ALASCA — Architecture logicielles avancées pour les systèmes cyber-physiques autonomiques

© Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie

Sorbonne Université
Jacques.Malenfant@lip6.fr

Cours 5

Simulation modulaire et DEVS

Objectifs pédagogiques du cours 5

- Comprendre les principales problématiques d'implantation des simulateurs.
- Comprendre les principales problématiques du passage à la **simulation modulaire et temps réel** voire **répartie**.
- Comprendre la mise en œuvre de la simulation par événements discrets selon **DEVS** proposant des modèles de simulation modulaires et un protocole d'exécution conjointe grâce à de la communication (événements) et à de la coordination.
- Introduire une (nouvelle) bibliothèque DEVS en Java.
- Apprendre à utiliser l'approche DEVS pour simuler un exemple concret de système de contrôle cyber-physique.

Plan

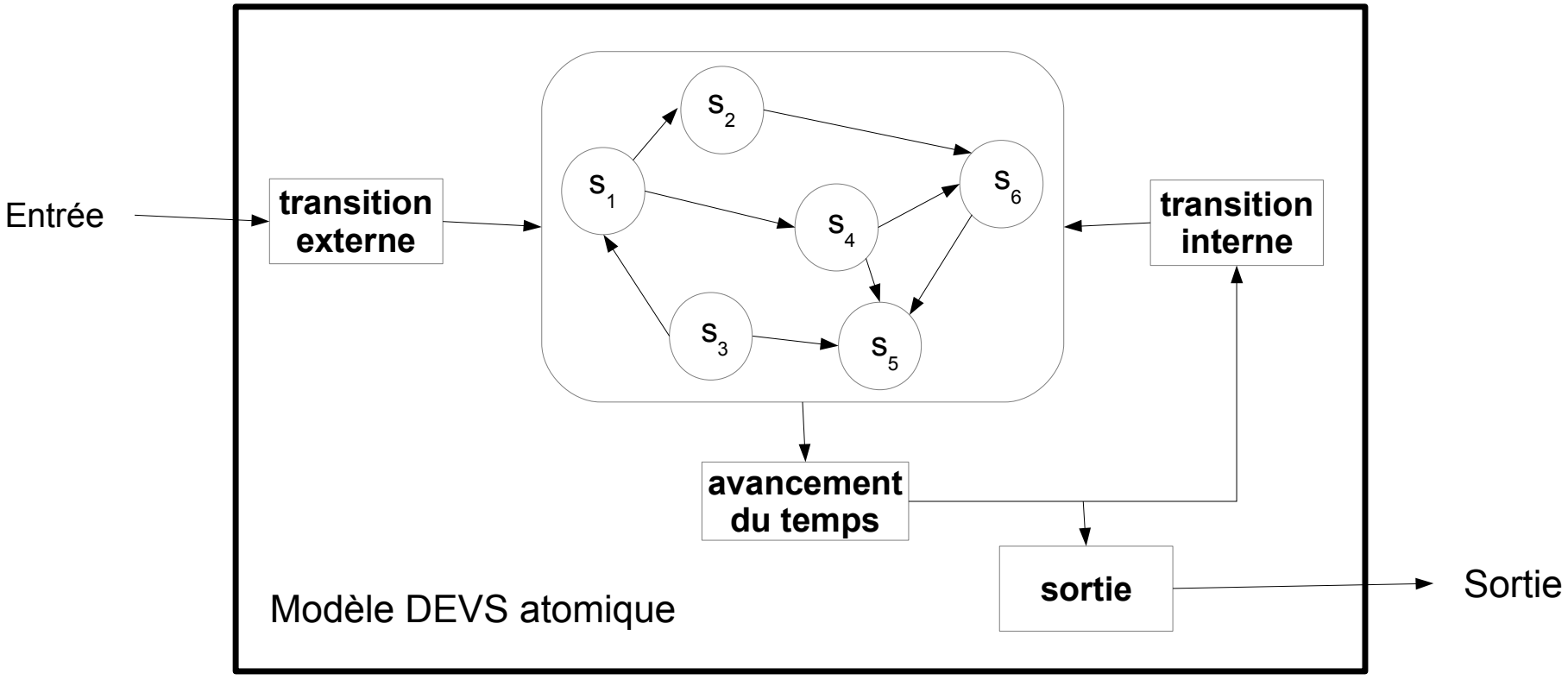
De l'intérêt de la simulation modulaire

- Les *logiciels* de simulation ont longtemps été monolithiques, à exécution centralisée et séquentielle.
  - Les versions discrètes utilisent une unique horloge et une unique liste des événements à exécuter ce qui garantit une exécution dans l'*ordre total* de leurs occurrences.
  - Les versions continues intègrent simultanément toutes les équations différentielles ensemble, ce qui garantit la connaissance des *toutes dernières* valeurs de *toutes* les variables utilisées dans le calcul croisé des dérivées et des variables.
- Ces logiciels implantent des *modèles* monolithiques qui
  - deviennent vite complexes, donc difficiles à modifier et à réutiliser ;
  - nécessitent rapidement des quantités de calcul très importantes et donc des simulations qu'on souhaiterait paralléliser et répartir.
- Pour ces raisons, des approches *modulaires* proposent :
  - des modèles composables plus faciles à définir et à réutiliser ;
  - des moteurs de simulation séparés des modèles pour utiliser différents algorithmes, par exemple pour la parallélisation et la répartition.

Modèles de simulation modulaire : DEVS

- En DEVS, l'approche *modulaire* décompose un modèle complet en modèles unitaires *échangeant entre eux des événements* dits *externes*.
- DEVS distingue deux types de modèles :
  - Les modèles ***atomiques*** jouent le rôle de simulateur d'une partie *insécable* du système à simuler ;
    - ils sont autonomes sauf pour l'échange d'événements externes et
    - ils implantent l'algorithme de simulation grâce à quatre méthodes :
      - une donnant le délai jusqu'au prochain événement interne,
      - une émettant les événements externes,
      - une exécutant les événements internes (transition interne),
      - une exécutant les événements externes reçus (transition externe).
  - Les modèles ***couplés*** *composent* des modèles atomiques ou couplés.
    - Ils définissent leurs interconnexions et les coordonnent pour les faire progresser *conjointement* dans la simulation suivant une horloge unique.
- La composition en DEVS est dîte *fermée* : le modèle couplé se présente aussi comme un modèle atomique ; en pratique, il propose aussi l'interface d'un modèle atomique.

Modèle DEVS atomique



Formalisation du protocole DEVS : modèles atomiques

$M = \langle \mathbf{X}, \mathbf{Y}, \mathbf{S}, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ où :

- **X** est l'ensemble des événements d'entrée,
- **Y** l'ensemble des événements de sortie,
- **S** l'espace d'états (interne)

Le comportement interne du modèle est défini par les fonctions :

- $ta(s) : \mathbf{S} \rightarrow \mathbb{R}^+$ d'avancement du temps disant dans combien de temps le modèle doit traiter son prochain événement interne ;
- $\delta_{int}(s) : \mathbf{S} \rightarrow \mathbf{S}$ de transition interne *i.e.*, traitement des événements internes ;
- $\lambda(s) : \mathbf{S} \rightarrow \mathbf{Y}$ produit un événement en sortie (externe) lors de la transition depuis l'état s ;

Et son comportement externe est défini par la fonction :

- $\delta_{ext}(s, e, x) : \mathbf{S} \times \mathbb{R}^+ \times \mathbf{X} \rightarrow \mathbf{S}$ de transition externe appelée lorsque $ta(s) > e$ (temps écoulé depuis le dernier événement) pour produire un nouvel état s' par traitement des événements externes venant d'être reçus.

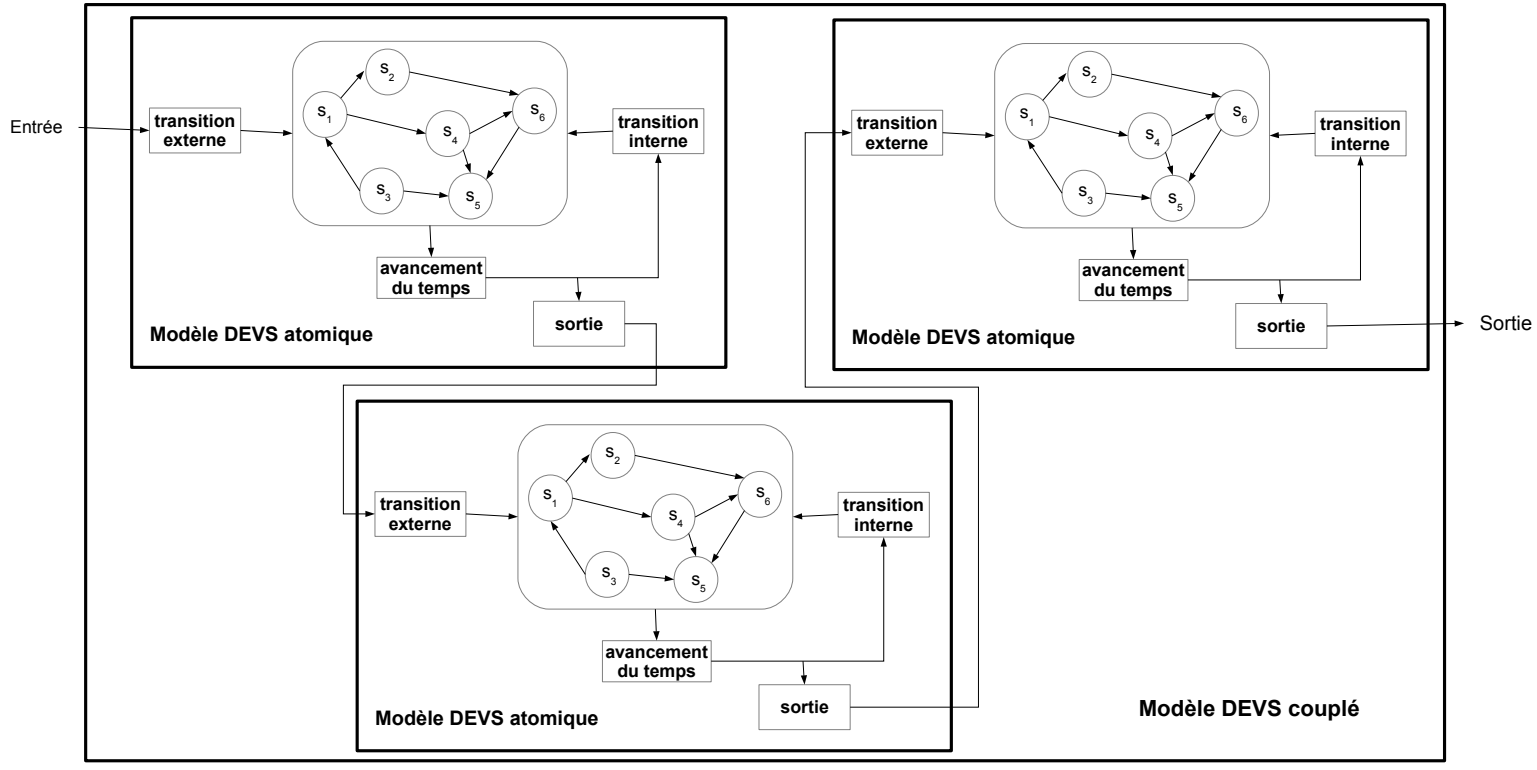
Simulation d'un modèle atomique isolé

- La simulation d'un modèle atomique isolé suit l'algorithme basique où le moteur de simulation avance d'un instant d'occurrence d'événements au suivant jusqu'à ce que l'horloge de simulation atteigne la fin de cette dernière :

Data : T, durée totale de la simulation
Data : t, horloge de simulation
1 initialiser $s \leftarrow s_{init}$ et $t \leftarrow ta(s)$;
2 **while** $t < T$ **do**
3 $\quad y \leftarrow \lambda(s)$;
4 $\quad$ **if** $y \neq \emptyset$ **then** émettre y;
5 $\quad s \leftarrow \delta_{int}(s)$;
6 $\quad t \leftarrow t + ta(s)$
7 **end**

- Lorsque le modèle est isolé, il peut émettre des événements y (ligne 4) mais ceux-ci ne seront pas pris en compte puisqu'il n'y a pas de modèles pour les recevoir.

Modèle DEVS couplé



- La *coordination* des sous-modèles se réalise en donnant à tour de rôle le contrôle à celui qui doit exécuter sa prochaine transition interne ou externe en boucle jusqu'à la fin de la simulation.
- Il s'agit de garantir que toutes les horloges de sous-modèles soient (suffisamment) synchronisées pour que l'ordre d'exécution des événements soit globalement respecté.

Formalisation du protocole DEVS : modèles couplés

$C = (X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select)$ où :

- X_{self}, Y_{self} : ensembles d'événements d'entrée et de sortie de C ;
- M_i est un sous-modèle avec $i \in D$ où $M_i = \langle \mathbf{X}_i, \mathbf{Y}_i, \mathbf{S}_i, \delta^i_{int}, \delta^i_{ext}, \lambda_i, ta_i \rangle$;
- $I_i \mid i \in D \cup self$: ensemble des indices des modèles influencés par M_i *i.e.*, recevant les événements émis par M_i ; notez que $\forall i \in D \cup \{self\}, I_i \subseteq D \cup \{self\}$ et $\forall i \in D \cup \{self\}, i \notin I_i$;
- $Z_{i,j} \mid i \in D \cup \{self\}, j \in I_i$: fonctions de traductions des événements exportés par le modèle i en événements importés par j ; notez que $Z_{i,j} : Y_i \to X_j$, $Z_{self,j} : X_{self} \to X_j$ et $Z_{i,self} : Y_i \to Y_{self}$;
- $select : 2^D \to D$: fonction de sélection du sous-modèle qui doit s'exécuter lorsque plusieurs possèdent le même délai jusqu'à l'exécution de leur prochain évenement interne.

Simulation à modèles couplés : moteur atomique coordonné

- Avec les modèles couplés, il faut assurer la coordination entre modèles atomiques pour les faire avancer tour à tour.
- Pour cela, l'algorithme des modèles atomiques est adapté et étendu pour interagir par messages (χ, μ, τ) avec son parent, où :

$\chi \in \mathbf{X} \cup \mathbf{Y} \cup \{done, *\}$	un événement ou un signal de coordination
$\mu \in D \cup \{self\}$	l'indice du modèle émetteur du message
$\tau \in [0, \infty[$	instant en temps simulé à interpréter selon le message

Variables locales

Data : t_L, instant (en temps simulé) du dernier évenement simulé
Data : t_N, instant (en temps simulé) du prochain évenement à simuler

---

Transition interne

Input $(*, p, t)$ *i.e.*, avancer au temps t;
$y \leftarrow \lambda_{self}(s)$;
if $y \neq \emptyset$ **then** $(y, self, t) \rightarrow p$;
$s \leftarrow \delta_{int}^{self}(s)$;
$t_L \leftarrow t$; $t_N \leftarrow t_L + ta_{self}(s)$;
$(done, self, t_N) \rightarrow p$

Transition externe

Input (x, p, t) *i.e.*, exécuter x au temps t;
$e \leftarrow t - t_L$;
$s \leftarrow \delta_{ext}^{self}(s, e, x)$;
$t_L \leftarrow t$;
$t_N \leftarrow t_L + ta_{self}(s)$;
$(done, self, t_N) \rightarrow p$

Simulation à modèles couplés : moteur de coordination

Variables locales

Data : i_n, indice du sous-modèle à exécuter immédiatement
Data : *active*, ensemble des indices de sous-modèles exécutant des événements à l'étape courante
Data : *parent*, parent du modèle

Algorithme principal (*main*)

initialiser les modèles et $t \leftarrow t_N$ du modèle racine;
while $t \leq T$ **do**
  $(*, main, t) \rightarrow racine$;
  wait for $(done, racine, t_N)$;
  $t \leftarrow t_N$
end

Transition interne d'un sous-modèle

Input $(*, p, t)$ *i.e.*, avancer au temps t;
$(*, self, t) \rightarrow i_n$
  where $i_n = select(\{i \in D | M_i.t_n = t\})$;
$active \leftarrow active \cup \{i_n\}$

Transition externe vers sous-modèles

Input (x, p, t) *i.e.*, exécuter x au temps t;
for $i \in I_{self}$ **do**
  $(Z_{self,i}(x), self, t) \rightarrow i$;
  $active \leftarrow active \cup \{i\}$;
end

Transition externe entre sous-modèles

Input (y, c, t) *i.e.*, exécuter y au temps t;
for $i \in I_c \backslash \{self\}$ **do**
  $(Z_{c,i}(y), self, t) \rightarrow i$;
  $active \leftarrow active \cup \{i\}$;
end
if $self \in I_c$ **then**
  $(Z_{c,self}(y), self, t) \rightarrow parent$;

Terminaison des sous-modèles

Input $(done, c, t)$;
$active \leftarrow active \backslash \{c\}$;
if $active = \emptyset$ **then**
  $t_L \leftarrow t$;
  $t_N \leftarrow min\{M_i.t_N | i \in D\}$;
  $(done, self, t_N) \rightarrow parent$
end

Moteurs de simulation DEVS

- En DEVS, les algorithmes précédents sont implantés par des *moteurs* de simulation.
  - En fait, un moteur de simulation implante un algorithme de simulation, dont ceux présentés aux transparents précédents ne sont qu'une des possibilités.
  - DEVS peut exécuter conjointement des moteurs ayant des algorithmes différents selon leur type de modèle (discret simple ou à ordonnancement d'événements, etc.).
  - Par contre, le mode de gestion du temps doit être partagé entre l'ensemble des moteurs (en temps simulé ou en temps réel).
- Lorsque des moteurs d'algorithmes *différents* sont utilisés *conjointement*, le protocole DEVS permet à un modèle couplé de coordonner des modèles de types différents via l'activation de leur moteurs hétérogènes.
  - Les algorithmes *classiques* des transparents précédents gèrent l'horloge de simulation en temps simulé de manière centralisée, passant la main à chaque modèle atomique à tour de rôle.

Simulation DEVS en temps réel

- Différents algorithmes possibles, dont les algorithmes *décentralisés* où :
  - les modèles atomiques ordonnancent comme tâches temps réel l'exécution de leurs transitions internes et externes.
  - La coordination disparaît : les modèles atomiques sont liées les uns avec les autres pour échanger directement les événements externes.
- Une transition interne se réalise par une tâche qui exécute la transition δ_{int} puis ordonnance la suivante au besoin.
- La réception d'un événement externe ordonnance une tâche immédiate exécutant δ_{ext}, annulant l'éventuelle transition interne précédemment ordonnancée pour en produire une nouvelle.
- À partir de l'arborescence des modèles, une *architecture à plat* est produite où un seul modèle couplé $G = (D, \{M_i\}, \{I_i\}, \{Z_{i,j}\})$ a pour sous-modèles tous les modèles atomiques de l'arborescence initiale, *i.e.* :
  - les M_i sont obtenus en renumérotant tous les modèles atomiques ;
  - les I_i sont ajustés selon la renumérotation ;
  - les $Z_{i,j}$ sont obtenues en ajustant en selon la renumérotation et en composant les chaînes de fonctions de traduction de l'arborescence initiale.

Algorithmes de simulation temps réel

Variable globale

Data : T, fin de la simulation en temps simulé

Variables locales

Data : t_L, instant (en temps simulé) du dernier évenement simulé

Data : t_N, instant (en temps simulé) du prochain évenement à simuler

---

Algorithme principal

$t \leftarrow$ temps simulé de départ;
for $i \in D$ **do**
  $(*, main, t) \rightarrow i$
end

Initialisation

Input $(*, from, t)$;
initialiser s;
$t_L \leftarrow t$; $t_N \leftarrow t_L + ta(s)$;
if $t_N \leq T$ **then** **ordonnancer** la première transition interne à t_N ;

Transition interne

$y \leftarrow \lambda(s)$;
if $y \neq \emptyset$ **then**
  for $i \in I_{self}$ **do**
    $(Z_{self,i}(y), self, t) \rightarrow i$
  end
end
$s \leftarrow \delta^i_{int}(s)$;
$t_L \leftarrow t$; $t_N \leftarrow t_L + ta(s)$;
if $t_N \leq T$ **then**
  ordonnancer la prochaine transition interne à t_N
end

Réception d'événement externe

Input $(x, from, t)$ *i.e.*, exécuter x au temps t;
ordonnancer la transition externe sur x à t

Transition externe

Input x;
if *une transition interne est ordonnancée* **then**
  annuler cette transition interne
end
$s \leftarrow \delta^i_{ext}(s, t - t_L, x)$;
$t_L \leftarrow t$; $t_N \leftarrow t_L + ta(s)$;
if $t_N \leq T$ **then**
  ordonnancer la prochaine transition interne à t_N
end

Moteurs de simulation temps réel

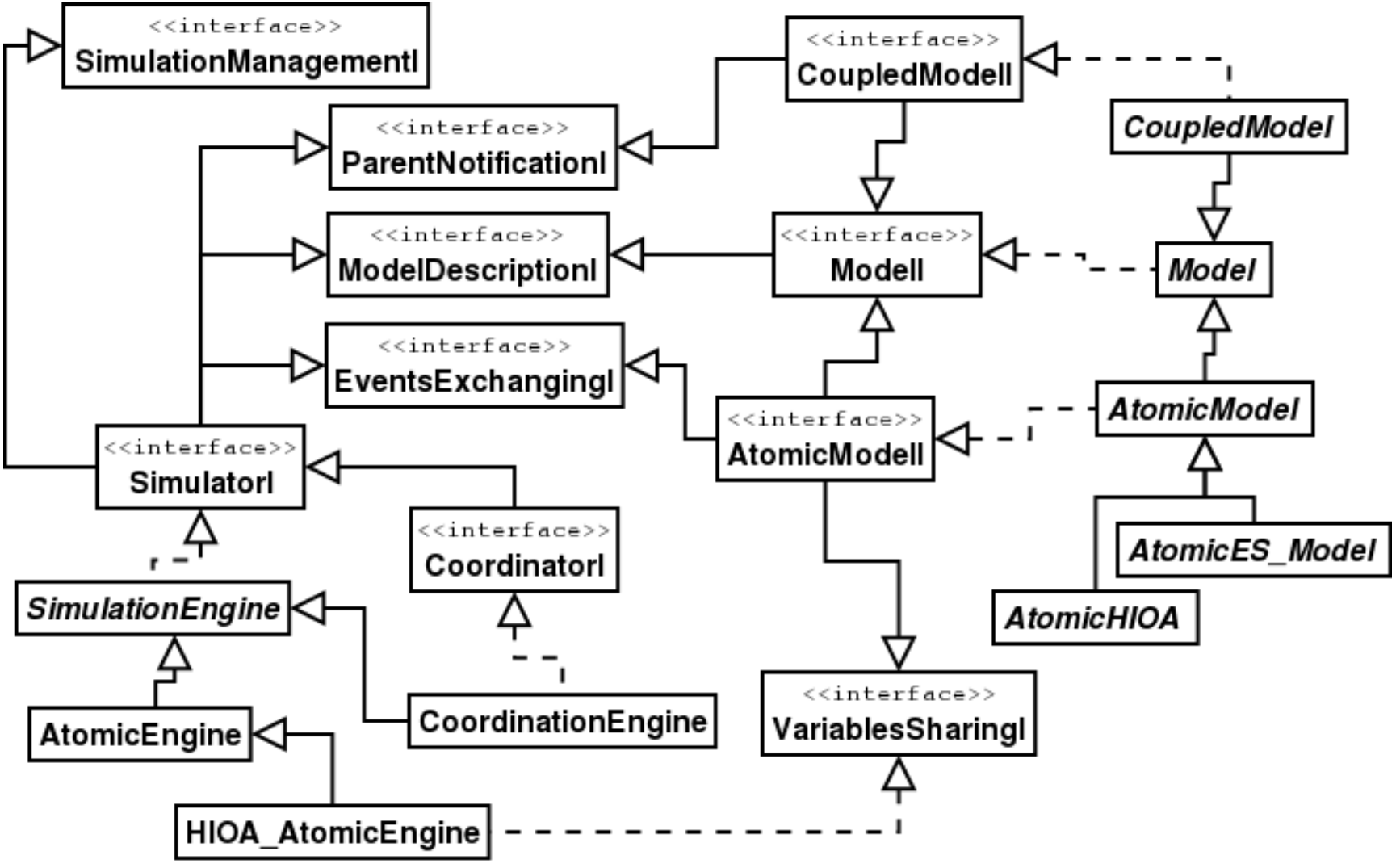
- Rappel : alignement des horloges sur le temps physique donc progression au rythme du passage du temps physique.
- Stratégie basique pour implanter une simulation temps réel à l'aide des algorithmes donnés au transparent précédent :
  1. Utiliser l'horloge de l'ordinateur comme horloge de simulation.
  2. Utiliser l'ordonnanceur du système d'exploitation pour réaliser l'ordonnancement des tâches de simulation.
- Le principal souci de cette stratégie est que l'ordonnanceur de base d'un système d'exploitation ne gère généralement pas l'ordre d'exécution des événements se produisant au même moment selon ce qui est voulu par la simulation.
- Le second est la précision de l'ordonnancement qui n'est pas très fine dans un système d'exploitation standard ($\approx 10^{-3}$ s), ce qui peut nécessiter de passer à un système d'exploitation temps réel qui offre une bien meilleure précision ($\leq 10^{-6}$ s).

Plan

DEVS en Java

- Des implantations de DEVS en Java ont déjà été réalisées, mais pas dans l'optique d'une intégration étroite dans un processus de développement logiciel des systèmes cyber-physiques.
- Notre idée, dans ses grandes lignes, consiste :
  - à implanter un *framework* DEVS avec différentes formes de modèles, de moteurs et de protocoles de simulation et coordination,
  - puis pour chaque application :
    - implanter chaque modèle atomique, ses méthodes de simulation, ses événements en entrée et en sortie, de même que les variables continues pour les HIOA ;
    - implanter chaque modèle couplé, comment ses entrées sont répercutées sur les sous-modèles, les sorties des sous-modèles réexportées et les connections exportation/importation entre les sous-modèles ;
    - *décrire* l'architecture complète à partir des *descriptions* des sous-modèles et de leurs inter-relations, puis instancier le simulateur en assemblant les modèles (instanciés et interconnectés) par construction *algorithmique*.
- Avec pour objectif à terme une intégration avec les composants.

Hiérarchie (partielle) des principales interfaces et classes



Description des interfaces et des classes des modèles I

`ModelDescriptionI` : méthodes d'accès à la description *statique* des modèles accessibles via les modèles ou leurs moteurs lors de la composition.

`ModelI` : comportements communs à tous les modèles.

`CoupledModelI` : comportements communs aux modèles couplés.

`ParentNotificationI` : comportements des modèles couplés ou de leurs moteurs permettant aux sous-modèles de notifier leur parent de leur statut (attente d'exécution d'événements externes, etc.).

`AtomicModelI` : comportements communs aux modèles atomiques.

`EventsExchangingI` : méthodes permettant d'échanger des événements entre modèles atomiques ou leurs moteurs.

`VariablesSharingI` : comportements associés au partage des variables entre modèles HIOA et leurs moteurs.

Description des interfaces et des classes des modèles II

`Model` : implante les comportements communs à tous les modèles.

`CoupledModel` : implante les comportements communs aux modèles couplés, dont le protocole DEVS ; les modèles couplés des utilisateurs en héritent pour définir leur rapport de simulation.

`AtomicModel` : implante les comportements communs aux modèles atomiques, dont le protocole DEVS ; les modèles atomiques des utilisateurs en héritent directement.

`AtomicHIOA` : implante les comportements communs aux modèles atomiques de type HIOA qui possèdent des variables continues.

`AtomicES_Model` : implante un type de modèles atomiques fonctionnant selon la vision « ordonnancement des événements » ; les modèles atomiques des utilisateurs ayant ce fonctionnement en héritent directement.

Description des interfaces et des classes des simulateurs

`SimulatorI` : comportements communs à tous les moteurs de simulation.

`SimulationManagementI` : comportements communs à tous les moteurs de simulation concernant la gestion des campagnes.

`CoordinatorI` : comportements communs spécifiques aux moteurs de simulation des modèles couplés.

`SimulationEngine` : implante les comportements communs à tous les moteurs de simulation.

`CoordinationEngine` : implante les comportements communs spécifiques aux moteurs de simulation des modèles couplés standards DEVS.

`AtomicEngine` : implante les comportements communs spécifiques aux moteurs de simulation des modèles atomiques standards DEVS.

`HIOA_AtomicEngine` : implante les comportements communs spécifiques aux moteurs de simulation des modèles atomiques de type HIOA.

Autres éléments d'une architecture de simulation DEVS

Temps, durée de simulation : gérés par des objets explicites connaissant leur unité de temps (classes `Time` et `Duration`), mais pour l'instant une architecture de simulation utilise une unique unité de temps.

Événements : `Event` implante les comportements communs à tous les événements ; les événements utilisateurs des modèles DEVS standards en héritent directement. Sa sous-classe `ES_Event` joue le même rôle pour les modèles à « ordonnancement des événements ».

Valeurs des variables continues : les valeurs des variables continues sont conservées dans des objets spécifiques (classe `Value` et ses sous-classes) pour les partager par référence entre modèles et l'ajout de services (historisation des valeurs, interpolation, etc.).

Rapports de simulation : pour automatiser la collecte de résultats, les modèles peuvent définir des rapports à la fin de chaque exécution (`SimulationReportI` et classes prédéfinies).

Trace et journalisation : outils fournis par la bibliothèque pour tracer et journaliser les actions en vue de déverminage, et pour tracer des courbes d'évolution pour les variables continues.

Architecture de simulation : ce sera vu un peu plus loin...

Plan

Classes de modèles atomiques basiques

- Créer un modèle atomique consiste à définir une sous-classe concrète d'une des classes abstraites fournies dans la bibliothèque choisie en fonction du type de modèle.
- La classe concrète définit des implantations pour les méthodes déclarées dans les interfaces dont le contenu dépend du comportement particulier du modèle utilisateur.
- Pour toutes celles héritant d'`AtomicModel`, on doit définir :
  - `timeAdvance` et `output` (*ta* et λ du protocole DEVS) ;
  - `userDefinedInternalTransition` (δ_{int}) pour l'exécution des différents types d'événements internes ;
  - `userDefinedExternalTransition` (δ_{ext}) pour l'exécution des différents types d'événements externes.
- La documentation de la bibliothèque présente les autres méthodes et les variables accessibles dans les classes abstraites et qui peuvent être utilisées ainsi que les relations entre les valeurs importantes (comme le temps simulé lors de l'exécution d'un événement).

Exemple : TicModel I

- L'événement `TicEvent` :

```
public class TicEvent extends Event {                  // superclasse de tout types d'événements
  private static final long serialVersionUID = 1L;      // événements sérialisables
  public TicEvent(Time timeOfOccurrence) {
    super(timeOfOccurrence, null);                      // pas de contenu pour ces événements
  }
}
```

- Le modèle `TicModel` :

```
@ModelExternalEvents(exported = {TicEvent.class})       // annotation ou ajout par programmation
public class TicModel extends AtomicModel {
  public static double STANDARD_DELAY = 60.0;           // délai standard entre tics
  protected Duration delay;                             // délai sous forme de Duration
  public TicModel(String uri,                           // URI du modèle
                  TimeUnit tu,                          // unité de temps (millisecondes, heures, ...)
                  SimulatorI engine                     // moteur de simulation associé
                 ) throws Exception {
    super(uri, tu, engine);
    this.delay = new Duration(STANDARD_DELAY, tu);
  }

  @Override public Duration timeAdvance() { return this.delay; }

  @Override public Vector<EventI> output() {
    Vector<EventI> ret = new Vector<EventI>();
    // la mise à jour de l'horloge se fait après, dans internalTransition
    Time t = this.getCurrentStateTime().add(this.getNextTimeAdvance());
    ret.add(new TicEvent(t)); return ret;
  }
}
```

Exemple : TicModel II

- Utilisation (hors architecture de simulation vue plus loin) :

Programme principal

```
// cas d'un modèle atomique solitaire
public class TestTicModel {
  public static void main(String[] args) {
    try {
      // création du moteur de simulation
      AtomicEngine e = new AtomicEngine();
      // création du modèle avec e comme moteur
      new TicModel("TicModel", TimeUnit.SECONDS, e);
      // lancement d'une simulation de 620 secondes
      e.doStandAloneSimulation(0.0, 620.0);
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

Trace des événements émis et des transitions internes exécutées[1]

```
1571673680575|TicModel|output TicEvent(60.0)
1571673680576|TicModel|at internal transition 60.0 60.0
1571673680576|TicModel|output TicEvent(120.0)
1571673680576|TicModel|at internal transition 120.0 60.0
1571673680576|TicModel|output TicEvent(180.0)
1571673680576|TicModel|at internal transition 180.0 60.0
1571673680576|TicModel|output TicEvent(240.0)
1571673680576|TicModel|at internal transition 240.0 60.0
1571673680576|TicModel|output TicEvent(300.0)
1571673680577|TicModel|at internal transition 300.0 60.0
1571673680577|TicModel|output TicEvent(360.0)
1571673680577|TicModel|at internal transition 360.0 60.0
1571673680577|TicModel|output TicEvent(420.0)
1571673680577|TicModel|at internal transition 420.0 60.0
1571673680577|TicModel|output TicEvent(480.0)
1571673680577|TicModel|at internal transition 480.0 60.0
1571673680577|TicModel|output TicEvent(540.0)
1571673680577|TicModel|at internal transition 540.0 60.0
1571673680577|TicModel|output TicEvent(600.0)
1571673680577|TicModel|at internal transition 600.0 60.0
```

[1] avec des instructions de trace qui n'apparaissent pas dans le code du transparent précédent.

Exécution individuelle des modèles atomiques

- Un modèle atomique peut être associé à un moteur de simulation pour être exécuté isolément.
  - Attention, tout modèle atomique n'est pas nécessairement exécutable en isolation s'il dépend d'autres modèles.
  - Un tel modèle peut importer et exporter événements ou variables, à condition d'avoir une logique d'exécution qui peut s'en passer.
  - Quand dans un assemblage une importation ou une exportation n'est pas connectée à un autre modèle, elle est simplement ignorée dans l'exécution correspondante.
- La bibliothèque fournit des moteurs de simulation adaptés (lorsque nécessaire) aux différents types de modèles atomiques.
  - Par exemple, pour les modèles basiques comme `TicModel`, on utilise le moteur `AtomicEngine`.
- Un moteur de simulation propose une méthode `doStandAlone-Simulation` permettant d'exécuter une simulation avec un instant simulé initial et une durée simulée fixée.

Exemple : wifi, partie bande passante

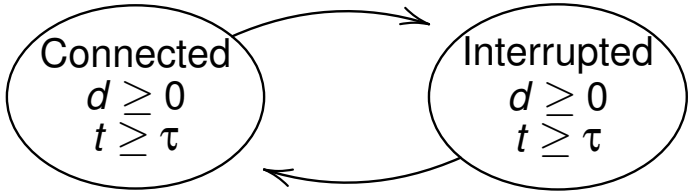
- Rappel (cours 3) : quatre modèles atomiques :
  1. **NetInt** : modèle à « ordonnancement d'événements » générant alternativement des événement d'interruption puis de reprise selon des lois de probabilités prédéfinies.
  2. **NetBand** : modèle HIOA produisant une variable bande passante continue et consommant les événements d'interruption et de reprise pour mettre à zéro puis reprendre l'évolution continue.
  3. **TicModel** : modèle basique générant des événements `TicEvent` à intervalle régulier prédéterminé (déjà présenté mais en plus générique).
  4. **BSensor** : modèle HIOA important la variable bande passante et les événements `TicEvent` pour produire des événements lectures à chaque fois qu'il reçoit un `TicEvent`.
- Le modèle couplé **WiFiModel** est obtenu de la composition des quatre précédents ; il est un TIOA qui émet les événements lecture de bande passante. Il sera couvert à la section suivante.

Modèles atomiques pour la bande passante WiFi

- Modèles HIOA (du cours 3) obtenus d'une phase de spécification (formelle) à traduire en modèles de simulation DEVS :

Name : **NetInt**	$U : \varnothing$ $Y : \varnothing$ $X : \{d, \tau, x_1, x_2\}$ $I : \varnothing$ $O : \{\text{Interrupt, Resume}\}$ $H : \varnothing$

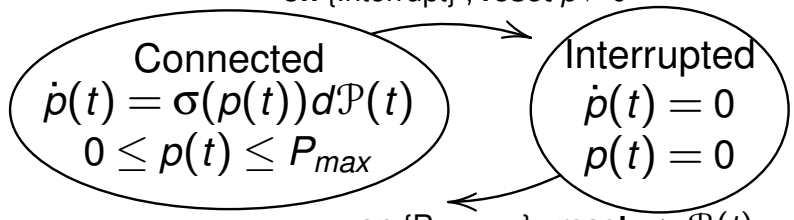
when $d{=}t{-}\tau$; **emit** {Interrupt}, **reset** $d \leftarrow x_2 \sim \textbf{Exp}[\lambda_2], \tau \leftarrow t$



when $d{=}t{-}\tau$; **emit** {Resume}, **reset** $d \leftarrow x_1 \sim \textbf{Exp}[\lambda_1], \tau \leftarrow t$

Name : **NetBand**	$U : \varnothing$ $Y : \{p\}$ $X : \varnothing$ $I : \{\text{Interrupt, Resume}\}$ $O : \varnothing$ $H : \varnothing$

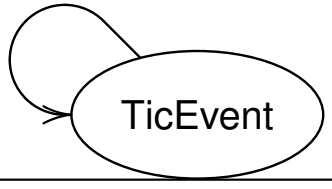
on {Interrupt} ; **reset** $p \leftarrow 0$



on {Resume} ; **reset** $p \leftarrow \mathcal{R}(t)$

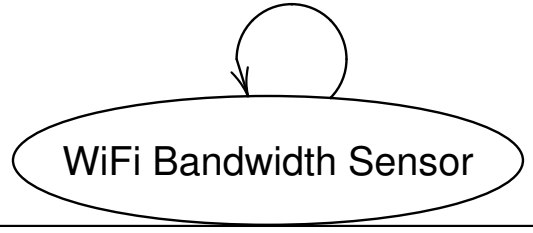
Name : **TicModel**	$U : \varnothing$ $Y : \varnothing$ $X : \{d\}$ $I : \varnothing$ $O : \{\text{TicEvent}\}$ $H : \varnothing$

when $d = t$; **emit** {TicEvent}, **reset** $d = t + \Delta$



Name : **BSensor**	$U : \{p\}$ $Y : \varnothing$ $X : \varnothing$ $I : \{\text{TicEvent}\}$ $O : \{\text{BandwidthReading}\}$ $H : \varnothing$

on {TicEvent} ; **emit** {BandwidthReading(p)}



U, Y, X : variables continues en entrée, sortie, internes ; I, O, H : événements en entrée, en sortie, internes.

Modèle **NetInt**

- Idée : générer une séquence d'événements d'interruption puis de reprise avec des durées aléatoires entre eux.
- Modèle à ordonnancement d'événements où chaque événement engendre le prochain à déclencher.
- Classe `WiFiDisconnectionModel`. Notez :
  - les annotations de la classe donnant les événements exportés ;
  - la classe hérite de `AtomicES_Model` ;
  - la définition du rapport de simulation, son contenu et le code inclus dans la classe pour l'engendrer ;
  - la définition explicite des états du modèle par type énumération ;
  - la définition des paramètres d'exécution explicites (densités de probabilités) et leur initialisation ;
  - l'utilisation des générateurs de nombre aléatoires de `common-math` ;
  - l'utilisation de la méthode héritée `scheduleEvent` pour ajouter les événements à l'ordonnancement.

Modèle **NetBand**

- Il s'agit d'un modèle HIOA exportant une variable `bandwidth` en utilisant une équation différentielle stochastique.
- Classe `WiFiBandwidthModel`. Notez :
  - les annotations de la classe donnant les événements importés et la variable exportée ;
  - la définition du rapport de simulation ;
  - la définition explicite des états du modèle par type énumération ;
  - la définition des paramètres d'exécution explicites (densités de probabilités) et leur initialisation ;
  - la définition de la variable du modèle `bandwidth` exportée, son annotation et son initialisation statique ;
  - l'intégration de l'équation différentielle stochastique en utilisant des générateurs de nombre aléatoires de `common-math` ;
  - le calcul en avance de la prochaine valeur de bande passante pour assurer qu'on ne déborde pas du maximum ni sous 0 ;
  - le traitement des événements importés pour basculer d'un état à l'autre et tenir à jour la variable continue.

Modèle **BSensor**

- Idée : le modèle HIOA importe la variable `bandwidth` d'un modèle de bande passante et un événement `TicEvent` puis produit un événement `BandwidthReading` portant la valeur instantanée de `bandwidth` à chaque réception de `TicEvent`.
- Classe `WiFiBandwidthSensorModel`. Notez :
  - les annotations donnant événements importé et exporté ainsi que la variable importée ;
  - la classe hérite de `AtomicHIOAwithEquations` ;
  - la définition du rapport de simulation ;
  - la définition de la variable `bandwidth` importée, son annotation mais absence d'initialisation laissée à la composition ;
  - le traitement des événements importés pour déclencher l'émission de l'événement exporté ;
  - la méthode `output` émettant les événements exportés.

Plan

La classe `CoupledModel` I

- C'est la classe abstraite permettant de créer tous les types de modèles couplés actuellement possibles dans la bibliothèque.
- Pour les modèles couplés n'ayant que des sous-modèles à événements discrets, on instancie un modèle couplé en fournissant :
  - son URI, son unité de temps simulé et son moteur de simulation ;
  - un tableau de ses sous-modèles (de taille supérieure à 1) ;
  - trois ensembles de correspondances (*maps*) définissant :
    1. la relation entre événements importés par le modèle couplé et ses sous-modèles les important ;
    2. la relation entre les événements exportés par les sous-modèles et leur ré-exportation par le modèle couplé ;
    3. les connexions entre événements exportés par des sous-modèles et importés par d'autres.

  Sachant qu'à chaque fois les événements peuvent ne pas être identiques mais nécessiter une traduction.
- Pour les modèles couplés ayant des sous-modèles possédant des variables continues s'ajoutent :

La classe `CoupledModel` II

- trois autres ensembles de correspondances (*maps*) définissant :
  1. la relation entre les variables importées par le modèle couplé qui sont importées par leurs sous-modèles ;
  2. la relation entre les variables exportées par le modèle couplé qui sont exportées par leurs sous-modèles ;
  3. les connexions entre les variables exportées par des sous-modèles qui sont importées par d'autres sous-modèles.

  Sachant qu'à chaque fois les noms des variables et leurs types (modulo leur conformité) peuvent ne pas être identiques et nécessiter des traductions.
- Il aurait été en théorie possible d'avoir une classe de modèles couplés hybrides héritant d'une classe de modèles couplés discrets, mais nous avons choisi par simplicité de ne proposer qu'une seule classe directement capable de traiter des sous-modèles de types homogènes ou hétérogènes.

Pourquoi définir des modèles couplés spécifiques

- Le rôle des modèles couplés est essentiellement de réaliser la composition/connexion puis la cordination de leurs sous-modèles.
- Les algorithmes de coordination sont entièrement implantés par la bibliothèque.
- Il ne devrait donc pas être nécessaire de définir des sous-classes de modèles couplés spécifiques à un problème mais simplement instancier la classe prédéfinie `CoupledModel`.
  - La bibliothèque prévoit tout de même cette définition pour gérer la création des rapports de simulation *du modèle couplé* à partir des rapports de simulation de ses sous-modèles.
  - Donc, assez souvent, les classes de modèles couplés de l'utilisateur se bornent à définir une classe pour leur rapport de simulation et implantent la méthode `getFinalReport` pour créer et retourner leur instance de rapport à partir des rapports des sous-modèles.

Assemblage

- L'assemblage des modèles est le processus par lequel les modèles de simulation vont être instanciés et inerconnectés pour ensuite pouvoir être exécutés.
- Comme déjà indiqué, la bibliothèque permet de décrire une architecture de simulation et cette description va ensuite être parcourue *algorithmiquement* pour instancier et connecter les modèles.
- La description d'une architecture désigne les classes à utiliser pour instancier les modèles et les paramètres à passer à leurs constructeurs lors de l'instantiation.
- Elle utilise :
  - des *descripteurs* de modèles atomiques ;
  - des *descripteurs* de modèles couplés.
- En fait, il s'agit d'une forme de DSL d'assemblage, mais dont les programmes sont représentés par des arbres de syntaxe abstraite, sans syntaxe concrète.

Descripteur de modèles atomiques

- Un descripteur de modèle atomique contient toute l'information nécessaire pour instancier le modèle ; il fournit une méthode `create` qui va réaliser cette instantiation.
- La méthode `create` prend les paramètres suivants :

`modelClass` : pour un modèle décrit par une classe `M`, l'instance de `Class<M>` *i.e.*, `M.class` ;

`modelURI` : l'URI du modèle à créer ;

`simulatedTimeUnit` : l'unité de temps de l'horloge de simulation du modèle ;

`amFactory` : une « fabrique » pour instantier le modèle si le protocole d'instantiation de `M` le requiert ;

`engineCreationMode` : indique quel type de moteur de simulation ququel doit être attaché l'instance de modèle (choix entre pas de moteur, un moteur atomique ou un moteur de coordination).

Exemple

- Création d'un descripteur possible du modèle `TicModel` :

```
AtomicModelDescriptor.create(
    TicModel.class,     // l'instance de Class<TicModel>
    TicModel.URI,       // Une URI possible (attention à l'unicité...)
    TimeUnit.SECONDS,   // unité de temps de l'horloge
    null,           // pas de fabrique donc la fabrique standard fournie
                    // attachement à un moteur de simulation atomique
    SimulationEngineCreationMode.ATOMIC_ENGINE)
```

- Cette méthode `create` s'appuie également sur le fait que les événements importés et exportés sont donnés par l'annotation sur la classe définissant le modèle atomique, ici `TicModel`.
- Pour l'utilisation de fabriques spécifiques, il faut se reporter à la documentation de la bibliothèque pour voir comment faire.

Descripteur de modèles couplés I

- Les descripteurs de modèles couplés sont créés directement par instantiation de la classe de descripteur.
- Outre l'ensemble des URIs de sous-modèles qui vont permettre de faire le lien avec les descripteurs de ces derniers dans l'architecture, on doit fournir les relations d'importation/exportation.
- Pour les événements, il y a trois relations à définir.
  1. Les relations entre événement importés par le modèle couplé et ceux des sous-modèles auxquels ils sont passés.
     - Par exemple, l'événement `InterruptionEvent` émis par le modèle `WiFiModel` et reçu par le modèle `PC` est retransmis à son sous-modèle `PCStateModel` :

```
// Création de la "map"
Map<Class<? extends EventI>,EventSink[]> imported =
                    new HashMap<Class<? extends EventI>,EventSink[]>();
imported.put(
  InterruptionEvent.class, // Type de l'événement importé
  new EventSink[] {
    new EventSink(PortableComputerStateModel.URI, // sous-modèle récepteur
                  InterruptionEvent.class)});     // type de son événement importé
```

Descripteur de modèles couplés II

2. Les relations entre les événements exportés par des sous-modèles et ceux qui sont ré-exportés par le modèle couplé :
   - Par exemple, l'événement `InterruptionEvent` exporté par `WiFiModel` qui est en fait ré-exporté depuis le sous-modèle `WiFiDisconnectionModel` :

```
Map<Class<? extends EventI>,ReexportedEvent> reexported =
                        new HashMap<Class<? extends EventI>,ReexportedEvent>();
reexported.put(
  InterruptionEvent.class,   // Événement exporté par WiFiModel
  new ReexportedEvent(WiFiDisconnectionModel.URI, // sous-modèle exportant
                        InterruptionEvent.class));  // type de son événement exporté
```

3. Les connexions entre sous-modèles exportant et important des événements.
   - Par exemple, les événements passés entre le modèle de déconnexion et la modèle de bande passante WiFi :

```
Map<EventSource,EventSink[]> connections =
                    new HashMap<EventSource,EventSink[]>();
EventSource from11 =
  new EventSource(WiFiDisconnectionModel.URI, // sous-modèle exportant
                    InterruptionEvent.class);   // type d'événement exporté
EventSink[] to11 =
  new EventSink[] {
    new EventSink(WiFiBandwidthModel.URI,     // sous-modèle important
                    InterruptionEvent.class)};  // type d'événement important
connections.put(from11, to11);
```

Descripteur de modèles couplés III

- Pour les variables, il y a similairement trois relations à définir.
  1. Les relations entre variables importées par le modèle couplé et passées aux sous-modèles.
  2. Les relations entre les variables exportées par des sous-modèles et ré-exportées par le modèle couplé.
  3. Les connexions entre des variables exportées par des sous-modèles et des variables importées par d'autres sous-modèles.
     - Par exemple, la variable `bandwidth` entre le modèle de bande passante et le modèle senseur :

```
Map<VariableSource,VariableSink[]> bindings =
                    new HashMap<VariableSource,VariableSink[]>();
VariableSource source11 =
  new VariableSource("bandwidth",      // nom de la variable exportée
                     Double.class,     // son type
                     WiFiBandwidthModel.URI);  // sous-modèle exportant
VariableSink[] sinks11 =
  new VariableSink[] {
    new VariableSink("bandwidth",      // nom de la variable importée
                     Double.class,     // son type
                     WiFiBandwidthSensorModel.URI)};  // sous-modèle important
bindings.put(source11, sinks11);
```

Exemple : le modèle `WiFiModel`

- Une fois l'ensemble des relations d'événements et de variables exportées et importées définies, on crée le descripteur du modèle couplé par simple instantiation :

```
new CoupledHIOA_Descriptor(
  WiFiModel.class,  // classe de modèle couplé à instancier
  WiFiModel.URI,    // URI du modèle à créer
  submodels ,       // ensemble des URIs des sous-modèles
  imported,         // événements importés par le modèle couplé
  reexported,       // événements re-exportés par le modèle couplé
  connections,      // événements exportés et importés par des sous-modèles
  null,             // pas de fabrique, donc la fabrique standard
                    // attachement du modèle couplé à un moteur de coordination
  SimulationEngineCreationMode.COORDINATION_ENGINE,
  null,             // pas de variable importée par le modèle couplé
  null,             // pas de variable exportée par le modèle couplé
  bindings)         // variables exportées et importées par des sous-modèles
```

Une architecture et son instantiation

- Une architecture complète est créée à partir des informations suivantes :
  1. L'URI du modèle couplé qui se trouve à sa racine.
  2. Un ensemble de correspondances entre URI de modèles atomiques et leurs descripteurs.
  3. Un ensemble de correspondances entre URI de modèles couplés et leurs descripteurs.
  4. L'unité de temps commune (pour le moment) à tous les modèles de l'architecture.
- Pour instancier cette architecture *algorithmiquement*, il faut exécuter la méthode `constructSimulator`, ce qui parcourt tous les descripteurs, crée et interconnecte les sous-modèles puis retourne une référence sur le moteur de simulation attaché au modèle à la racine de l'architecture.

Exemple : architecture MolèNE

- Création de l'architecture, son instantiation puis lancement d'une exécution d'une simulation :

```
ArchitectureI architecture =
  new Architecture(
    MoleneModel.URI,          // URI du modèle à la racine de l'architecture
    atomicModelDescriptors,   // map URI -> descripteurs de modèles atomiques
    coupledModelDescriptors, // map URI -> descripteurs de modèles couplés
    TimeUnit.SECONDS) ;       // unité de temps des horloges de simulation

SimulationEngine se = architecture.constructSimulator() ;
se.doStandAloneSimulation(0.0, 5000.0) ;
```

- La documentation de la bibliothèque et les exemples montrent aussi l'utilisation du *protocole de passage de paramètres d'exécution* avec la méthode `setSimulationRunParameters`, ce qui permet de programmer des campagnes de simulations automatiques.

Récapitulons...

1. Les moteurs de simulation classiques sont **monolithiques** et forcent généralement à définir des modèles de simulation qui sont eux-mêmes monolithiques.
2. DEVS offre une alternative consistant à définir des **modèles de simulation modulaires** qui décomposent le modèle global en modèles atomiques assemblés par des modèles couplés.
3. L'avantage de cette approche est de permettre la définition de modèles de simulation **réutilisables** sur étagères.
4. L'exécution des modèles de simulation est réalisée par des **moteurs de simulation** également modulaires qui peuvent implanter différents algorithmes et différentes modalités dont la simulation **temps réel**, la simulation **parallèle** et la simulation **répartie**.
5. Une **nouvelle bibliothèque DEVS en Java** est proposée et l'exemple MolèNE y est implanté à partir des modèles d'automates hybrides définis précédemment.

Pour aller plus loin : sélection de lectures recommandées

- *Guide to Modeling and Simulation of Systems of Systems*, B.P. Zeigler et H.S. Sarjoughian, Springer, 2013.
- *The split system approach to managing time in simulations of hybrid systems having continuous and discrete event components*, J. Naturo et al., Trans. of the Society for Modeling and Simulation, 88(3), 2012.
- *Simulation temps-réel distribuée de modèles numériques : application au groupe motopropulseur*, A. Ben Khaled, INRIA, thèse de l'U. de Grenoble, 2014.
- *The Discrete EVent System specification (DEVS) formalism*, Hans Vangheluwe, notes de cours, 2001.
- *Discrete Event Modelling and Simulation*, Hans Vangheluwe, notes de cours, 2001.