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 8

De l'entité adaptable — conception des actionneurs et méthodologie de développement

Objectifs pédagogiques du cours 8

- Comprendre les problématiques liées à la *fonctionnalité d'adaptation*, à la conception des *interfaces actionneurs* d'une entité adaptable).
- Introduire des *techniques d'implantation* des fonctions d'adaptation et des actionneurs.
- Établir des *principes de conception et de modélisation* qui vont servir de base à une méthodologie et un processus de développement logiciel spécifique aux entités adaptables (conception, développement, test, validation et déploiement).
- 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.
- Développer des *savoir-faire* dans la conception et la mise en œuvre des éléments gérés en informatique autonomique.

Plan

Problématiques

- La conception de la fonction adaptation est trop souvent abordée sous les hypothèses *simplificatrices* implicites suivantes :
  - les opérations d'adaptation sont *élémentaires* et *orthogonales*,
  - elles se réalisent *individuellement* sur *chaque* entité adaptable et
  - elles *n'interfèrent pas* avec le fonctionnement de l'élément lui-même, des autres éléments et du système en général,

  en réalité *fausses* pour tout système *réaliste* !
- Portant, elles induisent bien un effet de bord, un *changement d'état*, sur l'entité adaptable.
- Distinguons (à nouveau) opérations d'adaptation (implantation) et leur visibilité externe (interfaces), les questions se posant sont :
  - Quelles adaptations sont nécessaires et quelles opérations offrir ?
  - Quelles propriétés adaptables affectent-elles et comment ?
  - Comment les mettre en œuvre et les rendre robustes ?
  - Comment en tirer des interfaces cohérentes et flexibles ?
  - Comment gérer les adaptations multiples, dépendantes et réparties entre plusieurs entités ?

Opérations d'adaptation *élémentaires*

- Concevoir une entité adaptable suppose d'identifier des opérations d'adaptation *élémentaires* se caractérisant par leur :
  - *exclusion mutuelle*, ne pouvant s'exécuter en parallèle ni avec le fonctionnement normal de l'entité ni avec toute autre opération d'adaptation sur celle-ci ;
  - *complétude*, formant un passage direct d'un état de fonctionnement *cohérent* à un autre état de fonctionnement *cohérent* ;
  - *non interruptibilité* (ou *atomicité*), car leur interruption laisserait l'entité dans un état de fonctionnement incohérent, ses états intermédiaires étant dans ce cas incohérents (en général).
- La notion d'opération élémentaire dépend des opérations elles-mêmes, car ces propriétés impliquent de suspendre l'exécution de l'entité adaptée pendant toute la durée de leur exécution.
- Ex.: modification du nombre de threads dans un pool de threads ou d'un seuil dans un filtre sur des données.

Opérations d'adaptation composites

- D'autres opérations sont trop complexes, trop longues et impactant d'autres entités, pour être considérées élémentaires. Ex.: modifier le nombre de MV d'une application.
- Elles peuvent s'appuyer sur des opérations élémentaires, en les *composant* pour former une adaptation *composite* cohérente.
- Elles se caractérisent donc par leur :
  - *durée*, trop longue pour simplement suspendre l'exécution de l'entité pendant *toute leur exécution*, une gestion plus fine de l'exclusion mutuelle devenant nécessaire ;
  - *parallélisabilité*, se décomposant en plusieurs (sous-)adaptations ou calculs qui peuvent se faire (partiellement) en parallèle ;
  - *non indépendance*, pouvant devoir être entrelacées avec des opérations d'adaptation sur d'autres entités pour respecter des invariants locaux ou globaux du système.

Ces caractéristiques sont déjà visibles dans l'exemple d'adaptation à la bande passante du réseau sans fil.

Atomicité et adaptations atomiques

- Si une opération d'adaptation nécessite un temps significatif, comment traiter les évolutions et les événements se produisant *pendant* son exécution ?
- Il peut devenir nécessaire de l'interrompre en cas d'erreur ou si ses conditions de déclenchement ne sont plus réunies.
  Exemple : dans le cas de l'échange de données par réseau sans fil, que faire si la bande passante redevient forte (resp. faible) pendant une adaptation ?
- Lors d'une interruption, distinguons l'état du système entre :
  - un état *fatalement incohérent* imposant d'identifier et protéger des parties élémentaires atomiques ;
  - d'un état *partiellement incohérent* laissant les entités dans des états incohérents mais réparables ;
  - un état *cohérent* mais où l'adaptation n'a pas été complètement réalisée, devant aussi être réparé mais moins urgemment.

Opération atomique

Opération d'adaptation élémentaire ou composite totalement réalisée ou pas du tout, grâce à un mécanisme de réparation ou de retour arrière (*rollback*).

Supervision et coordination des opérations d'adaptation

- Les opérations d'adaptation réparties sur plusieurs entités nécessitent une coordination d'ensemble, mais comment ?
- Deux options sont possibles :
  1. une approche maître/esclave imposant une *entité de supervision*, l'une des entités touchées ou une entité tierce, ordonnançant les opérations d'adaptation en les appellant dans l'ordre ; ou
  2. une approche pair-à-pair, soit directement entre entités coopérantes.
- Dans les deux cas, cela va nécessiter l'insertion d'*instructions intermédiaires* dans les opérations d'adaptation pour assurer leur synchronisation avec le superviseur ou entre les entités elles-mêmes :
  - outils de synchronisation (sémaphores, barrières de synchronisation, jointures, etc.) ;
  - échanges de messages, signaux ou données (approches de type protocoles), possiblement pair-à-pair.

Opérations d'adaptation supervisables et coordonnables

- Un continuum d'approches de conception possibles allant :
  - d'opérations à gros grain : fondée sur des opérations de longue durée exécutant des notifications, des attentes et des synchronisations au sein de leur code supervisables de l'extérieur ;
  - à des opérations à grain fin : fondée des opérations élémentaires où les seules synchronisation possibles sont les points d'entrée et de sortie (*i.e.*, appel/retour), et donc où le superviseur synchronise uniquement en ordonnançant les opérations.
- Le choix ne peut se faire dans l'absolu, cela dépend de l'application et du contexte :
  - gros grain : simplifie la conception des entités en offrant directement des opérations d'adaptation complète et cohérente ;
  - grain fin : plus ouverte et flexible, facilitant la mise en œuvre de multiples opérations composites par composition des opérations élémentaires, mais peut être plus complexe à superviser.

Plan

Mise en œuvre des actionneurs

- L'actionneur est aux opérations d'adaptation ce que le capteur est aux opérations de mesure :
  - un actionneur utilise une opération d'adaptation implantée au sein de l'entité adaptable pour réaliser une action sur l'entité ;
  - il est rendu visible par une interface actionneur.
- Comme pour les capteurs, la grande variété des cas rend difficile toute tentative d'élaboration d'une méthodologie générale.
- Nous nous concentrons sur les concepts suivants :
  - relation entre opérations d'adaptation et fonctionnement de l'entité adaptable,
  - coopération et coordination entre actionneurs,
  - interruptibilité, traitement des erreurs et exécution atomique.

Adaptation versus exécution des entités adaptables

- Sauf pour des cas particuliers, adapter une entité doit se faire au moins partiellement en exclusion mutuelle avec l'exécution de ses services fonctionnels.
- La conception d'une entité adaptable doit donc assurer cette exclusion mutuelle, ce qui a des conséquences dont il faut limiter l'impact des sur l'exécution « normale ».
  - L'accès exclusif global, par exemple en utilisant des clauses `synchronize` systématiques, impliquerait potentiellement la sérialisation totale de toute exécution de code sur l'entité.
  - Prévoir plutôt des sections critiques correctement délimitées mises en œuvre par des verrous « read/write locks », par exemple.
- Dans cette optique, comme avec la concurrence en général, il faut bien cerner les sections critiques des opérations d'adaptation pour obtenir l'exclusion mutuelle strictement *nécessaire*.

Coopération entre adaptations multiples

- Exemple MolèNE : illustration des difficultés soulevées par l'adaptation de plusieurs entités adaptables :
  - adaptations intégrant ou retirant la compression/décompression dans les chaînes d'émission/réception de données.
- Clairement, se posent deux problèmes dans l'interaction entre le fonctionnement des deux entités et l'adaptation à réaliser :
  1. Comment entrelacer les étapes de mise en place de l'adaptation entre les deux entités ?
  2. Comment faire en sorte que l'émetteur ne commencent pas à compresser les données envoyées avant que le récepteur ne soit prêt à les décompresser (et vice versa) ?
     - *Chaque entité réalise ses propres adaptations, mais les deux doivent coopérer pour que les échanges de données se déroulent correctement en tout temps.*
     - La composition de ces opérations locales forme une opération inter-entités *atomique* qui doit être réalisée sans s'interrompre, complètement ou pas du tout.

Coopération entre opérations d'adaptation

- Pour l'activation de la compression, les deux entités vont devoir effectuer les opérations suivantes :
  1. Créer et initialiser les éléments logiciels de compression/décompression
  2. Acquérir l'accès exclusif à chacun des composants en interrompant les échanges de données
  3. Intégrer les éléments logiciels de compression/décompression dans chacun des composants
  4. Basculer dans le deux composants vers la chaîne d'émission avec compression/décompression
  5. Relâcher l'accès exclusif à chacun des composants pour reprendre les échanges de données
- Adaptation en deux phases, calcul puis basculement (*commit*), ou validation de l'adaptation complétée.
- Synchronisations avant d'interrompre les échanges, et à nouveau avant de reprendre.
- *Comment prendre en compte ces aspects ?*

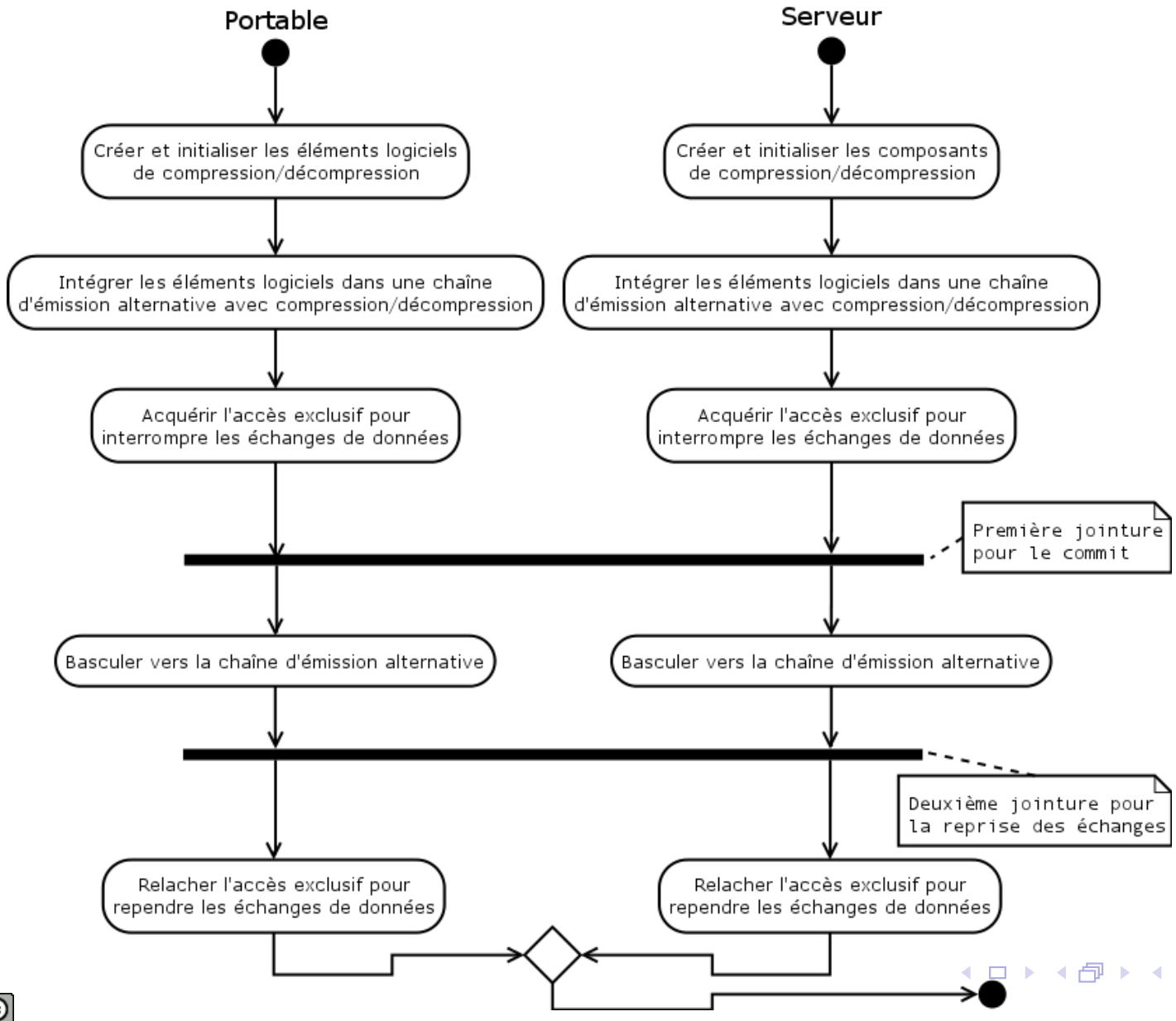
Protocoles de coopération entre actionneurs

Protocole

Ensemble des conventions nécessaires pour faire coopérer des entités indépendantes *i.e.*, assurer la cohérence entre leurs exécutions ainsi qu'établir et entretenir des échanges d'informations entre elles.

1. Un protocole peut s'exprimer de différentes manières avec différents outils, par exemple :
   - Automates, les états marquant la progression dans le protocole et les transitions indiquant les prochains événements et opérations possibles.
     C'est l'approche classique dans les domaines des réseaux et habituelle en programmation répartie, surtout pour le pair-à-pair.
   - En UML, les diagrammes d'activité permettent d'exprimer le déroulement des comportements et des échanges entre différentes entités coopérantes.
     C'est une approche plus couramment utilisée en modélisation et développement logiciel.

Modélisation des activités d'adaptation pour la compression



Mise en œuvre

- Les deux *jointures* du diagramme d'activités précédent peuvent se réaliser de plusieurs manières :
  - Utiliser une entité de supervision recevant les notifications des deux entités et leur envoyant des signaux pour les coordonner.
  - Utiliser conjointement un même synchroniseur.
  - Lier les deux entités directement en pair-à-pair pour qu'elles s'échangent directement des signaux.
- Ici, la solution la plus simple et peut-être la plus élégante est l'utilisation du sychroniseur *barrière de synchronisation* :
  - elle est créée puis liée à n processus devant se synchroniser ;
  - chaque processus appelle une méthode `wait` sur la barrière ;
  - les processus ayant appelé `wait` attendent jusqu'à ce que tous les autres l'aient fait et alors tous sont libérés et poursuivent leur exécution.
- Plus généralement, problématique de génie logiciel complexe : comment programmer et faire apparaître dans des interfaces les synchronisations et la coordination nécessaires entre actionneurs d'entités adaptables ?

Interruptibilité, traitement des erreurs et transactions

- Interruption, traitement d'erreurs et transactions ajoutent de la robustesse aux opérations d'adaptation.
- En complément de l'identification des sections critiques des opérations d'adaptation, une opération robuste devrait donc :
  - limiter la période de non-interruptibilité au *franchissement coordonné* des sections critiques de toutes les opérations d'adaptation composées ;
  - assurer que toute interruption mène à un abandon complet de l'opération et ramène toutes les entités dans un état prêt à fonctionner :
    - traiter toute exception levée de manière à ramener le(s) entité(s) touchée(s) dans un état cohérent, prête(s) à fonctionner ;
    - introduire des mécanismes *transactionnels* globaux des opérations d'adaptation (au-delà des exceptions individuelles), par exemple le retour arrière (*rollback*) pour revenir à l'état initial, avant le lancement de l'adaptation.

Méta-propriétés des actionneurs (à documenter)

1. Effets sur les propriétés adaptables : propriétés affectées, inertie, y compris de manière croisée (entre actionneurs).
2. Assertions :
   - Pré-conditions : conditions sur les paramètres, sur l'état de l'entité et les autres opérations à appeler avant.
   - Post-conditions : conditions sur les résultats, sur l'état de l'entité après l'exécution de l'opération.
   - Invariants : conditions maintenues sur l'état de l'entité.
3. Type et atomicité : élémentaire versus composite, exécutabilité comme tout ou partie d'une transaction d'adaptation.
4. Supervisabilité : moyens offerts pour suivre le progrès et intervenir pour les synchroniser et les coordonner.
5. Interruptibilité : possibilité ou non d'interrompre l'exécution de l'opération d'adaptation.
6. Propriétés temporelles : durée, période minimale entre les appels, inertie (délai entre l'action et son effet).
7. Coûts en ressources : de toutes natures.

Plan

Problématique du test des CPCS

- Outre les erreurs de programmation classiques, il faut s'assurer que le code fournit les *bonnes réactions* aux actionneurs *selon les données courantes* des capteurs et ce dans la *bonne temporisation* pour produire une *bonne trajectoire*.
- Idéalement, les tests se feraient donc toujours en contexte de déploiement, mais cela n'est que partiellement possible, pour des raisons de disponibilité, de coûts ou de sécurité.
- Dans une approche de tests dirigés par les modèles, tester un CPCS *hors déploiement* requiert d'avoir des modèles préalablement validés servant d'oracles pour :
  1. Fournir des entrées *réalistes*, remplaçant les données capteurs.
  2. S'assurer que chacune des sorties sur les actionneurs est *correcte* compte tenu des entrées.
  3. S'assurer que la *trajectoire* des sorties et de l'état (au sens de la théorie des systèmes et du contrôle) sont conformes.

Comment vérifier et valider les CPCS ?

- Au-delà de la mise au point du code, les CPCS doivent aussi être vérifiés et validés, ce qui exige une approche méthodique et complète.
- Comme pour tout système, la validation peut se faire par des tests, unitaires et d'intégration, permettant de s'assurer de leur bon fonctionnement et du respect de leurs méta-propriétés.
- Outre les difficultés rencontrées en test de mise au point, il faut assurer une *couverture complète* des scénarios possibles.
- Pour cela, il faut
  - des *expérimentations contrôlées*
  - faisant varier *systématiquement* les configurations initiales
  - et évoluer les valeurs des variables non-fonctionnelles
  - sur *l'entièreté de leurs domaines*,

  ce qui est très difficile et souvent très coûteux à faire sur des *systèmes réels déployés*.

Tests dirigés par les modèles en simulation

1. Les approches classiques fondées sur les cas de tests par entrées/sorties ponctuelles sont insuffisantes pour les CPCS.
   1. Il demeure possible de générer des ensembles de cas de tests *i.e.*, des données de capteurs à un instant précis, puis vérifier *à chaque fois* que les sorties sur les actionneurs sont correctes.
   2. Toutefois, cette approche ne va pas permettre de vérifier et valider que l'*enchaînement* bien *temporisé* des lectures de capteurs et des sorties appliquées aux actionneurs vont produire une trajectoire correcte.
2. Pour compléter les approches classiques, il faut être en mesure de produire et exécuter des enchaînements de lectures capteurs influencées par les réactions actionneurs puis s'assurer qu'elles soient fidèles à l'évolution attendue du monde physique.
3. À défaut de pouvoir s'en assurer dans le monde physique, l'approche proposée se fonde sur la *simulation*, par transformation des modèles de comportement en *modèles de simulation*.

Concept de co-simulabilité

- Question : comment produire des scénarios de test réalistes du point de vue du modèle de comportement ?
  - Faire calculer par simulation les trajectoires des données d'entrées et de sortie produites.
  - Connecter cette simulation avec le code pour ainsi fournir des enchaînements de données réalistes, comme si l'action des sorties calculées se produisaient dans le monde physique.
- *Cette idée peut être mise en œuvre par la co-simulation.*
  - La co-simulation consiste à simuler conjointement et faire interagir plusieurs phénomènes parallèles ; dans le cas présent il s'agit
    1. de simuler la plate-forme matérielle et l'environnement physique en interaction avec le logiciel (simulation SIL) puis
    2. de simuler uniquement l'environnement physique en interaction avec le logiciel déployé sur une plate-forme matérielle cible (simulation HIL).
  - En robotique, par exemple, tout projet sérieux commence ses tests par une co-simulation utilisant un *moteur physique* où le robot et l'environnement sont simulés (outils : Gazebo, Morse, ...).

Simulation et architectures logicielles CPS : objectifs

- La simulation paraît donc 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 :
  - d'implanter les composants individuellement ;
  - d'implanter des simulateurs SIL modulaires au sein de chaque composant puis connecter exécution du logiciel et simulation ;
  - de construire l'*architecture globale* de l'application en *assemblant* ses composants ;
  - de construire le *simulateur global* de l'application en *composant* les simulateurs modulaires individuels.

  Les deux dernières phases étant intégrées dans une nouvelle forme d'assemblage spécifique aux composants cyber-physiques.

Principe de co-simulabilité des entités adaptables

Principe de co-simulabilité

Toute entité adaptable doit être conçue et mise en œuvre de manière à pouvoir être co-simulée, d'abord pour la mettre au point puis pour la vérifier et la valider.

- L'objectif principal de cette co-simulabilité est de produire lors d'une exécution des valeurs de propriétés non-fonctionnelles correctes, cohérentes entre elles et cohérentes dans le temps.
- Un objectif qui peut aussi être très important dans certains cas est la simulation de certains aspects fonctionnels pour simplifier le déploiement des tests unitaires.
  Ex.: simuler sur un ordinateur unique une application qui devrait être déployée en réparti.
- À titre de preuve de concept, nous allons examiner une intégration BCM4Java/DEVS.

Plan

Principes d'intégration de DEVS4Java/BCM

- Proposition d'une *unité de composition autonome*, le *composant cyber-physique*, soutenant un génie logiciel fondé sur leur co-simulabilité, et pour cela intégrant :
  - un modèle de comportement (automate hybride) *modulaire* et *déclaratif* (spécification, conception, etc.) et
  - un modèle de simulation *modulaire* et *exécutable* (mise au point, test, V&V, etc.),

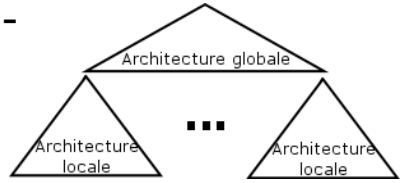
  puis faire en sorte que l'assemblage de composants cyber-physiques intègre ces modèles.
- Cette intégration pose toutefois quelques contraintes :
  - Les simulateurs DEVS *internes* aux composants devront être composés par des modèles couplés *inter-composants*.
    - Cela implique une architecture de simulation explicitant ses liens avec les composants *i.e.*, description du déploiement des modèles sur et entre les composants.
    - Cela implique également des échanges entre simulateurs passant potentiellement par des connexions entre composants répartis, d'où une pénalité potentielle en performance à prendre en compte.

Contraintes d'exécution de simulations sur des composants

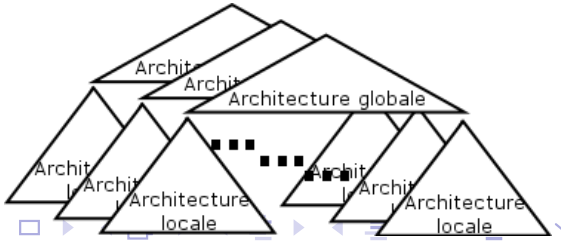
- Les HIOA sont composables par partage de variables continues et d'événements discrets or les modèles DEVS, par essence discrets, se composent uniquement par échange d'événements.
  - En DEVS, un phénomène continu doit être discrétisé à grain suffisamment fin pour assurer des calculs précis (ex.: intégration numérique par pas discrets).
  - De petits pas impliquent des échanges d'événements échantillonnant à haute fréquence les valeurs de variables partagées.
- Même à faible fréquence, les délais de communication entre composants répartis seraient très (trop) longs par rapport aux calculs effectués localement.
- D'où deux règles directrices :
  1. Que l'exécution de compositions de modèles DEVS issus d'HIOA partageant des variables continues soit *confinée à l'intérieur des composants*.
  2. Conséquemment, seule l'exécution de *compositions entre TIOA* puissent se faire *entre différents composants*.

Architectures DEVS sous BCM4Java

- Architecture en deux *niveaux principaux* :
  - une partie « basse » *décrites dans chaque composant* de base ⇒ architectures locales ;
  - une partie « haute » formée de modèles couplés composant les précédentes ⇒ architecture globale ;
  - Parties haute et basses sont décrites *séparément*, la partie haute voyant chaque partie basse composite comme un *modèle atomique*.



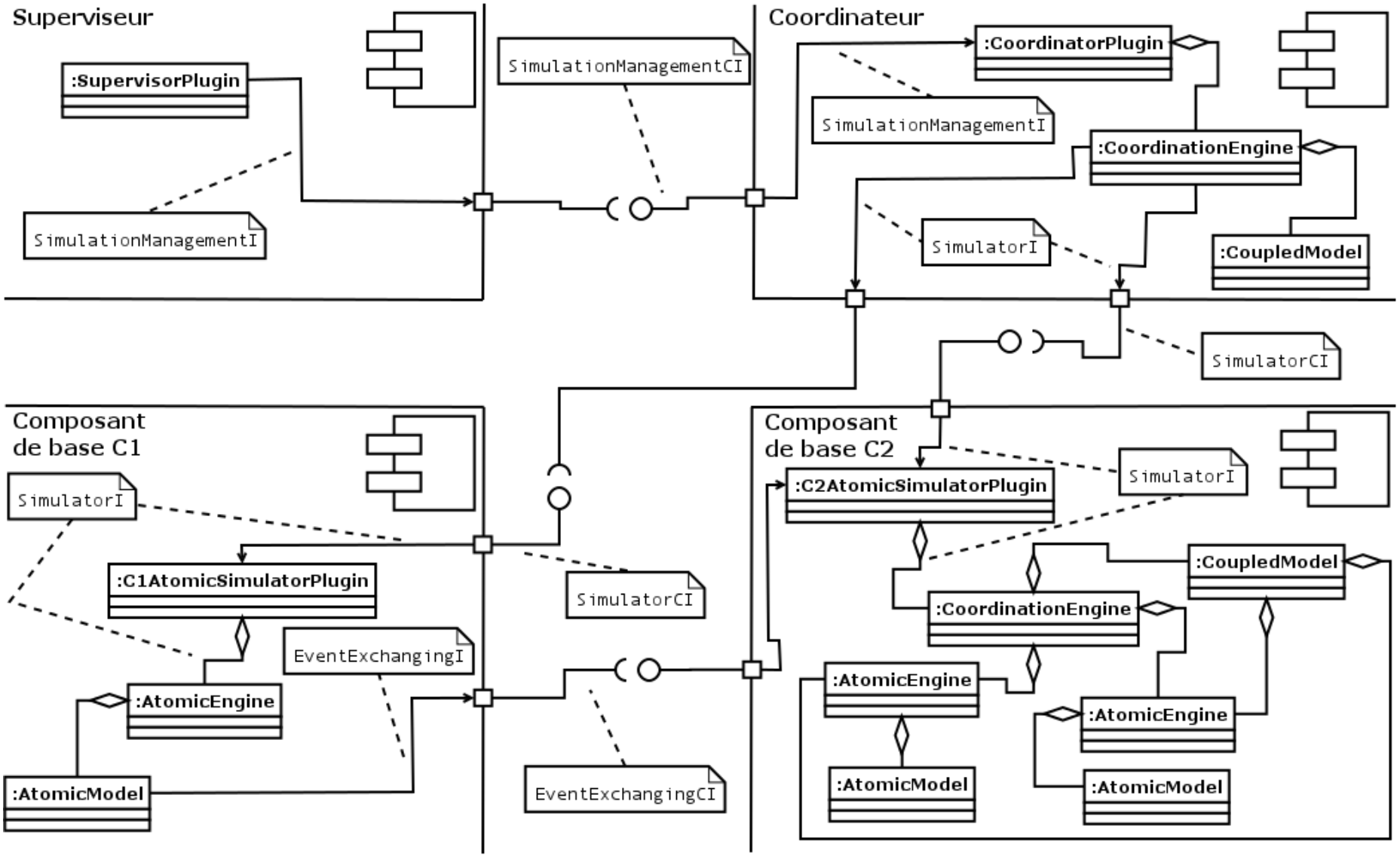
- Pour respecter l'abstraction du composant, tous les modèles de simulation d'un composant sont déployés sur ce composant.
  - Mais dans certains cas, on peut déroger pour exécuter certains modèles DEVS/HIOA sur des composants tiers *quand leur composition exige l'échange de variables continues*.
- Parce qu'il peut devenir utile d'utiliser différents simulateurs pour différents objectifs, il doit être possible de gérer plusieurs architectures globales et locales.



Exécution de simulations DEVS sous BCM4Java

- Comment définir et exécuter une architecture DEVS dans BCM4Java ?
  1. Attacher des modèles DEVS aux composants, ce qui se fait par l'intermédiaire de greffons (*plug-ins*) réutilisables.
  2. Interconnecter des modèles en passant par des connexions inter-composants *i.e.*, ports et connecteurs BCM4Java.
  3. Gérer la création de simulateurs globaux puis l'exécution et la gestion de campagnes de simulations par des greffons de supervision qui vont abstraire l'utilisateur de ces considérations.
- On peut distringuer trois rôles à jouer :
  1. Simulateur « unitaire » : associé à chaque composant de base.
  2. Coordonnateur : exécutant un modèle couplé de l'architecture globale (partie haute).
  3. Superviseur : assure
     - la création et l'interconnexion de l'architecture globale et
     - la gestion des campagnes de simulation (lancement, récupération puis analyse des résultats).

Schéma d'inter-connexion (partiel) DEVS via BCM4Java



Les greffons d'intégration DEVS/BCM 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`.
- Ils doivent aussi créer l'architecture locale et interagir pour créer la partie de l'architecture globale.
  - 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 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 (`EventsExchangingCI` et `ParentNotificationCI`), ce qui est aussi géré par les greffons.

Point de vue utilisateur : architectures locales

1. Programmer les modèles de simulation MIL, SIL (voire HIL) locaux, avec la bibliothèque DEVS en Java.
2. Créer une architecture locale, l'attacher au composant par un greffon de type simulation de modèle atomique, pour 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.
   - On masque l'architecture de simulation locale au sein du composant en création les modèles/moteurs lors de la création du composant *i.e.*, lors de la création et de l'installation de son greffon.

     Ex.: pour le composant `WiFi` de MolèNE, 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 de simulation combinées DEVS/BCM4Java

- Pour décrire un modèle dans l'architecture globale, on ajoute à son descripteur l'URI du port de reflexion entrant du composant qui le possède et où on va devoir le créer.
  - Les composants qui possèdent une architecture locale sont décrits dans l'architecture globale comme des modèles atomiques, masquant ainsi les détails de cette architecture locale.
  - Les composants coordinateurs, qui vont détenir des modèles couplés, n'ont pas à créer explicitement de greffon lors de leur création ; ils vont être créés lors du processus d'instantiation de l'architecture globale.
- Du point de vue de l'utilisateur, peu de changements par rapport au *framework* DEVS en Java ; ce sont les descripteurs spécifiques à l'intégration et les greffons qui implantent l'algorithme d'instantiation croisée DEVS/BCM4Java.
- L'ensemble des ajouts aux composants, des greffons, des ports et connecteurs spécifiques à cette intégration forment une extension de BCM4Java appelée BCM4Java-CyPhy.

Exemple WiFi I

- Dans l'exemple MolèNE, le modèle est détenu par un composant `WiFi` et son descripteur vu comme un modèle atomique 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
        atomicModelDescriptors,     // 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 crée l'architecture de simulation globale, instancie le greffon superviseur en lui passant cette architecture en paramètre„ ce dernier crée le simulateur global en appelant la méthode `constructSimulator` puis lance la simulation en appelant ici la méthode `startRTSimulation`. Exemple : en supposant que la variable `sp` contient la référence au greffon de supervision

```
sp.createSimulator();
this.sp.startRTSimulation(System.currentTimeMillis() + 1000, 0.0, 5000.0);
```

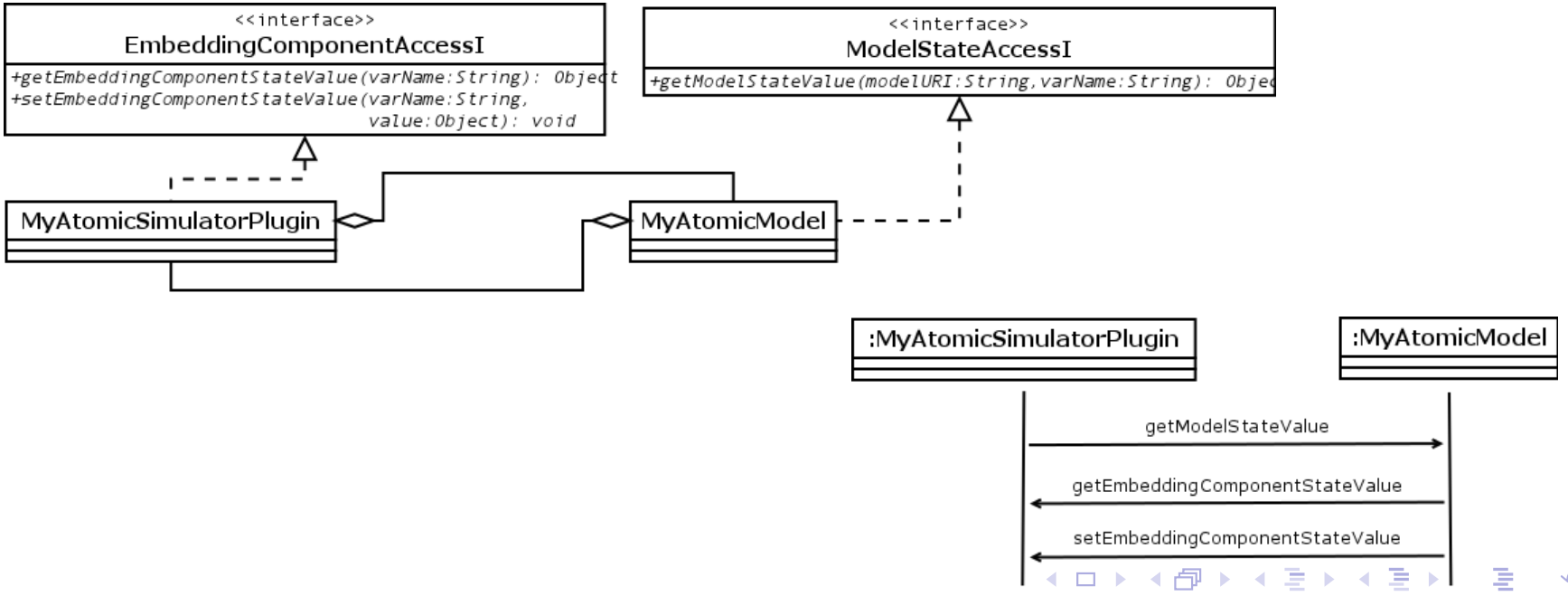
- Avec la possibilité d'utiliser comme dans le *framework* DEVS le protocole de passage de paramètres d'exécution par la méthode `setSimulationRunParameters`.
- *Nota :* la méthode `startRTSimulation` réalise une simulation *en temps réel* permettant des échanges avec le logiciel selon une même référence temporelle. Nous y revenons. . .

Embryon de composant cyber-physique : BCM4Java-CyPhy

- BCM4Java-CyPhy offre une classe abstraite `AbstractCyPhyComponent` dont doivent hériter les classes définissant les composants cyber-physiques qui définit :
  - une table de hachage liant l'URI du modèle racine de chaque architecture de simulation locale définie par le composant ;
  - deux tables de hachage liant les URIs des architectures globales de simulation aux URIs des modèles racines des architectures locales et l'inverse ;
  - une extension aux interfaces de réflexion pour les composants cyber-physiques qui offre des méthodes permettant de récupérer les URIs de certains ports spécifiques à la simulation DEVS pour faciliter la composition des modèles de simulation entre compsoants.
- D'autres éléments devront être ajoutées pour améliorer l'intégration BCM4Java/DEVS, définir les activités de déploiement (voir ci-après), utiliser des interfaces riches, spécifier les propriétés d'adaptation, etc.

Échanges de données composants/modèles

- Dans les simulations SIL, il est généralement nécessaire d'être capable de communiquer entre composants hôtes 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.
- Protocole des interfaces `ModelStateAccessI` et `EmbeddingComponentStateAccessI` :

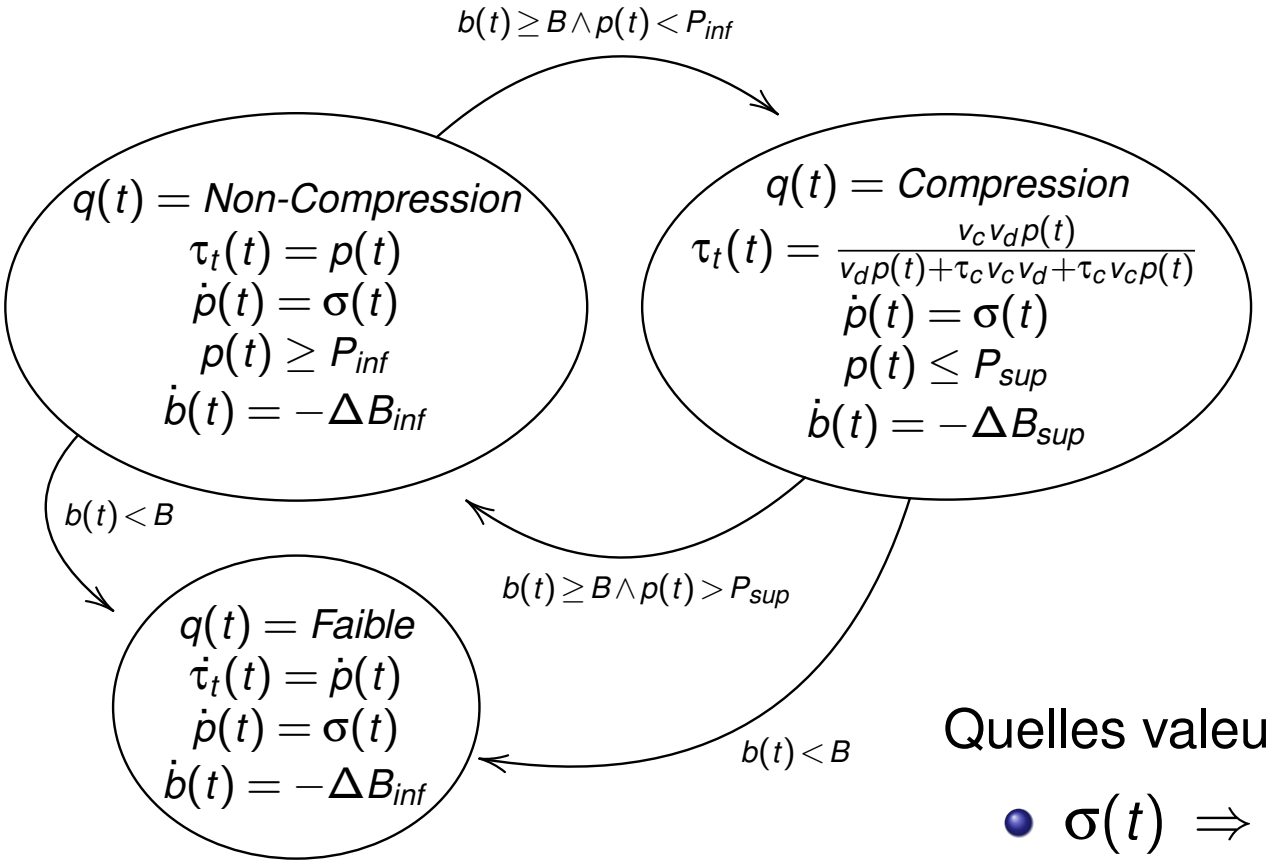


Déclenchement d'exécution de code composants/modèles

- Modèle vers composant :
  - Un modèle peut récupérer la référence sur son composant hôte grâce au protocole `setSimulationRunParameters` puis appeler ses méthodes ; le greffon de simulation atomique ajoute cette référence automatiquement aux paramètres.
  - Attention à respecter les exclusions mutuelles du composant hôte car les activités de simulation sont ordonnancées sur des *threads* distincts de ceux utilisés par ce dernier.
- Composant vers modèle :
  - Cohérent seulement si simulation temps réel.
  - Le greffon `RTAtomicSimulatorPlugin` servant à gérer les simulations temps réel offre la méthode : `triggerExternalEvent(String destinationModelURI, EventFactoryFI ef)`
    - `destinationModelURI` : URI d'un modèle receveur
    - `ef` : λ prenant en paramètre le temps simulé courant pour créer un événement devant être reçu et exécuté à cet instant.

Plan

Problématique — retour sur notre exemple MolèNE



Quelles valeurs pour :

- $\sigma(t) \Rightarrow$ mesures de terrain
- $P_{sup}, P_{inf}, B \Rightarrow$ choix
- $v_c, v_d, \tau_c, \Delta B_{inf}, \Delta B_{sup}$? ? ?
- durée des adaptations ? ? ?

Contrôle et identification du système

- Dans les CPS, la plupart des paramètres définissant les modèles de comportement dépendent du contexte de déploiement : performance du matériel, outils logiciels (ex.: quels outils de compression ?), etc.
- Pour les estimer de manière suffisamment précise pour établir un bon modèle de contrôle, il faut connaître ce(s) contexte(s). Deux options sont alors possibles :
  1. identifier au développement les contextes possibles et estimer ces paramètres à l'avance sur ces contextes pour les spécifier comme paramètres de configuration au préalable, ou
  2. attendre au déploiement et les mesurer *in situ* avant de commencer l'exécution ou pendant l'exécution pour (re)configurer au déploiement ou dynamiquement.
- Dans tous les cas, pour estimer ces paramètres, il faudra instrumenter l'entité adaptable pour ce faire.

Limites de la modélisation *amont* pour l'adaptabilité

- Le génie logiciel classique prévoit de modéliser en amont du déploiement pour des développements produisant du logiciel déployable sur des plates-formes très variées.
- Mais à cette étape, on ne peut établir qu'un modèle comportemental *générique*, puisque ses paramètres précis dépendent du contexte de déploiement logiciel et matériel (p.e., performance).
  Ex. : $p(k+1) = ap(k) + bu(k)$, *Cours 6.*
- Ce modèle générique doit ensuite être *identifié* pour donner un modèle *concret*, c'est-à-dire que des valeurs précises pour tous les paramètres, coefficients, etc. sont déterminées à partir de mesures faites *dans le contexte de déploiement*.
  Ex. : $p(k+1) = 0,43p(k) + 0,47u(k)$, *Cours 6.*
- Enfin, pour procéder aux tests unitaires et d'intégration, puis valider le système, il faudra *simuler* tout ou partie des modèles fonctionnels et comportementaux.

Auto-caractérisation et auto-configuration

- Automatiser ces fonctionnalités :
  - Proposer des procédures prédéfinies permettant d'estimer les caractéristiques de l'entité, de ses mesures, de ses opérations d'adaptation et de l'impact des opérations d'adaptation sur les propriétés adaptables.
  - Proposer une interface pour récupérer ces données, qui pourra être utilisées par l'entité de contrôle de manière à configurer son propre modèle et sa loi de commande.
- En réalité, cette idée poursuit le même but que certaines approches de contrôle adaptatif ou d'apprentissage automatique : estimer les paramètres du modèle de comportement de l'entité adaptable pour obtenir un modèle de contrôle correct et efficace.
- Un lien intéressant peut aussi être fait avec l'approche *BITE* pour *Built-In Test Equipment* de plus en plus utilisée dans l'industrie pour superviser les appareils et détecter les anomalies en cours de fonctionnement.

Récapitulons...

1. La capacité d'adaptation exigée dans les application réelles dépasse la simple juxtaposition d'opérations d'adaptation : il faut en ***superviser l'exécution***, les organiser comme ***transactions***, et les ***coordonner*** avec d'autres adaptations.
2. Pour être réellement contrôlable, une entité adaptable doit savoir ***caractériser de la manière la plus précise possible les effets*** des adaptations sur les mesures des capteurs pour appliquer la bonne loi de contrôle et les exprimer comme méta-propriétés.
3. Concevoir une entité adaptable dans un contexte **ouvert** est **difficile**, et ne peut être réalisé sans une **explicitation** complète de ses propriétés.
4. La validation par le test pouvant rarement se faire systématiquement dans tous les contextes de déploiement, la **simulation** est un outil essentiel à la mise au point des entités adaptables ce qui mène au concept de **composants simulables** comme pierre angulaire des systèmes cyber-physiques et autonomiques et de leur méthodologie de développement.
5. La plupart des propriétés non-fonctionnelles des entités adaptables ne peuvent être **quantifiées** que dans le contexte de déploiement ; il est donc très utile de disposer de moyens **automatiques** pour **identifier** le modèle de comportement de l'entité au moment de son **déploiement**.

Pour aller plus loin : sélection de lectures recommandées

- *Autonomic Computing*, P. Lalanda, J. McCann et A. Diaconescu, Springer-Verlag, 2013, chapitres 5 et 6.
- X. Dutreilh *et al.*, *From Data Center Resource Allocation to Control Theory and Back*, CLOUD 2010, juillet 2010.
- *Theory of Modeling and Simulation*, B.P. Zeigler et al., 2e édition, Academic Press, 2000.