DEVSimPy - v2.8

06/04/2016

Environnement de modélisation et de simulation à évènements discrets des systèmes complexes en langage Python

Le scientifique n'a de cesse de vouloir comprendre les phénomènes qui l'entourent. Il utilise pour cela des formalismes de description qui le conduisent souvent à construire des modèles à événements discrets de systèmes complexes. Ces modèles sont ensuite simulés afin de reproduire avec justesse le phénomène observé. Ce processus s'effectue par l'intermédiaire d'environnements logiciels de plus en plus performants et intuitifs. DEVSimPy est un logiciel de modélisation et de simulation des systèmes complexes décrits à partir du formalisme à évènements discrets DEVS. Ce logiciel est un projet Open Source sous licence GPL V3 et il est entièrement implémenté en langage Python.

Laurent CAPOCCHI
Maître de conférences en informatique
Laboratoire SPE UME CNRS 6134
capocchi@univ-corse.fr
http://spe.univ-corse.fr/spip.php?rubrique41

E some s'acconcenu per strada

SOMMAIRE

À PROPOS DE DEVSIMPY

Le présent document constitue une introduction à l'utilisation de l'environnement de modélisation et de simulation à événements discrets DEVSimPy. DEVSimPy est une interface graphique implémentée en langage Python permettant la modélisation et la simulation automatique des systèmes complexes décrits dans le formalisme DEVS (Discrete EVent system Specification). Ce logiciel à été initié par Laurent Capocchi, maître de conférences en informatique, puis proposé aux membres du projet TIC du laboratoire SPE (Sciences Pour l'Environnement) UMR CNRS 6134 de l'Université de Corse « Pasquale Paoli ».

Des informations complémentaires concernant DEVSimPy sont disponibles à l'adresse https://github.com/capocchi/DEVSimPy.

CADRE ET PUBLIQUE CONCERNÉ

Ce document s'adresse aux personnes désirant se familiariser avec le logiciel DEVSimPy. Il n'entre pas dans les détails de l'architecture logicielle de DEVSimPy.

Ce document n'est pas dédié à l'enseignement du langage Python et à la programmation des interfaces graphiques. Bien que le langage Python soit reconnu pour être un langage rapide à maîtriser (du fait notamment de sa syntaxe condensée, de son caractère dynamique et non typé), les programmeurs débutants dans le langage Python devront se perfectionner à partir de documents autres que celui-ci. Il en est de même pour la librairie graphique wxPython qui est utilisée par DEVSimPy.

Ce document n'est pas un manuel de référence. Certains détails sont volontairement évités afin de rendre le document plus simple et plus concis.

Chaque section débute par une petite introduction qui a pour but d'informer le lecteur sur le contenu de sa lecture. De cette manière il sera à même d'évaluer s'il est nécessaire ou non de lire la section en fonction de ses connaissances.

COPYRIGHT, LICENCE ET REDISTRIBUTION

Le logiciel DEVSimPy est un projet Open Source sous licence GNU GPL v3. Il est donc libre de droit et l'utilisateur a quatre libertés principales :

- la liberté d'utiliser le logiciel pour n'importe quel usage ;
- la liberté de modifier le programme pour répondre à ses besoins ;
- la liberté de redistribuer des copies ;
- la liberté de partager avec d'autres les modifications qu'il a faites.

Pour plus de détails concernant la licence GNU GPL v3 l'utilisateur peut consulter la page web : http://www.gnu.org/licenses/quick-guide-gplv3.fr.html.

CONTRIBUTION

Les commentaires, les questions, les corrections, les critiques et tout autre retour concernant le logiciel DEVSimPy ou le document présent sont les bienvenus. Vous pouvez envoyer ceux-ci par mail à l'adresse capocchi@univ-corse.fr.

Des vidéos d'utilisation de DEVSimPy peuvent être sur la chaîne Youtube. N'hésitez pas à laisser des commentaires.

INTRODUCTION

POURQUOI DEVSIMPY ?

Le projet libre DEVSimPy est né au sein de l'équipe du laboratoire SPE (Sciences Pour l'Environnement) de l'université de Corse « Pasquale Paoli ». Ce laboratoire est une UMR (Unité Mixte de Recherche) CNRS 6134 et travaille depuis plus de 20 ans dans le domaine de la modélisation et de la simulation de systèmes complexes. Il est notamment spécialisé dans le domaine des systèmes environnementaux modélisés avec le formalisme DEVS (Discrete EVent system Specification). C'est au travers de ces études que le laboratoire a publié plusieurs travaux sur des extensions du formalisme DEVS répondant à des problématiques issues des différents domaines d'applications comme :

- la simulation de feux de forêt ;
- le test comportemental de circuits digitaux ;
- la simulation comportementale de bassins versants ;
- la simulation de réseaux de capteurs ;
- la simulation de mythes et de légendes ;
- etc.

Ces travaux ont toujours été réalisés dans un objectif de généricité et de réalisabilité des modèles DEVS. Le langage de programmation utilisé (C, C++, C#, Java, Fortran, Python) pour implémenter ces travaux dépendait principalement de la nature du domaine d'application et des préférences pour une technologie particulière de la part des personnes en charge de l'étude. Il s'avère que depuis quelques années, la bibliothèque PyDEVS est de plus en plus utilisée au sein du laboratoire. Cette librairie est développée en langage Python et offre à l'utilisateur une API permettant de modéliser et de simuler des modèles DEVS. Bien que cette API soit embarquée dans l'excellent logiciel de multi-modélisation ATOM3, il n'existe pas d'interface graphique exploitant PyDEVS pour faciliter le développement exclusif de modèles DEVS. En effet, lorsque l'utilisateur doit implémenter le couplage entre les modèles DEVS, il doit instancier plusieurs fois une méthode par codage, dans un ordre bien particulier. Cette manipulation est souvent la source d'erreurs qu'il faut ensuite debugger en vérifiant tous les couplages. De plus, il faut réitérer cette opération pour chaque système et sous-système modélisé. De plus, si besoin est, il n'existe aucun moyen de réutiliser de manière simple des modèles couplés dans une autre partie du système.

L'idée de départ du projet libre DEVSimPy était d'apporter aux utilisateurs de PyDEVS une interface graphique facilitant la connexion des modèles DEVS ainsi que leur réutilisabilité. La première version de DEVSimPy permettait donc (et permet toujours) d'importer des classes PyDEVS et de les manipuler comme des objets visuelles afin de construire des modèles plus complexes. Le logiciel propose ensuite de simuler de manière automatique ces modèles par un simple clic qui a pour effet d'invoquer le simulateur PyDEVS.

A l'heure actuelle, DEVSimPy est une interface graphique permettant :

- l'exploitation de fichiers PyDEVS ;
- la création de modèles ;
- la réutilisation de modèles à partir de librairies ;
- la simulation interactive des modèles ;
- l'implémentation des concepts attachés aux différentes branches du formalisme DEVS (DEVS parallel, DEVS concurrent, DEVS floue, DEVS-SIG, DEVS-SMA, ...) ;

- l'expérimentation de nouveaux domaines d'applications et de concepts DEVS ;
- etc.

DEVSimPy a été pensé pour simplifier la phase finale du modélisateur qui consiste à coupler les modèles entre eux afin de les simuler pour enfin analyser leurs comportements. Pour ce faire, l'utilisateur peut utiliser DEVSimPy de différente manière :

- soit il implémente sa librairie de fichiers PyDEVS en utilisant un logiciel d'édition de code Python puis il importe ces derniers dans DEVSimPy afin de les interconnecter et de les simuler de manière automatique ;
- soit il crée, interconnecte et simule directement sa librairie de modèles au travers de DEVSimPy sans passer par un logiciel externe. Dans tout les cas les interconnexions entre modèles se font par un simple glisser-déposer ou par l'intermédiaire d´un gestionnaire d'interconnexions.

La création et l'importation de librairies de modèles est simplifiée par DEVSimPy. Une interface de dialogue concise est proposée à l'utilisateur afin de gérer ces librairies de modèles. Hormis le fait qu'ils puissent être simulés, les modèles peuvent également être sauvegardés dans un format compressé embarquant les dépendances directes des modules Python (import). Ce mode de stockage sépare le comportement (*.py*, ...) de l'aspect graphique (*.dat*, *.jpg*, *.png*) des modèles. De plus, il permet de sauvegarder la représentation des modèles et donc d'associer une couche graphique aux fichiers comportementaux PyDEVS.

L'exploitation des résultats de simulation est améliorée avec l'utilisation de DEVSimPy. En effet, le mode verbeux de PyDEVS est embarqué dans une fenêtre de log donnant la possibilité de naviguer dans la trace de la simulation par recherche de mot clés. De plus, DEVSimPy propose une gestion de plugin permettant d'étendre ces fonctionnalités de base à l'intérieur de morceaux de code activables en fonction des besoins et des applications. DEVSimPy possède entre autres un plugin qui permet de suivre pas à pas le comportement des modèles pendant la simulation. Enfin, des modèles de base permettent de tracer en temps réel des données temporelles dans des graphiques dynamiques. L'implémentation de tels plugins n'est pas traitée dans ce document mais si le lecteur est intéressé par ce chapitre il pourra se référer au guide du développeur.

LES SYSTÈMES COMPLEXES

L'être humain n'a de cesse de vouloir analyser et comprendre les phénomènes qui l'entourent. Lorsqu'il entreprend cette tâche, il tend naturellement vers la définition des acteurs, des données et de leurs interactions possibles au sein du phénomène évoluant dans un environnement donné. On dit qu'il définit **un système**. Il existe plusieurs manières de qualifier un système selon :

- qu'il réagisse ou non avec son environnement *(système ouvert/système fermé)* ;
- qu'il soit facilement compréhensible et prévisible *(système simple)* ;
- qu'il soit régi par un nombre fini de règles précises, simples mais imprévisibles *(système complexe)* ;
- qu'il soit incapable d'être régi par des règles compliquées tout en respectant des lois simples *(système chaotique)*.

Les phénomènes naturels sont souvent représentés par des **systèmes complexes**. Par exemple, un banc de poissons évoluant dans la mer présente toutes les caractéristiques d'un système complexe. C'est un système qui réagit avec son environnement et notamment avec les

autres espèces prédatrices susceptibles de les attaquer. Il est très difficile de prédire le mouvement du banc de poissons lors d'une attaque et il est également très compliqué de connaître l'évolution de sa population au cours du temps. Cependant, comme tout système complexe, il est régit par un ensemble de règles finies comme celles que l'on pourrait définir pour une espèce particulière de poisson.

LA SIMULATION À ÉVÉNEMENTS DISCRETS

Lorsque l'on veut représenter un système complexe, il est d'usage de définir **un modèle**. Le modèle permet de formaliser le comportement d'un système en spécifiant un ensemble de règles le plus souvent à l'aide des mathématiques. **La modélisation d'un système** conduit à le rendre manipulable par l'être humain car il possède alors un modèle qu'il pourra expérimenter à l'infini. L'avantage de posséder un modèle est qu'il peut être simulé. Ces simulations permettent de mettre en condition d'expérimentation le système afin d'observer son comportement. Un modèle valide est un modèle qui, lorsqu'il est simulé, reproduit parfaitement le comportement du système qu'il représente.

La modélisation d'un système nécessite l'utilisation d'un **formalisme de description**. Le choix du formalisme est conditionné par la représentation de l'espace, du temps et des états d'un système. Pendant le processus de modélisation, il est important de savoir si **le temps** et **les états** représentant le système sont considérés comme **discrets** ou **continus**. De la même manière, il est important de savoir si la notion d'**espace** est à prendre en considération dans l'évolution du système et si oui, si elle est modélisée de manière discrète ou continue. Lorsque qu'un système est décrit en considérant ses états et le temps de manière continu, il peut être modélisé à partir d'un formalisme comme les équations différentielles classiques. Lorsque la notion d'espace doit en plus être prise en compte, il est préférable d'utiliser les équations différentielles aux dérivées partielles. Lorsque le changement d'état d'un système est discret et que le temps est considéré comme continu, le formalisme utilisé peut être celui des **événements discrets** quelque soit la manière dont l'espace est considéré (discret, continu ou absent).

Le choix du formalisme (et donc la manière dont les états, le temps et l'espace sont considérés) est souvent conditionné par la culture scientifique du modélisateur. Traditionnellement, un système complexe est modélisé en considérant son évolution de manière continue dans le temps (ou dans l'espace) et son comportement est décrit à partir d'équations différentielles qui sont résolues de manière analytique ou numérique en fonction de la complexité du modèle. Dans ce type d'approche, l'évolution du système (de ces états) est modélisée indépendamment de l'échelle du temps et l'on peut dire que la résolution du modèle se fait avec une approche de simulation continue. Cette méthode de modélisation s'applique très bien pour des systèmes dont les états évoluent de manière continue dans le temps (ou dans l'espace) et dont les états doivent être observables à n'importe quel instant. Cependant, certains systèmes n'évoluent dans le temps qu'à des instants précis et il n'est pas nécessaire de connaître de manière précise leur comportement entre ces instants. Ils évoluent de manière discrète et seules importent leurs observations à ces instants précis ou ils changent d'état. Dans ce cas, **la modélisation et la simulation à événements discrets** peut être utilisée pour étudier ces systèmes. Il est important de noter que le choix du formalisme dépend aussi de la nature discrète ou continue du système que l'on désire modéliser. La simulation continue sera privilégiée dans le cas de l'étude d'un modèle de dynamique de populations dans un écosystème du fait de la présence de variables d'états continues comme la vitesse de déplacement des individus. Par contre, la simulation à événements discrets sera préférée pour la modélisation d'un système de chaîne de production industrielle dans lequel il est important de connaître l'évolution d'un produit par étapes.

LE FORMALISME DEVS

Le formalisme DEVS (Discrete EVent system Specification) [1, 2] a été introduit à la fin des années 70 par le professeur B.P Zeigler qui avait pour objectif de donner un socle mathématique solide à la modélisation et la simulation (M&S) des systèmes à événements discrets. Un système à événements discrets est un système pouvant être décrit à partir d'un ensemble d'états et de règles de transition entre ces états. Le formalisme DEVS permet la représentation d'un système comme un modèle ou un ensemble de modèles possédant des états et des transitions. De plus, il donne la possibilité de définir de manière distincte la structure d'un système.

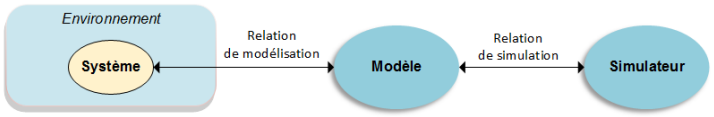


FIGURE 1 : FRAMEWORK M&S.

Le professeur Zeigler a proposé, dans [2] une architecture conceptuelle pour la modélisation et la simulation, des systèmes particulièrement adaptés au formalisme DEVS. Comme le montre la Figure 1, cette architecture présente trois entités :

- **le système** : c'est le phénomène observé dans un environnement donné. L'environnement donne les spécifications des conditions dans lesquelles évolue le système et permet son expérimentation et sa validation ;
- **le modèle** : c'est la représentation du système basée généralement sur la définition d'ensemble d'instructions, de règles, d'équations, de contraintes, permettant de générer un comportement après simulation. Le modèle défini un comportement et une structure pour un système évoluant dans un environnement donné ;
- **le simulateur** : c'est une entité qui est responsable de l'interprétation du modèle (exécuter ces instructions) pour générer son comportement.

Ces entités sont reliées pas deux relations :

- **la relation de modélisation** : elle est composée des règles de construction et de validation du modèle ;
- **la relation de simulation** : elle est composée de règles d'exécution du modèle qui permettent au simulateur de génère correctement le comportement attendu du système.

La séparation explicite entre les entités permet de bénéficier de plusieurs avantages comme simuler un modèle avec différents types de simulateurs ou différents types d'environnements.

L'engouement pour la programmation orientée objets au début des années 80 a conduit le professeur Zeigler à utiliser une approche « objet » pour définir son formalisme. Il définit notamment deux types de modèles : les **modèles atomiques** pour l'aspect comportemental des systèmes et les **modèles couplés** pour l'aspect structurel. Par définition, un modèle couplé peut contenir plusieurs modèles atomiques ou plusieurs modèles couplés. Par contre un modèle atomique est indissociable et tout modèle couplé peut être représenté par un modèle atomique unique. Le modèle atomique est la pièce élémentaire de la modélisation DEVS et il est définit par un ensemble d'états et de fonctions de transition entre ces états (automate).

L'un des avantages principaux du formalisme DEVS est qu'il offre **la simulation automatique** de ses modèles. En effet, le simulateur DEVS va exécuter les fonctions de transition des modèles atomiques et gérer la communication entre les modèles de manière automatique à partir d'un arbre de simulation. Il associe un simulateur par modèle atomique et un coordinateur par modèle couplé. Cette représentation hiérarchique permet de contrôler la simulation grâce à une distribution réglementée des événements entre modèles.

LES MODÈLES ATOMIQUES

Dans **le formalisme DEVS classique avec port,** un modèle atomique MA est spécifié par la définition de sept entités :

$$\boldsymbol{MA = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a \rangle}$$

Avec :

- **l'ensemble des entrées** : $X = \{(p, v) | p \in P_e, v \in V_x\}$ ou P_e et V_x sont deux ensembles finis représentant l'ensemble des ports d'entrées et des valeurs portées par les événements reçus en entrée ;
- **l'ensemble des sorties** : $Y = \{(p, v) | p \in P_s, v \in V_y\}$ ou P_s et V_y sont deux ensembles finis représentant l'ensemble des ports de sorties et des valeurs portées par les événements générées en sortie ;
- **l'ensemble des états** : $S = \{s_i | i \in R^+\}$;
- **la fonction de transition interne** : $\delta_{int}(S) \rightarrow S$;
- **la fonction de transition externe** : $\delta_{ext}(Q \times X) \rightarrow S$;
- **la fonction de sortie** : $\lambda(S) \rightarrow Y$;
- **la fonction d'avancement du temps** : $t_a(S) \rightarrow R_0^+ \cup \infty$.

Pour définir un modèle atomique, il faut dans un premier temps dénombrer les états possibles (S) du système ainsi que leur durée de vie ($t_a(S)$). Ensuite, il faut déterminer dans quel ordre ces états changent, indépendamment de toutes perturbations extérieures ($\delta_{int}(S)$) et quelle sortie sera générée après un tel changement d'état ($\lambda(S)$). Enfin, il faut ajouter à cet automate la réponse à une perturbation extérieure, c'est-à-dire comment le système change d'état après avoir reçu un message externe ($\delta_{ext}(Q \times X)$). Le nouvel état va dépendre de la valeur portée par l'événement présent sur le port d'entrée (X) mais également de l'ancien état *s* et du temps écoulé depuis le dernier changement d'état e ($Q = \{(s, e) | s \in S, 0 < e < t_a(s)\}$). Lorsqu'un message externe intervient alors qu'une transition interne est prévue au même instant, le formalisme classique avec port privilégie la fonction de transition externe. Cependant, une extension du formalisme (DEVS parallèle [2]) prévoit une autre fonction de transition supplémentaire qui permet de résoudre un tel conflit : $\delta_{conv}(S \times X^b)$ avec $\delta_{conv}(S \times \emptyset) = \delta_{int}(S)$. C'est à l'intérieur de cette fonction que le modélisateur a la possibilité d'exécuter la fonction de transition interne ou la fonction de transition externe. X^b est un ensemble de groupe d'événements d'entrée intervenant à la même date (notion de

La fonction $t_a(S)$ est invoquée en début de simulation (pour déterminer la liste des modèles imminents) puis à chaque changement d'état afin de déterminer la durée de vie de celui-ci. Si $t_a(s_i) = 0$ alors la durée de vie de l'état s_i est nulle et l'état prend fin aussitôt. Par contre, si $t_a(s_i) = +\infty$ alors la durée de vie de l'état s_i est infinie (état eternel) et si aucun événement extérieur est intercepté, le modèle ne changera plus jamais d'état.

bag). La fonction δ_{conv} traitera alors plusieurs événements arrivant à la même date sur plusieurs ports d'un modèle atomique.

La définition de ces spécifications doit se faire en ayant à l'esprit l'algorithme de simulation. Nous pouvons dire de manière simplifiée que le modèle atomique effectue un *cycle interne* récurrent : $\delta_{int}(S) \Longrightarrow t_a(S) \Rightarrow \lambda(S)$. Lorsqu'un événement externe intervient (qui n'est pas en conflit avec la fonction de transition interne), ce cycle interne est rompu et un nouveau *cycle externe* ponctuel prend place : $\delta_{ext}(X \times S) \Longrightarrow t_a(S) \Rightarrow \lambda(S) \Longrightarrow \delta_{int}(S) \Longrightarrow t_a(S)$. Si le modélisateur garde à l'esprit cet algorithme simplifié, il implémentera des modèles atomiques compatibles avec l'algorithme de simulation DEVS. Partant de ces considérations, tout modèle atomique qui ne possède pas de port d'entrée n'a pas besoin d'implémenter la fonction $\delta_{ext}(X \times S)$. On peut dire que ce type de modèle est un **générateur d'évènement**. Tout modèle atomique qui ne possède pas de port de sortie n'a pas besoin d'implémenter la fonction $\lambda(S)$. Ce type de modèle peut être qualifié de **collecteur d'évènements**.

EXEMPLE

Description comportementale : Nous allons donner les spécifications DEVS d'un modèle atomique nommé « EXEC » qui représente un système qui devient « actif » (s_1) lorsqu'il reçoit des entrées x_i sur un port d'entrée p_1, qui traite cette donnée durant un temps « *proc* » puis qui génère des sorties y_i sur un port de sortie p_2. Si un événement externe intervient pendant que le système est dans l'état actif son temps de vie sera diminué du temps écoulé depuis le dernier changement d'état (e). Si aucun événement n'intervient, le système reste dans un état « passif » (s_2). Le système est dans un état initial « passif ».

FIGURE 2 : MODÈLE ATOMIQUE EXEC.

Spécifications DEVS :

- $X = \{(x_i, p_1) | i \in R^+\}$;
- $Y = \{(y_i, p_2) | i \in R^+\}$;
- $S = \{s_1, s_2\}$;
- $\delta_{int}(S)$:
  - si $s = s_1$ alors $s = s_2$ pendant $t_a(s_1) = \infty$;
  - sinon s ne change pas.
- $\delta_{ext}(Q \times X)$:
  - si $s = s_2$ alors $s = s_1$ pendant $t_a(s_1) = proc$;
  - sinon $t_a(s_1) = t_a(s_1) - e$.
- $\lambda(S)$: si $s = s_1$ alors envoie d'un message de sortie avec la valeur y_i .
- $t_a(S)$:
  - si $s = s_1$ alors retourner *proc* ;
  - sinon retourner l'infinie.

Il arrive souvent que l'on remplace la fonction d'avancement du temps ($t_a(S)$) par la manipulation d'un attribut du modèle atomique nommé sigma (σ). La description ci-dessus s'écrit alors de la manière plus condensée pour $\delta_{int}(S)$, $\delta_{ext}(Q \times X)$ et $t_a(S)$:

- $\delta_{int}(S)$:

- si $s = s_1$ alors $s = s_2$ et $\sigma = \infty$;
- sinon s ne change pas.

- $\delta_{ext}(Q \times X)$:
  - si $s = s_2$ alors $s = s_1$ et $\sigma = proc$;
  - sinon $\sigma = \sigma - e$.
- $t_a(S)$: retourner σ.

Automate à états finis : Il est assez courant de représenter la description comportementale à l'aide d'un automate.

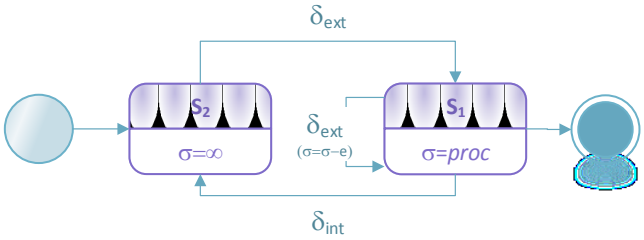


FIGURE 3 : AUTOMATE DU MODÈLE EXEC.

L'automate débute dans l'état s_2 avec une durée de vie infinie (Figure 3). Lorsqu'un évènement arrive sur le port d'entrée, l'état change en s_1 pendant un temps *proc*. Si un évènement externe intervient pendant ce temps, l'état ne change pas mais le temps de vie est mis à jour en considérant le temps passé depuis le dernier changement d'état (e).

Trajectoire d'états : La trajectoire d'états permet de tracer les changements d'état en fonction des événements d'entrées.

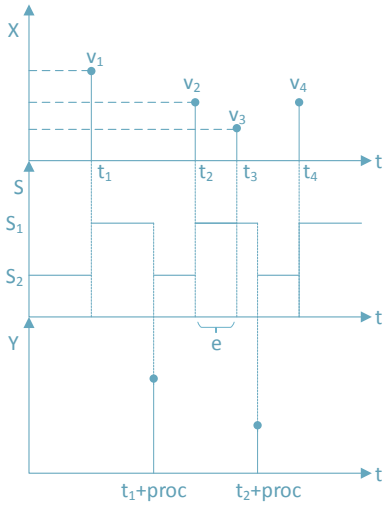


FIGURE 4 : TRAJECTOIRE D'ÉTATS DU MODÈLE EXEC.

Lorsqu'un événement intervient au temps t_1, le système passe de l'état passif s_2 vers l'état actif s_1 pendant un temps *proc* (Figure 4). Lorsque le temps t_1+proc est écoulé, le modèle génère un événement de sortie. L'événement en entrée intervenant au temps t_3 ne change pas l'état du système (s_1) mais la durée de vie de ce dernier est mis à jour en fonction de e.

Le travail du modélisateur est non seulement de déterminer combien de modèles atomiques sont nécessaires pour représenter le système étudié mais également d'implémenter le comportement de chacun d'eux. L'utilisation d'un environnement de développement DEVS orienté objets permettant l'implémentation d'un modèle atomique comme un objet devient un atout pour le modélisateur. De même, si l'environnement procure un éditeur de code avec toutes les fonctionnalités courantes (la vérification syntaxique, des raccourcis vers des commandes courantes comme le changement d'état, etc.) l'utilisateur sera aidé dans sa tâche de développement.

LES MODÈLES COUPLÉS

Dans le formalisme DEVS classique avec port, un modèle couplé MC est spécifié par la définition de sept entités :

$$\boldsymbol{MC = \langle X, Y, D, EOC, IC, EIC, select \rangle}$$

Avec :

- **l'ensemble des entrées** : $X = \{(p, v) | p \in P_e, v \in V_x\}$ ou P_e et V_x sont deux ensembles finis représentant l'ensemble des ports d'entrée et des valeurs portées par les événements reçus en entrée ;
- **l'ensemble des sorties** : $Y = \{(p, v) | p \in P_s, v \in V_y\}$ ou P_s et V_y sont deux ensembles finis représentant l'ensemble des ports de sortie et des valeurs portées par les événements générés en sortie ;
- **l'ensemble des modèles atomiques ou couplés** : $D = \{m_i | i \in R^+\}$;
- **l'ensemble des couplages de sortie externes** : $EOC = \{((m_i, p_{sj}), ((MC, p_{sk})) | i, j, k \in R^+\}$;
- **l'ensemble des couplages internes** : $IC = \{((m_i, p_{sj}), ((m_k, p_{el})) | i, j, k, l \in R^+\}$;
- **l'ensemble des couplages de sorties externes** : $EIC = \{((MC, p_{ei}), ((m_j, p_{ek})) | i, j, k \in R^+\}$;
- **la fonction de sélection** permettant de régler le conflit d'activation des modèles : $select(D) \rightarrow m_i$.

Un modèle couplé est utilisé pour structurer la modélisation. Le seul effet comportemental que l'on peut lui attribuer et lié à la fonction *select*. En effet, celle-ci détermine l'ordre dans lequel des modèles en conflit d'exécution (externe ou interne) doivent s'ordonnancer. L'implémentation reste bien sûr séquentielle. De plus, la fonction *select* est souvent utilisée lorsque les modèles atomiques DEVS implémentent une fonction de transition interne. Dans ce cas précis, l'exécution des modèles n'est pas ordonnée et les conflits d'exécution entre modèles interviennent. Dans le cas d'un système modélisé par un ensemble interconnecté de modèles atomiques qui s'exécutent les uns après les autres uniquement après avoir reçu les événements de leur voisin, la fonction *select* n'a pas besoin d'être implémentée.

EXEMPLE

Considérons un modèle couplé MC_1 composé de trois modèles atomiques MA_1, MA_2 et MA_3 et d'un autre modèle couplé MC_2 composé de deux modèles atomiques MA_4 et MA_5. Afin de simplifier l'exemple, nous allons attribuer un seul port d'entrée Pe_1 et un seul port de sortie Ps_1 à chacun des modèles. Voici une représentation graphique du modèle couplé MC_1 :

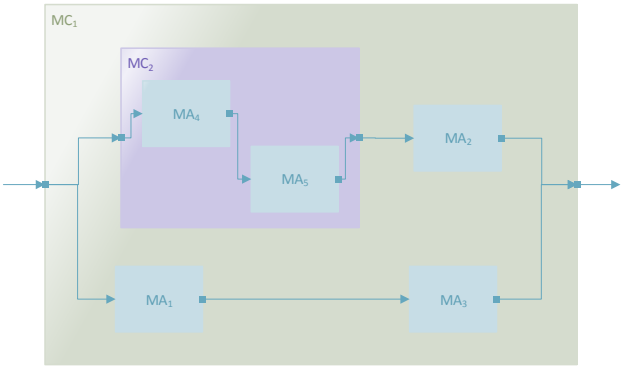


FIGURE 5 : EXEMPLE D'UN MODÈLE COUPLÉ DEVS.

Le modèle couplé est spécifié de la manière suivante dans DEVS :

- $X = \{(x_i, p_1) | i \in R^+\}$;
- $Y = \{(y_i, p_2) | i \in R^+\}$;
- $D = \{MA_1, MA_2, MA_3, MA_4, MA_5, MC_2\}$;
- $EOC = \{\big((MA_2, ps_1), (MC_1, ps_1)\big), ((MA_3, ps_1), (MC_1, ps_1))\}$;
- $IC = \{\big((MA_1, ps_1), (MA_3, pe_1)\big), ((MC_2, ps_1), (MA_2, pe_1))\}$;
- $EIC = \{\big((MC_1, pe_1), (MC_2, pe_1)\big), ((MC_1, pe_1), (MA_1, pe_1))\}$;
- $select = (MC_2, MA_1, MA_2, MA_3)$.

Dans l'exemple ci-dessus, lors d'un éventuel conflit d'exécution, le modèle couplé MC_2 est prioritaire sur MA_1 qui est prioritaire sur MA_2 qui est prioritaire sur MA_3. Cet ordre de priorité est implémenté dans la fonction *select*.

Le formalisme DEVS assure que tout modèle couplé est équivalent à un modèle atomique unique. En fait, un modèle atomique peut être décomposé en plusieurs sous-modèles atomiques organisés dans plusieurs modèles couplés. Indépendamment des performances de la simulation séquentielle (nous y reviendrons plus tard), le choix du nombre de modèles couplés utilisés pour modéliser un système ne relève que de considérations purement esthétiques. L'utilisateur organise ces modèles de manière hiérarchique comme il le désire dans des modèles couplés avec le **niveau de description** désiré.

LA HIÉRARCHIE DE DESCRIPTION

Si avec le formalisme DEVS la simulation est automatique, la modélisation reste une tâche ardue et incontournable pour le scientifique. Lorsque le comportement du système est déterminé et que les états sont définis, il faut dénombrer les modèles atomiques et couplés puis implémenter les modèles DEVS. La hiérarchie de description constitue le niveau avec

lequel le modélisateur organise sa modélisation. Il peut choisir de ne pas utiliser de modèles couplés et donc de représenter son modèle complet comme une interconnexion d'un nombre important de modèle atomique ou par un seul modèle atomique. Dans les deux cas, le niveau de description est très faible. Si par contre le modélisateur utilise des modèles couplés pour encapsuler plusieurs modèles atomiques ou couplé à des niveaux de description différents, le niveau de description du modèle complet est très fort.

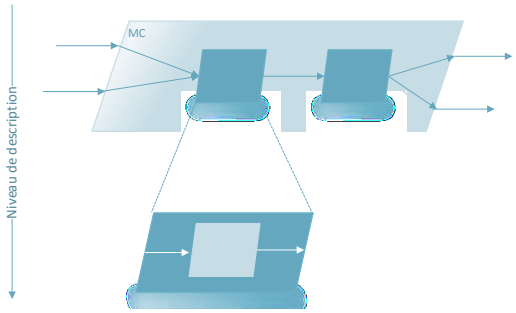


FIGURE 6 : NIVEAU DE DESCRIPTION DANS DEVS.

Comme le montre la Figure 6, le niveau de description du modèle ne modifie pas le nombre de ports d'entrée et de sortie des modèles. Ce n'est qu'une structuration hiérarchique des modèles.

Si l'on met de coté les notions de performance de simulation, le choix du nombre de niveaux de description n'est motivé que par une vue de l'esprit. En effet, généralement les modèles couplés sont utilisés pour encapsuler un certain nombre de modèles atomiques ou couplés censés représenter un certain niveau de complexité. Bien entendu, la notion de complexité est relative à ce que le modélisateur considère comme complexe. Nous ne pouvons pas donner de définition de la complexité pour un ensemble de modèles sauf dire que plus ce nombre est important, plus la représentation est « complexe » et qu'il vaut mieux, à ce moment là, utiliser un modèle couplé pour masquer cette « complexité ».

LA HIÉRARCHIE D'ABSTRACTION

Lorsque le modélisateur débute son travail de modélisation, il possède une certaine représentation du comportement du système avec un certain niveau de détail. Ce niveau de détail va lui permettre d'aborder le problème à différents **niveaux d'abstraction**. Par exemple, il utilisera un modèle atomique unique avec un certain nombre d'entrée et de sorties, si il désire implémenter le comportement macroscopique d'un modèle. Par contre, si il possède également une description microscopique du modèle avec un nombre de ports d'entrées et de sorties différent, il utilisera peut être un modèle couplé composé de plusieurs autres modèles (atomiques ou couplés). Les deux modèles macroscopique et microscopique donnent le même comportement d'un modèle mais avec des niveaux de détail différents. C'est ce que l'on appelle les niveaux d'abstraction.

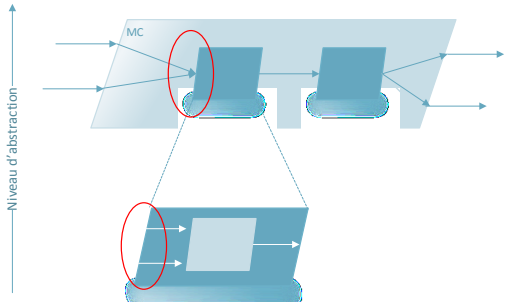


FIGURE 7 : NIVEAU D'ABSTRACTION DANS DEVS.

Comme le montre la Figure 7, le changement de niveau d'abstraction fait intervenir des nouveaux modèles plus détaillés lorsque le niveau d'abstraction diminue. Ici, un des modèles nécessite une description avec deux ports d'entrée lorsque son niveau de détail diminue alors qu'il n'avait qu'un seul port d'entrée dans un niveau d'abstraction plus haut. Tout le problème est maintenant de savoir comment la transformation se réalise au niveau du formalisme DEVS. À l'heure actuelle, des travaux de recherche sont menés pour définir de manière formelle cette notion dans le formalisme DEVS. Pour plus d'information concernant la hiérarchie d'abstraction, le lecteur peut se référer à [3, 4, 5].

LES EXTENSIONS

DEVS parallèle : dans cette version, la notion de « event bag » à été ajoutée. Cette notion intègre le fait que plusieurs événements intervenant au même instant peuvent être regroupés dans un *bag*. L'ensemble est noté X^b. De la même manière, un modèle atomique peut envoyer plusieurs événements en sortie au même instant. L'ensemble de sortie est noté alors Y^b. La fonction $\delta_{conv}(S \times X^b)$ est introduite afin de résoudre le conflit d'exécution entre les fonctions de transition interne et externe d'un modèle atomique avec le cas particulier ou $\delta_{conv}(S \times \emptyset) = \delta_{int}(S)$. L'association de ces deux notions (*bag* et δ_{conv}) permet de gérer les collisions entre les fonctions de transition interne et externe et en même temps de traiter plusieurs événements arrivants au même instant sur un modèle atomique. Dans le formalisme DEVS classique, à chaque arrivée d'un événement, la fonction de transition externe est invoquée. Il y avait donc autant d'invocations que de messages simultanés sur les ports d'un modèle atomique. Avec cette extension, les événements simultanés sont disponibles dans l'unique invocation de la fonction de transition. Cette extension permet donc d'améliorer les performances du simulateur.

DEVS continue : Ce n'est pas une extension du formalisme DEVS à proprement dit mais plutôt une utilisation de DEVS dans le cadre des systèmes continus. Les systèmes continus manipulent le temps comme une variable continue et sont souvent représentés de manière mathématique (équations différentielles ou partielles). La résolution d'un tel système, nécessite une discrétisation du temps à pas fixe ou variable (méthode de résolution Rush-Kutta). L'idée du professeur Zeigler [2] exploitée ensuite par le chercheur E. Kofman [3] était de remplacer la discrétisation du temps par la quantification des états d'un modèle. Cette considération à donné naissance à un modèle atomique **intégrateur** qui permet de résoudre, moyennant la définition d'un pas de quantification, les systèmes continus. Cette méthode est nommée « Quantized State System » (QSS). Les états peuvent être quantifiés avec un certain degré de précision qui dépend de l'ordre du polynôme utilisé : QSS1,2 [4] ou 3 [5] pour l'ordre 1,2 ou

3. Les dernières versions de cette extension sont capables d'adapter le pas de quantification en fonction de la dynamique du système et sont notées QSSB, QSSC ou LIQSS [6].

DEVS dynamique : Cette extension prévoit la modification structurelle des modèles DEVS (atomiques ou couplés) pendant la simulation. Cette extension a été introduite par le Professeur F. Barros dans [7]. Elle est réellement possible grâce au caractère abstrait de l'arbre de simulation qui mettra à jour les simulateurs et les coordinateurs à la volée en fonction des modifications structurelles des modèles atomiques et des modèles couplés. DEVS dynamique prévoit également l'ajout et la suppression pendant la simulation de modèles atomiques ou couplés.

DEVS concurrent : La simulation comparative et concurrente (SCC) est une technique qui est née dans le domaine du test de circuits au niveau portes. Elle consiste à exécuter une simulation de référence (dite aussi simulation saine, qui est un terme hérité du domaine test) avec plusieurs simulations concurrentes (dites simulations fautives qui est un terme également hérité du domaine du test dans le sens ou une simulation fautive est destinée à simuler un défaut particulier dans un circuit) en une seule exécution. Durant l'exécution, les simulations concurrentes sont comparées à **la simulation de référence** afin de supprimer ou de créer d'autres simulations concurrentes. Par exemple dans le cas du test de circuit, les simulations concurrentes qui rejoignent la **simulation saine** sont automatiquement éliminées puisque les défauts qu'elle simule ne seront pas observables en fin de simulation. La technique de la SCC a été introduite dans le formalisme DEVS pour donner naissance au formalisme BFS-DEVS [11]. Cette extension consiste en l'ajout d'une nouvelle **fonction de transition dite fautive** (δ_{fault}) qui sera exécutée après chaque fonction de transition externe. Une généralisation de ce formalisme a été ensuite proposée dans [12] et appliquée dans le domaine des réseaux de neurones.

Le langage Python

Des travaux de recherches ont montré l'efficacité du langage Python pour la modélisation et la simulation de systèmes à événements discrets [8]. Python est dynamique, non typé, portable et repose sur une syntaxe simple et réduit donc considérablement le temps d'implémentation des concepts sur lesquels reposent les logiciels. De plus, du fait de son caractère dynamique, il permet la modification des classes et des instances pendant l'interprétation d'un code et facilite la Programmation Orientée Aspect (POA). DEVSimPy est entièrement codé en langage Python et utilise plus particulièrement la bibliothèque graphique wxPython. Avec le projet libre DEVSimPy l'équipe du laboratoire SPE voulait développer une interface graphique permettant de construire des modèles couplés à partir de modèles atomiques décrits dans des fichiers PyDEVS. Ceci constitue le point de départ du projet et a donné naissance à un logiciel complet dont nous allons décrire l'utilisation dans la suite du document.

L'API PyDEVS

L'API PyDEVS a été implémentée au début des années 2000 pour permettre la simulation des modèles DEVS dans l'outil ATOM3. Aujourd'hui, l'API à été largement améliorée et mise à jour en fonction des avancées du formalisme DEVS. PyPDEVS [9] est une version de PyDEVS permettant de faire de la simulation parallèle et distribuée. La version 2.8 de DEVSimPy propose cette API mais nous ne la présenterons pas dans ce document.

INSTALLATION ET CONFIGURATION

DEVSimPy est un projet Open Source développé à partir de l'API graphique wxPython sous licence GNU GPL v3. Il est totalement portable et ne nécessite pas d'installation préalable mais s'exécute sur différentes plate-forme à partir des sources Python ou des fichiers binaires. DEVSimPy peut être téléchargé à l'adresse suivante : https://github.com/capocchi/DEVSimPy . Suivant la plate-forme d'accueil, il peut y avoir des particularités qui seront évoquées dans les sous-parties suivantes. Si l'utilisateur veut exécuter la dernière version en développement de DEVSimPy, il peut cloner cette version à partir du serveur git suivant : https://github.com/capocchi/DEVSimPy. Si l'utilisateur ne veut pas installer les dépendances nécessaires à DEVSimPy (Python, SciPy, NumPy, wxPython, …) il a la possibilité d'utiliser l'outil Portable Python.

WINDOWS

Sous Windows, il existe deux manières d'exécuter DEVSimPy : à partir des fichiers binaires ou à partir des fichiers sources.

L'utilisation de la version binaire ne nécessite l'installation d'aucune libraire supplémentaire. Tout est embarqué dans l'archive et l'utilisateur n'a qu'à exécuter le fichier *devsimpy.exe* par un double clique. Le fichier binaire à été généré à partir de l'outil GUI2EXE. L'exécution de DEVSimPy à partir des sources nécessite l'installation des librairies suivantes (si possible dans l'ordre) :

- Python en version 2.5+ (obligatoire) ;
- wxPython en version 2.6+ (obligatoire) ;
- Scipy et NumPy (optionnel) pour certaines librairies scientifiques ;
- Networkx , Profiler (optionnel) pour certains plugins.

Bien entendu à la date de rédaction de ce document, la dernière configuration de DEVSimPy utilise le couple python 2.7 et wxPython 2.8.12.1.

Une fois ces librairies installées et les sources téléchargées à partir du git, il suffit à l'utilisateur d'éditer le fichier *devsimpy.py* avec l'IDE Python et d'exécuter celui-ci en appuyant sur la touche F5 par exemple.

LINUX/UNIX

Sous Linux, l'exécution de DEVSimPy nécessite l'installation des paquets suivants (Ubuntu) :

- python en version 2.5+ ;
- python-wxversion ;
- python-wxgtk2.6 ou python-wxgtk2.8 ;
- python-scipy, python-matplotlib (optionnel) pour certaines librairies scientifiques ;
- python-networkx, python-profiler (optionnel) pour certains plugins.

Les utilisateurs des distributions basées sur Debian (comme Ubuntu) pourrons installer tout ces paquets par l'invocation dans une console de la commande *sudo apt-get install <nom du paquet>*.

Après l'installation de ces paquets et le téléchargement des sources, il suffit à l'utilisateur d'exécuter le fichier *devsimpy.py* ($> python devsimpy.py).

MACINTOSH

L'installation sous Mac OSX est identique à celle décrite dans le cas de la plate-forme Linux.

PROBLÈMES LIÉS À L'INSTALLATION

Il peut arriver que l'exécution de DEVSimPy empêche le rafraîchissement de l'interpréteur Python en arrière-plan. La solution consiste à exécuter le fichier *devsimpy.py* à partir d´un Bash Windows.

Il peut exister des avertissements au cours de l´utilisation de DEVSimPy liés à la couche graphique GTK. Si l´utilisateur veut éviter ces avertissements, il devra exécuter l´application en super utilisateur.

CONFIGURATION

Au cours de la première exécution de DEVSimPy, un fichier de configuration nommé *.devsimpy* est créé dans le répertoire personnel de l'utilisateur. Le chemin de ce répertoire dépend de la plate-forme qui supporte DEVSimPy :

- pour windows : C:\Users\<compte utilisateur>\AppData\Roaming ;
- pour Linux/Unix : /home/<compte utilisateur> ;
- pour Macintosh : /home/<compte utilisateur>.

Le fichier *.devsimpy* stocke des informations comme la version de DEVSimPy exécutée (champs version), la liste des chemins absolus des librairies externes autres que celles stockées dans le répertoire *Domain,* la liste des derniers fichiers récemment ouverts, la langue utilisée, la liste des plugins actifs, la liste des librairies à charger au démarrage, le profil graphique éventuellement défini par l'utilisateur.

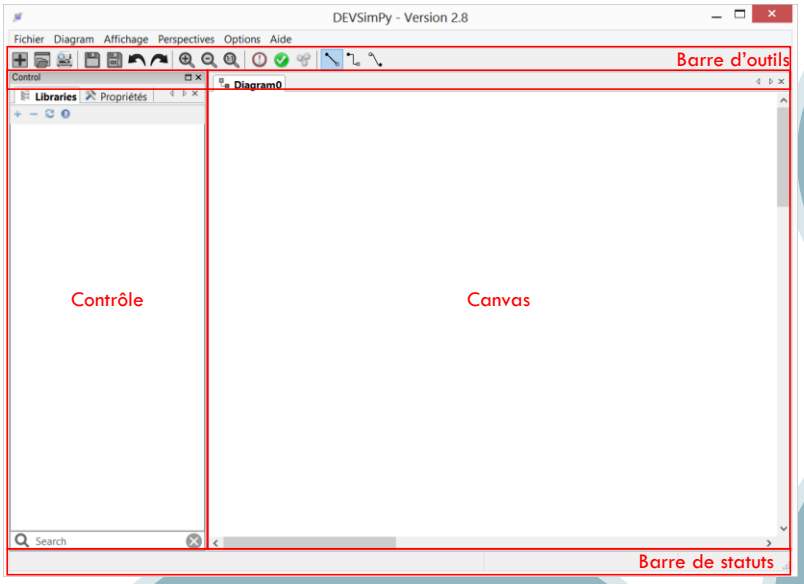
En cas de problème, il est possible de réinitialiser ce fichier soit en le supprimant, soit en spécifiant le paramètre -c (pour *clean*) dans la ligne de commande pour les plateformes Linux/Unix. Si le fichier *.devsimpy* n'est pas trouvé par DEVSimPy au démarrage, ce dernier le recréera immédiatement.

PREMIERS PAS AVEC DEVSIMPY

DÉMARRAGE

Une fois les dépendances logiciels nécessaires à l'installation de DEVSimPy (voir chapitre Installation et configuration), vous pouvez exécuter une instance du logiciel en interprétant le fichier *devsimpy.py*.

L'interface graphique de DEVSimPy à été pensée autour pour être modulaire. Chaque région de la fenêtre principale est dédiée à une fonctionnalité qui peut se détacher de l'interface pour être totalement autonome. La figure ci-dessous montre l'interface générale telle qu'elle apparaîtra à l'utilisateur au cours de la première exécution.



Canvas : c'est à l'intérieur de cette zone que l'on construit les diagrammes de modèles à simuler. Elle est construite sur le principe d'onglets que le l'on peut nommer et détacher. Par défaut, il existe un onglet, donc un diagramme, nommé *Diagram 0*.

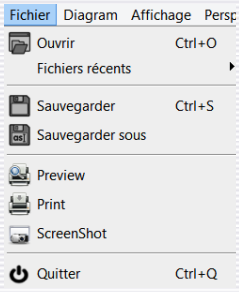
Contrôle : il est organisé sous forme d'onglets détachables dédiés à la gestion des librairies, la gestion des propriétés des modèles sélectionnés et la gestion de la simulation du diagramme courant (non visible par défaut). L'onglet *Library* permet d'organiser les librairies de modèles sous forme arborescente. L'onglet *Properties* permet d'éditer rapidement les propriétés des modèles sélectionnés dans la région *Canvas*.

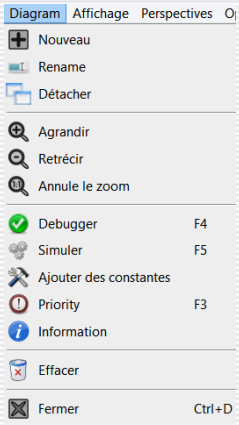
Barre d'outils : elle rassemble les raccourcis des principales actions comme :

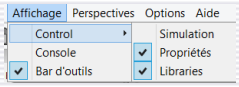
- l'ajout de diagrammes ;
- l'ouverture de diagrammes existant ;
- l'impression du diagramme courant ;
- la sauvegarde du diagramme courant ;
- la sauvegarde du diagramme courant sous un autre nom ;
- la restauration d'une action postérieure ;
- la restauration d'une action antérieure ;
- le zoom à l'intérieur du *canvas* sur le diagramme courant ;
- l'édition de la liste de priorité des modèles ;
- la simulation du diagramme courant ;
- le choix du type de connexion entre les modèles du diagramme courant.

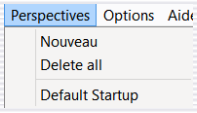
Barre de statuts : elle permet d'informer l'utilisateur sur les actions en cours comme :

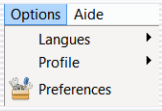
- La simulation;
- les copier/coller ;
- la modification de diagramme ;
- ...

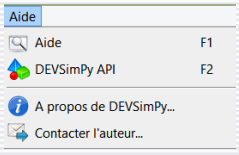












MENU PRINCIPAL

Parmi les fonctionnalités présentées ci-dessous certaines sont également accessibles par des raccourcis clavier. Le menu principal de DEVSimPy est organisé avec les sous-menus suivants :

Fichier :

- *Ouvrir : pour l'ouverture d'un diagramme ;*
- *Fichiers récents : propose les 5 derniers diagrammes ouverts ;*
- *Sauvegarde : pour sauvegarder un diagramme ;*
- *Sauvegarder sous : pour sauvegarder un diagramme sous un nouveau nom ;*
- *Preview, Print et ScreenShot : pour l'impression et la capture d'écran ;*
- *Quitter : pour quitter l'environnement DEVSimPy.*

Diagram : pour la gestion du « Canvas »

- *Nouveau : pour la création d'un nouveau diagramme ;*
- *Rename : pour renomer un diagramme ;*
- *Détacher : pour détacher le diagramme de DEVSimPy ;*
- *Agrandir, Rétrécir et Annuler Zoom : pour la gestion du zoom d'un diagramme ;*
- *Debugger, Simuler : pour déboguer et simuler un diagramme (le débogage est effectué automatiquement avant la simulation) ;*
- *Ajouter des constantes : pour ajouter des constantes au diagramme ;*
- *Priority : pour générer la priorité d'activation des modèles du diagramme ;*
- *Information : pour afficher des informations sur le diagramme ;*
- *Effacer : pour effacer le contenu d'un diagramme ;*
- *Fermer : pour fermer le diagramme (si il est modifié, la sauvegarde sera demandée).*

Affichage : pour l'activation/désactivation des panels :

- *Control : affiche le gestionnaire de librairies ;*
- *Librairie : affiche le gestionnaire des librairies (par défaut) ;*
- *Propriété : affiche le gestionnaire des propriétés des modèles (par défaut) ;*
- *Console : affiche un interpréteur Python (utile pour le débogage) ;*
- *Simulation : affiche le gestionnaire de simulation.*

Perspectives : pour la gestion des perspectives de travail. Ce sont des configurations particulières de votre espace de travail pouvant faire apparaître les panels de votre choix.

Options : pour la gestion de la langue utilisée pour l'interface, du « profiling » de la simulation et des préférences de DEVSimPy.

Aide : Pour obtenir l'aide en ligne ou envoyer un mail au développeur de DEVSimPy

CRÉATION D'UN MODÈLE ATOMIQUE

La création d'un nouveau modèle atomique DEVS dans DEVSimPy peut s'effectuer de deux manières différentes :

1. en utilisant le gestionnaire de création de modèles DEVS ;
2. en écrivant un fichier python compatible avec PyDEVS et en le glissant dans un *Canvas* de DEVSimPy.

Nous allons à présent décrire les deux procédures en détails au travers d'un exemple.

CRÉATION PAR LE GESTIONNAIRE DES MODÈLES

Le gestionnaire de modèles est invoqué par l'intermédiaire d'un clic droit sur le *Canvas* puis *Nouveau*.

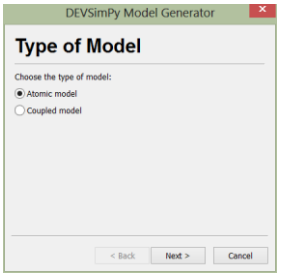


FIGURE 8 : TYPE DE MODÈLE.

La première étape consiste à choisir le type de modèle : atomique ou couplé. Ici nous voulons créer un modèle atomique (Figure 8). Nous choisissons donc le bouton radio *modèle atomique* (par défaut) puis nous continuons la création en appuyant sur le bouton *suivant*.

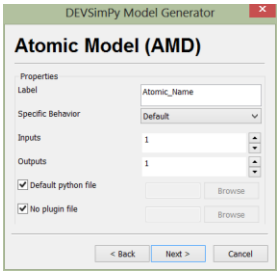


FIGURE 9 : INFORMATIONS.

La deuxième étape consiste à définir les propriétés graphiques et comportementales du modèle comme (Figure 9) :

1. son *label* (nom) : *Atomic_Name* (par défaut) ;
2. son comportement spécifique parmi : Generateur, Collecteur ou autre (par défaut) ;
3. son nombre de ports d'entrée (1 par défaut) ;
4. son nombre de ports de sortie (1 par défaut) ;
5. S'il faut implémenter un comportement (par défaut) ou si le comportement est déjà stocké dans un fichier python. Ce dernier cas est activé en décochant la check box. Le bouton de parcours des fichiers sur le disque est alors disponible à l'utilisateur qui peut aller chercher le fichier python désiré. Attention il faut que le fichier contienne un code python compatible avec DEVSimPy ;
6. Si le modèle n'a pas de plugin (par défaut) ou si il faut intégrer un plugin local déjà implémenté. La notion de plugin sera traitée plus tard dans le document mais à ce stade nous pouvons dire que DEVSimPy permet d'étendre les fonctionnalités de base des modèles au travers de la gestion des plugins locaux. Ce sont des fichiers python qui implémentent des fonctionnalités supplémentaires du modèle.

Enfin, nous avançons dans la configuration en appuyant sur le bouton *suivant*.

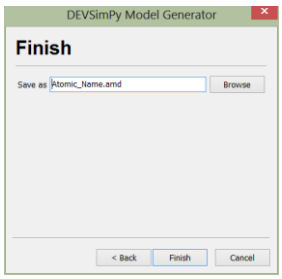
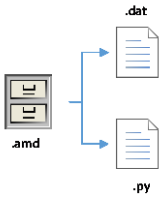


FIGURE 10 : FIN DE CRÉATION.

La dernière étape consiste à choisir le nom et l'emplacement du fichier qui va stocker les informations du modèle atomique (Figure 10).

Si le modèle à été créé avec l'interface DEVSimPy, vous devriez avoir un modèle atomique visible sur le *Canvas* (Figure 11). Vous savez que vous êtes en train de manipuler un modèle atomique au format *.amd* car une petit icone en forme de cube apparaît en haut à droite du modèle. Les ports d'entrée et de sortie sont notés par défaut in_i et out_j avec i le numéro du port d'entrée et j le numéro du port de sortie. Dans notre exemple, *in0* et *out0*.



Revenons sur le format du modèle atomique dont le fichier qui le représente possède l'extension *.amd*. Ce fichier est en fait un fichier compressé contenant (au minimum) deux fichiers (figure ci-contre): un fichier contenant les informations relatives à l'aspect graphique du modèle (au format *.dat*) et un fichier Python contenant la classe du modèle (au format *.py*). Nous remarquerons que le comportement est séparé de la vue.

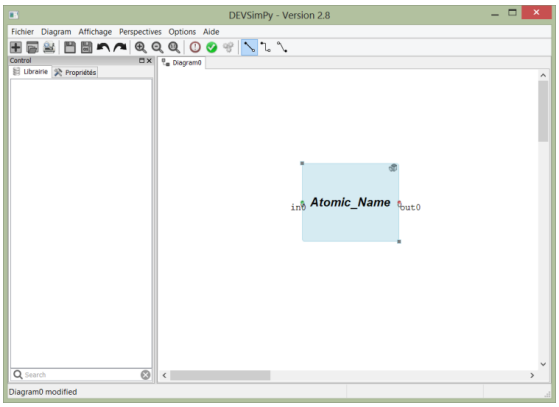


FIGURE 11 : NOUVEAU MODÈLE ATOMIQUE.

CRÉATION À PARTIR D'UN FICHIER PYTHON

Si l'utilisateur possède déjà des fichiers de classes PyDEVS correspondant à des modèles atomiques (hérités de la classe *AtomicDEVS*) il peut les réutiliser et les intégrer dans DEVSimPy. Pour cela il faut effectuer quelques adaptations :

1. Importer les classes mères adéquates en début de fichier :
   a. ***from*** *DomainInterface.DomainBehavior* ***import*** *DomainBehavior* : Cette ligne de code est indispensable car la classe du nouveau modèle atomique doit hériter de la classe mère *DomainBehavior* ;
   b. ***from*** *DomainInterface.Object* ***import*** *Message* : Cette classe est nécessaire si le modèle doit envoyer des événements sur ces ports de sortie (de type Générateur par exemple). Dans le cas d'un modèle atomique de type Collecteur, l'importation de cette classe n'est pas nécessaire.
2. Faire hériter la nouvelle classe de la classe mère *DomainBehavior* :
   class *Atomic_Name(DomainBehavior) ;*
3. Bien faire attention au nom des méthodes qui doivent respecter la notation imposée par l'API de PyDEVS :
   a. *extTransition* pour la fonction de transition externe ;
   b. *intTransition* pour la fonction de transition interne ;
   c. *outputFnc* pour la fonction de sortie ;
   d. *timeAdvance* pour la fonction d'avancement du temps.
4. DEVSimPy offre une autre méthode qui s'activera uniquement lorsque la simulation est terminée : la méthode *finish(self, msg)*.

Attention, lorsque le modèle atomique est crée à partir d'un fichier python, il faut que son nom soit identique à celui de la classe qu'il contient.

EXEMPLE

Voici un exemple du fichier Python *Atomic_Name.py* acceptable dans DEVSimPy pour le modèle atomique *Atomic_Name*.

Une fois le fichier python modifié, sa manipulation dans DEVSimPy est possible en le glissant et en le déposant dans un *Canvas*, soit à partir d'un explorateur de fichiers, soit à partir de la librairie DEVSimPy. Dans ce dernier cas, il faut que la librairie à laquelle appartient le fichier soit préalablement importée (voir le chapitre Manipulation des librairies, page 46).

```
# -*- coding: utf-8 -*-
from DomainInterface.DomainBehavior import DomainBehavior
from DomainInterface.Object import Message

class Atomic_Name(DomainBehavior):
''' DEVS Class for Atomic_Name model
'''
### Constructor
def __init__(self):
DomainBehavior.__init__(self)

### DEVS external transition function.
def extTransition(self): pass

### DEVS output function
def outputFnc(self): pass

        ### DEVS internal transition function
        def intTransition(self): pass

### DEVS time advance function
def timeAdvance(self): pass

### Additional function lunched just before the end of the simulation.
def finish(self, msg): pass
```

Il est possible d'obtenir un modèle atomique de type *.amd* à partir de ce type de fichier Python. En effet, après avoir été instancié dans un *Canvas*, un clique-droit sur le modèle donne accès à l'action *Exporter*. Cette action permet de transformer un modèle DEVSimPy issu d'un simple fichier python en un modèle atomique de type *.amd*. Ce nouveau modèle possède l'avantage de sauvegarder la configuration graphique que l'utilisateur aura défini à partir du modèle *.py*.

De la même manière, si l'utilisateur extrait le fichier python contenu dans l'archive au format *.amd*, il pourra instancier un modèle de type *.py* en faisant un glisse-déposer du fichier Python décompressé. Enfin, lorsqu'un modèle atomique hérite d'un autre modèle ou d'un module présent dans la librairie, l'importation peut se faire :

- lorsque le modèle est dans la librairie intégrée de DEVSimPy (sous-répertoires du répertoire *Domain* par défaut), par un *from <Domain>.<Nom du package>…<Nom des sous-package>…<Nom du module> import <Nom du module>*. Bien entendu, il faut que chaque package et sous-package contienne un fichier *__init__.py* avec la variable *__all__* correctement définie et que le nom du module (de la classe) possède le même nom que le fichier du module. DEVSimPy met en place un système d'importation automatique des dépendances. Il est également possible de ne pas spécifier *<Domain>* en début d'importation ;
- lorsque le modèle est dans un répertoire extérieur à l'architecture de DEVSimPy, il suffit de commencer par le nom du package de plus haut niveau. Les contraintes sont les mêmes que dans le premier cas, à part que *<Domain>* n'est plus optionnel.

PROPRIÉTÉS D'UN MODÈLE ATOMIQUE

Une fois le modèle atomique créé, l'utilisateur a la possibilité d'accéder à ses propriétés en utilisant l'onglet *Propriétés* du panneau de contrôle dans le cadre de gauche ou en cliquant droit sur le modèle (après l'avoir sélectionné) puis en cliquant sur l'action *Propriétés*.

La fenêtre de dialogue correspondant aux propriétés est présentée sur la Figure 12 avec :

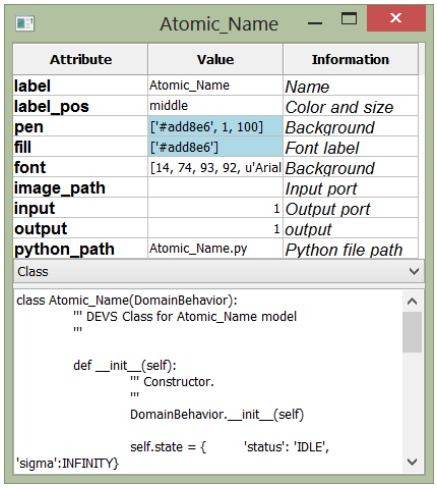


FIGURE 12 : PROPRIÉTÉS D'UN MODÈLE ATOMIQUE.

- **label** : le nom du modèle ;
- **label_pos** : la position du nom sur le modèle ;
- **pen** : la couleur, la taille et la transparence de la bordure ;
- **fill** : la couleur de fond ;
- **font** : la police du nom ;
- **image_path** : le chemin d'une image pouvant remplir le cadre du modèle ;
- **input** : nombre de port d'entrée ;
- **output** : nombre de port de sortie ;
- **python_path** : le chemin du fichier python qui contient la classe responsable du comportement du modèle ;
- **un menu déroulant** pour accéder au code en lecture seule de :
  - la classe (par défaut) ;
  - le constructeur ;
  - la fonction de transition interne ;
  - la fonction de transition externe ;
  - la fonction de sortie ;
  - a fonction d'avancement du temps ;
  - la fonction *finish*.

Nous allons à présent donner un comportement simple au modèle. Nous voulons que le modèle récupère les messages sur son unique port d'entrée et qu'il les transmette en sortie instantanément. Pour cela il nous faut accéder au code python du modèle pour le modifier.

> Il est possible d'accéder à la propriété *label* en appuyant simultanément sur la touche ctrl et le clique droit de la souris.

MODIFICATION DU CODE

Lorsque vous manipulez un modèle atomique crée à partir de DEVSimPy (*.amd*), vous accédez au code du modèle par un clique-droit sur celui-ci puis *Edition* suivi de *Model*. Nous reviendrons plus tard sur le lien *Tests*.

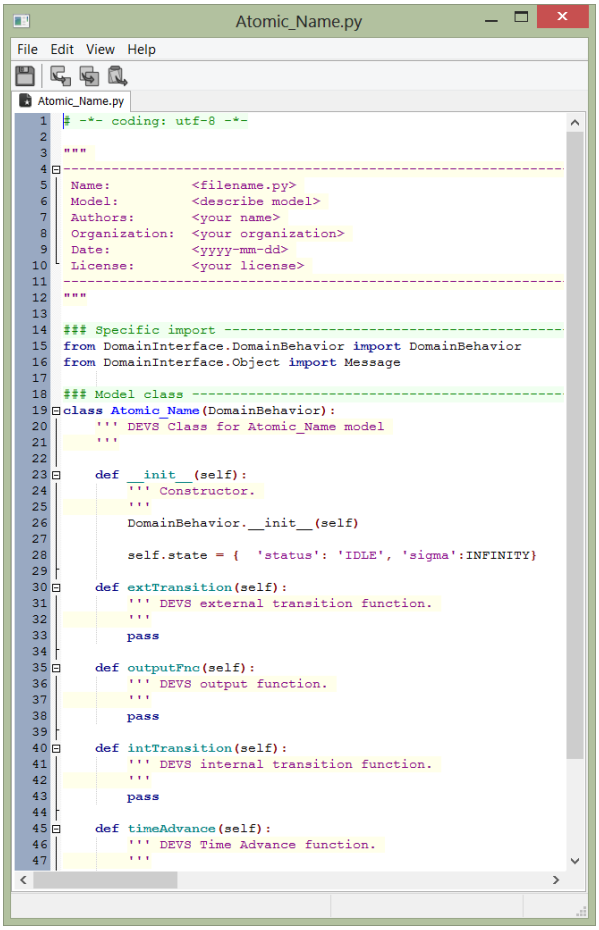


FIGURE 13 : ÉTIREUR DE CODE DANS DEVSIMPY.

L'éditeur de code (Figure 13) présente le code Python du modèle en commençant par des informations en commentaire mentionnant le nom du fichier, le nom du modèle, les auteurs du fichier, l'organisation à laquelle appartiennent les auteurs, la date de création et la licence. Ensuite, nous pouvons voir les importations nécessaires à la classe du modèle. La classe présente le constructeur (*__init__(self)*) avec un attribut particulier *self.state* qui permet de définir les variables d'états (ici le *status* et le *sigma*). Les fonctions de transition comme δ_{int} et δ_{ext} sont implémentées grâce aux méthodes *intTransition(self)* et *extTransition(self)*. La fonction de sortie λ est implémentée par la méthode *outputFnc(self)* et la fonction d'avancement du temps par la méthode *timeAdvance(self)*.

Le modélisateur devra donc implémenter la classe du modèle au travers de l'éditeur et pour ce faire, il a à disposition un ensemble de raccourcis lui facilitant la tâche. Par le biais du menu *Edit*, il est possible d'insérer des lignes de codes automatiquement pour :

- consulter un message sur un port d'entrée : *self.peek(<port d'entrée>) ;*
- envoyer un message sur un port de sortie : *self.poke(<port de sortie>, Message()) ;*
- définir un nouvel état : *self.state = {'status' :None, 'sigma' :INFINITY}.*

Toujours dans le même menu, l'utilisateur peut également ré-indenter le code. Bien entendu, le copier/coller de morceaux de code peut se faire au travers de l'utilisation du menu *Edit* ou par les raccourcis clavier *Ctrl+C* et *Ctrl+V*. De même le couper-coller se réalise par le raccourci *Ctrl+X* et *Ctrl+V*. Si l'utilisateur effectue un *Ctrl+D*, la ligne pointée par le curseur sera mise en commentaire. Pour dé-commenter une ligne, il suffit de se positionner dessus et de réaliser un *Ctrl+Shift+D*. Le menu *File* permet de sauvegarder le fichier ou de quitter l'éditeur. Le menu *View* permet de cacher la barre de statut en bas de la fenêtre d'édition.

Pendant l'implémentation, à chaque sauvegarde, une vérification syntaxique est réalisée par l'éditeur. Si une erreur syntaxique est présente, un message apparait dans la barre de statuts en bas de la fenêtre d'édition pour aider l'utilisateur dans le débogage. Le modèle ne sera sauvegardé que si le code ne contient aucune erreur. L'utilisateur peut forcer la sauvegarde du modèle mais le résultat n'est pas garanti. Il ne faut pas oublier qu'il est toujours possible d'accéder au fichier python édité en décompressant le fichier *.amd* en cas de problème.

EXEMPLE

Si nous reprenons l'exemple du modèle atomique présenté dans la section Les modèles atomique (page 9), nous pouvons implémenter le modèle atomique *EXEC* (Figure 14).

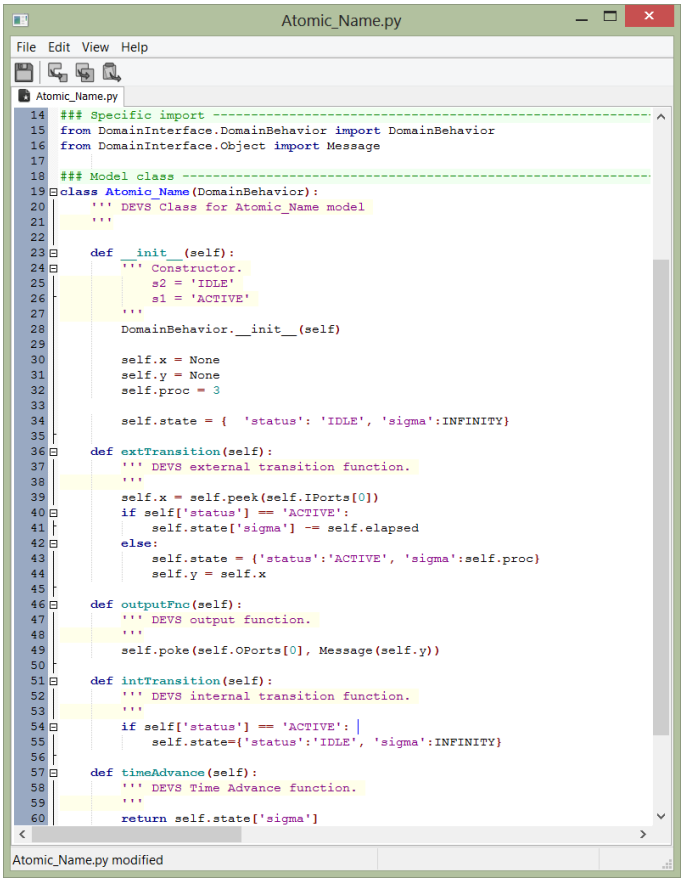


FIGURE 14 : CODE PYTHON DU MODÈLE ATOMIQUE EXEC.

Le constructeur présente quatre attributs :

- *self.x* et *self.y* pour stocker le message d'entrée et le message de sortie ;
- *self.proc* pour stocker le temps de traitement du modèle ;
- *self.state* pour stoquer les variables d'états comme le *status* initialement 'IDLE' et *sigma* initialement à l'infini pour rendre le modèle inactif et en attente d'un message.

La fonction de transition externe commence par stocker le message d'entrée grâce à un *self.peek* (ligne 39) sur l'unique port du modèle (*self.IPorts[0]*). Ensuite, l'état change en fonction de l'ancien état. Lorsque le modèle réceptionne un message alors qu'il a déjà reçu un message, la variable de sortie *self.x* n'est pas mise à jour. Seul le temps de vie de l'état est mis à jour grâce au temps écoulé depuis le dernier changement d'état : *self.elapsed*.

La fonction de sortie se contente d'envoyer le message de sortie sur l'unique port de sorite *self.OPorts[0]* grâce à la méthode *self.poke* (ligne 49).

La fonction de transition interne change l'état du modèle en passant de 'ACTIVE' à 'IDLE' avec une durée de vie infinie.

La fonction d'avancement du temps, comme dans beaucoup de modèles implémentés dans DEVSimPy se contente de renvoyer la variable d'état *sigma* (en accord avec les implémentations présentées dans l'ouvrage du Professeur B.P. Zeigler [2]).

SIMULATION D'UN MODÈLE ATOMIQUE

Dans DEVSimPy, le gestionnaire de simulation est invocable en cliquant sur l'icône de la barre d'outils (Figure 15).

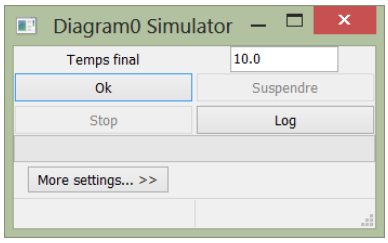


FIGURE 15 : DIALOGUE DE SIMULATION.

Le gestionnaire de simulation propose un champ d'acquisition permettant de renseigner le temps final de simulation (10 par défaut). Il n'existe pas d'unité prédéfinie et l'utilisateur donnera l'interprétation qu'il le désire à la valeur de ce temps de simulation en fonction du comportement des modèles qu'il veut simuler. La simulation démarre après avoir cliqué sur le bouton *Ok*. Une barre de progression montrera l'avancée de la simulation avec un compteur affiché en bas à droite de la fenêtre. Le bouton *Suspendre* permet de suspendre la simulation. Une particularité de DEVSimPy est qu'il permet de modifier les modèles pendant que leur simulation est suspendue. Nous reviendrons plus tard sur cette fonctionnalité qui fait l'originalité de DEVSimPy est qui est principalement due à l'utilisation du langage Python. La suspension de la simulation est levée en recliquant sur le bouton *Ok*. Un clique sur le bouton *Stop*, arrête la simulation définitivement. Le bouton *Log* permet de tracer la simulation. Le bouton *More settings* permet d'accéder à d'autres fonctionnalités que nous présenterons plus tard.

Avant de faire apparaitre la fenêtre du gestionnaire de simulation, une vérification de l'instanciation des modèles est effectuée. Cependant, il est possible d'effectuer cette tâche en cliquant sur l'icône de la barre d'outils

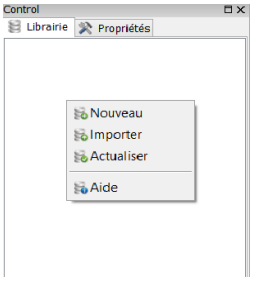


FIGURE 16 : CLIQUE-DROIT SUR LE PANEL LIBRAIRIES.

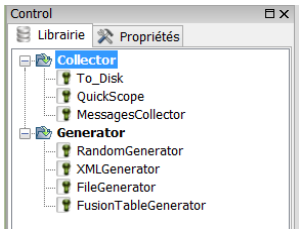


FIGURE 17 : LIBRAIRIES *COLLECTOR* ET *GENERATOR*.

EXEMPLE

Afin de simuler le comportement du modèle atomique *EXEC*, il faut connecter son port d'entrée à un modèle atomique *générateur* d'évènements et son port de sortie à un modèle atomique *collecteur* d'évènements. Pour cela nous allons utiliser des modèles existants dans des librairies de DEVSimPy. En effet, DEVSimPy possède deux librairies de modèles disponibles :

- **Generator** : est une librairie de modèles atomiques de type générateur d'évènements. Parmi ses modèles, le modèle *RandomGenerator* génère des événements portant des valeurs réelles aléatoires. Nous allons utiliser celui-ci pour simuler le modèle atomique *EXEC* ;
- **Collector** : est une librairie de modèles atomiques de type collecteur d'évènements. Elle est composée du modèle *MessageCollector* qui permet de stocker les messages reçus pendant la simulation et de les présenter dans un tableau après un double-clique sur le modèle.

Nous reviendrons plus en détail dans la suite de ce document sur la gestion de ces librairies. Afin d'accéder aux deux modèles atomiques supplémentaires nécessaires pour simuler le modèle *EXEC*, il suffit de faire un clique-droit sur le panel *Control* avec le panel *Librairies* activée puis de choisir *Importer* dans le menu contextuel (Figure 16). Ensuite, c'est en cochant les deux cases pour la librairie *Generator* et *Collector* (Figure 17) puis en appuyant sur le bouton *Ok* que l'on verra apparaître les deux librairies sélectionnées dans le panel *Librairie* (Figure 18). Après avoir déroulé les deux librairies, les deux modèles sont instanciables par un glisser-déposer sur le *Canvas*. Les deux modèles ont été renommés en *RandGen* et *MsgCol* par l'intermédiaire du panneau des propriétés accessible soit par un double-clique sur le modèle soit par l'intermédiaire du panel *Propriétés* dans le *Control*.

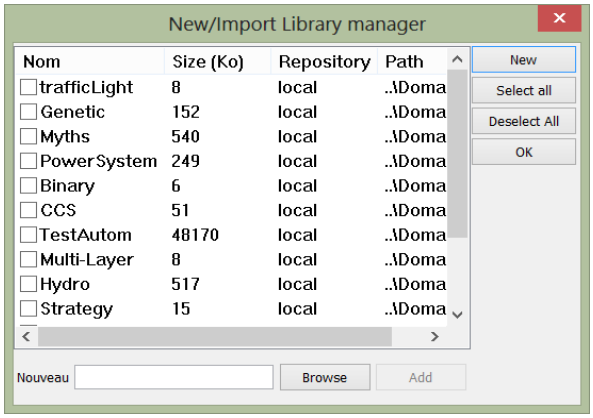


FIGURE 18 : FENÊTRE DE GESTION DES LIBRAIRIES.

L'interconnexion des modèles se fait en rejoignant les ports de sortie vers les ports d'entrée de chaque modèle. L'utilisateur doit presser le bouton de la souris dès la sélection d'un port d'entrée, et rester appuyé jusqu'à obtenir une petite croix sur un port de sortie. Lorsque celle-ci apparaît, le bouton droit de la souris est relâché pour rendre effective la connexion des deux ports. Si ces conditions ne sont pas réunies, la connexion n'a pas lieu. La procédure est identique en partant des ports de sortie pour rejoindre les ports d'entrée. Il est possible également de connecter les modèles en accédant au gestionnaire de connexion accessible par un clique droit sur un modèle puis « connecté à ». Une liste déroulante s'affiche pour proposer le modèle à connecter. Au clique sur le nom du modèle choisi, un gestionnaire de connexion s'affiche pour connecter les ports des deux modèles concernés.

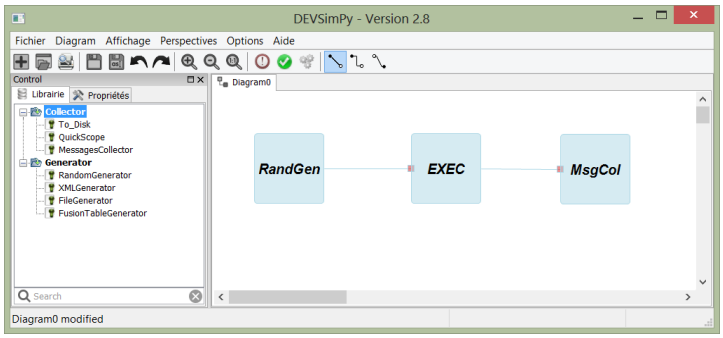


FIGURE 19 : IMPORTATION DES LIBRAIRIES COLLECTOR ET GENERATOR DANS DEVSIMPY.

Si les modèles à connecter ne possèdent qu'un port (entrée ou sortie), le gestionnaire n'est pas proposé et la connexion des deux ports s'effectue instantanément.

Les trois modèles interconnectés sur le *Canvas* (Figure 19) sont à présent simulable.

A la fin de la simulation, un double clique sur le modèle *MsgCol* fait apparaître un tableau avec deux lignes (Figure 20). Une ligne montrant un message arrivé au temps 3 avec la valeur 2 et une ligne suivante pour un message arrivant au temps 7 avec la valeur 5. Le modèle *EXEC* à donc envoyé au modèle *MsgCol* les messages aléatoires qu'il a reçu au temps 0 et 4 après 3 unités de temps (*self.proc=3*). Le modèle *MsgCol* à donc bien reçu un message au temps 3 (0+3) et 7 (4+3). Il n'aura pas reçu de message au temps 8+3=11 puis que le temps de simulation est de 10. Tous les messages reçus entre les temps 0 et 3, 4 et 8 et 8 et 10 ne sont pas considérés.

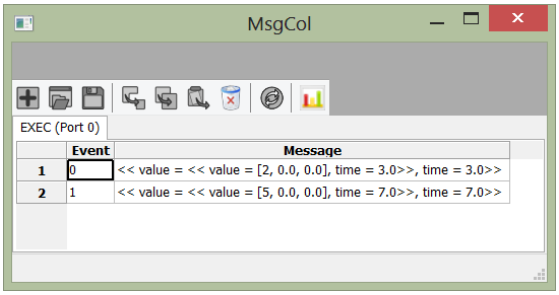


FIGURE 20 : FENÊTRE DE RÉSULTATS DU MODÈLE MSGCOL.

Vous pourrez sauvegarder votre configuration dans un diagramme en faisant un *Ctrl+S* ou en invoquant l'action *Sauvegarder-sous* dans le menu *Fichier*. Nous reviendrons plus tard sur la notion de diagramme.

Il arrive souvent que la simulation révèle des erreurs d'implémentation ou de syntaxe dans les méthodes d'un modèle atomique. Si, vous deviez déboguer votre modèle, il existe une méthode de débogage pour les modèles atomiques. En invoquant la méthode *self.debugger(self, msg)* dans n'importe quelle autre méthode du modèle atomique, il est possible ensuite d'afficher le *log* de cette fonction en cliquant droit sur un modèle atomique puis *Log*. Dans l'exemple précédent, si nous voulons connaître la valeur des messages à

chaque invocation de la méthode *extTransition* (fonction de transition externe DEVS), il nous suffit d'insérer après le *self.peek*, l'instruction *self.debugger(self.x)*.

Il existe un autre outil permettant le débogage dans DEVSimPy grâce à l'utilisation d'un plugin général nommé *Blink*. Ce plugin permet de dérouler la simulation pas à pas en affichant chaque appel des fonctions de transition et de sortie des modèles. De plus, la simulation peut être suivie de manière visuelle, car chaque modèle s'illumine à chaque fois qu'il est invoqué. Pour activer ce plugin, il suffit de se rendre dans le menu *Options* puis *Préférence* et *Plugins*. Nous reviendrons plus tard sur la notion de plugins généraux dans DEVSimPy.

Bien entendu, si vous insérez des *print* dans le code du modèle atomique, vous pouvez également déboguer votre code mais cette solution a tendance à polluer et rendre illisible votre interpréteur Python dès que le nombre de modèles devient important. Avec le plugin *Blink* il est possible de conserver les traces de la simulation et même de faire des recherches de mot-clé. Cette dernière fonctionnalité est très appréciée lorsque la simulation génère un nombre important d'information.

SAUVEGARDE D'UN MODÈLE ATOMIQUE

Une modèle atomique instancié à partir d'un fichier *.amd* est exportable. Lorsque l'utilisateur sauvegarde le code d'un modèle atomique *.amd* modifié, il met à jour automatiquement le modèle dans la librairie. Cependant, si l'utilisateur veut exporter le modèle il n'aura qu'à cliquer droit sur celui-ci puis se rendre dans le menu *Exporter*. Il pourra alors faire une copie du modèle en le renommant. Cette fonctionnalité permet par exemple d'enregistrer plusieurs versions d'un même modèle (même comportement enregistré dans un fichier Python embarqué dans le *.amd*) avec une interface graphique différente (*.dat* embarqué dans le nouveau *.amd* différent de l'original). Grâce à l'exportation, il est donc possible de créer de nouvelles librairies ou de compléter des librairies existantes.

Bien entendu, DEVSimPy n'implémente aucune procédure particulière pour l'exportation des modèles atomiques de type *.py* du fait que ce ne sont que des fichiers Python. Par conséquent, il suffit de les sauvegarder avec un autre nom pour en faire de nouveaux modèles. Par contre il n'est pas possible de sauvegarder l'aspect graphique de ces modèles. Toute copie d'un modèle atomique de type *.py* adopte un aspect graphique par défaut présentant un port d'entrée et un port de sortie à chaque fois qu'il est instancié (glisser-déposer dans le *Canvas* à partir du panel *Librairies* ou à partir de l'extérieur de DEVSimPy). À la différence d'un modèle atomique de type *.amd*, le modèle atomique de type *.py* n'enregistre aucune information concernant son aspect graphique. Il faut donc adapter cet aspect à chaque contexte dans lequel est instancié le modèle atomique *.py*.

Sauvegarder un modèle atomique de type *.amd*, c'est avant tout l'exporter mais c'est également sauvegarder le contexte dans lequel il est employé ou instancié. En d'autres termes, c'est aussi sauvegarder le diagramme dans lequel il est présent. Pour sauvegarder un diagramme, il suffit d'invoquer le menu *Fichier* puis *Sauvegarder* ou *Sauvegarder-sous* si il a déjà été sauvegardé. Le diagramme est alors enregistrable avec le format *.dsp* (pour DevSimPy) ou le format d'échange *.xml*. Le format *.dsp* est un fichier compressé contenant un fichier binaire (obtenu par un *CPickle*). Ce format est préféré au format *.xml* car il est plus rapide à restaurer cependant, il est difficilement lisible.

EXEMPLE

L'exemple de nos modèles interconnectés *RandGen*, *EXEC* et *MsgCol* peut être sauvegardé dans un diagramme que l'on nommera *exemple_quick_start.dsp*. Pour ce faire, il faut invoquer le menu *Fichier* puis l'action *Sauvegarder*. Une fenêtre de dialogue apparaît pour renseigner un nom et un emplacement pour sauvegarder notre diagramme.

FONCTIONNALITÉS GÉNÉRALES

Dans ce chapitre, le lecteur pourra trouver une description détaillée des fonctionnalités générales de DEVSimPy. Nous traiterons par exemple de la création des modèles et de leur sauvegarde dans des librairies, de l'utilisation des plugins et de la configuration du logiciel.

CRÉATION, IMPORTATION ET SAUVEGARDE DES MODÈLES

Du fait que DEVSimPy constitue une couche graphique améliorée du logiciel PyDEVS, l'utilisateur peut créer des modèles de deux façons différentes : par importation de fichiers PyDEVS ou par la création de modèles propres à DEVSimPy. Cette distinction repose sur un concept central dans DEVSimPy : **la séparation explicite entre le comportement et la représentation graphique des modèles**.

CRÉATION DES MODÈLES PAR IMPORTATION DES FICHIERS PYDEVS

Lorsque l'utilisateur choisit d'implémenter le comportement de ces modèles PyDEVS dans son logiciel de programmation il a ensuite la possibilité d'importer ceux-ci dans DEVSimPy pour mieux les manipuler. Pour ce faire il suffit de :

1. mettre à jours les importations du fichier :
    a. ***from*** *DomainInterface.DomainBehavior* ***import*** *DomainBehavior* pour un modèle atomique et ***from*** *DomainInterface.DomainStructure* ***import*** *DomainStructure* pour un modèle couplé. Cette ligne de code est indispensable car la classe du nouveau modèle atomique doit hériter de la classe mère *DomainBehavior* ou *DomainStructure* et non plus de la classe *AtomicDEVS* ou *CoupledDEVS* ;
    b. ***from*** *DomainInterface.Object* ***import*** *Message* : Cette classe est nécessaire si le modèle doit envoyer des événements sur ces ports de sortie (de type *Générateur* par exemple). Dans le cas d'un modèle atomique de type *Collecteur*, l'importation de cette classe n'est pas nécessaire.
2. faire hériter la nouvelle classe de la classe mère *DomainBehavior* pour un modèle atomique ***class*** *Atomic_Name(DomainBehavior)* et *DomainStructure* pour un modèle couplé ***class*** *Coupled_Name(DomainStructure)* ;
3. supprimer les routines responsables de la création des ports (entrée et sortie). Ces derniers sont générés automatiquement à partir des ports spécifiés de manière graphique.

Ensuite, il suffit de glisser-déposer les fichiers *.py* sur le *Canvas* de DEVSimPy à partir :

- d'un explorateur de fichiers classique ou ;
- du panel de contrôle *Librairie* si les fichiers ont été préalablement importés. Attention, pour que les fichiers *.py* apparaissent dans le panel de librairie après importation dans DEVSimPy, il faut que le fichier *__init__.py* contienne la variable *__all__* affectée à un tableau contenant le nom des fichiers implémentant des modèles qui héritent de la classe *DomainBehavior* pour un modèle atomique et *DomainStructure* pour un modèle couplé. Dans DEVSimPy, tout package qui ne contient pas de fichier *__init__.py* ne sera pas importé. Le fichier *__init__.py* permet de connaitre le nom des modules à importer dans la librairie de DEVSimPy. Pour plus d'information sur les fichiers *__init__.py* voir le site internet : http://docs.python.org/2/tutorial/modules.html.

L'utilisateur dispose ensuite d'une représentation graphique automatique de ces modèles avec un port d'entrée, un port de sortie et une couleur par défaut. L'utilisateur peut alors modifier

les propriétés des modèles et les connecter entre eux pour former un diagramme qu'il pourra sauvegarder ou simuler. Si l'utilisateur veut sauvegarder le modèle avec ses caractéristiques graphiques, il doit alors l'exporter dans un modèle DEVSimPy (*.amd* ou *.cmd*) par un clique-droit puis l'action *Exporter*.

La classe *Message* est importée à partir du module *DomainInterface.Object*. Cependant, si l'utilisateur veut implémenter sa propre classe *Message*, il suffit d'implémenter un module *Object.py* dans le répertoire contenant la classe important le module. Ensuite, il ne faut pas oublier d'ajouter dans le fichier *__init__.py* la liste python *__all__ = ['Object']* pour faciliter l'écriture de l'importation du module *Object* contenant la classe *Message*.

EXEMPLE

L'exemple montre une comparaison entre le code d'une classe écrite dans PyDEVS implémentant un générateur de Job avec la même classe pour DEVSimPy. Les lignes 6, 7, 17, 24 et 36 sont volontairement mises en gras dans la version DEVSimPy pour montrer les différences. La ligne 27 a été supprimée pour bien montrer qu'elle n'est plus nécessaire dans la version de DEVSimPy. La classe *DomainBehavior* n'ayant pas de paramètres dans son constructeur, le nom du générateur a été passé en tant qu'attribut de la classe *Generator*.

```
# -*- coding: Latin-1 -*-
# queueModels.py --- simple queueing system examples
#                     October 2002
#                    Jean-Sebastien Bolduc
#                     Hans Vangheluwe
#                  McGill University (MontrÃ©al)
#                 --------------------------------
import sys
import os.path
# Import code for model representation:
import pydevs
from pydevs.devs_exceptions import *
from pydevs.infinity import *
from pydevs.DEVS import *

class Generator(AtomicDEVS):
    """ Generates jobs.
    """

    # all arguments should be strictly positive integers
    def __init__(self, ia, ib, sa, sb, name=None):

        # Always call parent class' constructor FIRST:
        AtomicDEVS.__init__(self, name)

        # make a local copy of the constructor's arguments
        self.ia = ia
        self.ib = ib
        self.sa = sa
        self.sb = sb

        # add one output port
        self.OUT = self.addOutPort(name="OUT")
        self.elapsed = arbitrary_number

    def intTransition(self):
        self.state.first = False
        return self.state

    def outputFnc(self):
        job = Job(self.sa, self.sb)
        self.poke(self.OUT, job)

    def timeAdvance(self):
        if self.state.first == True: return 0
        else: return uniform(self.ia, self.ib)

class Job:
    IDCounter = 0   # class variable, globally unique

    def __init__(self, szl, szh):
        Job.IDCounter += 1
        self.ID = Job.IDCounter
        self.size = uniform(szl, szh)

    def   str   (self): return "(job %d, size %f)" % (self.ID, self.size)
```

```
# -*- coding: utf-8 -*-
# queueModels.py
#  DEVSimPy version
import sys
import os.path
from DomainInterface.DomainBehavior import DomainBehavior
from DomainInterface.Object import Message

class Generator(DomainBehavior):
    """ Generates jobs.
    """

    # all arguments should be strictly positive integers
    def __init__(self, ia, ib, sa, sb, name=None):

        # Always call parent class' constructor FIRST:
        DomainBehavior.__init__(self)

        # make a local copy of the constructor's arguments
        self.ia = ia
        self.ib = ib
        self.sa = sa
        self.sb = sb
        self.name = name

        # add one output port
        ~~self.OUT = self.addOutPort(name="OUT")~~
        self.elapsed = arbitrary_number

    def intTransition(self):
        self.state.first = False
        return self.state

    def outputFnc(self):
        job = Job(self.sa, self.sb)
        self.poke(self.OUT, Message(job))

    def timeAdvance(self):
        if self.state.first == True: return 0
        else: return uniform(self.ia, self.ib)

class Job:
    IDCounter = 0   # class variable, globally unique

    def __init__(self, szl, szh):
        Job.IDCounter += 1
        self.ID = Job.IDCounter
        self.size = uniform(szl, szh)

    def __str__(self): return "(job %d, size %f)" % (self.ID, self.size)
```

CRÉATION DES MODÈLES DEVSIMPY

L'utilisateur a la possibilité de construire ses modèles en utilisant l'interface graphique de DEVSimPy. Il lui suffit alors d'invoquer le gestionnaire de création des modèles par un clique-droit sur le *Canvas* qui va venir accueillir le nouveau modèle.

L'utilisateur se laisse guider par le gestionnaire en renseignant les champs indispensables à la création du modèle. Quatre types de modèles peuvent être créés :

- les modèles atomiques (au format *.amd*) ;
- les modèles couplés (au format *.cmd*) ;
- les ports d'entrée ;
- les ports de sortie.

Les ports d'entrée et de sortie ne seront proposés que lorsque le gestionnaire aura été invoqué à partir d'un *Canvas* appartenant à un modèle couplé. Lorsque les modèles sont créés, ils apparaissent directement dans le *Canvas* sur lequel a eu lieu le clique-droit qui a fait apparaître le gestionnaire de création des modèles.

CRÉATION D'UN MODÈLE ATOMIQUE DEVSIMPY

La création d'un modèle atomique dans DEVSimPy a déjà été traité dans la partie « Création d'un modèle atomique » (page 21) du chapitre « Premiers pas avec DEVSimPy » (page 19). Le gestionnaire de création des modèles DEVSimPy permet de renseigner les informations nécessaires à la création d'un modèle atomique comme : le nom, le nombre de ports d'entrée, de sortie, le fichier Python responsable du comportement (ou de la structure) du modèle, etc. Les autres propriétés comme les couleurs et les polices utilisées pour dessiner les modèles pourront être modifiée après la création en invoquant la fenêtre de propriété.

CRÉATION D'UN MODÈLE COUPLÉ

La procédure de création d'un modèle couplé par l'intermédiaire du gestionnaire de création de modèle est composée des mêmes étapes que pour la création d'un modèle atomique. La différence principale est que le modèle couplé ne possède pas de comportement et donc il ne sera pas nécessaire (à priori) de modifier le fichier Python embarqué dans l'archive *.cmd*.

CRÉATION D'UN PORT D'ENTRÉE OU DE SORTIE

La procédure de création des modèles de type port (entrée ou sortie) ne concerne que les modèles couplés. En effet, lorsque l'utilisateur double-clique sur un modèle couplé, il accède au contenu du modèle au travers d'une fenêtre externe à DEVSimPy. Lorsque l'utilisateur veut ajouter un port d'entrée sur un modèle couplé il doit à la fois ajouter un port d'entrée dans les propriétés du modèle atomique et il doit également ajouter un modèle de port d'entrée dans le modèle couplé (par l'intermédiaire du gestionnaire de création des modèles).

La modification du nombre de ports (entrée ou sortie) d'un modèle (atomique ou couplé) peut être effectuée en attendant quelque seconde sur le modèle après l'avoir sélectionné. Une fenêtre de dialogue contextuelle apparaitra pour donner la possibilité à l'utilisateur d'incrémenter ou de décrémenter le nombre de port d'entrée ou de sortie.

La Figure 22 montre les deux nouveaux boutons radio contextuels permettant d'accéder à la création de nouveaux modèles de ports.

Si nous choisissons la création d'un port d'entrée, la fenêtre de la Figure 23 apparaîtra pour continuer la création en renseignant :

- **le nom du port d'entrée** : il peut être choisi sans contrainte particulière à contrario du numéro d'identifiant ;

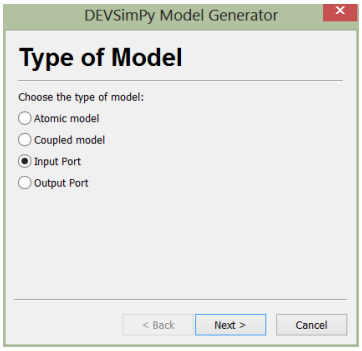


FIGURE 22 : CRÉATION D'UN PORT D'ENTRÉE.

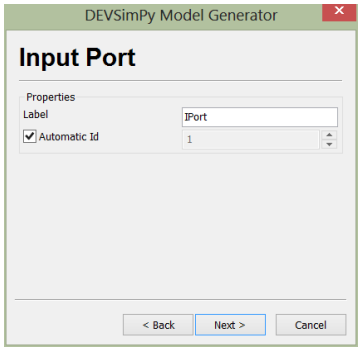


FIGURE 23 : INFORMATIONS SUR LE PORT D'ENTRÉE.

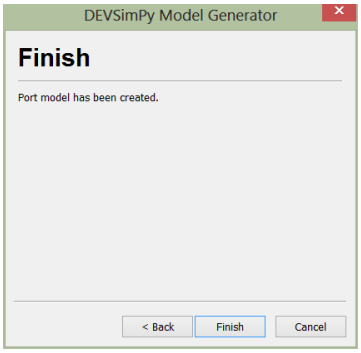


FIGURE 24 : ÈTAPE FINALE DE CRÉATION D'UN PORT D'ENTRÉE.

- **le numéro d'identifiant du port** : il est auto-incrémenté. Cependant, il peut arriver que cette auto-incrémentation ne soit pas mise à jour correctement. Dans ce cas, la case à cocher peut être désélectionnée et le numéro d'identifiant peut être configuré de manière manuelle. **Attention, chaque numéro d'identifiant doit être unique et doit correspondre à la place du port en partant du haut vers le bas. Si deux ports ont le même numéro d'identifiant ils sont considérés comme un seul et même port.**

Lorsque ces informations sont correctement renseignées, la fenêtre de la Figure 24 confirme la création du port d'entrée. À chaque étape, l'utilisateur peut revenir en arrière pour modifier les informations relatives à chacune.

SAUVEGARDE DES MODÈLES DEVSIMPY

Si le code Python des modèles *.amd* ou *.cmd* est modifié, il n'y a aucune sauvegarde à effectuer. Par contre, si les autres propriétés sont modifiées (comme le nombre de ports d'entrée ou de sortie et les propriétés graphiques) et que l'utilisateur veut conserver ces nouvelles configurations dans un nouveau modèle, il devra exporter le modèle modifié. Pour ce faire, un clique-droit sur le modèle sélectionné fera apparaître un menu contextuel contenant l'action *Exporter*.

Les modèles couplés peuvent être exportés au format *.cmd*, *.json* ou *.xml*. Par contre les modèles atomiques ne sont exportables qu'au format *.amd*. Les ports ne sont pas exportables puisqu'ils sont embarqués dans les modèles couplés.

Si le modèle est déjà sauvegardé dans une librairie importée (visible dans le panel *Librairies*) et qu'il est sauvegardé après modification, la mise à jour de la librairie est automatique.

CRÉATION, UTILISATION ET SAUVEGARDE DES DIAGRAMMES

Les diagrammes sont définis comme un ensemble de modèles atomiques ou couplés (*.amd*, *.cmd ou .py*) interconnectés. Ils se construisent en instanciant et en connectant des modèles DEVSimPy sur un *Canvas*.

CRÉATION DES DIAGRAMMES

Dans DEVSimPy les diagrammes vierges sont créés en cliquant sur la première icône de la barre d'outils ou par le menu *Diagram* puis *New*. Un raccourci permet d'obtenir le même effet : *Ctrl+N*. Ensuite, l'utilisateur construit le contenu du diagramme en : (i) déposant par glisser-déposer des modèles contenus dans des librairies importées sur le *Canvas* ou (ii) en créant des modèles par l'intermédiaire du gestionnaire de création de modèles DEVSimPy décrit dans la partie « Création des modèles DEVSimPy » de la page 34.

Les modèles couplés sont des diagrammes par définitions. Par conséquent, lorsque l'utilisateur double-clique sur un modèle couplé, son contenu apparait comme un diagramme dans une fenêtre externe. Toutes les manipulations possibles sur les diagrammes sont applicables notamment par l'intermédiaire de la barre d'outils.

DÉFINITION DES CONSTANTES DE DIAGRAMME

Il est possible de définir des constantes rattachées à un diagramme. Ces constantes sont destinées à être utilisées par les modèles du diagramme. Pour ce faire, il suffit d'un clique-droit sur le *Canvas* puis invoquer l'action *Définir des constantes*. La Figure 25 montre la fenêtre de dialogue permettant de définir ces constantes comme la constante A=1. Le bouton

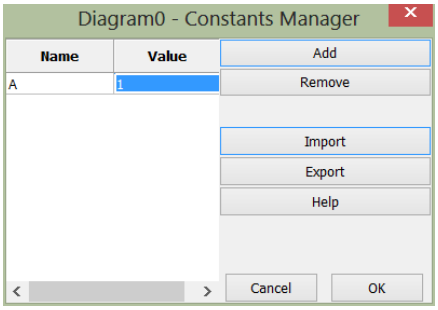


FIGURE 25 : FENÊTRE DE DÉFINITION DES CONSTANTES.

Add/Remove permet d'ajouter/supprimer des lignes au tableau des constantes (dans la partie gauche). Le bouton *Import/Export* permet d'importer/d'exporter des constantes à partir d'un fichier au format *.csv*. Le bouton *Help* permet d'accéder à une aide à la définition des constantes d'un diagramme.

Les constantes sont utilisables dans les paramètres des modèles *.amd* ou *.cmd* notamment dans la définition des expressions de formule (pour le modèle *NLFunction* de la sous-librairie *Continuous* de la librairie *PowerSystem*).

Le référencement aux constantes s'écrit de la manière suivante : *'Nom_du_diagramme'['Nom_de_la_constante']*. Attention, la définition des constantes n'est pas sauvegardée dans les diagrammes. Il faudra donc importer les constantes à chaque ouverture des modèles *.dsp* avant de les simuler.

SAUVEGARDE DES DIAGRAMMES

Lorsque le diagramme est construit, il est possible de le sauvegarder dans un fichier au format *.dsp*. Il faut invoquer le menu *Fichier* puis l'action *Sauvegarder* (*Ctrl+S*).

Enfin l'utilisateur peut ouvrir un diagramme existant soit en glissant un fichier *.dsp* à partir d'un explorateur de fichiers soit par le menu *Fichier* puis *Ouvrir*.

Il est également possible de retrouver les derniers fichiers ouverts par le menu *Fichier* puis l'action *Fichiers récents*. Par défaut le nombre de fichiers récents est fixé à 5. Cependant ce chiffre est un paramètre configurable dans le menu *Option/Préférences/General*.

Le fichier *.dsp* est une archive contenant un fichier binaire au format *.dat* généré avec une sérialisation du type *CPickle*. Il est possible de sauvegarder un diagramme au format *JSON*. Ce format permet de générer des fichiers lisibles par l'utilisateur (à contrario du fichier binaire) et donc de simplifier le débogage éventuel d'un diagramme. Par contre, son ouverture est un peu plus lente que le fichier binaire.

SIMULATION DES DIAGRAMMES

Une fois que le diagramme est construit, il est possible de le simuler. La simulation d'un diagramme consiste à créer une instance d'un simulateur PyDEVS décrit dans le chapitre L'API PyDEVS (page 16) à partir de ce diagramme. Il existe deux manières de lancer une simulation :

- en cliquant sur l'icône ;
- en rendant visible le panel *Simulation* dans le panel de Contrôle.

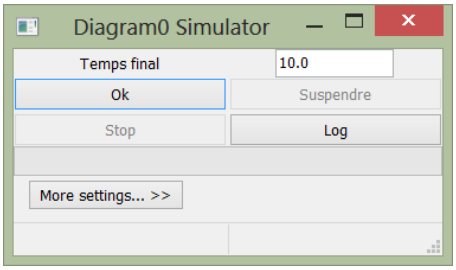


FIGURE 26 : FENÊTRE DE SIMULATION.

Dans le premier cas, la fenêtre présentée en Figure 26 apparaît, dans le second cas, le panel de la Figure 27 apparaît.

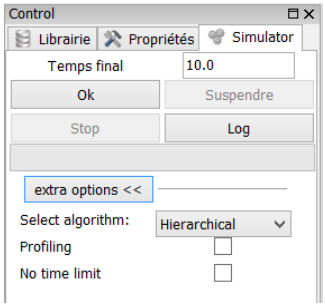


FIGURE 27 : PANEL DE SIMULATION.

Dans les deux cas, il est nécessaire de définir le temps final de simulation qui doit être supérieur à 0 (une vérification est effectuée avant l'instanciation du simulateur). Ensuite la simulation est exécutée en appuyant sur le bouton *Ok*. Le bouton *Suspend* permet de suspendre la simulation. Pendant que la simulation est suspendue, il est possible de modifier le comportement des modèles atomiques ou de modifier les paramètres de leur constructeur afin d'influencer les résultats de simulation. Le bouton *Stop* permet de mettre fin de manière définitive à la simulation en cours. Pendant la simulation, une barre de progression montre l'avancée de celle-ci et une information sur le temps écoulé est disponible dans la barre d'information en fond de fenêtre. À la fin de la simulation, le bouton *Log* permet d'accéder à une trace de la simulation dans le cas où le plugin *Verbose* est activé (nous reviendrons plus tard sur la notion de plugin général). Les options supplémentaires permettent de choisir :

- l'algorithme de simulation : hiérarchique (classique DEVS), à couplage directe (Direct-coupling DEVS) ;
- l'activation du profilage de la simulation afin de faire du débogage. Cette option va générer un fichier *.prof* de profilage visualisable avec des outils spécifiques. Lorsque cette option est activée, la simulation est plus lente et à la fin de celle-ci, le menu *Options/Profile* est accessible ;
- le conditionnement de la fin de la simulation non pas avec le temps final mais plutôt en fonction de « l'activité » des modèles par rapport au contenu de la liste des modèles imminents (voir le chapitre Gestions des options, page 37).

Avant d'instancier le simulateur PyDEVS, DEVSimPy instancie chaque modèle (atomique ou couplé) afin de s'assurer de la validité syntaxique de leur constructeur. Si cette vérification est validée, le simulateur est instancié. Il est possible d'effectuer cette vérification sans pour autant vouloir simuler le diagramme. Dans ce cas, l'utilisateur peut cliquer sur le bouton

Commentaire [c1]: Faire apparaitre la fenêtre de gestion des erreurs et en parler

GESTION DES PERSPECTIVES

Les perspectives ont été définies pour répondre à un besoin de configuration personnalisé de l'environnement graphique. L'utilisateur peut créer une nouvelle perspective par l'intermédiaire du menu *Perspectives* suivi de *New*. Ensuite, un nom de perspectives est demandé. Une liste des perspectives sera accessible par le même menu. Il est possible de supprimer des perspectives définies par l'utilisateur mais la perspective par défaut ne peut être supprimée.

GESTIONS DES OPTIONS

Le menu des options donne accès au sous-menu des langues et des préférences définies dans DEVSimPy.

L'interface de DEVSimPy est proposée en deux langues : Le français et l'anglais. Au démarrage, DEVSimPy adaptera la langue de son interface en fonction de la langue utilisée par le système d'exploitation. Cependant, l'utilisateur peut changer celle-ci par le menu *Options* puis *Langues*. Pour que la nouvelle interface prenne effet, un redémarrage est demandé à l'utilisateur. La proposition d'un logiciel en plusieurs langues nécessite un suivi permanent de la traduction des nouveaux termes. Il se peut que certains termes français ne soient pas traduit en anglais par faute de temps.

Le menu *Outils/Préférences* permet d'accéder au gestionnaire de préférences (Figure 28) proposant quatre sous-menus :

- **volet *Général*** (Figure 28): ce menu concerne les préférences générales comme le chemin absolu du répertoire des librairies, le chemin absolu du répertoire des plugins, le nom du répertoire qui stocke les éventuels fichiers de sortie de certains modèles atomiques, le nombre de fichiers récemment ouverts, la taille de l'historique des actions réalisées dans DEVSimPy, la taille de la police de l'interface, l'activation de la transparence des fenêtres. Cette dernière option dépend de l'OS qui accueille DEVSimPy. Si l'OS le permet, lorsque une fenêtre d'un modèle couplé est déplacée, elle devient transparente pour visualiser ce quelle cache. Cette fonctionnalité peut s'avérer être utile lorsque plusieurs fenêtres de modèles couplés sont ouvertes ;

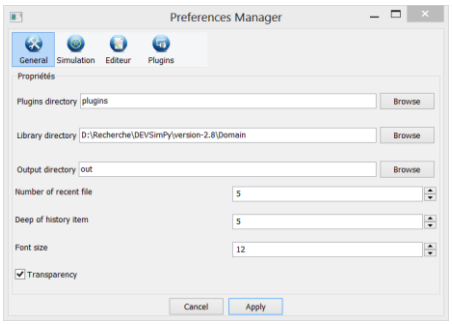


FIGURE 28 : PRÉFÉRENCES GÉNÉRALES.

- **volet *Simulation*** (Figure 29) : Les préférences de simulation concernent : les notifications sonores qui retentisse à la fin de chaque simulation (*Finish.wav*) et lorsqu'une erreur apparaît dans DEVSimPy (*Error.wav*). Le menu propose aussi de configurer la stratégie employée par défaut pour la simulation :
  - la première stratégie (*bag-based*) de simulation applique l'algorithme hiérarchique du classique DEVS basée sur le simulateur abstrait composé de coordinateur et de simulateur. Cependant, des améliorations on été apportées afin de pouvoir par exemple coupler plusieurs modèles sur un seul port (*bag-based*) ;
  - la deuxième stratégie de simulation (*direct-coupling*) applique l'algorithme du couplage direct. L'arbre de simulation ne possède plus de coordinateur. Le *root* dirige la simulation en organisant les exécutions des simulateurs dans le temps. Cette stratégie est nommée à couplage direct car les couplages sont présents uniquement entre les modèles atomiques. Coté modélisation, les modèles couplés peuvent être utilisés pour structurer les modèles de manière hiérarchique mais avec la deuxième stratégie de simulation, le simulateur abstrait est instancié en évitant les connexions entre modèles couplés .

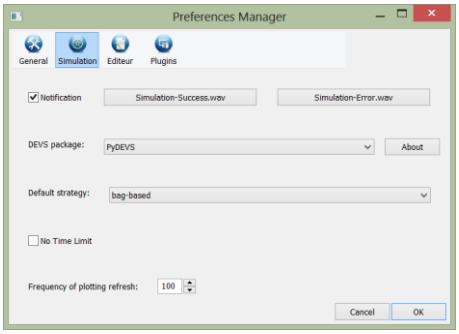


FIGURE 29 : PRÉFÉRENCE DE SIMULATION.

Les préférences de simulation proposent également d'activer de manière permanente la gestion de la fin des simulations en fonction de l'activité des modèles. La simulation se terminera lorsque la liste des modèles imminents du *root* est vide (plus aucune « activité » de simulation est prévue). Si la case à cocher est activée, cette fonctionnalité prendra effet pour chaque simulation.

La librairie *Collector* propose un modèle atomique nommé *QuickScope*. Ce modèle permet de tracer de manière dynamique (pendant la simulation) des variables réelles portées par des

événements (souvent le cas pour la librairie *PowerSystem*). Il est possible de configurer la fréquence de rafraîchissement du tracé grâce au dernier menu de la fenêtre des préférences de simulation.

- **volet *Éditeur*** (Figure 30) : La seule configuration possible pour l'éditeur du code concerne le choix d'utiliser un éditeur externe ou non. Si la case à cocher est activée, l'éditeur de code local de DEVSimPy sera utilisé par défaut à chaque fois que le code d'un modèle de type *.py* sera édité par l'utilisateur. Si cette case n'est pas cochée, un pop-up sera proposé à l'utilisateur pour qu'il choisisse d'utiliser ou non l'éditeur local. Dans le cas des modèles de type *.amd* ou *.cmd*, il n'y a qu'une seule alternative, c'est l'éditeur local qui est proposé. Cela est dû au fait que ce genre de modèles sont des archives et que seul l'éditeur local de DEVSimPy est capable de mettre à jour le fichier Python contenu dans l'archive ;

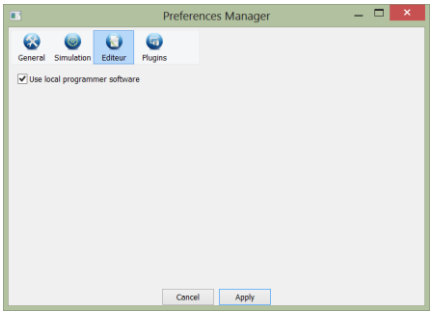


FIGURE 30 : PRÉFÉRENCES D'ÉDITION.

- **volet *Plugins*** (Figure 31) : Le panneau des préférences pour les plugins généraux permet de gérer les plugins présents dans le répertoire *plugins* de DEVSimPy (si celui-ci n'a pas été changé dans les préférences générales). Lorsque un plugin est activé-désactivé, une petite croix verte/noire est affichée en face. Lorsque le plugin présente des erreurs, une icône d'alerte est positionnée en face du nom du plugin. Les couleurs et les formes des icônes sont définies par la couche graphique de l'OS sur lequel DEVSimPy est interprété. Les figures présentées dans ce document présente DEVSimPy sur une plate-forme Windows 8 avec un thème particulier. Lorsqu'un plugin est sélectionné, une documentation le concernant est affichée dans la partie basse du panneau. Il est possible par l'intermédiaire des boutons de gauche de : sélectionner/désélectionner tout les plugins, ajouter/supprimer un plugin, rafraîchir la liste des plugins. Pour la surpression, l'utilisateur aura le choix de supprimer le plugin de la liste ou de supprimer définitivement le plugin du répertoire *plugins*. Lorsqu'un plugin est sélectionné et si le plugin le propose, il est possible de le configurer. Cependant, il faut que le développeur du plugin prévoit une fonction de configuration (nous verrons plus tard le développement de plugins généraux dans DEVSimPy).

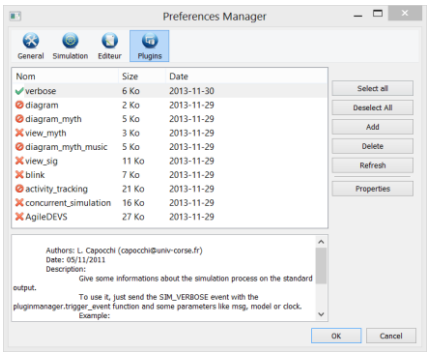


FIGURE 31 : PRÉFÉRENCES DES PLUGINS.

Les modifications des préférences seront appliquées lorsque l'utilisateur appuiera sur le bouton *Ok*. Si le bouton *Annuler* est invoqué, aucune modification ne sera appliquée.

> *Les plugins généraux sont des plugins qui agissent et étendent des fonctionnalités de DEVSimPy. Ils peuvent agir sur des diagrammes ou sur le processus de simulation. Les plugins locaux sont des plugins qui sont attachés aux modèles DEVSimPy (généralement des modèles .amd) et qui étendent le comportement de ces modèles.*

FONCTIONNALITÉS AVANCÉES

Cette section présente quelques fonctionnalités avancées de DEVSimPy.

MANIPULATION DES MODÈLES

Les actions disponibles sur les modèles permettent d'agir sur leur comportement et sur leur apparence. Le menu contextuel des actions possibles sur un modèle (*.amd* ou *.cmd*) est invoqué par un clique-droit sur le modèle. Les actions sont séparées en groupe : *Edition*, *Utilisation*, *Connexion*, *Exportation*, *Suppression* et *Propriétés*.

La Figure 32 montre l'apparition du menu contextuel des actions possibles sur un modèle atomique nommé *QuickScope*.

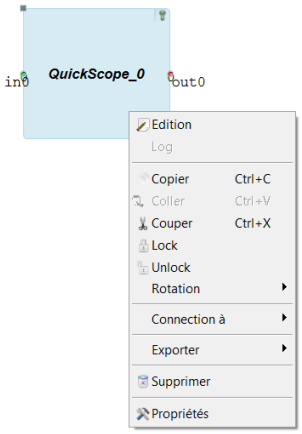


FIGURE 32 : CLIQUE-DROIT SUR LE MODÈLE ATOMIQUE.

EDITION DU CODE

L'édition du code python est possible par ce sous-menu. Il est demandé à l'utilisateur si il veut éditer le code du modèle avec l'éditeur local (embarqué dans DEVSimPy) ou sil veut utiliser un logiciel externe à DEVSimPy (Figure 33). Cette question n'est présente que parce que le modèle du *QuickScope* est de type *.py* et que l'utilisateur n'a pas coché l'option d'édition dans le menu des préférences (voir le chapitre « Gestions des options », page 37).

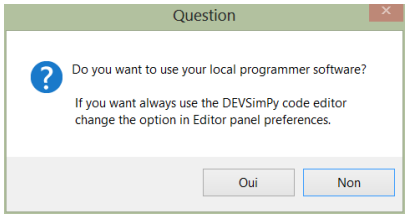


FIGURE 33 : DIALOGUE DU CHOIX DE L'ÉDITEUR.

La fenêtre d'édition du code permet de modifier celui-ci et d'enregistrer les modifications automatiquement. La coloration syntaxique et la numérotation des lignes facilite la lecture dans le code. De plus, chaque enregistrement implique auparavant une vérification syntaxique du code. En cas d'erreur, une information sera donnée dans la barre de statut informant l'utilisateur du type d'erreur et de la ligne concernée par celle-ci. Attention, l'enregistrement ne se fera que si toutes les erreurs sont réparées. Il est possible de forcer la sauvegarde mais l'issu de la simulation du modèle n'est pas garantie.

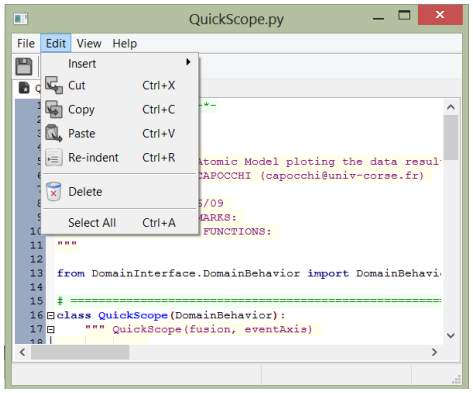


FIGURE 34 : MENU D'ÉDITION DU CODE PYTHON.

La barre de menu de l'éditeur de code donne accès à plusieurs fonctionnalités comme :

- **Fichier** : l'utilisateur pourra enregistrer le code en cliquant sur le sous-menu *Enregistrer* ou quitter l'application en cliquant sur *Quitter*. La sauvegarde peut se faire aussi par le raccourci clavier classique *Ctrl+S* ;
- **Edition** : sur la Figure 34, le menu d'édition permet d'effectuer les actions classiques comme le copier, le couper/coller, le coller, tout sélectionner. De plus, il est possible de ré-indenter le code. Le langage Python étant un langage interprété, il est obligatoire d'indenter le code sous peine d'être pénalisé d'erreurs syntaxiques. Python préconise l'indentation en utilisant des espaces. Cependant les tabulations sont également possibles bien que déconseillées. Le mélange d'indentation par tabulation et par espace est interdit par le langage Python. Donc il peut arriver que ce genre de situation se présente et l'utilisateur peut invoquer le sous menu *Re_indentation* pour déléguer la ré-indentation à l'éditeur de code DEVSimPy. Cette fonctionnalité est rendue possible grâce au script *reindent.py* de Tim Peters. Le sous-menu *Insertion* permet lui aussi de bénéficier de fonctionnalités automatiques comme la ré-indentation (Figure 35). En effet ce sous menu est composé des actions : *nouveau peek*,

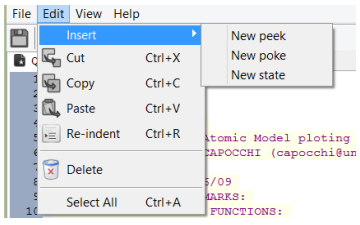


FIGURE 35 : MENU D'INSERTION DU CODE PYTHON.

nouveau poke, *nouvel état*. Ces fonctionnalités permettent d'insérer le code approprié pour respectivement acquérir un nouveau message sur un port d'entrée, envoyer un message sur un port de sortie et définir un nouvel état du modèle (voir le chapitre Modification du code, page 25). Enfin, il est possible de mettre en commentaire une ligne de code par le raccourcis clavier *Ctrl+D* et dé-commenter celle-ci par un *Shift+Ctrl+D*.

- **vue** : cette action permet de cacher la barre de statut en fond de fenêtre ;
- **aide** : cette action permet d'invoquer une aide pour l'utilisation de l'éditeur de code.

DÉBOGAGE DU CODE

Le débogage des modèles est une tâche importante dans le processus d'implémentation. Il est primordial de posséder au sein de DEVSimPy des fonctionnalités facilitant cette tâche. Lorsque l'utilisateur veut debugger le code d'un modèle, il peut utiliser des instructions *print* insérées directement dans le code de manière provisoire. Cependant, cette pratique, certes simple et rapide, reste lourde à gérer et à maintenir lorsque l'on veut pérenniser son code de débogage. DEVSimPy offre la possibilité d'utiliser une méthode spécifique self.*debugger(msg)* à la place d'un simple *print*. Lorsqu'elle est utilisée dans le code d'un modèle, elle va permettre de sauvegarder dans un fichier de log le temps d'appel et un message *msg* qui lui sera passé en paramètre. Pendant ou à la fin de la simulation, l'utilisateur aura accès au contenu de ce fichier par le biais du sous-menu *Log*. Ce sous-menu n'est activé que si l'utilisateur utilise la méthode self.*debugger(msg)* dans le code du modèle.

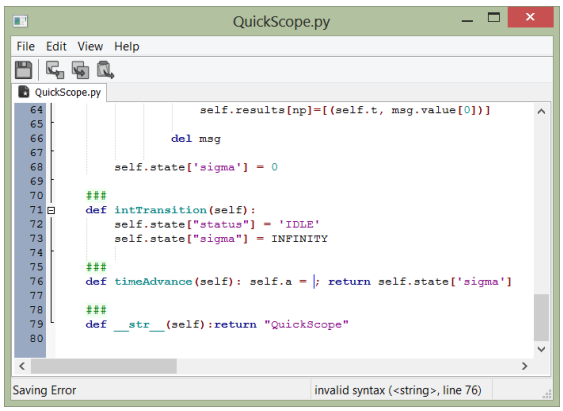


FIGURE 36 : EXEMPLE D'ERREUR DANS L'ÉDITEUR DE CODE.

Comme nous l'avons précisé dans la section précédente, l'utilisateur est guidé lorsqu'une erreur apparait dans le modèle au cours de la sauvegarde de son code modifié. Comme il est montré sur la Figure 36, une information apparait dans la barre de statut informant l'utilisateur sur le type d'erreur (invalid syntax) et sur la ligne qui contient l'erreur (line 76). Cependant, si la sauvegarde est forcée alors que le modèle possède une erreur, celle-ci réapparaitra au cours de la simulation et un gestionnaire de débogage sera alors proposé à l'utilisateur. Ce gestionnaire de débogage demandera à l'utilisateur si il veut être dirigé à la ligne concernée par l'erreur dans l'éditeur de code local.

Commentaire [c2]: Mettre une capture d'ecran de ce genre de gestionnaire.

Il existe un autre niveau de débogage lorsque l'utilisateur débute une simulation. En effet avant l'instanciation du simulateur abstrait, une vérification des constructeurs des modèles présents dans le diagramme est réalisée. Cette étape anticipe les éventuelles erreurs relevées pendant l'instanciation du simulateur abstrait. Si les modèles ont été codés par l'utilisateur qui veut les simuler et que celui-ci n'a pas forcé la sauvegarde de codes erronés, cette étape de débogage avant la simulation est toujours validée. Cependant, lorsque les modèles sont issus d'une librairie qui n'a pas été développée par l'utilisateur et que cette librairie contient des modèles erronés, l'étape de débogage en avant la simulation est avantageuse et permet un gain de temps car elle permet de mettre en évidence des erreurs de codage avant d'exécuter le simulateur.

Commentaire [c3]: Faire une captuer d'écran de ce type de gestionnaire

FONCTIONNALITÉS CLASSIQUES

DEVSimPy permet la manipulation classique des modèles comme : le copier-coller, le couper-coller. De plus, il est possible de figer un modèle sur un *Canvas* afin d'éviter de déplacer celui-ci malencontreusement. Lorsqu'un modèle est figé, un petit cadenas apparait en haut à gauche du modèle pour en informer l'utilisateur.

L'utilisateur peut également effectuer des rotations sur les modèles. Cette fonctionnalité s'avère utilise pour organiser les connexions entre les modèles.

CONNEXION

La connexion entre deux ports de nature différente (entrée vers sortie ou sortie vers entrée) est traditionnellement réalisée en faisant glisser, tout en restant appuyé sur le clique-droit de la souris, le curseur de la souris d'un port à l'autre. Lorsque le curseur prend la forme d'une croix à l'approche du port de destination, il suffit de relâcher le clique-droit de la souris pour rendre effective la connexion. Celle-ci est représentée par un trait plein. Si l'utilisateur clique droit sur une connexion, il peut la supprimer ou la figer. La suppression peut également être effectuée en pressant la touche du clavier *suppr*. Il est possible de changer la forme de la connexion en sélectionnant le motif dans la barre d'outils de DEVSimPy. Par défaut la connexion est rectiligne mais elle peut être de forme carré ou diagonale .

Il existe une autre manière d'effectuer une connexion entre deux ports grâce à un gestionnaire de connexion invoqué par le sous-menu contextuel du modèle. La Figure 37 présente ce gestionnaire avec la connexion du port 0 du modèle *Coupled_Name* et le port d'entrée 0 du modèle *Hidden_1*. Lorsque l'utilisateur appuie sur le bouton *Connection*, la connexion se réalise sur le *Canvas*. De la même manière, l'utilisateur peut supprimer la connexion en appuyant sur le bouton *Deconnection*. De plus, il est possible de connecter tous les ports d'entrée du modèle *Hidden_1* à l'unique port de sortie du modèle couplé *Coupled_Name*. Ce gestionnaire de connexion est avantageux lorsque plusieurs connexions sont à réaliser entre deux modèles (dans ce cas, le glisser-lâcher peut se révéler rébarbatif).

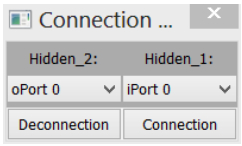


FIGURE 37 : DIALOGUE DE CONNECXION.

EXPORTATION

Pour accéder à la fonctionnalité d'exportation, il suffit de sélectionner le menu *Exporter*. Cependant, chaque modèle possède son type d'exportation :

- un modèle atomique/couplé de type *.py* peut être exporté en un modèle au format *.amd*/*.cmd*. L'avantage de ce type d'exportation est de transformer les modèles de type *.py* (généralement issus d'une adaptation des modèles de PyDEVS) en des modèles plus complets de type *.amd*. Avec un modèle de type *.amd*, l'utilisateur peut sauvegarder les propriétés graphiques du modèle et il peut aussi ajouter des plugins locaux ;
- un modèle atomique/couplé de type *.amd*/*.cmd* peut être exporté au même format. Cette procédure est utile lorsque l'on veut mettre à jour un modèle dans une librairie. L'exportation provoque l'apparition d'une fenêtre de dialogue pour renseigner le nom et le chemin du nouveau modèle.

Commentaire [c4]: Vérifier que dans DEVSimPy si un modèle couplé de type .py veut s'exporter on présente un format .cmd et non pas .amd….

PLUGINS

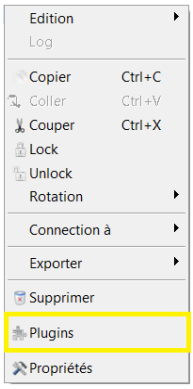


FIGURE 38 : MENU CONTEXTUEL POUR LES PLUGINS.

Lorsque le clique-droit est effectué sur un modèle de type *.amd* ou *.cmd* et que celui-ci possède un plugin local, un autre menu vient s'ajouter à la liste : *Plugins* (Figure 38). Ce menu va permettre à l'utilisateur d'interagir avec les plugins locaux du modèle. Lorsque le menu est cliqué, le fichier *plugins.py* contenu dans l'archive (*.amd* ou *.cmd*) est importé. Si il existe des erreurs d'importation dans le ce fichier, un pop-up informe l'utilisateur (Figure 39). Si le module et bien importé, la liste des éléments (fonctions, méthodes) contenue dans le plugin et chargée.

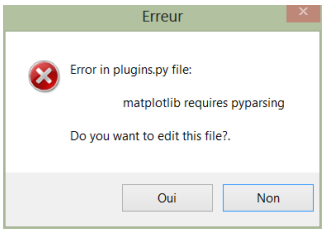


FIGURE 39 : DIALOGUE D'ERREUR DANS LE PLUGIN.

Sur la Figure 40, nous pouvons voir que le plugin contient une fonction *OnleftDClick* qui est surchargée. Elle est précédée d'une croix bleue qui signifie que la fonction n'est pas activée. Pour l'activer, il suffit de cliquer sur la croix. Si une croix rouge apparait, il existe des erreurs syntaxiques et l'utilisateur peut les corriger en cliquant sur le bouton *Editer* à gauche. Si l'utilisateur sélectionne la fonction et que son docstring Python existe, une documentation apparaîtra. Dans notre exemple, il n'existe pas de documentation pour *OnLeftDClick*. Cette fonction permet d'effectuer une action particulière au double-clique sur le modèle. En effet, le comportement par défaut d'un double-clique sur un modèle atomique dans DEVSimPy est d'ouvrir la fenêtre de dialogue de ses propriétés (sauf pour certains modèles atomiques comme le *QuikScope* qui ouvre une fenêtre présentant le tracé de variables). Pour un modèle couplé, un double-clique provoque l'apparition de la fenêtre présentant le contenu du modèle couplé. Ce plugin local permet donc d'étendre la fonctionnalité du double_clique sur le modèle.

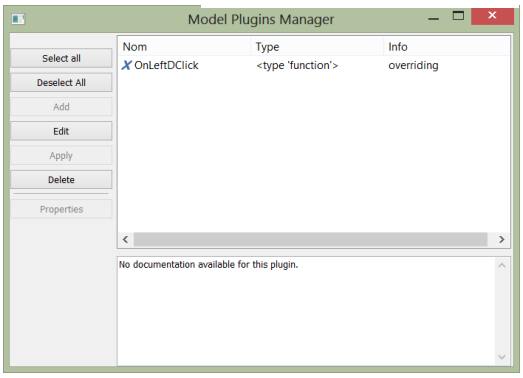


FIGURE 40 : GESTIONNAIRE DE PLUGINS.

Par l'intermédiaire des boutons à gauche de la fenêtre du gestionnaire de plugins locaux, l'utilisateur peut sélectionner/désélectionner les plugins, ajouter des plugins, éditer le code des plugins, appliquer les modifications, supprimer les plugins et éditer les propriétés d'un plugin.

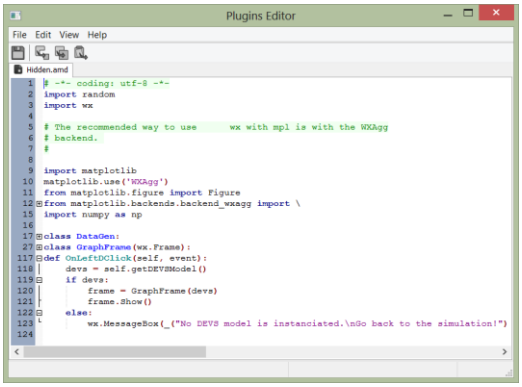


FIGURE 41 : EDITION DU CODE PYTHON D'UN PLUGIN.

Nous avons présenté la surcharge d'une fonctionnalité graphique (le double-clique) au travers d'un plugin local. Un développeur Python et un utilisateur confirmé de DEVSimPy (connaissant l'architecture logiciel de DEVSimPy) peut donc surcharger n'importe quelle méthode graphique d'un modèle.

La fenêtre de la Figure 41 présente l'édition de la fonction *OnLeftDClick*. Nous voyons que le fichier plugins implémente deux classes *GraphFrame* et *DataGen* qui sont ensuite utilisées pour implémenter *OnLeftDClick*. La classe *GraphFrame* hérite de la classe

wx.Frame et générera donc une fenêtre lorsqu'elle sera instanciée dans la fonction *OnLeftDClick*. On remarquera que l'instance du modèle DEVS est obtenu grâce à la méthode *self.getDEVSModel()* de l'instance du modèle *.amd*.

Nous remarquons également que seule la fonction est proposée en plugin et que les deux classes n'apparaissent pas dans la liste des plugins locaux. Si l'utilisateur veut surcharger des classes, il vaut mieux qu'il implémente un plugin général.

Enfin, le fichier *plugins.py* présenté ci-contre comprend plusieurs importations de modules comme *matplotlib* et *numpy*. Si ces deux modules ne sont pas installés, un pop-up apparaîtra à l'utilisateur.

SUPPRESSION

Grâce à ce menu, il est possible de supprimer le modèle du diagramme.

PROPRIÉTÉS

Ce menu permet d'accéder à la fenêtre de dialogue qui présente les propriétés du modèle. Cette fenêtre est aussi visible dans le panel *Propriétés* lorsque le modèle est sélectionné et que ce panel est rendu visible. Les propriétés sont séparées en deux catégories : *graphiques* et *comportementales*.

La présentation des propriétés se fait en trois colonnes : les attributs, les valeurs des attributs et les informations des attributs. Les attributs graphiques sont les suivants :

- **label** : nom du modèle apparaissant dans le diagramme ;
- **label_pos** : la position du label sur le model (haut, milieu ou bas) ;
- **pen** : la couleur et la taille des contours du modèle ;
- **fill** : la couleur de remplissage du modèle ;
- **image_path** : l'image optionnelle de remplissage du modèle ;
- **input** : le nombre de ports d'entrées ;
- **output** : le nombre de ports de sorties.

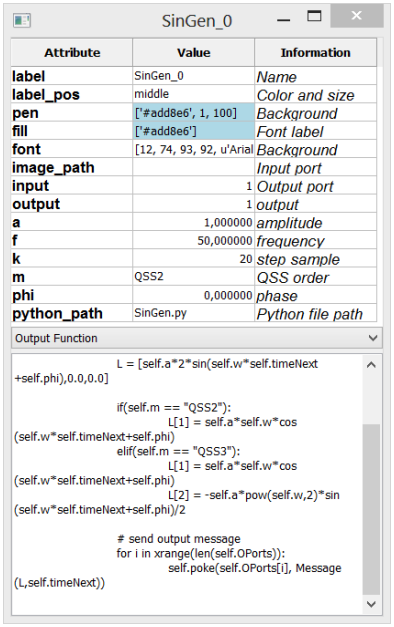


FIGURE 42 : EXEMPLE D'ATTRIBUTS SR LE MODÈLE SINGEN_0.

Les attributs comportementaux sont les paramètres du constructeur de la classe du modèle DEVSimPy. L'importation de ces paramètres est automatique dans DEVSimPy. Le type, la valeur par défaut et les informations des variables remontent automatiquement dans la fenêtre de dialogue à partir du patron du constructeur de la classe. **Attention, il faut absolument que les paramètres du constructeur possèdent une valeur par défaut**. Dans l'exemple de la Figure 42, les attributs comportementaux sont : *a=1.0*, *f=50.0*, *k=20*, *m= 'QSS2'* et *phi=0.0*. L'attribut *python_path* permet de modifier le fichier Python utilisé par DEVSimPy pour instancier les modèles PyDEVS. De plus, la colonne Information est construite automatiquement en fonction du *docstring* du constructeur de la classe (voir la section Gestion de la documentation des modèles, page 50). Ce fichier est consultable en lecture seule dans le bloc inférieur de la fenêtre. L'utilisateur peut naviguer dans les fonctions de transition à l'aide d'une liste de choix au dessus du présentateur de code.

L'utilisateur peut modifier les valeurs des champs des attributs en cliquant sur la cellule correspondante. En fonction du type de valeur, un éditeur d'attributs spécifique sera invoqué. Pour le nom du label, un éditeur de nom est proposé (Figure

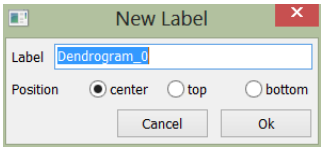


FIGURE 43 : DIALOGUE DE LABEL.

FIGURE 44 : PALETTE DE COULEUR.

43). Un raccourci clavier permet de retrouver cette fenêtre de dialogue : Ctrl+clique-droit de la souris. Pour le paramètre *Fill*, une palette de couleur est proposée (Figure 44).

MANIPULATION DES LIBRAIRIES

Les librairies permettent de rassembler et de structurer les modèles DEVS. DEVSimPy permet la création et la gestion des librairies par l'intermédiaire d'un gestionnaire de librairie.

CRÉATION

Créer une librairie consiste à construire un ensemble de modèles qui peuvent être organisés dans une hiérarchie de répertoires. Dans le langage Python, cela correspond à créer un paquetage de modules. Dans DEVSimPy, le répertoire accueillant les librairies importables est le répertoire *Domain* (situé à la racine de DEVSimPy). Bien entendu, il est possible d'importer des librairies externes au répertoire *Domain*. Il existe deux manières de créer une librairie de modèle DEVSimPy : par le système d'exploitation ou en passant par DEVSimPy.

CRÉATION PAR LE SYSTÈME D'EXPLOITATION

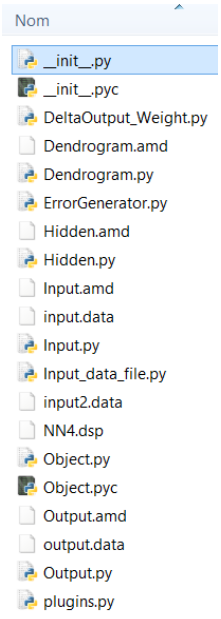


FIGURE 45 : FICHIER __INIT__.PY DE LA LIBRAIRIE NN.

La première étape consiste à créer un sous-répertoire R dans le répertoire *Domain* et de lui ajouter un fichier *__init__.py*. Avec le langage Python, ce fichier est préconisé pour simplifier l'écriture de l'importation des modules et pour les organiser (module). Dans DEVSimPy, ce fichier sert aussi à importer les modèles de type *.py*. Dans le fichier *__init__.py* il est impératif de définir la variable *__all__* comme la liste des noms de fichiers *.py* à considérer lorsque la librairie est importée dans DEVSimPy. Tous les noms de fichier qui ne sont pas définis dans la variable *__all__* ne seront pas visibles dans le panel de librairies importées. Par contre, tous les modèles de type *.amd* et *.cmd* n'ont pas besoin d'apparaitre dans le fichier *__init__.py* car ils sont dépendants de la plate-forme DEVSimPy et sont automatiquement visibles lorsqu'ils sont importés.

La deuxième étape consiste à créer ou copier/coller les modèles (*.py*, *.amd* et *.cmd*) dans le sous-répertoire. Il est possible d'organiser les fichiers en sous-répertoire mais il faut absolument un fichier *__init__.py* par sous-répertoire (Figure 45). Tout répertoire ne contenant aucun fichier *.amd* et *.cmd* mais plusieurs fichiers *.py* qui ne sont pas définis dans la variable *__all__* du fichier *__init__.py* ne sera pas visible dans DEVSimPy après son importation. Dans l'exemple de la Figure 46, la variable *__all__* du fichier *__init__.py* du répertoire NN présente les noms des fichiers *.py* (modèle de type *.py*) devant apparaitre pour une utilisation éventuelle comme modèle dans DEVSimPy (glisser/déposer dans le *Canvas*). Par contre les fichiers *Object.py* et *plugins.py* n'apparaissent pas dans la librairie de DEVSimPy car ils ne sont pas définis par la variable *__all__*.

```
__all__ = [
            "Output",
            "Hidden",
            "Input",
            "ErrorGenerator",
            "DeltaOutput_Weight",
            "Dendrogram"
]
```

FIGURE 46 : VARIABLE __ALL__.

La troisième étape consiste à importer le répertoire R dans DEVSimPy à partir du panel *Control*. Nous verrons plus tard comment effectuer cette étape relativement simple.

CRÉATION PAR DEVSIMPY

En cliquant-droit à l'intérieur du panel *Librairies*, un menu contextuel permet de (Figure 47):

- **New/Import** : créer une nouvelle librairie ou pour importer une librairie existante (dans *Domain* ou externe) ;
- **Actualiser** : d'actualiser les librairies ;
- **Aide** : d'aider à la gestion des librairies.

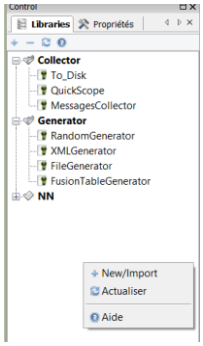


FIGURE 47 : MENU CONTEXTUEL DANS LE PANEL LIBRAIRIE.

Ces fonctionnalités sont aussi accessibles par la barre d'outils située en dessous de l'onglet *Librairies*. L'icône « plus » permet d'invoquer l'action New/Import, l'icône « moins » permet de

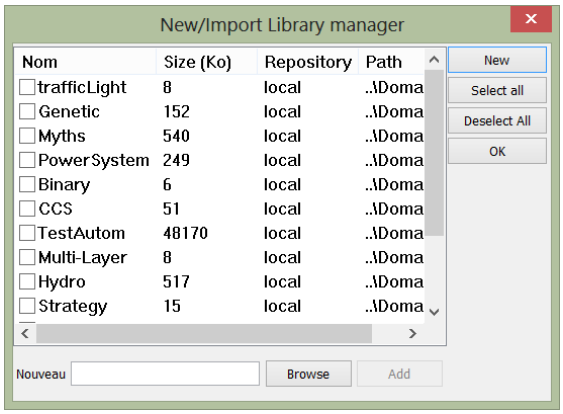


FIGURE 48 : GESTIONNAIRE DE LIBRAIRIE.

supprimer la librairie sélectionnée. L'icône représentant les deux flèches formant un cercle fermé, permet de recharger les librairies. Enfin, l'icône d'information donne un accès à une aide qui permet d'avoir plus d'information sur l'utilisation des librairies dans DEVSimPy.

La Figure 48 montre la fenêtre l'importation ou la création d'une librairie. Grâce aux cases à cocher, l'utilisateur peut sélectionner les librairies qu'il veut importer à partir du répertoire *Domain* de DEVSimPy. Si l'utilisateur veut importer une librairie externe (qui n'est pas représentée par un sous-répertoire du répertorie *Domain*), il doit insérer le chemin absolu de la librairie externe (par le bouton *Browse* par exemple) puis cliquer sur le bouton *Add*. Ensuite il devra cocher la librairie externe puis cliquer sur le bouton *Ok*. Enfin, l'utilisateur a la possibilité de créer une nouvelle librairie interne vierge en cliquant le bouton *New*. Un sous-répertoire du répertoire *Domain* sera crée avec un fichier *__init_.py* vierge.

NAVIGATION

Les librairies sont représentées de manière arborescente. La navigation au sein des librairies se fait en les dépliants par l'intermédiaire de l'icône « plus ». Les librairies se composent de trois types de modèles :

1. les fichiers python (dérivé des classes de PyDEVS) représentés par le logo Python ;
2. les fichiers *.amd* (modèles atomiques) représentés par un cube ;
3. les fichiers *.cmd* (modèle couplés) représentés par un assemblage de cube .

L'instanciation des modèles se fait par un glisser/déposer dans le *Canvas* qui représente le diagramme.

Dans l'exemple de la figure de la page précédente, DEVSimPy propose trois librairies : *Collector*, *Generator* et NN. Chacune de ces librairies possède une suite de modèles. La librairie *Generator* ne possède que des modèles de type *.py*. Par contre la librairie *NN* possède des modèles de type *.amd* (*Output*, *Input*, *Hidden* et *Dendogram*).

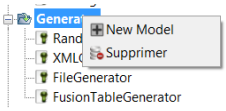


FIGURE 49 : MENU CONTEXTUEL D'UNE LIBRAIRIE.

En bas de cette fenêtre, il est possible de faire une recherche parmi les librairies importées dans DEVSimPy. L'utilisateur peut effectuer un certain nombre d'action sur les librairies (Figure 49):

- **Nouveau modèle** : permet d'ajouter un modèle DEVSimPy par l'intermédiaire du gestionnaire de création décrit dans le chapitre Création par le gestionnaire des modèles à la page 21 ;
- **Supprimer** : permet de supprimer une librairie. Attention, il sera proposé à l'utilisateur de supprimer la librairie du la liste ou de supprimer les sources de la librairie.

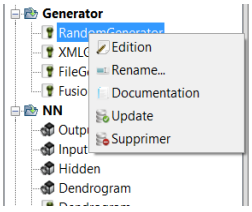


FIGURE 50 : MENU CONTEXTUEL D'UN MODÈLE DANS UNE LIBRAIRIE.

L'utilisateur peut effectuer un certain nombre d'actions sur les modèles sauvegardés dans les librairies. Un clique-droit sur un modèle fait apparaître le menu contextuel suivant (Figure 50) :

- **Edition** : permet d'éditer le code du modèle ;
- **Rename** : permet de renommer le modèle (dans le système de fichiers également) ;

- **Documentation** : permet d'afficher la documentation de la classe correspondant au modèle si cette documentation à été implémentée dans le *docstring* de la classe ;
- **Update** : permet de recharger le modèle dans la librairie ;
- **Supprimer** : permet de supprimer le modèle de la liste ou du disque.

DOCUMENTATION

L'exploitation des *docstrings* dans les modèles qui composent la librairie permet de construire une documentation qui s'affiche au travers d'un pop-up lorsque l'utilisateur survole le modèle dans le panel de libraire. Lorsque l'utilisateur survole le nom d'une librairie, la liste de ses modèles s'affiche ainsi que le nom de ses éventuelles sous-librairies. La Figure 51 montre le survol de la librairie *Generator*. La sous-librairie *fusionTable* n'est pas accessible car elle ne contient pas de modèle distanciable dans DEVSimPy (voir le chapitre Manipulation des librairies, page 46).

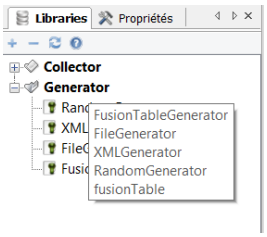


FIGURE 51 : SURVOL DE LA LIBRAIRIE GENERATOR.

GESTION DES PLUGINS

DEVSimPy permet l'extension de ces fonctionnalités par l'ajout de plugins dit « globaux ». La gestion de ces plugins se fait au travers d'une interface accessible par le menu *Options/ Preferences* puis l'onglet *Plugins*. DEVSimPy permet également d'étendre les fonctionnalités d'un modèle atomique par l'ajout de plugins dits « Locaux ». Ces plugins sont des fichiers Python embarqués dans les modèles *.amd* ou *.cmd*. La manipulation de ces fichiers peut se faire par le menu contextuel d'un modèle de type *.amd* ou *.cmd*.

LES PLUGINS LOCAUX

Un plugin local est un fichier Python qui ré-implémente des fonctionnalités existantes d'un modèle ou alors qui implémente de nouvelles fonctionnalités. Les fonctionnalités implémentées peuvent être de type graphiques (instance DEVSimPy) ou comportementales (instance DEVS).

Il existe plusieurs manières d'introduire un plugin dans un modèle :

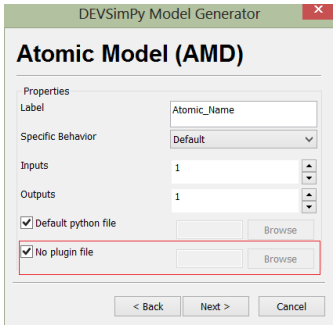


FIGURE 52 : GESTION DES PLUGINS LOCAUX PENDANT LA CRÉATION DES MODÈLES ATOMIQUE.

- À la construction d'un modèle (atomique dans l'exemple de la Figure 52), le gestionnaire de création de modèle donne la possibilité d'insérer un fichier plugin existant. Si la case à cocher (encadrée en rouge dans la Figure 52) est décochée, l'utilisateur peut inclure un fichier *plugins.py* dans l'archive du modèle.
- Lorsque le modèle ne possède pas de plugin local, le menu contextuel invoqué par un clic droit, fait apparaître un menu *Plugins*. Ce menu permet de manipuler le plugin du modèle déjà crée ou instancié à partir d'une librairie. Si le modèle ne possède pas de plugin, il est possible d'en ajouter un par cette interface montrée sur la Figure 53 en cliquant le bouton *Add*. Une fenêtre de dialogue permet de charger un fichier *plugins.py* à partir du disque. Attention, le fichier doit absolument se nommer *plugins.py*. Si une erreur est présente dans le fichier *plugins.py*, il ne sera pas importé et l'utilisateur est invité à l'éditer pour corriger les erreurs. Une fois l'erreur corrigée, il pourra réitérer l'ajout pour rendre effectif l'importation du plugin local.

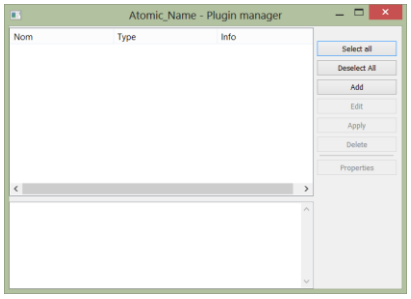


FIGURE 53 : GESTIONNAIRE DE PLUGINS LOCAUX.

Lorsque le plugin local est correctement embarqué dans le modèle, il est possible de l'activer. Le menu contextuel sur le modèle permet d'invoquer le gestionnaire de plugin local et la fenêtre de la figure Figure 54 s'affiche. Ici, le plugin est activé (symbole vert en face du nom) et contient une méthode graphique *OnLeftDClick*. Cette méthode est une fonction callback (attribut *<type function>*) invoquée à l'appel de l'événement utilisateur double clique droit de la souris. Le plugin surcharge (attribut *overriding*) cette fonctionnalité déjà existante pour les modèles *.amd* ou *.cmd*. Pour les modèles *.amd*, un double-clique droit provoque l'ouverture de

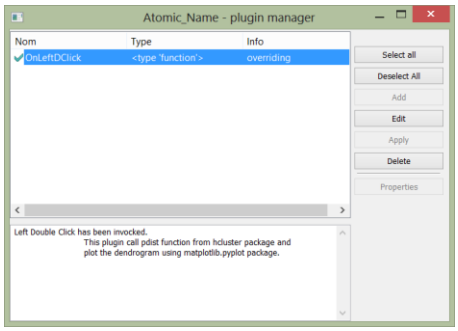


FIGURE 54 : ACTIVATION D'UN PLUGIN LOCAL.

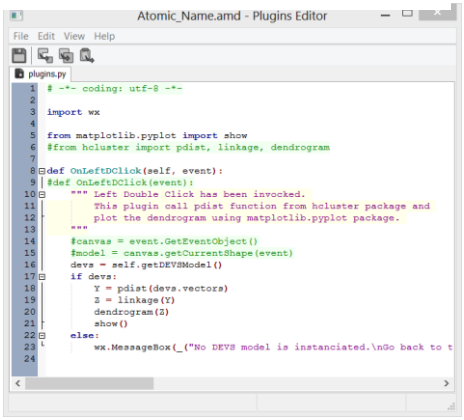


FIGURE 55 : EDITION DU CODE PYTHON POUR LES PLUGINS LOCAUX.

la fenêtre de propriété (sauf pour les modèles de la librairie *Collector*) alors que pour les modèles *.cmd* une fenêtre s'ouvre pour visualiser leur contenu (hiérarchie de description).

L'utilisateur peut trouver une description (*docstring*) de la fonction sélectionnée en dessous de la liste des plugins locaux disponibles. Il est possible de désactiver le plugin par un simple double-clique ou en appuyant sur le bouton *Deselect All*. Il est également possible de supprimer le plugin de l'archive par le bouton *Delete*. Enfin il est possible d'éditer le plugin par le bouton *Edit*. Une fenêtre d'édition permet de modifier le code comme il est montré sur la Figure 55.

La signature de la fonction *OnLeftDClick* est identique à celle d'une fonction *callback* avec un passage en paramètre de l'événement responsable de son exécution (le double clique droit de la souris). À l'intérieur de cette fonction, l'instance du modèle DEVS correspondant au modèle *.amd* est disponible par un simple *self.getDEVSModel()*. Le reste de la procédure est spécifique à l'action désirée. Ici, une fenêtre de la bibliothèque *matplotlib* est utilisée pour afficher des données de simulation. Un test est inséré pour bien vérifier qu'une simulation à bien été réalisée avant de double-clique sur le modèle. Après toute interaction avec le plugin, il est nécessaire d'appliquer les modifications en invoquant le bouton *Apply*. Cette action permet de réimporter le plugin dans le modèle de manière dynamique. L'utilisateur n'a pas besoin de re-instancier le modèle pour prendre en compte les modifications du plugin dans le modèle. Enfin le bouton *Propriétés* restera toujours inaccessible car il n'est utilisable que pour les plugins globaux.

L'écriture d'un plugin local n'est pas une chose simple car elle nécessite une bonne connaissance de l'architecture des modèles DEVSimPy ainsi que de leur API (Application Program Interface).

LES PLUGINS GLOBAUX

Les plugins globaux sont des plugins qui étendent les fonctionnalités du logiciel DEVSimPy. Ils sont souvent utilisés pour ajouter des fonctionnalités qui concernent la modélisation ou la simulation. Par exemple, le plugin général *blink* permet d'effectuer une simulation pas à pas. Lorsque ce plugin est activé, l'utilisateur déroule la simulation en cliquant successivement sur un bouton le faisant avancer dans la simulation. À chaque pas de simulation, une trace de celle-ci est affichée et les modèles changent de couleur en fonction du type de traitement qu'ils sont en train d'effectuer.

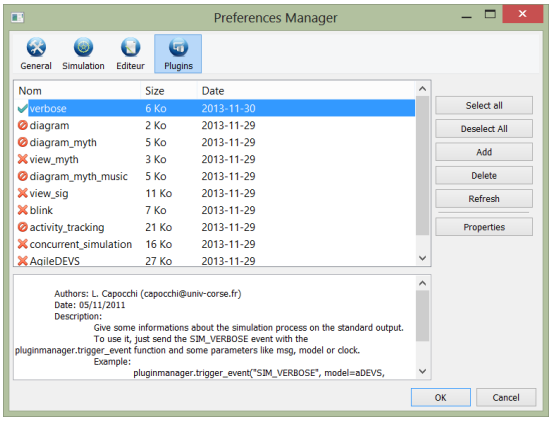


FIGURE 56 : GESTIONNAIRE DE PLUGINS GLOBAUX.

La fenêtre de gestion des plugins globaux est invoquée à partir du menu *Options/Préférences* puis l'onglet *Plugins*. La partie gauche de la fenêtre présente la liste des plugins disponibles (Figure 56). Cette liste est établie en fonction du contenu de la variable *__all__* définie dans le fichier *__init__.py* dans le répertoire plugins de DEVSimPy. Le chemin de ce répertoire est configurable dans les préférences de DEVSimPy. La partie de droite offre la possibilité d'interagir avec les plugins comme pour les plugins locaux. La partie inférieure de la fenêtre permet d'afficher la documentation

(*docstring*) d'un plugin lorsque celui-ci est sélectionné. Pour activer/désactiver un plugin, il suffit de double-cliquer dessus. Lorsqu'un plugin possède une erreur, une icône de sens interdit est affichée en face du nom.

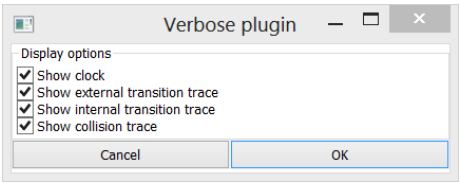


FIGURE 57 : OPTIONS DISPONIBLES POUR LE PLUGIN GLOBAL VERBOSE.

Nous remarquons que le bouton *Properties* est disponible pour le plugin *verbose* (voir figure ci-contre). En effet, si un plugin implémente une fonction *Config*, elle sera proposée à l'utilisateur pour configurer le plugin. Bien entendu, il faut que le développeur du plugin implémente les paramètres configurables du plugin à l'intérieur. Par exemple pour le plugin *verbose*, il est possible de définir quel type d'information on veut afficher (Figure 57).

L'écriture d'un plugin global n'est pas une chose simple car elle nécessite une bonne connaissance de l'architecture de DEVSimPy ainsi que de son API (Application Program Interface).

GESTION DE LA DOCUMENTATION DES MODÈLES

Toute la gestion de la documentation d'un objet dans DEVSimPy passe par sa *docstring*. La docstring est une chaîne de caractères qui est implémentée en dessous de la définition d'un objet ou grâce à un *setter*. Python et wxPython offrent ensuite automatiquement les méthodes (*getter* et *setter*) pour accéder à ces chaînes à partir des instances des objets.

En ce qui concerne DEVSimPy, il est possible d'accéder à la documentation :

- d'un modèle au travers de sa fenêtre de propriété ;
- d'un paramètre du constructeur d'une classe d'un objet ;
- d'un module d'une librairie en le survolant ou en invoquant son menu contextuel ;
- d'un plugin en le sélectionnant dans le gestionnaire de plugin.

En ce qui concerne la rédaction de la documentation d'un objet, il est conseillé de renseigner sa documentation pendant son implémentation. À ce sujet, la rédaction de la documentation des paramètres du constructeur d'une classe d'un modèle est très importante car elle aide énormément l'utilisateur qui veut utiliser le modèle. Pour se faire, il suffit d'insérer autant de lignes que de paramètres du constructeur (en excluant *self*) dans le *docstring* du constructeur. Pour chaque ligne le format à respecter est le suivant :

@param <*nom du paramètre*> [:|=] <*description du paramètre*>

Par exemple pour le modèle *To_Disk* de la librairie *Collector*, les spécifications sont :

Attribut	Valeur	Information
label	QuickScope_0	*Nom*
label_pos	center	*Color and size*
pen	['#add8e6', 1, 100]	*Background*
fill	['#add8e6']	*Font label*
font	[12, 74, 93, 92, u'Aria	*Background*
image_path		*Port d'entrée*
input	1	*Port de sortie*
output	1	*output*
xlabel		*xlabel*
ylabel		*ylabel*
eventAxis	☐	*Flag to plot*
fusion	☑	*Flag to plot all*
python_path	QuickScope.py	*Python file*

```
###
def __init__(self, fusion = True, eventAxis = False):
    """ Constructor.

        @param fusion : Flag to plot all signals on one graphic
        @param eventAxis : Flag to plot depending event axis
    """
```

EXEMPLES DE MODÉLISATION DANS DEVSIMPY

Le but de ce chapitre est de présenter la modélisation et la simulation d'un système avec le formalisme DEVS et DEVSimPy.

LE DIAGNOSTIQUE DES MACHINES ASYNCHRONES

Ce domaine d'application montre l'utilisation de DEVS avec le concept de quantification des états d'un système. Il utilise la librairie *PowerSystem* de DEVSimPy.

Le modèle de la machine asynchrone est obtenu par une mise en équation des courants statoriques et rotoriques du modèle orienté circuit proposé dans [10]. Il consiste en un système complet de six équations différentielles linéaires du 1er ordre à six inconnues.

MODÉLISATION

Cette section décrit le modèle orienté circuit et le modèle mathématique d'une machine à induction triphasée. Pour plus de détails le lecteur peut se rapporter à [10].

MODÈLE CIRCUIT

Le modèle orienté circuit d'une machine à induction triphasée est donné sur la Figure 58. Ce modèle peut être séparé en trois parties distinctes : Le stator, le rotor et le couplage magnétique entre ces deux entités.

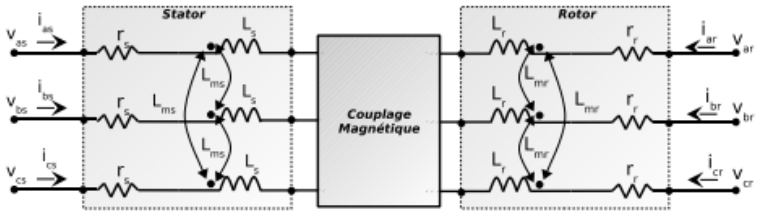


FIGURE 58 : MODÈLE ORIENTÉ CIRCUIT DE LA MACHINE À INDUCTION TRIPHASÉE.

Le stator est alimenté par un système triphasé équilibré composé des tensions sinusoïdales $v_{as}(t)$, $v_{bs}(t)$ et $v_{cs}(t)$. Chaque phase est caractérisée par une résistance r_s et une inductance L_s. Les interactions magnétiques entre chaque phase du stator sont fonction d'une inductance mutuelle L_{ms} et des courants statoriques voisins.

De même, le rotor est alimenté par un système triphasé équilibré composé des tensions sinusoïdales $v_{ar}(t)$, $v_{br}(t)$ et $v_{cr}(t)$. Chaque phase est caractérisée par une résistance r_r et une inductance L_r.

Les interactions magnétiques entre chaque phase du rotor sont fonction d'une inductance mutuelle L_{mr} et des courants rotoriques voisins. Les effets du rotor sur le stator (resp. du stator sur le rotor) sont fonction d'une inductance mutuelle L_{sr} (resp. L_{rs}), des courants rotoriques $i_{ar}(t)$, $i_{br}(t)$ et $i_{cr}(t)$ (resp. $i_{as}(t)$, $i_{bs}(t)$ et $i_{cs}(t)$) et de la position électrique du rotor $\theta r(t)$.

Les entités observées en sortie du système sont les courants (statoriques et rotoriques) ainsi que le couple électromagnétique de la machine $T_e(t)$. Voyons à présent comment mettre en équation le système électrique en fonction de ces entités.

MODÈLE MATHÉMATIQUE

Le but de cette section est de donner les relations en tension de chaque phase du coté du stator et du coté du rotor.

MODÈLE DU STATOR

Le modèle électrique du stator, représenté sur la Figure 59, fait intervenir à la fois des éléments passifs comme les résistances r_s et des éléments actifs comme les inductances propres L_s et les inductances mutuelles L_{ms}.
D'après la loi des mailles :

$$\begin{cases} v_{as}(t) = P_A^1 - P_A^2 & = r_s.i_{as}(t) + L_s\frac{d}{dt}i_{as}(t) - \frac{L_{ms}}{2}\left[\frac{d}{dt}i_{bs}(t) + \frac{d}{dt}i_{cs}(t)\right] \\ v_{bs}(t) = P_B^1 - P_B^2 & = r_s.i_{bs}(t) + L_s\frac{d}{dt}i_{bs}(t) - \frac{L_{ms}}{2}\left[\frac{d}{dt}i_{as}(t) + \frac{d}{dt}i_{cs}(t)\right] \\ v_{cs}(t) = P_C^1 - P_C^2 & = r_s.i_{cs}(t) + L_s\frac{d}{dt}i_{cs}(t) - \frac{L_{ms}}{2}\left[\frac{d}{dt}i_{as}(t) + \frac{d}{dt}i_{bs}(t)\right] \end{cases} \quad (1)$$

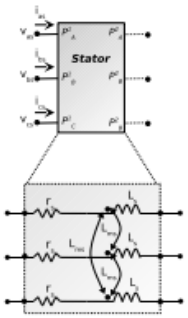


FIGURE 59 : STATOR.

Le système 1 montre que chaque phase x (x ∈ {a,b,c}) est composée des trois termes correspondants :

- $r_s.i_{xs}(t)$: la chute résistive,
- $L_s.\frac{d}{dt}i_{xs}(t)$: l'effet inductif produit par l'enroulement considéré,
- $-\frac{L_{ms}}{2}\left[\frac{d}{dt}i_{x's}(t) + \frac{d}{dt}i_{x''s}(t)\right]$: l'effet inductif produit par les enroulements voisins du stator.

Les tensions d'alimentations $v_{as}(t)$, $v_{bs}(t)$ et $v_{cs}(t)$ sont des fonctions sinusoïdales du temps.

MODÈLE DU ROTOR

Le modèle électrique du rotor, représenté sur la Figure 60, fait intervenir à la fois les résistances r_r, les inductances propres L_r et les inductances mutuelles L_{mr}.

D'après la loi des mailles :

$$\begin{cases} v_{ar}(t) = P_a^1 - P_a^2 & = r_r.i_{ar}(t) + L_r\frac{d}{dt}i_{ar}(t) - \frac{L_{mr}}{2}\left[\frac{d}{dt}i_{br}(t) + \frac{d}{dt}i_{cr}(t)\right] \\ v_{br}(t) = P_b^1 - P_b^2 & = r_r.i_{br}(t) + L_r\frac{d}{dt}i_{br}(t) - \frac{L_{mr}}{2}\left[\frac{d}{dt}i_{ar}(t) + \frac{d}{dt}i_{cr}(t)\right] \\ v_{cr}(t) = P_c^1 - P_c^2 & = r_r.i_{cr}(t) + L_r\frac{d}{dt}i_{cr}(t) - \frac{L_{mr}}{2}\left[\frac{d}{dt}i_{ar}(t) + \frac{d}{dt}i_{br}(t)\right] \end{cases} \quad (2)$$

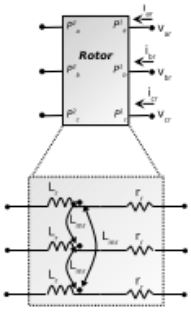


FIGURE 60 : ROTOR.

Le système 2 montre que chaque phase x (x ∈ {a,b,c}) est composée des trois termes correspondants :

- $r_r.i_{xr}(t)$: la chute résistive,
- $L_r\frac{d}{dt}i_{xr}(t)$: l'effet inductif produit par l'enroulement considéré,
- $-\frac{L_{mr}}{2}\left[\frac{d}{dt}i_{x'r}(t) + \frac{d}{dt}i_{x''r}(t)\right]$: l'effet inductif produit par les enroulements voisins du rotor.

Les tensions d'alimentations sont $v_{ar}(t)$, $v_{br}(t)$ et $v_{cr}(t)$ des fonctions sinusoïdales du temps.

MODÈLE DU COUPLAGE MAGNÉTIQUE STATOR/ROTOR

Le modèle électrique du couplage magnétique entre le stator et le rotor utilise les inductances mutuelles L_{sr} et L_{rs}, les courants statoriques et rotoriques induits et la position électrique du rotor qr .

L'effet du couplage a lieu pour chaque phase coté stator et rotor :

$$\begin{cases} L_{sr}\frac{d}{dt}\left[i_{ar}.\cos(\theta_r)+i_{br}.\cos(\theta_r+\frac{2\pi}{3})+i_{cr}.\cos(\theta_r-\frac{2\pi}{3})\right] & \text{phase a du stator} \\ L_{sr}\frac{d}{dt}\left[i_{ar}.\cos(\theta_r-\frac{2\pi}{3})+i_{br}.\cos(\theta_r)+i_{cr}.\cos(\theta_r+\frac{2\pi}{3})\right] & \text{phase b du stator} \\ L_{sr}\frac{d}{dt}\left[i_{ar}.\cos(\theta_r+\frac{2\pi}{3})+i_{br}.\cos(\theta_r-\frac{2\pi}{3})+i_{cr}.\cos(\theta_r)\right] & \text{phase c du stator} \end{cases} \quad (3)$$

$$\begin{cases} L_{rs}\frac{d}{dt}\left[i_{as}.\cos(\theta_r)+i_{bs}.\cos(\theta_r-\frac{2\pi}{3})+i_{cs}.\cos(\theta_r+\frac{2\pi}{3})\right] & \text{phase a du rotor} \\ L_{rs}\frac{d}{dt}\left[i_{as}.\cos(\theta_r+\frac{2\pi}{3})+i_{bs}.\cos(\theta_r)+i_{cs}.\cos(\theta_r-\frac{2\pi}{3})\right] & \text{phase b du rotor} \\ L_{rs}\frac{d}{dt}\left[i_{as}.\cos(\theta_r-\frac{2\pi}{3})+i_{bs}.\cos(\theta_r+\frac{2\pi}{3})+i_{cs}.\cos(\theta_r)\right] & \text{phase c du rotor} \end{cases} \quad (4)$$

La position électrique θ r est déduite de la position mécanique du rotor θ rm ou de l'intégration de la vitesse de rotation mécanique Ω par l'équation :

$$\theta_{rm} = p.\theta_r = p.\int_t(\Omega\, dt)$$

avec p le nombre de paire de pôles.

La vitesse de rotation mécanique du rotor Ω(t) est la solution de l'équation linéaire du premier ordre suivante :

$$J.\frac{d}{dt}\Omega(t)+f.\Omega(t)=T_e(t)-T_l$$

avec :

- J, f : le coefficient d'inertie et le frottement visqueux ;
- T_e, T_l : le couple électromagnétique et le couple de charge.

Le couple électromagnétique dépend des courants statoriques et rotoriques et de la position mécanique θ r:

$$\begin{aligned} T_e(t) = & -i_{as}(t).p.L_{sr}[i_{ar}(t).\sin(p.\theta_r)+i_{br}(t).\sin(p.\theta_r+\frac{2.\pi}{3})+i_{cr}(t).\sin(p.\theta_r-\frac{2.\pi}{3})] \\ & -i_{bs}(t).p.L_{sr}[i_{ar}(t).\sin(p.\theta_r-\frac{2.\pi}{3})+i_{br}(t).\sin(p.\theta_r)+i_{cr}(t).\sin(p.\theta_r+\frac{2.\pi}{3})] \\ & -i_{cs}(t).p.L_{sr}[i_{ar}(t).\sin(p.\theta_r+\frac{2.\pi}{3})+i_{br}(t).\sin[p.\theta_r-\frac{2.\pi}{3})+i_{cr}(t).\sin(p.\theta_r)] \end{aligned} \quad (7)$$

Nous pouvons à présent donner le modèle mathématique complet d'une machine à induction triphasée en regroupant, pour chaque phase dans le cas du stator et du rotor, les équations 1,2,3,4.

MODÈLE COMPLET

L'association des équations 1,2,3,4,6 et 5 permet de construire le modèle mathématique complet d'une machine à induction triphasé :

Nous allons à présent procéder à la résolution de ce système d'équations différentielles complet de huit équations à huit inconnues.

$$\begin{cases} v_{as}(t) & = r_s.i_{as}+L_s\frac{d}{dt}i_{as}-\frac{L_{ms}}{2}\left[\frac{d}{dt}i_{bs}+\frac{d}{dt}i_{cs}\right]+L_{sr}\frac{d}{dt}\left[i_{ar}.\cos(\theta_r(t))+i_{br}.\cos(\theta_r(t)+\frac{2\pi}{3})+i_{cr}.\cos(\theta_r(t)-\frac{2\pi}{3})\right] \\ v_{bs}(t) & = r_s.i_{bs}+L_s\frac{d}{dt}i_{bs}-\frac{L_{ms}}{2}\left[\frac{d}{dt}i_{as}+\frac{d}{dt}i_{cs}\right]+L_{sr}\frac{d}{dt}\left[i_{ar}.\cos(\theta_r(t)-\frac{2\pi}{3})+i_{br}.\cos(\theta_r(t))+i_{cr}.\cos(\theta_r(t)+\frac{2\pi}{3})\right] \\ v_{cs}(t) & = r_s.i_{cs}+L_s\frac{d}{dt}i_{cs}-\frac{L_{ms}}{2}\left[\frac{d}{dt}i_{as}+\frac{d}{dt}i_{bs}\right]+L_{sr}\frac{d}{dt}\left[i_{ar}.\cos(\theta_r(t)+\frac{2\pi}{3})+i_{br}.\cos(\theta_r(t)-\frac{2\pi}{3})+i_{cr}.\cos(\theta_r(t))\right] \\ v_{ar}(t) & = r_r.i_{ar}+L_r\frac{d}{dt}i_{ar}-\frac{L_{ms}}{2}\left[\frac{d}{dt}i_{br}+\frac{d}{dt}i_{cr}\right]+L_{rs}\frac{d}{dt}\left[i_{as}.\cos(\theta_r(t))+i_{bs}.\cos(\theta_r(t)-\frac{2\pi}{3})+i_{cs}.\cos(\theta_r(t)+\frac{2\pi}{3})\right] \\ v_{br}(t) & = r_r.i_{br}+L_r\frac{d}{dt}i_{br}-\frac{L_{ms}}{2}\left[\frac{d}{dt}i_{ar}+\frac{d}{dt}i_{cr}\right]+L_{rs}\frac{d}{dt}\left[i_{as}.\cos(\theta_r(t)+\frac{2\pi}{3})+i_{bs}.\cos(\theta_r(t))+i_{cs}.\cos(\theta_r(t)-\frac{2\pi}{3})\right] \\ v_{cr}(t) & = r_r.i_{cr}+L_r\frac{d}{dt}i_{cr}-\frac{L_{ms}}{2}\left[\frac{d}{dt}i_{ar}+\frac{d}{dt}i_{br}\right]+L_{rs}\frac{d}{dt}\left[i_{as}.\cos(\theta_r(t)-\frac{2\pi}{3})+i_{bs}.\cos(\theta_r(t)+\frac{2\pi}{3})+i_{cs}.\cos(\theta_r(t))\right] \\ T_e(t)-T_l & = J.\frac{d}{dt}\Omega(t)+f.\Omega(t) \\ \Omega(t) & = \frac{d}{dt}\theta_r(t) \end{cases} \quad (8)$$

MODÉLISATION DEVS

Le modèle DEVS du système étudié utilise la librairie *PowerSystem* qui implémente le modèle de l'intégrateur basé sur la méthode QSS (Quantized State System) [11].

Le modèle DEVS du système d'équations 8 est donné sur la Figure 61. On peut distinguer 3 parties principales qui correspondent à la partie stator (modèle couplé à gauche de la figure), la partie rotor (modèle couplé à droite de la figure) et la partie force électromotrice constituée des deux modèles couplés (bloc rectangulaire en bas de la figure). La mécanique est représentée par un modèle couplé nommé *Mecanical_Couple* (en bas de la figure) et la partie permettant le calcul du couple électromoteur T_e par un modèle couplé nommé *TI*.

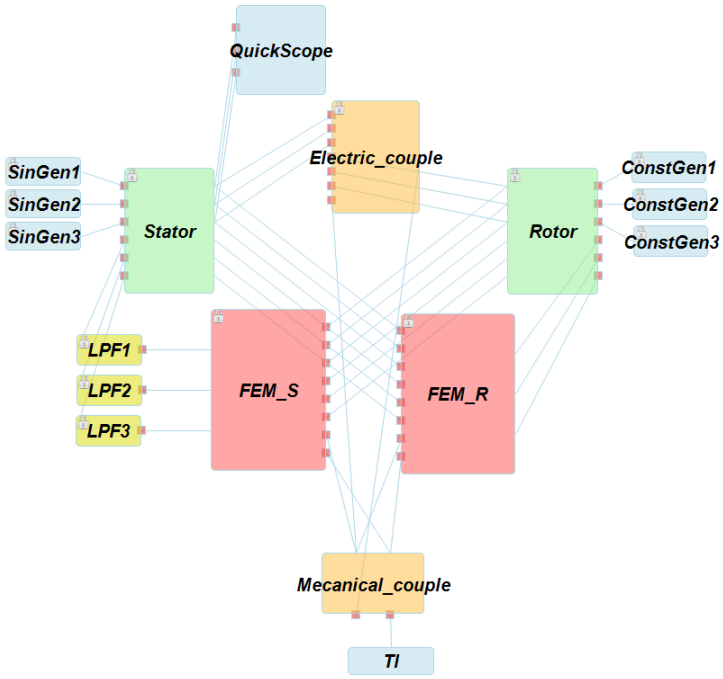


FIGURE 61 : MODÈLE DE LA MACHINE ÉLECTRIQUE.

Le schéma de la Figure 62 montre le contenu du modèle couplés *Stator*. Il est composé des trois phases qui présentent un sommateur, un intégrateur et surtout un filtre passe bas (LP_i) indispensable à la correction du bruit numérique introduit en début de simulation par l'intégrateur. Le calibrage de ces filtres sera discuté plus tard. Le modèle *Rotor* est identique mais présente des valeurs des paramètres différents.

L'insertion des filtres passe bas dans la modélisation est discutable. E. Kofman propose dans [11] d'introduire un modèle *Loop Breaking* qui, lorsqu'il est introduit dans une boucle algébrique (modèles interconnecté en boucle), permet de casser celle-ci dès qu'il relève une invariance de son événement présent sur son unique port d'entrée.

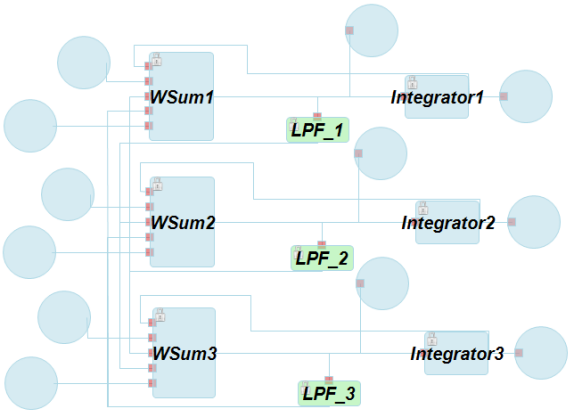


FIGURE 62 : MODÈLE COUPLÉ DU STATOR.

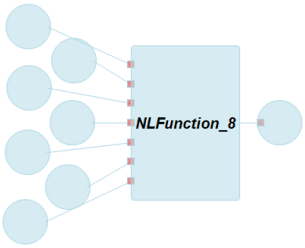


FIGURE 63 : MODÈLE COUPLÉ DU COUPLE ÉLECTRIQUE.

La Figure 63 montre le contenu du modèle couplé *Electrical_Couple*. Il présente le modèle atomique *NLFunction* qui permet de modéliser la fonction non linaire pour le calcul du couple T_e donnée par la formule 2.

La Figure 65 montre le contenu du modèle couplé *FEM_S*. Il présente le modèle atomique *NLFunction* qui permet de modéliser la fonction non linaire pour le calcul des couples électromagnétiques.

La Figure 64 montre le contenu du modèle couplé *Mecanical_couple*.

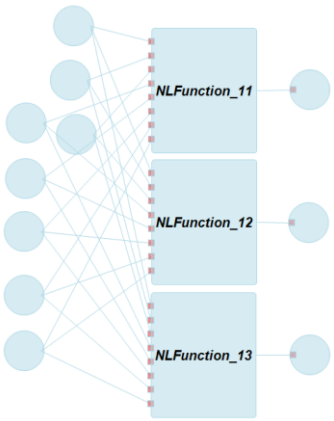


FIGURE 65 : MODÈLE COUPLÉ FEM_S.

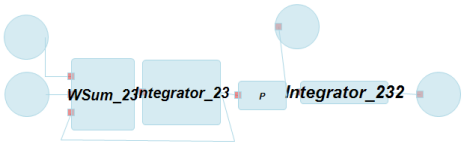


FIGURE 64 : MODÈLE COUPLÉ DE LA PARTIE MÉCANIQUE.

Les modèles atomiques *SinGen1*, *SinGen2* et *SinGen3* permettent la génération des tensions sinusoïdales $v_{as}(t)$, $v_{bs}(t)$ et $v_{cs}(t)$. Les modèles atomiques *ConstGen1*, *ConstGen2* et *ConstGen3* permettent la génération des tensions constantes v_{ar}, v_{br} et v_{cr}. Pendant la simulation nous observons les courants statoriques et rotoriques (resp. Is{a,b,c} et Ir{a,b,c}) ainsi que le couple électromoteur T_e et la vitesse mécanique $\Omega(t)$. Nous observerons éventuellement les forces électromotrices coté stator et rotor (resp. *FEM_S* et *FEM_R*). Nous allons à présent simuler le système dans différentes configurations.

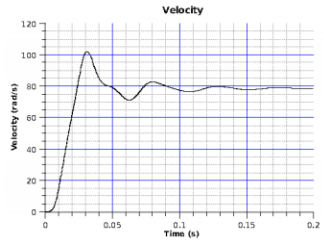
SIMULATION

Cette section présente quelques modes de fonctionnement simple de la machine MADA 5.5 KW (Machine Asynchrone à Double Alimentation). Enfin, nous présentons une discussion sur le calibrage du modèle (coefficients des filtres et des pas de quantification des intégrateurs) ainsi que sur la comparaison des résultats obtenus avec Matlab/Simulink. Les valeurs des paramètres du système 8 choisi pour les besoins de la simulation sont résumées dans le tableau suivant :

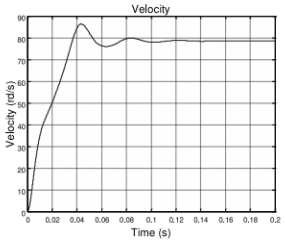
Tension composée efficace (U_m)	380V
Fréquence (f)	50Hz
Pôles (p)	4
Coefficient d'inertie (J)	0.1kg.m²
Coefficient d'atténuation (f)	0.001Nm.s/rad
Couple de charge nominale (T_{ln})	73Nm
Résistance au stator (r_s)	0.528Ω
Résistance au rotor (r_r)	0.282Ω
Inductance au stator (L_s)	0.04732H
Inductance au rotor (L_r)	0.01452H
Inductance magnétique au stator (L_{ms})	0.01732H
Inductance magnétique au stator (L_{mr})	0.005852H
Inductance mutuelle ($L_{sr} = L_{rs}$)	0.02259H

Nous considérons le système complet avec l'intégration des équations mécaniques. Nous allons simuler ce système pour un couple de charge Tl = 0Nm et Tl = 75Nm et une vitesse initiale nulle. Les résultats graphiques sont obtenus avec dq=0.001 (pour tous les intégrateurs du système) pour DEVS et une méthode ODE5 (pas fixe de 10^{-5} seconde) pour *Simulink*.

MACHINE À VIDE AVEC VITESSE INITIALE NULLE



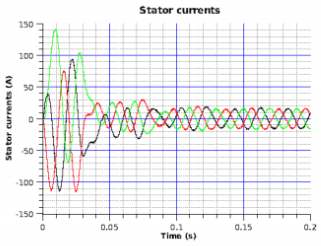
(a) DEVS



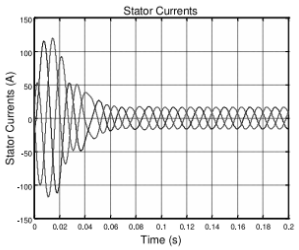
(b) Simulink

La figure ci-contre montre les résultats de simulation de la vitesse mécanique Ω(t) pour TI = 0.

Lorsque TI = 0, cela correspond à l'absence de charge sur la machine. Lorsque la machine est alimentée, elle passe par un régime transitoire durant lequel la vitesse rotorique augmente pour se stabiliser comme on peut le voir sur la figure ci-contre (a).



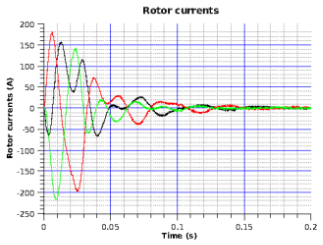
(a) DEVS

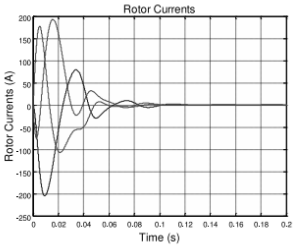


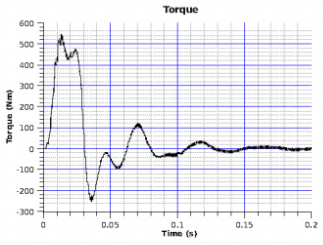
(b) Simulink

La figure ci-contre montre les résultats de simulation des courants statoriques et rotoriques à vide pour TI = 0.

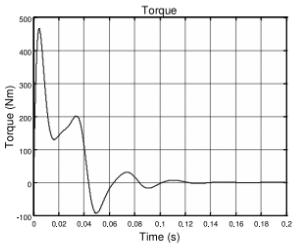
Les courants statoriques et rotoriques présentent le même régime permanent. Les courants statoriques oscillent entre -24 et 24 Ampère alors que les courants rotoriques tendent vers une valeur nulle.





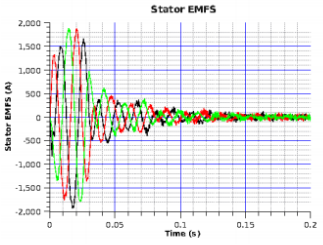


(a) DEVS

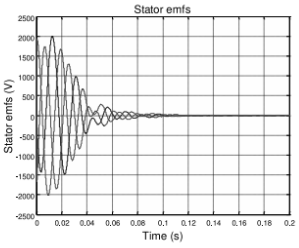


(b) Simulink

La figure ci-contre montre les résultats de simulation du couple électromoteur et de la force électromotrice coté stator avec TI = 0 Nm.



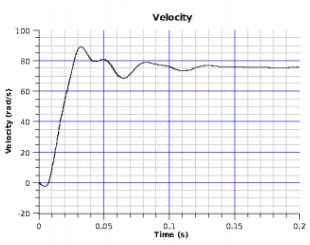
(c) DEVS



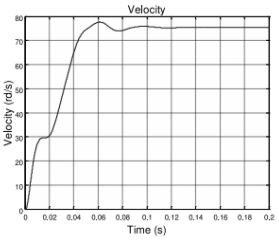
(d) Simulink

On remarque que les courbes du couple électromoteur ne sont pas identiques. Les forces électromotrices du coté du stator évoluent de la même manière.

MACHINE EN PLEINE CHARGE AVEC VITESSE INITIALE NULLE



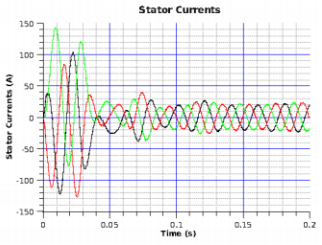
(a) DEVS



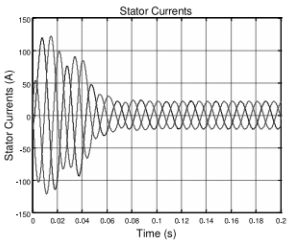
(b) Simulink

Nous imposons à présent un couple de charge Tl = 75Nm égale au couple de charge nominal.

La vitesse mécanique n'a pas le même régime transitoire mais converge vers la même valeur : 75.5 rad/s.

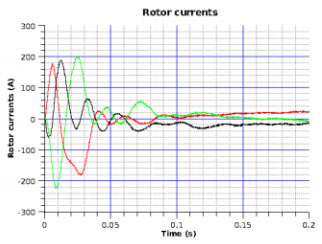


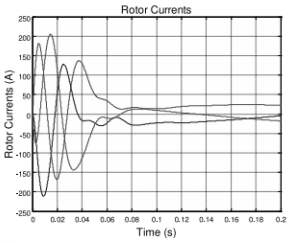
(a) DEVS

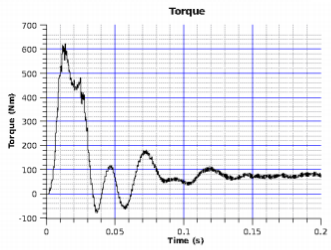


(b) Simulink

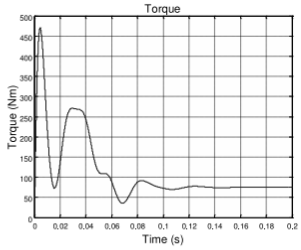
Les courants présentés sur la figure 10 montrent que l'imposition d'un couple de charge Tl = 75Nm implique un appel de courant au niveau du stator et de ce fait une augmentation de l'amplitude des courants rotoriques. Une fois de plus les courants ne présentent pas les mêmes régimes transitoires mais ils convergent vers les mêmes valeurs.



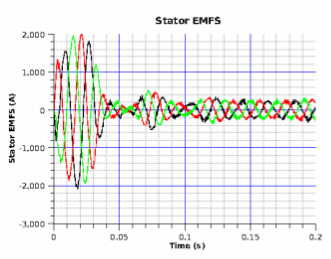




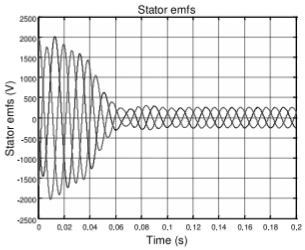
(a) DEVS



(b) Simulink



(c) DEVS



(d) Simulink

Nous remarquons que le couple électromoteur ne converge pas vers la même valeur et nous discuterons de ce résultat par la suite.

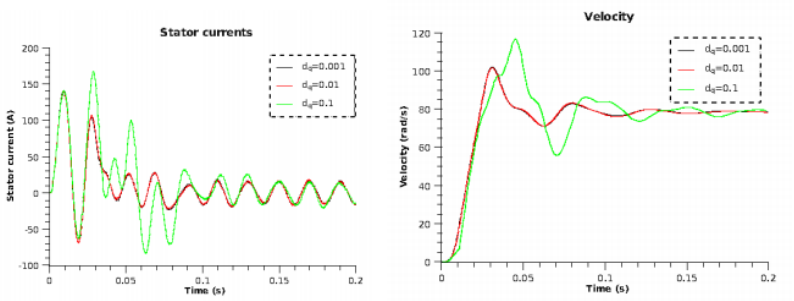
DISCUSSION

Au delà du fait qu'un petit pas de quantification engendre inévitablement des temps de simulation plus grand, la figure ci-dessous montre l'erreur commise sur les signaux lorsque le pas de quantification augmente. On peut noter que seul le régime transitoire est affecté par la valeur de dq. On peut donc, à priori, choisir un pas de quantification de l'ordre de 0.1 puisque seul le régime permanent est étudié par la suite.

Nous avons introduit des filtres passe bas au sein du système pour répondre à un problème de blocage de la simulation. En effet, au début de la simulation, le résultat de l'intégration (au sein du stator par exemple) donne des valeurs importantes qui impliquent une activation des modèles à des temps qui dépassent le temps final de simulation. Par conséquent et en accord avec les algorithmes DEVS, aucun modèle n'est activé et la simulation ne démarre pas. L'introduction de filtres permet de lisser les valeurs brutales et de donner des temps d'activation des modèles raisonnable (inférieur au temps final de simulation). La nouvelle version du logiciel *Simulink* effectue automatiquement ces rectifications mais les anciennes versions nécessitaient la même procédure. L'introduction de filtres passe bas permet donc de débloquer la simulation mais peut introduire une erreur numérique suivant la valeur des coefficients des filtres.

Les coefficients des filtres sont tous égaux à 2000 de manière à pouvoir laisser passer le maximum d'informations. Les valeurs de ces coefficients sont également suffisantes pour débloquer la simulation car les temps de réponse des filtres sont largement inférieurs aux constantes de temps des sous-systèmes filtrés.

À l'heure actuelle, nous pouvons dire que le choix des valeurs des coefficients influe sur la réponse mais aussi sur le temps de simulation. Plus les coefficients des filtres sont bas, plus les réponses filtrées sont tronquées et plus les temps de simulations sont importants du fait, peut être, de la forme exagérée des signaux traités. Dans [12], les auteurs montrent une nouvelle méthode permettant d'adapter le pas de quantification de manière dynamique pendant la simulation.



(a) courant statorique ias (b) vitesse mécanique

En ce qui concerne la comparaison des résultats avec *Simulink*, nous pouvons dire que les régimes permanents sont similaires à l'exception du couple électromoteur T_e. Les régimes transitoires évoluent différemment mais présente la même enveloppe. La simulation DEVS permet donc d'aboutir aux mêmes résultats que *Simulink*.

BIBLIOGRAPHIE

[1] B. Zeigler, Theory of Modeling and Simulation, New York: Wiley Interscience, 1976.

[2] B. Zeigler, T. G. Kim et H. Praehofer, Theory of Modeling and Simulation (second ed.), New York: Academic Press, 2000.

[3] P. A. Fishwick, *Hierarchical reasoning: simulating complex processes over multiple levels of abstraction,* 1986.

[4] S. Alexander, K. Frank et U. Adelinde M., «Modeling agents and their environment in multi-level-DEVS,» chez *Winter Simulation Conference*, 2012.

[5] S. Mittal, «Emergence in stigmergic and complex adaptive systems: A formal discrete event systems perspective,» *Cognitive Systems Research,,* vol. 21, pp. 22-39, 2013.

[6] E. Kofman et S. Junco, «Quantized State Systems. A DEVS Approach for Continuous System Simulation,» *Transactions of SCS,* vol. 3, n° %118, pp. 123-132, 2000.

[7] E. Kofman, «A Second Order Approximation for DEVS Simulation of Continuous Systems,» *Simulation: Transactions of the Society for Modeling and Simulation International,* vol. 2, n° %178, pp. 76-89, 2002.

[8] E. Kofman, « A Third Order Discrete Event Simulation Method for Continuous System Simulation.,» *Latin American Applied Research,* vol. 2, n° %136, pp. 101-108, 2006.

[9] G. Migoni, M. Bortolotto, E. Kofman et a. F. Cellier, «Linearly Implicit Quantization-Based Integration Methods for Stiff Ordinary Differential Equations,» *Simulation Modelling Practice and Theory,* 2013.

[10] F. J. Barros, «Dynamic Structure Discrete Event System Specification: A new Formalism for Dynamic Structure Modeling and Simulation,» *Winter Simulation Conference,* pp. 781-785, 1995.

[11] C. L., B. F., F. D. et P. Bisgambiglia, «BFS-DEVS: A General DEVS-Based Formalism For Behavioral Fault Simulation,» *Elsevier Simulation Pratice and Theory,* vol. 14, pp. 945-970, 2006.

[12] T. S., C. L. et C. G.A., «Wound Rotor Induction Generator Inter-Turn Short-Circuits Diagnosis Using a New Digital Neural Network,» *EEE Transactions on Industrial Electronics,* vol. 9,

2013.

[13] Gathmann, «Python as a discrete event simulation environment,» *Seventh International Python Conference,* 1998.

[14] C. Bin, Z. Lao-bing, L. Xiao-cheng et V. Hans, «Activity-based simulation using DEVS: increasing performance by an activity model in parallel DEVS simulation,» *Journal of Zhejiang University SCIENCE C,* vol. 15, n° %11, pp. 13-30, 2014.

[15]]. A. Yazidi, H. Henao, G. Capolino, D. Casadei et F. Filippetti., «Double-fed three-phase induction machine abc model for simulation and control purposes,» *In Proceedings of IEEE Industrial Electronics Conference (IECON'05),* vol. 4, p. 2560–2565, 2005.

[16] F. Cellier et E. Kofman, Continuous System Simulation, Secaucus, NJ, USA: Springer-Verlag New York, 2006.

[17] G. Migoni et E. Kofman, «Linearly Implicit Discrete Event Methods for Stiff ODEs,» *Latin American Applied Research (LAAR Journal),* 2009.