self.poke(self.WORDGENERATORPORT,"i")
    elif statedico['StateSet'] == "3":
        self.poke(self.WORDGENERATORPORT,"u")
    elif statedico['StateSet'] == "4":
        self.poke(self.WORDGENERATORPORT,"i")
    elif statedico['StateSet'] == "5":
        self.poke(self.WORDGENERATORPORT,"m")
```

Figure VI-42 : Fonction de sortie du modèle « *wordgenerator* »

Note : La récupération de l'état courant a été volontairement été supprimée de ces captures d'écran pour faciliter la lecture : répétons encore une fois qu'elle est quand même systématiquement générée au début de chaque fonction comportementale DEVS.

d) ***Génération de code : le modèle atomique* « enigmeMIU »**

Examinons à présent le code généré pour l'automate que nous avions transformé en un modèle MetaDEVS dans le chapitre précédent. La Figure VI-43 montre le code de la classe d'encapsulation des états et le code du constructeur du modèle atomique.

L'unique variable d'état du modèle se nomme *FSMState*. Dans le constructeur du modèle atomique, l'état initial de l'automate (qui est passé au constructeur de la classe d'encapsulation) est bien « 1 », et les ports sont au nombre de 2 : un port d'entrée « FSMREADINPUTPORT » et un port de sortie « FSMREADOUTPUTPORT ». Pour

mémoire, la variable d'état et les ports, absents du formalisme FSM de base, avaient été rajoutés lors de la transformation M2M de ce FSM vers MetaDEVS.

Intéressons-nous maintenant au résultat de la génération de la fonction de transition externe DeltaExt. Elle doit correspondre à la spécification initiale de l'automate, car nous avons établi dans le chapitre V que lors d'une transformation M2M d'un FSM vers metaDEVS, chaque transition du FSM devait être transformée en *DeltaExtRule*, la lecture d'une lettre étant assimilable à l'arrivée d'un évènement d'entrée.

```
class enigmeMIUEncapState:

    def __init__(self, FSMState):
        """Constructor"""

        self.FSMState=FSMState

    def get(self):
        return {'FSMState': self.FSMState}

    def __str__(self):
        return " FSMState = "+ str(self.FSMState)

class enigmeMIU(AtomicDEVS):

    ###

    def __init__(self, name=None):
        """Constructor"""
        AtomicDEVS.__init__(self, name)

        self.state = enigmeMIUEncapState("1")

        self.elapsed = 0

        self.FSMREADINPUTPORT = self.addInPort(name="FSMREADINPUTPORT")
        self.FSMREADOUTPUTPORT = self.addOutPort(name="FSMREADOUTPUTPORT")
```

Figure VI-43 : Classe d'encapsulation et constructeur du modèle « *enigmeMIU* »

Le code de la fonction correspondant à DeltaExt (Figure VI-44) montre que : de l'état « 1 » la lecture de la lettre « m » fait passer à l'état « 2 », de l'état « 2 » la lecture de la lettre « i » fait passer à l'état « 3 », de l'état « 3 » la lecture de la lettre « u » fait passer à l'état « 2 », et enfin la lecture de la lettre « m » depuis l'état « 3 » fait boucler cet état sur lui-même. Cette fonction est par conséquent en tout point conforme à l'automate que nous avions présenté dans le chapitre I, puis transformé vers MetaDEVS dans le chapitre V.

L'automate ne possédait à l'origine aucune information portant sur d'éventuelles transitions internes. Or, nous avons discuté dans le chapitre V de la nécessité de reconnaître les états finaux, ce qui signifie qu'un mot a été lu (i.e. reconnu) par l'automate. Nous avions donc, dans la définition de la transformation FSMToMetaDEVS, opté pour une solution qui consiste à créer une transition interne pour chaque état final (dont la durée de vie est inférieure à l'infini), et ce afin de déclencher la fonction de sortie qui nous permet de reconnaître un état final. Rappelons que la durée de vie d'un état non final était fixée à une valeur assimilable à l'infini. La fonction de transition interne générée, montrée sur la Figure

VI-45, est conforme à la fonction DeltaInt qui avait été générée lors de la transformation du FSM vers MetaDEVS, dans le chapitre précédent.

```
def extTransition(self):
    """External Transition Function."""

    inputFSMREADINPUTPORT = self.peek(self.FSMREADINPUTPORT)

    if inputFSMREADINPUTPORT == "m":
        if statedico['FSMState'] == "1":
            state.FSMState="2"
            return state
    if inputFSMREADINPUTPORT == "i":
        if statedico['FSMState'] == "2":
            state.FSMState="3"
            return state
    if inputFSMREADINPUTPORT == "u":
        if statedico['FSMState'] == "3":
            state.FSMState="2"
            return state
    if inputFSMREADINPUTPORT == "m":
        if statedico['FSMState'] == "3":
            state.FSMState="3"
            return state
    else :
        raise DEVSException(\
        "unknown state <%s> in enigmeMIU external transition function"\
        % state)
```

Figure VI-44 : Fonction de transition externe du modèle « *enigmeMIU* »

```
def intTransition(self):
    """Internal Transition Function"""

    if statedico['FSMState'] == "2":
        state.FSMState="3"
        return state
    elif statedico['FSMState'] == "3":
        state.FSMState="3"
        return state
    else :
        raise DEVSException(\
        "unknown state <%s> in enigmeMIU internal transition function"\
        % state)
```

Figure VI-45 : Fonction de transition interne du modèle « *enigmeMIU* »

Chaque transition ayant pour cible un état final est décrite (transition de l'état « 2 » vers l'état « 3 », transition de l'état « 3 » vers lui-même). Il n'existe pas d'autre transition interne dans ce modèle, dont l'évolution est principalement conditionnée par la réception d'évènements (lettres) sur son port d'entrée.

Concernant la fonction d'avancement du temps, les états non finaux se voyaient dotés, lors de la transformation vers MetaDEVS, d'une valeur supérieure à 999999. Lors de la génération de code, ils auront donc une durée égale à l'infini (voir 6.2.5.c)). L'unique état final possède, quant à lui, une durée de vie de 4000 unités de temps. Le résultat de la génération du code de la fonction TimeAdvance se trouve sur la Figure VI-46.

```
def timeAdvance(self):
    """Time-Advance Function"""

    if statedico['FSMState'] == "1":
        return INFINITY
    elif statedico['FSMState'] == "2":
        return INFINITY
    elif statedico['FSMState'] == "3":
        return 4000
    else :
        raise DEVSException(\
        "unknown state <%s> in enigmeMIU time advance function"\
        % state)
```

Figure VI-46 : Fonction d'avancement du temps du modèle « *enigmeMIU* »

Enfin, il nous reste à vérifier que la fonction de sortie est conforme au modèle MetaDEVS de l'automate « *enigmeMIU* ». Cette fonction est montrée sur la Figure VI-47. Les états susceptibles de conduire à une sortie et donc à une reconnaissance du mot lu sont les états « 2 » et « 3 » : la fonction de sortie est donc correctement générée.

```
def outputFnc(self):
    """Output Function"""

    if statedico['FSMState'] == "2":
        self.poke(self.FSMREADOUTPUTPORT,"__the word has been recognized__")
    elif statedico['FSMState'] == "3":
        self.poke(self.FSMREADOUTPUTPORT,"__the word has been recognized__")
```

Figure VI-47 : Fonction de sortie du modèle « *enigmeMIU* »

Note : En pratique, l'état « 2 » ayant une durée de vie infinie, il n'arrivera jamais à expiration, donc il n'y aura jamais de transition interne de l'état « 2 » vers l'état « 3 », et par conséquent le premier cas de la fonction de sortie, qui est également le premier cas de la fonction de transition interne (si l'état est égal à « 2 ») ne se réalisera jamais. Ceci est dû à une contrainte de programmation décrite dans le chapitre précédent (lors du passage M2T d'un FSM à MetaDEVS)

e) ***Simulation***

Passons à présent à la phase de simulation de notre modèle couplé, qui devra confirmer que l'automate est bien capable de reconnaître le mot « miuim ».

Le début de la simulation (Figure VI-48) montre que les conditions initiales à t=0 (moitié supérieure de la figure) des deux modèles atomiques sont correctes, l'automate est bien dans l'état « 1 » et y restera jusqu'à une durée infinie (à moins qu'un évènement extérieur ne vienne le perturber), et le générateur de lettres est lui aussi dans son état initial « 1 » qu'il quittera dans 1 unité de temps. À t=1 (moitié inférieure de la figure) , le générateur de lettres envoie la lettre « m » sur son port de sortie, elle est reçue par l'automate qui effectue une transition externe vers son état « 2 ».

La simulation se déroule comme prévu jusqu'à t=5 (Figure VI-49). À cet instant (moitié supérieure de la figure), le générateur de lettres envoie sa dernière lettre (la lettre « m ») sur son port de sortie, avant de passer dans l'état « final » dont la durée de vie est

infinie. Le générateur cesse donc d'émettre des lettres. Suite à la réception de « m », l'automate passe dans l'état « 3 », et attend jusqu'à t=4005.

```
Current Time:       0.00 ____________________________________________

    INITIAL CONDITIONS in model <coupledgeneratorandfsm.wordgenerator>
      Initial State:  StateSet = 1
      Next scheduled internal transition at time 1

    INITIAL CONDITIONS in model <coupledgeneratorandfsm.enigmemiu>
      Initial State:  FSMState = 1
      Next scheduled internal transition at time +INFINITY

Current Time:       1.00 ____________________________________________

    INTERNAL TRANSITION in model <coupledgeneratorandfsm.wordgenerator>
      New State:  StateSet = 2
      Output Port Configuration:
        port <WORDGENERATORPORT>: m
      Next scheduled internal transition at time 2

    EXTERNAL TRANSITION in model <coupledgeneratorandfsm.enigmemiu>
      Input Port Configuration:
        port <FSMREADINPUTPORT>: m
      New State:  FSMState = 2
      Next scheduled internal transition at time +INFINITY
```

Figure VI-48 : Début de la simulation : enigmeMIU

À t=4005 (moitié inférieure de la figure), en l'absence de réception d'évènements extérieurs, l'état « 3 » (état final) de l'automate arrive en fin de vie, il produit donc une sortie sur le port « FSMREADOUTPUTPORT » disant que le mot a été reconnu. La simulation vient donc valider notre modèle d'automate.

```
Current Time:       5.00 ____________________________________________

    INTERNAL TRANSITION in model <coupledgeneratorandfsm.wordgenerator>
      New State:  StateSet = final
      Output Port Configuration:
        port <WORDGENERATORPORT>: m
      Next scheduled internal transition at time +INFINITY

    EXTERNAL TRANSITION in model <coupledgeneratorandfsm.enigmemiu>
      Input Port Configuration:
        port <FSMREADINPUTPORT>: m
      New State:  FSMState = 3
      Next scheduled internal transition at time 4005

Current Time:    4005.00 ____________________________________________

    INTERNAL TRANSITION in model <coupledgeneratorandfsm.enigmemiu>
      New State:  FSMState = 3
      Output Port Configuration:
        port <FSMREADOUTPUTPORT>: __the word has been recognized__
      Next scheduled internal transition at time 8005
```

Figure VI-49 : Fin de la simulation et reconnaissance du mot

L'annexe 9 de ce document présente l'intégralité du code Python (interprétable sur la plateforme de simulation PyDEVS) généré pour cet exemple de modèle couplé MetaDEVS.

Conclusion du chapitre

Ce chapitre destiné aux transformations M2T dans le cadre de notre approche IDM/MDA MetaDEVS clôt la seconde et dernière partie de notre mémoire.

Nous avons montré comment le fait d'appliquer une approche M2T à des modèles MetaDEVS, de type PIM, permettait de les rendre productifs. Pour ce faire, nous avons tout d'abord fourni un exemple simple de *templates* de génération de code .html dans un but de documentation. Les *templates* que nous avons définis, quoi que simples, nous ont permis de mettre en évidence certains mécanismes, par exemple celui de parcours de modèles (dans le cas d'un modèle couplé). Par la suite, nous avons mis à profit ce mécanisme afin de créer les templates nous permettant de générer du code de simulation PyDEVS à partir de modèles MetaDEVS, en suivant les spécifications des concepteurs du simulateur PyDEVS.

Par la même occasion, nous pouvons réutiliser dans DEVSimPy tous nos modèles générés, car cette plateforme, sur laquelle travaillent les chercheurs du projet TIC de l'Università di Corsica, est basée sur un moteur PyDEVS.

Bien que la génération de code soit orientée PyDEVS, le fait d'avoir eu recours à des mécanismes d'héritage de templates et d'avoir structuré ces derniers de manière modulaire permet de réduire la charge de travail nécessaire pour transformer des modèles MetaDEVS vers une autre plateforme DEVS, programmée en JAVA par exemple.

Nous avons pris comme exemple de génération de code les modèles du chapitre V, que nous avons simulés, certains en tant que modèles atomiques, d'autres au sein de modèles couplés. Dans tous les cas, la génération de code est totalement automatique et le code final n'a jamais besoin d'être retouché, de quelque manière que ce soit. À aucun moment non plus, nous ne touchons au code du simulateur PyDEVS.

Nous avons démontré que notre approche MetaDEVS présentait plusieurs avantages au niveau de l'interopérabilité « externe » du formalisme DEVS, puisque nous utilisons ce formalisme pour simuler indifféremment des modèles qui proviennent de MetaDEVS tout comme des modèles issus d'autres formalismes puis transformés vers MetaDEVS.

Conclusion

Nous voici à présent arrivés au terme de la seconde et dernière partie de ce mémoire. Nous résumons en quelques lignes les points importants de ce document, puis nous donnons un bilan de nos travaux dans lequel nous décrivons la démarche scientifique que nous avons suivie, et nous donnons enfin quelques perspectives de recherche à court, moyen et long terme.

Nous avons débuté la première partie de ce travail par une présentation du monde de la modélisation et simulation, de ses concepts fondateurs, et de l'un des formalismes de modélisation les plus populaires : Discrete EVent system Specification (DEVS). Nous avons ensuite présenté une discipline du génie logiciel dont l'importance est grandissante : L'Ingéniérie Dirigée par les Modèles (IDM).

L'idée directrice de nos travaux est de faire bénéficier le formalisme DEVS des outils, méthodes et techniques provenant de l'IDM et en particulier de MDA. Il nous a fallu pour cela proposer un état de l'art décrivant les solutions proposées autour de DEVS et de l'IDM, que nous avons conclu par une synthèse comparative des méta-modèles DEVS existants.

La seconde partie de ce document a montré nos contributions, articulées autour de MetaDEVS, le méta-modèle pour DEVS que nous avons défini. MetaDEVS offre une représentation des modèles DEVS indépendante des plateformes. Il sert de « pivot » capable d'apporter des solutions d'interopérabilité à plusieurs niveaux.

Ces solutions existent tout d'abord au niveau des modèles, et nous avons montré comment transformer deux formalismes basés sur les états et les transitions (BasicDEVS et les automates à états finis) vers DEVS. Ces transformations sont de type M2M.

Tout modèle devant être simulé, nous avons également proposé des solutions pour transformer automatiquement des modèles MetaDEVS en code de simulation, et nous avons ici aussi donné des exemples complets de génération de code et de résultats de simulation. La plateforme cible qui a été choisie est PyDEVS. Cela nous permet également de pouvoir utiliser nos modèles dans DEVSimPy.

Bilan des travaux

Nous présentons ici la démarche scientifique que nous avons suivie. Nous nous sommes efforcés la faire transparaître à travers la structuration de ce document.

L'idée directrice de nos travaux a été de chercher à utiliser l'IDM pour améliorer la création, la modification, le stockage, la réutilisabilité et le partage (entre des plateformes de simulation différentes) des modèles DEVS. Nous avions également comme objectif de faire bénéficier d'autres formalismes de la puissance de simulation DEVS, il nous fallait pour cela créer des passerelles entre ces formalismes et DEVS. En d'autres termes, il s'agissait d'améliorer l'interopérabilité externe de DEVS.

Après nous être penchés sur les concepts clefs de l'IDM et de MDA, à savoir les modèles, les méta-modèles et les transformations de modèles, il nous a paru évident d'apporter une première contribution en créant un méta-modèle, contenant tous les concepts de DEVS, qui soit indépendant de toute plateforme, et qui permette de créer des *Platform Independent Models*, des PIM. Cette indépendance vis-à-vis des plateformes leur confère un autre avantage : une « durée de vie » plus longue que des modèles DEVS implémentés dans des langages objet standards.

Il existe actuellement plusieurs méta-modèles de DEVS, mais leur principal défaut reste le manque de concepts pour permettre de spéficier intégralement les fonctions atomiques, autrement qu'en y écrivant directement du code objet. Ceci conduit de ce fait à une perte d'indépendance vis-à-vis des plateformes, et nécessite une bonne maîtrise de la programmation orientée-objet. Un autre problème récurrent reste la spécification des états, beaucoup d'approches se limitant à des modèles contenant des états énumérés et unidimensionnels.

Pour notre part, nous avons choisi, certes au prix d'une perte en termes de puissance d'expression, de permettre uniquement la création de fonctions simples, composées de règles contenant chacune des conditions et des actions : les conditions décrivent des tests effectués sur les variables d'état, les ports, au moyen d'opérateurs binaires. Les actions décrivent des mises à jour de variables (en fonction, par exemple, des entrées). Nous avons également créé le concept de StateVar, qui nous permet de spécifier simplement des modèles à états multidimensionnels, en partant du principe simple que l'état d'un modèle à un instant donné est l'ensemble des valeurs des variables d'état qui le composent.

Ces concepts ont été rassemblés dans notre contribution MetaDEVS, méta-modèle défini en termes de MOF (syntaxe abstraite) et de contraintes OCL (sémantique statique). Nous avons utilisé l'environnement EMF pour implémenter ce méta-modèle.

Une fois que nous disposions de MetaDEVS, nous avons naturellement cherché à transformer d'autres formalismes vers DEVS. La littérature scientifique dédiée stipule que tout formalisme basé sur les états-transitions et/ou sur les évènements discrets peut être exprimé en termes de DEVS.

Nous avons donc cherché à caractériser quels formalismes pouvaient être candidats à une transformation vers MetaDEVS. Toujours dans un esprit IDM/MDA, nous avons cherché à définir des correspondances de concepts génériques entre ces formalismes et DEVS, le but étant de rester le plus longtemps possible en-dehors de tout cas concret de transformation et ainsi de produire un certain nombre de règles de transformation indépendantes du formalisme source.

Nous avons ensuite appliqué ces règles à deux cas concrets. Le premier concerne un petit formalisme que nous avions été amenés à créer durant nos travaux, BasicDEVS, et le second concerne un formalisme populaire en M&S : les automates à états finis. Pour ces deux formalismes, nous avons suivi une démarche analogue à celle qui nous a conduits à créer MetaDEVS et nous avons décrit leurs méta-modèles, toujours en termes de MOF et d'OCL.

L'application des règles de transformation s'est faite tout d'abord avec des règles exprimées en pseudo-langage, puis avec des règles concrètes, sous l'environnement Eclipse EMF, au moyen du *plugin* ATL, langage de transformation M2M permettant de spécifier des règles en mêlant les styles déclaratif et impératif.

Nous avons ainsi pu transformer de manière totalement automatique des modèles BasicDEVS, et des modèles FSM, en modèles DEVS. Nous avons par ailleurs montré que beaucoup de règles étaient réutilisables.

Enfin, dans le but de faire bénéficier ces modèles de la puissance de simulation DEVS, mais aussi de simuler nos modèles DEVS créés avec MetaDEVS, il nous a fallu générer du code à partir de nos modèles MetaDEVS. Nous avons pour cela suivi une approche par

template, et implémenté des règles de transformation M2T, vers la plateforme PyDEVS, en utilisant pour cela un patron de code fourni par les programmeurs de PyDEVS. Nous avons discuté de la possibilité de créer des passerelles vers d'autres langages orientés-objet, en modifiant un minimum nos templates. L'implémentation s'est faite encore une fois dans l'environnement EMF, au moyen du *plugin* Acceleo. Le code généré a ensuite été exécuté dans la console Python, en faisant appel au simulateur PyDEVS.

Nous avons ainsi montré que nous pouvions simuler conjointement, au sein de DEVS, des modèles qui, au départ, provenaient de formalismes différents. Il nous a fallu pour cela les transformer d'abord en modèles MetaDEVS (M2M) puis transformer ces modèles en code de simulation (M2T).

Perspectives de recherche

Ce document n'étant qu'une photographie à l'instant *t* de l'état de nos travaux, ces derniers sont amenés à évoluer. Nous présentons brièvement ici quelques axes de recherche qui nous tiennent à cœur, à court, moyen et long terme.

Court terme

- **Extensibilité des expressions DEVS**

Dans un autre registre, il est important que nous continuiions à augmenter les capacités d'expression de MetaDEVS : nous allons pour cela mettre en œuvre les capacités d'extensibilité de ce méta-modèle et permettre la création de conditions et d'expressions complexes. Une condition complexe pourra par exemple comporter une imbrication de conditions, liées entre elles par des connecteurs logiques, tandis qu'une action complexe pourra décrire par exemple une boucle, une incrémentation/décrémentation de variable, la création d'un nombre aléatoire…etc.

- **Syntaxe concrète**

Enfin, nous voulons doter MetaDEVS d'une syntaxe concrète, cela passe par la création d'un éditeur (disponible sous forme de plug-in Eclipse), et la création d'un formalisme graphique associé à MetaDEVS avec GMF.

Moyen terme

- **Transformations M2T vers d'autres plateformes de simulation**

Pour valider notre approche M2T, nous comptons également implémenter des transformations vers une ou plusieurs autres plateformes de simulation, et ainsi améliorer la structuration et la généricité de nos templates, donc leur efficacité.

- **Création de modèles auxiliaires grâce au M2M (modèles « générateurs » par exemple)**

Les générateurs sont des modèles atomiques DEVS très importants en M&S puisqu'ils permettent de générer un certain nombre de données, qui peuvent être fixées à l'avance ou aléatoires. Notre but est ici de suivre une approche M2M pour créer rapidement de tels

générateurs. Par exemple, à partir d'une chaîne de caractères, nous désirons créer le générateur de lettres associé, pour éventuellement le coupler à un automate. En quelques secondes, nous pourrons ainsi créer des modèles DEVS de générateurs de lettres spécifiques, qui nous serviront de modèles auxiliaires pour effectuer des tests.

- **Autres transformations M2M**

Nous désirons également implémenter une transformation M2M depuis les réseaux de Petri vers DEVS, puis, à plus long terme, d'autres formalismes plus éloignés des formalismes basés sur les états-transitions, tels que les Systèmes Multi-Agents (SMA).

Long terme

- **Extensions de DEVS**

En se basant sur les capacités d'extensibilité de MetaDEVS, nous avons pour but, à plus long terme, de permettre la prise en compte de certaines extensions de DEVS, ce qui passe par l'ajout de fonctions spécifiques à MetaDEVS.

- **Assistant pour la construction de modèles DEVS**

Il nous paraît important de permettre à des scientifiques non informaticiens de pouvoir spécifier et simuler des modèles DEVS. Pour ce faire, nous avons comme idée de créer un assistant générique d'aide à la construction de modèles DEVS. Un assistant ou wizard est une aide en ligne constituée d'un ensemble de questions destinées à guider le concepteur d'un modèle. Dans le contexte de MetaDevs, l'objectif est de proposer un outil générique de création d'assistant pour la définition de modèles DEVS basés sur le métamodèle MetaDevs.

L'outil proposé est un outil générique qui permet de définir différents questionnaires spécialisés basés sur le métamodèle MetaDEVS. Un tel outil contribuera à la définition d'environnements spécialisés à des domaines d'application spécifiques. Par exemple, si l'on souhaite définir des modèles Devs dans le contexte de la modélisation de processus métiers, on pourra définir un questionnaire spécifique. La démarche conception d'un tel outil comportera 3 étapes en suivant une approche similaire à la conception de l'outil GMF d'Eclipse :

– Définition d'un métamodèle de questionnaireDEVS

  Un questionnaireDEVS est composé d'étapes et de différents types de questions (par exemple choix unique, choix multiple). Chaque question sera associée à un méta-attribut ou une méta-référence modélisant la réponse à la question. Il s'agit de modéliser un processus séquentiel de questionnement. Ce métamodèle sera construit en utilisant le patter Composite. Il est complètement indépendant du métamodèle MetaDevs ou d'un quelconque autre métémodèle en dehors du méta formalisme utilisé pour le spécifier.

– Définition d'un métamodèle d'actions d'assistance

  Il s'agit de modéliser les actions à réaliser pour définir un modèle DEVS. Ces actions sont de deux types : instanciation de métaclasse et modification de valeur d'un méta-attribut ou d'une méta-référence.

– Définition d’un mapping entre les actions d’assistance et les éléments du questionnaire

  Le mapping permettra d’associer des actions de création Devs aux réponses des questions définies dans le questionnaire.

Références

[Abdulla et al. 2001] P.A. Abdulla et A. Nyl´en. Timed Petri nets and BQOs. In ICATPN'01, volume 2075 of LNCS, pages 53–72. Springer-Verlag, June 2001

[Agrawal et al. 2002] Agrawal A., Levendovszky T., Sprinkle J., Shi F., Karsai G., "Generative Programming via Graph Transformations in the Model Driven Architecture", Workshop on Generative Techniques in the Context of Model Driven Architecture, OOPSLA , Nov. 5, 2002, Seattle, WA.

[Agrawal et al. 2003] Agrawal, A., G. Karsai, et F. Shi, Graph Transformations on Domain-Specific Models, Institute for Software Integrated Systems (Rapport), Novembre 2003

[Agrawal 2004] A. Agrawal. Model Based Software Engineering. Graph Grammars and Graph Transformations. Area Paper. Vanderbilt University. EECS. April 8, 2004. http://www.isis.vanderbilt.edu/sites/default/files/Agrawal_A_4_8_2004_Model_Base.pdf (Consulté en juillet 2012)

[Akehurst et al. 2002] D. H. Akehurst, S.Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.): UML 2002 - The Unified Modeling Language 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings, LNCS 2460, 243-258, 2002

[Akehurst et al. 2005] D. H. Akehurst, W. G. Howells, and K. McDonald-Maier. Kent model transformation language. In Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005, Oct. 2005. URL http://sosym.dcs.kcl.ac.uk/events/mtip05/programme.html.

[Alur et al. 1994] Alur, R., Dill, D., 1994. A theory of timed automata. Theoretical Computer Science B 126, 183‑235

[Andries et al. 1996] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Kuske, D. Plump, A. Sch¨urr, and G. T¨antzer. Graph transformation for specification and programming. Technical Report 7/96, Universit¨at Bremen, 1996.

[Badros 2000] G. Badros, 2000. "JavaML: A Markup Language for Java Source Code." *Proceedings of the 9th International World Wide Web Conference* (Amsterdam, Netherlands, May. 15-19), 159- 7.

[Barros 1996] F. J. Barros, "Modelling and Simulation of Dynamic Structure Discrete Event Systems: A Systems Theory Approach", University of Coimbra, Coimbra, Thèse de doctorat, 1996

[Barros 1997] F. J. Barros, "Modelling Formalisms for Dynamic Structure Systems", ACM Transactions on Modelling and Computer Simulation, vol. 7, 501-515, 1997

[Barros 1998] F. J. Barros, "Abstract simulators for the DSDE formalism", présenté à Winter Simulation Conference (WSC), Washington DC, 407-412, 1998

[Batory et al. 2003] Don S. Batory, Jacob Neal Sarvela, Axel Rauschmayer, “Scaling Step-Wise Refinement”, International Conference on Software Engineering, pp 187-197, 2003.

[Bérard et al. 2005] Béatrice Bérard, Franck Cassez, Serge Haddad, O. H. Roux, and Didier Lime. Comparison of the Expressiveness of Timed Automata and Time Petri Nets. In Proceedings of the 3rd International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS’05), Lecture Notes in Computer Science, Uppsala, Sweden, September 2005. Springer. Extended version with proofs.

[Bergero et al. 2011] Bergero, Federico and Kofman, Ernesto - "PowerDEVS: a tool for hybrid system modeling and real-time simulation" (first ed.). Society for Computer Simulation International, San Diego, 2011

[Bernard, 1865] Claude Bernard, Introduction à l'étude de la médecine expérimentale (1865). Paris: Éditions Garnier-Flammarion, 1966, 318 pp

[Bézivin et al. 2001] J. Bézivin, O. Gerbé, "Towards a Precise Definition of the OMG/MDA Framework", ASE'01, Novembre 2001

[Bézivin et al. 2002a] Bézivin J., Blanc X., «Vers un important changement de paradigme en génie logiciel» Journal Développeur Référence - http://www.devreference.net/, Juillet 2002, p. 1-9.

[Bézivin et al. 2002b] Bézivin J., Blanc X., «Promesses et Interrogations de l’approche MDA», Journal Développeur Référence - http://www.devreference.net/, Septembre 2002, p. 1-14.

[Bézivin 2004] Bézivin J., « Sur les principes de base de l’ingénierie des modèles », RSTI-L’Objet, 10(4):145–157, 2004.

[Bézivin et al. 2005a] Jean Bézivin et Ivan Kurtev : Model-based Technology Integration with the Technical Space Concept. In Metainformatics Symposium, Esbjerg, Denmark, 2005. Springer-Verlag.

[Bézivin et al. 2005b] Jean Bézivin, Christian Brunette, Régis Chevrel, Frédéeric Jouault, and Ivan Kurtev. Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF). In Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05, San Diego, California, USA, 2005

[Blanc 2005] Blanc X., « MDA en action - Ingénierie logicielle guidée par les modèles », Paris, Eyrolles, 2005.

[Bolduc et al. 2001] Jean-Sébastien Bolduc and Hans Vangheluwe. A modelling and simulation package for classical hierarchical DEVS. MSDL technical report MSDL-TR-2001-01, McGill University, June 2001 (accédé en octobre 2012)
http://msdl.cs.mcgill.ca/projects/projects/DEVS/PythonDEVS/PythonDEVS.pdf

[Booch 1993] G. Booch, "Object-Oriented Analysis and Design with Applications (2nd Edition)", Addison-Wesley Professional, Septembre 1993

[Booch et al. 2004] Booch G., Brown A., Iyengar S., Rumbaugh J., Selic B. The IBM MDA Manifesto, The MDA Journal, May 2004, http://www.bptrends.com

[Borland, 2003] Borland, S., "Transforming Statechart models to DEVS", 2003 - msdl.cs.mcgill.ca/people/sborla/thesis.pdf

[Braun et al. 2003] P. Braun and F. Marschall. The Bi-directional Object-Oriented Transformation Language. Technical Report, Technische Universität München, TUM-I0307, May 2003

[Capocchi et al. 2011] Laurent Capocchi, Jean-François Santucci, Bastien Poggi, Céline Nicolai, DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems, 2nd International Track on Collaborative Modeling & Simulation - CoMetS'11, Paris : France (2011)

[Cetinkaya et al. 2010] Cetinkaya Deniz, Verbraeck Alexander et Seck Mamadou D., A metamodel and a DEVS implementation for component based hierarchical simulation modeling, Proceedings of the 2010 Spring Simulation Multiconference, SpringSim '10, Orlando, Floride, 2010

[Cetinkaya et al. 2011a] Cetinkaya Deniz, Verbraeck Alexander et Seck Mamadou D., MDD4MS: a model driven development framework for modeling and simulation, Proceedings of the 2011 Summer Computer Simulation Conference, La Hague, Pays-Bas, 2011

[Cetinkaya et al. 2011b] Cetinkaya Deniz et Verbraeck Alexander, Metamodeling and model transformations in modeling and simulation, 3048-3058. In Proceedings of the 2011 Winter Simulation Conference (WSC). 2011

[Cetinkaya et al. 2012] Cetinkaya Deniz, Verbraeck Alexander et Seck Mamadou D., Model transformation from BPMN to DEVS in the MDD4MS framework, Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, Orlando, Floride, 2012

[Cheon et al. 2004] S. Cheon, C. Seo, S. Park, B.P. Zeigler, Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Networked System, Advanced Simulation Technologies Conference, Arlington, VA, 2004

[Cho et al. 2001] Y. Cho, B.P. Zeigler, H. Sarjoughian, Design and Implementation of Distributed Real-Time DEVS/CORBA, IEEE Sys. Man. Cyber. Conf., Tucson, Oct. 2001

[Chow et al. 1994] Chow, Alex Chung Hen, & Zeigler, Bernard P. 1994. Parallel DEVS : a parallel, hierarchical, modular, modeling formalism. Pages 716–722 of : Proceedings of the 26th conference on Winter simulation. Society for Computer Simulation International.

[Clark et al. 2004] Clark T., Evans A., Sammut P., and Willans J. Applied Metamodelling: A Foundation for Language Driven Development v 0.1. Xactium, 2004

[Clark et al. 2008] Tony Clark, Paul Sammut et James Willians : SUPERLANGUAGES – Developing Languages and Applications with XMF. First Edition, 2008.

[Cleaveland 2001] C. Cleaveland. Program Generators with XML and Java. Prentice-Hall, Upper Saddle River, NJ, 2001

[Combemale 2008] Benoit Combemale, Approche de méta-modélisation pour la simulation et la vérification de modèle. Thèse de doctorat, Institut National Polytechnique de Toulouse, juillet 2008

[Czarnecki et al 1999] K. Czarnecki, U. Eisenecker, "Generative Programming: Methods, Techniques, and Applications", Addison-Wesley, 1999.

[Czarnecki et al. 2003] Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)

[Czarnecki et al. 2005] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Gl¨uck and M. Lowry, editors, GPCE 2005 - Generative Programming and Component Enginering. 4th International Conference, Tallinn, Estonia, Sept. 29 – Oct. 1, 2005, Proceedings, volume 3676 of LNCS, pages 422–437. Springer, 2005

[Czarnecki et al. 2006] Krzysztof Czarnecki, Simon Helsen: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3): 621-646 (2006)

[Diaw et al. 2010] Samba Diaw, Redouane Lbath, Bernard Coulette. Etat de l'art sur le développement logiciel basé sur les transformations de modèles. Dans : Technique et Science Informatiques, Hermès Science , Numéro spécial Ingénierie Dirigée par les Modèles, Vol. 29, N. 4-5/2010, p. 505-536, juin 2010.

[Djafarzadeh et al. 2005] Djafarzadeh R, Wainer G, Mussivand T. DEVS modeling and simulation of the cellular metabolism by mitochondria. Proceedings of the Society for Modeling and Simulation International Spring Simulation Multiconference 2005

[DOME 1999] "Dome Guide", Honeywell, Inc. Morris Township, N.J, 1999

[Ebert et al. 1994] J. Ebert and G. Engels, "Structural and behavioural views on OMTclasses", pp. 142-157, 1994

[Ehrig et al. 2005] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005). Springer-Verlag, 2005

[Engstrom et al. 2000] Eric Engstrom , Jonathan Krueger, “Building and Rapidly Evolving Domain-Specific Tools with DOME”, Proceedings of IEEE International Symposium on Computer-Aided Control System Design (CACSD), Anchorage, 2000 pp. 83-88

[Essar et al. 2001] R. Essar, J. Janneck and M. Naedele, “The Moses Tool Suite - A Tutorial”, Version 1.2, Computer, Engineering and Networks Laboratory, ETH Zurich, 2001.

[Estublier et al. 2005] J. Estublier, J-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B. Baudry M. Bouzhegoub, J-M. Jézéquel, M. Blay, and M. Riveil. Action Spécifique CNRS sur l’Ingénierie Dirigée par les Modèles. Rapport de synthèse 1.1.2, CNRS, janvier 2005.

[Favre et al. 2006] Favre J., Estublier J., Blay-Fornarino M., L'ingénierie Dirigée par les Modèles. Au delà du MDA, Cachan, Hermes-Lavoisier, 2006.

[Fehling 1993] Rainer Fehling, A Concept of Hierarchical Petri Nets with Building Blocks. Lecture Notes in Computer Science, Vol. 674; Advances in Petri Nets 1993, pages 148-168. Springer-Verlag, 1993.

[Filippi, 2003] Filippi, J. « Une architecture logicielle pour la multi-modélisation et la simulation à évènement discrets de systèmes naturels complexes » Thèse de Doctorat, Université de Corse , 2003

[Finkelstein 1992] Clive Finkelstein "Information Engineering: Strategic Systems Development". Sydney: Addison-Wesley, 1992

[Fishman 1978] G. S. Fishman, *Principles of Discrete Event Simulation*, Wiley, New York, 1978

[Fishwick 1995] P.A. Fishwick. Simulation Model Design and Execution : Building Digital Worlds, Englewood Cliffs, NJ : Prentice Hall, 1995

[Fleurey 2006] Franck Fleurey. Langage et méthode pour une ingénierie des modèles fiable. PhD thesis, Université de Rennes 1, October 2006

[Frankel 2003] David S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, John Wiley & Sons, 2003 ISBN 0-471-31920-1

[de Frutos Escrig et al. 2000] D. de Frutos Escrig, V. Valero Ruiz, and O. Marroqu´ın Alonso. Decidability of properties of timed-arc Petri nets. In ICATPN’00, Aarhus, Denmark, volume 1825 of LNCS, pages 187–206, June 2000.

[van Deursen et al. 1998] Arie van Deursen and Paul Klint. Little languages, little maintenance ? Journal of Software Maintenance, 10 : 75–93, 1998.

[Gamma et al. 1995] Erich Gamma, Richard Helm et Ralph Johnson : Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[Gardner et al. 2003] Gardner, T., Griffin, C., Koehler, J. , Hauser, R. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. July 2003.

[Garredu et al. 2009] S. Garredu, E. Vittori, and J.-F. Santucci. 2009. A DEVS-oriented intuitive modeling language. In Proceedings of the 2009 Spring Simulation Multiconference (SpringSim '09). Society for Computer Simulation International, San Diego, CA, USA, Article 155 , 8 pages.

[Garredu et al. 2012a] S. Garredu, E. Vittori, J.-F. Santucci and P.-A. Bisgambiglia, A Meta-Model for DEVS - Designed following Model Driven Engineering Specifications, Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Rome, Italy, 28 - 31 July, 2012.

[Garredu et al. 2012b] S. Garredu, E. Vittori, J.-F. Santucci and D. Urbani, Enriching a DEVS meta-model with OCL constraints, Proceedings of the 24th European Modeling and Simulation Symposium (EMSS), September 19-21 2012, Vienne, Autriche (en cours de parution)

[Gerber et al. 2002] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, Graph Transformation: First International Conference (ICGT 2002), Barcelona, Spain, October 7-12, 2002. Proceedings, volume 2505 of Lecture Notes in Computer Science, pages 90–105, Heidelberg, Germany, 2002. Springer-Verlag.

[Giambiasi et al. 2003] N. Giambiasi, J. Paillet et F. Châne. Simulation and verification II: from timed automata to DEVS models. In Proceedings of the 35th Conference on Winter Simulation: Driving innovation (New Orleans, Louisiana, December 07 - 10, 2003). Winter Simulation Conference, 923-931. 2003.

[Gitzel et al. 2004] R. Gitzel, A. Korthaus, The Role of Metamodeling in Model-Driven Development, In Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI2004), 2004

[Glinsky et al. 2005] Ezequiel Glinsky, Gabriel Wainer “DEVStone: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments”, Proceeding DS-RT '05 Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications, Pages 265-272, ISBN:0-7695-2462-1, 2005

[Glushkov 1961] Victor M. Glushkov, « The abstract theory of automata », dans Russian Math. Surveys, vol. 16, 1961, p. 1–53

[Greenfield et al. 2004a] "Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools" by Jack Greenfield and Keith Short, with contributions by Steve Cook and Stuart Kent. ISBN 0471202843

[Greenfield et al. 2004b] Greenfield, J. & Short, K. "Moving to Software factories", Software development, http://www.sdmagazine.com, Juillet 2004

[Griffin 2004] C. Griffin. Model Transformtion Framework (MTF). IBM Hursley, available from IBM Alphaworks, 2004.

[Harel 1987] Harel D., *Statecharts* : *A visual formalism for complex systems*, Science of Computer Programming, 8(3):231-274, 1987.

[HLA 2000] IEEE 1516-2000, « IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules », Institute of Electrical and Electronics Engineers, May 1, 2000.

[Hoare, 1968] – C.A.R Hoare - *An Axiomatic Basis for Computer Programming*, 1968

[Hofstadter 1979] – Douglas Hofstadter, Gödel, Escher, Bach : Les Brins d'une Guirlande Éternelle, Dunod, ISBN 978-2-10-052306-1

[Hong et al. 2004] Hong, S.-Y. and T. G. Kim. Embedding UML Subset into Object-oriented DEVS Modeling Process, Proceedings of the Summer Computer Simulation Conference, San Jose, CA, July 2004

[Hopcroft et al. 1976] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1976.

[Huang et al. 2005] Huang, D., and H. S. Sarjoughian. 2004. Software and simulation modeling for real-time software-intensive systems. Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications. Washington, DC.

[Hwang 2005] M.H. Hwang, ``Generating Finite-State Global Behavior of Reconfigurable Automation Systems: DEVS Approach``, Proceedings of 2005 IEEE-CASE, Edmonton, Canada, 2005

[Jacobs et al. 2002] Jacobs, P.H.M., N.A. Lang, and A. Verbraeck. "DSOL; a Distributed Java Based Discrete Event Simulation Architecture". In E. Yücesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes, eds. Proceedings of the 34th Winter Simulation Conference, San Diego, California, 793-800, 2002

[Jacobson et al. 1999] I. Jacobson, G. Booch, and J. Rumbaugh, "The Unified Software Development Process", Addison-Wesley Object Technology Series, Addison-Wesley Professional, Février 1999.

[Janneck 2000] J. Janneck, "Graph-type definition language (GTDL)—specification", Technical report, Computer, Engineering and Networks Laboratory, ETH Zurich, 2000.

[Janousek et al. 2006] Vladimír Janousek, Petr Polásek and Pavel Slavícek. "Towards DEVS Meta Language". In: ISC 2006 Proceedings, Zwijnaarde, BE, 2006, p. 69-73, ISBN 90-77381-26-0

[Jensen 1992] Jensen, K. (1992). Colored Petri nets: Basic concepts, analysis methods and practical use, Vol. 1. New York: Springer.

[Jézéquel 2008] Jean-Marc Jézéquel. Model transformation techniques. 2008 (accédé septembre 2012) http://people.irisa.fr/Jean-Marc.Jezequel/enseignement/ModelTransfo.pdf

[Jouault et al. 2006a] 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

[Jouault et al. 2006b] Jouault, Frédéric, Kurtev Ivan, «On the Architectural Alignment of ATL and QVT», In Proceedings of the 2006 ACM symposium on Applied computing, session Model transformation, Dijon, 2006, New York, ACM Press, p. 1188-1195.

[Jouault et al. 2008] Frédéric Jouault , Freddy Allilaire , Jean Bézivin , Ivan Kurtev, ATL: A model transformation tool, Science of Computer Programming, v.72 n.1-2, p.31-39, June, 2008

[Kalnins et al. 2004] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In U. Assmann, editor, Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004), pages 12–26, Linköping, Sweden, June 2004. Research Center for Integrational Software Engineering, Linköping University.

[Kang et al. 1990] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990

[Kelly et al. 2007] Steven Kelly, Juha-Pekka Tolvanen, Domain-Specific Modeling, John Wiley & Sons, 2007

[Kent et al. 2002] S. Kent, O. Patrascoiu, “Kent Modelling Framework Version – Tutorial”, Computing Laboratory, University of Kent, Canterbury, UK, Draft, December 2002.

[Kifer et al. 1995] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of the ACM, 42(4):741–843, July 1995

[Kim et al. 1998] Kim, J.Y., Tag C. Kim. A heterogeneous simulation framework based on the devs bus and the high level architecture. In 30th Winter Simulation Conference, pages 48–49. IEEE Computer Society, Dec 1998

[Kim et al. 2003] Y.J. Kim, J.H. Kim, and T.G. Kim. Heterogeneous simulation framework using devs bus. Simulation, 79(1) :3, 2003

[Kleppe et al. 2004] A. Kleppe, S. Warmer, W. Bast, "MDA Explained. The Model Driven Architecture: Practice and Promise", Addison-Wesley, April 2003

[Kofman et al. 2003] Kofman, E., M. Lapadula, and E. Pagliero, PowerDEVS: A DEVS-based Environment for Hybrid System Modeling and Simulation, Technical Report LSD0306, LSD, Universidad Nacional de Rosario, Argentina, 2003

[Königs 2005] A. Königs. Model transformation with triple graph grammars. In Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005, Oct. 2005.

[Kurtev et al. 2002] Kurtev, I., Bézivin, J. et Akşit, M. (2002) Technological Spaces: An Initial Appraisal. In: International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, 30 Oct - 1 Nov 2002, Irvine, USA. pp. 1-6.

[Labiche et al. 2005] Labiche, Yvan et Wainer, Gabriel A., "Towards the Verification and Validation of DEVS Models", Proceedings of the 1st Open International Conference on Modeling & Simulation, Clermont-Ferrand, France, 2005

[Lara et al. 2002a] J. Lara , H. Vangheluwe, "Using AToM as a Meta CASE Tool", 4th International Conference on Enterprise Information Systems, Universidad de Castilla-La Mancha, Ciudad Real (Spain), 3-6, April 2002

[Lara et al. 2002b] J. Lara, H. Vangheluwe, "Computer Aided Multi-Paradigm Modeling to Process Petri-Nets and Statecharts", 1st International Conference on Graph Transformation, Barcelona (Spain), 7-12, October 2002

[Lara et al. 2002c] Juan De Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, volume 2306 of LNCS. Springer-Verlag, 2002.

[Lawley et al. 2005] M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In Proceedings of Model Transformations in Practice Workshop (MTIP) at MoDELS Conference, Montego Bay, Jamaica, 2005, Oct. 2005.

[Lédeczi et al. 2001] Akos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing domain-specific design environments. Computer, 34(11):44–51, 2001.

[Lee 2001] Edward A. Lee, "Overview of the Ptolemy Project", *Technical Memorandum* UCB/ERL M01/11 March 6, 2001.

[Lei et al. 2009] Yonglin Lei, Weiping Wang, Qun Li, and Yifan Zhu, A transformation model from DEVS to SMP2 based on MDA, Simulation Modelling Practice and Theory, Vol. 17, Nr. 10 (2009) , p. 1690-1709.

[Levendovszky et al. 2004] Levendovszky T., Lengyel L., Mezei G., Charaf H.: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, 2nd International Workshop on Graph Based Tools (GraBaTs); workshop at ICGT 2004, Rome, Italy, 2004

[Levytskyy et al. 2003a] A. Levytskyy, E. J. Kerckhoffs, E. Posse, and H. Vangheluwe, "Creating DEVS components with the meta-modelling tool AToM^3" in 15th European

Simulation Symposium (ESS), A. Verbraeck and V. Hlupic, Eds. Society for Modeling and Simulation International (SCS), October 2003, pp. 97 – 103, delft, The Netherlands.

[Levytskyy et al. 2003b] Levytskyy, A. and E.J.H. Kerckhoffs. 2003. "From Class Diagrams to Zope Products with the Meta-Modelling Tool AToM3". In A. Bruzzone and Mhamed Itmi, editors, Summer Computer Simulation Conference, pp. 295 – 300. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada.

[Macdonald 1986] Ian Macdonald "Information engineering". in: Information Systems Design Methodologies. T.W. Olle et al. (ed.). North-Holland, 1986

[Marschall et al. 2003] F. Marschall and P. Braun. Model Transformations for the MDA with BOTL. In Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, University of Twente, Enschede, The Netherlands, June 26-27, 2003, pp. 25-36

[Mason 2000] Kim Mason, "Moses Formalism Creation – Tutorial", Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092, Switzerland, February 9, 2000.

[Mayer et al. 1995] R. J. Mayer, C.P. Menzel, M.K. Painer, P.S. deWitte, T. Blinn, B. Perakath, "Information Integration For Concurrent Engineering (Iice) Idef3 Process Description Capture Method Report", Human Resources Directorate Logistics Research Division, Knowledge Based Systems, Incorporated, Texas 77840-2335, September 1995.

[Mens et al. 2006] Tom Mens,, Krzysztof Czarnecki and Pieter Van Gorp, A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science (ENTCS) Volume 152, March, 2006, pp. 125-142

[Merlin 1971] P.M. Merlin. A study of the recoverability of computing systems. PhD thesis, University of California, Irvine, CA, 1974.

[Metacase 2000] "ABC To Metacase Technology", White Paper, MetaCase Consulting, Finland, August, 2000.

[Metacase 2001] "Domain-Specific Modelling: 10 Times Faster Than UML", White Paper, MetaCase Consulting, Finland, January, 2001.

[Microsoft 2012] Visual Studio Visualization and Modeling SDK, http://archive.msdn.microsoft.com/vsvmsdk, (consulté en juillet 2012)

[Mittal 2006] Mittal, S., Extending DoDAF to Allow DEVS-Based Modeling and Simulation, Special issue on DoDAF, Journal of Defense Modeling and Simulation JDMS, 3(2), 2006

[Mittal et al. 2007a] S. Mittal, J. L. R. Martín., B.P. Zeigler « DEVSML: automating DEVS execution over SOA towards transparent simulators », Proceedings of the 2007 ACM Spring Simulation Multiconference, March 25-29, 2007, Norfolk, VA, USA, Vol. 2, pp. 287-295.

[Mittal 2007] Mittal, S. DEVS "Unified Process for Integrated Development and Testing of Service Oriented Architectures", Ph.D. thesis, University of Arizona, May 2007

[Mittal et al. 2012] Saurabh Mittal, Scott A. Douglass, DEVSML 2.0: The Language and the Stack, DEVS Symposium, Spring Simulation Multiconference 2012, Orlando

[Minsky, 1968] M.L. Minsky. Matter, Mind and Models Semantic Information Processing, 1968

[Mooney et al. 2009] J. Mooney et H.S. Sarjoughian. A Framework for executable UML models, In High Performance Computing & Simulation Symposium, Spring Simulation Conference, pages 1-8, 2009

[Muller et al. 2005] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel, Weaving Executability into Object-Oriented Meta-Languages.In S. Kent L. Briand, editor, Proceedings of MODELS/UML'2005, LNCS, Montego Bay, Jamaica, October 2005. Springer

[Neighbors 1980] J. Neighbors, "Software Construction Using Components", Ph.D. Thesis, ICS-TR-160, University of California at Irvine, 1980.

[Neighbors 1983] J. Neighbors, "Draco 1.2 Users Manual", University of California at Irvine, 1983.

[Niere et al. 1999] J. Niere, A. Zündorf: Tool Demonstration: Testing and Simulating Production Control Systems Using the Fujaba Environment. In Proc. of International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands, Lecture Notes in Computer Science (LNCS) 1779. Springer Verlag, 1999

[Nikolaidou et al. 2007] M. Nikolaidou, V. Dalakas, G.-D. Kapos, L. Mitsi, and D. Anagnostopoulos, « A UML 2.0 profile for DEVS: Providing code generation capabilities for simulation » in Proceedings of 16th International Conference on Software Engineering and Data Engineering (SEDE-2007), Las Vegas, USA, July 2007, (Invited paper)

[Nikolaidou et al. 2008] M. Nikolaidou, V. Dalakas, L. Mitsi, G.-D. Kapos, D. Anagnostopoulos, « A SysML Profile for Classical DEVS Simulators » (Conference Paper) Proceedings of the 2008 The Third International Conference on Software Engineering Advances, 978-0-7695-3372-8, Pp 445-450, 2008, 10.1109/ICSEA.2008.24, IEEE Computer Society

[Nutaro 2010] James J. Nutaro. Building Software for Simulation: Theory and Algorithms, with Applications in C++. Wiley. 2010

[OMG-QVT-2011] Site de l'OMG (Object Management Group), documentation officielle http://www.omg.org/spec/QVT/1.1/PDF/

[Partsch et al. 1983] H. Partsch et R. Steinbrüggen. Program transformation systems. ACM Computing Surveys, 15(3):199–236, 1983

[Peterson 1981] J. Peterson, "Petri Net Theory and the Modeling of Systems", *Prentice-Hall*, *Englewood Cliffs, NJ,* 1981

[Petri 1962] Petri, C.A., Kommunikation mit Automaten. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962, Seconde Edition : New York: Griffiss Air Force Base, Technical Report RADC-TR-65--377, Vol.1, 1966, Pages: Suppl. 1, English translation

[Posse et al. 2003] Ernesto Posse, Jean-Sébastien Bolduc, Hans Vangheluwe. Generation of DEVS Modelling & Simulation Environments. In Proceedings of the 2003 Summer Computer Simulation Conference SCSC 2003.

[Prax, 2003] Jean-Yves Prax « Le Manuel du Knowledge Management », DUNOD, 2003

[Quesnel et al. 2001] Quesnel G., Ramat E., Soulie J.C., Duvivier D., Duboz R., Virtual Laboratory Environment : un environnement de multimodélisation et de simulation de systèmes complexes, 2012. Studia Informatica Universalis, 10 (1) : 205-234

[Ramalho et al. 2003] F. Ramalho, J. Robin, and R.S.M.D. Barros, "XOCL - an XML Language for Specifying Logical Constraints in Object Oriented Models", ;presented at J. UCS, 2003, pp.956-969.

[Ramchandani 1974] C. Ramchandani, Analysis of asynchronous concurrent systems by timed Petri nets. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974

[RFP-QVT] http://www.omg.org/cgi-bin/doc?ad/01-11-14.pdf (Accédé en Octobre 2012)

[Richters et al. 2002] M. Richters, M. Gogolla, "OCL: Syntax, semantics, and tools", pp. 447-450, 2002.

[Risco-Martín et al. 2007a] J. L. Risco-Martín, S. Mittal, M. A. López-Peña, and J. M. de la Cruz, (2007), "A W3C XML schema for DEVS scenarios", *in*Maurice J. Ades, ed., 'SpringSim (2)' , SCS/ACM, , pp. 279-286

[Risco-Martín et al. 2007b] Risco-Martin, J.L., Mittal, S., Zeigler, B.P., Cruz, J.L, "From UML Statecharts to DEVS State Machines using XML", Multi-paradigm Modeling, IEEE/ACM International Conference on Model Driven Engineering Languages and Systems , 2007

[Robinson et al. 1994] Keith Robinson & Graham Berrisford – "Object-Oriented SSADM" – Prentice Hall – ISBN : 0-13-3094444-8, 1994

[Romeo 2012] Environnement ROMEO pour la modélisation et la simulation des réseaux de Petri http://romeo.rts-software.org/ (accédé en juillet 2012)

[Rumbaugh et al. 2005] Rumbaugh, J., Jacobson, I. et Booch, G., "The unified modeling language reference manual", The Addison-Wesley object technology series, Addison-Wesley, 2005

[Sarjoughian et al. 2000] Hessam S. Sarjoughian et B. P. Zeigler. "DEVS and HLA: Complementary Paradigms for Modeling and Simulation?" Simulation: Transactions of the Society for Modeling and Simulation International 17, no. 4 (2000): 187-97.

[Sarjoughian et al. 2012] Sarjoughian, Hessam et Markid, Abbas Mahmoodi, EMF-DEVS modeling, Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, Orlando, Florida, 2012

[Schulz et al. 2000] Schulz, S., T. C. Ewing, and J. W. Rozenblit. 2000. Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling, Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems. Edinburgh.

[Sen et al. 2008] Sagar Sen, Benoit Baudry, and Hans Vangheluwe, Domain-Specific Model Editors with Model Completion. In Models in Software Engineering, Holger Giese (Ed.). Lecture Notes In Computer Science, Vol. 5002. Springer-Verlag, Berlin, Heidelberg 259-270, 2008

[Sendall et al. 2003] Sendall, S., Kozaczynski, W.: Model Transformation - The Heart and Soul of Model-Driven Software Development. IEEE Software, Special Issue on Model Driven Software Development (2003) 42–45

[Seo et al. 2004] Seo, C., Park, S., Kim, B., Cheon, S., Zeigler, B.P., Implementation of Distributed High-performance DEVS Simulation Framework in the Grid Computing Environment, Advanced Simulation Technologies conference (ASTC), Arlington, VA, 2004

[Seo 2009] C. Seo, "Interoperability between DEVS Simulators using Service Oriented Architecture and DEVS Namespace", Ph.D. dissertation, Electrical and Computer Engineering Dept., University of Arizona, Spring 2009.

[Siegel 1996] Jon Siegel, CORBA, Fundamentals and Programming, Wiley Computer Publishing Group, 1996

[Simulink 2002] “Simulink Reference”, The Mathworks, Inc., July 2002.

[SISO 2008] Simulation Interoperability Standards Organisation, SISO-REF-019-2008: Discrete-Event Systems Specification (DEVS) SG Final Report http://www.sisostds.org/ProductsPublications/ReferenceDocuments.aspx

[Smolander 1991] Smolander, K., Lyytinen, K., Tahvanainen, V.-P., and Marttiin, P., "MetaEdit: A flexible graphical environment for methodology modelling", Proceedings of CAiSE'91, 3rd Intl. Conference on Advanced Information Systems Engineering, Springer Verlag, pp. 168–193, 1991

[Softeam 2008] Softeam, support de formation Objecteering 6/MDA Modeler version 2.0, Janvier 2008.

[Soley 2004] Richard Mark Soley, An Overview of The OMG's Model Driven Architecture, Business Process Trends, vol.1 n°4, 2004 (accédé octobre 2012) http://www.bptrends.com/publicationfiles/bptspotlight0622.pdf

[Somogyi et al. 1996] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In Proceedings of the Australian Computer Science Conference, pages 499–512, Glenelg, Australia, Feb. 1995

[Song 2006] Song, H., Infrastructure for DEVS Modelling and Experimentation. Master's thesis. McGill University. School of Computer Science. (2006)

[Steinberg et al. 2003] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks, Eclipse Modeling Framework 2nd Edition, Addison Wesley, 2009

[Stoy, 1977] - Joseph E. Stoy - Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. MIT Press, Cambridge, Massachusetts, 1977.

[Sztipanovits et al. 1995] Janos Sztipanovits, Gabor Karsai, Csaba Biegl, Ted Bapty, Ákos Lédeczi, and Amit Misra. Multigraph : an architecture for model-integrated computing. In ICECCS, pages 361–368, 1995.

[Sztipanovits et al. 1997] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. Computer, 30(4) :110–111, 1997.

[Taentzer 2003] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. B¨ohlen, editors, AGTIVE, volume 3062 of Lecture Notes in Computer Science, pages 446–453. Springer, 2003. ISBN 3-540-22120-4

[Tardieu et al. 1989] Hubert Tardieu, Arnold Rochfeld et René Colletti, La méthode Merise : Principes et outils, Paris, Éditions d'organisation, 1983 (réimpr. 1989, 1991, 1994, 2000 et 2003), 328 p. -ISBN 2-7081-1106-X et 2708124730

[Thomas 2008] Frédéric Thomas : Contribution à la prise en compte des plates-formes logicielles d'exécution dans une ingénierie générative dirigée par les modèles. Thèse de doctorat, Université d'Evry, 2008.

[Tolvanen et al. 2003] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+ : defining and using domain-specific modeling languages and code generators. In OOPSLA '03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 92–93, New York, NY, USA, 2003. ACM Press.

[Touraille et al. 2009a] L.Touraille, M.K. Traoré, D. Hill, "On the Interoperability of DEVS components : On-Line vs. Off-Line Strategies.", 2009, UMR CNRS 6158, LIMOS/RR-09-04, 13 p.

[Touraille et al. 2009b] Touraille L., Traore M.K., Hill D.R.C., « A Markup Language for the Storage, Retrieval, Sharing and Interoperability of DEVS Models », Proceedings of the 2009 ACM/SCS Spring Simulation Multiconference, March 22-27, 2009, San Diego, CA, USA, 6 p

[Touraille et al. 2010] L.Touraille, M.K. Traoré, D. Hill, « SimStudio : une Infrastructure pour la Modélisation, la Simulation et l'Analyse de Systèmes Dynamiques Complexes », UMR CNRS 6158, LIMOS/RR-10-13, 2010, 12 p. (2010)

[Touraille et al. 2011] Luc Touraille, Mamadou K. Traoré, and David R. C. Hill. 2011. A model-driven software environment for modeling, simulation and analysis of complex systems. In Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (TMS-DEVS '11). Society for Computer Simulation International, San Diego, CA, USA, 229-237.

[Traoré 2009] Traoré M.K. A Graphical Notation for DEVS. HPCS : a joint DEVS and HPC Symposium. San Diego, CA, March 22-27. 7 pp. 2009

[Turing 1936] Alan Mathison Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, vol. 2:42, coll. « Proceedings of the London Mathematical Society », 1936, p. 230-265
http://www.thocp.net/biographies/papers/turing_oncomputablenumbers_1936.pdf

[Uhrmacher 2001] A. M. Uhrmacher, Dynamic structures in modeling and simulation: a reflective approach, ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 11 Issue 2, April 2001, pp. 206 - 232

[Vangheluwe 2000] Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In: Varga A (ed) IEEE International Symposium on Computer-Aided Control System Design. IEEE Computer Society Press, Anchorage, Alaska, pp 129–134, 2000

[Varro et al. 2002] D. Varro, G. Varro and A. Pataricza. Designing the automatic transformation of visual languages. Science of Computer Programming, vol. 44(2):pp. 205--227, 2002

[Visser 2004] Visser, E.: Program-transformation.org – A Taxonomy of Program Transformation (2004)
http://www.programtransformation.org/Transform/ProgramTransformation.

[Vojtisek et al. 2005] D. Vojtisek and J.-M. Jézéquel. MTL and Umlaut NG: Engine and framework for model transformation. Online, July 2004. URL http://www.ercim.eu/publication/Ercim_News/enw58/vojtisek.html. ERCIM News No. 58, (Accédé en Octobre 2012).

[Wainer 1998] Wainer G., Discrete-event cellular models wth explicit delays, thèse de doctorat, Université d'Aix-Marseille III, 1998

[Wainer 2002] Wainer G., "CD++: a toolkit to define discreteevent models". Software, Practice and Experience. Vol. 32, No.3. pp. 1261-1306. November 2002.
[Wainer et al. 2002] Wainer, Gabriel A., Giambiasi, Norbert - N-dimensional Cell-DEVS Models, *Discrete Event Dynamic* Systems, Springer Netherlands, 2002

[Wainer et al. 2008] Wainer Gabriel, Liu Qi and Chazal Julien, Quinet Lo, Performance analysis of web-based distributed simulation in DCD++: a case study across the Atlantic Ocean, Proceedings of the 2008 Spring simulation multiconference, SpringSim '08, Ottawa, Canada, 2008

[Wainer 2009] Gabriel A. Wainer, Discrete-Event Modeling and Simulation: A Practitioner's Approach, CRC Press, 2009, ISBN 1420053361

[Wainer et al. 2009] Gabriel A. Wainer, Qi Liu, Tools for Graphical Specification and Visualization of DEVS Models, SIMULATION: Transactions of the Society for Modeling and Simulation International, Volume 85, Number 3, page 131-158 - Mar. 2009

[Wainer et al. 2010] Wainer Gabriel A., Al-Zoubi Khaldoon, Dalle Olivier, Hill, R.C., Mittal S., Martín, J.L. Risco, Sarjoughian Hesam, Touraille L., Traoré Mamadou K., Zeigler Bernard P. "Standardizing DEVS model representation", in Discrete-Event Modeling and Simulation: Theory and Applications, Wainer, G. and Mosterman, P. (Ed.) (2011) pp. 427-458, Chap. 17

[Wainer et al. 2011] WAINER Gabriel, GLINSKY Ezequiel, GUTIERREZ-ALCARAZ Marcelo - Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark, SIMULATION July 2011 87: 555-580, first published on January 20, 2011

[Willink 2003] Edward D. Willink, UMLX : A Graphical Transformation Language for MDA, Eclipse Document
http://www.eclipse.org/gmt/umlx/doc/MDAFA2003-4/MDAFA2003-4.pdf

[Zeigler 1976] B.P. Zeilger, Theory of Modeling and Simulation, New-York: Wiley-Interscience, 1976

[Zeigler 1984] Bernard Zeigler, Multifacetted Modeling and Discrete Event Simulation, Academic Press, London; Orlando, 1984 (ISBN 978-0-12-778450-2)

[Zeigler 1987] Bernard Zeigler, « Hierarchical, modular discrete-event modelling in an object-oriented environment », dans SIMULATION, vol. 49, 1987, p. 219–230

[Zeigler et al. 1996] Zeigler, B.P., Yoonkeun Moon, Doohwan Kim, Jeong Geun Kim, DEVS-C++: a high performance modelling and simulation environment, Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences: Information Systems Organizational Systems and Technology, pp. 350-359 vol.1, 19996

[Zeigler et al. 1997] Zeigler, B.P., Yoonkeon Moon, Doohwan Kim, Ball, G. - The DEVS Environment for High-Performance Modeling and Simulation. IEEE C S & E, 1997. 4(3): p. 61-71

[Zeigler et al. 1998] B.P. Zeilger, et Ball, G. « The DEVS/HLA Distributed Simulation Environment And Its Support for Predictive Filtering », ECE Dept., UA, Tucson, AZ, 1998.

[Zeigler et al. 1999a] B.P. Zeilger, D. Kim, et S.J. Buckley. 1999. « Distributed supply chain simulation in a DEVS/CORBA execution environment », in Proceedings of the 1999 Winter Simulation Conference, ed, P.A. Famngton, H.B. Nembhard, D.T. Sturrock, and G.W. Evans, 1333-1340.

[Zeigler et al. 1999b] B.P. Zeilger, G. Ball, H.J. Cho, et J.S. Lee, « Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions » in Simulation Interoperation Workshop (SIW), Orlando, FL, 1999.

[Zeigler et al. 2000] B.P. Zeigler, H. Praehofer, T.G. Kim, "Theory of Modeling and Simulation: Integrating Discrete and Continuous Complex Dynamic Systems", 2nd Edition, Academic press 2000, ISBN 0-12-778455-1.

[Zengin et al. 2010] Zengin, A. et Sarjoughian, H., DEVS-Suite simulator: A tool teaching network protocols, Proceedings of the 2010 Winter Simulation Conference (WSC), Baltimore, USA, Décembre 2010

[Zhang et al. 2005] Zhang, M., Zeigler, B.P., Hammonds, P., DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies, ITEA Journal, July 2005

[Zinoviev 2005] D. Zinoviev, "Mapping DEVS Models onto UML Models," Proc. of the 2005 DEVS Integrative M&S Symposium, San Diego, CA, April 2005, pp. 101-106

Liste des Acronymes

API : Application Programming Interface

AS : Action Semantics

AToM^3 : A Tool for Multi-formalism and Meta-Modelling

ATL : ATLas Transformation Language

(E)BNF : (Extended) Backus-Naur Form

BPMN : Business Process Modeling Notation

CASE : Computer Aided Software Engineering

CIM : Computation Independent Model

CORBA : Common Object Request Broker Architecture

DESS : Differential Equation System Specification

DEVS : Discrete EVent system Specification

DEVS-MS : DEVS Meta-Simulator

DSML : Domain Specific Modeling Language

DSOL : Distributed Simulation Object Library

DTD : Document Type Definition

DTSS : Discrete Time System Specification

ER (diagrammes) : Entity-Relationship

FD-DEVS : Finite-Deterministic DEVS

FSM : Finite State Machine

GMSE : Generative Multiple formalisms modeling and Simulation Environment

GMT : Generative Modeling Technologies

GPL : General Public License

HLA : High-Level Architecture

HTML : HyperText Markup Language

IDM : Ingénierie Dirigée par les Modèles

KM3 : Kernel Meta Meta Model

LHS, RHS : Left Hand Side, Right Hand Side

M&S : Modeling and Simulation (Modélisation et Simulation)

M2M : Model To Model (transformation)

M2T : Model To Text (transformation)

MDA : Model Driven Architecture

MDD : Model Driven Development

MMD4MS : Model Driven Development Framework for Modeling and Simulation

MDE : Model Driven Engineering (equivalent anglais de l'IDM)

MOF, EMOF, CMOF : Essential/Complete Meta-Object Facility

OCL : Object Constraint Language

OMA : Object Management Architecture

OMG : Object Management Group

PIM : Platform Independent Model

PSM : Platform Specific Model

QVT : Query/View/Transformation

RdP : Réseau de Petri

RFP : Request For Proposal

SCD : Simplified Class Diagrams

SE : Simulation Engine (moteur de simulation)

SISO : Simulation Interoperability Standards Organization

SMA : Systèmes Multi-Agents

SMP, SMP2 : Simulation Model Portability

SOA : Service-Oriented Architecture

TCP/IP : Transmission Control Protocol / Internet Protocol

UML : Unified Modeling Language

VIATRA : VIsual Automated model TRAnsformations

XMI : XML Metadata Interchange

XML : eXtensible Markup Language

XSLT : eXtensible Stylesheet Language

Table des illustrations

ANNEXES

Pour présenter ces techniques de modélisation, nous nous basons sur [Fishwick 1995] qui propose de les classer en quatre catégories différentes :

Modélisation conceptuelle. Elle est basée sur la création d'un modèle conceptuel informel, qui nous informe sur les caractéristiques du système. Un croquis de maison, par exemple, peut être considéré comme un modèle conceptuel, mais la plupart du temps un modèle conceptuel comporte également un texte dans lequel sont clairement identifiés les limites du système, ses variables, ses composants, et sa structure. Il existe différents formalismes spécialement dédiés à la modélisation conceptuelle, par exemple le Modèle Conceptuel de Données, ou MCD (domaine des bases de données) ;

Modélisation déclarative. Cette technique vise à décrire les évolutions du système en fonction des états et des transitions qui le composent. Elle possède en son sein diverses sous-techniques, basées soit sur les états, soit sur les évènements. Par exemple, un réseau de Petri (1.2.2.c)) non marqué peut être employé pour représenter un graphe d'états, ou un graphe d'évènements ;

Modélisation fonctionnelle (ou comportementale). Dans ce cas, le modèle est vu comme une boîte noire et le signal d'entrée est défini en fonction du temps. Les sorties sont calculées en fonction du signal d'entrée grâce à une fonction interne ;

Modélisation spatiale. Ces techniques supposent la prise en compte de l'aspect spatial, au cours du temps, c'est-à-dire la localisation dans l'espace-temps des différents composants du système.

Concernant les processus de modélisation, il existe une abondante littérature liée au cycle de vie des modèles et ses différentes phases, par exemple [Zeigler et al. 2000]. Nous nous limiterons ici à décrire une partie de ce cycle de vie commençant à la formulation du problème, et se terminant après qu'un premier modèle ait vu le jour.

Formulation du problème. Le cycle de modélisation et simulation a comme point de départ un problème qui nécessite d'être étudié afin d'être compris, voire résolu dans certains cas. À ce stade, on doit être en mesure de comprendre le comportement du système, qui peut être naturel ou artificiel, existant ou fictif. Il faut également comprendre l'organisation interne de ce système, définir le cadre expérimental ;

Analyse de l'existant. Il existe peut-être des solutions concernant des problèmes similaires, il faut donc se documenter sur ce qui existe dans le domaine que nous étudions. Cela revient à effectuer un état de l'art du domaine. Si une ou des solutions existent, il peut être judicieux de s'en inspirer, après les avoir évaluées en détail ;

Identification des variables d'entrée/sortie. À ce stade, le système est encore considéré comme une boîte noire, mais on peut néanmoins définir quelles sont les variables qu'il manipule (qu'il admet en entrée et produit en sortie) et lesquelles sont modifiables par une intervention humaine (se traduisant par un stimulus en entrée) ;

Création d'un modèle conceptuel. On est maintenant en mesure de construire une description du système à haut niveau d'abstraction, incluant sa structure et son comportement.

Tous les composants du système sont maintenant identifiés, on connaît les différentes variables d'état et les relations qui les lient ;

Mise en relation du modèle conceptuel avec les variables identifiées. Cela passe par une étude plus approfondie du système (s'il existe) afin de collecter des données. C'est à ce moment-là que l'on va introduire l'aspect temporel dans la description car le système, et ses composants, évoluent avec le temps. On va également se demander quelles variables sont qualitatives, lesquelles sont quantitatives, quelles données vont être déterministes, quelles données vont être stochastiques (aléatoires), voire quelles données vont être floues ;

Construction du modèle. C'est une fois toutes les étapes précédentes effectuées que l'on construit une représentation détaillée du système, basée à la fois sur les entrées/sorties identifiées et sur le modèle conceptuel. On choisit pour cela un paradigme (ou méthodologie), et on l'utilise pour définir des objets, leurs attributs, des méthodes (dans le cas d'une implémentation orientée-objet). On met en place également la structure du système, en imbriquant, au besoin, des modèles.

Trois types différents de validité (on parle aussi de précision) existent pour un modèle. L'ensemble de ces trois types de validité peuvent être vus comme formant la relation de modélisation. Chaque type de validité se situe à un niveau d'abstraction différent.

La plus simple, la plus intuitive, est la validité réplicative : pour chaque expérimentation possible dans le cadre expérimental, les évolutions du modèle et du système coïncident au niveau de leurs entrées et sorties ;

La validité prédictive s'appuie sur la validité réplicative et va plus loin que cette dernière : pour un cadre expérimental donné, si on initialise le modèle dans un certain état, et pour les mêmes entrées, ce que le modèle produira en sortie sera analogue avec les sorties produites par le système. De plus, ce modèle doit être à même de prédire les comportements du système qui n'auraient pas été observés lors de l'expérimentation ;

Enfin, la validité structurelle, de plus haut niveau d'abstraction que les deux précédentes, s'assure qu'il y a correspondance entre système et modèle au niveau des transitions (changements d'état) mais attend également, si les modèles sont imbriqués, que leur hiérarchie soit analogue à celle observée dans le système.

Sur la figure de la page suivante sont illustrés ces trois types de validités, en fonction du niveau d'abstraction auquel elles se situent.

La vérification (sous-entendu : du simulateur) est, quant à elle, le processus par lequel on vérifie que le simulateur d'un modèle génère un comportement correct (on parle de simulation « correcte ») de ce dernier. Le terme « comportement correct » signifie plus précisément que les changements d'état, les transitions et les sorties effectuées par le simulateur du modèle sont identiques à ceux du modèle lui-même : plus formellement, un modèle et son simulateur (correct) sont liés par des homomorphismes.

Validité et vérification sont cependant loin d'être des concepts manichéens, bien au contraire. Il serait par exemple absurde de répondre par « oui » ou par « non » à la question : « mon modèle est-il valide ? » car par définition un modèle est une vue simplifiée d'un système, et subit donc au cours de la modélisation une certaine perte de précision. Par conséquent, la réponse à la question précédente dépend des objectifs initiaux et du but recherché. Se poser la question de la validité de notre modèle, à différents niveaux d'abstraction, s'intéresser à la vérification de notre simulateur, permet simplement de lister, d'évaluer, les qualités et les défauts de ces derniers et donc de les améliorer.

En pratique, par abus de langage, on utilise le terme « validation » pour désigner indifféremment les deux notions exposées ci-dessus ainsi que le processus qui les met en œuvre. Examinons à présent ce dernier.

Processus général de type « V&V »

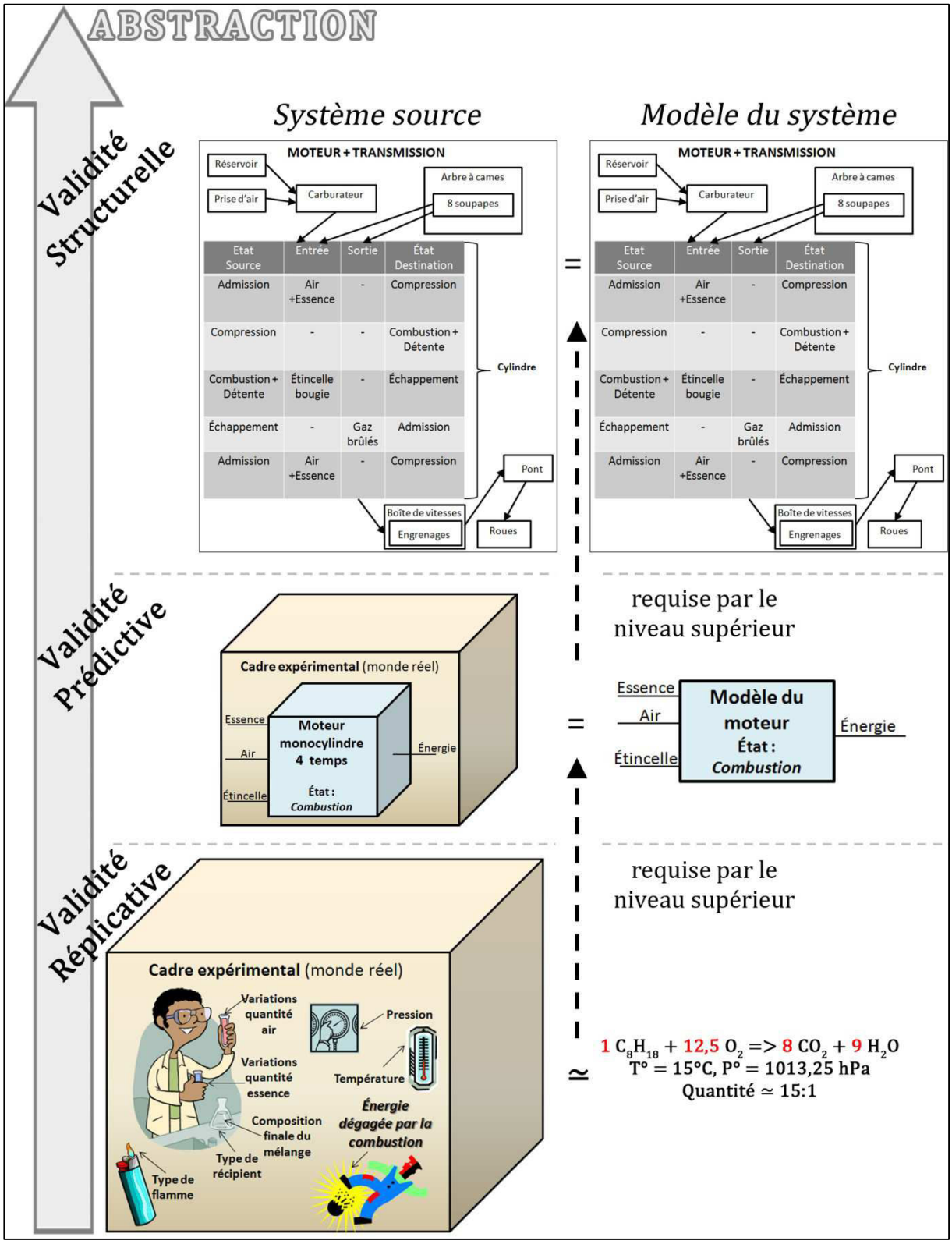
Au vu des concepts et exemples exposés précédemment, on peut décomposer le processus de vérification et de validation en plusieurs étapes. Nous avons choisi de présenter chaque étape sous forme de question-réponse :

- **Est-ce que les concepts du modèle et les concepts du problème étudié sont les mêmes ?** Le but est de s'assurer que ce que l'on appelle « problème »

correspond bien au système réel. Toutes les simplifications, les suppositions, doivent, à ce stade, être justifiées (cf. 1.1.5) ;

- **Est-ce que la conception est valable ?** Confirmer que les concepts du modèle sont fidèlement représentés par la technique de modélisation employée ;
- **Le modèle du système est-il valide ?** Le modèle doit correspondre au système étudié, dans les limites du cadre expérimental. De plus, le niveau de précision du modèle doit être suffisant. Enfin, et cet aspect est très important, il est nécessaire d'évaluer la qualité et la pertinence des données manipulées par le modèle ;
- **Est-ce que l'implémentation logicielle du modèle reflète sa spécification ?** Tester le comportement du système avec le plus de cas possibles (valeurs limites, etc…) ;
- **Est-ce que les résultats fournis par la simulation sont cohérents ?** Comparer ces derniers avec ceux que l'on possède déjà sur le système.

La figure de la page suivante reprend l'exemple du chapitre 1 (le fonctionnement d'un moteur à explosion « 4 temps ») et illustre les trois types de validités que l'on peut appliquer à ce modèle, selon le niveau d'abstraction considéré.



Les trois « validités » du modèle et leurs niveaux d'abstraction respectifs

DOmain Modeling Environment

DOME [DOME 1999] [Engstrom et al. 2000], dont les fondations remontent en 1992, est un méta-outil qui permet le déploiement rapide d'éditeurs de DSL, en se basant sur des notations graphiques. Il a été codé au moyen du langage SmallTalk. DOME possède un méta-formalisme (langage de méta-modélisation) appelé « *DOME Tool Specification*» (ou DTS), langage propriétaire basé sur les entités-relations. Le DTS combine deux notations : une graphique pour spécifier les classes, les propriétés, les contraintes structurelles et les attributs du langage graphique (le tout à l'aide de formes graphiques basiques), et une textuelle, proche de la programmation fonctionnelle (« Lisp-Like »), qui fait office de langage de script pour implémenter des comportements particuliers. La notation graphique est appelée DTSL « DOME Tool Specification Language» et la notation textuelle est baptisée Alter.

Kent Modeling Framework

L'environnement KMF [Kent et al. 2002] est en développement à l'Université du Kent (Canterbury) en Angleterre[1]. Il utilise UML 1.3 et XMI 1.0 comme langage de méta-modélisation, et il est apparemment possible d'exprimer une sémantique statique sur le méta-modèle grâce à des contraintes OCL. Une fois les diagrammes de classe UML créés, ils sont envoyés à une entité appelée ToolGen dont le rôle est de créer un ensemble de fichiers Java pour implémenter l'éditeur correspondant au langage de modélisation désiré. Ces fichiers peuvent être compilés pour générer une GUI spécifique au langage de modélisation. Le langage utilisé pour transformer les modèles est le *Kent Model Transformation Language* [Akehurst et al. 2005].

MOSES

MOSES [Mason 2000] est un environnement de modélisation, simulation, implémentation et vérification, créé en Suisse par la Commission pour la Technologie et l'Innovation (CTI)[2]. Il repose sur un langage de méta-modélisation textuel appelé *Graph Type Definition Language* (GDTL). Ce langage est employé pour décrire des formalismes ou des langages de modélisation [Janneck 2000]. GDTL permet l'expression de syntaxes abstraite et concrète, et propose une visualisation graphique des objets créés. Concernant l'expression de la sémantique statique (i.e. contraintes) rien n'est clairement spécifié dans la littérature relative à cet environnement [Essar et al. 2001]. La sémantique dynamique (sémantique contrôlant l'exécution) peut s'exprimer à l'aide du langage Java sur une plateforme baptisée Hades.

XMF-Mosaic

XMF-Mosaic (sous licence commerciale, société Xactium[3]) est un outil graphique destiné à développer des langages de modélisation (DSL) et à déployer les outils nécessaires au travail avec ces langages incluant éditeurs, analyseurs de code et transformateurs. Il est

[1] http://www.cs.kent.ac.uk/projects/kmf/documents.html
[2] http://www.tik.ee.ethz.ch/~shapes/moses.html
[3] http://www.xactium.com

intégré dans l'environnement Eclipse. Le formalisme de modélisation employé est XMF (*eXecutable Metamodelling Facility*) [Clark et al. 2004] et plus récemment XCORE. La sémantique statique des méta-modèles est décrite au moyen du langage XOCL (eXtensible Object Constraint Language), langage de contraintes balisé de style XML [Ramalho et al. 2003].

Jamda

L'environnement open-source Jamda[1] (*JAva MDA*) permet de générer des applications Java à partir de modèles « métier », grâce à une approche MDA. Il ne supporte pas MOF pour la création de nouveaux méta-modèles mais il est possible d'introduire de nouveaux éléments de modélisation en surclassant les classes Java existantes. Son originalité est de proposer une de génération de code M2T par « parcours de modèle » (ou « *visitor-based* ») (voir 2.2.3.b).

MetaEdit+

MetaEdit+ est un outil sous licence commerciale développé par la société finlandaise MetaCase [Smolander 1991] [Tolvanen et al. 2003] [Metacase 2000] [Metacase 2001]. Il s'agit d'un des plus anciens et des plus connus environnement de méta-modélisation. Il se compose de deux sous-outils distincts :

- Le *MetaEdit+ Workbench*, outil pour créer un langage de modélisation (méta-modélisation),
- Le *MetaEdit+ Modeler*, outil destiné à utiliser un langage de modélisation.

Le premier est basé sur des boîtes de dialogue interactives qui permettent de spécifier des objets du domaine ainsi que leurs relations (méta-modélisation). Chaque objet peut avoir des propriétés. À l'origine il n'était pas possible de créer une hiérarchie mais il semble que les dernières versions de MetaEdit+ intègrent désormais ce paramètre. Le second outil est utilisé pour créer des modèles. Récemment, la société MetaCase a publié une extension gratuite compatible avec Visual Studio[2]. Cet outil est donc à rapprocher des *Software Factories* (voir 2.3.3).

Topcased

Dans le monde de l'open-source, le projet TOPCASED[3], environnement de type CASE, créé en 2008, est un environnement logiciel créé pour faciliter la création de DSL dans le domaine des systèmes critiques embarqués (initialement aéronautique). TOPCASED s'appuie essentiellement sur l'environnement Eclipse (ainsi que plusieurs *plugins* associés) et le formalisme UML. Il supporte, outre la méta-modélisation, la transformation de modèles et est employé par une communauté active d'utilisateurs (particuliers, industriels…).

GME

GME[4] (*Generic Modeling Environment*) [Sztipanovits et al. 1997] est un environnement visuel de méta-modélisation à part entière, développé à l'université Vanderbilt

[1] http://jamda.sourceforge.net
[2] http://visualstudiogallery.msdn.microsoft.com
[3] http://www.topcased.org
[4] http://www.isis.vanderbilt.edu/Projects/gme

de Nashville. Des ponts vers l'environnement EMF ont été établis dans [Bézivin et al. 2005b]. GME permet de spécifier des méta-modèles, au moyen de diagrammes de classe UML, inclus dans le méta-formalisme MetaGME. Ces méta-modèles définissant un environnement de modélisation particulier sont ensuite instanciés dans un environnement de modélisation spécialisé dans le domaine représenté. GME a été l'objet de nombreuses mises à jour, ce qui a conduit à la création de plusieurs versions de cet environnement, qui en est actuellement à sa version 12 (téléchargeable gratuitement). GME est un des outils de type MIC les plus aboutis.

Fujaba

À l'origine surnommé ainsi (*From UML to Java And Back Again*) car il permettait la rétro-ingénierie, Fujaba [Niere et al. 1999] est un outil de type CASE orienté IDM/MDA. Omme beaucoup d'autres outils de ce type, il est désormais disponible sous forme de plugin dans l'environnement Eclipse.

Autres environnements

Il a existé, et existe encore beaucoup d'autres environnements MIC, que nous présentons rapidement ici.

Parmi les environnements industriels qui ont participé aux débuts de l'IDM mais qui n'ont pas « survécu », on peut citer OptimalJ, édité pat la société Compuware[1] (arrêté en 2008 - permettait de modéliser des applications et de générer automatiquement du code Java), et trois MIC orientés MDA, dont les éditeurs étaient d'ailleurs membres de l'OMG : ActStyler[2] de la firme Interactive Objects, Codagen Architect[3] de la société Codagen (utilisaient les concepts spécifiques à MDA : PIM, PSM, code…), et l'environnement de Rational (IBM) XDE[4] (*eXtended Developement Environment*).

[1] http://www.compuware.com
[2] http://www.omg.org/mda/mda_files/ArcStyler5_Whitepaper_220205.pdf
[3] http://www.omg.org/mda/mda_files/CodagenMDA-Final.pdf
[4] http://www.ibm.com/developerworks/rational/products/xde

```
module _'ExtendedDEVS.ecore'
import ecore : 'http://www.eclipse.org/emf/2002/Ecore#/';

package extendeddevs : extendeddevs = 'extendeddevs'
{
	package DEVSModel : DEVSModel = 'DEVSModel'
	{
		abstract class DEVSModel
		{
			invariant nameNotEmpty: self.name.size() > 0;
			property types : Types::Type[*] { ordered composes };
			attribute name : String[1] { ordered };
			property inputPort : Ports::InputPort[*] { ordered composes };
			property outputPort : Ports::OutputPort[+] { ordered composes };
		}
		class CoupledDEVS extends DEVSModel
		{
			invariant testjesaispasquoi: self.EOC->collect(EOC_coupled_out)-
>collect(oclContainer()) = self;
			property contains : DEVSModel[+] { ordered composes };
			property EIC : Coupling::EIC[*] { ordered composes };
			property EOC : Coupling::EOC[+] { ordered composes };
			property IC : Coupling::IC[+] { ordered composes };
		}
		class AtomicDEVS extends DEVSModel
		{
			invariant noIdenticalStateVars:
		    self.is_defined_by->collect(DEVSid)->asSet()->size() = self.is_defined_by-
>collect(DEVSid)->size();
			property current_or_initial_state : Conditions::StateVarComparison[+] {
ordered composes };
			property is_defined_by : DEVSXpressions::StateVar[+] { ordered composes };
			property contains_external_transition_function : DEVSFunctions::DeltaExt[1] {
ordered composes };
			property contains_internal_transition_function : DEVSFunctions::DeltaInt[1] {
ordered composes };
			property contains_lambda_function : DEVSFunctions::Lambda[1] { ordered
composes };
			property contains_time_advance_function : DEVSFunctions::TimeAdvance[1] {
ordered composes };
			property handles : DEVSXpressions::DEVSXpression[*] { ordered composes };
		}
	}
	package Rules : Rules = 'Rules'
	{
		abstract class Rule
		{
			property tests : Conditions::StateVarComparison[+] { ordered composes };
		}
		abstract class TransitionFunctionRule extends Rule
		{
			property changes_state : Actions::StateChangeAction[+] { ordered composes };
		}
		class DeltaExtRule extends TransitionFunctionRule
		{
			property tests_input_event : Conditions::InputPortComparison[1] { ordered
composes };
		}
		class DeltaIntRule extends TransitionFunctionRule;
		class LambdaRule extends Rule
		{
			property sends_message : Actions::OutputAction[1] { ordered composes };
		}
		class TimeAdvanceRule extends Rule
		{
			attribute ta_value : ecore::EInt[1] = '9999999999' { ordered };
		}
	}
	package Types : Types = 'Types'
	{
		abstract class Type;
		class RealType extends Type;
		class StringType extends Type;
		class IntegerType extends Type;
		class CharType extends Type;
```

```
		class BooleanType extends Type;
	}
	package Conditions : Conditions = 'Conditions'
	{
		abstract class Condition;
		abstract class BasicCondition extends Condition
		{
			property right_member : DEVSXpressions::DEVSXpression[1] { ordered };
			attribute comparator : Comparator[?] { ordered };
		}
		abstract class ComplexCondition extends Condition;
		class StateVarComparison extends BasicCondition
		{
			invariant svcTypeConstraint: self.left_member.is_always_typed =
self.right_member.is_always_typed;
			property left_member : DEVSXpressions::StateVar[1] { ordered };
		}
		class InputPortComparison extends BasicCondition
		{
			invariant ipcTypeConstraint: self.left_member.is_always_typed =
self.right_member.is_always_typed;
			property left_member : Ports::InputPort[1] { ordered };
		}
		enum Comparator { serializable }
		{
			EQUALS = 1;
			UPPER_THAN = 2;
			LOWER_THAN = 3;
			LOWER_OR_EQUAL_THAN = 4;
			UPPER_OR_EQUAL_THAN = 5;
		}
		enum LogicalConnector { serializable }
		{
			AND = 1;
			OR = 2;
			XOR = 3;
		}
	}
	package Actions : Actions = 'Actions'
	{
		class Action;
		class ComplexAction extends Action;
		class StateChangeAction extends Action
		{
			invariant stateChangeTypeConstraint: self.state_to_be_changed.is_always_typed
= self.new_value.is_always_typed;
			property state_to_be_changed : DEVSXpressions::StateVar[1] { ordered };
			property new_value : DEVSXpressions::DEVSXpression[1] { ordered };
		}
		class OutputAction extends Action
		{
			invariant outputActionTypeConstraint: self.port.is_always_typed =
self.message.is_always_typed;
			invariant portBelongsToCurrentAtomic: self.port.oclContainer() =
self.oclContainer().oclContainer().oclContainer();
			property port : Ports::OutputPort[1] { ordered };
			property message : DEVSXpressions::DEVSXpression[1] { ordered };
		}
	}
	package DEVSFunctions : DEVSFunctions = 'DEVSFunctions'
	{
		abstract class DEVSFunction;
		abstract class TransitionFunction extends DEVSFunction;
		class DeltaInt extends TransitionFunction
		{
			property is_defined_by_deltaint_rules : Rules::DeltaIntRule[+] { ordered
composes };
		}
		class DeltaExt extends TransitionFunction
		{
			property is_defined_by_deltaext_rules : Rules::DeltaExtRule[*] { ordered
composes };
		}
		class TimeAdvance
		{
```

```
                        property is_defined_by_ta_rules : Rules::TimeAdvanceRule[+] { ordered composes
};
                }
                class Lambda extends DEVSFunction
                {
                        property is_defined_by_lambda_rules : Rules::LambdaRule[*] { ordered composes
};
                }
        }
        package DEVSXpressions : DEVSXpressions = 'DEVSXpressions'
        {
                abstract class DEVSXpression
                {
                        property is_always_typed : Types::Type[1] { ordered };
                }
                abstract class DEVSComplex extends DEVSXpression;
                class StateVar extends DEVSXpression
                {
                        invariant idNotEmpty: self.DEVSid.size() > 0;
                        invariant StateVarTypeConstraint: self.is_always_typed =
self.initial_value.is_always_typed;
                        attribute DEVSid : String[1] { ordered };
                        property initial_value : LitteralBasicValue[?] { ordered };
                }
                abstract class LitteralBasicValue extends DEVSXpression;
                class RealValue extends LitteralBasicValue
                {
                        attribute real_val : ecore::EDouble[1] { ordered };
                }
                class StringValue extends LitteralBasicValue
                {
                        invariant stringIsString: self.is_always_typed.oclType().name = 'StringType';
                        invariant notEmpty: self.str_val.size() > 0;
                        attribute str_val : String[1] { ordered };
                }
                class IntValue extends LitteralBasicValue
                {
                        invariant intIsInt: self.is_always_typed.oclType().name = 'IntegerType';
                        attribute int_val : ecore::EInt[1] { ordered };
                }
                class CharValue extends LitteralBasicValue
                {
                        invariant charIsChar: self.is_always_typed.oclType().name = 'CharType';
                        attribute char_val : ecore::EChar[1] { ordered };
                }
                class BooleanValue extends LitteralBasicValue
                {
                        invariant boolIsBool: self.is_always_typed.oclType().name = 'BooleanType';
                        attribute bool_val : Boolean[1] { ordered };
                }
        }
        package Coupling : Coupling = 'Coupling'
        {
                abstract class Coupling;
                class IC extends Coupling
                {
                        invariant ICtypes: self.IC_out.is_always_typed = self.IC_out.is_always_typed;
                        invariant ICsubmodelOutputPort: self.oclContainer() =
self.IC_out.oclContainer().oclContainer();
                        invariant ICsubmodelInputPort: self.oclContainer() =
self.IC_in.oclContainer().oclContainer();
                        property IC_out : Ports::OutputPort[1] { ordered };
                        property IC_in : Ports::InputPort[1] { ordered };
                }
                class EOC extends Coupling
                {
                        invariant EOCcurrentModelOutputPort: self.oclContainer() =
self.EOC_coupled_out.oclContainer();
                        invariant EOCtypes: self.EOC_coupled_out.is_always_typed =
self.EOC_out.is_always_typed;
                        invariant EOCsubmodelOutputPort: self.oclContainer() =
self.EOC_out.oclContainer().oclContainer();
                        property EOC_out : Ports::OutputPort[1] { ordered };
                        property EOC_coupled_out : Ports::OutputPort[1] { ordered };
                }
```

```
		class EIC extends Coupling
		{
			invariant EICtypes: self.EIC_coupled_in.is_always_typed =
self.EIC_in.is_always_typed;
			invariant EICcurrentModelInputPort: self.oclContainer() =
self.EIC_coupled_in.oclContainer();
			invariant EICsubmodelInputPort: self.oclContainer() =
self.EIC_in.oclContainer().oclContainer();
			property EIC_in : Ports::InputPort[1] { ordered };
			property EIC_coupled_in : Ports::InputPort[1] { ordered };
		}
	}
	package Ports : Ports = 'Ports'
	{
		abstract class Port extends DEVSXpressions::DEVSXpression
		{
			attribute portID : String[?] { ordered };
		}
		class InputPort extends Port;
		class OutputPort extends Port;
	}
}
```

ANNEXE 5 : Code ATL de la transformation BasicDEVSToDEVS

```
--@path BasicDEVS=/DEVS/BasicDEVS.ecore
-- @path ExtendedDEVS=/DEVS/ExtendedDEVS.ecore


module BasicToExtended;
create OUT: ExtendedDEVS from IN: BasicDEVS;

helper context BasicDEVS!Port def: isInPort(): Boolean =
    if self.oclIsTypeOf(BasicDEVS!InPort) then
        true
    else
        false
    endif;

helper context BasicDEVS!Transition def: isAutoTransition(): Boolean =
    if self.oclIsTypeOf(BasicDEVS!AutoTransition) then
        true
    else
        false
    endif;

helper context BasicDEVS!Transition def: isEventTransition(): Boolean =
    if self.oclIsTypeOf(BasicDEVS!EventTransition) then
        true
    else
        false
    endif;



rule model2AtomicModel {
    from
        model: BasicDEVS!Model
    to
        devsModel2: ExtendedDEVS!AtomicDEVS (
            name <- model.model_name,
            is_defined_by <- sv,
            contains_internal_transition_function <- dint,
            contains_external_transition_function <- dext,
            contains_lambda_function <- lambda,
            contains_time_advance_function <- ta,
            types <- sT,

            --collecte des états ci-dessous et création des LBV correspondantes
            handles <- model.is_composed_of -> collect(e | thisModule.CreateLBVStringValue(e.state_name, sT)),

            --collecte des évènements d'entrée ci-dessous et création des LBV correspondantes
            handles <- model.evolves_with_transitions -> select(e | not(e.isAutoTransition())) ->
            collect(e | thisModule.CreateLBVStringValue(e.is_triggered_by.value, sT)),

            --collecte des évènements de sortie ci-dessous et création des LBV correspondantes
            handles <- model.evolves_with_transitions -> select(e | e.isAutoTransition())->
            collect(e | thisModule.CreateLBVStringValue(e.triggers.value, sT)) ,

            inputPort <- model.has_ports -> select(e | e.isInPort()) -> collect(e |
                    thisModule.InPort2AtomicInputPort(e, sT)),
            outputPort <- model.has_ports -> select(e | not(e.isInPort())) -> collect(e |
                    thisModule.OutPort2AtomicOutputPort(e, sT))
        ),
        sv: ExtendedDEVS!StateVar (
            DEVSid <- 'StateSet',
            is_always_typed <- sT,
            --recherche, parmi les StringValues déjà créées, laquelle correspond au nom de l'état initial du modèle BasicDEVS
            initial_value <- devsModel2.handles -> select(e | e.str_val=model.current_or_initial_state.state_name)
        ),

        ta: ExtendedDEVS!TimeAdvance (
            is_defined_by_ta_rules <- model.is_composed_of -> collect(e | thisModule.
                    createTARule(e, sv, devsModel2))
        ),
        dint: ExtendedDEVS!DeltaInt (
            is_defined_by_deltaint_rules <- model.evolves_with_transitions -> select(e |
                    e.isAutoTransition()) -> collect(e | thisModule.createDeltaIntRule(e,
                    sv, devsModel2)) --sans le collect il n'y aurait qu'une fonction
                    -- trouvée
        ),

        lambda: ExtendedDEVS!Lambda (
        is_defined_by_lambda_rules <- model.evolves_with_transitions -> select(e |
                    e.isAutoTransition()) -> collect(e | thisModule.createLambdaRule(e,sv,devsModel2))
        ),

        dext: ExtendedDEVS!DeltaExt (
        is_defined_by_deltaext_rules <- model.evolves_with_transitions -> select(e | not(e.isAutoTransition()))
        --sans le collect il n'y aurait qu'une fonction trouvée -> collect(e | thisModule.createDeltaExtRule(e, sv, devsModel2))
        ),

        sT: ExtendedDEVS!StringType

}

rule CreateLBVStringValue (name: String, st: ExtendedDEVS!StringType){
    to
        lbv: ExtendedDEVS!StringValue
    do {
        lbv.is_always_typed<-st;
        lbv.str_val<-name;
        lbv;
    }
}

rule createDeltaExtRule (eventtrans: BasicDEVS!EventTransition,sv: ExtendedDEVS!StateVar,
        ip: ExtendedDEVS!AtomicDEVS ){
    to
        deltaExtRule: ExtendedDEVS!DeltaExtRule (
        )
                do {
                    if (not(eventtrans.isAutoTransition())) {

                        deltaExtRule.tests<-thisModule.createSVC(eventtrans.source_state,
                                sv, ip);
                        deltaExtRule.tests_input_event<-thisModule.createIPC(eventtrans.
```

```
    to
        deltaExtRule: ExtendedDEVS!DeltaExtRule (
        )
                do {
                    if (not(eventtrans.isAutoTransition())) {

                        deltaExtRule.tests<-thisModule.createSVC(eventtrans.source_state,
                                sv, ip);
                        deltaExtRule.tests_input_event<-thisModule.createIPC(eventtrans.
                                is_triggered_by, sv, ip);
                        deltaExtRule.changes_state<-thisModule.createChange(eventtrans.
                                target_state, sv, ip);
                        deltaExtRule;
                }
            }
        }

rule createDeltaIntRule (at: BasicDEVS!AutoTransition, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!AtomicDEVS){
    to
        dintrule: ExtendedDEVS!DeltaIntRule (
            tests <- thisModule.createSVC(at.source_state, sv, m),
            changes_state <- thisModule.createChange(at.target_state, sv, m)
        )

    do {
        dintrule;
    }
}

rule createLambdaRule (at: BasicDEVS!AutoTransition, sv: ExtendedDEVS!StateVar, m: ExtendedDEVS!AtomicDEVS) {
    to
        lambdarule: ExtendedDEVS!LambdaRule (
            tests <- thisModule.createSVC(at.source_state, sv, m),
            sends_message <- thisModule.createOutputAction(at.triggers, m)
        )
    do {
       lambdarule;
    }
}

rule createSVC(s: BasicDEVS!State, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!AtomicDEVS){    --crée une StateVarComparison à partir d'un état, de
        -- la State Var globale et du modèle DEVS passés en paramètre
    to
        creeSVC: ExtendedDEVS!StateVarComparison (
            left_member <- sv,
            right_member <- m.handles -> select(e | e.str_val = s.state_name)
        )
        do {
        creeSVC;
    }
}

rule createIPC(s: BasicDEVS!InputEvent, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!AtomicDEVS){    --crée une InputPortComparison à partir d'un état
        -- passé
        -- en paramètre
    to
        creeIPC: ExtendedDEVS!InputPortComparison (
            left_member <- m.inputPort -> select(e | e.portID = s.occurs_on.portID),
            right_member <- m.handles -> select(e | e.str_val = s.value)
        )
        do {
        creeIPC;
    }
}

rule createChange(targetS: BasicDEVS!State, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!AtomicDEVS){    --création de l'action StateChangeAction, prend en
        -- parmametre un état, la variable globale, le modèle
    to
        creechange: ExtendedDEVS!StateChangeAction (
            state_to_be_changed <- sv,
            new_value <- m.handles -> select(e | e.str_val = targetS.state_name)
        )
        do {
        creechange;
    }
}

rule createOutputAction(p: BasicDEVS!OutputEvent, m: ExtendedDEVS!AtomicDEVS){
 --création de l'action OutputAction
    to
        creeout: ExtendedDEVS!OutputAction (
            message <- m.handles -> select(e | e.str_val=p.value),
            port <- m.outputPort -> select(e | e.portID=p.is_sent_on.portID)
        )
        do {
        creeout;
    }
}

rule createTARule (s: BasicDEVS!State, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!LitteralBasicValue){
    to
        tarule: ExtendedDEVS!TimeAdvanceRule (
        )
        do
        {
            tarule.tests<-thisModule.createSVC(s, sv, m);
            tarule.ta_value<-s.maxDuration;
```

```
206             tarule;
207             }
208 }
209
210 rule InPort2AtomicInputPort (ip: BasicDEVS!InPort, st: ExtendedDEVS!StringType){
211     to
212         extendedinputport: ExtendedDEVS!InputPort (
213         )
214     do{
215             extendedinputport.portID <- ip.portID;
216             extendedinputport.is_always_typed<-st;
217             extendedinputport;
218     }
219 }
220
221 rule OutPort2AtomicOutputPort (ip: BasicDEVS!OutPort, st: ExtendedDEVS!StringType){
222     to
223         extendedoutputport: ExtendedDEVS!OutputPort (
224         )
225     do{
226             extendedoutputport.portID <- ip.portID;
227             extendedoutputport.is_always_typed<-st;
228             extendedoutputport;
229     }
230 }
231
```

```
--@path FSM=/DEVS/FSM.ecore
-- @path ExtendedDEVS=/DEVS/ExtendedDEVS.ecore

module FSM2DEVS;
create OUT: ExtendedDEVS from IN: FSM;

helper context FSM!State def: isInitial(): Boolean =
    if self.isInitial=true then
        true
    else
        false
    endif;

helper context FSM!State def: isFinal(): Boolean =
    if self.isFinal=true then
        true
    else
        false
    endif;

rule FSM2AtomicModel {
    from
        fsm: FSM!ClassicFSM
        using {
    etatinitial : FSM!ClassicState = fsm.hasState -> select (e | e.isInitial());
    }
    to
        devsModel: ExtendedDEVS!AtomicDEVS (
            name <- fsm.name,
            is_defined_by <- sv,
            contains_internal_transition_function <- dint,
            contains_external_transition_function <- dext,
            contains_lambda_function <- lambda,
            contains_time_advance_function <- ta,
            types <- sT,
            inputPort <- iPort,
            outputPort <- oPort,
            handles <- thisModule.CreateLBVStringValue('__the word has been recognized__', sT),

            --collecte des labels possibles sur les étiquettes = langage reconnu par le FSM
            handles <- fsm.hasTransition -> collect(e | thisModule.CreateLBVStringValue(e.label, sT)),

            --collecte des états ci-dessous (sauf états initiaux) et création des LBV correspondantes
            handles <- fsm.hasState -> collect(e | thisModule.CreateLBVStringValue(e.stateName, sT))
        ),

        sv: ExtendedDEVS!StateVar (
            DEVSid <- 'FSMState',
            is_always_typed <- sT,
            -- utilise first sinon il croit que c'est une collection (doublons?)
            initial_value <- devsModel.handles -> select (e | e.str_val=etatinitial->first().stateName)
        ),

        iPort : ExtendedDEVS!InputPort (
            portID <- 'FSMReadInputPort',
            is_always_typed <- sT
            ),
        oPort : ExtendedDEVS!OutputPort (
            portID <- 'FSMReadOutputPort',
            is_always_typed <- sT
            ),

        ta: ExtendedDEVS!TimeAdvance (
        is_defined_by_ta_rules <- fsm.hasState -> collect(e | thisModule.createTARule(e, sv, devsModel))
        ),

        dint: ExtendedDEVS!DeltaInt (
            is_defined_by_deltaint_rules <- fsm.hasTransition -> select (e | e.targetState.isFinal())
            -> collect(e | thisModule.createDeltaIntRule(e,sv,devsModel))
        ),
        dext: ExtendedDEVS!DeltaExt (
            is_defined_by_deltaext_rules <- fsm.hasTransition -> collect(e | thisModule.createDeltaExtRule(e,sv,devsModel))
                    --sans le collect il n'y aurait qu'une fonction trouvée
        ),
        lambda: ExtendedDEVS!Lambda (
            is_defined_by_lambda_rules <- fsm.hasTransition -> select (e | e.targetState.isFinal())
            -> collect(e | thisModule.createLambdaRule(e, sv, devsModel))
        ),
        sT: ExtendedDEVS!StringType
}

rule CreateLBVStringValue (name: String, st: ExtendedDEVS!StringType){
    to
        lbv: ExtendedDEVS!StringValue
    do {
        lbv.is_always_typed<-st;
        lbv.str_val<-name;
        lbv;
    }
}

rule createDeltaExtRule (eventtrans: FSM!Transition, sv: ExtendedDEVS!StateVar,
```

```
        a: ExtendedDEVS!AtomicDEVS ){
    to
        deltaExtRule: ExtendedDEVS!DeltaExtRule (
        )
                do {
                        deltaExtRule.tests<-thisModule.createSVC(eventtrans.sourceState, sv, a);
                        deltaExtRule.tests_input_event<-thisModule.createIPC(eventtrans, sv, a);
                        deltaExtRule.changes_state<-thisModule.createChange(eventtrans, sv, a);
                        deltaExtRule;
            }
        }

rule createDeltaIntRule (trans: FSM!Transition, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!AtomicDEVS){
    to
        dintrule: ExtendedDEVS!DeltaIntRule (
            tests <- thisModule.createSVC(trans.sourceState, sv, m),
            changes_state <- thisModule.createChange(trans, sv, m)
        )

    do {
        dintrule;
    }
}

rule createLambdaRule (tr:FSM!Transition, sv: ExtendedDEVS!StateVar, m: ExtendedDEVS!AtomicDEVS) {
    to
        lambdarule: ExtendedDEVS!LambdaRule (
            tests <- thisModule.createSVC(tr.sourceState, sv, m),
            sends_message <- thisModule.createOutputAction(tr,m)
        )
    do {
       lambdarule;
    }
}

rule createTARule (s: FSM!State, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!LitteralBasicValue){
    to
        tarule: ExtendedDEVS!TimeAdvanceRule (
        )
        do
        {
            tarule.tests<-thisModule.createSVC(s, sv, m);
            if (s.isFinal()=true) {
            tarule.ta_value<-4000;
            }
            else {
            tarule.ta_value<-9999999;
            }
            tarule;
            }
}

rule createOutputAction(fsm: FSM!Transition, m: ExtendedDEVS!AtomicDEVS){
 --création de l'action OutputAction
    to
        creeout: ExtendedDEVS!OutputAction (
            message <- m.handles -> select(e | e.str_val='__the word has been recognized__'),
            port <- m.outputPort
        )
        do {
        creeout;
    }
}

rule createSVC(s: FSM!State, sv: ExtendedDEVS!StateVar, m:ExtendedDEVS!AtomicDEVS){
    --crée une StateVarComparison à partir d'un état, de
    -- la State Var globale et du modèle DEVS passés en paramètre
    to
        creeSVC: ExtendedDEVS!StateVarComparison (
            left_member <- sv,
            right_member <- m.handles -> select (e | e.str_val = s.stateName)
        )
        do {
        creeSVC;
    }
}

rule createIPC(fsm: FSM!Transition, sv: ExtendedDEVS!StateVar, m:ExtendedDEVS!AtomicDEVS){
--crée une InputPortComparison à partir d'un état passé en paramètre

    to
        creeIPC: ExtendedDEVS!InputPortComparison (
            left_member <- m.inputPort,
            right_member <- m.handles -> select(e | e.str_val = fsm.label)
        )
        do {
        creeIPC;
    }
}

```

```
rule createChange(fsm: FSM!Transition, sv: ExtendedDEVS!StateVar, m:
        ExtendedDEVS!AtomicDEVS){    --création de l'action StateChangeAction, prend en
        -- parmametre un état, la variable globale, le modèle
    to
        creechange: ExtendedDEVS!StateChangeAction (
            state_to_be_changed <- sv,
            new_value <- m.handles -> select(e | e.str_val = fsm.targetState.stateName)
        )
        do {
        creechange;
    }
}
```

ANNEXE 7 : Code Acceleo de la transformation de DEVS vers PyDEVS

```
[comment encoding = ISO-8859-1/]
[module ExtendedToPythonGenerationFile('extendeddevs')]

[template public getFirst(a : DEVSModel) ? (a.ancestors()->isEmpty()=true)]
[comment @main /]
[file (a.name.concat('AutoGen.py'), false, 'ISO-8859-1')]
# -*- coding: iso-8859-15 -*-
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
# [a.name.concat('AutoGen.py')/] --- Based on the template for atomic- and coupled-DEVS descriptive
classes by
#                       --------------------------------
#                            Jean-Sébastien  BOLDUC
#                                  Hans Vangheluwe
#                         McGill University (Montréal)
#                       --------------------------------
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##

# Add the directory where pydevs lives to Python's import path
import sys
import os.path
sys.path.append(os.path.expanduser('~/src/projects/PythonDEVS/'))

# Import code for DEVS model representation:
import pydevs
from pydevs.devs_exceptions import *
from pydevs.infinity import *
from pydevs.DEVS import *

# Import for uniform random number generators
from random import uniform
from random import randint

#    ======================================================================    #
[generateModel(a)/]
[generateExperiment(a)/]
[/file]
[/template]

[template public generateModel(a : DEVSModel)]
[/template]

[template public generateModel(a : AtomicDEVS)]

class [a.name.concat('EncapState')/]:

    """Encapsulate the Atomic DEVS' state
    This is not absolutely necessary, but it makes
    it easier to
    (1) stick to the DEVS formalism
    (and not modify the state "behind the back of the simulator" --
    this will be enforced once we have a DEVS compiler), and
    (2) to produce a clean simulation trace (thanks to __str__).
    """

    ###

    def __init__(self, [for (v:StateVar|a.is_defined_by) separator(', ')][v.toString()/][/for]):
        """Constructor (parameterizable).
        """

        # set the initial state
        # the state may be multi-dimensional:
        # consist of multiple state variables
        # (make a local copy of the constructor's arguments)

        [for (v:StateVar|a.is_defined_by)]
        self.[v.toString()/]=[v.toString()/]
        [/for]

    def get(self):
        return {[for (v:StateVar|a.is_defined_by) separator(', ')]'[v.toString()/]':
self.[v.toString()/][/for]}

    def __str__(self):
```

```
        return [for (v:StateVar|a.is_defined_by) separator(' + " and " + ')]" [v.toString()/] = "+
str(self.[v.toString()/])[/for]

class [a.name/](AtomicDEVS):

    ###

    def __init__(self, name=None):
        """Constructor (parameterizable).
        Giving a name (string) to a model is not mandatory
        (a globally unique name "A<i>" is assigned by default).
        If a name is given, the simulation trace becomes far more
        readable.
        """

        # Always call parent class' constructor FIRST:
        AtomicDEVS.__init__(self, name)

        # TOTAL STATE:
        #  Define 'state' attribute (initial sate):
        self.state = [a.name.concat('EncapState')/]([for
(v:StateVar|a.is_defined_by)separator(',')][v.initial_value.toString()/][/for])

        # ELAPSED TIME:
        #  Initialize 'elapsed time' attribute if required.
        #  (by default (if not set by user), self.elapsed is initialized to 0.0)
        self.elapsed = 0
        #  With elapsed time for example initially 1.1 and
        #  this SampleADEVS initially in
        #  a state which has a time advance of 60,
        #  there are 60-1.1 = 58.9 time-units remaining until the first
        #  internal transition

        # HV: check in simulator that elapsed can not be larger than
        # timeAdvance

        # PORTS:
        #  Declare as many input and output ports as desired
        #  (usually store returned references in local variables):
        [comment Déclaration des ports d'entrée et de sortie /]
        [declarePort(a)/]

    def extTransition(self):
        """External Transition Function."""

        # Compute and return the new state based (typically) on current
        # State, Elapsed time parameters and calls to 'self.peek(self.IN)'.
        # When extTransition gets invoked, input has arrived
        # on at least one of the input ports. Thus, peek() will
        # return a non-None value for at least one input port.
        [comment récupération des évènements sur chaque port d'entrée /]

                [comment vérification que dext n'est pas vide/]
        [if a.contains_external_transition_function.is_defined_by_deltaext_rules->isEmpty()=false]

        [let lavar : Integer = a.inputPort->collect(InputPort)->count(InputPort)]
        ## j'ai compte [lavar/] ports d'entree

            [for (pin: InputPort | a.inputPort)]
        input[pin.toStringPort()/] = self.peek(self.[pin.toStringPort()/])
        [/for][/let]

        [comment récupération de l'état courant /]
        [retrieveState(a)/]

        [for (dext: DeltaExtRule |
a.contains_external_transition_function.is_defined_by_deltaext_rules)]
        if input[dext.tests_input_event.toStringCond()/]:
            if [dext.toStringMultipleCond()/]:
                [dext.toStringAct()/]
        [/for]
        else :
            raise DEVSException(\
        "unknown state <%s> in [a.name/] external transition function"\
            % state)
```

```
        [/if]

    def intTransition(self):
        """Internal Transition Function.
        """

        # Compute and return the new state  based (typically) on current State.
        # Remember that intTransition gets called after the outputFnc,
        # timeAdvance time after the current state was entered.
        [comment dans ce template on va générer le delta int pour chaque état /]

        [comment récupération de l'état courant /]
        [retrieveState(a)/]

        if [for (dint: DeltaIntRule |
a.contains_internal_transition_function.is_defined_by_deltaint_rules)
       separator ('        elif ')] [dint.toStringMultipleCond()/]:
            [dint.toStringAct()/]
        [/for]
        else :
            raise DEVSException(\
            "unknown state <%s> in [a.name/] internal transition function"\
        % state)

    def outputFnc(self):
        """Output Function.
        """
        # Send events via a subset of the atomic-DEVS'
        # output ports by means of the 'poke' method.
        # More than one port may be poke-d.
        # The content of the messages is based (typically) on the current State.
        #
        # BEWARE: ouput is based on the CURRENT state
        # and is produced BEFORE making the internal transition.
        # Often, we want output based on the NEW state (which
        # the system will transition to by means of the internal
        # transition. The solution is to keep the NEW state in mind
        # when poke-ing the output.
        # This will only work if the internal transition function
        # is deterministic !
        [comment dans ce template on génère les lambda rules en tenant compte du fait/]
        [comment que la règle générée est peut être la dernière de la liste/]

                [comment vérification que lambda n'est pas vide/]
                [if a.contains_lambda_function.is_defined_by_lambda_rules->isEmpty()=false]

        [comment récupération de l'état courant /]
        [retrieveState(a)/]

        if [for (lambda: LambdaRule | a.contains_lambda_function.is_defined_by_lambda_rules)
           separator ('        elif ')] [lambda.toStringMultipleCond()/]:
            self.poke(self.[lambda.sends_message.port.toStringPort()/],[lambda.toStringAct()/])
        [/for]
                [/if]

    def timeAdvance(self):
        """Time-Advance Function.
        """
            # Compute 'ta', the time to the next scheduled internal transition,
        # based (typically) on the current State.
            [comment dans ce template on génère les lambda rules en tenant compte du fait/]
        [comment que la règle générée est peut être la dernière de la liste/]


        [comment récupération de l'état courant /]
        [retrieveState(a)/]

        if [for (taRule: TimeAdvanceRule | a.contains_time_advance_function.is_defined_by_ta_rules)
       separator('        elif ')] [taRule.toStringMultipleCond()/]:
            return [taRule.toStringTA()/]
        [/for]
        else :
            raise DEVSException(\
            "unknown state <%s> in [a.name/] time advance function"\
```

```
            % state)

[/template]

[template public generateModel(c : CoupledDEVS)]
[for (sub: DEVSModel | c.contains) separator('\n')]
[generateModel(sub)/]
[/for]

#    ====================================================================    #
class [c.name/](CoupledDEVS):
    """Sample Coupled-DEVS descriptive class: compulsory material.
    """
###
    def __init__(self, name=None):
        """Constructor (parameterizable).
        Giving a name (string) to a model is not mandatory
        (a globally unique name "C<i>" is assigned by default).
        If a name is given, the simulation trace becomes far more
        readable.
        """

        # Always call parent class' constructor FIRST:
        CoupledDEVS.__init__(self, name)

        # PORTS:
    #  Declare as many input and output ports as desired
    #  (usually store returned references in local variables).
    #  Note that giving a name (string) to a port is not mandatory
    #  (a globally unique name "port<i>" is assigned by default).
    #  If a name is given, the simulation trace becomes far more
    #  readable however.
        [comment Déclaration des ports d'entrée et de sortie /]
        [declarePort(c)/]

        #  Note: at the root level, the Coupled DEVS will often
    #  not have any InPorts nor OutPorts (i.e., be "autonomous").


        # SUB-MODELS:
    #  Declare as many sub-models as desired. Method 'addSubModel' takes as a
    #  parameter an INSTANCE of an atomic- or coupled-DEVS descriptive class,
    #  and returns a reference to that instance (usually kept in a local
    #  variable for future reference):

            [for (sub: DEVSModel | c.contains)]
        self.[sub.name.toLower()/] = self.addSubModel([sub.name/](name="[sub.name.toLower()/]"))
        [/for]

    # COUPLINGS:
    #  Method 'connectPorts' takes two parameters 'p1' and 'p2' which are
    #  references to ports.
    #  The coupling is to BEGIN at 'p1' and to END at 'p2'.
    #  Each input/output port can have
    #  multiple incoming/outgoing connections.
    #  Note how PythonDEVS does NOT support Z_i,j transfer
    #  functions (as defined in DEVS) yet.
    #  On the other hand, PythonDEVS does have named ports.
        [declareCouplage(c)/]

    def select(self, immList):
        """Select function.
        Parameter 'immList' is a list of references to sub-models which
        have a scheduled internal transition occuring at the same time.
        To implement tie-breaking, select() needs to return
        exactly ONE submodel immList(i).
        """

###
[/template]


[comment AFFICHAGE DES DEVSEXPRESSION /]
[comment Le premier template vide correspond à la classe abstraite /]
[comment Les StateVar, les LBV...etc, sont des DEVSExpression, les Port sont gérés à part sinon BUG /]
```

```
[template public toString(devsexp : DEVSXpression)]
[/template]
[comment Un template pour la classe DEVSComplex qui invoque les autres /]

[comment Un template pour chaque sous-classe terminale /]
[template public toString(sv : StateVar)]
[sv.DEVSid/]
[/template]
[template public toString(lbv : LitteralBasicValue)]
[/template]
[template public toString(lbv : StringValue)]
"[lbv.str_val/]"
[/template]
[template public toString(lbv : IntValue)]
[lbv.int_val/]
[/template]
[template public toString(rv : RealValue)]
[rv.real_val/]
[/template]
[template public toString(lbv : CharValue)]
"[lbv.char_val/]"
[/template]
[template public toString(lbv : BooleanValue) post(trim())]
[if (lbv.bool_val=true)]True
[else]False
[/if]
[/template]

[comment affiche correctement le nom du port : en majuscules, pour respecter les conventions PyDEVS/]
[template public toStringPort(devsPort : Port)]
[devsPort.portID.toUpper()/]
[/template]

[comment dans ce template on va vérifier si on a un ta infini défini par la valeur arbitraire 999999 /]
[template public toStringTA(tar : TimeAdvanceRule)]
[if (tar.ta_value >= 999999)]
INFINITY
[else]
[tar.ta_value/]
[/if]
[/template]

[comment AFFICHAGE DES OPERATEURS avec la fonction pos-traitement "trim"]
[comment qui évite de renvoyer un retour chariot par ligne de if /]
[template public rewriteComp(comp : Comparator) post(trim())]
[if self.toString()='EQUALS']==
[elseif self.toString()='UPPER_THAN']>
[elseif self.toString()='LOWER_THAN']<
[elseif self.toString()='UPPER_OR_EQUAL_THAN']>=
[elseif self.toString()='LOWER_OR_EQUAL_THAN']<=
[/if]
[/template]

[comment ce template récupère le ou les StateVar via l'EncapState ainsi que]
[comment leurs valeurs courantes (dictionnaire) via la méthode get() /]
[template public retrieveState(a : AtomicDEVS)]
statedico = self.state.get()
state = self.state
[/template]

[template public toStringMultipleCond(rule : Rule)]
[for (cond : Condition | rule.tests) separator (' and ')][cond.toStringCond()/][/for]
[/template]

[comment Ce template appelle toString sur les membres gauche et droit et rewriteComp()]
[comment pour afficher le comparateur selon les spécifs du langage  /]
[template public toStringCond(cond : Condition)]
[/template]
[template public toStringCond(svc : StateVarComparison)]
statedico['['/]'[svc.left_member.toString()/]']'/] [svc.comparator.rewriteComp()/]
[svc.right_member.toString()/]
[/template]
[template public toStringCond(ipc : InputPortComparison)]
[ipc.left_member.toStringPort()/] [ipc.comparator.rewriteComp()/] [ipc.right_member.toString()/]
```

```
[/template]

[comment Ce template génère le code correspondant aux Actions dans MetaDEVS /]
[comment Il y a récupération du ou des changements d'état spécifiés par l'utilisateur/]
[comment Si pas tous les chagements ne sont spécifiés (état multi-dimensionnel), on remet les anciennes
valeurs /]
[template public toStringAct(rule : Rule)]
[/template]
[template public toStringAct(tfRule : TransitionFunctionRule)]
[for (sca : StateChangeAction | tfRule.changes_state)separator('\n')]
state.[sca.state_to_be_changed.DEVSid/]=[sca.new_value.toString()/]
[/for]
return state
[/template]
[template public toStringAct(lambdaRule : LambdaRule)]
[lambdaRule.sends_message.message.toString()/]
[/template]

[comment Ce template déclare les ports d'entrée et de sortie d'un DEVSModel/]
[comment en PyDEVS, syntaxe identique pour les modèles atomiques et couplés/]
[template public declarePort(aDEVSModel : DEVSModel)]
[for (pin: InputPort | aDEVSModel.inputPort)]
self.[pin.toStringPort()/] = self.addInPort(name="[pin.toStringPort()/]")
[/for]
[for (pout: OutputPort | aDEVSModel.outputPort)]
self.[pout.toStringPort()/] = self.addOutPort(name="[pout.toStringPort()/]")
[/for]
[/template]

[template public declareCouplage(aCoupledDEVS : CoupledDEVS)]
[for (ic: IC | aCoupledDEVS.IC)]
# INTERNAL COUPLING
self.connectPorts(self.[ic.IC_out.eContainer().oclAsType(DEVSModel).name.toLower()/].[ic.IC_out.toStrin
gPort()/],
self.[ic.IC_in.eContainer().oclAsType(DEVSModel).name.toLower()/].[ic.IC_in.toStringPort()/])
[/for]

[for (eic: EIC | aCoupledDEVS.EIC)]
# EXTERNAL INPUT COUPLING
self.connectPorts(self.[eic.EIC_coupled_in.toStringPort()/], s
elf.[eic.EIC_in.eContainer().oclAsType(DEVSModel).name.toLower()/].[eic.EIC_in.toStringPort()/])
[/for]

[for (eoc: EOC | aCoupledDEVS.EOC)]
# EXTERNAL OUTPUT COUPLING
self.connectPorts(self.[eoc.EOC_out.eContainer().oclAsType(DEVSModel).name.toLower()/].[eoc.EOC_out.toS
tringPort()/],
self.[eoc.EOC_coupled_out.toStringPort()/])
[/for]
[/template]
```

ANNEXE 7 : Code Acceleo de la transformation de DEVS vers PyDEVS

```
[template public generateExperiment(aDEVSModel : DEVSModel)]
[comment ce template génère un fichier .py à part, destiné à lancer la simulation/]
[file (aDEVSModel.name.concat('ExperimentAutoGen.py'), false, 'ISO-8859-1')]
# -*- coding: iso-8859-15 -*-
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
# [aDEVSModel.name.concat('ExperimentAutoGen.py')/].py --- Template for PythonDEVS experiments
#                       -------------------------------
#                                November 2005
#                               Hans Vangheluwe
#                         McGill University (Montréal)
#                       -------------------------------
#       created: 20/11/05
# last modified: 20/11/05
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##

# Add the directory where pydevs lives to Python's import path
import sys
import os.path
sys.path.append(os.path.expanduser('~/src/projects/PythonDEVS/'))

# Import code for model simulation:
import pydevs
from pydevs.infinity import *
from pydevs.simulator import *

#    ====================================================================    #

# Import the model to be simulated
from [aDEVSModel.name.concat('AutoGen')/] import *

#    ====================================================================    #

# 1. Instantiate the (Coupled or Atomic) DEVS at the root of the
#  hierarchical model. This effectively instantiates the whole model
#  thanks to the recursion in the DEVS model constructors (__init__).
#
[aDEVSModel.name.toLower()/] = [aDEVSModel.name/](name="[aDEVSModel.name.toLower()/]")

#    ====================================================================    #

# 2. Link the model to a DEVS Simulator:
#  i.e., create an instance of the 'Simulator' class,
#  using the model as a parameter.
sim = Simulator([aDEVSModel.name.toLower()/])

#    ====================================================================    #

# 3. Run the simulator on the model until a termination_condition is
# met, that is: the termination_condition function returns True.
# The termination_condition function receives two arguments
# from the simulation kernel:
#  * a reference to the root model
#  * the current time
#
# sim.simulate(termination_condition= ..., verbose= ...)
#

# Typically, we set verbose=True when we want to debug a model and
# verbose=False when we are only interested in statistics/performance
# measures at the end of the simulation.

# Some typical termination_condition functions
#

# A. Simulate forever.
#    The termination_condition function never returns True.
#
def terminate_never(model, clock):
    return False

#sim.simulate(termination_condition=terminate_never,
#             verbose=True)

# B. Simulate until a specified end_time is reached.
#    When the end_time is reached/exceeded, the
```

```
#    termination_condition function returns True.
#
def terminate_whenEndTimeReached(model, clock, end_time=5000):
    if clock >= end_time:
      return True
    else:
      return False

sim.simulate(termination_condition=terminate_whenEndTimeReached,
             verbose=True)

# C. Simulate until a specified end_state is reached.
#    When the end_state is reached, the
#    termination_condition function returns True.
#
#def terminate_whenStateIsReached(model, clock, end_state="someState"):
#    aSubModel = model.getSubModel(name="subModelName")
#   if aSubModel.state.get() == end_state:
#        return True
#    else:
#        return False

#sim.simulate(termination_condition=terminate_whenStateIsReached,
#             verbose=True)

#    ================================================================    #

# Print statistics/performance measures
#
# Typically, extra state variables are added (but not printed
# in the behaviour trace) and updated in models to keep track
# of relevant performance measures such as
#  * mininum, maximum, mean, standard deviation of state variables
#  * distributions (GPSS TABULATE) of state variables' values
#
# At the end of the simulation experiment, these statistics are printed.
#
# Note that typically, verbose=True when we want to debug a model and
# verbose=False when we are only interested in statistics/performance
# measures at the end of the simulation.

#    ================================================================    #

# Note how multiple experiments can be performed in sequence ...
# (or in some control structure if for example a parameter/initial
# state sweep is performed, for example when parameter estimation (shooting)
# or optimization is performed).
[/file]
[/template]
```

ANNEXE 8 : Code Acceleo de la transformation de DEVS vers HTML

```
[comment charset = UTF-8 /]
[module DEVS2HTMLDOC('extendeddevs')]
[template public gerFirst (aDEVSModel : DEVSModel) ? (aDEVSModel.ancestors()->isEmpty()=true)]
[comment @main /]
[file (aDEVSModel.name.concat('_Doc.html'), false, 'ISO-8859-1')]
<html>
Ce fichier de documentation se nomme <b>[aDEVSModel.name.concat('_Doc.html')/]</b>
et il a été généré automatiquement
<br><br>
[if aDEVSModel.oclIsTypeOf(AtomicDEVS)]
Ce modèle DEVS est atomique : il ne contient pas de sous-modèles
[else]Ce modèle DEVS est couplé, il s'appelle [aDEVSModel.name/]
et voici son arborescence complète
<br><br>
Modèle racine : [aDEVSModel.toString()/]<br>
[parcours(aDEVSModel)/]
[/if]
</html>
[/file]
[/template]

[template public parcours(aDEVSModel : DEVSModel)]
[/template]

[template public parcours(aDEVSCoupled : CoupledDEVS)]
<br>
[aDEVSCoupled.toString()/] contient <b>[aDEVSCoupled.contains->size()/]</b> sous-modèles :
<br>
[aDEVSCoupled.contains->sortedBy(oclIsTypeOf(CoupledDEVS)).toString().concat(' ')/]
<br>
[aDEVSCoupled.contains.parcours()/]
[/template]


[template public toString(aDEVSModel : DEVSModel)]
[/template]

[template public toString(aDEVSModel : CoupledDEVS)]
<b><i>[aDEVSModel.name/]</i></b>
[/template]

[template public toString(aDEVSModel : AtomicDEVS)]
<i>[aDEVSModel.name/]</i>
[/template]
```

ANNEXE 9 : Modèle couplé générateur + FSM (code Python généré)

```
# -*- coding: iso-8859-15 -*-
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
# coupledGeneratorandFSMAutoGen.py --- Based on the template for atomic- and coupled-DEVS descriptive
classes by
#                      --------------------------------
#                           Jean-Sébastien  BOLDUC
#                                                                  Hans Vangheluwe
#                        McGill University (Montréal)
#                      --------------------------------
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##

# Add the directory where pydevs lives to Python's import path
import sys
import os.path
sys.path.append(os.path.expanduser('~/src/projects/PythonDEVS/'))

# Import code for DEVS model representation:
import pydevs
from pydevs.devs_exceptions import *
from pydevs.infinity import *
from pydevs.DEVS import *

# Import for uniform random number generators
from random import uniform
from random import randint

#    ====================================================================    #

class wordGeneratorEncapState:

    """Encapsulate the Atomic DEVS' state
    This is not absolutely necessary, but it makes
    it easier to
    (1) stick to the DEVS formalism
    (and not modify the state "behind the back of the simulator" --
    this will be enforced once we have a DEVS compiler), and
    (2) to produce a clean simulation trace (thanks to __str__).
    """

    ###

    def __init__(self, StateSet):
        """Constructor (parameterizable).
        """

        # set the initial state
        # the state may be multi-dimensional:
        # consist of multiple state variables
        # (make a local copy of the constructor's arguments)

        self.StateSet=StateSet

    def get(self):
        return {'StateSet': self.StateSet}

    def __str__(self):
        return " StateSet = "+ str(self.StateSet)

class wordGenerator(AtomicDEVS):

    ###

    def __init__(self, name=None):
        """Constructor (parameterizable).
        Giving a name (string) to a model is not mandatory
        (a globally unique name "A<i>" is assigned by default).
        If a name is given, the simulation trace becomes far more
        readable.
        """
```

```
    # Always call parent class' constructor FIRST:
    AtomicDEVS.__init__(self, name)

    # TOTAL STATE:
    #  Define 'state' attribute (initial sate):
    self.state = wordGeneratorEncapState("1")

    # ELAPSED TIME:
    #  Initialize 'elapsed time' attribute if required.
    #  (by default (if not set by user), self.elapsed is initialized to 0.0)
    self.elapsed = 0
    #  With elapsed time for example initially 1.1 and
    #  this SampleADEVS initially in
    #  a state which has a time advance of 60,
    #  there are 60-1.1 = 58.9 time-units remaining until the first
    #  internal transition

    # HV: check in simulator that elapsed can not be larger than
    # timeAdvance

    # PORTS:
    #  Declare as many input and output ports as desired
    #  (usually store returned references in local variables):
    self.WORDGENERATORPORT = self.addOutPort(name="WORDGENERATORPORT")


def extTransition(self):
    """External Transition Function."""

    # Compute and return the new state based (typically) on current
    # State, Elapsed time parameters and calls to 'self.peek(self.IN)'.
    # When extTransition gets invoked, input has arrived
    # on at least one of the input ports. Thus, peek() will
    # return a non-None value for at least one input port.


def intTransition(self):
    """Internal Transition Function.
    """

    # Compute and return the new state  based (typically) on current State.
    # Remember that intTransition gets called after the outputFnc,
    # timeAdvance time after the current state was entered.

    statedico = self.state.get()
    state = self.state

    if statedico['StateSet'] == "1":
        state.StateSet="2"
        return state
    elif statedico['StateSet'] == "2":
        state.StateSet="3"
        return state
    elif statedico['StateSet'] == "3":
        state.StateSet="4"
        return state
    elif statedico['StateSet'] == "4":
        state.StateSet="5"
        return state
    elif statedico['StateSet'] == "5":
        state.StateSet="final"
        return state
    else :
        raise DEVSException(\
        "unknown state <%s> in wordGenerator internal transition function"\
    % state)

def outputFnc(self):
```

```
        """Output Function.
        """
        # Send events via a subset of the atomic-DEVS'
        # output ports by means of the 'poke' method.
        # More than one port may be poke-d.
        # The content of the messages is based (typically) on the current State.
        #
        # BEWARE: ouput is based on the CURRENT state
        # and is produced BEFORE making the internal transition.
        # Often, we want output based on the NEW state (which
        # the system will transition to by means of the internal
        # transition. The solution is to keep the NEW state in mind
        # when poke-ing the output.
        # This will only work if the internal transition function
        # is deterministic !


        statedico = self.state.get()
        state = self.state

        if statedico['StateSet'] == "1":
            self.poke(self.WORDGENERATORPORT,"m")
        elif statedico['StateSet'] == "2":
            self.poke(self.WORDGENERATORPORT,"i")
        elif statedico['StateSet'] == "3":
            self.poke(self.WORDGENERATORPORT,"u")
        elif statedico['StateSet'] == "4":
            self.poke(self.WORDGENERATORPORT,"i")
        elif statedico['StateSet'] == "5":
            self.poke(self.WORDGENERATORPORT,"m")

    def timeAdvance(self):
        """Time-Advance Function.
        """
            # Compute 'ta', the time to the next scheduled internal transition,
        # based (typically) on the current State.

        statedico = self.state.get()
        state = self.state

        if statedico['StateSet'] == "final":
            return INFINITY

        elif statedico['StateSet'] == "1":
            return 1

        elif statedico['StateSet'] == "2":
            return 1

        elif statedico['StateSet'] == "3":
            return 1

        elif statedico['StateSet'] == "4":
            return 1

        elif statedico['StateSet'] == "5":
            return 1

        else :
            raise DEVSException(\
            "unknown state <%s> in wordGenerator time advance function"\
            % state)


class enigmeMIUEncapState:

    """Encapsulate the Atomic DEVS' state
    This is not absolutely necessary, but it makes
```

```
    it easier to
    (1) stick to the DEVS formalism
    (and not modify the state "behind the back of the simulator" --
    this will be enforced once we have a DEVS compiler), and
    (2) to produce a clean simulation trace (thanks to __str__).
    """

    ###

    def __init__(self, FSMState):
        """Constructor (parameterizable).
        """

        # set the initial state
        # the state may be multi-dimensional:
        # consist of multiple state variables
        # (make a local copy of the constructor's arguments)

        self.FSMState=FSMState

    def get(self):
        return {'FSMState': self.FSMState}

    def __str__(self):
        return " FSMState = "+ str(self.FSMState)

class enigmeMIU(AtomicDEVS):

    ###

    def __init__(self, name=None):
        """Constructor (parameterizable).
        Giving a name (string) to a model is not mandatory
        (a globally unique name "A<i>" is assigned by default).
        If a name is given, the simulation trace becomes far more
        readable.
        """

        # Always call parent class' constructor FIRST:
        AtomicDEVS.__init__(self, name)

        # TOTAL STATE:
        #  Define 'state' attribute (initial sate):
        self.state = enigmeMIUEncapState("1")

        # ELAPSED TIME:
        #  Initialize 'elapsed time' attribute if required.
        #  (by default (if not set by user), self.elapsed is initialized to 0.0)
        self.elapsed = 0
        #  With elapsed time for example initially 1.1 and
        #  this SampleADEVS initially in
        #  a state which has a time advance of 60,
        #  there are 60-1.1 = 58.9 time-units remaining until the first
        #  internal transition

        # HV: check in simulator that elapsed can not be larger than
        # timeAdvance

        # PORTS:
        #  Declare as many input and output ports as desired
        #  (usually store returned references in local variables):
        self.FSMREADINPUTPORT = self.addInPort(name="FSMREADINPUTPORT")
        self.FSMREADOUTPUTPORT = self.addOutPort(name="FSMREADOUTPUTPORT")


    def extTransition(self):
        """External Transition Function."""

        # Compute and return the new state based (typically) on current
```

```
    # State, Elapsed time parameters and calls to 'self.peek(self.IN)'.
    # When extTransition gets invoked, input has arrived
    # on at least one of the input ports. Thus, peek() will
    # return a non-None value for at least one input port.


    ## j'ai compte 1 ports d'entree

    inputFSMREADINPUTPORT = self.peek(self.FSMREADINPUTPORT)

    statedico = self.state.get()
    state = self.state

    if inputFSMREADINPUTPORT == "m":
        if statedico['FSMState'] == "1":
            state.FSMState="2"
            return state
    if inputFSMREADINPUTPORT == "i":
        if statedico['FSMState'] == "2":
            state.FSMState="3"
            return state
    if inputFSMREADINPUTPORT == "u":
        if statedico['FSMState'] == "3":
            state.FSMState="2"
            return state
    if inputFSMREADINPUTPORT == "m":
        if statedico['FSMState'] == "3":
            state.FSMState="3"
            return state
    else :
        raise DEVSException(\
    "unknown state <%s> in enigmeMIU external transition function"\
        % state)


def intTransition(self):
    """Internal Transition Function.
    """

    # Compute and return the new state  based (typically) on current State.
    # Remember that intTransition gets called after the outputFnc,
    # timeAdvance time after the current state was entered.

    statedico = self.state.get()
    state = self.state

    if statedico['FSMState'] == "2":
        state.FSMState="3"
        return state
    elif statedico['FSMState'] == "3":
        state.FSMState="3"
        return state
    else :
        raise DEVSException(\
        "unknown state <%s> in enigmeMIU internal transition function"\
    % state)

def outputFnc(self):
    """Output Function.
    """
    # Send events via a subset of the atomic-DEVS'
    # output ports by means of the 'poke' method.
    # More than one port may be poke-d.
    # The content of the messages is based (typically) on the current State.
    #
    # BEWARE: ouput is based on the CURRENT state
    # and is produced BEFORE making the internal transition.
    # Often, we want output based on the NEW state (which
    # the system will transition to by means of the internal
```

```
        # transition. The solution is to keep the NEW state in mind
        # when poke-ing the output.
        # This will only work if the internal transition function
        # is deterministic !


        statedico = self.state.get()
        state = self.state

        if statedico['FSMState'] == "2":
            self.poke(self.FSMREADOUTPUTPORT,"__the word has been recognized__")
        elif statedico['FSMState'] == "3":
            self.poke(self.FSMREADOUTPUTPORT,"__the word has been recognized__")

    def timeAdvance(self):
        """Time-Advance Function.
        """
            # Compute 'ta', the time to the next scheduled internal transition,
        # based (typically) on the current State.


        statedico = self.state.get()
        state = self.state

        if statedico['FSMState'] == "1":
            return INFINITY

        elif statedico['FSMState'] == "2":
            return INFINITY

        elif statedico['FSMState'] == "3":
            return 4000

        else :
            raise DEVSException(\
            "unknown state <%s> in enigmeMIU time advance function"\
            % state)


#    ======================================================================    #
class coupledGeneratorandFSM(CoupledDEVS):
    """Sample Coupled-DEVS descriptive class: compulsory material.
    """
###
    def __init__(self, name=None):
        """Constructor (parameterizable).
        Giving a name (string) to a model is not mandatory
        (a globally unique name "C<i>" is assigned by default).
        If a name is given, the simulation trace becomes far more
        readable.
        """

        # Always call parent class' constructor FIRST:
        CoupledDEVS.__init__(self, name)

        # PORTS:
    #  Declare as many input and output ports as desired²
    #  (usually store returned references in local variables).
    #  Note that giving a name (string) to a port is not mandatory
    #  (a globally unique name "port<i>" is assigned by default).
    #  If a name is given, the simulation trace becomes far more
    #  readable however.


        #  Note: at the root level, the Coupled DEVS will often
    #  not have any InPorts nor OutPorts (i.e., be "autonomous").


        # SUB-MODELS:
```

ANNEXE 9 : Modèle couplé générateur + FSM (code Python généré)

```
    #  Declare as many sub-models as desired. Method 'addSubModel' takes as a
    #  parameter an INSTANCE of an atomic- or coupled-DEVS descriptive class,
    #  and returns a reference to that instance (usually kept in a local
    #  variable for future reference):

        self.wordgenerator = self.addSubModel(wordGenerator(name="wordgenerator"))
        self.enigmemiu = self.addSubModel(enigmeMIU(name="enigmemiu"))

    # COUPLINGS:
    #  Method 'connectPorts' takes two parameters 'p1' and 'p2' which are
    #  references to ports.
    #  The coupling is to BEGIN at 'p1' and to END at 'p2'.
    #  Each input/output port can have
    #  multiple incoming/outgoing connections.
    #  Note how PythonDEVS does NOT support Z_i,j transfer
    #  functions (as defined in DEVS) yet.
    #  On the other hand, PythonDEVS does have named ports.
        self.connectPorts(self.wordgenerator.WORDGENERATORPORT, self.enigmemiu.FSMREADINPUTPORT) #
INTERNAL COUPLING

    def select(self, immList):
        """Select function.
        Parameter 'immList' is a list of references to sub-models which
        have a scheduled internal transition occuring at the same time.
        To implement tie-breaking, select() needs to return
        exactly ONE submodel immList(i).
        """

###
```