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 6

Simulation modulaire et DEVS

Objectifs pédagogiques du cours 6

- Comprendre les principales problématiques d'implantation des simulateurs au passage à la **simulation modulaire et répartie** voire **temps réel**.
- 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.
- Développer le principe d'une **intégration de la simulation au sein d'un modèle composants** pour les CPCS en vue de soutenir une méthodologie de développement propre à ces derniers.
- Introduire une (nouvelle) bibliothèque DEVS en Java.
- Comprendre l'intégration des modèles de simulation DEVS au sein du modèle à composants BCM.
- Apprendre à utiliser conjointement composants et simulateurs DEVS pour implanter un exemple concret de CPS.

Plan

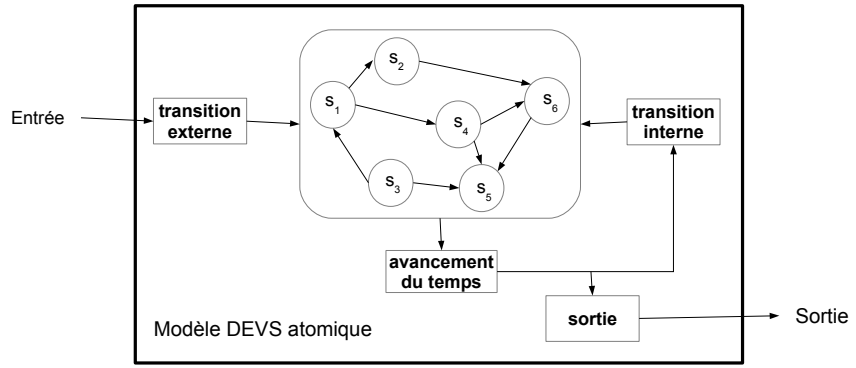
De l'intérêt de la simulation modulaire

- Les moteurs de simulation ont longtemps été monolithiques, à exécution centralisée et séquentielle.
  - Les moteurs discrets utilisent une unique horloge et une unique liste des événements à exécuter ce qui garantit une exécution dans l'ordre de leurs occurrences.
  - Les moteurs continus 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.
- Mais cela va de pair avec un modèle monolithique qui
  - devient vite complexe et donc difficile à modifier,
  - nécessite rapidement des quantités de calcul très importantes et donc des simulations qu'on souhaiterait paralléliser.
- Pour ces deux raisons, sont proposées des approches *modulaires* où les modèles sont plus faciles à définir et à réutiliser ainsi qu'ouvrant la voie à une exécution parallèle et répartie des simulations.

Modèles de simulation modulaire : DEVS

- En DEVS, l'approche *modulaire* consiste à décomposer un modèle en sous-modèles *échangeant 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.
    - Selon la modalité d'exécution, il peut les coordonner pour les faire progresser *conjointement* dans la simulation.
- 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

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

- $\mathbf{X}$ est l'ensemble des événements d'entrée,
- $\mathbf{Y}$ l'ensemble des événements de sortie,
- $\mathbf{S}$ l'espace des états

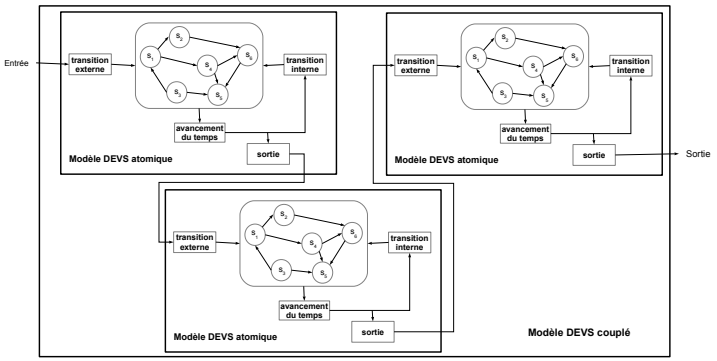
La dynamique interne du modèle est définie par les fonctions :

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

Et son comportement externe est défini par la fonction :

- $\delta_{ext}(s, e, x) : \mathbf{S} \times \mathbb{R}^+ \times \mathbf{X} \to \mathbf{S}$ 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.

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 composé avec $i \in D$;
- $I_i \mid i \in D \cup self$: ensemble des indices des modèles influencés 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.

Les notes de cours d'H. Vangheluwe (voir références au dernier transparent) montrent comment implanter les fonctions des modèles atomiques pour les modèles couplés, ce qui revient à implanter la coordination.

Moteurs de simulation DEVS

- Pour exécuter les modèles atomiques et couplés, DEVS introduit des *moteurs* de simulation.
- Un modèle de simulation vu comme atomique est attaché à un moteur qui va l'exécuter en appelant en boucle ses méthodes jusqu'à la fin fixée pour l'exécution courante.
- Un moteur de simulation implante un algorithme de simulation et DEVS autorise à avoir des moteurs ayant des algorithmes différents, qu'ils soient imposés par le type de modèle à simuler (discret simple, à ordonnancement d'événements, etc.) ou par le type d'exécution souhaitée (séquentielle, parallèle, répartie, etc.).
- DEVS autorise également des simulations où des moteurs d'algorithmes *différents* sont utilisés *conjointement* ; dans ce cas les moteurs implantent également un protocole qui permet à un moteur couplé de coordonner des modèles de types différents via l'activation de leur moteurs hétérogènes.

Moteurs de simulation répartis et temps réel

- Chaque modèle DEVS possède sa propre horloge de simulation.
- Passer au parallélisme peut se résumer à laisser chaque modèle atomique avancer à son propre rythme, mais alors comment synchroniser les horloges pour maintenir la cohérence globale ?
- Avec le protocole DEVS de base, la coordination s'en assure, mais en le suivant strictement, on rencontre deux limitations :
  - un seul modèle atomique pouvant s'exécuter à la fois, cela élimine toute forme de parallélisme ;
  - au passage à une exécution répartie, les nombreux appels entre modèles à chaque pas de simulation imposent des délais insupportables s'ils deviennent des appels à distance (RPC/RMI).
- Pour garantir la cohérence, il faut s'assurer (en théorie) qu'un événement externe émis au temps t_e du modèle émetteur soit reçu et exécuté au même temps t_r par le receveur *i.e.*, $t_e = t_r$.

  Que faire pour éviter qu'à la réception $t_r > t_e$ sans pour autant bannir tout parallélisme ?

Synchronisation des horloges de simulation réparties

- Trois approches de synchronisation ont été étudiées :
  1. conservatrice : synchronisation totale de toutes les horloges (ce que réalise le protocole de coordination DEVS) ;
  2. sans synchronisation : laisser les horloges évoluer à leurs rythmes ce qui produit des erreurs de simulation (inversion d'événements) mais qui peuvent être admissibles dans certains cas ;
  3. optimiste (*time warp*) : les horloges avancent à leur rythme mais l'arrivée d'un événement dans le passé force un retour en arrière.
- En synchronisation conservatrice, trois techniques :
  1. Synchronisation explicite (sémaphore, ...), mais coûteuse.
  2. Ordonnancement fixé à l'avance garantissant le bon ordre d'exécution, mais ne fonctionne mal/pas en réparti.
  3. Synchronisation implicite fondée sur le temps (*time-triggered*) :
     - en temps réel (resp. accéléré) : synchronisation du temps simulé (resp. avec un facteur d'accélération) sur des horloges matérielles synchronisées, synchronisant implicitement les modèles.

  mais, n'étant pas parfaite, elle peut produire des occurences dans le passé (généralement récent) dues aux erreurs de synchronisation des horloges matérielles et aux délais de communication.

Simulation et architectures logicielles CPS : objectifs

- La simulation est essentielle pour tester systématiquement le code des systèmes cyber-physiques et autonomiques.
- Chaque composant ayant un comportement autonomique ou cyber-physique, il serait intéressant :
  - de modéliser *individuellement* leur comportement et leurs dépendances envers d'autres modèles ;
  - d'attacher ces modèles de comportement aux composants.
- Selon l'approche « composants », il devient alors possible :
  - de construire le *modèle global* d'une application en *composant* les modèles de chacun des composants,
  - de construire le *simulateur global* d'un système en *composant* leurs simulateurs modulaires.

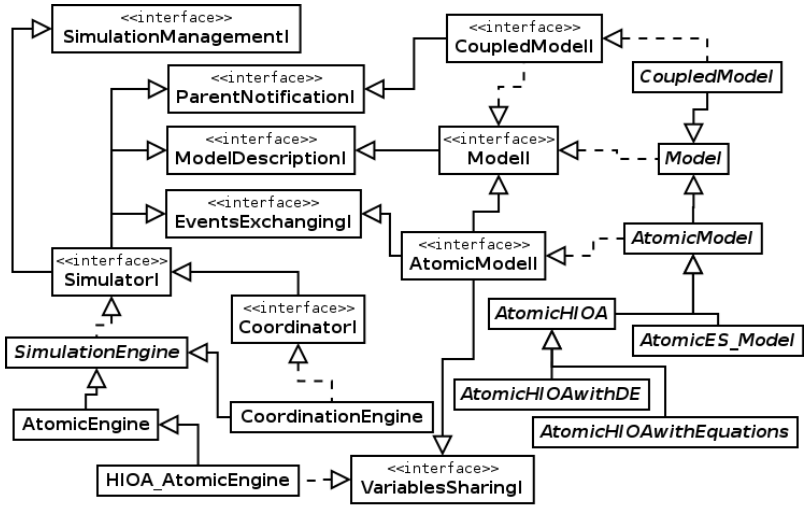
  Et intégrer ces formes de compositions dans la composition entre composants CPS.
- À titre de preuve de concept, nous allons examiner l'intégration d'une telle fonctionnalité en BCM, ce qui exige d'abord de pouvoir programmer des simulateurs DEVS en Java.

Plan

DEVS en Java

- Des implantations de DEVS en Java ont déjà été réalisées, mais
  - elles ne s'intéressent qu'à la simulation et au protocole DEVS ;
  - elles n'ont pas le niveau d'abstraction supplémentaire pour décrire une architecture de simulation par composition explicite de modèles.
- Notre idée, dans ses grandes lignes, consiste donc :
  - à implanter un *framework* DEVS avec différentes formes de modèles, de moteurs et de protocoles de simulation et coordination,
  - puis pour chaque application :
    - d'implanter chaque modèle atomique, ses méthodes de simulation, ses événements en entrée et en sortie, de même que les variables continues pour les HIOA ;
    - d'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 un simulateur complet en assemblant les modèles (instanciés et interconnectés) par construction *algorithmique*.

Hiérarchie (partielle) des principales interfaces et classes



Description des interfaces et des classes des modèles I

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

`ModelI` : définit les comportements communs à tous les modèles.

`CoupledModelI` : définit les comportements communs aux modèles couplés.

`ParentNotificationI` : définit les 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` : définit les comportements communs aux modèles atomiques.

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

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

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

`CoupledModel` : implante les comportements communs aux modèles couplés, incluant les comportements standards du protocole DEVS ; les modèles couplés des utilisateurs n'utilisant que ce protocole en héritent directement, principalement pour définir la forme de leurs rapports de simulation.

`AtomicModel` : implante les comportements communs aux modèles atomiques, incluant les comportements standards du protocole DEVS ; les modèles atomiques des utilisateurs n'utilisant que ce protocole en héritent directement.

Description des interfaces et des classes des modèles II

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

`AtomicHIOAwithDE` : implante les comportements communs aux modèles atomiques de type HIOA dont les variables continues évoluent selon des équations différentielles ; les modèles HIOA des utilisateurs utilisant ce genre d'évolutions en héritent directement.

`AtomicHIOAwithEquations` : implante les comportements communs aux modèles atomiques de type HIOA dont les variables continues évoluent selon des équations (algébriques) ; les modèles HIOA des utilisateurs utilisant ce genre d'évolutions en héritent directement.

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

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

`CoordinatorI` : définit les 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 : ils sont 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 : la classe `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 événements des modèles à vision « 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 permettre le partage par référence entre les modèles et l'ajout des services (historisation des valeurs, interpolation, etc.).

Rapports de simulation : pour automatiser les campagnes de simulation, tous les modèles définissent la forme des rapports par lesquels ils retournent leurs résultats à la fin de chaque exécution (interface `SimulationReportI` et les classes `AbstractSimulationReport` et `StandardCoupledModelReport`).

Trace et journalisation : des outils sont fournis par la bibliothèque pour tracer et journaliser les actions en vue de déverminage, et aussi 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` du protocole DEVS ;
  - `userDefinedInternalTransition` pour l'exécution des différents types d'événements internes ;
  - `userDefinedExternalTransition` 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 {
  private static final long serialVersionUID = 1L ;
  public TicEvent(Time timeOfOccurrence) {
    super(timeOfOccurrence, null);
  }
}
```

- Le modèle `TicModel` :

```
@ModelExternalEvents(exported = {TicEvent.class})
public class TicModel extends AtomicModel {
  protected Duration delay ;
  public TicModel(String uri, TimeUnit tu, SimulatorI engine) throws Exception {
    super(uri, simulatedTimeUnit, simulationEngine) ;
    this.delay = new Duration(60.0, TimeUnit.SECONDS) ;
  }

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

  @Override public Vector<EventI> output() {
    Vector<EventI> ret = new Vector<EventI>() ;
    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) :

```
public class TestTicModel {
  public static void main(String[] args) {
    try {
      AtomicEngine e = new AtomicEngine() ;
      new TicModel("TicModel", TimeUnit.SECONDS, e) ;
      e.doStandAloneSimulation(0.0, 620.0) ;
    } catch (Exception e) {
      throw new RuntimeException(e) ;
    }
  }
}
```

```
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
```


Classes abstraites de modèles atomiques prédéfinies

La bibliothèque DEVS offre les classes abstraites suivantes à étendre pour créer des modèles atomiques (les niveaux d'items représentent la relation d'héritage) :

- `AtomicModel` : permet de définir des modèles à événements discrets basiques par définition des méthodes
  - `AtomicHIOA` : modèles à événements discrets mais aussi avec des variables continues *i.e.*, de la simulation continue ou hybride.
    - `AtomicHIOAwithDE` : modèles HIOA dont certaines variables continues sont calculées par intégration numérique supposant donc le calcul de dérivées (d'autres pouvant être calculées par équations algébriques).
    - `AtomicHIOAwithEquations` : modèles HIOA dont toutes les variables continues sont calculées à partir d'équations algébriques uniquement.
  - `AtomicES_Model` : modèles discrets à ordonnancement d'événements (*Event Scheduling*).

Exécution individuelle des modèles atomiques

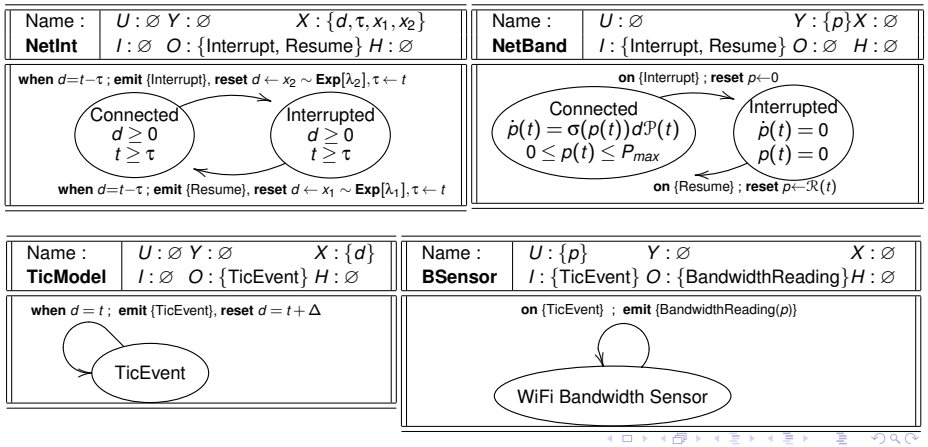
- Un modèle atomique peut être associé à un moteur de simulation pour être exécuté isolément.
  - Attention, tout modèle atomique n'est pas nécessairement exécutable en isolation s'il dépend d'autres modèles.
  - Il peut importer et exporter événements ou variables, à condition d'avoir une logique d'exécution qui peut s'en passer.
  - Quand dans un assemblage une importation ou une exportation n'est pas reliée 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 `doStandAloneSimulation` permettant d'exécuter une simulation avec un instant simulé initial et une durée simulée fixée.

Exemple : wifi, partie environnement

- 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 obtenus d'une phase de spécification (formelle) à traduire en modèles de simulation DEVS :



Modèle **NetInt**

- Idée : générer une séquence d'événements d'interruption puis de reprise avec une durée aléatoire *d* entre eux.
- Modèle à ordonnancement d'événements où chaque événement engendre le prochain à déclencher dans un délai *d*.
- 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 la bibliothèque `common-math` ;
  - l'utilisation de la méthode héritée `scheduleEvents` pour ajouter les événements à l'ordonnancement.

Modèle **NetBand**

- Il s'agit d'un modèle exportant une variable `bandwidth` en utilisant une équation différentielle stochastique.
- Modèle hybride avec équation différentielle.
- Classe `WiFiBandwidthModel`. Notez :
  - les annotations de la classe donnant les événements importés ;
  - la classe hérite de `AtomicHIOAwithDE` ;
  - 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 ;
  - l'utilisation des générateurs de nombre aléatoires de la bibliothèque `common-math` ;
  - la définition de la variable du modèle `bandwidth` exportée, son annotation et son initialisation statique ;
  - 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 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`.
- Modèle HIOA, pour pouvoir importer et travailler sur des variables continues, mais de type « avec équations » uniquement.
- Classe `WiFiBandwidthSensorModel`. Notez :
  - les annotations donnant événements importé et exporté ;
  - 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 discrets, on instancie un modèle couplé en fournissant :
  - son URI, son unité de temps simulé et son moteur de simulation (si nécessaire, voir plus loin, sinon `null`) ;
  - un tableau de ses sous-modèles (de taille supérieure à 1) ;
  - trois ensembles de correspondances (*maps*) définissant :
    - la relation entre événements importés par le modèle couplé et ses sous-modèles les important ;
    - la relation entre les événements exportés par les sous-modèles et leur ré-exportation par le modèle couplé ;
    - 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.

La classe `CoupledModel` II

- Pour les modèles couplés ayant des sous-modèles possédant des variables continues s'ajoutent :
  - trois autres ensembles de correspondances (*maps*) définissant :
    - la relation entre les variables importées par le modèle couplé qui sont importées par leurs sous-modèles ;
    - la relation entre les variables exportées par le modèle couplé qui sont exportées par leurs sous-modèles ;
    - 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 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 cordination de leurs sous-modèles.
- Cette coordination est totalement définie par DEVS, donc il ne devrait pas être nécessaire de définir des classes de modèles couplés spécifiques à un problème mais simplement instancier la classe existante.
- 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.

Fermeture sur la composition versus exécution

- Rappel : l'ensemble des modèles DEVS atomiques est fermé sur la composition *i.e.*, le modèle couplé résultant d'une composition de sous-modèles se présente également comme un modèle atomique.
- En pratique, cela veut dire :
  - qu'en plus des méthodes implantant la coordination ds sous-modèles, le modèle couplé possède également une implantation des méthodes des modèles atomiques ;
  - qu'un modèle couplé peut être exécuté individuellement comme s'il s'agissait d'un modèle atomique.

Assemblage

- L'assemblage des modèles est le processus par lequel les modèles de simulation vont être instanciés et connectés pour ensuite pouvoir être exécutés.
- Comme déjà indiqué, l'une des innovations de la bibliothèque est de pouvoir décrire une architecture de simulation qui va pouvoir 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.

Descripteur de modèles atomiques

- Un descripteur de modèle atomique contient toute l'information nécessaire pour instancier le modèle et 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 simulaiton 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 ce qui est d'utiliser des 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 qui sont passés à ses sous-modèles.
     - Par exemple, l'événement `InterruptionEvent` émis par le modèle `WiFiModel` et reçu pr 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 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.
     - 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 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, il faut exécuter la méthode `constructSimulator`, ce qui 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.

Plan

Principes d'intégration de DEVS/BCM

- Rappel : l'intégration DEVS/BCM vise à définir des modèles de comportements de type systèmes hybrides (automates hybrides) pour chaque composant puis de les attacher à ces derniers sous la forme de modèles DEVS.
- En BCM, les composants peuvent être répartis sur différentes JVM et différents ordinateurs hôtes, donc la communication entre ceux-ci peut passer par le réseau (numérique).
- Cette possibilité nous pousse à imposer que tout système hybride attaché à un composant soit un TIOA *i.e.*, ils ne peuvent importer et exporter des *événements* mais *pas de variables*.
- L'architecture de simulation au sens de la bibliothèque DEVS va devoir être répartie sur plusieurs composants, ce qui implique d'étendre cette notion à la correspondance entre parties d'architectures de simulation et composants (description de la liaison modèles de simulation/composants).

Éléments d'intégration DEVS/BCM

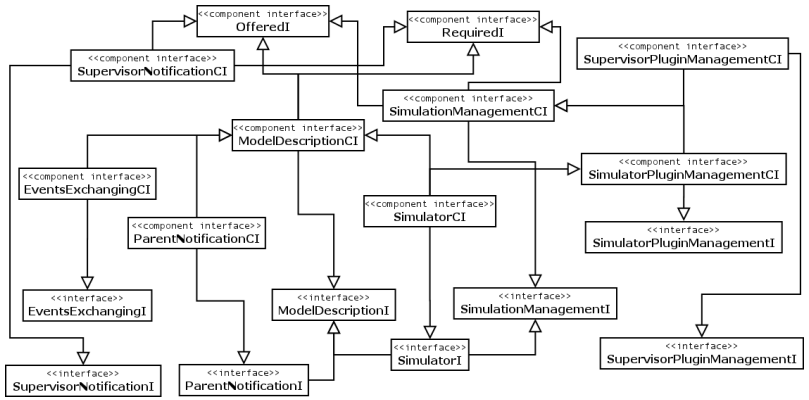
- L'intégration du *framework* de simulation DEVS dans BCM consiste à :
  - attacher des modèles DEVS aux composants, ce qui se fait par des greffons (*plug-ins*) ;
  - interconnecter des modèles en passant par des connections inter-composants *i.e.*, ports et connecteurs BCM et
  - gérer la création de simulateurs globaux puis l'exécution et la gestion de campagnes de simulations, ce qui peut également se faire par des greffons spécifiques.
- On peut distinguer trois rôles à jouer :
  1. le simulateur atomique,
  2. le coordonnateur exécutant un modèle couplé et
  3. le superviseur qui assure
     - la création et l'interconnexion de l'architecture de simulation projetée sur une architecture de composants et
     - la gestion des campagnes de simulation (lancement, récupération des résultats puis analyse de ces derniers).

Implantation de l'intégration DEVS/BCM

- Pour intégrer le *framework* DEVS et BCM, il faut d'abord faire en sorte que les appels entre modèles/moteurs de simulation puissent se faire entre composants.
  - Cela implique d'avoir des interfaces de composants pour chaque interface du *framework* DEVS.
  - Le principe de ce passage a été de créer des interfaces de composants qui *étendent* les interfaces du *framework*.

  Ex.: l'interface de composants `ModelDescriptorCI` étend les interfaces de composants `OfferedI` et `RequiredI` de même que l'interface `ModelDescriptorI` du *framework* DEVS.
- Ensuite, l'intégration exige d'avoir des ports entrants et sortants de même que des connecteurs pour chacune de ces interfaces de composants (fournis par la bibliothèque).
- Les greffons auront pour premier rôle d'ajouter les interfaces de composants offertes et requises ainsi que les ports à leur composant hôte puis de gérer les interconnnexions correspondantes.

Hiérarchie des principales interfaces d'intégration



Les greffons d'intégration DEVS/BCM et leurs rôles I

- Outre la gestion des interfaces, ports et des connecteurs, les greffons vont recevoir les appels de méthodes entre moteurs et modèles situés dans différents composants.
  - Ils doivent donc implanter (au sens Java) les interfaces appropriées du *framework* DEVS pour jouer un rôle de *proxy* en relayant ces appels aux moteurs et modèles qu'ils contiennent. Ex.: les greffons jouant le rôle de simulateurs (atomiques ou de coordination) doivent implanter l'interface `SimulatorI`.
  - Des interfaces (comme `AbstractSimulatorPluginI`) regroupent en les étendant toutes les interfaces du *framework* DEVS que les greffons doivent implanter.
- Ils doivent aussi créer l'architecture de simulation à la fois en leur sein pour les modèles et moteurs qu'ils doivent contenir et interagir pour créer la partie de l'architecture inter-composants.
  - Le greffon de simulation de modèles atomiques permet de créer des architectures de simulation locales (sous-arbres) puis utilise la fermeture de la composition pour présenter au reste de l'architecture ce sous-arbre local comme un modèle atomique.

Les greffons d'intégration DEVS/BCM et leurs rôles II

- Le greffon de supervision connaît l'architecture globale (arborescence depuis la racine jusqu'aux racines des sous-arbres locaux vues comme des modèles atomiques) et l'instantiation de cette architecture crée les modèles couplés, leurs moteurs de coordination et leurs greffons sur les composants désignés.

- Enfin, outre les liens des modèles couplés vers leurs sous-modèles directs, le *framework* DEVS utilise certaines références Java dans les architectures de simulation pour interagir :
  - directement entre modèles atomiques par l'interface `EventsExchangingI`,
  - des sous-modèles vers leur modèle couplé parent par l'interface `ParentNotificationI`.

  Ces connexions vont dans certains cas devoir se faire via des connexions entre composants, ce qui est gérer par les greffons.

Création de modèles/moteurs locaux aux composants

- Les modèles/moteurs DEVS locaux sont des arborescences qui peuvent importer et exporter des événements (ce sont des TIOA).
- L'idée est de créer une architecture locale, de l'attacher au composant par le greffon puis de le présenter à l'architecture globale comme un modèle atomique.
  - Ceci produit une forme d'abstraction en boîte noire qui convient bien à la programmation par composants.
- L'idée est donc de cacher au sein du composant l'architecture locale en s'arrangeant pour en créer les modèles/moteurs lors de la création du composant *i.e.*, lors de la création et de l'installation du greffon.
  Ex.: pour le composant `WiFi` on aurait

```
Architecture localArchitecture =
  new Architecture(WiFiModel.URI, atomicModelDescriptors,
                   coupledModelDescriptors, TimeUnit.SECONDS) ;
AtomicSimulatorPlugin asp = new AtomicSimulatorPlugin() ;
asp.setPluginURI(WiFiModel.URI) ;
asp.setSimulationArchitecture(localArchitecture) ;
this.installPlugin(asp) ;
```


Architectures combinées DEVS/BCM

- Pour décrire un modèle dans l'architecture globale, il suffit d'ajouter à sa description l'URI du port de reflexion entrant du composant qui possède ou va devoir créer ce modèle.
- Les composants qui possèdent une architecture locale sont décrit comme des modèles atomiques, masquant ainsi les détails de cette architecture locale.
- Les composants qui vont détenir des modèles couplés n'ont pas à créer de greffon lors de leur création ; les greffons vont être créés lors du processus d'instantiation de l'architecture de simulation globale.
- Du point de vue de l'utilisateur, il y a donc assez peu de changements par rapport à une architecture du *framework* DEVS ; ce sont les descripteurs spécifiques à l'intégration DEVS/BCM et le greffon de supervision qui implantent l'algorithme d'instantiation croisée DEVS/BCM.

Exemple WiFi I

- Dans l'exemple WiFi, le modèle de la bande passante WiFi est détenu par un composant `WiFi` et son descripteur dans l'architecture globale est créé par :

```
ComponentAtomicModelDescriptor.create(
  WiFiModel.URI,      // URI du modèle
  null,               // pas d'événements importés
  (Class<? extends EventI>[])
    new Class<?>[]{WiFiBandwidthReading.class,  // événements exportés
                   InterruptionEvent.class,
                   ResumptionEvent.class},
  TimeUnit.SECONDS,  // unité de temps
                     // URI du port de réflexion entrant du composant hôte
                     // ici, on les a préalablement mises dans une "map"
  modelURIs2componentURIs.get(WiFiModel.URI))
```

- Les autres modèles « atomiques » décrits, le modèle couplé global appelé `MoleneModel` est décrit par :

Exemple WiFi II

```
ComponentCoupledModelDescriptor.create(
  MoleneModel.class,     // classe à instancier pour créer le modèle
  MoleneModel.URI,       // URI du modèle à créer
  submodels,             // ensemble des URIs de ses sous-modèles
  null,                  // pas d'événements importés
  null,                  // pas d'événements exportés
  connections,           // connexions exportation/importation
  null,                  // pas de fabrique, fabrique standard
  SimulationEngineCreationMode.COORDINATION_ENGINE, // moteur
                 // URI du port de réflexion entrant du composant hôte
  modelURIs2componentURIs.get(MoleneModel.URI))
```

- Et enfin, l'architecture et le greffon superviseur sont créés par :

```
ComponentModelArchitecture architecture =
  new ComponentModelArchitecture(
        MoleneModel.URI,            // URI du modèle couplé racine
        atomicModelDescriptors1,    // descripteurs des modèles «~atomiques~«
        coupledModelDescriptors,    // descripteurs des modèles couplés
        TimeUnit.SECONDS) ;         // unité de temps de simulation
this.sp = new SupervisorPlugin(architecture) ;
sp.setPluginURI("supervisor") ;
this.installPlugin(this.sp) ;       // dans le composant superviseur hôte
```


Création et exécution

- À l'exécution, le composant superviseur utilise la référence sur son greffon pour créer l'architecture de simulation en appelant la méthode `createSimulator` puis lancer la simulation en appelant la méthode `doStandAloneSimulation`.
  Exemple : en supposant que la variable `sp` contient la référence au greffon de supervision

```
sp.createSimulator() ;
sp.doStandAloneSimulation(0, 5000L) ;
```

- Avec la possibilité d'utiliser comme dans le *framework* DEVS le protocole de passage de paramètres d'exécution.

Embryon de composant cyber-physique en BCM

- Pour aller vers des composants spécifiquement cyber-physiques, la bibliothèque d'intégration BCM/Simulation DEVS offre une classe abstraite `AbstractCyPhyComponent` àfaire hériter par les classes définissant les composants CPS utilisateurs portant les modèles atomiques.
- Pour l'instant, elle définit :
  - une variable `asp` qui va contenir la référence au greffon de type `AtomicSimulatorPlugin` facilitant la référence à ce dernier dans le code du composant ;
  - une méthode abstraite `createLocalArchitecture` qui doit donc être implantée dans les composants CPS des utilisateurs pour construire et retourner l'architecture de simulation locale à ce composant.
- D'autres propriétés devront être ajoutées pour améliorer l'intégration BCM/DEVS mais aussi la capacité à s'exécuter en temps réel, des interfaces riches intégrant les besoins en ressources, l'observabilité, etc.

Communication entre composants et modèles

- Dans les simulations SIL, il est généralement nécessaire d'être capable de communiquer entre composants et modèles :
  - le composant peut avoir besoin d'accéder une valeur de variable d'un modèle qu'il contient ;
  - le modèle peut vouloir accéder à une donnée du composant.
- L'intégration DEVS/BCM prévoit pour l'instant des moyens limités pour ce faire :
  - Le greffon de simulation implante l'interface `ModelStateAccessI` qui fournit un protocole permettant d'accéder une valeur dans un modèle par un appel à une méthode `getModelStateValue`.
  - Le composant peut implanter l'interface `EmbeddingComponent-StateAccessI` qui fournit un protocole permettant à un modèle d'accéder à une valeur contenu dans son composant hôte par un appel à une méthode `getEmbeddingComponentStateValue` ; le protocole de passage de paramètres d'exécution peut alors être utilisé pour initialiser la référence dans le(s) modèle(s).
- Exemple du sèche-cheveux.

Récapitulons...

1. Parmi les techniques de simulation utilisables pour tester et valider les entités autonomiques, on trouve principalement la **simulation hybride modulaire répartie *en temps réel*** comme une extension des modèles DEVS de simulation par événements discrets modulaires.

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.