SimStudio : une Infrastructure pour la Modélisation, la Simulation et l'Analyse de Systèmes Dynamiques Complexes

Luc Touraille[1], Mamadou K. Traoré[2], David R.C. Hill[3]

Research Report LIMOS/RR-10-13

19 mai 2010

[1] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, touraille@isima.fr
[2] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, traore@isima.fr
[3] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, drch@isima.fr

Abstract

SimStudio is a Modeling & Simulation framework based on the DEVS formalism (Discrete EVent Systems Specification). SimStudio architecture aims at integrating in a single platform tools for modeling, simulation, analysis and collaboration, by proposing model transformation features (code generation, among others) in order to smooth the modeling and simulation cycle. To achieve this, SimStudio is built upon the DEVS formalism, recognize by many as a "universal" simulation formalism, and upon a unifying language for representing DEVS models. Models are at the heart of the infrastructure we propose, in line with the Model-Driven Engineering (MDE) approach, at the boundary between software engineering and simulation.

Keywords: Modeling & Simulation, interoperability, framework, DEVS, MDE

Résumé

SimStudio est un cadriciel de Modélisation & Simulation basé sur le formalisme DEVS (Discrete EVent Systems Specification). L'architecture de SimStudio vise à intégrer dans une même plate-forme des outils de modélisation, de simulation, d'analyse et de collaboration, en proposant des fonctionnalités de transformations de modèles (entre autres de la génération de code) afin de simplifier le cycle de modélisation et de simulation. Pour ce faire, SimStudio se base sur le formalisme DEVS qui est reconnu par beaucoup comme un formalisme de simulation « universel », et sur un langage unificateur de représentation des modèles DEVS. Les modèles sont au cœur de l'infrastructure que nous proposons, qui se positionne donc dans le domaine de l'Ingénierie Dirigée par les Modèles (IDM), à la frontière entre le génie logiciel et la simulation.

Mots-clés : Modélisation & Simulation, interopérabilité, cadriciel, DEVS, IDM

1. Introduction

Les scientifiques français s'accordent sur un certain nombre de difficultés que rencontre le domaine de la Modélisation et de la Simulation (M&S) [CGTI 2005] [GSAT 2005]. Celles-ci portent sur plusieurs points :

– Le manque de collaborations interdisciplinaires. En effet, les équipes de modélisation et les informaticiens paraissent souvent travailler en parallèle et non en collaboration, ce qui mène d'un côté à des logiciels peu aboutis et rarement robustes, et de l'autre à des logiciels coupés de la réalité du métier, ou trop peu diffusés.

– La simulation de modèles de plus en plus compliqués, soit parce qu'ils approchent de plus en plus la complexité du réel (multi-échelles, niveau de granularité plus fin), soit parce qu'ils intègrent plus d'aspects que par le passé.

– Le manque en grands systèmes de M&S pérennes. Il existe à ce jour peu de suites logicielles complètes, stables et reconnues pour la réalisation d'un projet de M&S.

Le projet SimStudio [Traore 2008] a pour objet la conception d'un atelier logiciel qui réponde en partie à ces problématiques. Cette plate-forme autorise la conception de modèles de façon collaborative via des interfaces web, en ayant accès à des bibliothèques de modèles développés par la communauté. Grâce à des transformations de modèles, et notamment à des générations de code, la création de solutions opérationnelles est facilitée. SimStudio exploite le potentiel de l'ingénierie des modèles et du formalisme DEVS (Discrete EVent System Specification), présenté en section 2. Plusieurs travaux ont déjà été réalisés dans cette optique, et sont évoqués en section 3. Outre les composants de modélisation et de simulation, l'architecture SimStudio est également conçue pour inclure des fonctionnalités d'analyse et de visualisation. Son architecture, exposée en section 4, est basée sur un système de plugins qui permettra l'ajout de modules et l'intégration d'outils existants, via des transformations de modèles. Les communications entre ces différents modules se feront grâce à un langage de représentation des modèles intitulé DML (DEVS Markup Language) [Touraille et al. 2009], dont nous donnons un aperçu en section 5.

2. Le formalisme DEVS

Basé sur la théorie des systèmes, le formalisme DEVS [Zeigler et al. 2000] fournit une base solide pour la modélisation et la simulation des systèmes à évènements discrets, mais également pour l'intégration de modèles hétérogènes. Il en existe de nombreuses variantes, mais seules deux sont fondamentales : Classic DEVS (C-DEVS) et Parallel DEVS (P-DEVS). Ces deux formalismes décrivent la même classe de systèmes ; nous concentrons nos travaux sur P-DEVS car il permet plus facilement de paralléliser les simulations que C-DEVS, qui est totalement séquentiel. DEVS a été utilisé depuis plusieurs années dans de nombreuses applications et nous l'avons appliqué à des domaines très différents [Kim et al. 2000] [Muzy et al. 2005] [Farooq et al. 2007] [Innocenti et al. 2009].

DEVS permet de représenter un système de deux façons [Zeigler et al. 2000]. La première consiste à concevoir un *modèle atomique*, qui décrit le système comme une entité possédant un état (S), un comportement autonome, via des changements d'état internes (δ_{int}) et un processus de gestion du temps (ta), une réactivité par rapport à l'environnement (δ_{ext}), et un mécanisme de génération d'évènements (λ). Les communications avec l'environnement se font via des ports d'entrées (X) et des ports de sorties (Y), qui reçoivent ou génèrent des évènements auxquels sont associés des valeurs. La deuxième façon de spécifier un système avec DEVS est de définir un *modèle couplé*. Celui-ci possède des ports d'entrées et de sorties, et agrège des composants, qui peuvent être soit des modèles atomiques, soit eux-mêmes des modèles couplés. Ces composants sont reliés par leurs ports de sortie et d'entrée (couplages) ; ainsi, les évènements générés par un modèle deviennent des stimuli pour d'autres.

La force de DEVS réside dans son pouvoir d'expression. En effet, il a été construit de façon incrémentale à partir de spécifications minimales, en se plaçant à chaque étape à un niveau d'abstraction apportant des connaissances supplémentaires, tout en démontrant les morphismes avec les niveaux inférieurs. On peut donc prouver [Zeigler et al. 2000] qu'il existe un modèle DEVS pour tout système à évènements discrets. Mais on peut aller plus loin ; en effet, le protocole DEVS peut être utilisé comme simulateur « universel », permettant le couplage de modèles décrits dans des formalismes et selon des paradigmes hétérogènes. Deux méthodes existent pour intégrer un modèle non-DEVS dans le cadre DEVS : la co-simulation et la transformation de modèles [Schmidt 2006]. Cette dernière consiste à transformer des modèles non-DEVS vers le formalisme DEVS, afin d'obtenir une spécification des modèles dans un langage uniforme. Vangheluwe représente les différentes

transformations possibles à l'aide d'un arbre de transformation de formalismes (*Formalism Transformation Graph*, FTG) [Vangheluwe 2000].

3. Travaux apparentés

Les simulateurs DEVS étant précisément définis par le formalisme, leur implémentation est assez simple. On peut par conséquent trouver de nombreuses bibliothèques de simulation DEVS. Parmi celles-ci, DEVSJava [DEVSJava 2004] propose plusieurs outils, par exemple une interface graphique de modélisation. Cependant, DEVSJava ne prend pas encore en compte le cycle complet de M&S, et reste spécifique à DEVS. Dans le même ordre d'idées, JDevs [Filippi et al. 2002] inclut un module graphique d'édition de modèles, ainsi qu'un simulateur DEVS et deux outils de visualisation 2D et 3D.

En ce qui concerne l'intégration de modèles hétérogènes, on peut distinguer deux approches :

– Une approche centrée sur les modèles, qui consiste à standardiser la représentation des modèles afin de pouvoir les échanger et les composer facilement, comme nous l'avons fait avec DML. Plusieurs travaux avaient déjà été entrepris dans cette direction, mais étaient soit trop restrictif pour profiter pleinement du pouvoir d'expression de DEVS [Risco-Martín et al. 2007] [Janoušek et al. 2006], soit spécifique à un langage de programmation donné [Mittal et al. 2007a].

– Une approche centrée sur les simulateurs (co-simulation), dans laquelle ce ne sont pas les modèles qui sont standardisés mais les communications inter-simulateurs. Ce domaine a été exploré par le Department of Defense américain, qui a proposé HLA (High Level Architecture) [HLA 2000] comme solution au problème d'interopérabilité. Cette solution semble très utilisée pour les simulations militaires qui font intervenir des simulateurs hétérogènes et distribués géographiquement, mais nécessite un processus de mise en œuvre relativement lourd. Une approche plus souple, utilisant une architecture orientée service et plus précisément des services web, peut être trouvée dans [Mittal et al. 2007b].

Le projet Virtual Laboratory Environment (VLE) [Quesnel et al. 2009] s'inscrit lui aussi dans cette optique de multi-modélisation, en permettant à l'utilisateur de spécifier ses modèles selon différentes variantes de DEVS (Cell-DEVS, DS-DEVS…) et de les coupler via le principe de co-simulation. Il comporte des modules de conception et de visualisation graphiques. Cependant, les aspects collaboration et accès aux ressources de calcul intensif ne sont pas pris en compte, et la plate-forme reste spécifique DEVS.

James II [Himmelspach 2007] est un cadriciel de M&S générique qui accueille des plugins définissant des formalismes et des simulateurs, sous forme de classes Java. Cependant, à notre connaissance, cette plate-forme ne permet pas de coupler des modèles définis dans des formalismes hétérogènes. D'autre part, les plugins sont très fortement couplés à l'architecture, limitant l'intégration aisée d'outils existants.

Enfin, dans [Posse et al. 2003], les auteurs définissent un métamodèle DEVS à l'aide de l'outil AToM^3 [De Lara et al. 2002], une plate-forme de métamodélisation et de transformation de modèles. Ils génèrent alors un environnement graphique de modélisation DEVS, et spécifient également les transformations nécessaires pour générer du code Python utilisable dans le simulateur PythonDEVS.

4. Architecture SimStudio

Basé sur le formalisme DEVS, le cadriciel SimStudio vise à proposer un environnement web complet de Modélisation & Simulation, qui assiste le modélisateur depuis la conception de son modèle jusqu'à l'analyse des résultats. Afin de pouvoir incorporer les outils existants et ajouter aisément de nouvelles fonctionnalités, SimStudio utilise une architecture logicielle modulaire à base de plugins. La plate-forme est constituée d'un noyau de simulation, sur lequel viennent se greffer des composants logiciels qui se déclinent selon plusieurs axes (**Erreur ! Source du renvoi introuvable.**) :

– Un axe Modélisation, qui regroupe les outils graphiques et textuels pour la conception de modèles. On peut également avoir des modules « adaptateurs », qui sont chargés de transformer une spécification de modèle fourni par un outil existant en un format compréhensible par SimStudio.

– Un axe Analyse, dans lequel on peut trouver des outils d'analyse formelle, d'analyse statistique, etc.

– Un axe Visualisation, gérant l'affichage des résultats de simulation sous forme de graphiques, d'animations, mais pouvant également fournir des fonctionnalités de réalité virtuelle, en laissant l'utilisateur agir sur le déroulement de la simulation.

– Un axe de Gestion, qui fournit les services transversaux tels que la gestion des utilisateurs, le stockage et la recherche de modèles, l'installation de nouveaux plugins, etc.

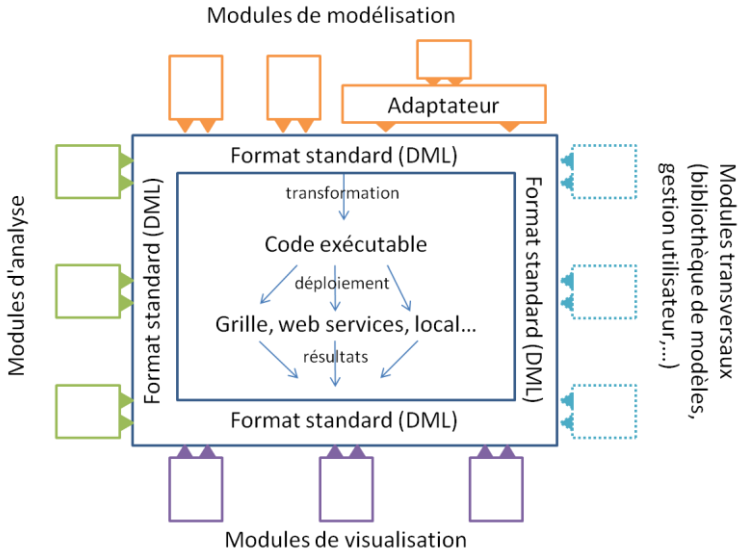

Figure 1. *Vue d'ensemble de l'architecture SimStudio*

Le rôle du noyau de simulation est de générer automatiquement une solution opérationnelle à partir de la spécification d'un modèle, puis de gérer son exécution sur différents types de plates-formes, selon le choix de l'utilisateur. Cela peut aller d'une simple exécution sur un serveur ou en local à une distribution sur grille de calcul ou sur cluster.

Cependant, pour que tous ces modules puissent communiquer entre eux, SimStudio doit se baser sur un langage commun ; nous pensons que le formalisme DEVS est l'outil le plus adapté pour remplir ce rôle, en raison de sa puissance d'expression. En effet, l'utilisation de DEVS comme dénominateur commun apporte deux avantages très importants :

– La possibilité de coupler des modèles hétérogènes, qui peuvent utiliser des paradigmes différents (continu/discret), des formalismes différents (équations différentielles, réseaux de Petri, automates cellulaires,...), des niveaux d'abstraction différents, etc.

– L'accès à une sémantique opérationnelle parfaitement définie. DEVS sépare strictement modèles et simulateurs. Par conséquent, un simulateur DEVS est « universel », dans le sens où il peut simuler tout modèle DEVS. En mettant à la disposition des modélisateurs les transformations nécessaires pour convertir un modèle en modèle DEVS, et l'accès à un simulateur performant et configurable, on évite les efforts d'implémentation sans cesse renouvelés : l'utilisateur peut se concentrer sur la modélisation, sans se soucier de l'aspect simulation.

Un pré-requis indispensable est par conséquent la définition d'un métamodèle DEVS qui soit assez générique pour supporter les modèles existants et assez flexible pour être utilisable en pratique. Ce dernier permettra une approche d'Ingénierie Dirigée par les Modèles, dans laquelle la définition de nouveaux plugins (à l'exception des modules de gestion) sera réduite à l'écriture de transformations depuis (resp. vers) un modèle DEVS, conforme au métamodèle, vers (resp. depuis) d'autres modèles, représentant autant de vues sur le même système étudié.

Dans la section suivante, nous présentons le métamodèle DEVS que nous avons développé, intitulé DML (DEVS Markup Language).

5. DML, un métamodèle DEVS unificateur

L'architecture SimStudio est basée sur l'utilisation d'une représentation standardisée de tous les aspects des modèles DEVS, définie par un métamodèle. Ce dernier doit prendre en compte les différents cas d'utilisation possibles, et doit donc spécifier les éléments suivants :

– La structure des modèles DEVS, c'est-à-dire les variables d'états, les ports d'entrée et de sorties et les couplages.

– La dynamique des modèles atomiques, décrite par les différentes fonctions de transitions (interne, externe et confluente).

– Le paramétrage et l'initialisation des modèles.

– Le comportement des modèles, correspondant à des résultats de simulation, sous forme de trajectoires.

– Éventuellement des attributs graphiques, pour la visualisation des modèles et des résultats.

D'autre part, pour remplir sa fonction unificatrice, ce métamodèle doit définir une famille de modèles indépendants de toute plate-forme [Bézivin et al. 2008], rejoignant le principe de Platform-Independent Model (PIM) du MDA (Model Driven Architecture de l'Object Management Group). En effet, les modélisateurs doivent disposer d'un « langage commun » pour s'échanger et faire interagir leurs modèles, qui peuvent avoir été développés pour des simulateurs et dans des langages de programmation différents. La principale difficulté réside donc dans la représentation de la dynamique des modèles : comment décrire un algorithme indépendamment de tout langage de programmation, sans pour autant priver le modélisateur de toutes les bibliothèques et fonctionnalités propres à ces langages ?

Avant de répondre à cette question, nous devons choisir le métamétamodèle auquel se conformera DML. Pour des raisons évidentes de standardisation et d'interopérabilité, nous effectuons les échanges entre les composants de SimStudio via des documents XML (eXtensible Markup Language) [Abiteboul et al. 1999]. Dans un but de prototypage rapide, nous avons donc spécifié notre métamodèle DEVS à l'aide de XML Schema, en raison de sa facilité de mise en œuvre et des nombreux outils existants, notamment pour la réalisation de transformations, entre autres grâce au langage Extensible Stylesheet Language Transformation (XSLT). Dans un second temps, il pourrait être intéressant d'utiliser un standard de métamodélisation plus « générique », par exemple le MetaObject Facility (MOF) [Bézivin et al. 2008] en association avec le standard XML Metadata Interchange (XMI).

Dans les sous-sections suivantes, nous présentons plus précisément DML selon les différents axes évoqués précédemment.

5.1. *Structure des modèles*

La structure des modèles DEVS repose sur la notion d'ensemble ; cette notion est facilement capturée en XML par le concept de type, largement utilisé en programmation. Les ports d'entrée et de sortie, les variables d'état et les paramètres sont donc représentés par des instances de types, qui peuvent être :

– intrinsèques, c'est-à-dire supportés par défaut, comme les entiers, les flottants ou les chaînes de caractères ;

– définis par l'utilisateur, à l'aide de schémas XML. Ce dernier peut ainsi créer ses propres types afin de les utiliser dans ses modèles. Les schémas XML peuvent aussi être générés à partir de classes ou structures, dans le cas où le modèle fait l'objet de rétro-ingénierie depuis un langage de programmation ;

– dépendants des bibliothèques. En effet, dans des applications réelles, le modélisateur peut avoir besoin d'utiliser certains types définis dans des bibliothèques d'un langage donné. Pour trouver un compromis entre l'indépendance vis-à-vis de la plate-forme et la liberté nécessaire au modélisateur, nous proposons une notion proche de celle de *type abstrait* [Liskov et al. 1974]. Avant de pouvoir être utilisés, ces types doivent être liés à une implémentation spécifique dans le langage de programmation cible. Par exemple, un type abstrait *RandomNumberGenerator* pourrait être implémenté en Java par la classe `umontreal.iro.lecuyer.rng.MT199937` [L'Ecuyer et al. 2002] et en C++ par `boost::mt19937`.

Le Code 1 montre un exemple de description des variables d'état d'un modèle atomique. Ce modèle possède deux variables : *sigma*, un nombre réel, et *buffer*, une file d'attente, instance du type abstrait *Queue* ; afin de pouvoir générer le code correspondant dans les langages cibles, il est nécessaire de spécifier les liaisons entre types abstraits et types concrets. Pour cela, nous définissons des modèles de bibliothèques, qui indiquent les types à utiliser pour remplir le rôle des types abstraits. Les Codes 2 et 3 donnent la correspondance du type *Queue* dans la bibliothèque standard du Java et dans celle du C++, la Standard Template Library (STL).

```
<state>
  <variable>
    <name>buffer</name>
    <type kind="libDependent">Queue</type>
  </variable>
  <variable>
    <name>sigma</name>
    <type>double</type>
  </variable>
</state>
```

Code 1. *Description des variables d'état*

```
<type id="Queue">
  java.util.LinkedList<String>
</type>
```

Code 2. *Extrait du modèle « Standard Java Library »*

```
<type id="Queue">
  std::queue<std::string>
</type>
```

Code 3. *Extrait du modèle STL*

Ces modèles de bibliothèques seront utilisés lors des transformations de modèles DEVS vers une solution opérationnelle, permettant ainsi de générer automatiquement un modèle spécifique dans le langage cible (rejoignant la notion de Platform Specific Model du MDA). Ainsi, les instances du type *Queue* seront en Java des listes chaînées de chaînes de caractères, et en C++ des files de chaînes de caractères. Les différentes transformations nécessaires sont détaillées dans le paragraphe 5.4.

5.2. *Représentation des algorithmes*

L'approche utilisée pour représenter la dynamique des modèles se base sur des principes similaires. En effet, nous avons besoin de représenter la logique des fonctions du modèle de façon indépendante de la plate-forme, tout en laissant le modélisateur libre d'utiliser les fonctionnalités propres à un langage ou à une bibliothèque. Par conséquent, nous utilisons pour décrire ces fonctions un mélange de code générique, qui possède une correspondance directe dans les langages impératifs existants, et de fragments de code abstraits, qui doivent être implémentés pour chaque langage cible (cf. Code 4).

Dans cet exemple, on teste la valeur de la variable *sigma*, puis, si elle est égale à zéro, on ajoute un élément dans le buffer. L'opération d'ajout dans la file d'attente ne peut être décrite de façon générique car elle dépend de la bibliothèque utilisée. Par exemple, en Java, l'ajout se fera avec la méthode *offer* de la classe *java.util.LinkedList*, tandis qu'en C++, avec la STL, il correspondra à l'appel de la méthode *push* de *std::queue*.

D'autre part; certaines portions de code ne dépendent pas uniquement du langage utilisé mais également du simulateur cible. Par exemple, récupérer une valeur sur un port d'entrée ne se fait pas de la même façon selon les implémentations. Il est donc nécessaire de pouvoir spécifier des fragments de code qui soient dépendants du simulateur, comme *getValueOnPort* dans notre exemple. Ainsi, nous utilisons trois niveaux d'indépendances : vis-à-vis du langage de programmation, des bibliothèques et des simulateurs. Comme précédemment, ces indépendances devront être « résolues » à l'aide de modèles de plates-formes et de transformations.

```
<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>
```

Code 4. *Représentation de code « semi-générique » en XML*

Grâce à cette solution, DML laisse une grande liberté dans l'écriture de modèles. Il permet de représenter la plupart des modèles DEVS déjà développés dans les cadriciels existants, et minimise la quantité de code à écrire pour porter un modèle d'une plate-forme à une autre. Seuls les types et les fragments de code abstraits nécessitent d'être concrétisés en une implémentation spécifique. Cette étape peut même être simplifiée en construisant collectivement des « catalogues » de types et de fragments accompagnés de leurs implémentations dans les langages les plus populaires. De même, les fournisseurs de cadriciels DEVS peuvent publier le modèle de leur plate-forme pour permettre l'intégration de leur outil à SimStudio.

Outre les fonctions propres aux modèles DEVS, ce type de code « semi-générique » est également utilisé dans DML pour décrire les phases de paramétrage et d'initialisation. En effet, ces deux étapes se résument parfois à de simples assignations de valeurs, mais peuvent également être plus compliquées (génération de nombres aléatoires, lecture de fichiers, etc.).

5.3. *Résultats de simulation*

Le comportement d'un modèles DEVS est entièrement représenté par l'ensemble de toutes ses trajectoires d'entrée/sortie possibles, c'est-à-dire tous les couples (trajectoire d'entrée, trajectoire de sortie) que le modèle peut générer, une trajectoire d'entrée (respectivement de sortie) étant l'ensemble des évènements générés sur les ports d'entrée (respectivement de sortie) lors d'une session de simulation. Cependant, il est généralement utile au modélisateur de connaître l'évolution des différentes variables d'état, ainsi que le comportement des sous-modèles dans le cas d'un modèle couplé. Ces résultats de simulation sont représentés en DML par des ensembles de valeurs, annotées par un temps de simulation. Nous pouvons alors utiliser ces trajectoires pour analyser le comportement du modèle, représenter graphiquement la simulation, ou encore les donner en entrée à un autre modèle.

5.4. *Vue d'ensemble des relations de conformité et transformations*

Dans les paragraphes précédents, nous avons plusieurs fois évoqué les transformations nécessaires pour obtenir une solution opérationnelle à partir d'un modèle DML. La **Erreur ! Source du renvoi introuvable.** présente ces différentes transformations, ainsi que les relations de conformité entre les modèles, métamodèles et métamétamodèles utilisés dans SimStudio. Dans cet exemple, nous transformons un modèle DML quelconque en

code compréhensible par DEVS-MetaSimulator (DEVS-MS), le simulateur que nous avons développé pour le noyau de SimStudio, écrit en C++.

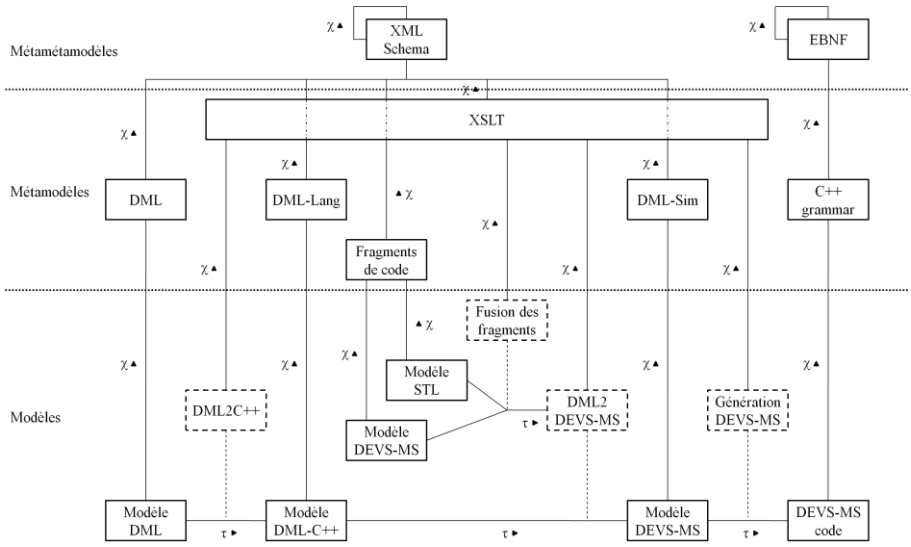

Figure 2. *Modèles, métamodèles, métamétamodèles et modèles de transformation mis en jeu dans le noyau de SimStudio*

Nous reprenons dans cette figure les notations utilisées dans [Favre et al. 2006] :

– χ signifie « est conforme à » ;
– τ signifie « est transformé en ».

D'autre part, nous dénotons les modèles de transformation par des rectangles pointillés. Ceux-ci sont reliés par une ligne pointillée aux relations « est transformé en » (τ) pour lesquelles ils sont mis en œuvre.

La meilleure façon d'analyser ce schéma consiste à le lire de bas en haut et de gauche à droite. Au cœur de l'architecture se situent les modèles DML ; ceux-ci sont conformes au métamodèle DML présenté précédemment, lui-même conforme au métamétamodèle XML Schema. L'obtention d'une solution opérationnelle s'effectue par raffinements successifs (transformations vers une plate-forme cible), jusqu'à générer le code source du modèle pour un simulateur donné.

Dans un premier temps, nous effectuons une transformation d'un modèle DML vers un modèle dit « DML-Lang », spécifique à un langage de programmation cible. Cette transformation, représentée par un modèle XSLT, consiste à remplacer les portions de code « génériques » par leurs équivalents dans le langage de destination. Sont concernés les éléments de type assignation, branchement conditionnel, bouclage, etc., ainsi que les définitions de types utilisateurs (classes).

Le métamodèle « DML-Lang » est très similaire à DML, à l'exception des éléments codes qui contiennent maintenant du contenu mixte (texte et balises XML) : la structure générique des fonctions a été générée, mais les fragments dépendants des bibliothèques et des simulateurs n'ont pas été transformés. Cela fait l'objet d'une seconde transformation.

Cette dernière fait intervenir deux types de modèles de plates-formes : les modèles de bibliothèque, et les modèles de simulateur, tous étant conforme au métamodèle « Fragments de code » et spécifiant les correspondances entre les opérations abstraites utilisées dans le modèle DML et les opérations concrètes à effectuer. Les développeurs de cadriciels DEVS doivent avoir la possibilité de redéfinir des correspondances exprimées dans le modèle de bibliothèque, par exemple pour utiliser certaines structures de données s'intégrant dans leur hiérarchie d'héritage, ou encore s'ils souhaitent éviter l'utilisation de certains modules de la bibliothèque ; c'est la raison pour laquelle nous gérons les deux types d'opérations de façon conjointe, grâce à l'utilisation d'une transformation d'ordre supérieur (« Fusion des fragments ») qui, à partir des deux modèles de plate-forme, crée une nouvelle transformation (« DML2DEVS-MS »). Concrètement, cette transformation génère un modèle XSLT qui applique les remplacements définis par les deux modèles (bibliothèque et simulateur) en donnant la priorité à ceux spécifiés pour le simulateur. La transformation générée peut alors être appliquée au modèle DML-Lang afin d'obtenir un modèle DML-Sim. A l'issue de cette transformation, tous les

éléments du modèle DML représentant du code (notamment toutes les fonctions du modèle DEVS) ont été transformés vers le langage cible.

Reste enfin à générer le code complet permettant l'intégration du modèle au cadriciel cible. Généralement, cela passe par la génération d'une classe, de la définition des ports d'entrée et de sorties, des couplages, etc. Il faut d'autre part prendre en compte les contraintes inhérentes au cadriciel : dérivation d'une classe de base, attributs requis, etc. Les opérations à effectuer pouvant être très hétérogènes, nous laissons les concepteurs de simulateurs DEVS libres de définir leurs propres modèles de transformation, conformes à XSLT.

A terme, nous disposerons de différents dépôts de modèles, donnant accès à :

– des modèles DML, ayant vocation à être réutilisés soit en l'état, soit en tant que composants de modèles couplés ;

– des modèles de bibliothèques, offrant ainsi un large choix à l'utilisateur ;

– des modèles de simulateurs et les modèles de transformation associés, afin d'élargir le spectre d'application de DML au maximum d'outils existants.

6. Conclusion et travaux futurs

Dans cet article nous avons présenté l'architecture de SimStudio, un environnement de M&S qui agrège dans une seule plate-forme des outils de modélisation, d'analyse, de simulation et de visualisation. Accessible à travers un navigateur internet, il permettra de capitaliser les avancées théoriques du domaine, et accélérera le cycle de M&S : la conception de modèles sera facilité grâce aux modules de modélisation et à la mise à disposition de bibliothèques de modèles ; la création de code exécutable, parallélisable sur des ressources de calcul intensif, sera automatisée au maximum, grâce à des transformations de modèle et à de la génération de code ; et enfin, l'analyse des modèles et des résultats de simulation sera effectuées dans la plate-forme même, via des plugins d'analyse.

Doté d'une architecture modulaire, SimStudio permet l'ajout de nouvelles fonctionnalités et l'intégration d'outils existants grâce à une approche d'ingénierie dirigée par les modèles et plus précisément à l'utilisation d'un métamodèle DEVS intitulé DEVS Markup Language.

La suite de nos travaux portera sur plusieurs points. Tout d'abord, à partir des travaux présentés nous envisageons la retro-ingénierie et la refonte de modèles que nous avons déjà réalisés avec DEVS. De plus, nous souhaitons définir le métamodèle DEVS non plus sous forme de schéma XML, mais avec un environnement de métamodélisation plus complet, en utilisant par exemple le MOF. Cela permettrait notamment de spécifier plus précisément la sémantique associée au métamodèle, qui n'est ici qu'implicite. D'autre part, nous essaierons également de définir des *patrons de M&S* (solution générique à un problème récurrent auquel sont confrontés les modélisateurs). Nous pensons que l'identification de ces situations et l'intégration de solutions appropriées dans SimStudio permettra de réduire la complexité du développement de bon nombre de modèles et de simulations.

7. Bibliographie

[Abiteboul et al. 1999] Abiteboul S., Buneman P., Suciu D., *Data on the Web: From Relational to Semistructured Data and XML*, Morgan Kaufmann, 1999.

[Bézivin et al. 2008] Bézivin J., Vallecillo-Moreno A., García-Molina J., Rossi G., « MDA at the Age of Seven: Past, Present and Future », *European Journal for the Informatics Professional*, vol. IX, No. 2, April 2008, pp. 4-7.

[CGTI 2005] Groupe Conseil Général des Technologies de l'Information, « La politique française dans le domaine du calcul scientifique », Rapport n° II.B.14, Mars 2005.

[De Lara et al. 2002] De Lara J., Vangheluwe H., « AToM3: A Tool for Multi-formalism and Meta-modelling », *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, 6-14 Avril 2002, Grenoble, France, Lecture Notes In Computer Science, Vol. 2306, pp. 174-188.

[DEVSJava 2004] ACIMS software site: http://www.acims.arizona.edu/SOFTWARE/software.shtml Last accessed May 2009.

[Farooq et al. 2007] Farooq U., Wainer G., Balya B., « DEVS Modeling of Mobile Wireless Ad Hoc Networks », *Simulation Modelling Practice and Theory*, vol. 15, No. 3, pp. 285-314, 2007.

[Favre et al. 2006] Favre J., Estublier J., Blay-Fornarino M., *L'ingénierie dirigée par les modèles. Au delà du MDA*, Hermès – Lavoisier, ISBN 2746212137, 2006.

[Filippi et al. 2002] Filippi J.B., Delhom M., Bernardi F., « The JDEVS hybrid modelling and simulation environment », *Proceedings of the iEMSs First Biennial Meeting: International Congress on Environmental Modelling and Software* (iEMSs 2002), International Environmental Modelling and Software Society, Lugano, Switzerland, Vol. 3, pp. 283-288, 2002.

[GSAT 2005] Groupe Simulation – Académie des Technologies, « Enquête sur les frontières de la simulation numérique en France. La situation en France et dans le monde. Diagnostic et propositions », Rapport de l'Académie des Technologies, Mai 2005.

[Himmelspach 2007] Himmelspach J., « Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations- und Experimentiersystems - Entwicklung und Evaluation effizienter Simulationsalgorithmen », Thèse de doctorat, Universität Rostock 2007.

[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.

[Innocenti et al. 2009] Innocenti E, Silvani X., Muzy A., Hill D., « A software framework for fine grain parallelization of cellular models with OpenMP: Application to fire spread », *Environmental Modelling & Software*, Vol 24, 2009, pp. 819-831.

[Janoušek et al. 2006] Janoušek V., Polášek P., Slavíček P., « Towards DEVS Meta Language », *ISC 2006 Proceedings*, June 7, 2006, Palermo, Italy, pp. 69-73.

[Kim et al. 1998] Kim J.Y., Kim T.G. « A Heterogeneous Simulation Framework Based on DEVS Bus and High Level Architecture », *Proceedings of the 1998 Winter Simulation Conference*, December 13-16, 1998, Washington, DC, USA, vol. 1, pp. 13-16.

[Kim et al. 2000] Kim D., Cao H, Buckley S.J., « Modeling and Simulation of Supply Chain Management Based on DEVS and CORBA Framework », *Proceedings of the 2000 AI, Simulation and Planning in High Autonomy Systems Conference*, March 6-8, 2000, Tucson, Arizona, USA, pp. 282-289.

[L'Ecuyer et al. 2002] L'Ecuyer P., Meliani L., Vaucher J., « SSJ: A Framework for Stochastic Simulation in Java », *Proceedings of the 2002 Winter Simulation Conference*, December 8-11, 2002, San Diego, California, USA, Vol. 1, pp. 234-242.

[Liskov et al. 1974] Liskov B., Zilles S., « Programming with abstract data types », *ACM SIGPLAN Notices*, Vol. 9, No 4, 1974, pp. 50-59.

[Mittal et al. 2007a] Mittal S., Risco-Martín J.-L., Zeigler B.P., « 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 et al. 2007b] Mittal S. Risco-Martín J.-L., Zeigler B.P., « DEVS-based simulation web services for net-centric T&E », *Proceedings of the 2007 Summer Computer Simulation Conference*, July 15-18, 2007, San Diego, CA, USA, pp. 357-366.

[Muzy et al. 2005] Muzy A., Innocenti E., Aïello A., Santucci J.F., Santoni P.A, Hill D., « Modelling and simulation of ecological propagation processes: application to fire spread », *Environmental Modelling & Software*, Vol. 20, Issue 7, July 2005, pp. 827-842.

[Posse et al. 2003] Posse E., Bolduc J.-S., « Generation of DEVS Modelling & Simulation Environments », *Proceedings of the 2003 SCS Summer Computer Simulation Conference*, July 2003, Montréal, Canada, pp. 295-300.

[Quesnel et al. 2009] Quesnel G., Duboz R., Ramat E., « The Virtual Laboratory Environment - An Operational Framework for Multi-Modelling, simulation and analysis of complex dynamical systems », *Simulation Modelling Practice and Theory*, vol. 17, April 2009, pp. 641-643.

[Risco-Martín et al. 2007] Risco-Martín J.-L., Mittal S., López-Peña M. A., De La Cruz J. M., « A W3C XML schema for DEVS scenarios », *Proceedings of the 2007 ACM Spring Simulation Multiconference*, March 25-29, 2007, Norfolk, VA, USA, Vol. 2, pp. 279-286.

[Schmidt 2006] Schmidt D.C., « Model-Driven Engineering - Guest Editor's Introduction », *IEEE Computer*, February 2006, Vol. 39, No. 2, pp. 25-31.

[Touraille et al. 2009] 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.

[Traore 2008] Traore M.K., « SimStudio: a Next Generation Modeling and Simulation Framework », *Proceedings of the ACM/IEEE 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, March 3-7, 2008, Marseille, France, Article No 67.

[Vangheluwe 2000] Vangheluwe H.L.M., « DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling », *IEEE International Symposium on Computer-Aided Control System Design*, 25-27 September, 2000, Anchorage, Alaska, USA, pp. 129-134.

[Zeigler et al. 2000] Zeigler B.P., Praehofer H., Kim T.G., *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, Academic Press, ISBN 0127784551, 2000.