République Algérienne Démocratique Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université d'Ibn Khaldoun – Tiaret

Faculté des Mathématiques et de l'Informatique

Département Informatique

Thème

JDS (JAVA DEVS Simulator) un outil pour la modélisation et de simulation à évènements discrets

Pour l'obtention du diplôme de Master II

Spécialité : Génie Informatique

Option : Système d'information et technologie web

Réalisé par : Kacher Abdelhafidh.

Dirigé par : M. Mostefaoui Kadda.

Année universitaire 2015-2016

Dédicace

A mes parents. Que dieu leur procure bonne santé et longue vie.

A la lumière de mes jours, la source de mes efforts, la flamme de Mon cœur, ma vie et mon bonheur ; MAMAN que j'adore.

A celui que j'aime beaucoup et qui m'a soutenu tout au long de Ce projet et bien sûr A mes amis

Karim, Amine, Hbib, Oussama, Yacine, Makhlouf, Fateh, Sabri, Kady, ...

Sans oublié mes grands-parents, mes tantes, mes oncles, mes cousins et mes cousines.

A toute ma famille, mes amis.

Et à tous ceux qui ont contribué de près ou de loin pour que ce

Projet soit possible, je vous dis merci.

HAFID

Remerciements

Nous tenons tout d'abord à remercier le bon Dieu de nous avoir guidé et donné la force et la volonté pour atteindre notre objectif.

Nous remercions nos très chers parents pour leur soutien et leur patience.

Nous tenons à exprimer nos vives gratitudes et notre profonde reconnaissance à notre encadreur Mr. Mostefaoui kadda pour ses précieux conseils qui nous ont gardés sur le droit chemin afin de réaliser ce modeste travail.

Nous remercions tous nos enseignants depuis l'école primaire jusqu'à l'université, qui ont contribué à notre formation, auxquels nous exprimons notre plus grand respect et profonde reconnaissance.

Nous plus vifs remerciements aux membres de jury qui nous font l'honneur de présider et d'examiner ce modeste travail.

Egalement, nos remerciements à tous ceux qui nous ont aidés de près ou de loin dans la réalisation de ce projet de fin de cycle.

KACHER Abdelhafidh

Résumé

Les outils de modélisation et de simulation sont des outils incontournable permettent aux utilisateurs de modéliser et simuler les processus complexes sur un ordinateur. Le travail faisant l'objet de ce mémoire consiste à la mise en œuvre d'un outil de modélisation et simulation baptisé JDS (Java DEVS Simulator) basé sur les évènements discrets dont le cœur de simulation se base sur le formalisme DEVS (Discrete EVent system Specification), et permet la modélisation et la simulation visuelle des systèmes complexes à évènements discrets résolubles par la simulation à événement discrets.

Mots-clés : DEVS; Systèmes complexes; Modélisation; Simulation; Evènements discret ; Java.

Sommaire

Liste des figures

Introduction Générale

La modélisation et la simulation sont les outils les plus puissant du domaine de l'informatique qui sont devenus une nécessité pour les études des systèmes complexes même dans des disciplines variées pouvant être très éloignées de l'informatique (l'écologie, la biologie, la mécanique...).

Le formalisme DEVS [6] (Discrete Event System Specification) fournit une méthodologie pour la construction de modèles de simulation modulaires, hiérarchiques et réutilisables dans le but de simuler des systèmes dynamiques complexes. Ce formalisme de modélisation et de simulation, issu des mathématiques discrètes, permet de modéliser des systèmes compliques dans une très large variété de domaine. Il est basé sur les évènements discrets pour la modélisation de systèmes discrets. Le modèle est vu comme un réseau d'interconnexions entre des modèles atomiques et couples formant une hiérarchie de modèles. Les modèles sont en interaction via l'échange d'évènements estampillés.

L'objectif principal de notre travail est la conception et l'implémentation d'un environnement de modélisation et de simulation libre dont le cœur de simulation se base sur le formalisme DEVS. Cet outil destiné à modéliser et à simuler les systèmes complexes est une interface graphique implémentée en langage Java permettant la modélisation et la simulation des systèmes complexes décrits dans le formalisme DEVS. Ce document s'articule en quatre chapitres :

- Le premier chapitre présente les concepts clés. Parmi ces concepts, nous détaillerons plus particulièrement ceux de système, modèle et simulation, et on approfondit dans le formalisme DEVS et dans ses extensions.
- Le second chapitre fait une revue des principales implémentations logicielles existantes dans le domaine de la modélisation et de la simulation de systèmes complexes basé sur DEVS.
- Le troisième chapitre présente la partie modélisation de notre système, composée des diagrammes des cas d'utilisation, de séquences et d'activité, et la représentation de l'architecture par le diagramme de package et les diagrammes des classes.
- Le chapitre quatre est consacré à l'implémentation et la validation de notre outil de modélisation et de simulation JDS (Java DEVS Simulator).

Chapitre I

Introduction à la méthodologie de modélisation et de simulation DEVS

Introduction

Dans ce chapitre, la première partie présente les concepts clés qui fondent la méthodologie de modélisation et de simulation. Parmi ces concepts, nous détaillerons plus particulièrement ceux de système, modèle et simulation. Nous porterons également notre attention sur les étapes de la construction d'un modèle et ce, sous les deux aspects complémentaires de la théorie de la modélisation et de la simulation que constituent, les niveaux et les formalismes de spécification d'un système.

La seconde partie sera consacrée aux méthodes de multi-modélisation qui nous permettent de définir un environnement général de modélisation de systèmes dynamiques complexes. L'exposé de ces méthodes montrera qu'un cadre général unifié est nécessaire pour l'étude de ces systèmes.

I. La modélisation et la simulation

La modélisation est l'établissement d'un modèle pour répondre à des questions, et pour une définition plus complète des modèles, nous pouvons nous référer à celle de Minsky « *Pour un opérateur O, un objet M est un modèle d'un objet A si O peut utiliser M pour répondre à des questions de A.*» [1].

1. La modélisation des systèmes

La théorie de la modélisation et de la simulation s'est basée sur la théorie générale des systèmes [2], pour proposer des formalismes et des méthodes permettant de décrire des systèmes.

La théorie générale des systèmes distingue la structure (constitution interne d'un système) du comportement (ses manifestations externes). Elle définit les modèles comme causaux (dont les sorties sont la conséquence d'une entrée) et déterministes (à une entrée donnée ne peut correspondre qu'une seule sortie). Cette théorie sert de base à bon nombre de formalismes (les équations différentielles, les automates à états finis, les réseaux de Petri ...). Sa forme générale est pour un système $\mathbf{A \equiv < \tau, X, \Omega, S, Y, \delta, \lambda >}$ où :

- $\boldsymbol{\tau}$: base de temps, est la formalisation de la variable indépendante de temps;

- **X** : ensemble des états d'entrée, constitue toutes les activations d'entrées possibles pour le système, chaque entrée étant associée à un intervalle de temps;
- La dynamique de ce système est décrite dans la fonction de transition $\boldsymbol{\Omega}\ (\tau) \rightarrow \chi$ qui applique les états d'entrée courants à l'état courant pour transiter vers un nouvel état;
- **S** : représente l'ensemble des états que peut prendre le modèle;
- $\boldsymbol{\delta}\ (\Omega \times S) \rightarrow S$: fonction, de transition (fait évoluer l'état du modèle en fonction d'activations);
- **Y** : ensemble des états de sortie;
- Le système peut générer une sortie obtenue en appliquant la fonction de sortie $\boldsymbol{\lambda}\ (S) \rightarrow Y$ à partir de l'état courant. fonction de sortie.

Les fonctions de transition et de sortie sont activées pour étudier l'évolution d'un modèle pendant la simulation, Grâce à la théorie des systèmes, les algorithmes de simulation développés pour une structure de modèle peuvent être utilisés pour tous les modèles utilisant cette structure.

Lorsque l'on veut représenter un système, 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.

2. La simulation

La simulation consiste à étudier l'évolution de l'état d'un modèle à travers le temps. Il est important de savoir si le temps et les états représentant le système sont considérés comme discrets ou continus. Selon que les états du système sont spécifiés de manière dénombrable ou non dans un modèle, on parle de simulation discrète ou continue. La figure 1 présente la simulation continue de l'évolution d'un modèle à travers le temps, ce type d'analyse nécessite une description de type mathématique analytique du modèle car il doit être possible de donner l'état du système en tout temps. Cette méthode de simulation s'applique très bien pour des

systèmes dont les états évoluent de manière continue dans le temps et dont les états doivent être observables à n'importe quel instant.

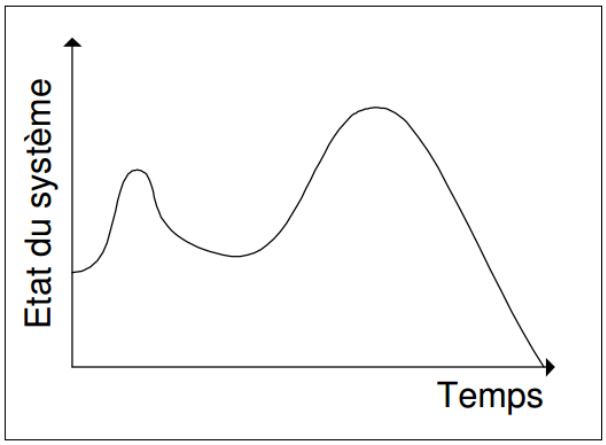

Figure 1 : Simulation continue.

La complexité de tels modèles grandit toutefois avec le nombre de paramètres et il devient rapidement impossible de modéliser des systèmes complexes de manière purement analytique. Il est alors nécessaire de décrire ces systèmes dans des approches de simulation discrète. Dans une approche de simulation discrète, l'état futur du modèle dépend de son état actuel. Les méthodes de simulation discrètes correspondent aux méthodes de résolution d'équations différentielles de type Euler ou Runge-Kutta [3].

La figure 2 représente une simulation discrète. La simulation de ce type de modèles implique que les changements d'états s'effectuent de manière discrète dans le temps. Cette méthode de simulation s'applique très bien pour des systèmes qui 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 où ils changent d'état.

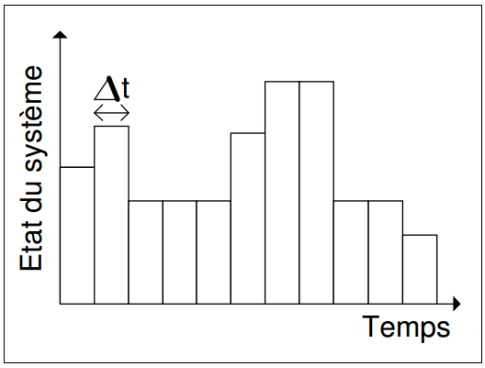

Figure 2 : Simulation discrète, dirigée par horloge.

Il existe plusieurs façons de gérer le temps en simulation discrète. On dit ainsi que la simulation est :

- **Dirigée par une horloge:** lorsque l'état du modèle est réévalué à intervalles réguliers. Dans la figure 2, un intervalle Δt sépare deux transitions d'états.
- **Dirigée par les événements:** lorsque des événements arrivant à des intervalles de temps irréguliers déclenchent les transitions d'états (la figure 3), la simulation est alors à événements discrets.

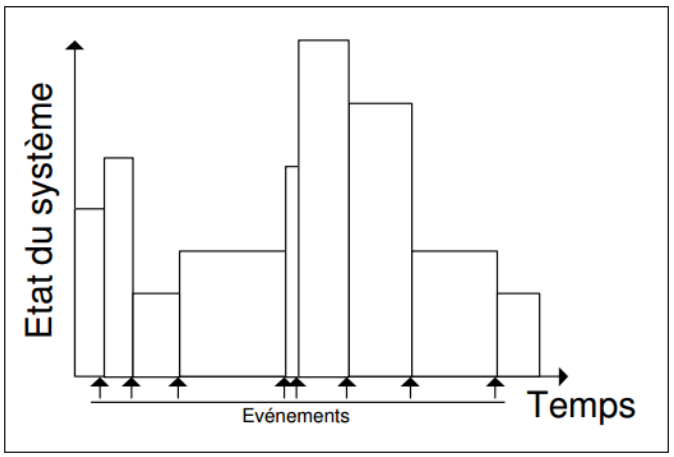

Figure 3 : Simulation à événements discrets.

Les méthodologies de simulation à événements discrets présentent de nombreux avantages sur la simulation dirigée par horloge. En effet, il est possible de simuler un modèle en temps discret grâce aux événements discrets en programmant des événements d'activation à intervalles réguliers. L'intérêt de l'utilisation de la simulation à événements discrets apparaît lorsque le phénomène simulé utilise des échelles de temps très différentes (de l'ordre de la seconde pour une partie du modèle et de l'année pour une autre) ; dans ce cas, si la simulation est dirigée par une horloge, la règle veut que le pas de temps utilisé soit celui du modèle utilisant la plus petite échelle de temps, même si le sous-modèle n'est actif que pendant une petite partie du temps complet de la simulation. Les événements discrets permettent de ne pas réévaluer l'état du modèle lorsque ce n'est pas jugé nécessaire. Plusieurs travaux de Kofman et S. [4] et de Zeigler [5] portent ainsi sur des méthodes de quantification en états de systèmes continus où la résolution de modèle ne se fait plus à intervalles de temps fixes, mais où des événements sont générés lorsque le système dépasse un seuil comme nous l'illustrons dans la figure 4.

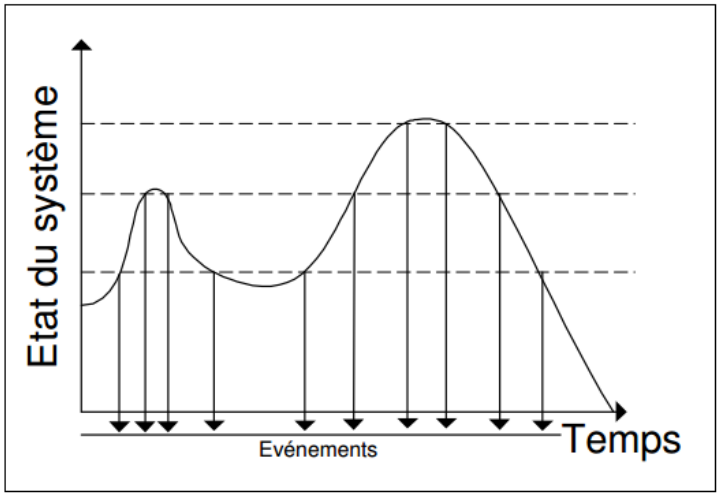

Figure 4 : Quantification de système continu.

Malgré ces cadres de spécification de modèles, bien définis en théorie des systèmes, il est encore impossible d'obtenir un modèle simulable à partir de spécifications informelles d'un système. Des méthodologies ont donc été développées pour simplifier cette procédure.

3. Le formalisme DEVS

Le formalisme DEVS (Discrete EVent system Specification) [6] a été introduit à la fin des années 70 par le professeur B.P Zeigler qui avait pour objectif de donner une base mathématique solide à la modélisation et la simulation 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.

Le professeur Zeigler a proposé une architecture conceptuelle pour la modélisation et la simulation [7], des systèmes particulièrement adaptés au formalisme DEVS. Comme la Figure 5 montre, 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 et 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 par 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.

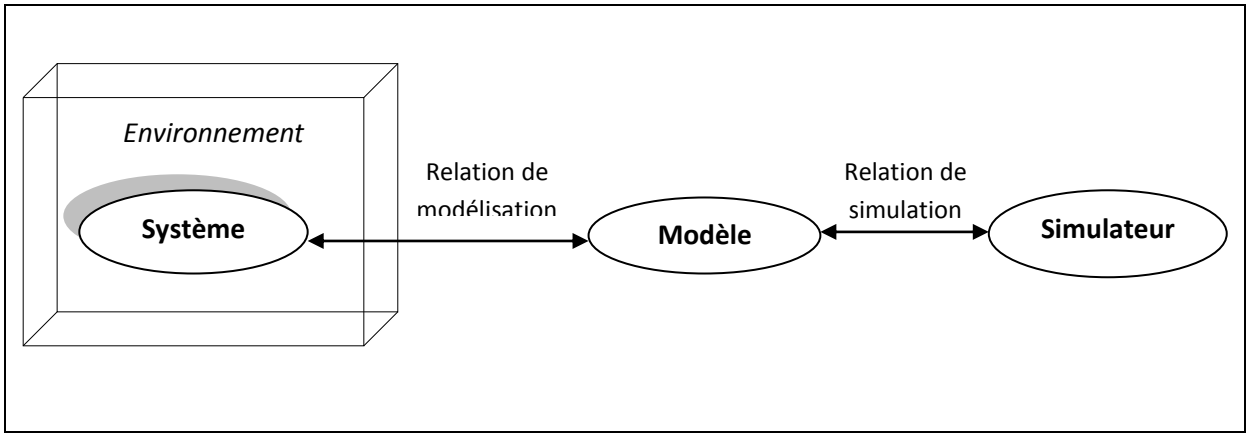

Figure 5 : Concepts de modélisation et simulation

Zeigler à utiliser une approche 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. 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.

3.1 Les modèles atomiques :

Dans le formalisme DEVS classique avec port, un modèle atomique est spécifié par la définition de sept entités : MA ≡ **< X, Y, S, δ_{int}, δ_{ext}, λ, t_a >** Avec :

- **l'ensemble des entrées: X** = { (p , v) | (p ∈ P_e , v ∈ V_x) } où 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 ∈ P_s, v ∈ V_y)} où 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 ∈ R^+)} L'ensemble des états séquentiels internes;
- **La fonction de transition interne: δ_{int}** (S) → S qui place le modèle dans l'état suivant après le temps renvoyé par la fonction d'avancement du temps;
- **La fonction d'avancement du temps: t_a** (S) → R^+ qui renvoie le temps de vie de l'état courant (temps jusqu'à la prochaine transition interne);
- **La fonction de transition externe: δ_{ext}** (Q×X) → S qui programme les changements d'états en fonction d'activations d'entrées;
- **La fonction de sortie: λ** (S) →Y qui génère des événements de sortie juste avant la transition interne.

La fonction (**t_a** (S)) est invoquée au début de la 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) = +∞ alors la durée de vie de l'état s_i est infinie et si aucun événement extérieur est intercepté, le modèle ne changera plus jamais d'état.

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 (**δ_{int}** (S)) et

quelle sortie sera générée après un tel changement d'état ($\boldsymbol{\lambda}$ (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 ($\boldsymbol{\delta_{ext}}$ (Q×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** ($Q = \{(s, e) \mid (s \in S, 0<e<t_a(s)\}$). 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 [7]) prévoit une autre fonction de transition supplémentaire qui permet de résoudre un tel conflit ($\boldsymbol{\delta_{conv}}$ ($S \times X^b$) avec $\boldsymbol{\delta_{conv}}$ ($S \times \emptyset$) = $\boldsymbol{\delta_{int}}$ (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. La fonction $\boldsymbol{\delta_{conv}}$ 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 $\boldsymbol{\delta_{int}}$ (S) → $\mathbf{t_a}$ (S) → $\boldsymbol{\lambda}$ (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 sa place $\boldsymbol{\delta_{ext}}$(X×S) →$\mathbf{t_a}$ (S) → $\boldsymbol{\lambda}$ (S) → $\boldsymbol{\delta_{int}}$ (S) → $\mathbf{t_a}$ (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 $\boldsymbol{\delta_{ext}}$ (X×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 $\boldsymbol{\lambda}$ (S). Ce type de modèle peut être qualifié de collecteur d'évènements.

La figure 6 montre un exemple d'un modèle atomique nommé **MA** qui représente un système qui devient actif ($\mathbf{s_1}$) lorsqu'il reçoit des entrées $\mathbf{V_x}$ sur un port d'entrée **P1**, qui traite cette donnée durant un temps (**proc**) puis qui génère des sorties $\mathbf{V_y}$ sur un port de sortie **P2**. 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 ($\mathbf{s_2}$). Le système est dans un état initial passif.

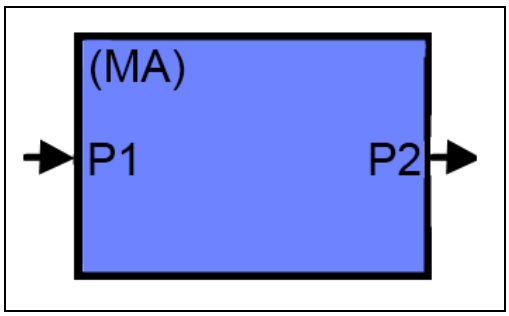

Figure 6 : Modèle atomique MA

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

- $\mathbf{X} = \{(P1, v_x) \mid (P1 \in P_e, v_x \in V_x \text{ et } x \in R^+)\}$;
- $\mathbf{Y} = \{ (P2, v_y) \mid (P2 \in P_s, v_y \in V_y \text{ et } y \in R^+)) \}$;
- $\mathbf{S} = \{ s_1, s_2 \}$;
- $\boldsymbol{\delta_{int}}$ (S): *si* $\mathbf{s = s_1}$ *alors* $\mathbf{s = s_2}$ pendant $\mathbf{t_a(s_1) = \infty}$; *sinon* **s** ne change pas.
- $\boldsymbol{\delta_{ext}}$ (Q×X): *si* $\mathbf{s = s_2}$ *alors* $\mathbf{s = s1}$ pendant $\mathbf{t_a(s_1) = proc}$; *sinon* $\mathbf{t_a(s_1) = t_a(s_1) - e}$.
- $\boldsymbol{\lambda}$ (S) : *si* $\mathbf{s = s_1}$ *alors* envoie d'un message de sortie avec la valeur $\mathbf{v_y}$;
- $t_a(s)$: *si* $\mathbf{s = s_1}$ *alors* retourner **proc**; sinon retourner l'infinie(∞).

La figure 7 montre la trajectoire d'états du modèle atomique **MA**, cette trajectoire permet de tracer les changements d'état en fonction des événements d'entrées.

Lorsqu'un événement intervient au temps $\mathbf{t_1}$, le système passe de l'état passif $\mathbf{s_2}$ vers l'état actif $\mathbf{s_1}$ pendant un temps **proc** (Figure 7). Lorsque le temps $\mathbf{t_1+proc}$ est écoulé, le modèle génère un événement de sortie. L'événement en entrée intervenant au temps $\mathbf{t_3}$ ne change pas l'état du système ($\mathbf{s_1}$) mais la durée de vie de ce dernier est mis à jour en fonction de **e**.

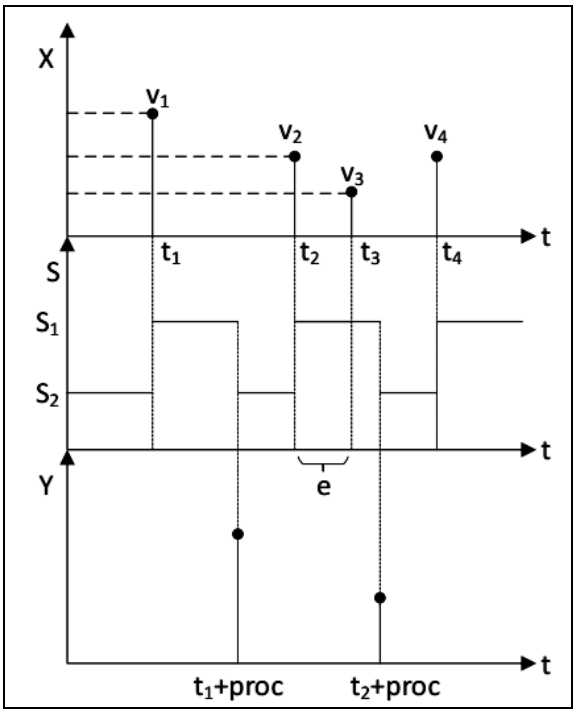

Figure 7 : La trajectoire d'états du modèle MA

3.2 Les modèles couplés :

Dans le formalisme DEVS classique avec port, un modèle couplé est spécifié par la définition de sept entités **MC ≡ < X, Y, D, EOC, IC, EIC, select >** avec **:**

- **l'ensemble des entrées: X** = {(p, v) | p ∈ $\mathbf{P_e}$, v ∈ $\mathbf{V_x}$ } où $\mathbf{P_e}$ et $\mathbf{V_x}$ 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 ∈ $\mathbf{P_s}$, v ∈ $\mathbf{V_y}$ } où $\mathbf{P_s}$ et $\mathbf{V_y}$ 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 ∈ $\mathbf{R^+}$ };
- **l'ensemble des couplages de sortie: EOC** = { $\big((m_i , p_{sj}) , (MC , p_{sk})\big)$ | i,j,k ∈ $\mathbf{R^+}$ };
- **l'ensemble des couplages internes** : **IC** ={ $\big((m_i , p_{sj}) , (m_k , p_{el})\big)$ | i , j , k , l ∈ $\mathbf{R^+}$ };
- **l'ensemble des couplages d'entrée: EIC** = { $\big((MC , p_{ei}) , (m_j , p_{ek})\big)$ | i , j , k ∈ $\mathbf{R^+}$ };
- **la fonction de sélection** permettant de régler le conflit d'activation des modèles (**select**(D) → 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.

Dans un modèle couplé, un port de sortie d'un modèle **m1** ∈ **D** peut être connecté à l'entrée d'un autre **m2** ∈ **D** mais pas directement à lui-même. Modèles couplés comme atomiques sont autonomes et peuvent être stockés séparément. Suivant les implémentations, la structure interne d'un modèle couplé peut être cachée pour créer des composants de plus haut niveau.

La figure 8 montre une représentation graphique d'un modèle couplé (**MC1**) composé de trois modèles atomiques (**MA1**, **MA2** et **MA3**) et d'un autre modèle couplé (**MC2**) composé de deux modèles atomiques (**MA4** et **MA5**). Afin de simplifier l'exemple, chacun de ces modèle a un seul port d'entrée **Pe1** et un seul port de sortie **Ps1**.

Le modèle couplé **MC1** est spécifié de la manière suivante dans DEVS :

- **X** = $\{(Pe1, v_x) \mid (Pe1 \in P_e, v_x \in V_x \text{ et } x \in R^+)\}$;
- **Y** = $\{(Ps1, v_y) \mid (Ps1 \in P_s, v_y \in V_y \text{ et } y \in R^+)\}$;
- **D** = {MA1, MA2, MA3, MA4, MA5, MC2};
- **EOC** = $\{((MA2, Ps1), (MC1, Ps1)), ((MA3, Ps1), (MC1, Ps1))\}$;
- **EI** = $\{((MA1, Ps1), (MA3, Pe1)), ((MC2, Ps1), (MA2, Pe1))\}$;
- **EIC** = $\{((MC1, Pe1), (MC2, Pe1)), ((MC1, Pe1), (MA1, Pe1))\}$;

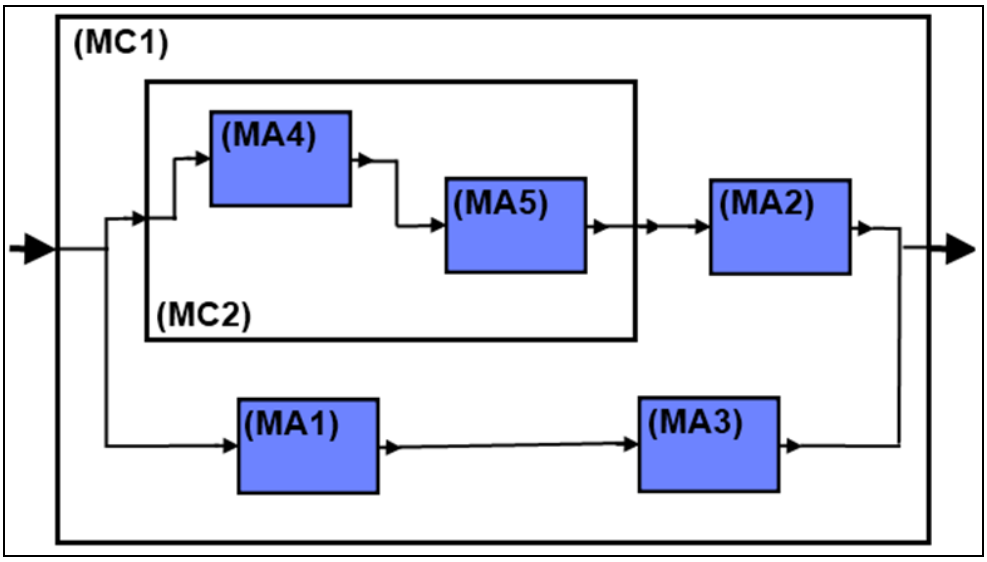

Figure 8 : Modèle couplé MC1

Le formalisme DEVS présente une séparation explicite entre la modélisation et la simulation, un modèle DEVS est directement simulable dans le contexte d'un cadre d'expérimentation donné. La simulation de modèles DEVS est dirigée par les événements qui activent les transitions d'états des modèles. La figure 9 présente le fonctionnement d'un modèle atomique en rapport à ces activations.

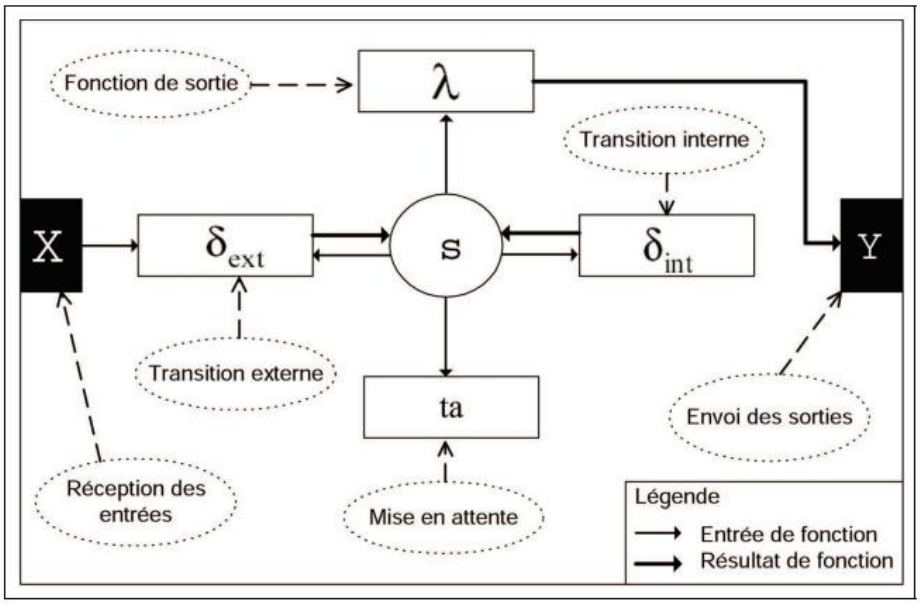

Figure 9 : Fonctionnement d'un modèle atomique

L'une des caractéristiques les plus importantes de DEVS est que des modèles très complexes peuvent être simulés d'une manière très simple et efficace. La figure 10 montre l'architecture du simulateur DEVS, où chaque classe **simulator** (simulateur) est associée à

chaque modèle atomique et chaque classe **coordinator** (coordonnateur) est associée à chaque modèle couplé. Les simulateurs et les coordonnateurs des couches hiérarchiques communiquent entre eux avec des messages. Les coordinateurs envoient des messages à leurs fils, alors ils exécutent les fonctions de transition. Quand un simulateur exécute une transition, il calcule son prochain état (Lorsque la transition est interne) et il envoie la valeur de sortie à son coordonnateur père. Lorsqu'un coordinateur exécute une transition, il envoie des messages à certains de ses fils afin qu'ils exécutent leurs fonctions de transition correspondantes. Lorsqu'un événement de sortie produit par l'un de ses fils doit être propagé à l'extérieur du modèle couplé, le coordinateur envoie un message portant la valeur de sortie à son propre coordonnateur père.

Chaque simulateur ou coordinateur possède une variable locale (**tn**) qui indique le temps de la prochaine transition interne se produira. Dans les simulateurs, cette variable est calculée en utilisant la fonction d'avancement de temps du modèle atomique correspondant, et dans les coordinateurs est calculée comme le minimum **tn** de leurs fils. Ensuite, le coordinateur racine (coordinator-root) vérifier seulement ce temps et avance le temps global (**t**) à cette valeur puis il envoie un message à son fils pour effectue la transition suivante, puis il répète ce cycle jusqu'à la fin de la simulation.

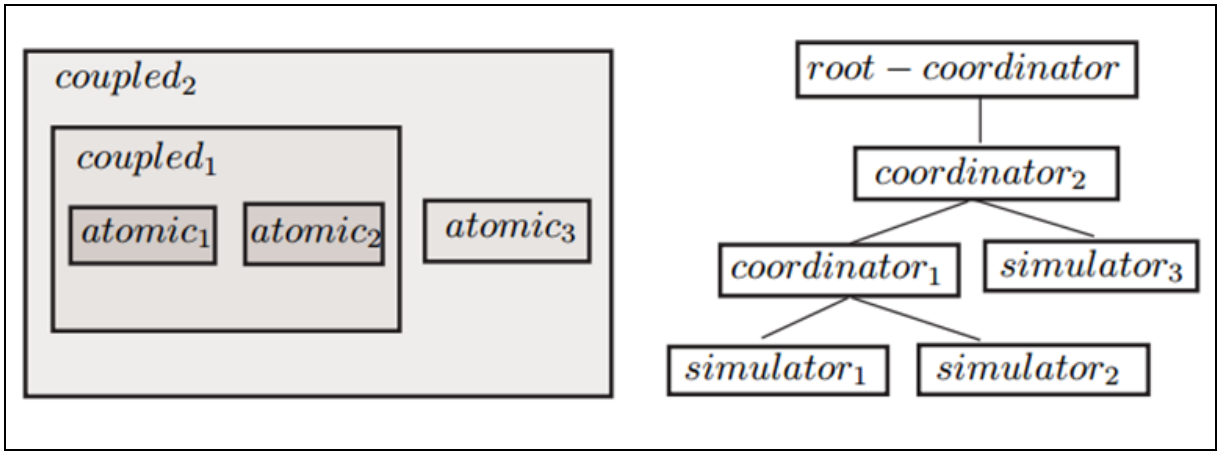

Figure 10 : Fonctionnement d'un modèle atomique

4. Les Extensions de DEVS

4.1 DEVS parallèle :

Dans cette version, la notion d'Event-Bag a été ajoutée. Cette notion intègre le fait que plusieurs événements intervenant au même instant peuvent être regroupés dans un bag. L'ensemble est noté $\mathbf{X^b}$. De la même manière, un modèle atomique peut envoyer plusieurs événements en sortie au même instant. L'ensemble de sortie est noté alors $\mathbf{Y^b}$. La fonction $\boldsymbol{\delta_{conv}}$ $(S\times X^b)$ 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 où $\boldsymbol{\delta_{conv}}(S\times\emptyset)=\boldsymbol{\delta_{int}}(S)$. L'association de ces deux notions (bag et $\boldsymbol{\delta_{conv}}$) 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.

4.2 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 [7] exploitée ensuite par le chercheur E. Kofman [8] 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 [9] ou 3 [10] 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 [11].

4.3 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 [12]. 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.

4.4 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 [13]. 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 appliquée dans le domaine des réseaux de neurones [14].

II. La multi-modélisation

La simulation d'un modèle est effectuée si une description mathématique suffisante du système ne peut être fournie pour répondre aux questions que l'on se pose d'une manière analytique. Nous avons vu que le formalisme DEVS et sa sémantique étaient adaptés pour définir un environnement de modélisation et de simulation hiérarchique dans lequel les modèles peuvent être facilement agrégés et décomposés tout en garantissant leur intégrité. Aucun modèle n'étant adapté à tous les problèmes, il est évident qu'il est nécessaire de réaliser des compositions pour pouvoir répondre à des questions complexes. Pour cela Ören [15] a développé le concept de Multi-modélisation, plus tard étendu par Fishwick [16].

Un multi-modèle est un modèle composé de manière récursive d'autres modèles qui peuvent être de différents types. Nous définissons un type de modèle comme un ensemble de modèles utilisant une même technique de modélisation. La technique décrit une organisation et une conceptualisation particulière pour un modèle tandis qu'un formalisme définit une grammaire et éventuellement un processus de résolution pour décrire ces modèles.

Un multi-modèle présente plusieurs niveaux de description en généralisant ou spécialisant un système et ses parties de plus en plus à chaque niveau. Le plus haut niveau est constitué de boites noires, c.à.d. des composants dont la fonctionnalité est cachée à leur niveau et révélée à des niveaux plus profonds, donc plus le niveau est élevé moins il y a de composants. Le couplage interne d'un composant est défini au plus bas niveau, mais l'interface des composants est conservée pour permettre le couplage de composants de même niveau. Le niveau le plus bas est en général le seul niveau simulable. La figure 11 présente la décomposition hiérarchique du modèle de composition **C1** en modèles **A1**, **A2** et **C2**, et la décomposition du modèle **C2** en modèles **A3** et **A4**, ici seuls les modèles atomiques constituant le niveau le plus bas de la hiérarchie de description sont simulables.

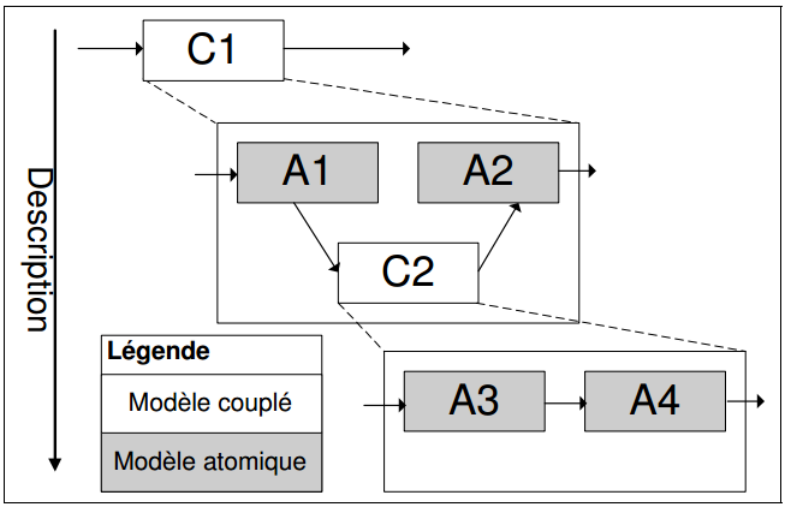

Figure 11 : Décomposition de systèmes hiérarchiques

Les multi-modèles présentent aussi plusieurs niveaux d'abstraction en généralisant un système et ses éléments à chaque niveau. Au plus bas niveau un composant dispose de toutes les propriétés prévues. A de plus hauts niveaux ses propriétés sont réduites et les informations concernant ces propriétés (comportement et états) sont passées à travers les niveaux.

Conclusion

L'examen liminaire des concepts régissant la description et l'utilisation de modèles de systèmes physiques complexes et de la nécessité du recours à la multi-modélisation, méthode nécessaire pour permettre la description de systèmes complexes, nous invitent à opter pour le formalisme DEVS dont nous avons montré la pertinence comme base unificatrice permettant la coopération de différents types de modèles dans un environnement de multi-modélisation.

Nous nous devons de présenter dans le chapitre qui suit, une revue représentative des implémentations et des outils de modélisation de systèmes complexes existants basés sur le formalisme DEVS.

Chapitre II

Les implémentations DEVS existantes

Introduction

Ce chapitre aborde les principales implémentations logicielles existantes dans le domaine de la modélisation et de la simulation de systèmes complexes. Nous évoquerons essentiellement les environnements de multi-modélisation basés sur le formalisme DEVS, Cette section présente une revue des environnements basés sur DEVS et ses architectures.

I. DEVSJAVA et DTE (DEVS Tracking Environment)

DEVSJAVA [17], développé à l'Université de l'Arizona, est un environnement écrit en Java, basé sur DEVS. L'environnement DEVSJAVA fournit les classes de base permettant d'étendre la méthodologie DEVS en utilisant les capacités du langage Java, est un outil utilisé pour la simulation parallèle des modèles de DEVS. Il donne une vue hiérarchique du modèle de la simulation en utilisant le style de compositions. La conception du DEVSJAVA sépare le contrôle de l'exécution du noyau du simulateur et l'affichage. La visualisation des modèles et ses animations sont supportés par le SimView (la fenêtre graphique de la visualisation de la simulation dans DEVSJAVA) qui supporte l'interaction avec l'utilisateur et le contrôleur de l'exécution de la simulation.

La figure 12 représente la structure hiérarchique des composants logiciels de DEVSJAVA [18]. Des partitions incluent une séparation entre le moteur de modélisation et de simulation (genDEVS.modeling et genDEVS.simulation) et l'interface graphique (SimView). Les classes du package genDEVS.modeling et leurs relations représentent une réalisation du modélisateur d'objets de DEVS, et les packages genDEVS.simulation réalise le simulateur abstrait de DEVS. Les classes dans SimView sont capables d'accéder et d'afficher des informations sur les modèles et leurs exécutions. A côté de ces packages, il existe un autre module important (GenCol), est basé sur l'API Java Collections, il fournit des composants de calcul qui permettent la manipulation et la spécification d'objets à des niveaux plus élevés de spécification.

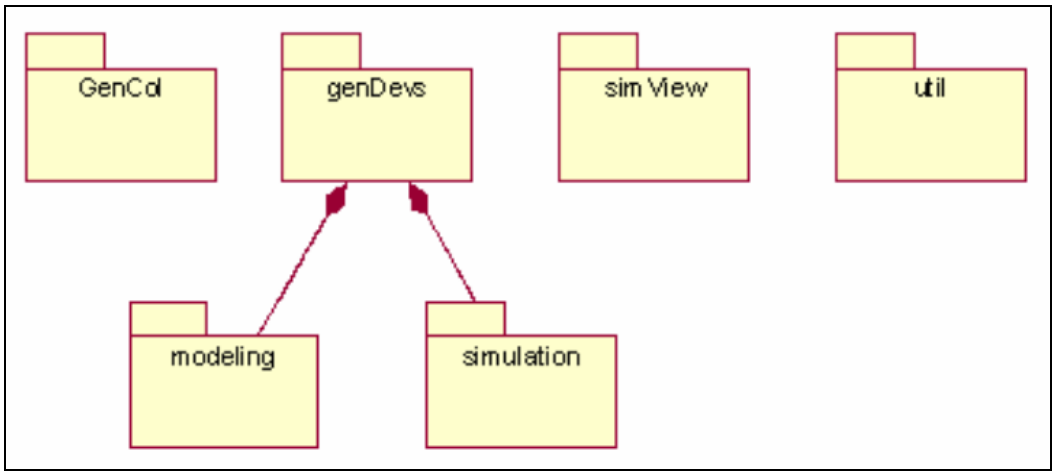

Figure 12: La Hiérarchie de packages de DEVSJAVA.

La figure 13 montre le SimView de DEVSJAVA. Dans la fenêtre principale le modèle couplé **HierarModel** a une entrée (**in**), et deux sorties (**outFinalC1** et **outFinalC2**) et il contient d'autres composants représentent d'autres modèles couplés (**HierarCoupledModel3** et **HierarCoupledModel2**). Le modèle **HierarCoupledModel3** contient un autre modèle couplé (**HierarCoupledModel1**) et un composant d'un modèle atomique (**third**). Dans le SimView, les composants atomiques sont représentés comme des boîtes pleins avec des ports d'entrée et de sorties. Le SimView peut également d'afficher certains aspects du comportement du modèle au cours d'une simulation. Comme illustré dans la figure 13, un bouton (**step**) pour avancer l'exécution de la simulation par un seul pas d'événement à la fois. Par contre, un bouton pour exécuter la simulation qui laisse les événements continuent d'exécuter l'un après l'autre sans l'intervention de l'utilisateur. Cela permet à l'utilisateur d'avancer plus rapidement la simulation.

Le DTE (DEVS Tracking Environment) [19] est développé pour supporter la conception automatisée en observant les entrées et les sorties, l'état de phase et sigma des modèles. Sa conception est basée sur l'architecture MVC (Model View Control) classique [20]. En plus des modules MVC, un module appelé la couche **Façade** est aussi utilisée. Les données disponibles dans le module Façade sont accessibles par le module **Control** (Contrôleur) et le module **View** (Vue). Avec l'architecture MFVC (Model Façade View Control), les ensembles des données de simulation peuvent être affichés par un ou plusieurs

Views. Les données sont générées par le module **Model** (Modèle) sont obtenu lors de l'exécution de la simulation et mis à la disposition du module **View**.

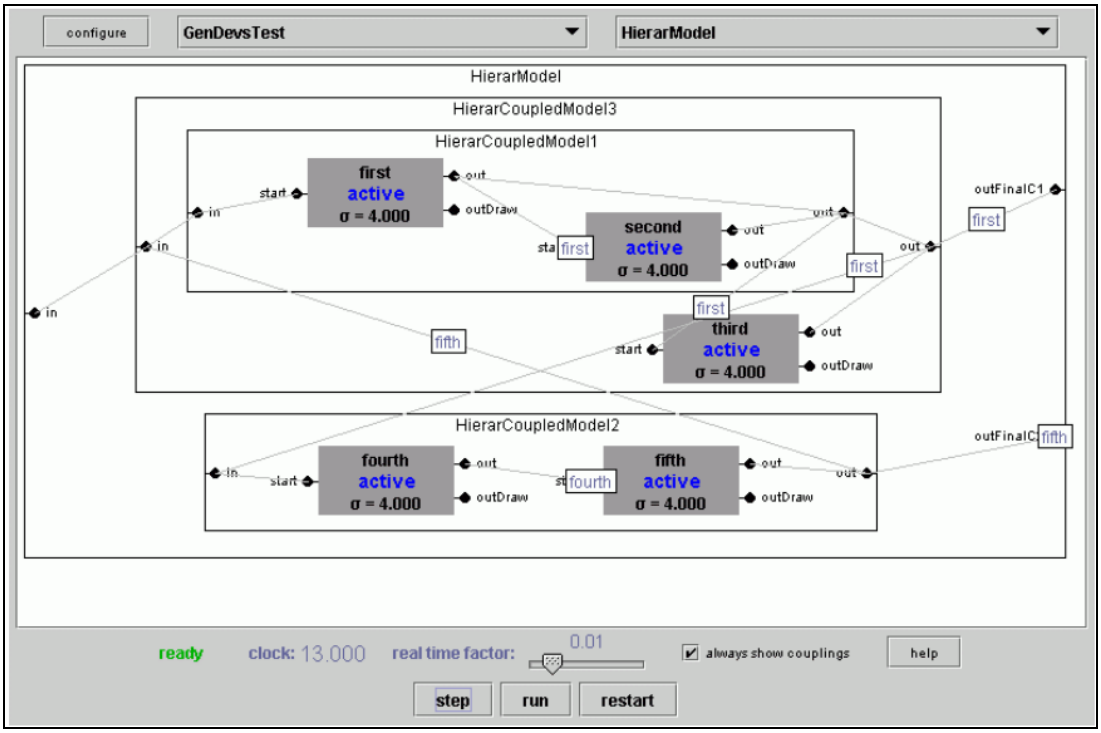

Figure 13: L'interface graphique SimView de DEVSJAVA

La fonctionnalité clé de DTE est la simplification de la conception de l'expérimentation pour la simulation du modèle, son interface graphique permet a l'utilisateur de sélectionner le composant du modèle pour être contrôlé et donc concevoir des expérimentations en terme d'entrées, de sorties et variables d'état de composants. Les ensembles de données de modèles de simulation qui incluent les états comme le temps de l'évènement suivant et le temps du dernier évènement peuvent être dynamiquement suivis. Donc l'utilisateur aura la possibilité d'observer les données de la simulation pour n'importe quel nombre de modèles atomique et couplés. Les données peuvent aussi être affichées dans TrackingLogger (l'enregistreur de la traque) et peuvent être exportés en fichiers CSV.

TimeView [21] est un module développé pour l'affichage de l'ensemble de données en temps réel dans un plan de deux dimensions. Pour tracer les données de simulation basés sur le temps, les coordonnés de **X** sont définis pour représenter le temps, ces valeurs sont entières et leur unité peut être alphanumérique (par exemple, seconde). Les coordonnés de **Y** peuvent

être des nombres ou caractères, ces valeurs sont générées comme des entrées ou des sorties qui sont générées par les modèles atomiques ou couplées.

Pour chaque composant d'un modèle couplé ou atomique, tous ces ports d'entrées et de sortie peuvent être affichés indépendamment. La trajectoire du temps pour tous les variables d'un modèle qui sont sélectionné pour être suivis sont combinées dans une seule visualisation.il n'y a pas de support pour superposant plusieurs variables dans un seul plan et plusieurs trajectoires de temps de plusieurs modèles ne peuvent pas être combinés dans une seule trajectoire.

II. DEVS-Suite

DEVS-Suite [22] est une application basée sur Java utilisée pour représenter les modèles DEVS et leurs interactions dans un environnement graphique interactif. DEVS-Suite est la nouvelle génération du simulateur DEVSJAVA, il supporte la conception visuelle de l'expérimentation et de visualisation des données générées par les modèles sélectionnées de façon dynamique et affichés dans la trajectoire basé sur le temps. Ces capacités complètes l'animation des composants des modèles de DEVS et leurs interactions.

DEVSJAVA et DTE sont utilisés pour le développement du simulateur DEVS-Suite, le moteur de simulation dans DEVS-Suite est le même utilisé dans DEVSJAVA et DTE. Comme il est mentionné dans la section précédente, le simulateur offre des fonctionnalités intégrées qu'ils peuvent aider les utilisateurs à concevoir expérimentations et de contrôler la simulation plus facilement.

L'architecture du simulateur DEVS-Suite [23] sépare les modèles de simulation de son contrôleur et sa visualisation. Comme illustré dans les Figures 14 et 15, le diagramme de package de DEVS-suite se compose de packages de **Model**, **Façade**, **Controller**, et **View** et leurs sous- packages [24]. La conception et l'implémentation de la couche de **Façade** est étendu de la DTE. Généralement, sa connexion avec le module **Model** est modifié afin d'inclure le **SimView** de DEVSJAVA au module **View** qui possède les packages de **SimView**, **TimeView**, et **TrackingLog**. Avec la présence du module **Façade**, **View** permet de combiner l'animation et la traque (l'affichage des données simulées en temps d'exécution). La conception du module **Controller** est étendue pour gérer la vitesse d'animation de la simulation.

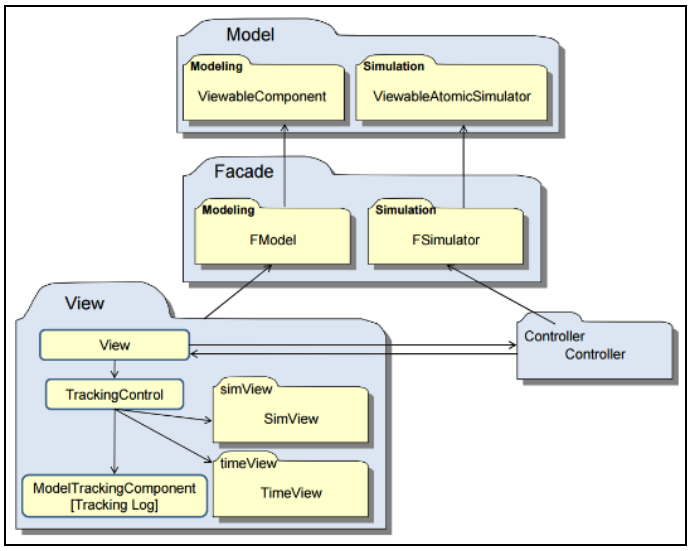

Figure 14: Diagramme de packages de MFVC de DEVS-Suite

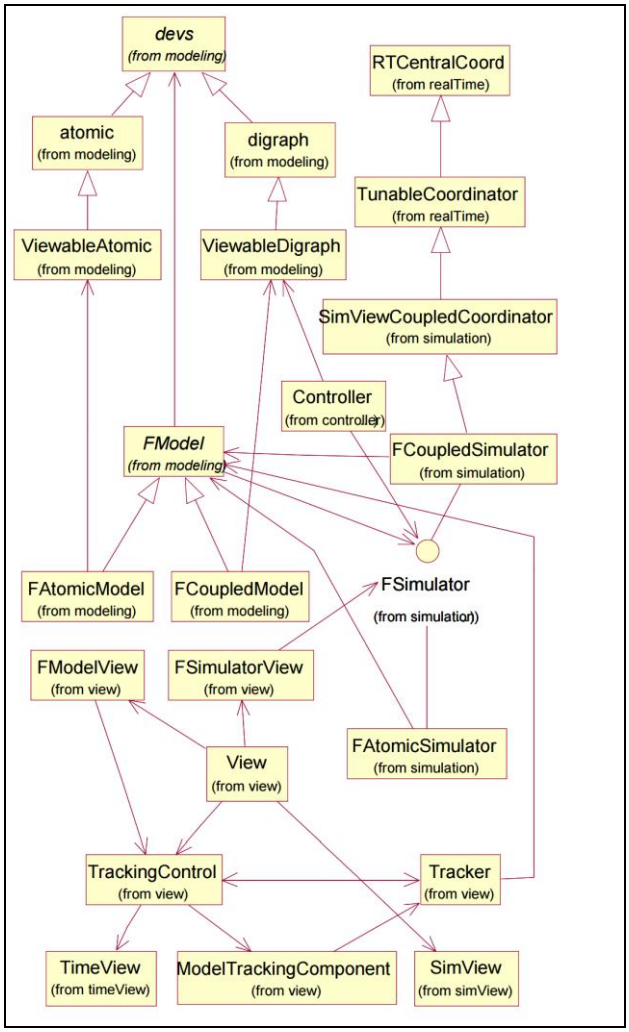

Figure 15: Diagramme de classes de DEVS-Suite

L'interface d'utilisateur de DEVS-Suite [25] contient quatre partie, **ModelViewer** dans coin haut gauche de l'écran, le contrôle du simulateur en bas gauche de l'écran, le **SimView** dans le coin bas droit et le **TimeView** dans le coin bas droit (la figure 16). Afin de faciliter l'utilisation de l'espace de l'affichage, le **ModelViewer** et **SimulatorControl** sont combinés dans une partie que nous appelons MVSC (ModelViewer-SimulatorControl). Donc l'utilisateur peut choisir d'afficher **TimeView**, **SimView**, ou bien le MVSC dans l'interface du DEVS-Suit puisque deux des trois parties peut être caché et aussi l'utilisateur a la possibilité d'afficher ces trois parties en même temps. Les bloques et arborescences de composants de modèles hiérarchiques sont disponibles. La vue d'arborescences est utilisée pour le choix des composants du modèle; pour les modèles atomiques, des variables d'état prédéfinis et des variables basiques du simulateur peuvent être choisis et traquées. La dynamique de chaque modèle atomique et couplé peut être affichée individuellement dans le **TimeView**. Les utilisateurs ont la flexibilité de sélectionner les options de l'animation et de la visualisation de traque pour n'importe quel nombre des modèles atomiques ou couplés, et ils peuvent mettre l'unité pour les données qui sont surveillées, ainsi que l'axe de temps. L'incrémentation du temps, unités et les données sélectionnées pour observation sont dans la figure 17.

Figure 16: L'interface d'utilisateur de DEVS-Suite

Le simulateur de DEVS-Suite supporte la logique de contrôle qui est fournis par DEVSJAVA et le DTE. L'exécution de la simulation peut être contrôlée en utilisant les boutons **Run**, **Step**, **Step(n)**, **Request Pause**, et **Reset** (la figure 16). Les autre contrôleurs sont **Show Coupling**, **real-time factor** et un nouveau contrôle appelé **Animation speed**. La logique de contrôle gère l'exécution de la simulation et les données qui sont fournis au **SimView**. DEVS-Suite gère l'animation du **SimView** et les trajectoires de **TimeView** indépendamment, c'est un avantage parce que la vitesse de l'animation des messages ne peut pas être la même où les trajectoires de temps peuvent être affichés.

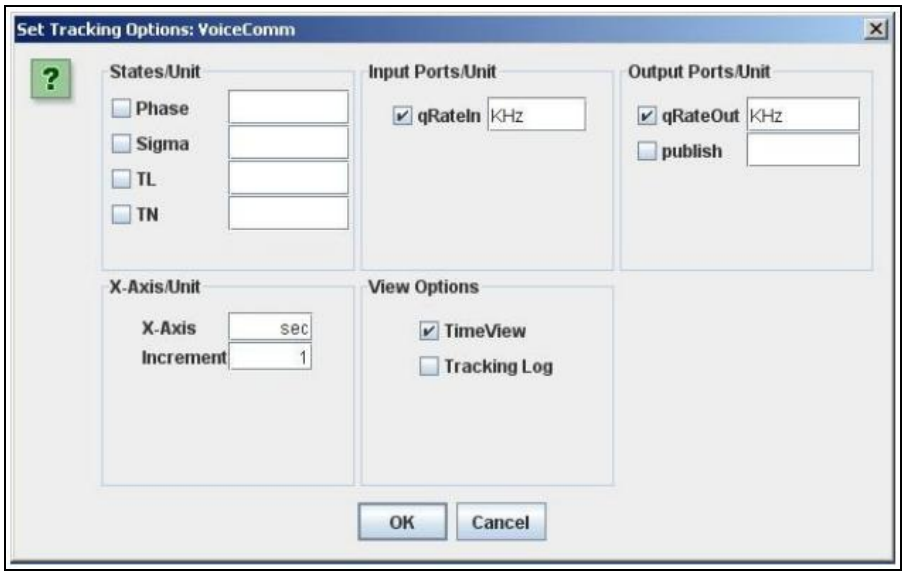

Figure 17: La fenêtre de Tracking dans DEVS-Suite

III. CD++

CD++ [26] est un outil de modélisation et de simulation, cet outil implémente les formalismes DEVS parallèle et Cell-DEVS. Cell-DEVS [27] est un environnement dédié à la modélisation de systèmes qui pouvant être représentés sous la forme d'espaces cellulaires actifs. Cell-DEVS se base sur le formalisme des automates cellulaires [28] qui est très utilisé pour décrire de tels types de modèles. Les automates cellulaires évoluent en exécutant une fonction de transition globale qui met à jour l'état de toutes les cellules. Le comportement de cette fonction globale dépend du résultat d'une fonction s'exécutant localement dans chacune des cellules, de manière synchrone et parallèle en utilisant l'état des cellules voisines. Chaque cellule est définie comme un modèle atomique utilisant des retards de synchronisation, et il peut être intégré à un modèle couplé représentant un espace cellulaire. Un modèle Cell-DEVS

atomique est spécifié par **AC ≡ < X , Y , I , S , N , delay , d , δ_{int} , δ_{ext} , λ , τ , D>,** où **S** est l'ensemble des états de la cellule ,**N** est un ensemble d'événements d'entrée, **I** est l'interface du modèle et **delay** (le retard) définit le type de retard pour la cellule, et **d** est sa durée. Il existe aussi plusieurs fonctions (**δ_{int}, δ_{ext}, λ, τ** pour les calculs locaux et **D** pour la fonction de la durée de l'état). Chaque cellule utilise les entrées N pour calculer l'état suivant, ils sont reçus par l'interface du modèle, et sont utilisées pour activer la fonction locale. Un retard peut être associé à chaque cellule. Une fois que le comportement d'une cellule est définie, ils peuvent être combinés en un modèle couplé **CC ≡ < Xlist , Ylist , I , X , Y , n , {t1…tn} , N, C, B, Z, λ>,** où **Xlist** est une liste de couplage d'entrée, **Ylist** est une liste de couplage de sortie, **n** définit la dimension de l'espace de la cellule, **{tl, …, tn}** est le nombre de cellules dans chaque dimension et est l'ensemble du voisinage, **C** est l'espace de la cellule, **B** est l'ensemble des cellules de bordure et la fonction **Z** définit les couplages internes et externes. Le modèle couplé est construit comme une matrice de cellules atomiques, chacun est connecté à son voisinage.

Ces spécifications ont été utilisées comme la base théorique pour développer CD++. Les définitions de DEVS et Cell-DEVS, ont été utilisés pour définir des modèles exécutables. CD++ a été construit comme une hiérarchie de classes créée en C++, où chaque classe correspond à une entité de simulation. Il existe deux classes abstraites de base, Model et Processor. La classe Model est utilisée pour représenter le comportement des modèles atomiques et couplés, et l'autre implémente les mécanismes de simulation. La Figure 18 montre la hiérarchie de classes de CD++.

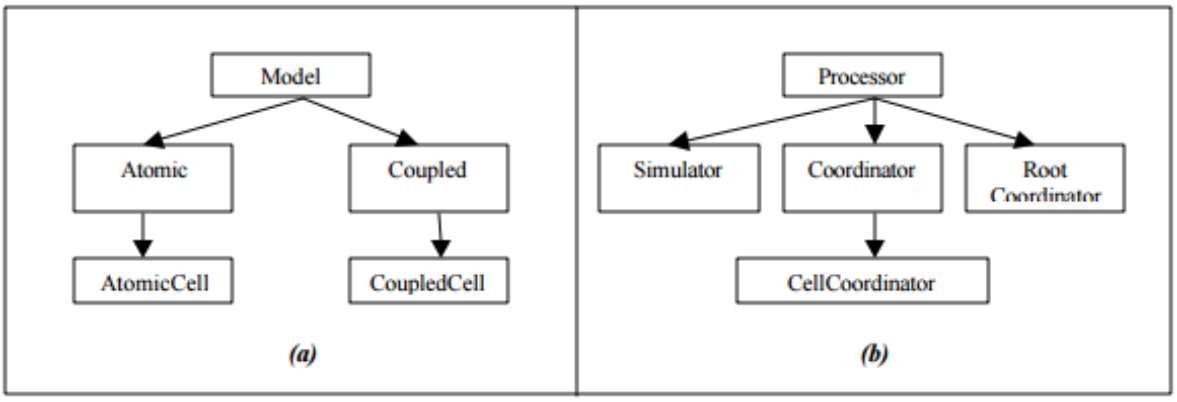

Figure 18: la hiérarchie de classes de CD++.

La classe **Atomic** implémente le comportement d'un composant atomique et la classe **Coupled** implémente les mécanismes d'un modèle couplé. Pour les modèles cellulaires, des modèles atomiques propres sont utilisés pour représenter les cellules. Pour ce faire, les classes **AtomicCell** et **CoupledCell** sont définis comme des sous-classes d'Atomic et **Coupled** respectivement, la classe **AtomicCell** est utilisée pour définir la fonctionnalité de l'espace cellulaire, et **CoupledCell** gère un groupe de cellules atomiques.

Un objet simulator gère un objet atomique associé et la manipulation de l'exécution de ses fonctions, Un objet **coordinator** gère un objet couplé associé. Un seul **root-coordinator** existe dans une simulation. Il gère les aspects globaux de la simulation. En plus, le **root-coordinator** lance et arrête le processus de simulation, et il reçoit les résultats de sortie qui doivent être envoyés à l'environnement.

Le processus de simulation est géré par messages, les processeurs échangent des messages pour avancer l'exécution d'un modèle. Chaque message contient des informations pour identifier l'émetteur et le récepteur. Deux catégories principales des messages existent, des messages de synchronisation et de contenu. Ces catégories sont composées de plusieurs types de messages.

L'outil fournit un langage de spécification qui permet de décrire le couplage entre les modèles, des valeurs initiales et des événements d'entrée externes. Les modèles atomiques sont écrits en C ++ et doivent être dérivé de la classe **Atomic** (la figure 19). Les méthodes qui déterminent le comportement d'un modèle atomique sont d'initialisation (exécutée lorsque la simulation démarre et initialise les variables du modèle), de la transition interne (exécutée quand une transition interne est prévue), de la transition externe (exécutée lorsqu'un événement externe arrive), et de sortie (génère la sortie du modèle et est exécutée avant la fonction de transition interne). CD++ fournit des fonctions qui peuvent être utilisés à partir des modèles atomiques (holdIn, passivate, sendOutput ...) permettent au modèle de définir l'état actuel et sa durée.

```
class Atomic : public Model  {
  public:
   virtual ~Atomic() {}  // Destructor

  private:
    State st;

  protected:
   enum State { active, passive } ;

   Atomic( const string &name = "Atomic" ) : Model( name ), st( passive ) {}
    // Constructor

   virtual Model &initFunction() = 0 ;
   virtual Model &externalFunction( const ExternalMessage & ) = 0 ;
   virtual Model &internalFunction( const InternalMessage & ) = 0 ;
   virtual Model &outputFunction( const InternalMessage & ) = 0 ;
   Model &holdIn( const State &, const Time & ) ;
   Model &passivate();
   Model &state( const State &s ) { st = s; return *this; }
   const State &state() const {return st;}
}
```

Figure 19: Exemple d'un modèle atomique dans CD++.

Les modèles couplés en CD++ sont définies en utilisant un langage de spécification définis pour cette raison. Le modèle couplé au niveau supérieur est toujours définie en utilisant la clause **[top]**, La figure 20 montre un exemple d'un modèle couplé qui contient deux modèles atomiques (transducer et generator) et un modèle couplé (Consumer). Le modèle a une sortie (out), et les clauses de liaison montrent comment les sorties sur un modèle sont connectées aux entrées d'un autre.

```
[top]
components : transducer@Transducer generator@Generator Consumer
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out

[Consumer]
components : queue@Queue processor@Processor
in : in
out : out
Link : in in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor out
```

Figure 20: Exemple d'un modèle couplé dans CD++.

CD++ inclut également un langage de spécification pour décrire les modèles Cell-DEVS. Ces définitions sont basées sur les spécifications formelles définies précédemment et peut être complétées en considérant quelques paramètres comme la taille, les influences, le voisinage, et les frontières, sont utilisées pour générer l'espace cellulaire complet. Le comportement de la fonction de traitement local est défini en utilisant un ensemble de règles. Dans CD++ les modèles Cell-DEVS sont un cas particulier de modèle couplé. Alors, lors de la définition d'un modèle cellulaire, tous les paramètres du modèle couplé sont disponibles, en plus, des paramètres supplémentaires sont nécessaires pour définir les dimensions de l'espace cellulaire, le temps de retard, les valeurs initiales par défaut, et les règles de transition locales.

La figure 21 montre un exemple du Cell-DEVS. Dans ce modèle, chaque cellule peut être occupé par une entité vivante (valeur = 1), ou il peut être vide (une cellule morte est vide, par exemple, la valeur = 0). Une cellule vivante reste vivante que si elle a trois ou quatre voisins vivant. Sinon, c'est morte, Une cellule devient vivante quand il y a exactement trois voisins vivants d'une cellule vide. Le modèle couplé Cell-DEVS dans la figure 21 est définie par sa taille (width = 20, height = 20), sa frontière (wrapped, ce qui signifie que les cellules dans une frontière communiquer ses résultats aux voisins de la frontière en face), la forme du voisinage (Voisinage de Moore), et le type de retard (transport). Les règles représentent le comportement de chaque cellule dans le modèle.

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1-1) life(1,0) life(1,1)
initialvalue : 0
initialvalue : 5    00000001110000000000
initialvalue : 7    00000100100100000000
initialvalue : 8    00000101110100000000
initialvalue : 9    00000100100100000000
initialvalue : 11   00000001110000000000
localtransition : life-rule

[life-rule]
rule : 1 100 { (0,0) = 1 and trueCount = 5 }
rule : 1 100 { (0,0) = 0 and trueCount = 3 }
rule : 0 100 { t }
```

Figure 21: Exemple d'un modèle Cell-DEVS.

CD++Modeler est une interface graphique fournie avec l'outil qui permet à l'utilisateur de définir des modèles DEVS. CD++Modeler fournit une interface graphique qui permet aux utilisateurs de construire des modèles atomiques et couplés graphiquement dans un mode glisser-déposer. Les modèles atomiques et couplés peuvent être générés graphiquement, Les ports d'entrée et de sortie peuvent être définis, et d'autres services sont inclus. La figure 22 montre une vue générale de l'interface CD++Modeler.

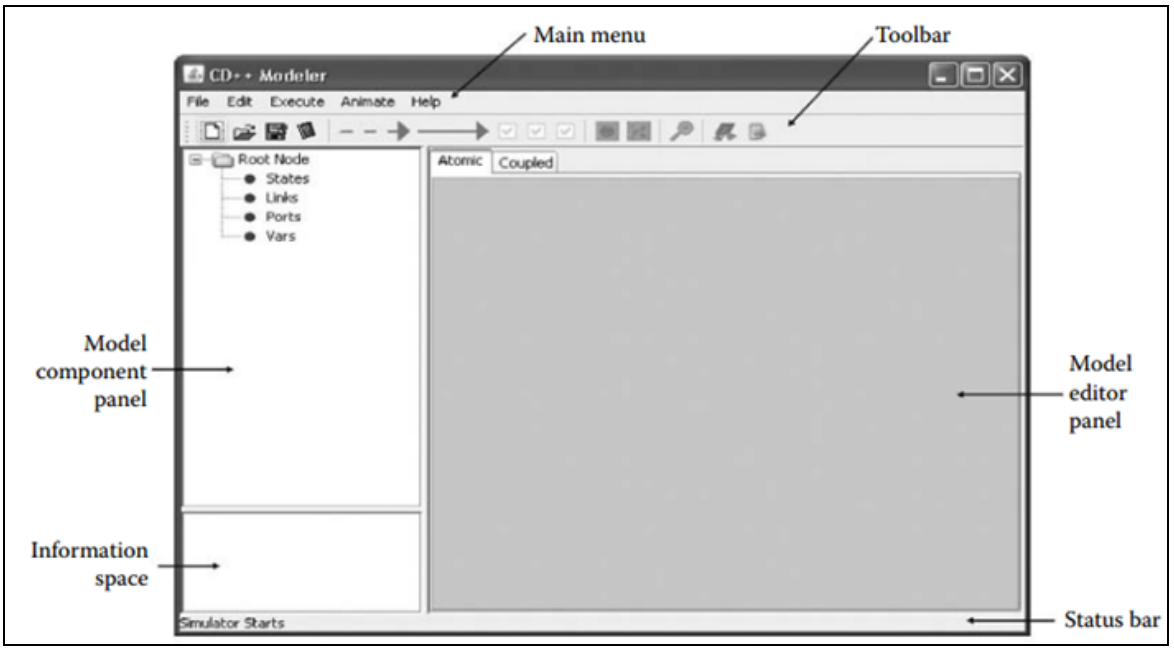

Figure 22: Exemple d'un modèle Cell-DEVS.

IV. PowerDEVS

PowerDEVS est un outil pour la modélisation et la simulation DEVS, orienté à la simulation de systèmes hybrides [29]. Développé par E. Kofman, Université de Rosario. PowerDEVS permet de définir des modèles atomiques en C++ qui peut être alors graphiquement couplées pour créer des systèmes plus complexes. Il donne la possibilité d'effectuer des simulations en temps réel, permettant la conception et l'implémentation automatique des contrôleurs numériques. Il peut être interconnecté avec Scilab [30].

Une autre caractéristique remarquable de PowerDEVS, est la possibilité d'exécuter des simulations sous un système d'exploitation en temps réel (RTAI) [31] synchronisant avec une horloge en temps réel, qui permet la conception et l'implémentation automatique de contrôleurs numériques synchrones et asynchrones. Combiné avec sa bibliothèque de simulation de systèmes continus. PowerDEVS est aussi un outil efficace pour la simulation en temps réel de systèmes physiques.

PowerDEVS est composé de différents programmes indépendants (l'éditeur de modèle, l'éditeur atomique, le préprocesseur, l'interface de simulation et un espace de travail correspondant à une instance Scilab). Toutes ces applications ont été programmées en C++

avec les bibliothèques graphiques QT, Sauf l'éditeur de modèle qui a été programmé en Visual Basic. L'éditeur de modèle est le programme principal de PowerDEVS, il fournit l'interface graphique et le lien avec le reste des applications. En plus de la construction et la gestion des modèles et des bibliothèques, Il permet de lancer la simulation (en invoquant le préprocesseur). La fenêtre principale de l'éditeur de modèle (la figure 23) permet à l'utilisateur de créer et d'ouvrir les modèles et les bibliothèques. Il y a aussi quelques fonctionnalités avancées qui peuvent être gérés à partir de la fenêtre principale comme modifier les bibliothèques actives, et la configuration des barres d'outils et le menu pour invoquer les nouvelles applications externes.

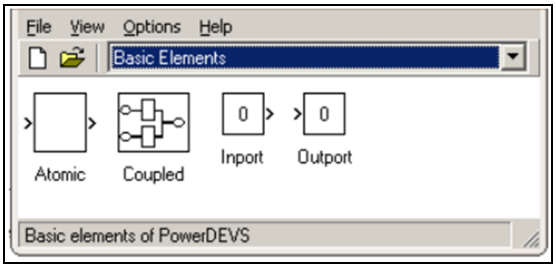

Figure 23: La fenêtre principale de l'éditeur de modèle (Model Editor).

La figure 24 montre une fenêtre de modèle avec un modèle composé de cinq sous modèles. La fenêtre de modèle fournir toutes les éditions typique graphique, alors les blocs peuvent être copiés, redimensionné, etc.

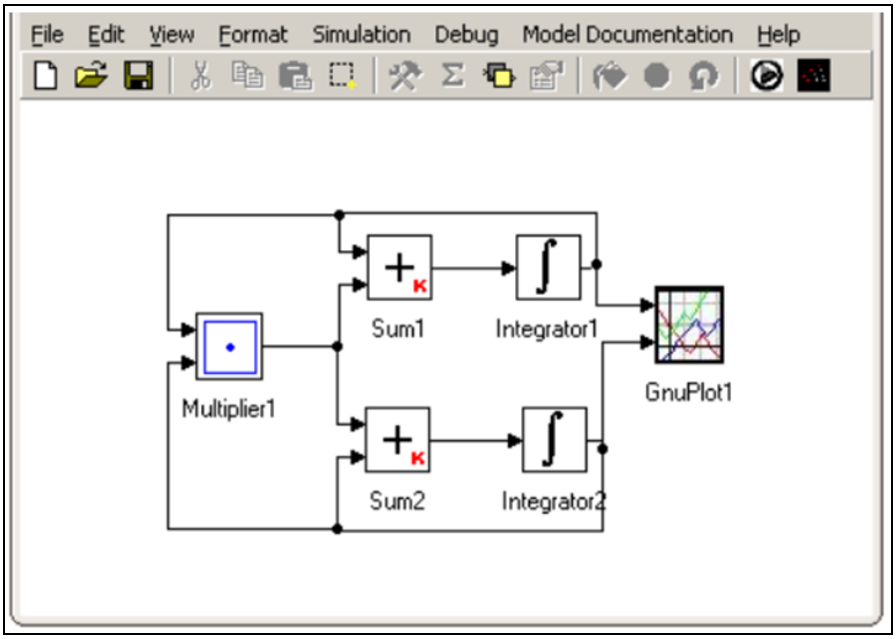

Figure 24: La fenêtre de modèle (Model Window).

La fenêtre d'édition de bloc (la figure 25) permet de configurer l'aspect graphique du bloc, De choisir les paramètres de bloc (dans le cas des modèles atomiques) et pour sélectionner le fichier qui contient le code associé avec les définitions du modèle DEVS.

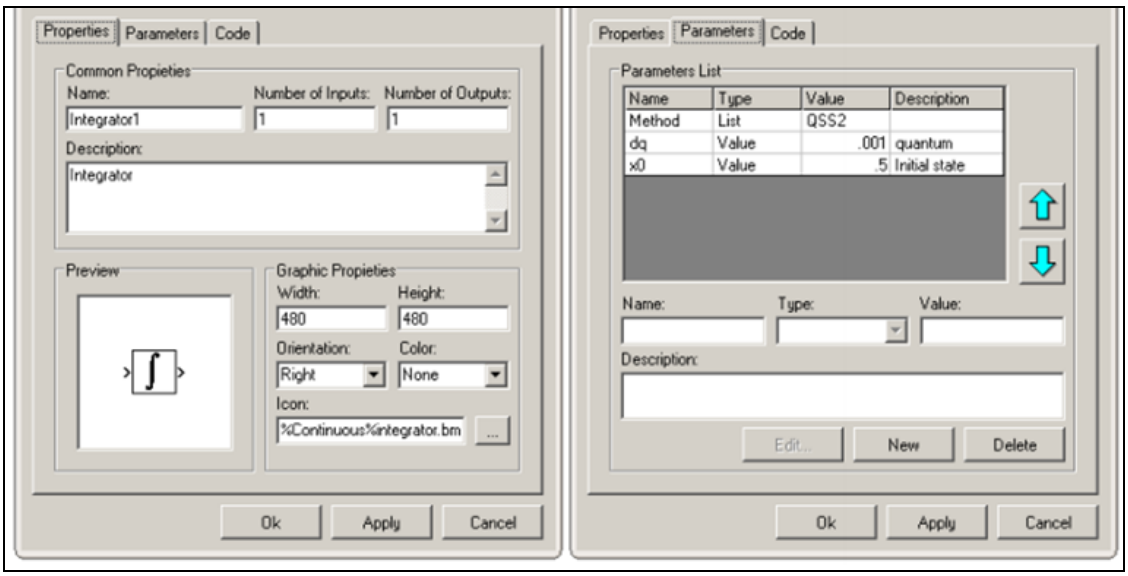

Figure 25 : La fenêtre d'édition de bloc (Block Edition Window).

Les paramètres de blocs sont définis et sélectionnés dans la fenêtre d'édition du bloc. Après avoir défini, leurs valeurs peuvent être modifiées par un double clic sur le bloc (la figure 26).

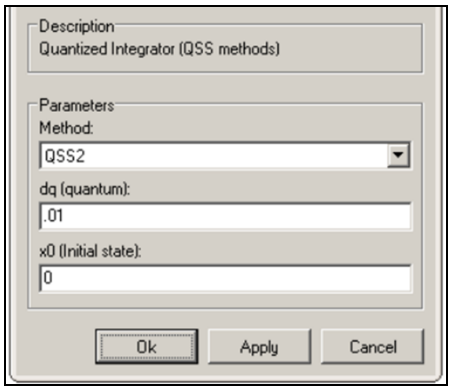

Figure 26: Fenêtre de modification des valeurs de paramètres.

Les modèles couplés n'ont pas de code associé, mais ils ont quelques fonctionnalités supplémentaires qui peuvent être modifiés à partir de la fenêtre d'édition de bloc et le menu d'édition (par exemple les priorités internes et l'ordre des ports d'entrée et de sortie).

L'éditeur atomique facilite l'édition du code C++ correspondant à chaque modèle atomique. Il peut être invoqué de la fenêtre d'édition de bloc pour modifier un code existant ou créer un nouveau code. La fenêtre principale de l'éditeur atomique est représentée dans la figure 27.

En utilisant le l'éditeur atomique, l'utilisateur doit définir les variables qui forment l'état et la sortie du modèle atomique et les variables qui représentent les paramètres du système. Après, le code C++ de l'avancement de temps (Time advance) et les fonctions de la transition interne (Internal transition) et de sortie(Output) doivent être placées dans les fenêtres correspondantes. Il y a deux fenêtres supplémentaires (init et exit) où l'utilisateur peut aussi ajouter un code qui est exécuté avant les débuts de simulation (pour mettre des états initiaux et des paramètres) et un code qui est exécuté à la fin de la simulation (pour fermer quelques dossiers ouverts, par exemple). Quand le modèle est sauvé, le code est automatiquement accompli et stocké dans les fichiers **.cpp** et **.h**.

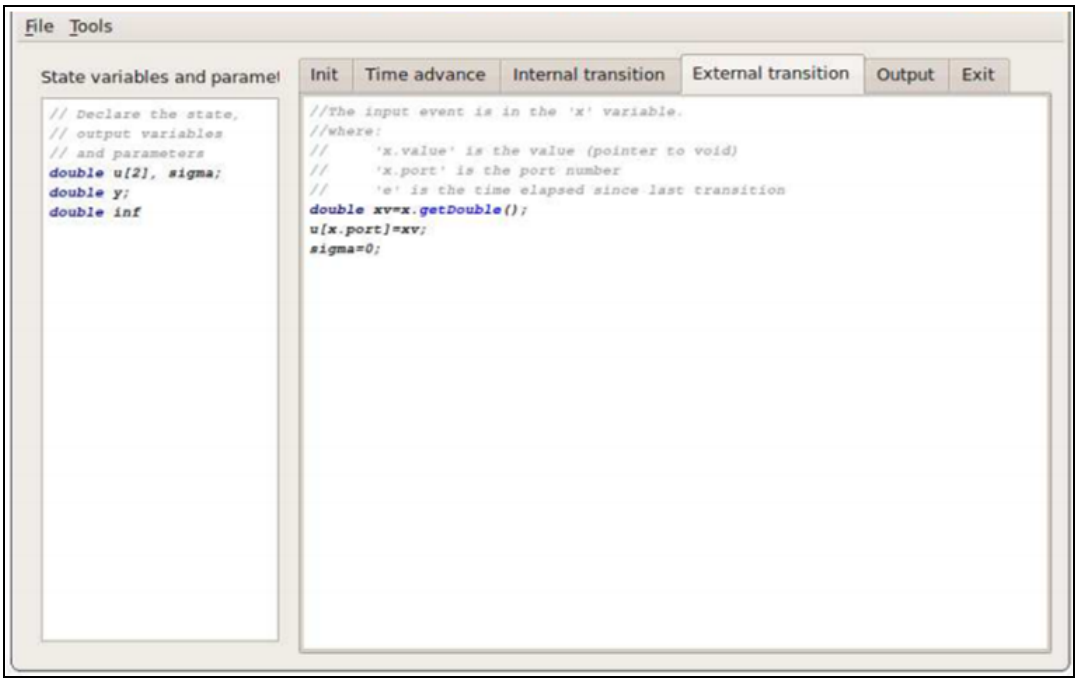

Figure 27: Fenêtre principale de l'édition de modèles atomique.

Le préprocesseur prend un fichier **.pdm** ou **.pds** produit par l'éditeur de modèle et produit le programme de simulation. Il traduit le fichier **.pdm** vers un fichier **.h** qui lier les simulateurs et les coordinateurs selon la structure de couplage et les paramètres de bloc. Le préprocesseur peut être invoqué de façon transparente en utilisant la commande de simulation rapide.

PowerDEVS fournit une interface graphique pour l'exécution de la simulation (la figure 28). L'interface permet aussi de changer quelques paramètres pour configurer l'expérimentation :

- **Final Time** : dit combien de temps pour simuler le modèle.
- **Simulations to run** : plusieurs simulations peuvent être exécutées à la fois. Cela peut être utile lorsque les statistiques à partir des résultats de simulation doivent être calculées.
- **Illegitimate check break** : PowerDEVS arrête la simulation si le temps n'avance pas après un nombre choisi de pas.
- **Simulate step-by-step** : la simulation peut être avancé effectuant une étape (ou plusieurs) à l'unité de temps, et les résultats peuvent être analysés entre les étapes.
- **Synchronize time** : Synchroniser le temps: La simulation peut être exécutée synchronisé avec le temps réel.

Figure 28: Fenêtre de simulation de CD++.

V. DEVSimPy

DEVSimPy [32] est un projet open source lancé par l'équipe de modélisation et de simulation du laboratoire SPE (Sciences pour l'Environnement, Université de Corse). Le but de cet outil est de fournir aux développeurs un cadre de modélisation et de simulation en langage Python. Cet outil utilise la bibliothèque **wxPython** qui est une combinaison de la bibliothèque de classes **wxWidgets** (C++) et de langage Python. Le noyau de modélisation et de simulation dans DEVSimpy est basée sur l'API PythonDEVS[33] qui offre un ensemble cohérent de classes pour construire un système modulaire et de réaliser sa simulation hiérarchique. L'idée du projet DEVSimPy était de fournir une interface graphique avec le noyau Python DEVS pour la manipulation des modèles d'une manière visuellement dynamique. Tous les modèles DEVS implémentés avec l'API PythonDEVS sont valides dans DEVSimPy. Nombreux aspects ont été ajoutés à l'interface, en particulier de simplifier le couplage entre les modèles, pour les enregistrer dans un format spécifique ou pour modifier le code correspondant. Si le fichier PythonDEVS peut être considéré comme le modèle par défaut géré par DEVSimPy, **.amd** et **.cmd** ont été introduits pour représenter respectivement les modèles atomiques et les modèles couplés, Ce sont des fichiers compressés contenant séparément un fichier comportemental et de graphique. Le fichier de comportement implémente la spécification DEVS avec les règles de PythonDEVS et le fichier graphique permet l'affichage du modèle dans l'interface DEVSimPy, l'approche consistant à séparer le comportement et le composant graphique du modèle fournit un moyen de gérer ces deux aspects d'une manière efficace. Le modèle couplé de haut niveau (appelé diagramme) qui est composé des modèles (.amd, .cmd, .py) a un format spécifique nommé **.dsp**. Les développeurs ont la possibilité de partager le fichier **.dsp** associé à des bibliothèques afin de simuler des diagrammes qui ont déjà été construits.

La philosophie de DEVSimPy est d'être un environnement ouvert, extensible et participative pour les développeurs. Un gestionnaire de plug-in est proposé afin d'étendre les fonctionnalités de DEVSimPy permettant leur activation et désactivation à travers une fenêtre de dialogue.

DEVSimPy est implémenté de façon orientée objet avec l'utilisation de modèles de conception pour assurer une évolution du logiciel parfait. Le MVC (Le contrôleur de vue du modèle) a été utilisé pour organiser l'interaction avec les utilisateurs et de diviser l'architecture logicielle en deux entités(le noyau PythonDEVS et le moteur DEVSimPy), cette séparation

aide à faciliter le développement, les tests et la maintenance de DEVSimPy. Les utilisateurs peuvent choisir au moment de l'exécution entre le simulateur DEVS classique ou d'autres simulateurs proposés par les développeurs. DEVSimPy offre la possibilité de modifier le comportement de modèles au cours du processus de simulation et fournit une interface de visualisation des résultats.

La figure 29 montre l'interface générale de DEVSimPy qui contient quatre régions principales, Canvas (c'est à l'intérieur que l'on construit les diagrammes de modèles à simuler), Contrôle (sous forme d'onglets détachables dédiés à la gestion des librairies, la gestion des propriétés des modèles sélectionnés et de la gestion de la simulation du diagramme courant), Barre d'outils (rassemble les raccourcis des actions comme l'ajout de diagrammes, l'ouverture de diagrammes existant, la sauvegarde et l'impression du diagramme courant, la simulation, etc..), Barre de status (elle permet d'informer l'utilisateur sur les actions en cours comme le processus de simulation, copier/coller, la modification de diagramme, etc..).

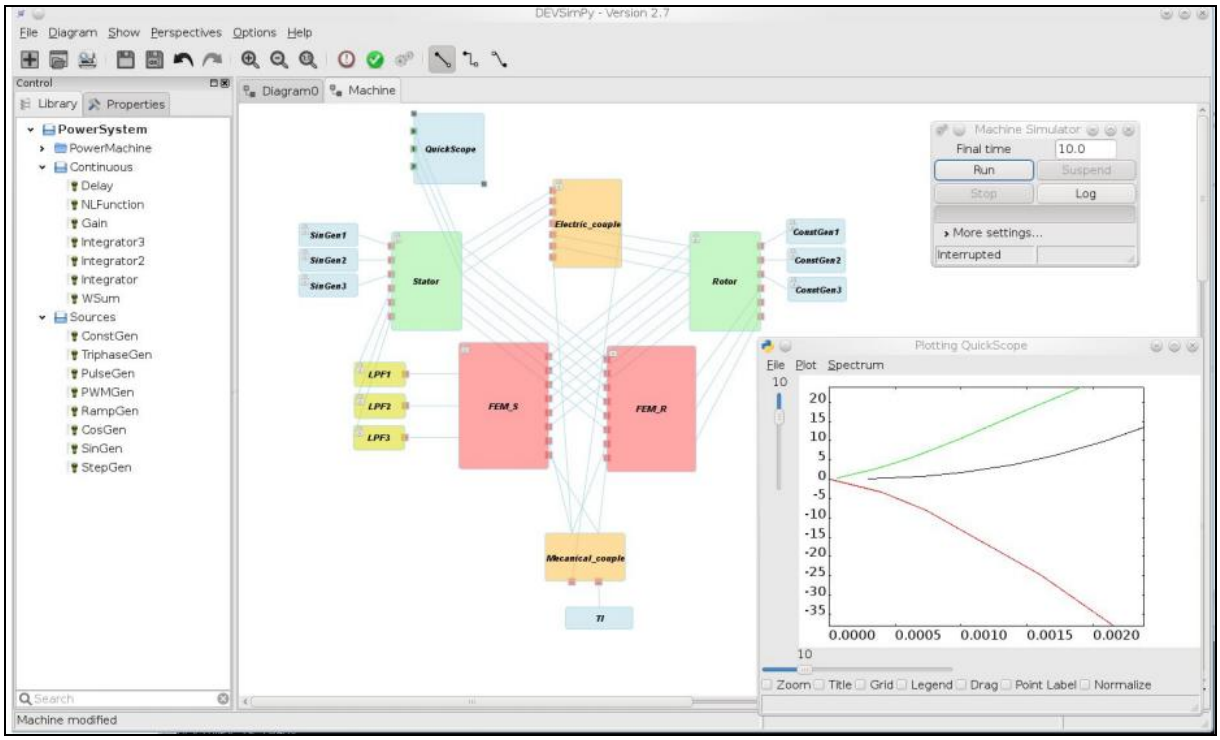

Figure 29: L'interface générale de DEVSimPy.

DEVSimPy constitue une couche graphique amélioré du logiciel PyDEVS, l'utilisateur peut créer des modèles de deux façons différentes, par l'importation de fichier PyDEVS ou

par la création de modèles propres à DEVSimPy. Cette distinction repose sur un concept central dans DEVSimPy, c'est la séparation entre le comportement et la représentation graphique des modèles. Lorsque l'utilisateur choisit d'implémenter le comportement des modèles PyDEVS dans son logiciel de programmation il a la possibilité ensuite d'importer ceux-ci dans DEVSimPy pour mieux les manipuler, et lorsque il choisit de construire ses modèles en utilisant DEVSimPy il lui suffit d'invoquer le gestion de création des modèles par un clic droit sur le canvas qui va accepter le modèle. L'utilisateur se laisse guider par le gestionnaire en renseignant les champs indispensables à la création des modèles.

Lorsque les modèles sont créés ils apparaissent directement dans Canvas sur lequel a eu lieu le clic-droit qui a fait apparaître le gestionnaire de création. L'utilisateur doit maintenant décrire le comportement DEVS dans un fichier python qui est invoqué en cliquant sur le menu contextuel Edit. Si les modèles sont modifiés par la suite et que l'utilisateur veut sauvegarder les modifications il devra alors exporter les modèles par un clic-droit sur ceux-ci. Les modèles couplés peuvent être exportés en CMD, en XML ou bien en JS. Par contre les modèles atomiques ne sont exportables qu'en AMD. Si les modèles sont déjà stockés dans une librairie visible dans le panel «Library» et qu'ils sont sauvegardés après modification, la mise à jour de la librairie est automatique.

Avec DEVSimPy la simulation des diagrammes est accessible par une pression sur le bouton "Simulation" dans la barre d'outils. La fenêtre ci-dessous (Figure 30) apparaîtra pour laisser le soin à l'utilisateur de lancer la simulation.

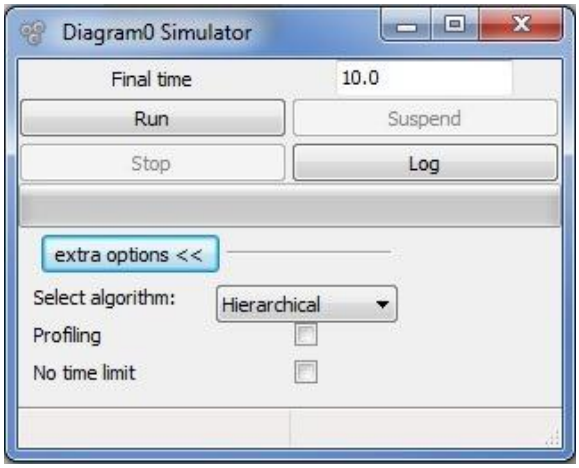

Figure 30 : Fenêtre de simulation dans DEVSimPy.

En plus, DEVSimPy a été construit avec un ensemble de fonctionnalités supplémentaires pour modéliser et simuler les modèles DEVS:

- Avec DEVSimPy, le modélisateur peut utiliser des plugins pour contrôler et étendre le comportement d'un ensemble de modèles (appelé plug-in général) ou d'un modèle simple (appelé plug-in spécifique). L'utilisateur contrôle tous ces plugins en utilisant un gestionnaire de plug-in pour les activer ou désactiver.

- Une autre fonctionnalité qui rend DEVSimPy unique est la possibilité de modifier le code du modèle lors de la simulation. Le modélisateur peut décider de suspendre la simulation pour changer le code du modèle et de redémarrer la simulation pour inclure le nouveau comportement du modèle. Cette fonctionnalité est souvent utilisée pour visualiser les résultats de la simulation et lorsque le modélisateur décide d'interagir avec le modèle et veut modifier le comportement d'un ou plusieurs modèles pour contrôler les résultats de simulation.

- Avec DEVSimPy, le modélisateur peut exporter ou importer des modèles à partir de bibliothèques dynamiques les bibliothèques composées par des modèles locaux ou sur le serveur Web. Lorsque les modèles sont stockés dans un répertoire local, l'importation est faite par le chemin de fichiers .amd, .cmd ou .py. Lorsque les modèles sont stockés sur le serveur Web (http (s) ou ftp), l'importation est faite à partir de l'URL du fichier de modèle, cette fonctionnalité est utile lorsque les développeurs veulent gérer les fichiers DEVSimPy de manière collaborative.

- Une fonctionnalité intéressante dans DEVSimPy est l'option de profilage une fois que la simulation est terminée. Les développeurs peuvent analyser le temps d'exécution, le nombre d'exécution de la fonction de transition pour tous les modèles atomiques, cette fonctionnalité est souvent utilisé pour mesurer une certaine notion d'activités du modèle et peut également être utilisé au cours de la simulation.

VI. JDEVS

JDEVS [34] est un outil pour la modélisation et la simulation DEVS. Développé (par J. B. Filippi, l'université de Corse, France) en JAVA et utilisé pour la modélisation des systèmes naturels. JDEVS est composé de cinq modules indépendants (un moteur de simulation, une interface graphique de modélisation, un module de stockage, un module de connexions au SIG et les cadres expérimentaux de visualisation et de simulation) présentés en figure 31. JDEVS prend pour base la technique de modélisation DEVS.

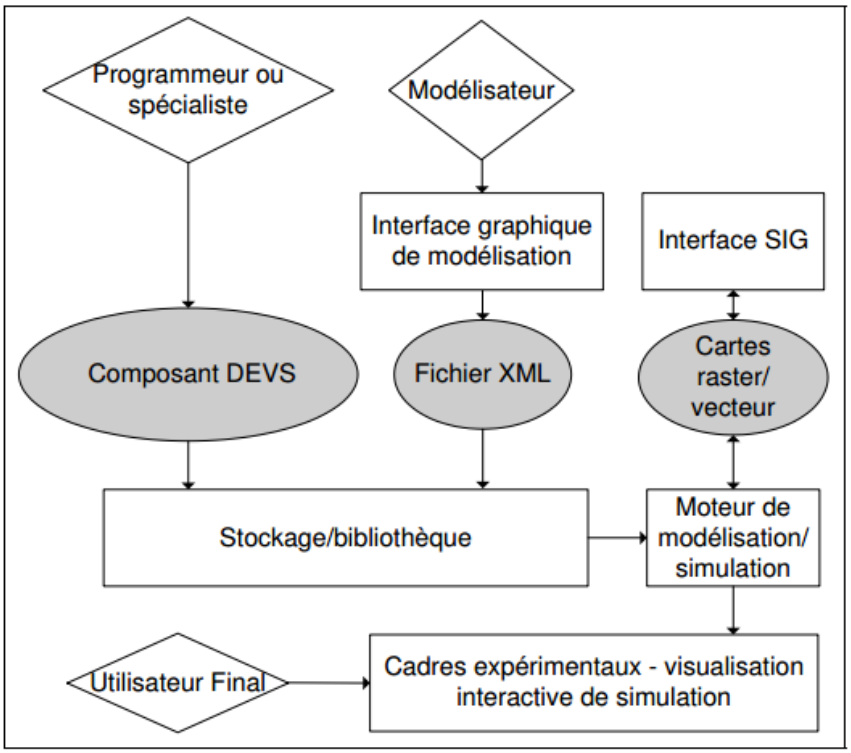

Figure 31 : Vue des modules du logiciel JDEVS.

Le moteur de modélisation et de simulation est une implémentation de la méthodologie DEVS avec ports. Les modèles DEVS atomiques doivent pour l'instant être transformé en instructions JAVA.

La modélisation de modèles atomiques peut toutefois se faire directement dans l'interface et en utilisant Java. Aussi, il est possible d'ajouter un composant atomique vide, de définir ses interfaces d'entrée et de sortie et de générer le squelette du code du modèle. Ce modèle vide est ensuite enregistré dans la bibliothèque et compilé. La figure 32 présente le code généré automatiquement pour un exemple d'un modèle atomique (DevsAtom).

```
public class DevsAtom extends BasicModel {
      Port i1 = new Port(this,"i1","IN");
      Port o1 = new Port(this,"o1","OUT");

public DevsAtom () {
      super("DevsAtom");
      states.setProperty("A","");  }
    EventVector outFunction(Message m) {
        return new EventVector();}
    void intTransition() {}
    EventVector extTransition(Message m) {
        return new EventVector();}
    int advanceTime(){return A;}
}
```

Figure 32: Code source JAVA pour un composant atomique.

Les fonctions de sortie et de transition externe renvoient des vecteurs d'événements qui sont ajoutés à la liste générale des événements. La seule tâche de programmation qui incombe au spécialiste du domaine, est de spécifier la dynamique des modèles atomiques à travers ses quatre méthodes. Une fois le modèle atomique créé, il est stocké dans la bibliothèque pour être utilisé plus tard dans une composition de modèle à l'intérieur d'un modèle couplé.

L'interface de modélisation permet aux utilisateurs de construire graphiquement leurs modèles. La figure 33 représente une vue de l'outil constitué d'un modèle couplé avec deux ports d'entrée et un port de sortie. De plus, l'interface dispose d'une barre de menu simple permettant d'ajouter des modèles vides (atomique, couplés ou d'autres techniques de modélisation), d'ouvrir une bibliothèque et de supprimer des modèles. Modèles atomiques et couplés ont des panneaux de propriétés différents, la figure 33 présente celui d'un modèle couplé permettant d'ajouter et de supprimer des ports et de changer son nom.

Le panneau de modèles atomiques figure 34 permet d'inspecter et de modifier la valeur des paramètres du modèle, de lancer l'éditeur correspondant et de compiler le modèle afin de le recharger dans l'interface graphique.

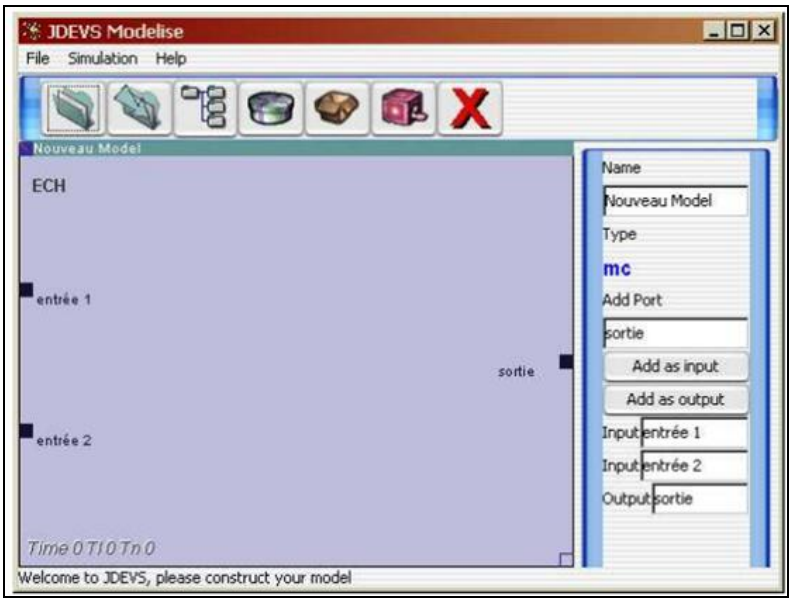

Figure 33: Interface de modélisation contenant un modèle couplé et son panneau de propriétés.

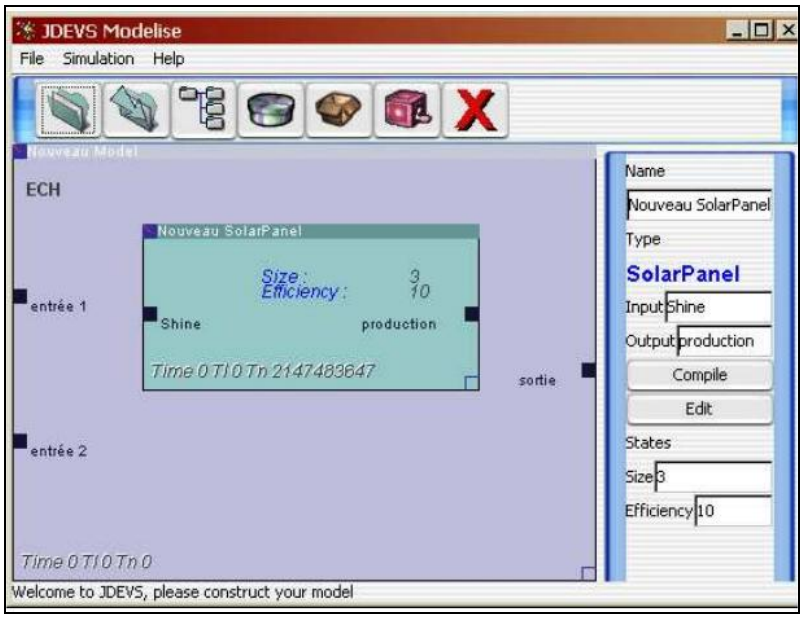

Figure 34: Interface de modélisation contenant un modèle atomique et son panneau de propriétés.

Pour ajouter des modèles existant il suffit de les glisser depuis le composant bibliothèque (figure 35) vers l'espace de travail. La partie visible du module de stockage de JDEVS est un composant graphique présenté en figure 35, ce composant présente les modèles par domaine et sous-domaines sous la forme d'un arbre.

Le module de stockage permet de formater la structure des modèles pour pouvoir l'enregistrer au format XML et la stocker dans une bibliothèque du type HmLib [35]. Le but d'une telle bibliothèque est d'offrir un moyen facile et performant de stocker et de récupérer les modèles. La bibliothèque HmLib peut aussi être considérée comme une base de données orientée objet, ce qui facilite aussi le stockage des modèles en XML. En plus de la structure, il est aussi possible de stocker les liens d'héritage et d'abstraction entre modèles. Les modèles stockés dans la bibliothèque (habituellement sous la forme de code source) sont appelés "hors-contexte". Pour récupérer un modèle, il faut l'instancier puis le mettre "en-contexte" en le replaçant dans l'état où avait été stocké l'objet. Plusieurs scénarios de simulation peuvent ainsi créer plusieurs modèles "en-contexte" à partir d'un modèle "hors-contexte".

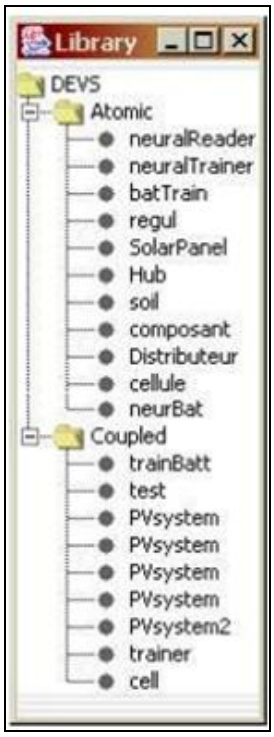

Figure 35: Composant de l'interface graphique présentant les modèles dans la bibliothèque

Les cadres expérimentaux sont les interfaces utilisateur de simulation des modèles stockés dans la bibliothèque. Les expérimentations sur des modèles en diagramme peuvent être effectuées directement depuis l'interface graphique de modélisation. Il est ainsi possible de vérifier le comportement d'un modèle en cours de création et donc d'ajuster ses paramètres ou corriger les erreurs de conception sans explicitement utiliser le module cadre expérimental. L'application permettant d'exécuter les expérimentations se présente avec la même interface

que l'interface de modélisation, chaque modèle possède un panneau de propriétés. Le panneau de propriété affiché est le panneau du modèle actif (i.e. le composant sélectionné).

La figure 36 présente un modèle en cours de simulation dans l'interface. Le lancement de la simulation depuis l'interface graphique permet de charger une liste d'événements d'entrée et de spécifier les paramètres de l'expérimentation, ces actions sont effectuées depuis le panneau de simulation (la figure 36, en haut à gauche). L'option "track simulation" permet de réaliser la simulation en insérant un temps d'attente entre l'envoi de deux événements d'entrée pour vérifier le comportement du modèle. La liste des événements à traiter est affichée à l'intérieur de chaque modèle couplé et évolue au cours de la simulation. Il est possible de changer les paramètres de chaque modèle atomique durant la simulation en l'activant puis en modifiant les valeurs à l'intérieur de son panneau de propriétés (la figure 36, à droite). Le résultat de simulation est une liste d'événements de sortie qui peut être soit affichée (la figure 36, en bas à gauche), soit sauvegardé dans un fichier.

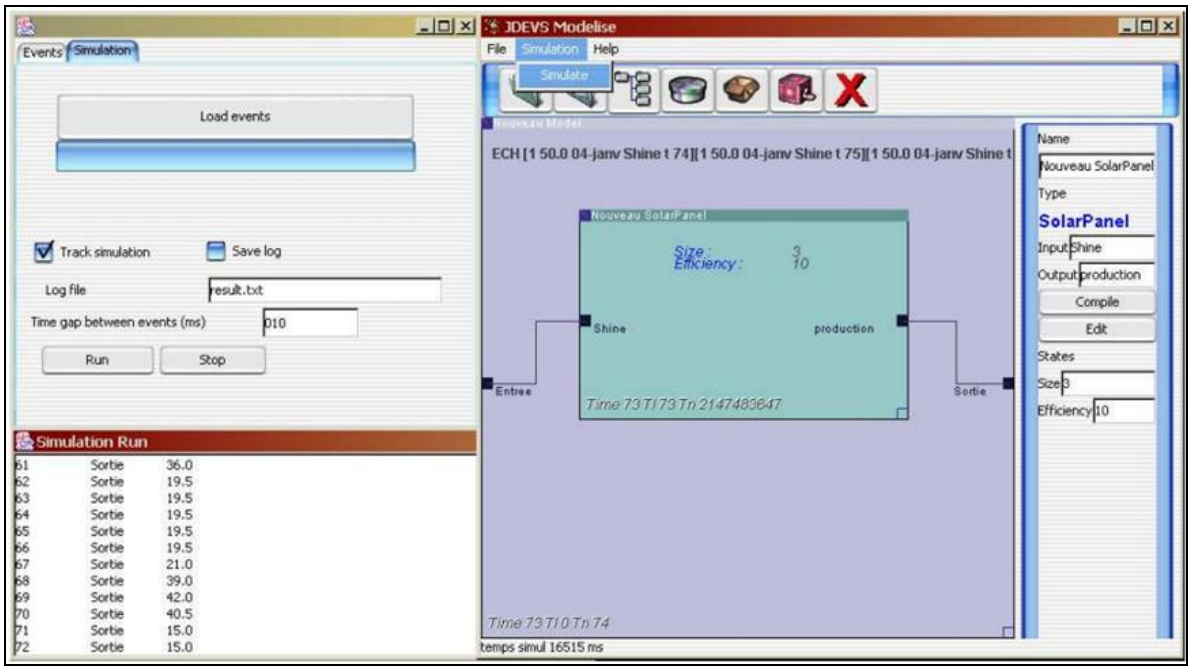

Figure 36: Interface de modélisation de l'environnement JDEVS

Conclusion

Comme nous l'avons constaté, les logiciels proposent notamment l'usage d'un formalisme unificateur, DEVS. 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.

Dans le chapitre suivant, nous allons définir et modéliser l'architecture de l'outil de modélisation et de simulation que nous allons le développer dans ce travail.

Chapitre III

Modélisation et l’architecture de système

Introduction

Pour mener à bien le projet, nous devons tout naturellement avoir recours à un formalisme de conception à savoir UML (Unified Modeling Language)[36], qui est le langage de modélisation graphique qui va nous permettre de comprendre et de décrire les besoins, de spécifier et documenter les systèmes ainsi que d'esquisser les architectures logicielles.

Dans ce chapitre, nous allons modéliser notre outil en utilisant le langage de modélisation objet UML. La conception de notre application se base sur le diagramme des cas d'utilisation, les diagrammes de séquences et les diagrammes d'activité, et la représentation de l'architecture par le diagramme de package et les diagrammes des classes.

I. Le Diagramme des cas d'utilisation

Les cas d'utilisation permettent de structurer les besoins des utilisateurs et les objectifs correspondants d'un système. Ils centrent l'expression des exigences du système sur ses utilisateurs en clarifiant et en organisant leurs besoins (les modéliser). Pour cela, les cas d'utilisation identifient les utilisateurs du système (acteurs) et leurs interactions avec le système. Ils permettent de classer les acteurs et structurer les objectifs du système. Un acteur représente un rôle joué par une personne qui interagit avec le système. Par définition, les acteurs sont à l'extérieur du système.

1. Le diagramme de cas d'utilisation du système

La figure37 présente le diagramme final des cas d'utilisation. Nous avons un seul acteur, qui est le modélisateur (l'utilisateur), il est chargé tout d'abord de créer un nouveau projet ou de charger un projet déjà créé et sauvegardé. S'il a choisi de créer un nouveau projet il doit construire les modèles (couplés et atomique) de son projet, et il peut également sauvegarder ce projet et le charger plus tard. Si le modèle créé est atomique, le modélisateur doit générer son code source et ensuite le compiler. Enfin, si le projet est implémenté correctement, le modélisateur peut lancer sa simulation, et peut également l'arrêter et enregistrer le résultat final.

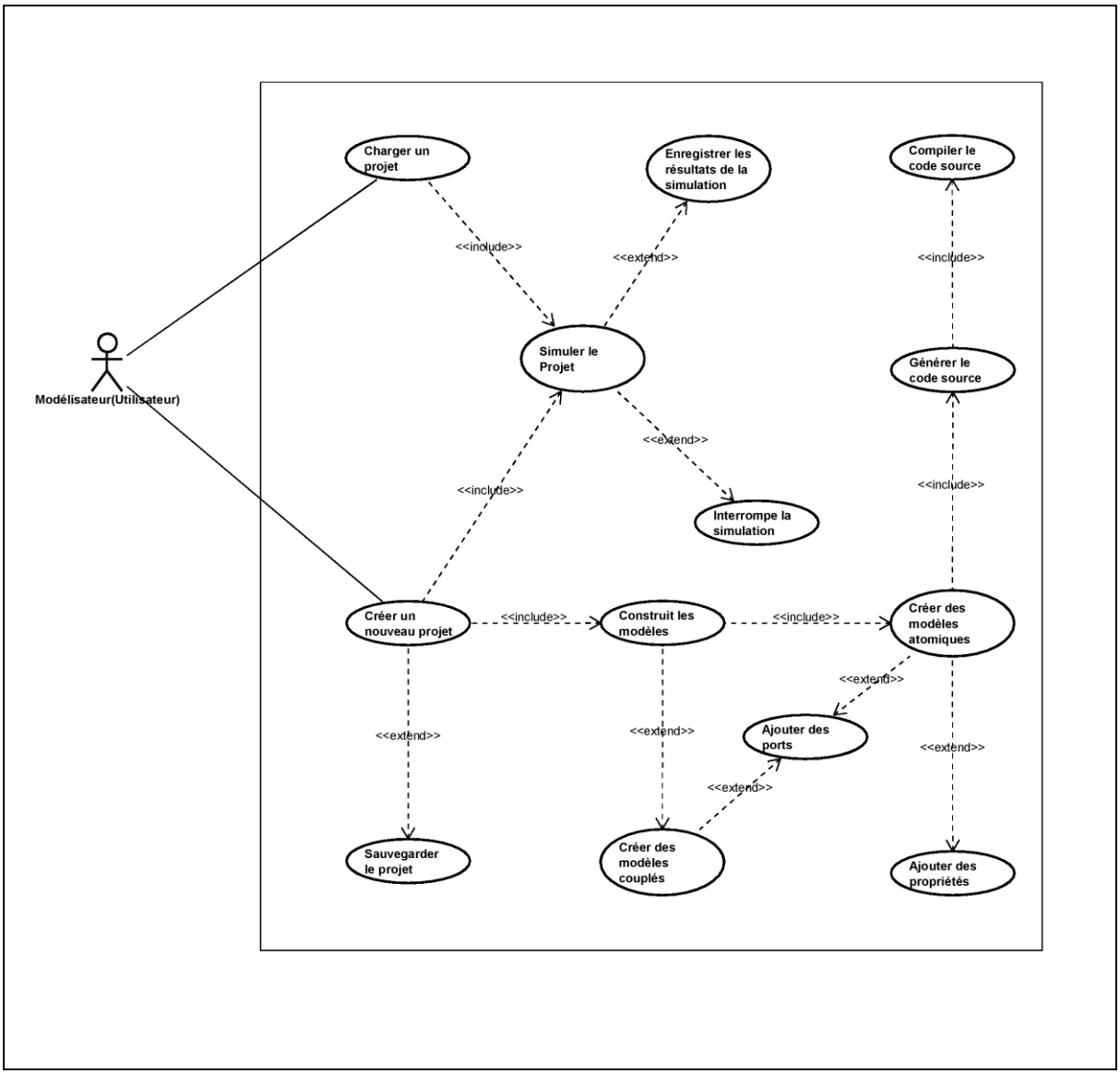

Figure 37 : Le diagramme de cas d'utilisation du système.

II. Les Diagrammes des séquences et d'activité

1. Le diagramme de séquence

Le diagramme de séquence est une variante du diagramme de collaboration mais, il possède intrinsèquement une dimension temporelle et ne représente pas explicitement les liens entre les objets privilégiant ainsi la représentation temporelle à la représentation spatiale. Il est plus apte à modéliser les aspects dynamiques du système. Nous présenterons les diagrammes des séquences du cas « Création d'un modèle », « Simulation », « Sauvegarde d'un projet» et « Chargement d'un projet ».

2. Le diagramme d'activité

Le diagramme d'activité est attaché à une catégorie de classes et décrit le déroulement des activités de cette catégorie. Le déroulement s'appelle "flot de contrôle". Il indique la part prise par chaque objet dans l'exécution d'un travail. Il sera enrichi par les conditions de séquence. Nous présenterons les diagrammes des activités du cas « Création d'un modèle », « La simulation », « Sauvegarde d'un projet» et « Chargement d'un projet ».

3. Diagramme de séquences et d'activité «Création d'un modèle»

Lorsque le modélisateur veut créer un nouveau modèle, il doit invoquer la fenêtre de gestionnaire de création des modelés, le système donc affichera cette fenêtre. Lorsqu'il accède à la fenêtre de la gestionnaire de création des modèles, il doit tout d'abord sélectionner le type de modèle, ensuite il doit saisir les informations relatives au modèle. Après que le modélisateur à confirmer la création de modèle, le système affichera le modèle créé dans l'espace de travail.

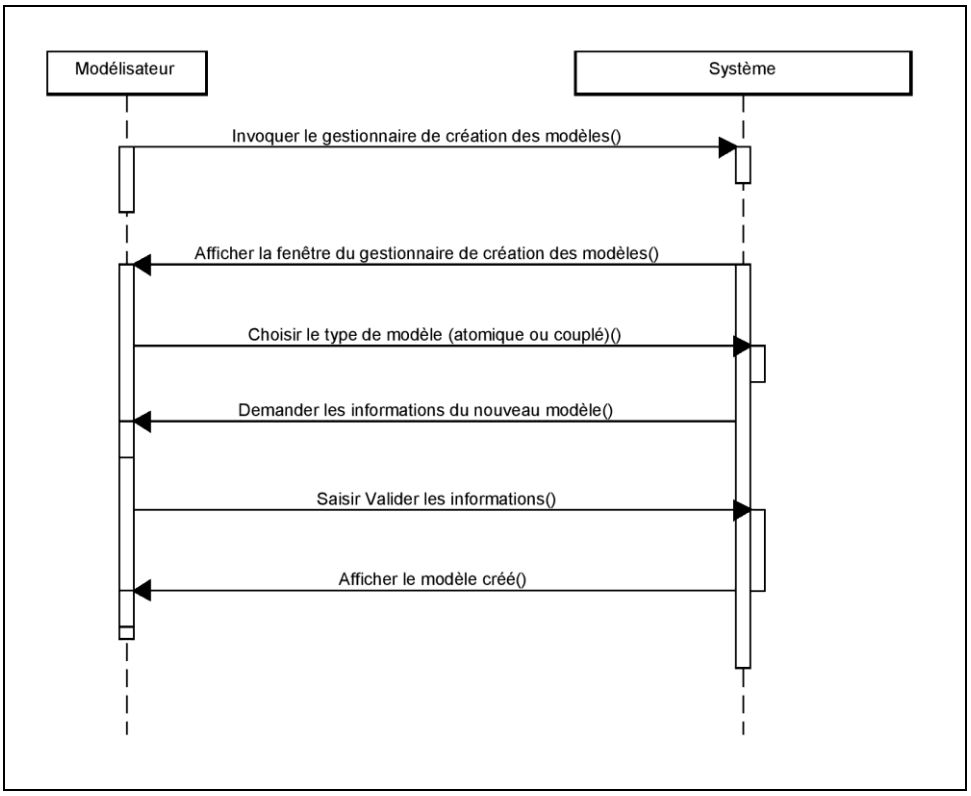

Figure 38 : Diagramme de séquences «Création d'un modèle».

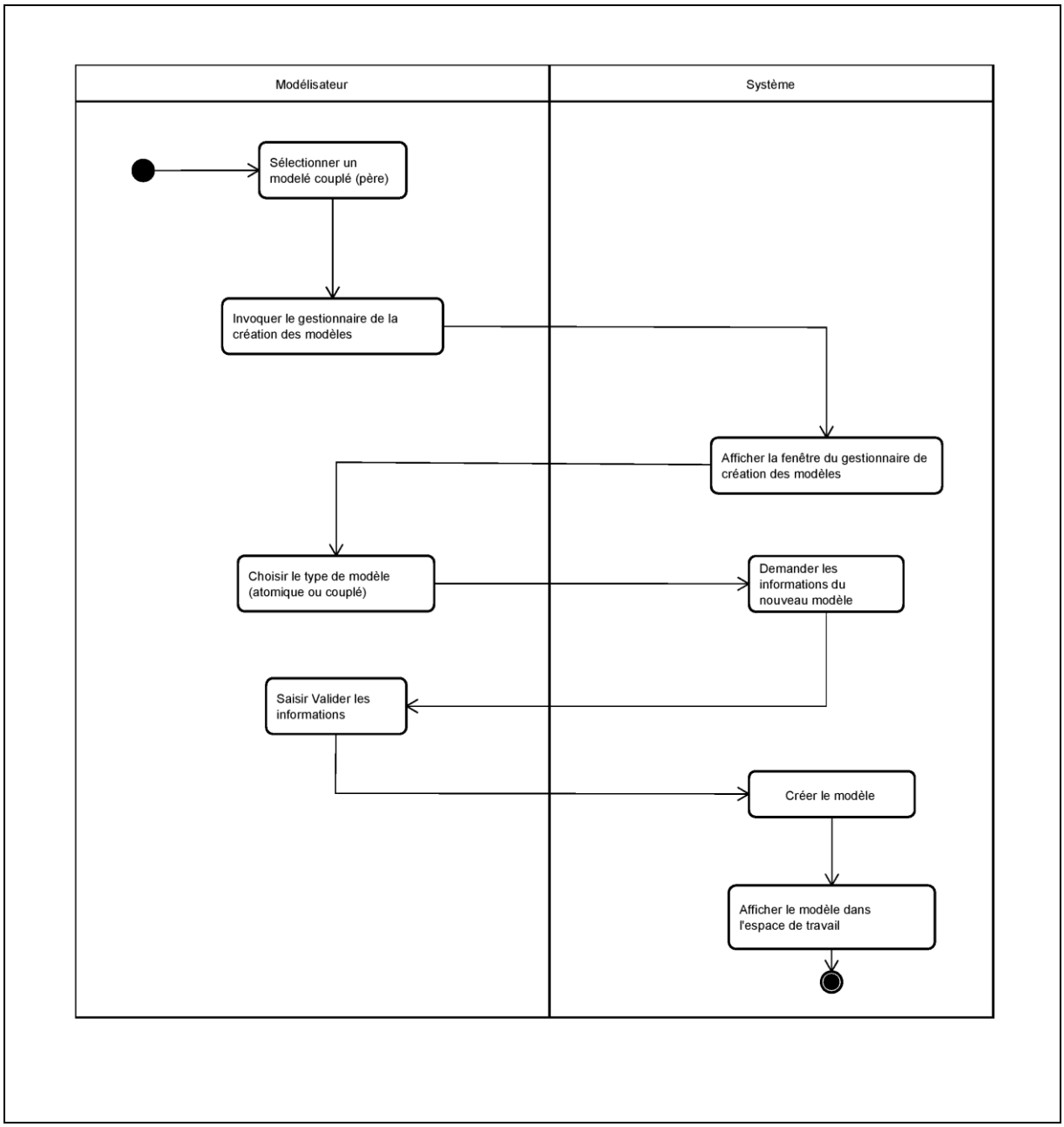

Figure 39 : Diagramme d'activité «Création d'un modèle».

4. Diagramme de séquences et d'activité «La simulation»

Lorsque le modélisateur veut simuler son projet, il doit invoquer la fenêtre de gestionnaire de la simulation, le système donc affichera cette fenêtre. Lorsque il accède à la fenêtre du gestionnaire de la simulation, il doit tout d'abord charger la liste d'événements d'entrée et de spécifier les paramètres de l'expérimentation, ensuite le système lui répond par un message informant le modélisateur que les événements ont été chargés. Après, que le modélisateur a

spécifié les paramètres de l'expérimentation, il peut lancer la simulation. Enfin, le système affichera un message informant l'utilisateur que la simulation a été terminée.

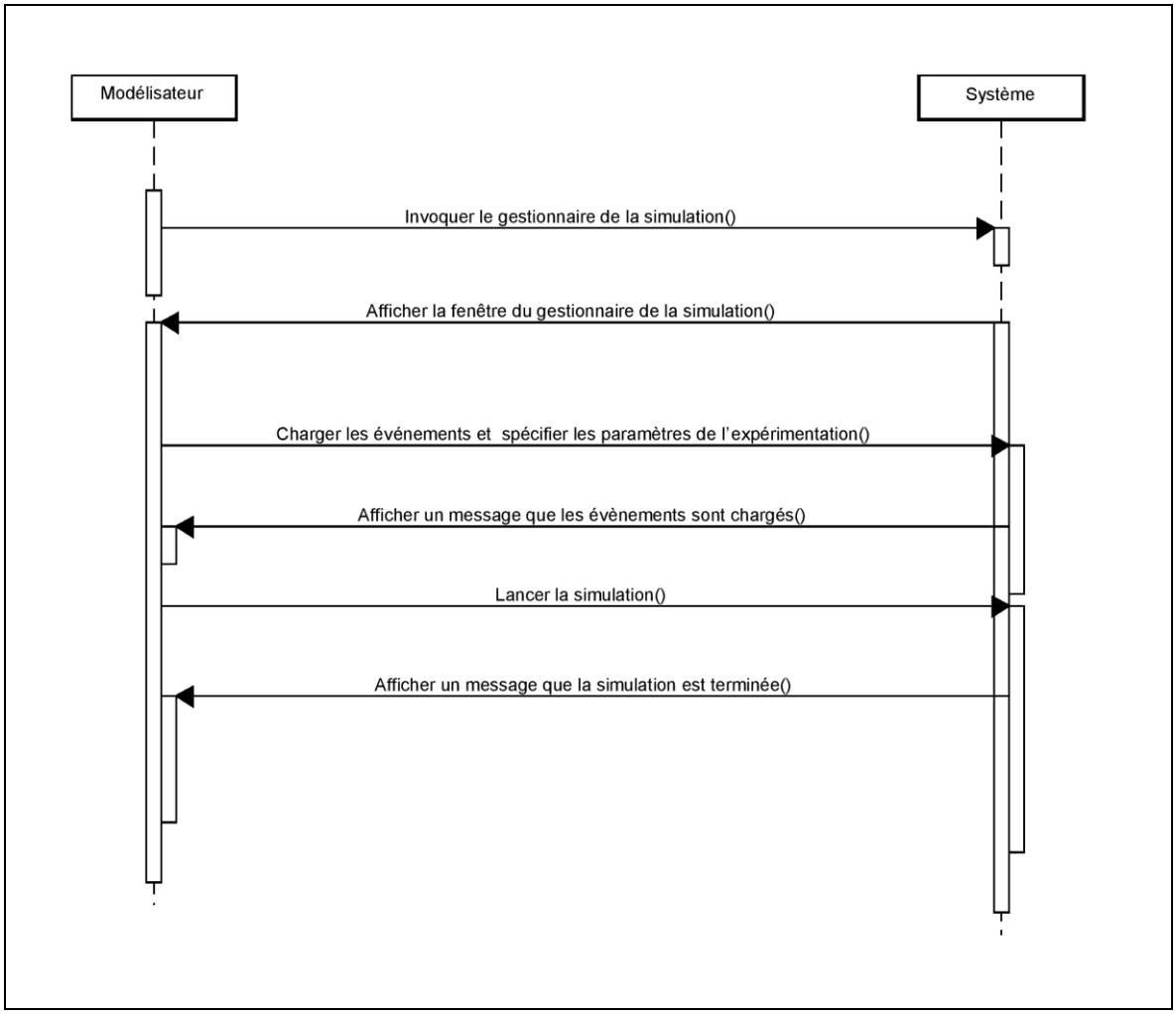

Figure 40 : Diagramme de séquences «Simulation».

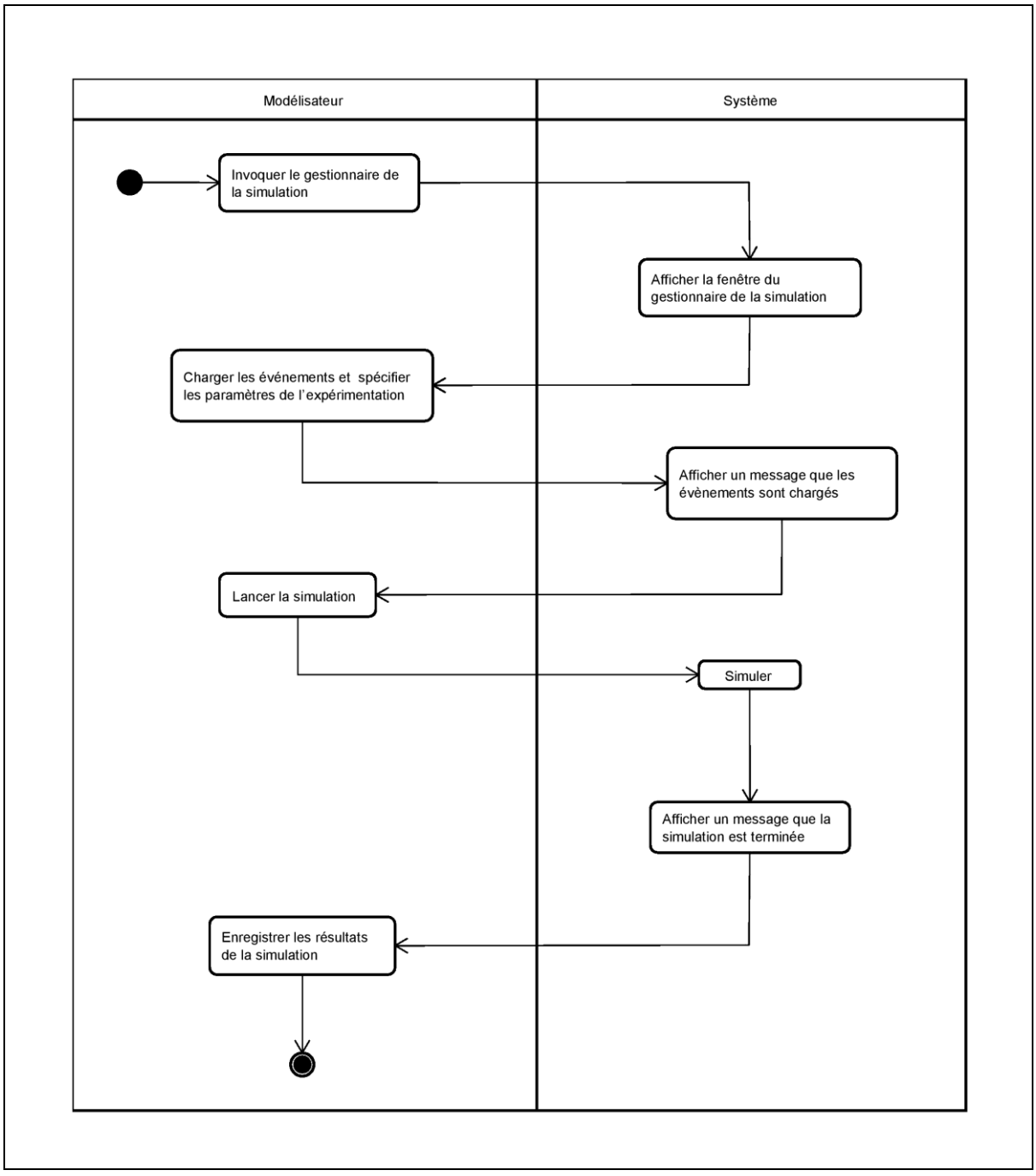

Figure 41 : Diagramme d'activité «Simulation».

5. Diagramme de séquences et d'activité «Sauvegarde d'un projet»

Lorsque le modélisateur veut sauvegarder un projet, le système génère un fichier XML contenant la hiérarchie et la représentation graphique des modèles. Ensuite, il enregistrera le fichier XML dans le dossier de projet. Enfin, le système affichera un message informant le modélisateur que le projet a été sauvegardé.

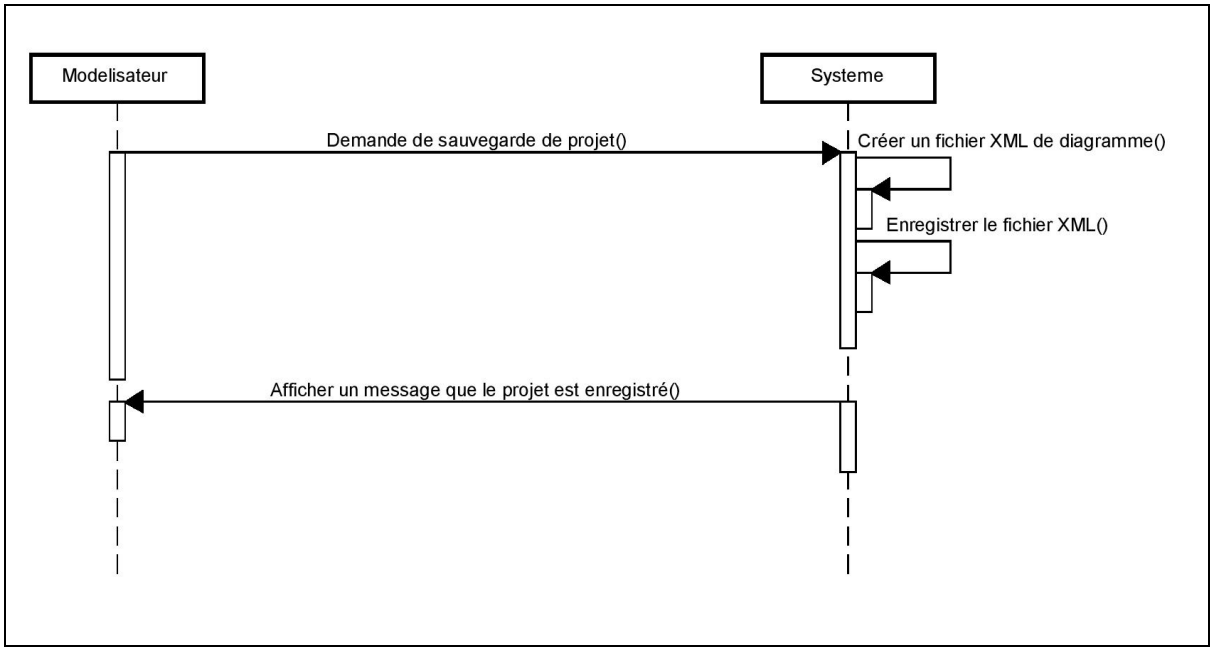

Figure 42 : Diagramme de séquences «Sauvegarde d'un projet».

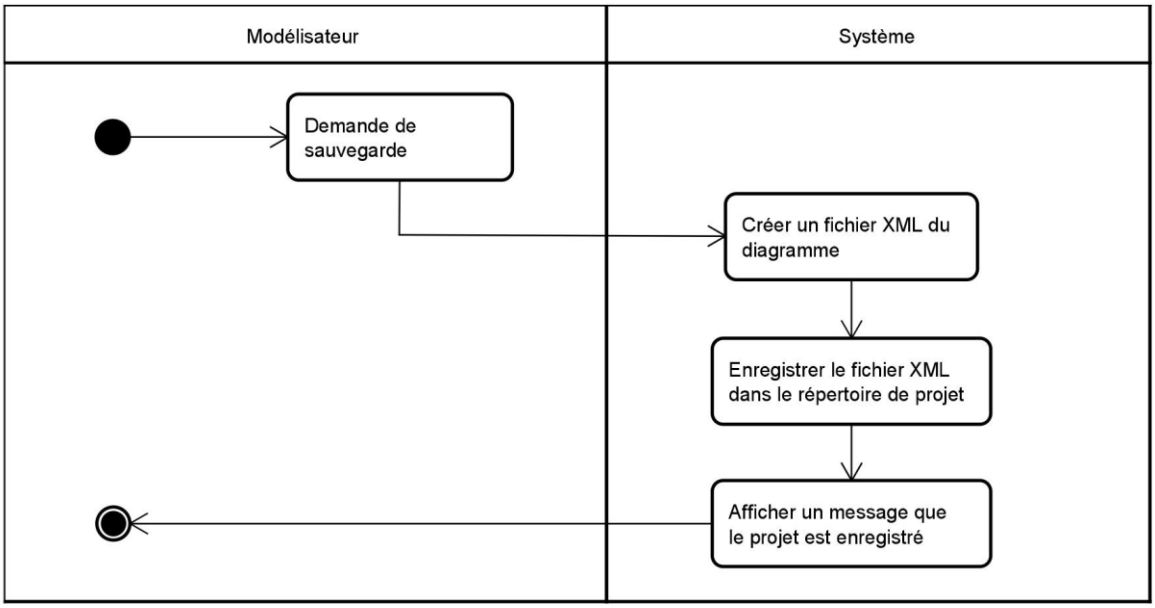

Figure 43 : Diagramme d'activité «Sauvegarde d'un projet».

6. Diagramme de séquences et d'activité «Chargement d'un projet»

Lorsque le modélisateur veut charger un projet déjà sauvegardé, le système affiche la fenêtre de sélection des fichiers. Ensuite, le modélisateur choisira le fichier XML correspondant au projet qu'il veut charger. Le système chargera le fichier XML sélectionné, puis il va vérifier s'il est compatible. Finalement, le système va construire les modèles selon le fichier XML chargé et affichera ces modèles dans l'espace de travail.

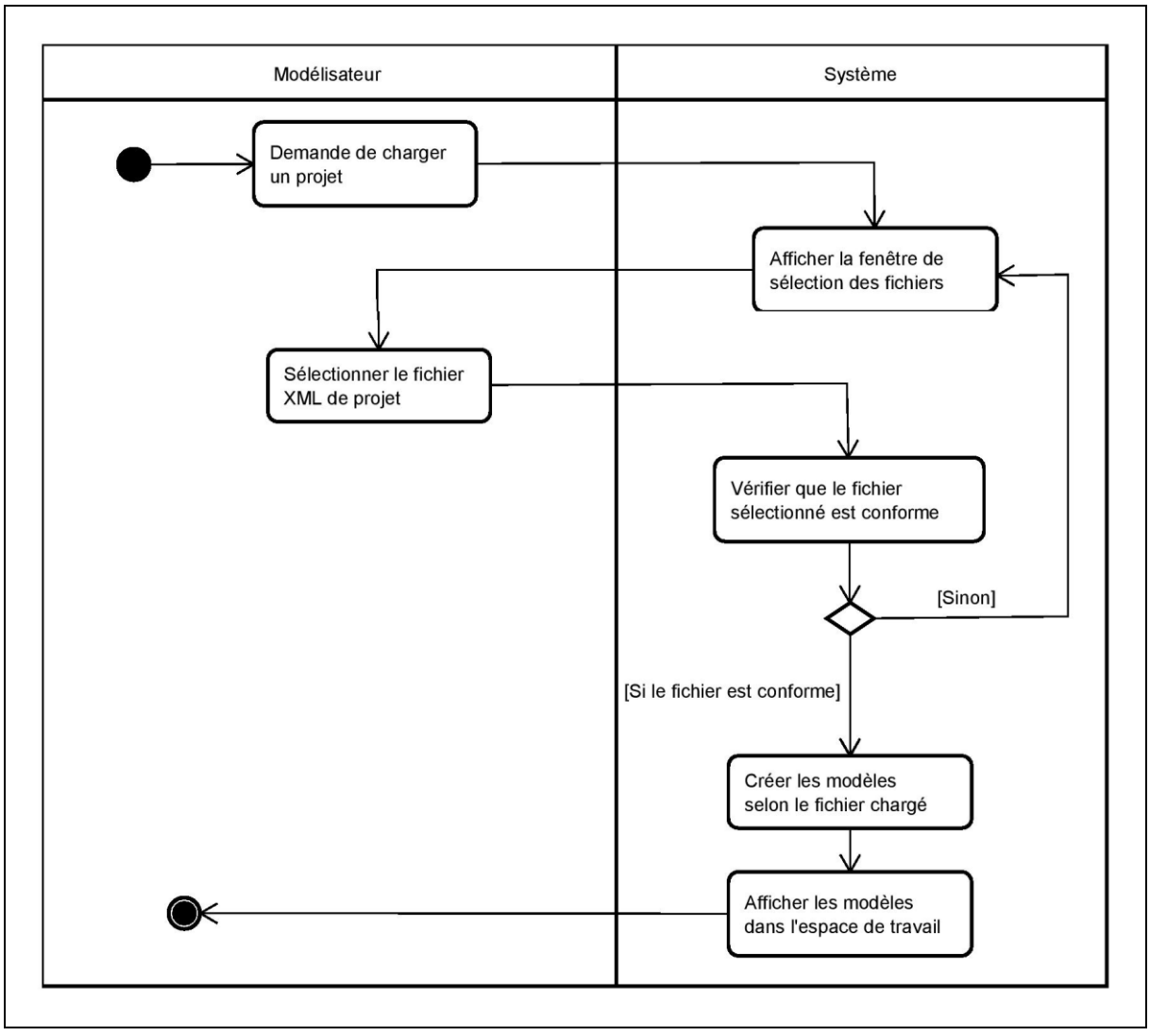

Figure 44 : Diagramme d'activité «Chargement d'un projet».

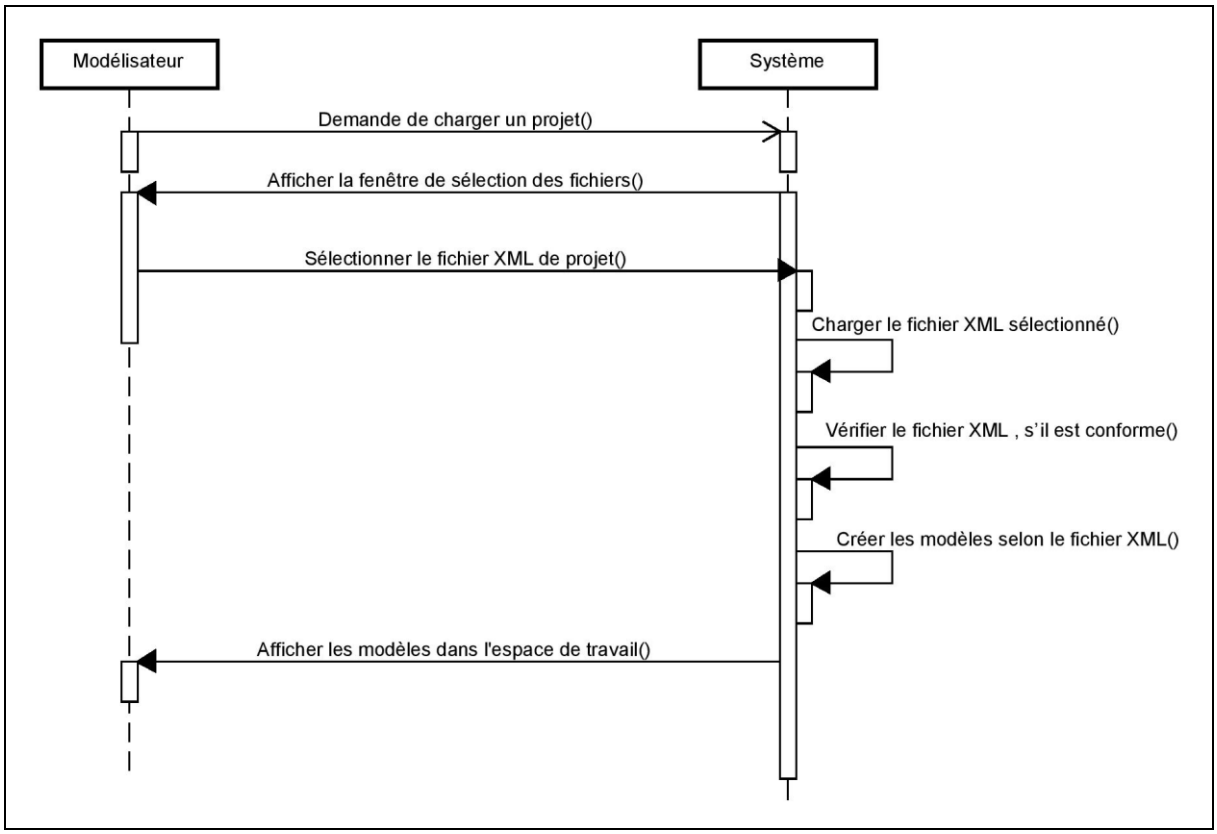

Figure 45 : Diagramme de séquences «Chargement d'un projet».

III. L'architecture du système

Le système est composé de trois packages principaux constituant la partie générique. La figure46 présente ces trois packages, le package moteur-DEVS, le package cadres expérimentaux et le package interface-graphique.

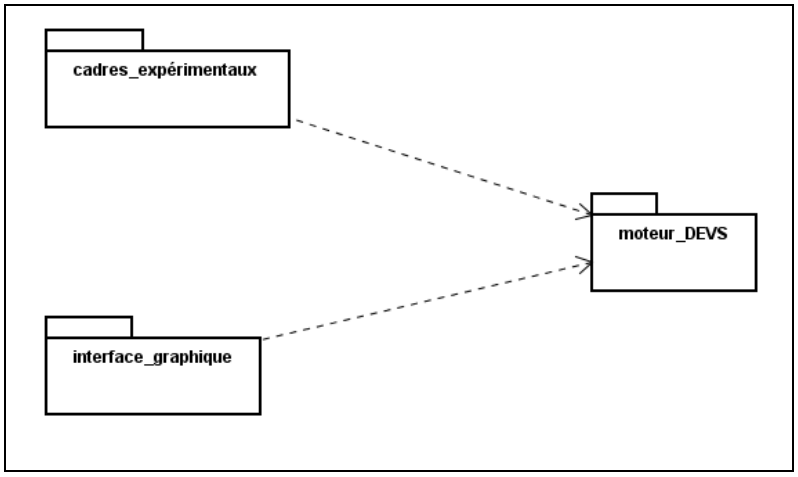

Figure 46 : Diagramme de packages de l'architecture

1. Le package moteur-DEVS

Le package moteur-DEVS (la figure 47) contient les classes nécessaires à la définition des modèles et à leur simulation. L'architecture du package suit l'architecture hiérarchique définie dans le formalisme DEVS [7]. Le package moteur-DEVS est composé de deux sous-packages:

1.1 Le sous-package modélisation

Le sous-package modélisation contient les classes permettant la modélisation de modèles DEVS. Nous pouvons noter que dans ce package, la classe Modèle peut comprendre des objets de la classe Port. En effet tous les couplages de modèles s'effectuent par des liens entre les ports. Chaque objet de la classe Modèle communique avec les autres modèles par la médiation d'objets de la classe Message.

1.2 Le sous-package simulation

Le sous-package simulation contient les classes implémentant les simulateurs abstraits des modèles DEVS. Chaque objet de la classe Modèle du sous-package modélisation est associé à un objet de la classe Processeur associée. Le passage d'informations se fait grâce aux objets de la classe Evènement contenus dans des listes de la classe Vecteur_Evènement.

Tous les modèles créés à l'aide de l'environnement héritent de la classe ModèleAtomique ou ModèleCouplé. Grâce à ces composants de plus haut niveau de spécification et aux interfaces de modèles basées sur les ports, il est possible d'utiliser une interface graphique générique pour aider à la composition de multi-modèles. Le package interface-Graphique présenté dans la section suivante est composé des éléments de base permettant la multi-modélisation.

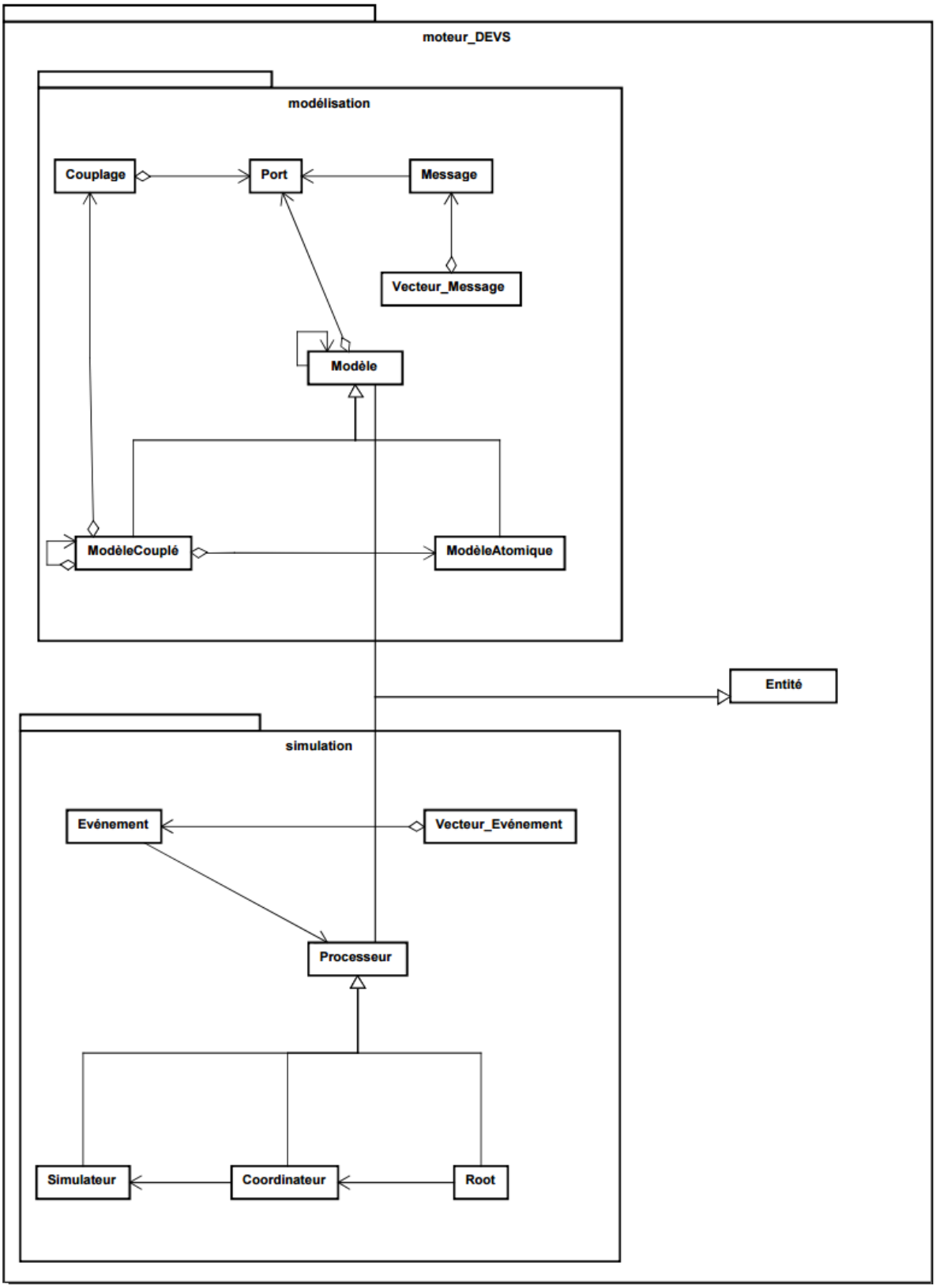

Figure 47 : Diagramme des classes du package moteur-DEVS

2. Le package interface-graphique

Le package interface-graphique contient les composants graphiques permettant la composition visuelle et interactive de modèles. La figure 48 présente les classes composant ce package. A chaque objet de la classe Composant-Modèle correspond un objet de la classe Panel permettant de modifier ses propriétés. Les objets de la classe Composant-Modèle sont contenus et affichés par la classe Espace-de-Travail, qui est le conteneur graphique de plus haut niveau, il contient notamment toutes les méthodes d'interaction utilisateur (ajout, sélection, déplacement et couplage de modèles). Deux classes spécialisent la classe Composant-Modèle, la classe Composant-Modèle-Composition et la classe Composant-Modèle-Atomique. Les Composant-Modèle-Composition sont des conteneurs graphiques pour tous les sous-modèles du modèle associé au composant, ils affichent récursivement tous les sous-modèles (atomiques et/ou couplés) qu'ils contiennent. Les Composant-Modèle-Atomique ont eu un comportement spécifique qui ne peut être défini qu'à l'aide d'un éditeur spécialisé. La classe Application définit l'interface utilisateur telle qu'elle sera utilisée par le modélisateur et contient un objet de la classe Espace-de-Travail lorsqu'un un modèle est chargé en mémoire.

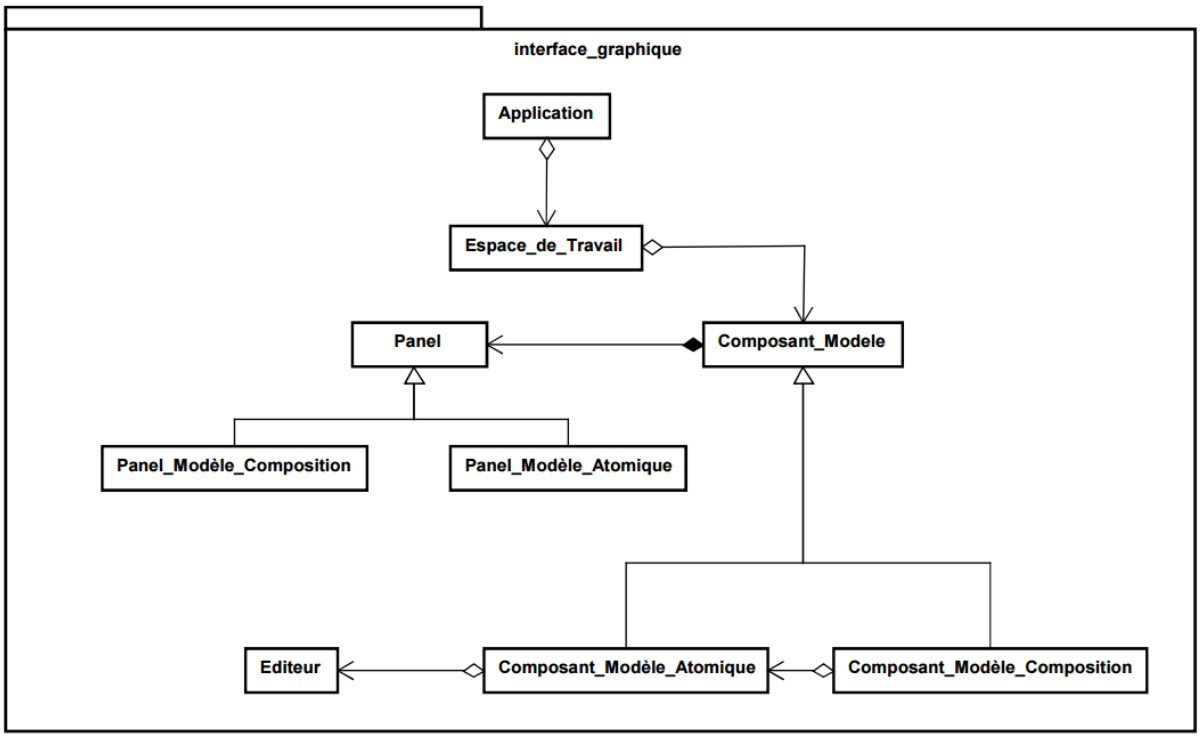

Figure 48 : Diagramme des classes du package interface-graphique

3. Le package cadres-expérimentaux

Le package cadres-expérimentaux (la figure 49) contient les classes nécessaires à la visualisation et au contrôle de la simulation. Comme pour les Composants-Modèle du package interface-graphique, chaque Processeur du package moteur-DEVS est associé à un composant graphique Composant-Processeur, capable d'afficher les propriétés du modèle associé au processeur en cours de simulation. Ainsi, pendant la simulation, l'état des modèles atomiques est affiché par un Composant-Simulateur et celui des modèles de composition par un Composant-Coordinateur.

Les objets de la classe Espace-Simulation sont les conteneurs pour les objets ComposantProcesseur. La classe Espace-Simulation est connectée au Composant-Processeur attaché au processeur de plus haut niveau de l'arbre de simulation (Classe Root du package moteur-DEVS). Chaque objet du type Espace-Simulation possède un Controlleur-de-Simulation qui est attaché au processeur de plus haut niveau (Root) pour lui envoyer tous les événements d'entrée et récupérer tous les événements d'entrée et de sortie générale du modèle dans un objet de type Logger.

La classe Espace-Simulation est chargée des interactions utilisateur et récupère les événements d'entrée (clavier et souris). Ces événements sont, traités par le Composant-Controle, associés à chaque Composant-Processeur et dépendants de la technique utilisée qui détermine comment transformer ces interactions en événements d'entrée du modèle.

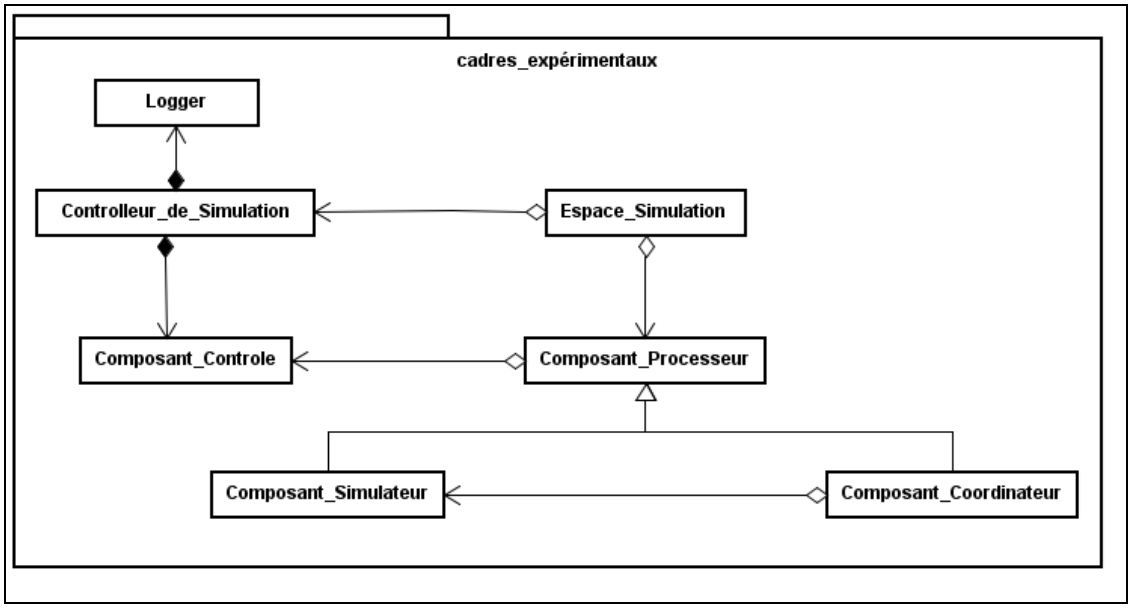

Figure 49 : Diagramme des classes du package cadres-expérimentaux

Conclusion

Dans la première partie de ce chapitre, nous avons décrit les principaux diagrammes, considères les plus importants et les plus adéquats a notre projet, ces diagrammes sont choisis parmi les quatorze diagrammes de conception UML. Et dans la seconde partie nous avons présenté l'architecture de notre système et les trois packages principaux constituant la partie générique.

Nous nous devons de présenter dans le chapitre qui suit, l'implémentation et une représentation des différents modules de notre outil, et la validation du système par un exemple.

Chapitre IV

Implémentation et validation

Introduction

Dans ce chapitre, consacré à l'implémentation et la validation de l'outil JDS (Java DEVS Simulator), nous allons présenter le langage de développement utilisé et montrer les principales interfaces et fenêtres des différents modules de cet outil et ses fonctionnalités. Enfin nous présenterons un exemple d'expérimentation afin de valider l'outil, ainsi sa spécification et son implémentation sous JDS.

I. Langage utilisé

Le langage Java [37] est un langage généraliste de programmation synthétisant les principaux langages existants lors de sa création en 1995 par Sun Microsystems, puis il a été racheté par Oracle Corporation. Il permet une programmation orientée-objet, modulaire et reprend une syntaxe très proche de celle du langage C.

Outre son orientation objet, le langage Java a l'avantage d'être modulaire (on peut écrire des portions de code génériques, c.-à-d. utilisables par plusieurs applications), rigoureux (la plupart des erreurs se produisent à la compilation et non à l'exécution) et portable (un même programme compilé peut s'exécuter sur différents environnements). En contrepartie, les applications Java ont le défaut d'être plus lentes à l'exécution que des applications programmées en C par exemple.

Figure 50 : Logo actuel de Java.

II. Représentation de JDS

JDS (Java DEVS Simulator) est composé de trois modules indépendants, un moteur de simulation, une interface graphique de modélisation et les cadres expérimentaux de visualisation et de simulation.

Figure 51 : Logo de JDS.

1. Le moteur de modélisation et de simulation JDS

Le moteur de simulation JDS est le même utilisé dans JDEVS [34]. Comme il est mentionné dans le chapitre 2, le moteur de modélisation et de simulation est une implémentation de la méthodologie DEVS. Le formalisme DEVS propose des interfaces bien définies pour la description de systèmes. Ces interfaces nous permettent de pouvoir utiliser des modèles programmés dans de nombreux langages orientés objets et d'y accéder ensuite grâce à des appels de méthodes à distance (comme Java RMI). La modélisation de modèles atomiques peut toutefois se faire directement dans l'interface et en utilisant Java.

2. L'interface graphique JDS

La figure 52 montre la fenêtre d'accueil telle qu'elle apparaîtra à l'utilisateur au cours de la première exécution. Cette fenêtre permet à l'utilisateur de choisir de créer un nouveau projet ou de charger un projet existant.

Figure 52 : La fenêtre d'accueil de JDS.

Après le lancement du projet par l'utilisateur, l'interface de modélisation JDS (Figure 53) s'affiche, et permet aux utilisateurs de construire graphiquement leurs modèles. Cette interface se compose de quatre volets, l'espace de travail, le panneau de propriétés de modèle, la barre d'outils et la console. L'implémentation de cette interface est basée sur la modélisation décrite dans le chapitre précédant. A l'intérieur de L'espace de travail (partie centrale de la figure 53), le modélisateur peut construire les modèles à simuler. Le panneau de propriétés de modèle (situé au-dessus de l'espace de travail) est un panneau dédié à la gestion des propriétés de modèle sélectionné dans l'espace de travail et permet d'éditer rapidement ses propriétés. De plus, l'interface dispose d'une barre de menu simple rassemble les raccourcis des principales actions comme ajouter un nouveau modèle, lancer un nouveau projet, sauvegarder le projet en cours (en XML), charger les projets existants, prend une capture d'écran de l'espace de travail et la simulation. La console est un panneau de texte qui affiche des messages à propos des opérations comme la génération de code d'un modèle atomique et de sa compilation.

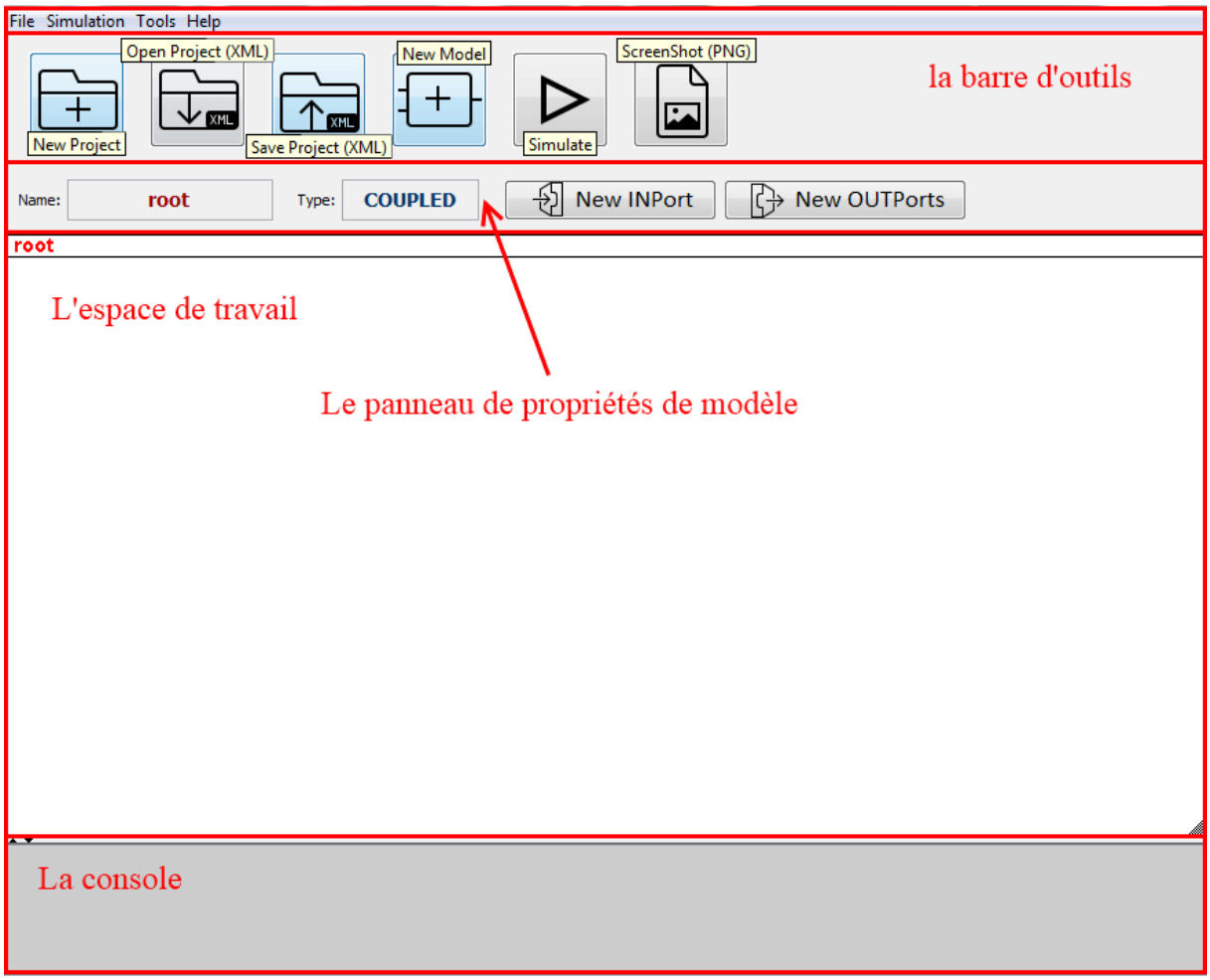

Figure 53 : L'interface de modélisation de JDS.

L'utilisateur a la possibilité de construire ses modèles en utilisant l'interface graphique de JDS. Il suffit alors d'invoquer le gestionnaire de création des modèles par un clic droit sur un modèle couplé et choisir **New Model**, ou par un clic sur le bouton de nouveau modèle sur la barre d'outils qui va venir accueillir le nouveau modèle. La première étape consiste à choisir le type de modèle (atomique ou couplé) (la figure 54 à gauche), ensuite le gestionnaire de création des modèles permet de renseigner les informations nécessaires à la création d'un modèle (son nom, le nombre de ses ports d'entrée et de sortie, sa taille et ses coordonnées) (la figure 54 à droite). Finalement le modèle créé peut être visible sur L'espace de travail (la figure 55).

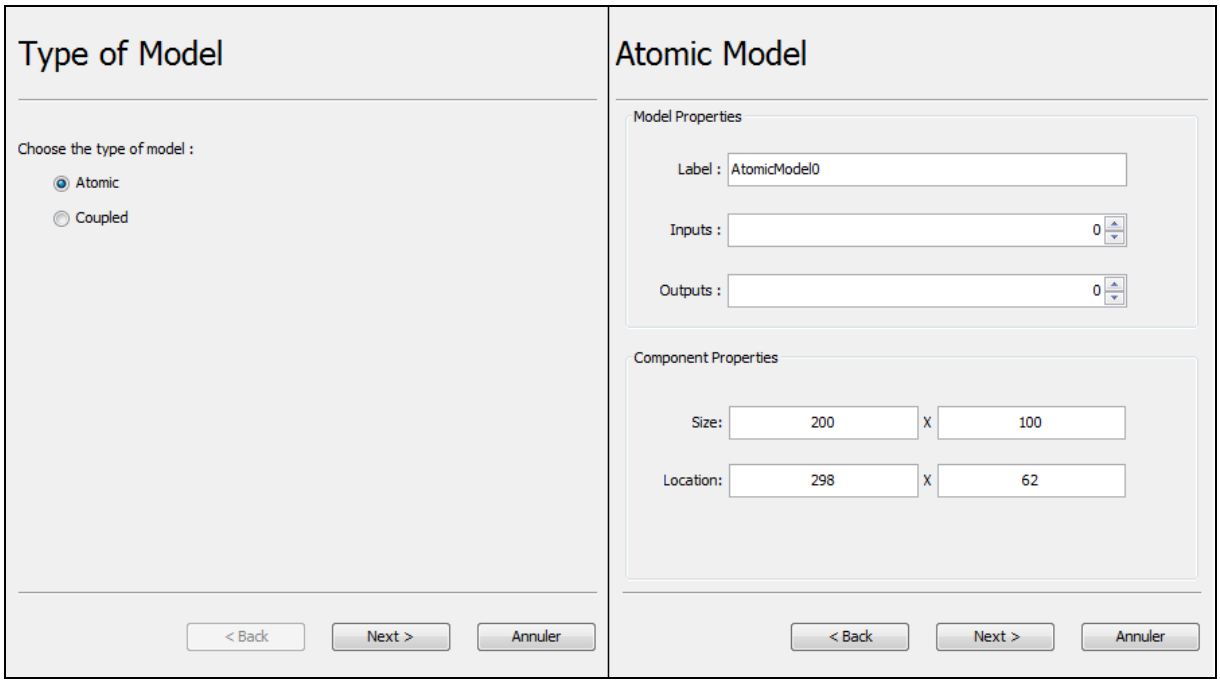

Figure 54 : Le gestionnaire de création des modèles.

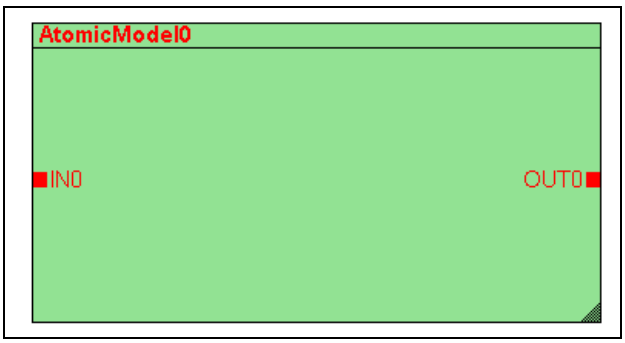

Figure 55 : Un modèle atomique créé par JDS.

Si le modèle créé est un modèle atomique, un fichier source est créé et stocké contenant la classe du modèle au format .java (par exemple AtomicModel0.java) qui contient son comportement. La différence principale entre les modèles atomiques et couplés dans JDS est que le modèle couplé ne possède pas de comportement et donc il ne sera pas nécessaire de générer son fichier source.

Lorsque l'utilisateur veut ajouter un port sur un modèle il doit seulement cliquer sur le bouton **NewINPort** ou **NewOUTPort** qui se trouvent dans le panneau de propriétés de

modèle et créer le port par son nom à travers l'intermédiaire du gestionnaire de création des ports (la figure 56). Un port est représenté par une poignée rouge dans un composant et les couplages s'effectuent en glissant le curseur de ports source vers destination.

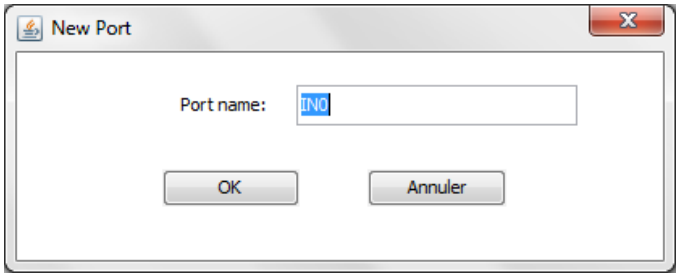

Figure 56 : Le gestionnaire de création des ports.

Les modèles couplés et atomiques sont notés par défaut **AtomicModeli** et **CoupledModelj** avec **i** le numéro du modèle atomique et **j** le numéro du modèle couplé (par exemple AtomicModel0 et CoupledModel0). Et de la même manière les ports d'entrée et de sortie sont notés par défaut **INi** et **OUTj** avec **i** le numéro du port d'entrée et **j** le numéro du port de sortie (par exemple IN0 et OUT0).

2.1 Propriétés d'un modèle atomique

Une fois le modèle atomique créé, l'utilisateur a la possibilité de manipuler et d'ajouter des ports et des propriétés par son panneau de propriétés (la figure 57). Lorsque l'utilisateur manipule un modèle atomique créé, il accède au code du modèle par un clic sur le bouton **Source Code**.

Figure 57 : Le panneau de propriétés d'un modèle atomique.

La figure 58 présente le code Java généré automatiquement pour le modèle atomique AtomicModel0 = < X(IN0), S (status), Y(OUT0), δint, δext, λ, ta >, nous pouvons voir les importations nécessaires à la classe du modèle au début du code. La classe présente le constructeur qui permet de définir les variables d'états (status). Devant ce constructeur se trouve la déclaration de ports du modèle (IN0 et OUT0). Les fonctions de transition sont implémentées grâce aux méthodes **intTransition** et **extTransition**. La fonction de sortie est implémentée par la méthode **outFunction**. Les fonctions de sortie et de transition externe renvoient des vecteurs d'événements qui sont ajoutés à la liste générale des événements. La

seule tâche de programmation qui incombe au modélisateur, est de spécifier la dynamique des modèles atomiques à travers ses trois méthodes.

```
import java.util.Vector;
import devsjava.modelisation.model.*;
import devsjava.simulation.*;
import devsjava.simulation.processor.*;
import devsjava.modelisation.*;
import devsjava.modelisation.message.*;
import devsjava.modelisation.model.*;
import devsjava.*;
import devsjava.simulation.EventVector;
import devsjava.modelisation.model.*;

public class AtomicModel0 extends AtomicModel {

    // INPUT Ports
    Port IN0 = new Port(this, "IN0", "IN");
    // OUTPUT Ports
    Port OUT0 = new Port(this, "OUT0", "OUT");

    public AtomicModel0() {
        super("AtomicModel0");
        // Properties
        states.setProperty("status", "");
    }

    @Override
    public EventVector outFunction(Message m) {
        // TODO add your outFunction code here:
        return new EventVector();
    }

    @Override
    public void intTransition() {
        // TODO add your intTransition code here:
    }

    @Override
    public EventVector extTransition(Message m) {
        // TODO add your extTransition code here:
        return new EventVector();
    }
}
```

Figure 58 : Un exemple d'un code Java généré automatiquement pour le modèle atomique.

Apres l'implémentation et la sauvegarde, l'utilisateur peut compiler ce code par un clic sur le bouton **Compile** (la figure 57), une vérification syntaxique est réalisée, Si une erreur syntaxique est présente, un message apparait dans la console en bas du l'espace de travail pour aider l'utilisateur dans le débogage. L'utilisateur a toujours la possibilité d'accéder au fichier source édité et le recompiler en cas de problème.

2.2 Création, Utilisation et Sauvegarde des diagrammes

Le diagramme est un modèle couplé par définitions, dans JDS le diagramme principal d'un projet est le modèle couplé **root** (la figure 59). Il est défini comme un ensemble de modèles atomiques et couplés interconnectés, et il se construit en instanciant et en connectant des modèles sur l'espace de travail.

Lorsque le diagramme est construit, il est possible de le sauvegarder dans un fichier XML. Pour sauvegarder un diagramme, il suffit d'invoquer le menu **File** puis **Save Project** ou en cliquant sur le bouton **Save Project** de la barre d'outils. Enfin l'utilisateur peut ouvrir un diagramme existant soit en cliquant sur le bouton **Open Project** soit par le menu **File** puis **Open Project**.

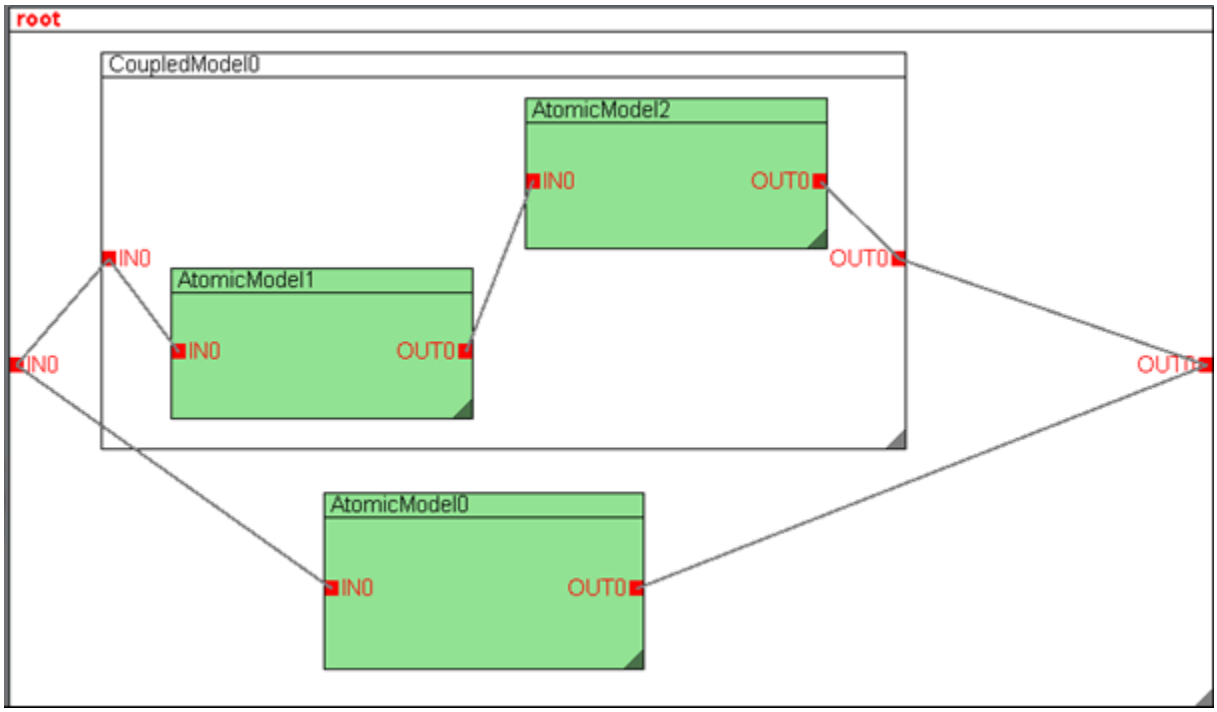

Figure 59 : le diagramme principal d'un projet.

3. Cadres expérimentaux

Une fois que le diagramme est construit, il est possible de le simuler. Les cadres expérimentaux sont les interfaces utilisateur de simulation des modèles. Les expérimentations sur des modèles en diagramme peuvent être effectuées directement depuis l'interface graphique de modélisation.

La figure 60 présente un modèle en cours de simulation dans l'interface JDS. Le lancement de la simulation depuis l'interface graphique permet de charger une liste d'événements d'entrée et de spécifier les paramètres de l'expérimentation, ces actions sont effectuées depuis le gestionnaire de simulation (la figure 61), qui est invocable en cliquant sur le bouton **Simulate** de la barre d'outils. La liste des événements à traiter est affichée à l'intérieur du modèle couplé **root** et évolue au cours de la simulation.

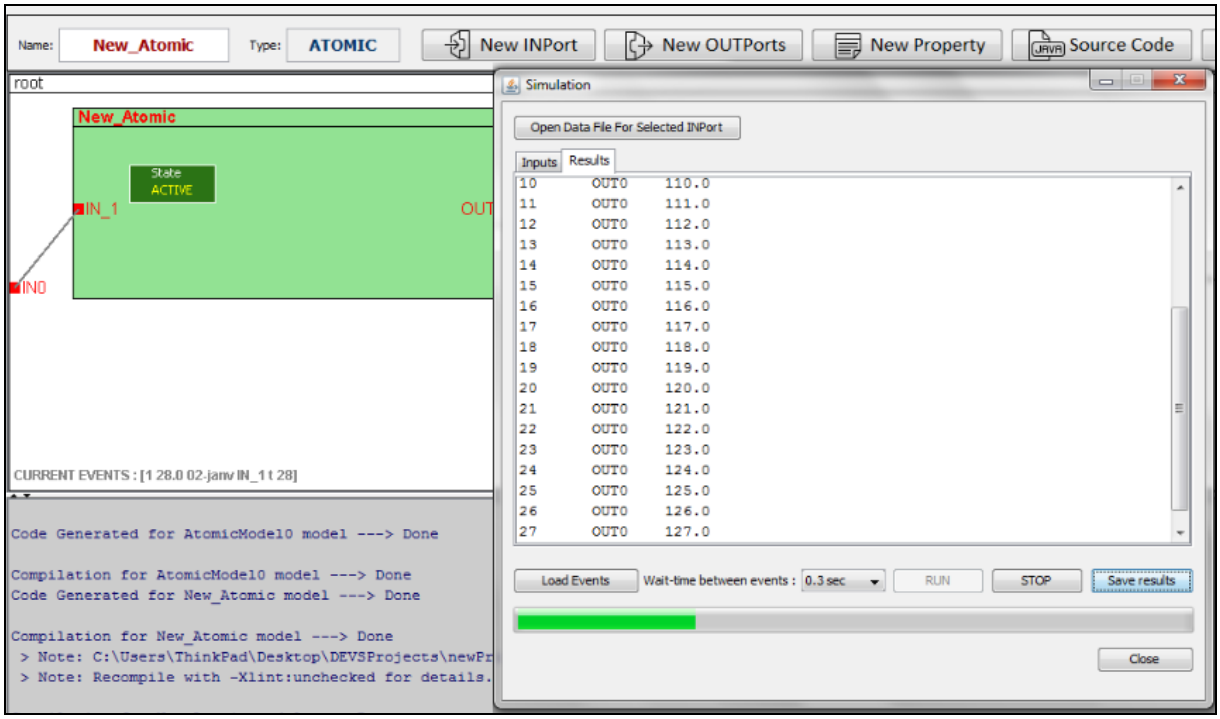

Figure 60 : Un modèle en cours de simulation dans l'interface JDS.

Le gestionnaire de simulation (la figure 61) propose une liste déroulant permettant de renseigner le temps restant entre les événements (0 par défaut). La simulation démarre après

avoir cliqué sur le bouton **RUN**. Une barre de progression montrera l'avancée de la simulation avec un compteur affiché en bas à gauche de la fenêtre. Un clic sur le bouton **STOP**, arrête la simulation définitivement. Le résultat de simulation est une liste d'événements de sortie qui peut être affichée (la figure 60) et l'utilisateur peut le sauvegarder dans un fichier texte en cliquant sur le bouton **Save results**.

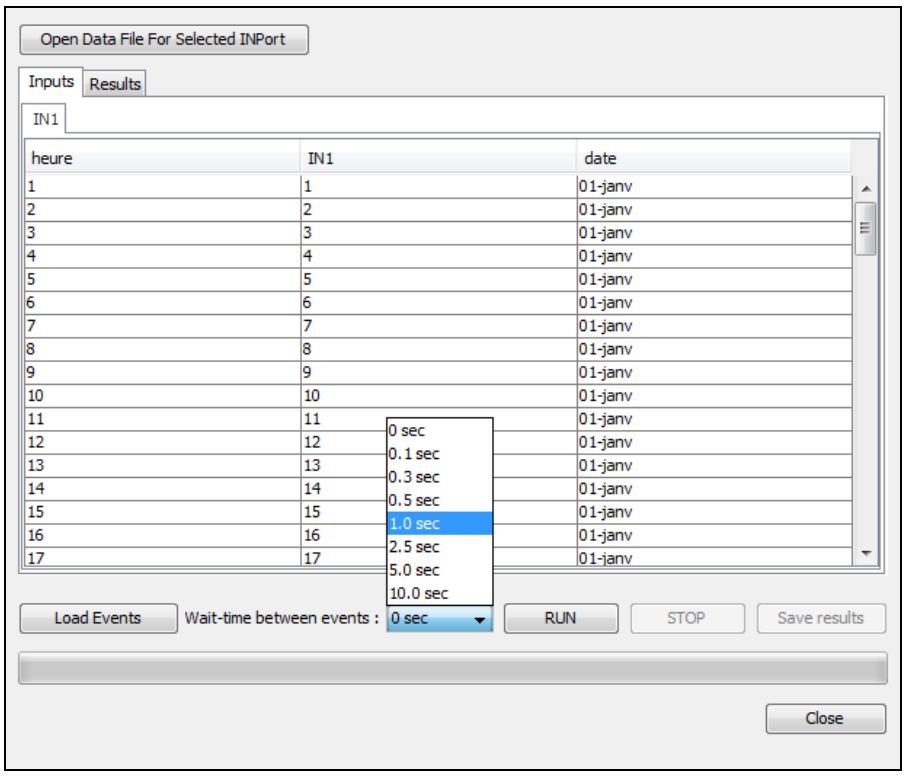

Figure 61 : Le gestionnaire de simulation.

La simulation ne se fera que si tous les modèles atomiques sont compilés et que tous les ports sont coordonnées.

4. Fonctionnalités avancées

4.1 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 (atomique ou couplé) est invoqué par un clic-droit sur le modèle (la figure 62). Les actions sont **NewInputPort**, **NewOutputPort**, **Newproperty**, **SourceCode** et **Compile** pour les modèles atomiques et **NewInput** Port, **NewOutput** Port et **NewModel** pour les modèles couplés.

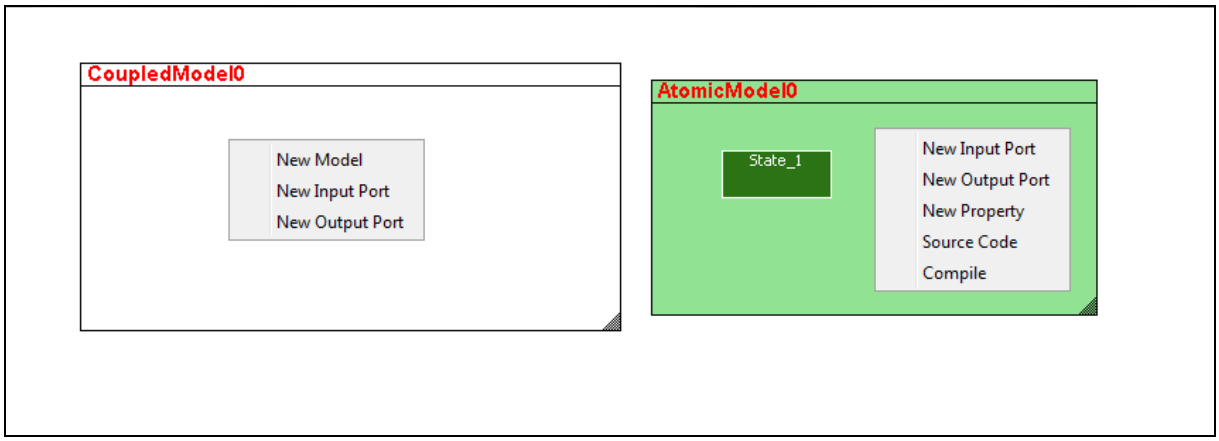

Figure 62: Le menu contextuel des actions possibles sur un modèle (atomique ou couplé).

4.2 Edition du code

L'édition du code Java d'un modèle atomique est possible par la fenêtre d'édition du code (la figure 63) qui permet de modifier celui-ci et d'enregistrer les modifications. La numérotation des lignes facilite la lecture dans le code. De plus, chaque enregistrement d'un code implique auparavant sa compilation, donc une vérification syntaxique. En cas d'erreur, une information sera affichée dans le panneau console informant l'utilisateur du type d'erreur et de la ligne concernée par celle-ci.

4.3 Connexion entre les ports

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-gauche de la souris. Lorsque le curseur approche du port de destination, il suffit de relâcher le clique-gauche de la souris pour rendre effective la connexion.

File

```
import devsjava.modelisation.model.*;
import devsjava.*;
import devsjava.simulation.EventVector;
import devsjava.modelisation.model.*;

public class AtomicModel0 extends AtomicModel {


 // INPUT Ports
 // OUTPUT Ports



public AtomicModel0() {
      super("AtomicModel0");
 // Properties
}
     public EventVector outFunction(Message m) {
        // TODO add your outFunction code here:
        return new EventVector();
     }

     public void intTransition() {
       // TODO add your intTransition code here:
     }

     public EventVector extTransition(Message m) {
       // TODO add your extTransition code here:
       return new EventVector();
     }
 }

```

Figure 63: La fenêtre d'édition du code.

III. Le stockage

XML est un langage basé sur les balises ou la structure du document (balises, cardinalités et relations) est définie strictement, soit à l'aide d'un schéma XML (XML Schema), soit à l'aide d'un DTD (Document Type Définition ou définition du type de document).

La structure de fichier devant définir les modèles DEVS n'est toutefois pas encore standardisée. Donc nous avons utilisé le DTD (la figure 64) utilisé dans JDEVS pour permettre le stockage de ces modèles.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE MODEL [
      <!ELEMENT MODEL (TYPE, NAME, CLASS ?, BOUNDS ?, INPUT*,
           OUTPUT*, CHILD*, EIC ?, EOC ?, IC ?, EXECUTIVE ?)>
           <!ELEMENT NAME (#PCDATA)>
           <!ELEMENT TYPE (#PCDATA)>
           <!ELEMENT CLASS (URI)>
           <!ELEMENT BOUNDS (LOCX, LOCY, WIDTH, HEIGHT)>
           <!ELEMENT LOCX(#PCDATA)>
           <!ELEMENT LOCY (#PCDATA)>
           <!ELEMENT WIDTH (#PCDATA)>
           <!ELEMENT HEIGHT (#PCDATA)>
           <!ELEMENT PORT (URI ?, #PCDATA)>
           <!ELEMENT URI (#PCDATA)>
           <!ELEMENT CHILD (MODEL* | URI*)>
           <!ELEMENT INPUT (PORT*)>
           <!ELEMENT OUTPUT (PORT*)>
           <!ELEMENT LINK (PORT, PORT)>
           <!ELEMENT EIC (LINK*)>
           <!ELEMENT EOC (LINK*)>
           <!ELEMENT IC (LINK*)>
           <!ELEMENT EXECUTIVE (MODEL* | URI*)> ]
>
```

Figure 64: Document de définition de format XML des modèles couplés.

Dans ce DTD XML la balise **NAME** est le nom du modèle, la balise **TYPE** contient le type de modèle qui est définit par le fichier (Atomique et couplé). Dans le cas d'un modèle atomique, **CLASS** est le lien URI (Uniform Ressource Identifier) vers la ressource contenant, la balise **BOUNDS** est utilisée uniquement par l'interface graphique pour dessiner le modèle dans un canevas, **INPUT** est l'ensemble des ports d'entrée, **OUTPUT** l'ensemble des ports de sortie. La balise **CHILD** est l'index des composants du modèle couplé (dans l'ordre de priorité) ces composants sont, soit décrits récursivement dans des balises de type **MODEL**, soit par un lien (balise URI) vers le fichier XML correspondant. Les couplages sont définis par des balises **LINK** prenant un couple de **PORT** précédés par le lien URI du modèle auxquels ces ports appartiennent s'il ne s'agit pas de ports concernant le modèle défini par le fichier. **EIC** est l'ensemble des couplages d'entrée, **IC** est l'ensemble des couplages internes et **EOC** est l'ensemble des couplages de sortie.

IV. Expérimentation d'un contrôleur de carrefour à feux

Le modèle DEVS d'un feu de circulation possède deux ports d'entrées (button et signal), et un seul port de sortie (out), comme montré sur la figure 65.

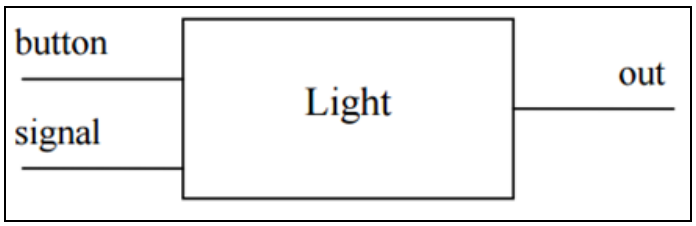

Figure 65: Le modèle DEVS d'un feu de circulation.

Le port d'entrée **button** est destiné à recevoir un événement d'entrée externe de personnes qui veulent traverser une intersection quand le feu de circulation est en vert. Le port d'entrée **signal** est destiné à recevoir un message d'entrée externe ou une sortie d'autre feu de circulation. Le feu de circulation a trois phases, rouge, vert et orange, il sera en rouge pendant un certain temps (30 sec), puis il passe au vert (25 sec),après il passera à l'orange pendant un certain temps aussi (5 sec) et finalement il retour au rouge et ainsi de suite. Par exemple si le feu de circulation est en vert et recevoir un message sur le port d'entrée **button**, il passera au rouge puis envoyer un message via le port de sortie out à l'autre feu de circulation qui reçoive ce message sur le port **signal** pour devenir vert.

1. La spécification DEVS

Une spécification DEVS d'un feu de circulation peut être décrite comme,

FA = (X, S, Y, δint, δext, λ, ta) où :

- X = {"button", "signal"};
- Y = {"out"};
- S = {"RED", "GREEN", "ORANGE"};
- δint ("RED",σ) = "GREEN", 25);
- δint ("GREEN",σ) = "ORANGE", 5);
- δint ("ORANGE",σ) = "RED", 30);
- δext ("GREEN",σ,e,x) = ("RED", 30);

- δext ("RED",σ,e,x) = ("GREEN", 25);
- λ ("RED",σ) = “Envoyer un signal à l'autre feu de circulation”;
- ta (phase, σ) = σ.

2. L'implémentation dans JDS

Chaque feu de circulation est représenté par un modèle atomique (la figure 67), et deux feux de circulation sont connectés entre eux dans un modèle couplé qui représente un contrôleur de carrefour à feux. L'implémentation d'un seul feu de circulation en JDS se présente dans la figure67.

La figure 66 montre un modèle couplé représentant notre système de contrôleur de carrefour à feux. Notre système a deux feux de circulation connectés entre eux. Le port de sortie **out** du premier feu (TrafficLightA) est connecté avec le port d'entrée signal du deuxième feu(TrafficLightB), et le port de sortie out du deuxième feu(TrafficLightB) est connecté avec le port d'entrée signal du premier feu(TrafficLightA). Le premier feu (TrafficLightA) est initialisé avec la phase RED (rouge), et le deuxième feu (TrafficLightB) est initialisé avec la phase GREEN (vert) et ils ont différents temps restant qui peuvent être définis par l'utilisateur.

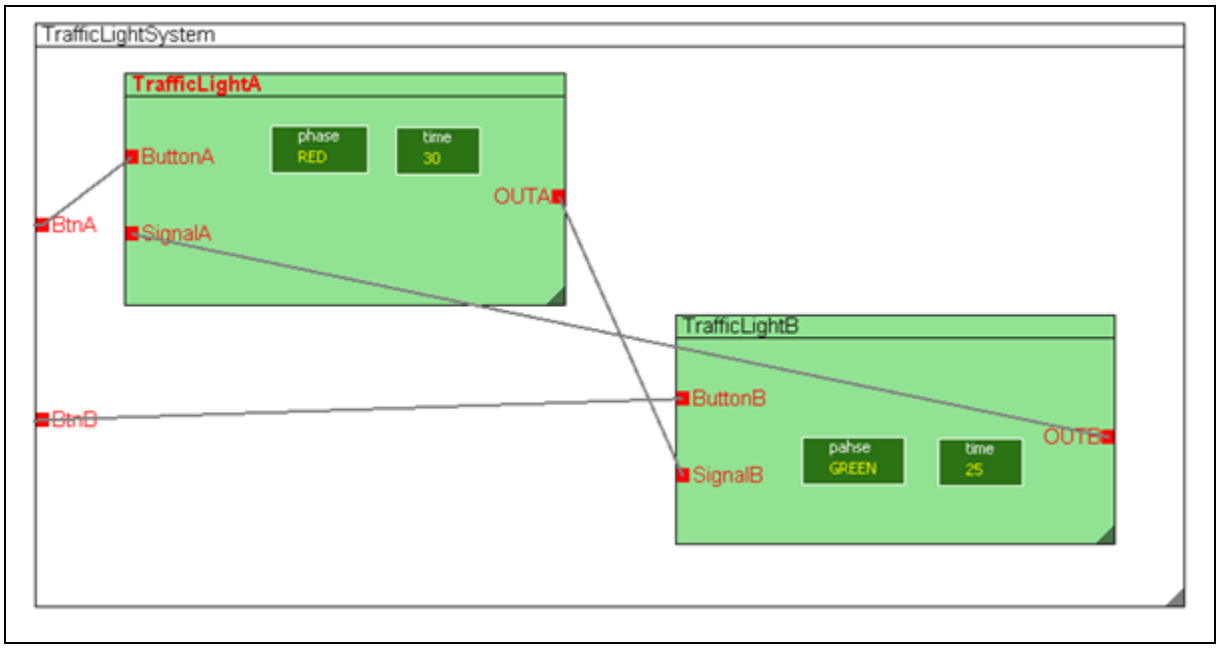

Figure 66: Le modèle couplé représentant le système de contrôleur de carrefour à feux.

```
public class TrafficLightA extends AtomicModel {
    // INPUT Ports
    Port ButtonA = new Port(this, "ButtonA", "IN");
    Port SignalA = new Port(this, "SignalA", "IN");
    // OUTPUT Ports
    Port OUTA = new Port(this, "OUTA", "OUT");

    int t;
    String phase;
    boolean signal;

    public TrafficLightA() {
        super("TrafficLightA");
        signal=false;
        phase = "RED";
        t=30;
        states.setProperty("phase", phase);
        states.setProperty("time", String.valueOf(t));}

    public EventVector outFunction(Message m) {
        EventVector e = new EventVector();
        if(signal){
            e.add(new Event(time,OUTA,new Information(1))) ;
            signal=false;}
        return e;}

    public void intTransition() {
        if(t==0){
            if(phase.equals("RED")){
                phase = "GREEN"; t=25;
                states.setProperty("phase", phase);
                states.setProperty("time", String.valueOf(t));
            } else if(phase.equals("GREEN")){
                phase = "ORANGE";
                t=5; states.setProperty("phase", phase);
                states.setProperty("time", String.valueOf(t));
            } else if(phase.equals("ORANGE")){
                phase = "RED"; t=30;
                states.setProperty("phase", phase);
                states.setProperty("time", String.valueOf(t)); } }
        states.setProperty("time", String.valueOf(t)); }

    public EventVector extTransition(Message m) {
        t--;
        EventVector e = new EventVector();
        if (m.getPort() == SignalA) {
            if (states.getProperty("phase").equals("RED")) {
                phase = "RED"; t=30;
                states.setProperty("phase", phase);
                states.setProperty("time", String.valueOf(t));
             } else if (states.getProperty("phase").equals("GREEN")) {
                phase = "RED"; t=30;
                states.setProperty("phase", phase);
                states.setProperty("time", String.valueOf(t)); } }

        if (m.getPort() == ButtonA) {
            if(m.getInfo().getValue() == 1){
                signal=true;
                if (states.getProperty("phase").equals("RED")) {
                    phase = "GREEN"; t=25;
                    states.setProperty("phase", phase);
                    states.setProperty("time", String.valueOf(t));
                } else if (states.getProperty("phase").equals("GREEN")) {
                    phase = "GREEN"; t=25;
                    states.setProperty("phase", phase);
                    states.setProperty("time", String.valueOf(t)); } } }

        e.add(new Event(time, OUTA, new Information()));
        return e;
    }
}
```

Figure 67: Le code généré pour un feu de circulation.

Conclusion

Dans ce chapitre nous avons présenté notre outil de modélisation et de simulation basée sur le formalisme DEVS JDS (Java DEVS Simulator), et présenté les différents modules de cet outil, y compris les modules de la modélisation et la simulation. Enfin nous avons illustré un exemple d'expérimentation implémenté dans JDS afin de valider ce dernier.

Conclusion Générale

Dans ce mémoire nous avons présenté les concepts clés qui fondent la méthodologie de modélisation et de simulation et les étapes de la construction d'un modèle et ce, sous les deux aspects complémentaires de la théorie de la modélisation et de la simulation qui constituent, les niveaux et les formalismes de spécification d'un système. Aussi, nous avons illustré les principales implémentations logicielles existantes dans le domaine de la modélisation et de la simulation de systèmes complexes basés sur le formalisme DEVS (DEVSJAVA, DEVSuite, CD++, PowerDEVS, DEVSimpy et JDEVS).

Ensuite nous avons présenté la modélisation, l'implémentation et la validation de notre outil de modélisation et simulation à évènements discrets JDS implémenté en Java. Dans la deuxième partie de ce mémoire nous avons détaillé l'architecture et toutes les fonctionnalités implémentées dans JDS. Enfin nous avons validé cet outil par un exemple d'expérimentation implémenté sous JDS.

Comme perspectives de notre travail nous envisagerons d'intégrer les outils de SoftComputing (réseaux de neurones, logique flou etc…) dans le système JDS afin le rendre un outil de multi-modélisation pour modéliser les systèmes complexes a paramètres incertains.

Bibliographie

[1]: M. Minsky, Matter, minds and models, Marvin Minsky Ed, 1968.

[2]: A. Wymore, A Mathematical Theory of Systems Engineering - The elements, Krieger Publishing, 1977.

[3] : J. Hubbard, Equations différentielles et systèmes dynamiques,Vuibert. ISBN: 284225015X, 1999.

[4] : E. Kofman, Quantized state systems. A DEVS approach for continuous systems simulation, Transactions of SCS, 2001, pp.123–132.

[5] : B. Zeigler, DEVS theory of quantization, Rapport Technique N6133997K- 0007 ECE Dept. Université d'Arizona DARPA, 1998.

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

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

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

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

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

[11] : E. Kofman et S. Junco, Quantized State Systems. A DEVS Approach for Continuous System Simulation, Transactions of SCS vol. 3, 2000, pp. 123-132.

[12] : E. Kofman, A Second Order Approximation for DEVS Simulation of Continuous Systems, Simulation: Transactions of the Society for Modeling and Simulation International vol. 2, 2002, pp. 76-89.

[13] : 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, 2006, pp. 945-970.

[14] : 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.

[15] : Orën, Dynamic templates and semantic rules for advisors and certifiers. Knownledge Based Simulation : Methodology and application., 1991, pp. 53–76.

[16] : Fishwick P., Simulation Model Design and Execution : Building Digital Worlds. Prentice Hall, englewood cliffs edition, 1995.

[17] H. S. Sarjoughian et B. P. Zeigler. DEVSJAVA: Basis for a DEVS-based collaborative m&s environment. Simulation Series, 1998, pp.29–36.

[18] H. S. Sarjoughian et Ranjit.S., 'Software Architecture for Object-Oriented Simulation Modeling and Simulation Environments: Case Study and Approach', Université d'arizona, Tucson, Septembre 2003, pp.6-7.

[19] Sarjoughian, H. S. et R. Singh, Building Simulation Modeling Environments Using Systems Theory and Soft-ware Architecture Principles, Proceedings of the Advanced Simulation Technology Conference Washington DC USA, 2004, pp. 235-240.

[21] Kim S.,Sarjoughian H. S. et Elamvazhuthi V., 'DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring', Université d'arizona Tucson, pp.2.

[23] Kim S.,Sarjoughian H. S. et Elamvazhuthi V., 'DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring', Université d'arizona Tucson, pp.3.

[24] Kim S., Simulator for Service-based Software Systems: Design and Implementation with DEVSSuite. MS Thesis. Computer Science and Engineering. Arizona State University, Tempe, AZ, USA, 2008.

[25] Kim S.,Sarjoughian H. S. et Elamvazhuthi V., 'DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring', Université d'arizona Tucson, pp.4-5.

[26] G. A. Wainer, CD++: a toolkit to define discrete-event models. Software, Practice and Experience. Wiley, November 2002, pp. 1261–1306.

[27] G. Wainer et W. Chen, A framework for remote execution and visualization of cell-DEVS models, 2003, pp. 626–647.

[28] A. Ilachinski, CELLULAR AUTOMATA, A Discrete Universe. World Scientific Publishing, 2001, Co. ISBN 981-02-4623-4.

[29] F. Bergero et E. Kofman, PowerDEVS: a tool for hybrid system modeling and real-time. SIMULATION, Janvier 2011, pp. 113–132.

[30] Campbell S., Chancelier J. et Ramine Nikoukhah, Modeling and Simulation in Scilab/Scicos. Springer, 2006.

[31] P. Mantegazza, E. L. Dozio, et S. Papacharalambous, Rtai: Real time application interface. Linux J., page 10.

[32] L. Capocchi, J.-F. Santucci, B. Poggi, et C. Nicolai. DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems, WETICE, June 2011, pages 170–175.

[33] Y. Van Tendeloo et H. Vangheluwe, The Modular Architecture of the Python(P)DEVS Simulation Kernel Work In Progress paper. In DEVS 14: Proceedings of the Symposium on Theory of M&S, April 2014, pages 387–392. SCS International.

[34] J. B. Filippi, 'The JDEVS modeling and simulation environment', Université de Corse , 2003 , pages.2-3.

[35] Bernardi, F. Conception de bibliothèques hiérarchisées de modèles réutilisables selon une approche orientée objet. Thèse de Doctorat, Université de Corse, 2002.

[36] Gabay, J et Gabay, D. uml 2 analyse et conception. Dunod, Paris, 2008.

Webographie

[20] https://msdn.microsoft.com/en-us/library/ff649643.aspx, consulté le 16 Mai 2016.

[22] http://acims.asu.edu/software/devs-suite/, consulté le 16 Mai 2016.

[37] http://www.oracle.com/technetwork/java/javase/documentation/, consulté le 5 Juin 2016.