Ingénierie des Modèles

Exécution de modèles

Eric Cariou

Master TIIL-A 2ème année

Université de Bretagne Occidentale
UFR Sciences & Techniques – Département Informatique

Eric.Cariou@univ-pau.fr

Modèles à l'exécution

- L'IDM généralise l'usage des modèles partout où on peut le faire
  - Usage productif des modèles
  - Y compris dans le système en cours d'exécution
- Usage « passif »
  - Un modèle est utilisé pour représenter l'architecture du système, pour prendre des décisions...
  - Ex : *models@run.time*
    - Discipline où un modèle représente l'état courant du système
      - Lien de causalité entre le système et le modèle
    - Analyse du modèle pour détecter d'éventuels problèmes
    - En cas de problème, on adapte le système en cours d'exécution

Modèles à l'exécution

- Usage « actif »
  - Le modèle définit le (ou une partie du) comportement du système
  - Ex : une machine à états qui contrôle le fonctionnement d'un ascenseur
    - En fonction de l'interaction de l'utilisateur avec les boutons de contrôle de l'ascenseur
      - Les transitions entre les états se font via les événements générés par l'utilisateur
      - Les états définissent les opérations métier à exécuter
        - Ouverture/fermeture de portes
        - Monter/descendre à un certain étage
  - Exécution de modèle
    - Le système prend en entrée un modèle qui définit le comportement du système et l'interprète

Modèles exécutables

- Deux principales variantes de l'exécution de modèle
  - Compilation de modèle
    - On traduit le contenu du modèle vers du code classique via éventuellement des frameworks dédiés
      - On a vu la génération de code Java à partir de diagrammes de classes UML
      - Mais il manque le contenu des méthodes
        - Elles peuvent être spécifiées en fUML
        - Code abstrait que l'on peut définir sur des modèles UML
  - Simulation de modèle
    - En phase de conception, on simule l'exécution du modèle
    - Permet de détecter au plus tôt des erreurs de conception
    - Le modèle simulé n'est pas (forcément) ensuite présent en tant que tel à l'exécution

Modèles exécutables

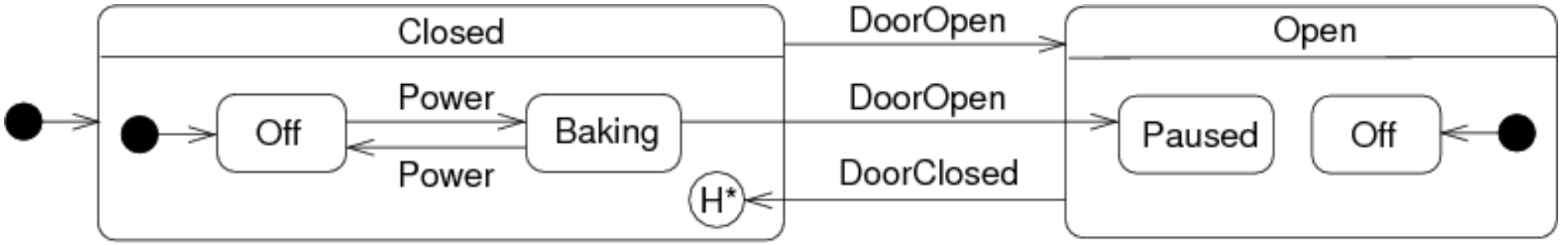
- Dans UML
  - Les diagrammes comportementaux sont a priori exécutables
    - Machines à états, diagrammes de séquence, d'activité …
  - Les diagrammes structurels ne le sont pas (par défaut)
    - Diagramme de classe, de composant, d'objet …
- En IDM, peut créer ses propres langages de modélisation
  - DS(M)L : Domain Specific (Modeling) Language
  - Si on crée des langages de modèles exécutables
    - i-DSML (interpreted DSML) ou xDSL (executable DSL)

Constituants d'un xDSL

- Éléments principaux d'un xDSL
  - Méta-modèle définissant les éléments qu'on trouvera dans le modèle
    - Partie statique
      - Pour définir le contenu métier du modèle
      - Méta-modèle « classique »
    - Partie dynamique
      - Pour définir l'état dans lequel se trouve le modèle au cours de son exécution
      - Spécifique aux xDSL
  - Sémantique d'exécution
    - Définit comment le modèle évolue au fil du temps
    - Chaque évolution = un pas d'exécution
  - Moteur d'exécution
    - Implémente la sémantique d'exécution
    - Prend en entrée un modèle et l'exécute

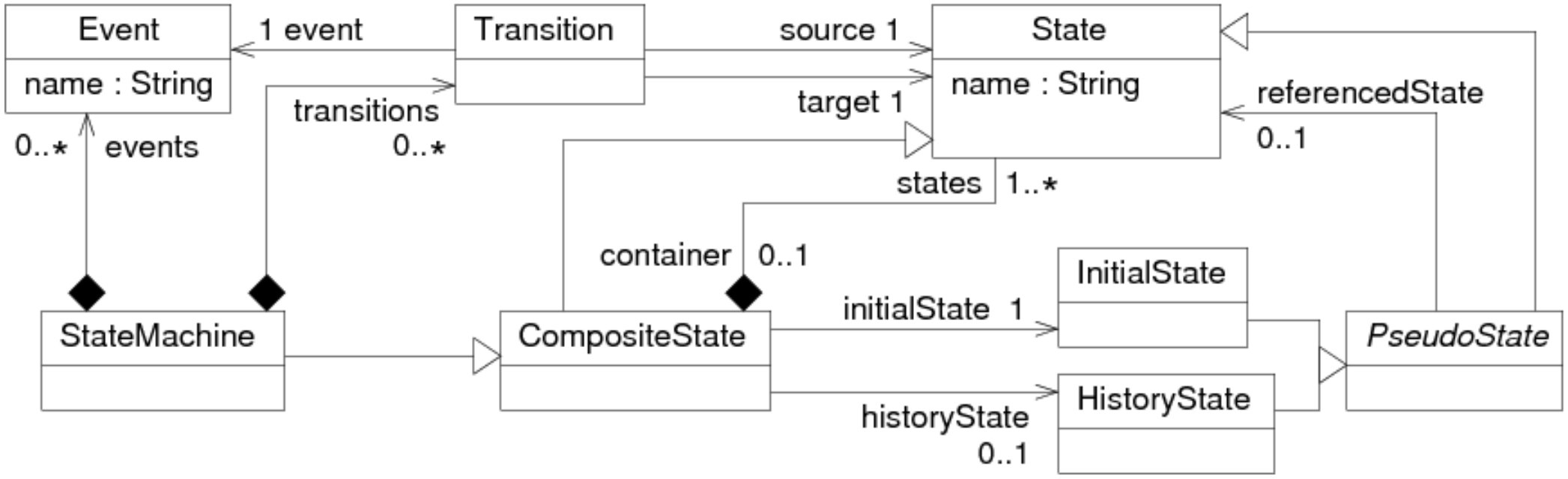
Exemple de xDSL

- Machines à états simplifiées
  - États composites
  - État historique
- Exemple de modèle
  - Four à micro-onde
  - Événements
    - « Power » : bouton de marche/arrêt de cuisson
    - « DoorOpen » : ouverture de la porte
    - « DoorClosed » : fermeture de la porte



Partie statique

- Définit tous les éléments structurels « classiques »
  - État, composite, transition, état initial, historique, transition, événement …
  - La machine à état est l'élément racine du méta-modèle
    - Instance unique dans le modèle
    - Est définie comme un composite pour définir son ensemble d'états
    - Contient en plus les événements et transitions de la machine à états



Partie statique

- Invariants OCL pour compléter le méta-modèle
  - Une machine à états est le seul état sans container

```
context StateMachine inv noContainerForStatemachine:
self.container.oclIsUndefined()

context State inv containerForAllStates:
not self.oclIsTypeOf(StateMachine) implies
not self.container.oclIsUndefined()
```

  - Un état initial référence forcément un état

```
context InitialState inv initialStateNeverEmpty:
not self.referencedState.oclIsUndefined();
```

  - Les pseudos-états référencent un état de leur composite

```
context CompositeState inv pseudoStatesInComposite:
self.states->includes(self.initialState.referencedState) and
self.states->includes(self.initialState) and
not self.historyState.oclIsUndefined() implies (
   self.states->includes(self.historyState) and
   not self.historyState.referencedState.oclIsUndefined()
   implies self.states->includes(self.historyState.referencedState) )
```

Partie statique

- Invariants OCL sur les transitions
  - Une seule transition partant du même état avec le même événement

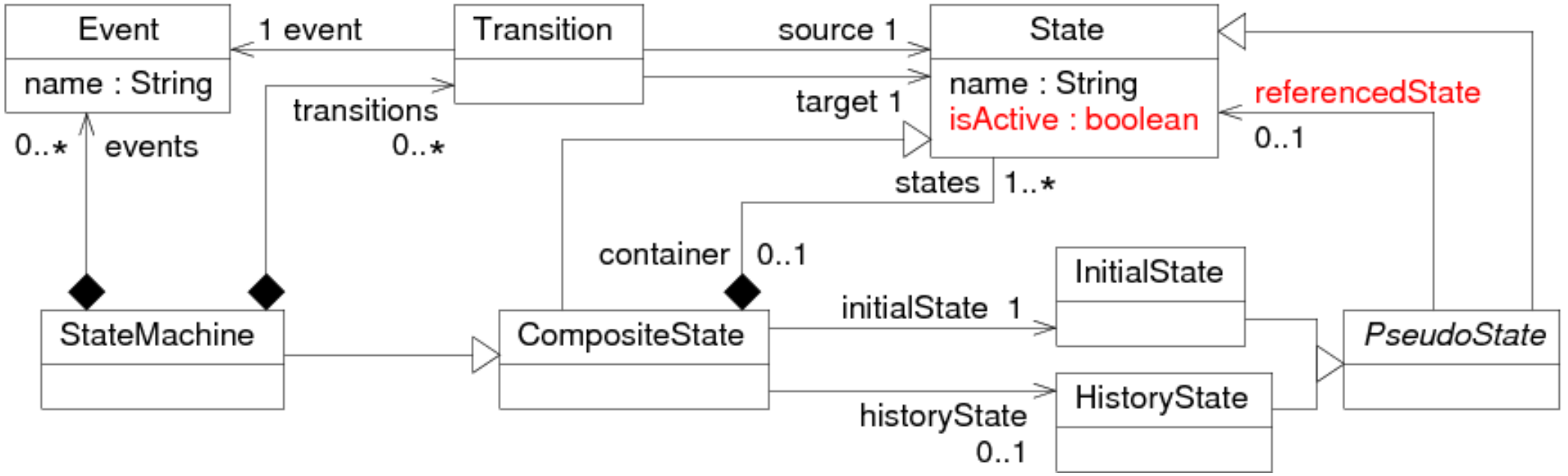
```
context Transition inv transEventSource:
Transition.allInstances()->forAll(t : Transition |
    t.source = self.source and t.event = self.event implies self = t)
```

  - Pas de transition partant d'un état historique, pas de transition vers ou au départ d'un état initial ni d'une machine à états

```
context Transition inv transForbiddenStates:
not self.source.oclIsTypeOf(HistoryState) and
not self.source.oclIsTypeOf(InitialState) and
not self.target.oclIsTypeOf(InitialState) and
not self.source.oclIsTypeOf(StateMachine) and
not self.target.oclIsTypeOf(StateMachine)
```

Partie dynamique

- Définition de l'état courant du modèle en cours d'exécution
  - Quel(s) est/sont le(s) état(s) actifs
    - Rajoute un attribut booléen `isActive` dans `State`
  - Si un composite contient un état historique, il doit référencer le dernier état qui était actif
    - L'association `referencedState` sert à cela
      - Elle est statique pour un état initial (ne change pas)
      - Elle est dynamique pour un état historique (est modifiée pendant l'exécution)



Partie dynamique

- Invariants OCL pour compléter la partie dynamique
- Principalement pour assurer la cohérence des états actifs
  - Si la machine à états est inactive, alors tous les états du modèle sont inactifs
  - Si elle est active, alors un et un seul de ses états contenus (premier niveau) est actif
    - Si cet état est composite, alors un et un seul de ses états est actif
      - A vérifier récursivement
    - Un état composite inactif a tous ses états inactifs et cela récursivement jusqu'à tous les états feuilles
      - Valable aussi pour la machine à états

Partie dynamique

- Fonctions et attributs OCL qui vont aider à l'écriture des invariants sur la cohérence des états actifs
  - Deux pseudos-attributs pour récupérer les états normaux (exclusion des pseudo états) et les composites dans l'ensemble des états d'un composite

```
context CompositeState def: normalStates : Set(State) =
self.states->reject(s : State | s.oclIsKindOf(PseudoState))

context CompositeState def: compositeStates : Set(State) =
self.states->select(s : State | s.oclIsKindOf(CompositeState))
```

  - Un état composite ne contient aucun état actif (et récursivement)

```
context CompositeState def: unactiveSubTree() : Boolean =
self.normalStates->forAll(s : State | not s.isActive) and
self.compositeStates->forAll(s : State |
                s.oclAsType(CompositeState).unactiveSubTree())
```

  - Un état composite contient un et un seul état actif (et récursivement)

```
context CompositeState def: activeSubTree() : Boolean =
self.normalStates->select(s : State | s.isActive))->size() = 1 and
self.compositeStates->forAll(s : State |
    if s.isActive then s.oclAsType(CompositeState).activeSubTree()
    else s.oclAsType(CompositeState).unactiveSubTree()
    endif)
```

Partie dynamique

- Invariant de cohérence des états actifs de la machine à états en utilisant les fonctions précédentes
  - Soit toute la machine à états est désactivée
  - Soit elle est activée avec les mêmes règles qu'un composite
  - ```
    context StateMachine inv activeStateHierarchyConsistency:
    if self.isActive
    then self.activeSubTree()
    else self.unactiveSubTree()
    endif
    ```
    ```

Sémantique d'exécution

- Spécifie la façon dont le modèle évolue durant son exécution
  - Ex : pour les machines à états, précise comment suivre les transitions en fonction de l'occurrence d'événements et des états actifs
- Plusieurs manières de spécifier la sémantique pour une exécution de modèles
  - Sémantique axiomatique
  - Sémantique translationnelle
  - Sémantique opérationnelle
- Sémantique axiomatique
  - Logique de Hoare, contrats …
  - Définit des contraintes/propriétés qui doivent être respectées avant et après la réalisation d'un pas d'exécution
  - Ne permet pas d'implémenter un moteur d'exécution mais est utile si on veut vérifier que l'exécution se déroule correctement

Sémantique d'exécution

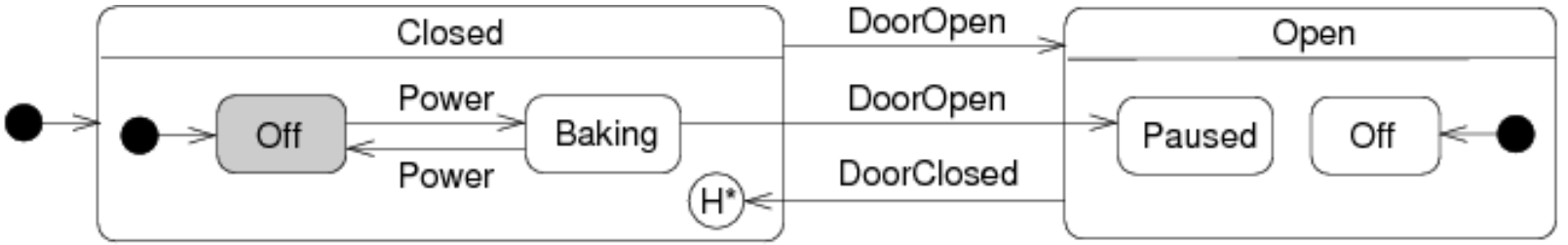
- Sémantique translationnelle
  - Traduit le modèle vers un autre espace technologique pour lequel il existe des outils d'exécution/simulation
    - Nécessite d'établir une correspondance sémantique entre les éléments du modèle et ceux de l'autre espace
- Différents usages
  - « compiler » un modèle
    - Ex : génération de code Java pour la libraire PauWare à partir du modèle d'une machine à états (http://www.pauware.com)
  - Faire de la simulation/vérification
    - Ex : traduction du modèle en un réseau de Petri ou génération de spécification formelle en B pour utiliser des outils de vérification dédiés

Sémantique d'exécution

- Sémantique opérationnelle
  - Définit de manière programmatique comment faire évoluer le modèle
  - Un moteur écrit dans un langage de programmation interprète le modèle et implémente cette sémantique
  - Exemples pour des xDSL définis en Ecore
    - Implémentation directe en Java/EMF
    - Implémentation via une transformation de modèle en ATL
      - Conceptuellement pertinent :
        un pas d'exécution = une transformation endogène
      - Par contre peu pratique à réaliser
        - Nécessite un moteur annexe qui lance la transformation quand par exemple un événement doit être traité par la machine à états
        - ATL est moins pratique pour coder des parties algorithmiques un peu complexes

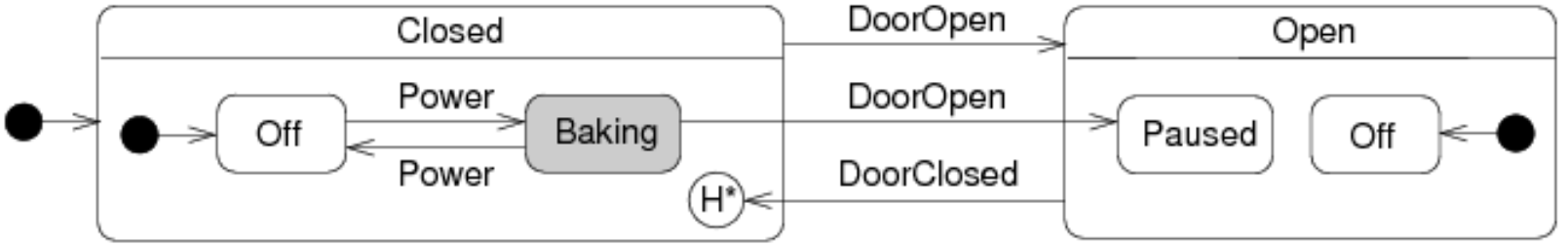
Ex. séquence d'exécution

Pas initial



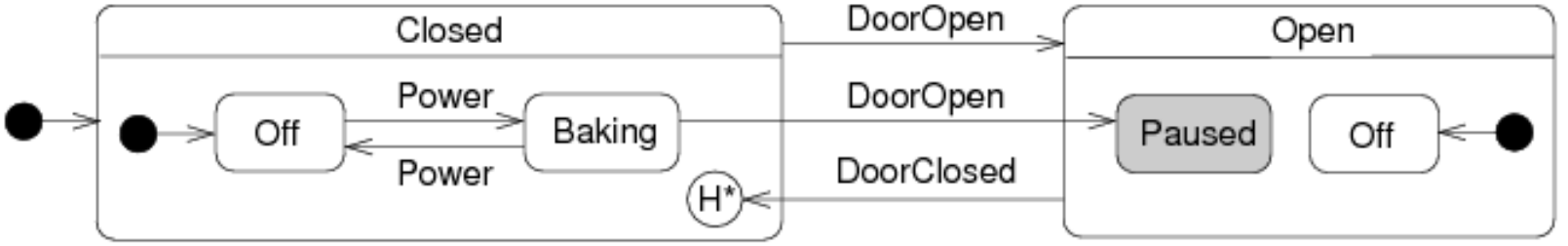
Evenement "Power"

Pas d'exec. 1



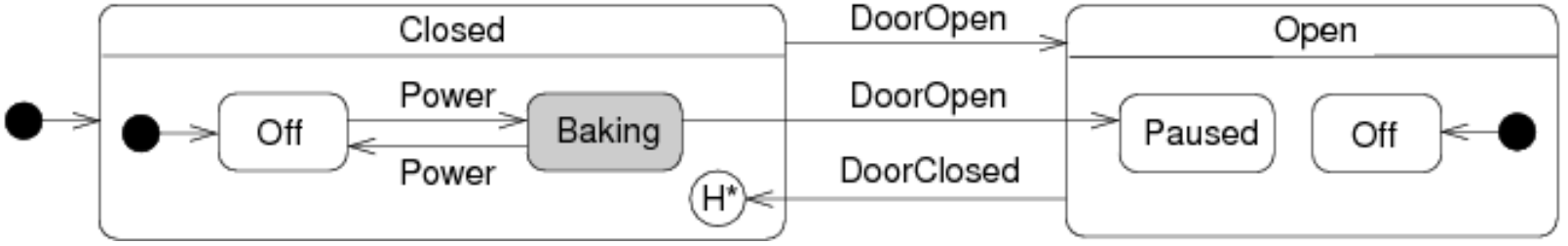
Evenement "DoorOpen"

Pas d'exec. 2



Evenement "DoorClosed"

Pas d'exec. 3



Ex. séquence d'exécution

- Pas initial
  - Active les états initiaux de la machine à états
  - Ici, l'instance de la machine à états, le composite « Closed » et l'état « Off » auront tous trois leur attribut `isActive` à vrai
- Pas d'exécution
  - Pour un événement, s'il existe une transition à suivre par rapport aux états actifs, change les états actifs
    - Réalise un pas d'exécution
    - Seule la partie dynamique du modèle est modifiée par l'exécution
      - Modifier la partie statique = rajouter/supprimer/modifier des états ou des transitions = modifier le contenu métier du modèle
- Entre pas 1 et 2, pour l'événement « DoorOpen »
  - Deux transitions éligibles : de « Baking » vers « Paused », de « Closed » vers « Open »
  - Sémantique UML : suit la plus interne (comme sur la figure)
  - Sémantique statecharts de Harel : suit celle entre les deux composites

Sémantique opérationnelle

- Pour notre méta-modèle de machines à états, il faut définir un ensemble de méthodes Java
  - Rendre toute une hiérarchie d'états inactive
  - Activer une hiérarchie d'états conformément aux règles décrites précédemment
    - En positionnant les états historiques des composites s'ils existent
  - Initialiser la machine à états : activer les états initiaux
  - Rechercher s'il existe une transition partant de l'état actif racine ou un de ses super-états pour un événement
  - Traiter l'occurrence d'un événement
    - Suivre la transition requise si elle existe en modifiant la hiérarchie des états actifs
    - Désactive la hiérarchie de l'état source et active celle du cible

Sém. op. : historiques et initialisation

```
// Positionne l'état en paramètre comme référencé par
// l'éventuel état historique que contient son composite
public void setAsHistory(State s) {
   if (s.getContainer() != null) {
      if (s.getContainer().getHistoryState() != null)
         s.getContainer().getHistoryState().setReferencedState(s);
   }
}

//Initialise la machine à états en activant tous les états initiaux
public void initStateMachine(StateMachine sm) {
   sm.setIsActive(true);
   State s = sm.getInitialState().getReferencedState();
   while (s != null) {
     s.setIsActive(true);
     this.setAsHistory(s);
     if (s instanceof CompositeState)
       s = ((CompositeState)s).getInitialState().getReferencedState();
     else s = null;
  }
}
```

Sém. op. : désactivation hiérarchie

```
// Désactive la hiérarchie haute d'un état
public void unactivateUpStateHierarchy(State s) {
   State up = s.getContainer();
   while (up != null) {
      up.setIsActive(false);
      up = up.getContainer();
   }
}

// Désactive la hiérarchie basse d'un état
public void unactivateDownStateHierarchy(State s) {
   if (s instanceof CompositeState)
      for (State down : ((CompositeState)s).getStates()) {
        down.setIsActive(false);
        if (down instanceof CompositeState)
            this.unactivateDownStateHierarchy(down);
   }
}

// Désactive un état et toute sa hiérarchie
public void unactivateStateHierarchy(State s) {
   s.setIsActive(false);
   this.unactivateUpStateHierarchy(s);
   this.unactivateDownStateHierarchy(s);
}
```

Sém. op. : activation hiérarchie

```
// Active la hiérarchie haute d'un état
public void activateUpStateHierarchy(State s) {
   State up = s.getContainer();
   while (up != null) {
      up.setIsActive(true);
      this.setAsHistory(up);
      up = up.getContainer();
   }
}

// Active la hiérarchie basse d'un état
public void activateDownStateHierarchy(State s) {
   if (s instanceof CompositeState) {
      State init = ((CompositeState)s).getInitialState().getReferencedState();
      init.setIsActive(true);
      this.setAsHistory(init);
      if (init instanceof CompositeState) this.activateDownStateHierarchy(init);
   }
}

// Active toute la hiérarchie d'un état
public void activateStateHierarchy(State s) {
   s.setIsActive(true);
   this.setAsHistory(s);
   this.activateUpStateHierarchy(s);
   this.activateDownStateHierarchy(s);
}
```

Sém. op. : recherche transition

```
// Retourne l'état actif le plus en bas de la hiérarchie
public State getLeafActiveState(CompositeState comp) {
   for (State s: comp.getStates())
     if (s.isIsActive())
        if (s instanceof CompositeState)
           return this.getLeafActiveState((CompositeState)s);
        else return s;
   return null;
}

// Retourne la transition partant de l'état actif le plus bas pour l'événement
// précisé ou null si aucune transition n'a été trouvée
public Transition getTriggerableTransition(String evt, StateMachine sm) {
   boolean fini = false;
   State activeState = this.getLeafActiveState(sm);
   Transition trans = null;
   while(!fini) {
      for (Transition t : sm.getTransitions())
         if (t.getEvent().getName().equals(evt) && (activeState == t.getSource())) {
            trans = t;
            fini = true;
         }
      if (!fini) {
         activeState = activeState.getContainer();
         if (activeState == null) fini = true;
      }
   }
   return trans;
}
```

Sém. op. : traitement d'un événement

```
// Traite un événement : recherche une transition à suivre puis si elle
// existe, désactive la hiérarchie de l'état source puis modifie
// la hiérarchie des états actifs à partir de la cible de la transition
// en prenant en compte le cas où la cible est un état historique
public void processEvent(String event, StateMachine sm) {
   Transition trans = this.getTriggerableTransition(event, sm);
   if (trans != null) {
      this.unactivateStateHierarchy(trans.getSource());
      State target = trans.getTarget();
      if (target instanceof HistoryState) {
         State histState = ((HistoryState)target).getReferencedState();
         if (histState == null)
            target =
                target.getContainer().getInitialState().getReferencedState();
         else target = histState;
      }
      this.activateStateHierarchy(target);
   }
}
```

Sémantique opérationnelle

- Moteur d'exécution en Java/EMF des machines à états
  - Exécute bien les modèles, fait évoluer les états actifs conformément à la sémantique d'exécution à chaque occurrence d'événement
    - Mais … c'est tout !
  - Il manque les opérations métiers associées aux états, les gardes aux transitions …
- Solutions
  - Rajouter au méta-modèle de quoi spécifier des opérations
    - Nécessite de la structure : variables, classes, types …
    - Nécessite un langage d'action : affectations, calculs, tests, boucles ...
    - En gros : modéliser un langage de programmation
  - Se contenter de représenter le comportement en pouvant rajouter le métier à coté
    - Les machines à états en PauWare exécutent des méthodes Java standard qui implémentent la partie métier

Exécution de diagrammes UML

- A la base, la spécification UML n'avait pas prévu d'exécuter les modèles comportementaux
  - Aucune partie dynamique pour aucun de ces diagrammes
    - Pas obligatoire mais dans ce cas c'est le moteur d'exécution qui gère lui même en interne l'état courant du modèle en cours d'exécution
  - Une sémantique d'exécution définie informellement et partiellement en anglais
- Aujourd'hui, plusieurs spécifications OMG pour faire de l'exécution de diagrammes UML
  - fUML : sémantique d'exécution de diagrammes d'activités
    - Peut définir le comportement exécutable d'une méthode d'une classe ou autre chose
    - ALF : syntaxe textuelle concrète de fUML similaire à celle d'un langage de programmation
  - PSCS : sémantique d'exécution des structures composites
  - PSSM : sémantique d'exécution des machines à états

PauWare

- PauWare : http://www.pauware.com
  - Librairie Java permettant de « programmer » des machines à états
  - Implémente la sémantique des machines à états UML 2.X
    - États imbriqués, concurrents
    - Opérations associées aux états (en entrée, sortie et son activité)
    - Gestion des gardes et des opérations des transitions
    - ...
- Equivalence entre machine à états UML et code Java PauWare
  - Le modèle (machine à état ici) est entièrement et totalement présent dans le code et est une partie du code
    - Le comportement dynamique est spécifié par la machine à état
    - La logique métier est implémentée dans les opérations associées aux états et aux transitions
  - L'exécution du programme consiste à exécuter la machine à état
    - Via le moteur PauWare

PauWare : micro-onde

- Implémentation en Java/PauWare de l'exemple du micro-onde
  - Ajout d'un objet métier gérant le micro-onde
    - Lumière : éteinte ou allumée
    - Magnétron : en marche ou arrêté
    - Porte : ouverte ou fermée
  - Ensemble d'opérations métier pour modifier l'état de ces éléments
  - Création ensuite du comportement avec l'API PauWare
    - La hiérarchie des états avec des opérations métier associées aux états
    - Des transitions entre les états avec là aussi des opérations métiers associées

Micro-onde : classe métier

```
public class MicrowaveBusiness {

    private boolean lightOn = false;
    private boolean doorOpen = false;
    private boolean magnetronOn = false;

    public void stop() {
        lightOn = false;
        magnetronOn = false;
    }

    public void heat() {
        lightOn = true;
        magnetronOn = true;
    }

    public void pause() {
        magnetronOn = false;
        lightOn = true;
    }

    public void openDoor() {
        doorOpen = true;
    }

    public void closeDoor() {
        doorOpen = false;
    }

    public String toString() {
        return "[ Light on: "+lightOn+", magnetron on: "+magnetronOn+", door open: "+doorOpen+ " ]";
    }
}
```

Micro-onde : machine à états

```
public class MicrowaveStateMachine {

    // les états de la machine à états
    protected AbstractStatechart open;
    protected AbstractStatechart closed;

    protected AbstractStatechart offOpen;
    protected AbstractStatechart offClosed;
    protected AbstractStatechart baking;
    protected AbstractStatechart paused;

    // la machine à états
    protected AbstractStatechart_monitor stateMachine;

    public void buildAndStartMicrowave(MicrowaveBusiness mwb)
                                                    throws Statechart_exception {

       // création des états simple avec associations des activités métiers
       // et précision si un état est un état initial de son composite
       offOpen = new Statechart("Off");
       offOpen.doActivity(mwb, "stop");
       offOpen.inputState();
       ...
```

Micro-onde : machine à états

```
...
offClosed = new Statechart("Off");
offClosed.doActivity(mwb, "stop");
offClosed.inputState();

baking = new Statechart("Baking");
baking.doActivity(mwb, "heat");

paused = new Statechart("Paused");
paused.doActivity(mwb, "pause");

// création des 2 états composites avec un history state pour Closed
closed = offClosed.xor(baking).name("Closed");
closed.deep_history();
closed.inputState();

open = offOpen.xor(paused).name("Open");

// création de la machine à états
stateMachine = new Statechart_monitor(closed.xor(open),"Microwave", true);
...
```

Micro-onde : machine à états

```
        ...
        // création des transitions entre états avec des opérations métier
        // pour gérer l'ouverture de la porte
        stateMachine.fires("DoorOpen", closed, open, true, mwb, "openDoor");
        stateMachine.fires("DoorOpen", baking, paused, true, mwb, "openDoor");
        // transition implicite vers l'état historique de Closed :
        stateMachine.fires("DoorClosed", open, closed, true, mwb, "closeDoor");
        stateMachine.fires("Power", offClosed, baking);
        stateMachine.fires("Power", baking, offClosed);

        // démarre la machine à états
        stateMachine.start();
    }

    public void stopMicrowave() throws Statechart_exception {
        stateMachine.stop();
    }

    public void runEvent(String name, MicrowaveBusiness mwb) throws Exception {
        // traite l'occurrence d'un événement en exécutant les transitions et
        // opérations requises
        stateMachine.run_to_completion(name);
        System.out.println("Etat métier après "+name+ " : "+mwb);
    }
}
```

Micro-onde : exécution

- Si on exécute ce code :

```
MicrowaveStateMachine sm = new MicrowaveStateMachine();
MicrowaveBusiness business = new MicrowaveBusiness();
sm.buildAndStartMicrowave(business);
sm.runEvent("Power", business);
sm.runEvent("DoorOpen", business);
sm.runEvent("Power", business);
sm.runEvent("Foo", business);
sm.runEvent("DoorClosed",business);
sm.runEvent("Power", business);
sm.stop();
```

- Donne la trace d'exécution suivante :

```
Etat métier après Power : [ Light on: true, magnetron on: true, door open: false ]

Etat métier après DoorOpen : [ Light on: true, magnetron on: false, door open: true ]

Etat métier après Power : [ Light on: true, magnetron on: false, door open: true ]

Etat métier après Foo : [ Light on: true, magnetron on: false, door open: true ]

Etat métier après DoorClosed : [ Light on: true, magnetron on: true, door open:
false ]

Etat métier après Power : [ Light on: false, magnetron on: false, door open: false ]
```

Résumé constituants d'un xDSL