View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

Co-simulation à base d'outils multi-agents : un cas d'étude avec NetLogo

Thomas Paris, Laurent Ciarletta, Vincent Chevrier

▶ **To cite this version:**

Thomas Paris, Laurent Ciarletta, Vincent Chevrier. Co-simulation à base d'outils multi-agents : un cas d'étude avec NetLogo. JFSMA 2018 - 26èmes Journées Francophones sur les Systèmes Multi-Agents, Oct 2018, Métabief, France. hal-01902786

HAL Id: hal-01902786
https://hal.archives-ouvertes.fr/hal-01902786

Submitted on 23 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Co-simulation à base d'outils multi-agents : un cas d'étude avec NetLogo

Thomas Paris[*], Laurent Ciarletta[*] et Vincent Chevrier[*]

[*] Université de Lorraine, CNRS, LORIA, F-54000 Nancy, France
prénom.nom@loria.fr

Octobre 2018

Résumé

La simulation multi-agent a démontré son intérêt pour la simulation de systèmes complexes. Cet article aborde la question de savoir comment peut-on représenter un phénomène puis le simuler lorsque plusieurs simulateurs multi-agents sont nécessaires. Cela soulève un problème majeur : Les simulateurs multi-agents ne sont pas conçus (en général) pour être utilisés conjointement à d'autres simulateurs.

Cet article présente une première réflexion sur l'intégration rigoureuse de simulateurs multi-agents dans une plateforme de co-simulation. Nous appuyons notre réflexion sur l'exemple du simulateur NetLogo et de la plateforme de co-simulation MECSYCO.

Mots-clés : Système complexe, Système Multi-Agent, Co-simulation, MECSYCO , NetLogo

Abstract

Multi-agent approach has demonstrated its benefits for complex system modeling and simulation. This article focuses on how to represent and simulate a system as a set of several interacting simulators, with a focus on the case of multi-agent simulators. This raises a major challenge: multi-agent simulators are not conceived (in general) to be used with other simulators.

This article presents a preliminary study about the rigorous integration of multi-agent simulators into a co-simulation platform. The work is grounded on the NetLogo simulator and the co-simulation platform MECSYCO.

***Keywords**: Complex system, Multi-agent system, Co-simulation,* MECSYCO *, NetLogo*

1 Introduction

La Modélisation et Simulation (M&S) de systèmes complexes est l'un des enjeux actuels majeurs de recherche. L'une des difficultés est de pouvoir combiner plusieurs perspectives d'un même système [Seck and Honig, 2012] au sein d'une représentation cohérente (multi-modélisation). Cela implique la gestion de phénomènes à différents niveaux (micro et macro), à différentes échelles (temporelles comme spatiales), et selon différentes perspectives. La gestion de ces hétérogénéités rend nécessaire le développement de nouvelles approches et outils.

L'une des approches prometteuses pour faire face à ces défis est la co-simulation [Gomes et al., 2018]. Elle consiste à faire interagir plusieurs simulateurs au sein d'une même simulation en assurant la synchronisation et les échanges de données. Cela permet la réutilisation de simulateurs existants et donc des outils déjà éprouvés dans des domaines spécifiques. Cependant, il faut garder à l'esprit que l'intégration de simulateurs au sein d'une co-simulation implique de gérer leurs hétérogénéités aux niveaux logiciel (i.e. pouvoir les piloter pour les faire interagir) et formel (i.e. rendre compatibles les différentes dynamiques des simulateurs).

En parallèle, l'approche multi-agent est particulièrement adaptée à la représentation de systèmes composés d'un grand nombre d'entités hétérogènes en interaction (i.e. la définition des systèmes complexes [Ramat, 2006]). Elle permet aussi de représenter à la fois les niveaux individuel et collectif [Michel et al., 2009], c'est donc une approche de choix pour modéliser et simuler les systèmes complexes.

La question qui nous intéresse dans cet article est *comment peut-on modéliser et simuler un système complexe à partir de plusieurs modèles multi-agents*, chacun offrant une perspective complémentaire de ce système. Nous adopterons une approche par co-simulation pour y répondre. La question revient alors à comment faire

interagir différents simulateurs multi-agents pour qu'ils échangent de l'information et synchronisent leur exécution. Nous nous limitons dans cet article au couplage spatial entre systèmes multi-agents : un agent n'est présent que dans un simulateur à la fois, les agents présents dans différents simulateurs ne peuvent interagir directement. Les interactions sont limitées aux évènements transmis d'un simulateur à un autre. Notre objectif est d'intégrer rigoureusement des simulateurs multi-agents dans des co-simulations hybrides mêlant des simulateurs continus et discrets. Nous ne considèrerons ni les approches ad-hoc (sources d'erreurs), ni la réécriture des modèles dans un seul simulateur (source d'erreurs, de perte de temps, ...).

Le problème que nous nous sommes fixés peut être résolu en répondant à deux questions : i) comment gérer le temps et la synchronisation des simulateurs multi-agents avec le reste de la co-simulation ?; et ii) comment gérer l'échange d'information entre les différents simulateurs de la co-simulation ? Nous avons démontré qu'il est possible de répondre à ces questions au travers de l'intergiciel MECSYCO [Camus, 2015] en utilisant le formalisme DEVS comme base formelle pour l'intégration. Nous détaillons dans cet article comment nous avons répondu à ces questions et construit un wrapper DEVS dans le cas du simulateur multi-agent NetLogo [Wilensky, 1999].

La suite de l'article est organisée comme suit : La section 2 présente les travaux en lien avec cet article. Ensuite, la section 3 introduit les concepts utilisés dans notre approche. La section 4 présente les principes de notre proposition qui est ensuite détaillée section 5. La section 6 détaille la mise en œuvre autour de quelques exemples illustratifs. Enfin, notre approche est discutée section 7 et la section 8 conclut.

2 Travaux connexes

Plusieurs types de travaux ont abordé la simulation d'un système multi-agent comme l'intégration de différents sous-systèmes. Une première question est se savoir à quel niveau du processus de modélisation cette intégration peut se faire. Nous reprenons la division du processus de modélisation de [Galán et al., 2009] en quatre étapes (voir Figure 1). De ce point de vue, l'intégration de sous-systèmes peut se faire

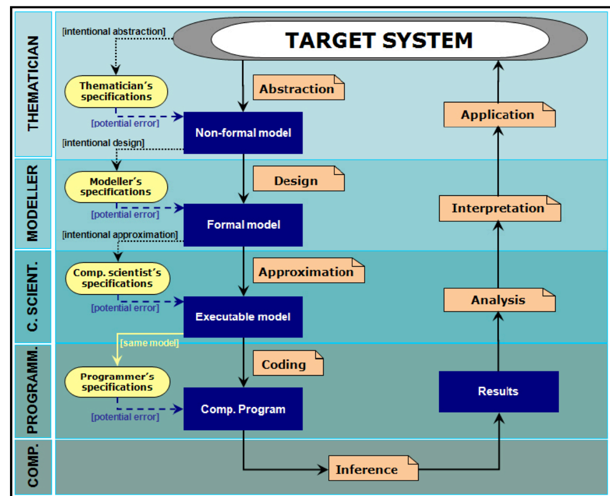

Figure 1: Les différentes étapes de l'activité de modélisation. Source [Galán et al., 2009]

à l'interface de ces 4 niveaux.

Dans le premier cas, une formulation conceptuelle décrit l'intégration en répondant aux besoins d'échanges d'informations entre composantes et de synchronisation des composantes, par exemple des patterns [Mathieu et al., 2016, Gangat, 2013], des modèles [Morvan et al., 2011, Maudet et al., 2013]. Ces travaux considèrent tous que l'intégration des différentes composantes du système multi-agent se fait au sein du même cadre conceptuel et du même outil.

Dans le deuxième cas, l'intégration rigoureuse se fait grâce à un formalisme pivot. C'est le cas pour VLE (Virtual Laboratory Environment) [Quesnel et al., 2009], duquel nous reprenons les idées concernant le wrapping.

Le troisième cas considère l'intégration comme un problème d'interopérabilité logicielle, les problèmes conceptuels et formels sont traités le plus souvent de manière ad-hoc.

D'autres simulateurs multi-agents sont capables de simuler un système sous la forme de plusieurs environnements logiques ou physiques (comme Madkit[1], ou Repast [North et al., 2013]), mais ne permettent pas d'inclure des environnements provenant d'autres outils. Un effort pour une intégration plus générique a été réalisé dans [Behrens et al., 2011] en considérant une interface pour les interactions entre agent et environnement. À notre connaissance, les problèmes d'intégration des simulateurs multi-agents au sein d'une co-simulation et de la gestion rigoureuse de la synchronisation ne sont pas

[1]http://www.madkit.net/madkit/

traités. Plus globalement, les travaux existants (excepté VLE) proposent un ensemble de concepts agents pour définir l'intégration de composant dans une simulation mais leur formalisation n'est pas ouverte pour s'intégrer à un cadre plus vaste.

3 Pré-requis

3.1 DEVS

DEVS (Discrete EVent System specification) est un formalisme évènementiel proposé par Bernard P. Zeigler dans les années 70 [Zeigler et al., 2000]. Son intérêt dans notre travail est sa capacité à intégrer les autres formalismes. En effet, grâce à sa propriété d'universalité, DEVS se place comme un formalisme pivot pour l'intégration de nouveaux formalismes [Vangheluwe, 2000].

Comme résumé dans [Quesnel et al., 2005], l'intégration au sein du formalisme DEVS peut se faire selon deux stratégies, le mapping ou le wrapping. La première consiste à rendre complètement équivalent un modèle issue du formalisme cible à un modèle atomique (ou couplé) DEVS. La seconde consiste à créer une interface autour du formalisme cible de sorte à pouvoir le manipuler comme un modèle atomique DEVS. L'avantage du wrapping est qu'il permet de réutiliser des modèles existants déjà implémentés dans des outils de simulation. C'est le choix que nous faisons dans cet article.

3.1.1 Description des modèles

Le formalisme DEVS fait une distinction entre un modèle atomique (dit comportemental) et un modèle couplé (dit structurel). Un modèle atomique DEVS M_i est décrit par l'ensemble :

$$M_i = (X_i, Y_i, S, \delta_{ext}, \delta_{in}, \lambda, ta)$$

Où :

- $X_i = \{(p, v)|p \in IPorts_i, v \in V_{X_i}\}$ est l'ensemble des ports d'entrée ($IPorts_i$) et de leurs valeurs admissibles (V_{X_i}).
- $Y_i = \{(p, v)|p \in OPorts_i, v \in V_{Y_i}\}$ est l'ensemble des ports de sortie ($OPorts_i$) et de leurs valeurs admissibles (V_{Y_i}).
- S est l'ensemble des états du modèle.
- $\delta_{ext} : Q \times X_i \rightarrow S$ est la fonction de transition externe qui décrit comment le modèle évolue à la réception d'un évènement externe. $Q = (s, e)|s \in S, 0 \leq e \leq ta(s)$ correspond à un état total du modèle (un état de l'ensemble S associé au temps e passé dans cet état).
- $\delta_{in} : S \rightarrow S$ est la fonction de transition interne qui décrit la dynamique interne du modèle (i.e. comment le système change d'état lorsque le temps e passé dans l'état courant atteint la limite).
- $\lambda : S \rightarrow Y i$ est la fonction de sortie qui décrit l'évènement sortant du modèle selon son état courant. Elle permet de récupérer les évènements de sortie du modèle.
- $ta : S \rightarrow \mathbb{R}^+_{0,\infty}$ est une fonction qui donne le temps du prochain évènement interne du modèle. Son rôle est crucial dans la synchronisation des modèles DEVS lors de la simulation.

Le modèle couplé décrit la structure du système (i.e. comment les modèles atomiques sont connectés ensemble pour représenter le système). La structure du modèle couplé n'est pas utilisée directement pour ce travail et n'est donc pas détaillée ici.

3.1.2 Wrapping et simulation

Le formalisme DEVS propose un algorithme de simulation abstrait. Il impose cinq méthodes (détaillées plus loin) pour la simulation d'un modèle atomique. La création d'un wrapper DEVS autour d'un simulateur multi-agent correspond à la création d'une interface implémentant ces cinq fonctions pour manipuler le simulateur. Une fois le wrapper réalisé, le simulateur peut être utilisé comme un modèle atomique et s'intégrer rigoureusement dans une co-simulation DEVS.

3.2 NetLogo

NetLogo [Wilensky, 1999] est un environnement de modélisation de systèmes multi-agents. Il est multi-plateforme et fournit une API permettant de piloter son simulateur ce qui simplifie son intégration. Un modèle NetLogo est composé d'une interface graphique qui permet à l'utilisateur d'interagir avec la simulation et de voir son évolution, et d'un script (en langage NetLogo) qui

décrit le comportement des agents (appelés *turtles*), la dynamique de l'environnement et, plus généralement, quelles actions seront réalisées à chaque pas de simulation.

Par convention, les paramètres de la simulation sont initialisés par la méthode *setup* et la simulation est déroulée par appels successifs à la méthode *go* qui fait évoluer la simulation pas à pas. L'interface graphique contient des boutons associés à ces commandes et des curseurs pour sélectionner les valeurs des paramètres.

3.3 MECSYCO

L'intergiciel MECSYCO[2] [Camus et al., 2015, Camus, 2015] est une plateforme de wrapping DEVS qui s'appuie sur l'universalité de DEVS pour la modélisation multi-paradigme et la co-simulation de systèmes complexes. MECSYCO a été utilisé pour la M&S des réseaux électriques intelligents (smart grids) dans le contexte d'un partenariat entre le LORIA/Inria et EDF R&D [Vaubourg et al., 2015].

MECSYCO est basé sur le paradigme AA4MM (*Agent & Artifacts for Multi-Modeling*) [Siebert et al., 2010] (lui-même issu d'une idée originale de [Bonneaud, 2008]) qui considère une co-simulation hétérogène comme un système multi-agent. Dans cette perspective, chaque couple modèle/simulateur correspond à un agent et les échanges de données entre les simulateurs correspondent aux interactions entre les agents[3]. L'originalité par rapport à d'autres approches de multi-modélisation multi-agent, est d'envisager les interactions de manière indirecte grâce à des entités passives de calcul appelées artéfacts [Ricci et al., 2007]. En suivant ce paradigme multi-agent des concepts jusqu'à leurs implémentations, MECSYCO garantit une architecture de co-simulation extensible (i.e. des fonctionnalités peuvent être facilement ajoutées, comme un système d'observation) décentralisée et distribuable. MECSYCO implémente les concepts de AA4MM en suivant le protocole de simulation DEVS pour coordonner les exécutions des simulateurs et générer les interactions entre les modèles.

[2] MECSYCO est disponible sur `www.mecsyco.com` sous licence AGPL.

[3] Il faut noter que la notion de système multi-agent apparait sur deux niveaux dans cet article : comme une architecture d'intergiciel de co-simulation, et comme des modèles de simulation à intégrer dans une co-simulation.

Nous avons développé dans MECSYCO des wrappers pour des outils de modélisation de systèmes discrets ou continus comme par exemple les simulateurs réseaux NS-3 et OMNeT++ [Vaubourg et al., 2016], le standard FMI [Blochwitz et al., 2012], et des wrappers NetLogo dédiés à des exemples applicatifs particuliers [Camus et al., 2015].

4 Proposition

Jusqu'à présent, l'intégration de NetLogo dans MECSYCO nécessitait de spécifier un nouveau wrapper pour chaque modèle. Cette limite était due à l'absence de représentation déclarative des concepts DEVS; i.e. la déclaration des ports d'entrée, des ports de sortie et des paramètres (éléments spécifiques à chaque modèle) était faite directement dans le wrapper, le rendant spécifique à un modèle.

La création d'un wrapper DEVS pour MECSYCO implique de créer une interface entre le simulateur et les cinq fonctions du protocole de simulation DEVS :

- *init* initialise le modèle (paramètres et état initial) avant la simulation.
- *processExternalEvent* demande au simulateur d'exécuter les évènements externes entrants. Elle est dépendante de l'ensemble X_i des ports d'entrée et des évènements autorisés sur ces ports.
- *processInternalEvent* demande au simulateur d'exécuter les évènements internes à un temps donné (fait progresser la simulation suivant le temps).
- *getOutputEvent* retourne les évènements de sortie du modèle. Elle est fortement liée à l'ensemble Y_i des ports de sortie et des évènement admissibles sur ces ports.
- *getNextInternalEventTime* retourne le temps du prochain évènement interne. La politique d'exécution du simulateur doit avoir une signification temporelle pour déterminer cette valeur.

C'est au travers de ces cinq fonctions que seront traitées les questions de gestion du temps, de la synchronisation et de l'échange d'information entre les simulateurs. Pour avoir un wrapper

générique, ces fonctions doivent être indépendantes des modèles simulés. Cela signifie que chaque modèle m_i doit spécifier ses ensembles de ports d'entrée et de ports de sortie (respectivement X_i et Y_i). Ces éléments sont rarement présents dans les modèles multi-agents (qui ne sont généralement pas conçus pour être connectés à d'autres modèles et dont l'architecture n'est pas pensée avec des ports). Pour résoudre ce problème, nous proposons de définir explicitement ces ensembles et les évènements correspondants dans un document d'interface associé au modèle (de la même manière qu'un document XML de description est associé au modèle dans le standard FMI). Nous devons également exprimer ce qui doit être fait lorsque un évènement d'entrée est reçu, comment récupérer les données correspondant à des évènements de sortie et comment initialiser les paramètres.

Comme les simulateurs multi-agents utilisent des méta-modèles et des stratégies d'implémentation variés, nous ne visons pas une méthode générique d'intégration de ces simulateurs. Nous souhaitons plutôt proposer un wrapper générique spécifique à NetLogo en utilisant l'interpréteur de commande disponible dans son API.

Il faut souligner que la méthode *processInternalEvent* fait évoluer le temps de la simulation et ne peut pas être décomposée en sous-commandes. Cela a pour conséquence qu'il est impossible de faire interagir des agents de simulateurs différents au cours du même pas de simulation (ou sinon nous ne respectons plus le protocole DEVS).

5 Le wrapper DEVS pour NetLogo

Cette section présente l'encapsulation de NetLogo par l'intermédiaire d'un wrapper et de documents d'interface. La Figure 2 résume les principes de l'approche.

5.1 Documentation d'interface

Jusqu'à présent, les wrappers NetLogo de MECSYCO contenaient directement dans leur code Java les informations relatives aux port d'entrée et de sortie, tout comme les déclarations des commandes à exécuter pour chacune des cinq méthodes DEVS. Nous proposons de fournir ces informations séparément dans un document

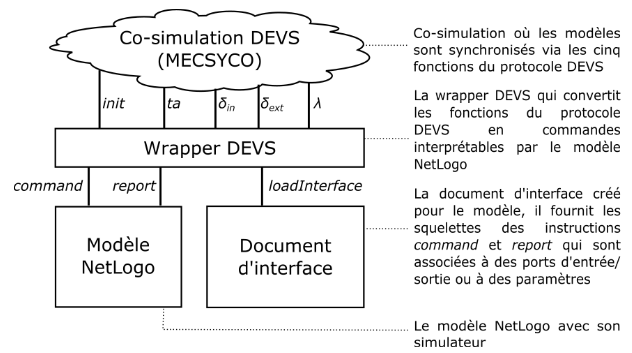

Figure 2: Principe de fonctionnement du wrapper NetLogo.

d'interface qui contiendra les noms des ports d'entrée et des ports de sortie, ainsi que la déclaration de ce qu'il faut exécuter pour chaque méthode du protocole DEVS.

init() : Nous supposons que la méthode *setup* peut être utilisée comme fonction d'initialisation[4]. Nous ajoutons le concept de paramètre dans la documentation pour pouvoir changer les valeurs par défaut utilisées dans la fonction *setup* (valeurs généralement choisies via l'interface graphique).

getNextInternalEventTime() : Nous supposons un protocole de simulation à pas de temps constant dans lequel chaque *tick* n'a pas de signification temporelle. C'est alors au modélisateur de donner au *tick* une signification temporelle pour le reste de la co-simulation (e.g. un *tick* représente 0.1 unité de temps de la simulation).

processInternalEvent() : Nous supposons[5] que la méthode *go* contient les commandes à exécuter à chaque *tick* et qu'elle peut donc correspondre à la méthode *processInternalEvent()*.

processExternalEvent(): Le concept de ports d'entrée n'existe pas dans NetLogo, il faut donc les définir pour chaque modèle. Il faut aussi préciser comment prendre en compte les évènements d'entrée sur chacun des ports.

getOutputEvent() : Là encore, le concept de ports de sortie n'existe pas dans NetLogo, ils doivent donc être définis pour chaque modèle. Il faut là encore définir comment récupérer les évènements de sortie sur chaque port.

[4]Dans les autres cas, soit la documentation fournit les commandes à appeler, soit elle fournit le code à exécuter.

[5]Dans d'autres cas, cela doit être spécifié comme pour la méthode *setup*.

5.2 Gestion du temps de simulation

Comme dans [Quesnel et al., 2005], nous choisissons de laisser le modélisateur choisir une durée fixe t correspondant à un *tick* NetLogo. Ainsi, la fonction ta renverra toujours $currentTime + t$ peut importe l'état du modèle. C'est une solution simple. Comme NetLogo permet au modélisateur de définir lui-même la progression du temps de simulation (via la méthode *tick-advance*), des approches plus complexes peuvent être envisagées. Le wrapper peut facilement être étendu en ce sens, notamment en faisant un appel à la méthode *ticks* et en retournant la valeur appropriée.

5.3 Gestion des ports d'entrée, des ports de sortie et des paramètres

Comme vu précédemment, l'architecture de NetLogo n'a pas le concept de ports associés à des évènements. Elle propose cependant une API permettant d'interagir avec le modèle via un interpréteur de commandes. Il suffit donc de fournir une chaine de caractères correspondant soit à une *command* NetLogo à exécuter pour modifier le modèle, soit à un *report* NetLogo pour récupérer des données du modèle.

Nous proposons de définir dans la documentation des modèles les noms des ports d'entrée et de sortie associés aux méthodes NetLogo à exécuter.

Dans le cas des évènements d'entrée, nous distinguons 3 cas :
1) les ports acceptent des évènements qui ne dépendent pas de données issues d'autres simulateurs : le modèle doit exécuter des commandes (sans paramètre) définies par le modélisateur;
2) les ports acceptent des évènements qui contiennent des données simples issues d'un autre simulateur : ces données doivent être intégrées dans la simulation grâce à des commandes NetLogo (avec paramètres) qui modifient des variables d'environnement ou des attributs de *turtles*;
3) les ports acceptent des évènements qui contiennent une liste de données : le nombre de commandes doit être adapté pour traiter ces données.

Dans le cas des évènements de sortie, deux cas peuvent être envisagés :

1) l'évènement contient une ou plusieurs données (la valeur d'un attribut, ...). Dans ce cas, un ou plusieurs *reports* seront utilisés pour accéder aux valeurs. Ce type d'évènement n'a pas d'impact sur le modèle NetLogo.

2) Dans le second cas l'évènement a un impact sur le modèle (par exemple des *turtles* qui sortent du modèle). Une ou plusieurs commandes devront être utilisées pour définir cet impact (e.g. suppression des *turtles* du modèle).

La fonction δ_{ext} de l'interface DEVS convertira les évènements d'entrée en méthodes *command* NetLogo, tandis que la fonction λ utilisera des méthodes *report* pour collecter des données et les transcrire en évènements (des méthodes *command* peuvent être utilisées pour maintenir la cohérence du modèle, e.g. la suppression de *turtles*).

Les paramètres spécifiés dans le documents d'interface sont associés à une unique méthode *command*, ce qui permet de modifier un paramètre issu de l'interface graphique avant l'appel de la fonction *setup*.

5.4 Le wrapper

Le principe de base de création du wrapper est d'associer chaque méthode du protocole DEVS à du code NetLogo. Cela est réalisé directement en Java en utilisant l'API NetLogo.

Comme le code à exécuter est spécifié dans le document d'interface, le wrapper devient générique et peut être utilisé pour chaque modèle NetLogo.

Pour résumer, comme NetLogo propose deux méthodes liées à l'initialisation (*setup*) et à la simulation d'un pas (*go*), nous les réutilisons. Quand un pas de simulation est exécuté, le temps de simulation avance d'une durée fixe. Comme la simulation d'un pas est atomique (il ne peut pas être décomposé et nous n'avons pas d'équivalent à la fonction donnant le temps écoulé depuis le dernier évènement), le traitement d'un évènement d'entrée est effectué au pas de simulation suivant en modifiant l'état du modèle, le prochain appel de *go* calculera alors la réaction du modèle.

6 Preuve de concepts

6.1 Objectif de l'expérience

Le but des expériences ci-dessous et de montrer la faisabilité et les possibilités offertes par l'approche proposée, notamment concernant l'intégration de modèles multi-agents NetLogo comme composant modulaire d'une co-simulation avec la plateforme MECSYCO.

Nous proposons deux expériences. La première

```
model WolfSheepPredationExample version "1.0"
path "My Models/NetLogo/Wolf Sheep Predation.nlogo"
interface
    parameters // model parameters
        grass : true command "set grass? %s" // always true
        grass_regrowth_time : 10 command "set grass-regrowth-time %s" //small value in order to see effect of e
        // the following just put the same values as the GUI. Only for illustration purposes.
        initial_number_sheep : 100 command "set initial-number-sheep %s"
        initial_number_wolves : 50 command "set initial-number-wolves %s"

    inputs
        //Inputs with the commands to process accordingly
        grass_regrowth : Integer initOption parameter command "set grass-regrowth-time %s"
        wolf_hunt : Integer initOption no command "ask n-of %s wolves [die]"
        sheep_coming : Integer initOption no command "create-sheep %s [set color white set size 1.5  set label-
    outputs
        //outputs (NB we use double instead of int because of some trouble with netlogo representation ... :-(
        nb_sheep : Double initOption no report Double "count turtles with [breed = sheep]"
        nb_wolves : Double initOption no report Double "count turtles with [breed = wolves]"
        nb_grass : Double initOption no report Double "grass"

information
    keywords
        "Predation"
        "NetLogo"
    description
        "This model represents wolf-sheep predation system example of NetLogo, it is an example of the integrat
simulator
    simulation variables
        //simulation variable
        stopTime: 1000. variability parameter // Time of the end of simulation
        discretization: 1. variability parameter //correspondence between Netlogo tick and the simulation time
```

Figure 3: Résumé de la description DSL de l'exemple 1.

montre l'utilisation d'évènements entrants qui modifient les variables d'environnement du modèle ou qui déclenchent des effets sur les agents (à l'image de ce qui pourrait être fait avec l'interface graphique). La seconde montre un couplage spatial entre plusieurs modèles NetLogo, avec des évènements qui permettent de transférer des agents entre modèles.

Nous avons deux remarques à faire avant de décrire notre preuve de concept. La première est que nous ne rappelons pas ce qu'est une co-simulation MECSYCO, nous fournissons juste une description intuitive au travers des exemples. Un lecteur intéressé pourra trouver plus de détails sur le site de MECSYCO où plusieurs tutoriels expliquent les concepts sous-jacents. La deuxième remarque est que nous utilisons un DSL (voir Figure 3) pour décrire les documents d'interface. Le détail des possibilité offertes par le DSL dépasse le cadre de l'article. Nous ne fournissons donc que les éléments clés nécessaires à la compréhension.

6.2 Variation sur le modèle Proie-Prédateur

Le modèle *Wolf-Sheep-Predation*[6] décrit comment des populations de loups et de moutons interagissent et évoluent au cours du temps comme dans un écosystème Proie-Prédateur. De nombreux paramètres peuvent être changés pour observer leurs impacts sur la dynamique des populations. Nous ne modifions pas la dynamique du modèle originel, nous l'étendons juste pour illustrer la possibilité de modifier les paramètres et de définir des ports d'entrée qui modifient des propriétés du modèle au cours de la simulation. Concrètement nous voulons :

- choisir les valeurs initiales des paramètres
- fournir des ports d'entrée qui modifient des propriétés de l'environnement
- collecter périodiquement des informations sur les populations pour faire des graphiques (extérieurs à NetLogo).

Ces éléments sont détaillés ci-dessous. Ils sont résumés dans la table 1 et suivis par leur définition dans le DSL que nous avons réalisé pour MECSYCO. Notons que les paramètres des commandes de NetLogo sont identifiés par "%s".

Nous utilisons les paramètres suivants : `grass`, la dynamique de l'herbe est active (contrairement à la valeur par défaut dans l'interface graphique); `grass_regrowth_time`, le temps nécessaire pour la repousse de l'herbe en nombre de *tick*); `initial_number_sheep`, nombre initial de moutons; et, `initial_number_wolves`, nombre initial de loups.

[6] Fourni dans la bibliothèque de modèles de NetLogo.

Nous créons les ports d'entrée suivants (chaque port contient le code à appliquer pour les modifications) : `grass_regrowth` modifie le temps de repousse de l'herbe (nous avons défini un nom de port différent du nom de la variable d'environnement); `sheep_coming`, augmente le nombre de moutons; et `wolf_hunt` diminue le nombre de loups. Chaque port correspond à un type d'évènement pouvant provenir de différents modèles (chacun envoyant un seul évènement à un temps donné).

Table 1: Résumé du document d'interface.

PARAMETERS		
Name	**Command**	**Value**
grass	"set grass? %s"	true
grass_regrowth_time	"set grass-regrowth-time %s"	10
initial_number_sheep	"set initial-number-sheep %s"	100
initial_number_wolves	"set initial-number-wolves %s"	50
INPUT PORTS		
Name	**Command**	
grass_regrowth	"set grass-regrowth-time %s"	
sheep_coming	"create-sheep %s [set color white set size 1.5 set label-color blue - 2 set energy random (2 * sheep-gain-from-food) setxy min-pxcor max-pycor]"	
wolf_hunt	"ask n-of %s wolves [die]"	
OUTPUT PORTS		
Name	**Report statement**	
nb_sheep	"count turtles with [breed = sheep]"	
nb_wolves	"count turtles with [breed = wolves]"	
nb_grass	"grass"	

Nous définissons les ports de sortie suivants qui seront connectés à un système externe pour afficher les graphiques : `nb_sheep`, nombre actuel de moutons; `nb_wolves`, nombre actuel de loups; et `nb_grass`, quantité d'herbe.

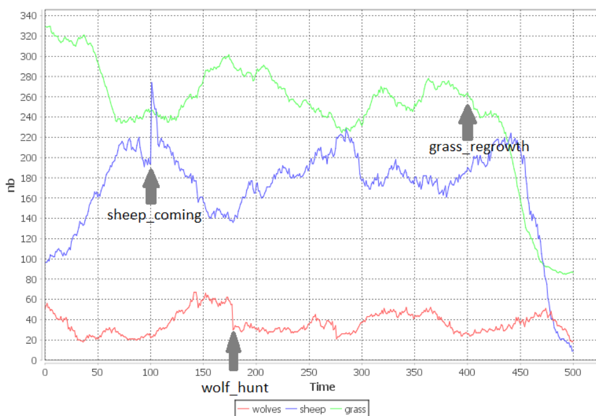

Figure 4: Impact des différents évènements sur le modèle *Wolf-Sheep-Predation*.

La Figure 4 montre les résultats obtenus avec cette configuration. Nous pouvons observer l'impact des différents évènements (arrivée de 100 moutons à t=100; mort de 25 loups à t=175, augmentation du temps de repousse de l'herbe à t=400).

Cette preuve de concept montre que nous sommes capables de i) fournir des valeurs initiales (e.g. le paramètre *grass* et actif) pour modifier des caractéristiques du modèle (éliminer des loups, ajouter des moutons ou modifier des propriétés de l'environnement) grâce à des évènements d'entrée (provenant d'autres modèles) et ii) de récupérer des informations (ici le nombre de loups, de moutons et la quantité d'herbe) grâce à des évènements de sortie qui pourront être utilisés par d'autres modèles (ici un système d'affichage).

6.3 Couplage spatial

Dans cette expérience, nous montrons la possibilité de transférer des *turtles* d'un modèle NetLogo à un autre. Nous connectons ensemble les trois modèles suivants : un modèle proie-prédateur qui "envoie" des moutons à un modèle *Pedestrian* (un modèle dans lequel les *turtles* avancent de la gauche vers la droite comme le feraient des piétons dans un couloir) qui, quand les *turtles* arrivent à son extrémité, les envoie dans un autre modèle proie-prédateur.

Le premier modèle est le même que pour l'expérience précédente, nous ajoutons un nouveau port de sortie `sheep_escaping` qui est associé aux moutons présents sur le bord droit du modèle. Les données sont collectées comme une liste des attributs des moutons (leur ordonnée et leur énergie).

Le deuxième modèle est initialement vide et les moutons, provenant du port `sheep_escaping`, arrivent par le port d'entrée appelé `left_in`. Les moutons avancent de la gauche vers la droite. Un port de sortie (`right_out`) est associé aux moutons présents sur le bord droit.

Le troisième modèle est encore un modèle proie-prédateur sur lequel nous ajoutons un port d'entrée (`sheep_loop_arrival`) qui accepte une liste de moutons pour les créer sur le bord gauche de l'environnement avec les attributs qui sont spécifiés dans l'évènement qui provient d'un port `sheep_escaping`. Le modèle est initialisé avec des populations de moutons et de loups nulles. La Figure 5 illustre les connexions entre les modèles et donne un aperçu des trois fenêtres NetLogo durant la co-simulation.

Ces connexions sont possibles parce que les types des évènements sont compatibles entre les ports que nous connectons, de plus le modèle *Pedestrian* définit un attribut "*energy*" pour pouvoir

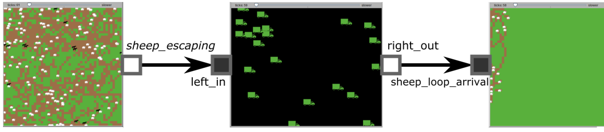

Figure 5: Connexions entre les modèles pour le couplage spatial.

assurer la cohérence.

Un point intéressant à noter est que le modèle *Pedestrian* que nous avons utilisé ici permet de connecter plusieurs instances, l'une suivant l'autre, en réutilisant le même document d'interface. Le modèle devient un composant modulaire pour la plateforme de co-simulation.

Un autre point intéressant à noter est que nous n'avons utilisé qu'un seul document d'interface pour le modèle proie-prédateur, celui a été enrichi au fur et à mesure des cas d'utilisation. En pratique le document d'interface n'a pas besoin d'être retouché sauf si un nouveau cadre d'utilisation impliquant de nouveaux ports est trouvé pour le modèle.

7 Discussion

La possibilité d'intégrer dans une co-simulation un modèle NetLogo, comme un composant qui peut être facilement ajouté/retiré ou échangé, permet de définir des modèles de systèmes comme le couplage de différents sous-modèles.

Il peut être intéressant de comparer notre approche avec LevelSpace [Hjorth et al., 2015], une extension de NetLogo permettant de connecter plusieurs modèles NetLogo. Les deux approches ont des points communs (e.g. utilisation de *command* et de *report* pour gérer l'interaction entre les modèles), mais les objectifs sont différents. LevelSpace reste dans le cadre de NetLogo, profitant d'un contrôle plus important sur les modèles et n'a pas pour objectif le couplage avec des modèles issus d'autres outils. Au contraire, notre approche vise le couplage avec d'autres simulateurs en utilisant un intergiciel, une intégration dans DEVS ainsi qu'une démarche modulaire basée sur des ports.

La question qui n'est pas abordée dans notre approche est à quel point un modèle NetLogo existant doit être adapté pour pouvoir être intégré à une co-simulation DEVS. À l'heure actuelle, la diversité des modèles disponibles dans NetLogo nous empêche de fournir une réponse générique à cette question. Cependant plusieurs pistes peuvent être envisagées. Concernant la modification du modèle, l'interface graphique de NetLogo autorise l'utilisateur à modifier les paramètres et à exécuter des actions entre chaque pas d'exécution. Ceci est compatible avec notre approche : un évènement d'entrée peut faire la même chose. Bien que nous ayons restreint notre approche au couplage spatial, nous autorisons les agents à entrer et sortir des modèles. La sortie des agents peut être gérée en fournissant les propriétés que ces agents doivent vérifier et i) en les exportant sous forme de liste composée de leurs attributs et ii) en les retirant de leur modèle d'origine. Les agents entrants peuvent être représentés par un évènement d'entrée qui contient la liste des attributs nécessaires pour créer les *turtles* correspondantes.

Nous avons simplifié l'intégration de NetLogo en utilisant les commandes *setup* et *go* plutôt que des commandes définies par l'utilisateur. Ces choix peuvent être changés et ne remettent pas en cause les principes utilisés. Le DSL permet alors au modélisateur de se focaliser sur les entrées/sorties du modèle NetLogo pour l'intégrer à une co-simulation MECSYCO sans avoir à programmer dans un autre langage (i.e. Java dans notre cas).

Comme un système peut être composé de plusieurs sous-systèmes, la question des performances et du passage à l'échelle des simulations peut être posée. Bien que nous ne nous soyons pas focalisés sur les performances lors de la conception de MECSYCO, son architecture permet de distribuer l'exécution des simulateurs sur plusieurs machines. Cela permet un premier niveau de passage à l'échelle en limitant l'impact du nombre de simulateurs sur le temps d'exécution.

Cet article se concentre sur l'intégration de NetLogo dans DEVS. Nous utilisons une approche déclarative pour combler le fossé entre le protocole de simulation DEVS et les primitives du modèle/simulateur. Les lecteurs peuvent s'interroger sur la généralisation de cette approche à d'autres simulateurs multi-agents. D'après notre expérience, il est difficile de proposer une approche systématique à cause de la diversité d'architecture des plateformes multi-agents, d'autant qu'il n'y a encore aucun standard sur lequel se reposer au niveau formel. Au niveau logiciel, l'intégration est aisée si une API est fournie et si le simulateur est ouvert (i.e. peut

être manipulé).

8 Conclusion et perspectives

Cet article présente un travail préliminaire sur l'intégration d'un simulateur multi-agent dans une co-simulation. Notre proposition (implémentée sur la plateforme NetLogo) est basée sur un wrapper et une documentation qui précise i) les paramètres initiaux, les ports d'entrée et de sortie; et ii) les codes NetLogo correspondants à l'implémentation des fonctions du protocole DEVS dans le modèle. Cette documentation peut être écrite grâce à un DSL dédié et être utilisée directement au sein du wrapper NetLogo générique (une classe Java dans notre cas) de la plateforme MECSYCO.

Nous fournissons deux preuves de concept qui montre la possibilité d'intégrer des modèles NetLogo dans MECSYCO, et de les faire interagir entre eux ou avec d'autres simulateurs déjà intégrés sans effort de programmation supplémentaire (excepté le code NetLogo à renseigner dans la documentation).

Cette intégration ouvre la possibilité de réutiliser dans une co-simulation une grande variété de modèles NetLogo existants (avec ou sans autres simulateurs multi-agents).

Cependant, notre proposition impose que le modèle soit adapté pour être utilisé au sein d'une co-simulation. Une deuxième limite provient de la stratégie d'intégration qui impose que chaque échange d'information fasse avancer le temps de la simulation; cela interdit les interactions entre agents situés dans des simulateurs différents.

En perspective de ce travail, nous souhaitons confronter notre approche à d'autres modèles NetLogo pour étudier plus précisément dans quelles mesures un modèle doit être adapté pour être intégré à une co-simulation, et pour valider conceptuellement notre approche avant de la confronter à d'autres simulateurs multi-agents (avec lesquels nous ferons face aux mêmes problèmes conceptuels ainsi qu'à de nouveaux problèmes d'intégration logicielle).

Références

[Behrens et al., 2011] Behrens, T. M., Hindriks, K. V., and Dix, J. (2011). Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61(4) :261–295.

[Blochwitz et al., 2012] Blochwitz, T., Otter, M., Åkesson, J., et al. (2012). Functional mockup interface 2.0 : The standard for tool independent exchange of simulation models. In *Proc. 9th International Modelica Conference*, pages 173–184.

[Bonneaud, 2008] Bonneaud, S. (2008). *Des agents-modèles pour la modélisation et la simulation de systèmes complexes - Application à l'écosystémique des pêches.* PhD thesis.

[Camus, 2015] Camus, B. (2015). *Environnement Multi-agent pour la Multi-modélisation et Simulation des Systèmes Complexes.* PhD thesis, Université de Lorraine.

[Camus et al., 2015] Camus, B., Bourjot, C., and Chevrier, V. (2015). Combining DEVS with multi-agent concepts to design and simulate multi-models of complex systems (WIP). In *Proc. of TMS/DEVS 15*, pages 85–90. SCS.

[Galán et al., 2009] Galán, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., del Olmo, R., López-Paredes, A., and Edmonds, B. (2009). Errors and artefacts in agent-based modelling. *Journal of Artificial Societies and Social Simulation*, 12(1) :1.

[Gangat, 2013] Gangat, Y. (2013). *Architecture Agent pour la modélisation et simulation de systèlmes complexes multidynamiques : une approche multi-comportementale basée sur le pattern "Agent MVC".* PhD thesis, Université de La Rélunion.

[Gomes et al., 2018] Gomes, C., Thule, C., Broman, D., Gorm Larsen, P., and Vangheluwe, H. (2018). Co-simulation : a survey.

[Hjorth et al., 2015] Hjorth, A., Head, B., and Wilensky, U. (2015). LevelSpace NetLogo extension.

[Mathieu et al., 2016] Mathieu, P., Morvan, G., and Picault, S. (2016). Simulations multi-agents multi-niveaux : quatre patterns de conception. In et Julien Saunier, F. M., editor, *JFSMA'16)*, Systèmes multi-agents et simulation, pages 117–126. Cépaduès.

[Maudet et al., 2013] Maudet, A., Touya, G., Duchêne, C., and Picault, S. (2013). Improving multi-level interactions modelling in a multi-agent generalisation model : first thoughts. In *Proceedings of 16th ICA Workshop on Generalisation and Multiple Representation, Dresden, Germany.*

[Michel et al., 2009] Michel, F., Ferber, J., Drogoul, A., et al. (2009). Multi-agent systems and simulation : a survey from the agents community's perspective. In Uhrmacher, A. and Weyns, D., editors, *Multi-Agent Systems : Simulation and Applications, Computational Analysis, Synthesis, and Design of Dynamic Systems*, pages 3–52. CRC Press - Taylor and Francis.

[Morvan et al., 2011] Morvan, G., Veremme, A., and Dupont, D. (2011). IRM4MLS : the influence reaction model for multi-level simulation. In *Multi-Agent-Based Simulation XI*, volume 6532 of *LNCS*, pages 16–27. Springer.

[North et al., 2013] North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., and Sydelko, P. (2013). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1) :3.

[Quesnel et al., 2005] Quesnel, G., Duboz, R., and Ramat, E. (2005). Wrapping into DEVS Simulator : A Study Case. *International Mediterranean Modeling Multiconference*, pages pp. 374–382.

[Quesnel et al., 2009] Quesnel, G., Duboz, R., and Ramat, E. (2009). The virtual laboratory environment - an operational framework for multi-modelling, simulation and analysis of complex systems. *Simulation Modelling Practice and Theory*, (17) :641–653.

[Ramat, 2006] Ramat, E. (2006). Introduction à la simulation : principaux concepts. In *Modélisation et Simulation Multi-Agent : application pour les Sciences de l'Homme et de la Société*, pages 37–60.

[Ricci et al., 2007] Ricci, A., Viroli, M., and Omicini, A. (2007). Give agents their artifacts : the A&A approach for engineering working environments in MAS. In *AAMAS '07*. ACM.

[Seck and Honig, 2012] Seck, M. D. and Honig, H. J. (2012). Multi-perspective modelling of complex phenomena. *Comput. Math. Organ. Theory*, 18(1) :128–144.

[Siebert et al., 2010] Siebert, J., Ciarletta, L., and Chevrier, V. (2010). Agents and artefacts for multiple models co-evolution : building complex system simulation as a set of interacting models. In *Proc. of AAMAS '10*. AAMAS/ACM.

[Vangheluwe, 2000] Vangheluwe, H. L. (2000). DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on*, pages 129–134. IEEE.

[Vaubourg et al., 2016] Vaubourg, J., Chevrier, V., Ciarletta, L., and Camus, B. (2016). Co-simulation of ip network models in the cyber-physical systems context, using a devs-based platform. In SCS/ACM, editor, *Communications and Networking Simulation Symposium (CNS'16)*.

[Vaubourg et al., 2015] Vaubourg, J., Presse, Y., Camus, B., et al. (2015). Multi-agent multi-model simulation of smart grids in the MS4SG project. In *Proc. PAAMS 15*, pages 240–251. Springer.

[Wilensky, 1999] Wilensky, U. (1999). Netlogo (and netlogo user manual). *Center for connected learning and computer-based modeling, Northwestern University. http ://ccl.northwestern.edu/netlogo.*

[Zeigler et al., 2000] Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of modeling and simulation : integration discrete event and continuous complex dynamic systems.* Academic press.