**Ressource : éléments pour une programmation d’une machine à états SysML**

*Concernant les diagrammes états-transitions SysML, on se reportera à la ressource : éléments sur quelques diagrammes SysML (fichier Ressource\_Elements\_sur\_Diagrammes\_SysML).*

• Un **diagramme états-transitions (statechart)** décrit graphiquement une machine à états.

• Une **machine à états** ou **machine d’état** (ou **automate**) permet de modéliser le comportement d’un **système à événements discrets** (SED).

• Un tel **système** est dit **réactif** car il répond à des stimuli internes ou externes.

**Occurrence d’un événement**

• En SysML-UML, le comportement des machines à états est géré par les événements : seule **l’occurrence** (l’apparition) d’un **événement** peut déclencher une transition.

|  |  |
| --- | --- |
| • Un **événement** est un fait qui apparait à un instant donné. Il est supposé instantané.  • L’événement front montant de , noté , correspond au passage de la valeur 0 à la valeur 1 de la variable binaire . Pour une variable booléenne, le front montant correspond au passage de la valeur faux à la valeur vrai.  • Similairement, l’événement front descendant de , noté , correspond au passage de la valeur 1 à la valeur 0 de la variable binaire . Pour une variable booléenne, le front descendant correspond au passage de la valeur vrai à la valeur faux. | ,  : *notations hors SysML* |

**Détection et mémorisation de l’occurrence d’un événement par programme**

• Le caractère fugitif de l’occurrence d’un événement nécessite qu’elle soit détectée et mémorisée afin de pouvoir être traitée par un programme.

• Un événement correspondant au passage d’une valeur à une autre, son occurrence peut être détectée en observant la valeur juste un peu avant ce passage et juste un peu après. Par exemple, l’occurrence de l’événement front montant, noté , d’une variable binaire est détectable en observant le passage de sa valeur 0 à sa valeur 1, c'est-à-dire en observant sa valeur à l’instant précédent le front et sa valeur à l’instant suivant le front, résultat qu’il convient alors de mémoriser. Algorithme :

-------------------------------------------------------------------------------

**si** (a = 1) **et** (a\_precedent = 0) **alors**

front\_montant\_de\_a := **vrai**

// mémorisation d’une occurrence de l’événement front montant de a

**fin\_si**

a\_precedent := a

-------------------------------------------------------------------------------

• Remarques :

- Si une ou plusieurs variables contribuent au calcul des occurrences de plusieurs événements (ou des conditions de gardes), les valeurs de ces variables ne doivent pas changer pendant toute la durée du calcul pour éviter des aléas de fonctionnement. Ces variables sont dites figées.

- Un événement peut apparaitre suite à une évolution interne de la machine ou suite à une évolution d’un signal externe connecté à une entrée du système. Un signal externe peut être connecté :

🡪 soit à une entrée spéciale avertissant le système de l’occurrence d’un événement externe ; une telle entrée est appelée une entrée de demande d’interruption (Interrupt Request parfois abrégé en IRQ) ; la demande est réalisée matériellement ;

🡪 soit à une entrée standard que le système devra lire pour connaitre sa valeur ; la lecture se fait alors par scrutation (polling) de l’entrée à intervalles de temps réguliers ; c’est le programme qui lit l’entrée.

- Avec le codage ci-dessus, la mémorisation des occurrences nécessite autant de variables booléennes que d’événements. Alternativement, on peut utiliser une structure de données (liste par exemple). L’occurrence de l’événement est alors ajoutée (à la liste par exemple).

- L’espace de stockage des occurrences des événements est appelé le **« pool »** des événements.

**Effacement de l’occurrence d’un événement par programme**

• Une fois l’occurrence d’un événement prise en compte par le programme, il convient d’effacer sa mémorisation précédemment faite. Algorithme :

-------------------------------------------------------------------------------

front\_montant\_de\_a := **faux**

// effacement de l’occurrence de l’événement front montant de a

-------------------------------------------------------------------------------

• Remarque : pour une structure de données (liste par exemple), l’occurrence de l’événement est alors enlevée (à la liste par exemple).

**Simultanéités formelle et technologique des occurrences de deux événements**

• Deux événements sont corrélés s’ils dépendent d’une même variable ou de variables dépendantes.

Exemples : et sont corrélés, si et alors et sont corrélés.

• Formellement, les occurrences de deux événements non corrélés ne se produisent jamais simultanément.

Si a et b sont deux variables indépendantes alors

• En pratique, pour toute réalisation technologique, les occurrences de deux événements non corrélés peuvent être vues comme simultanées (s’il n’y a pas d’impossibilité physique) car leurs détections et leurs traitements ne peuvent pas être instantanés. Il est donc nécessaire d’envisager les simultanéités technologiques et de fixer un ordre de priorité des traitements consécutifs.

**Principes de base du traitement par une machine à états SysML**

• Principes de base :

* Initialisation :
  + La machine à états exécute sa transition (composée dans le cas général) initiale.
  + Puis elle entre en attente dans l’état initial (dans sa configuration d'état stable initiale dans le cas général).
* Durant toute la vie de la machine :
  + La machine mémorise les occurrences des événements (dans un « pool » - espace de stockage).
  + Etant dans un état actif stable (dans une configuration d'état actif stable dans le cas général), la machine choisit l’occurrence de l’événement à traiter (parmi celles mémorisées) selon un ordre de priorité défini. Alors :
    - La machine efface la mémorisation de l’occurrence de l’événement choisie (du « pool »).
    - La machine traite l’occurrence de l’événement choisie conformément aux règles d’évolution, entrant ainsi dans une phase transitoire d’évolution.
    - Puis la machine entre à nouveau en attente ; la machine a soit atteint un nouvel état actif stable (une configuration d'état actif stable dans le cas général), soit est restée dans l’état actif stable de départ (configuration d'état actif stable dans le cas général).

• Remarque :

- La machine à états traite une seule occurrence d'événement à la fois. Autrement dit, une occurrence d’événement ne peut être traitée que si le traitement de la précédente occurrence est terminé.

- Ce choix a été fait pour éviter les complications résultant de conflits de concurrence.

- Ce principe est appelé le paradigme comportemental de l'exécution jusqu’à la fin, ou jusqu’à l'achèvement (run-to-completion paradigm – RTC paradigm).

- En conséquence, la machine passe d’une configuration d’état (actif) stable à une autre, entre deux traitements d’occurrences d’événements. Pour mémoire, une machine à états est dans une configuration d’état (actif) stable si aucune transition n'est validée et qu’aucun comportement de sortie (exit/) ou d’entrée (entry/) n’est en cours. Les comportements durant les états (do/) peuvent se poursuivre.

**Architecture générale de l’algorithme**

• Ces principes permettent de donner l’architecture générale de l’algorithme pour une programmation d’une machine à états SysML.

-------------------------------------------------------------------------------

// déclarations et initialisations

…

// procédure appelée à chaque demande d’interruption accordée

// et de calcul des occurrences des événements liés

**procédure** interruption\_1()

**début**

…

// calcul et mémorisation des occurrences des événements

…

**fin**

// procédure d’acquisition des entrées

// et de calcul des occurrences des événements liés

**procédure** acquisition\_entrees()

**début**

…

// calcul et mémorisation des occurrences des événements

…

**fin**

// procédures de traitement des occurrences des événements

**procédure** traitement\_evenement\_1()

**début**

// effacement de la mémorisation de l’occurrence de l’événement

…

// traitement de l’occurrence de l’événement

…

**fin**

**procédure** traitement\_evenement\_2()

**début**

// effacement de la mémorisation de l’occurrence de l’événement

…

// traitement de l’occurrence de l’événement

…

**fin**

// transition initiale de la machine

…

// programme principal

**itérer**

// acquisition des entrées (polling) et calcul des occurrences des événements

acquisition\_entrees()

// choix de l’occurrence de l’événement compte tenu de l’ordre de priorité

// et appel de la procédure de traitement correspondante

**si** (occurrence de evenement\_1 mémorisée) **alors** // événement le plus prioritaire

traitement\_evenement\_1()

**sinon\_si** (occurrence de evenement\_2 mémorisée) **alors**

traitement\_evenement\_2()

…

**fin\_si**

**fin\_itérer**

-------------------------------------------------------------------------------

• Remarque : l’effacement de la mémorisation de l’occurrence de l’événement choisie et le traitement consécutif ont été mis dans une procédure de traitement pour structurer le programme.

**Exemple : monostable numérique non redéclenchable**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| • Description :    Une horloge servant de base de temps est connectée à l’entrée clk de la machine à états. On choisit les types suivants pour les entrées, sorties, variables et états.   |  |  | | --- | --- | | *Entrées* | start, stop : binaire | | *Entrée d’interruption* | clk : binaire | | *Sortie* | s : binaire | | *Variable* | ctr : entier | | *Etats* | idle, count : booléen | | *Diagramme états-transitions* |

• Remarque 1 : on note simplement « start » et « stop » sur le diagramme comme le veut l’habitude pour les signaux. L’interprétation est bien que c’est l’occurrence de l’événement front montant de « start » (↑start) – ou « stop » (↑stop) – qui déclenche la transition (si les conditions sont satisfaites).

• Remarque 2 : le choix des types de données est important pour une bonne programmation.

* Une donnée de type « binaire » ou « bit » peut prendre soit la valeur 0, soit la valeur 1 ; bit est une contraction de « binary digit » qui peut se traduire par « chiffre binaire ».
* Une donnée de type « booléen » peut prendre soit la valeur faux (false), soit la valeur vrai (true). Le résultat d’une comparaison est formellement du type booléen. Ce type permet de simplifier l’écriture des tests.
* D’autres types sont bien sûr possibles. Leur définition fine dépend du langage. Les types choisis ici pourront être adaptés à d’autres contextes.
* Un signal électrique (une tension en général) « logique » peut prendre soit le niveau bas (low), soit le niveau haut (high). C’est le cas pour notre exemple des signaux : start, stop, clk et s, connectés sur les entrées-sorties, qui correspondent respectivement aux variables de mêmes noms du programme : start, stop, clk et s de type binaire. En pratique, on considère qu’une tension est au niveau bas – ou au niveau haut – si elle est comprise dans une certaine plage de tension.

• Remarque 3 : le signal clk est connecté sur une entrée d’interruption. Une demande d’interruption (interrupt request) permet de prendre en compte « immédiatement » l’occurrence de l’événement souhaité (↑clk ou ↓clk, voire les deux) afin d’incrémenter la variable ctr de type « entier ». Ainsi la valeur de ctr est mise à jour en « temps réel ». Sans entrer dans les détails, cela est possible grâce au mécanisme de gestion d’une interruption par un processeur : lors d’une demande d’interruption, le processeur suspend le programme en cours (si la demande est plus prioritaire que l’exécution du programme en cours), exécute le programme associé à l’interruption jusqu’à sa fin, puis reprend le programme précédent là où il l’avait quitté.

**• Algorithme :**

-------------------------------------------------------------------------------

// déclarations et initialisations

// entrées, sorties, variables

start := **0**

start\_precedent := **0**

stop := **0**

stop\_precedent:= **0**

s := **0**

clk := **0**

ctr := **0**

ctr\_precedent := **0**

ctr\_max := **100**

// états

idle := **faux**

count := **faux**

// événements

when\_ctr\_sup\_ou\_egal\_ctr\_max := **faux**

front\_montant\_start := **faux**

front\_montant\_stop := **faux**

// procédure appelée à chaque demande d’interruption accordée

// et de calcul des occurrences des événements liés

**procédure** interruption\_clk()

**début**

**si** count **alors**

ctr := ctr + **1**

// calcul et mémorisation des occurrences des événements

**si** (ctr >= ctr\_max) **et** (ctr\_precedent < ctr\_max) **alors**

when\_ctr\_sup\_ou\_egal\_ctr\_max := **vrai**

**fin\_si**

ctr\_precedent := ctr

**fin\_si**

**fin**

// procédure d’acquisition des entrées

// et de calcul des occurrences des événements liés

**procédure** acquisition\_entrees()

**début**

// lire une seule fois les entrées pour la cohérence des calculs

start := entree\_logique\_processeur\_start

stop := entree\_logique\_processeur\_stop

// calcul et mémorisation des occurrences des événements

**si** (start = 1) et (start\_precedent = 0) **alors**

front\_montant\_start := **vrai**

**fin\_si**

**si** (stop = 1) et (stop\_precedent = 0) **alors**

front\_montant\_stop := **vrai**

**fin\_si**

start\_precedent := start

stop\_precedent := stop

**fin**

// procédures de traitement des occurrences des événements

**procédure** traitement\_stop()

**début**

// effacement de la mémorisation de l’occurrence de l’événement

front\_montant\_stop := **faux**

// traitement de l’occurrence de l’événement

**si** count **alors**

count := **faux**

idle := **vrai**

s := **0**

**fin**

**procédure** traitement\_when\_ctr\_max()

**début**

// effacement de la mémorisation de l’occurrence de l’événement

when\_ctr\_sup\_ou\_egal\_ctr\_max := **faux**

// traitement de l’occurrence de l’événement

**si** count **alors**

count := **faux**

idle := **vrai**

s := **0**

**fin**

**procédure** traitement\_start()

**début**

// effacement de la mémorisation de l’occurrence de l’événement

front\_montant\_start := **faux**

// traitement de l’occurrence de l’événement

**si** idle **alors**

idle := **faux**

count := **vrai**

ctr := **0**

s := **1**

**fin**

// transition initiale de la machine

idle := **vrai**

s := **0**

// programme principal

**itérer**

// acquisition des entrées et calcul des occurrences des événements

acquisition\_entrees()

// choix de l’occurrence de l’événement compte tenu de l’ordre de priorité

// et appel de la procédure de traitement correspondante

**si** (front\_montant\_stop) **alors** // événement le plus prioritaire

traitement\_stop()

**sinon\_si** (when\_ctr\_sup\_ou\_egal\_ctr\_max) **alors**

traitement\_when\_ctr\_max()

**sinon\_si** (front\_montant\_start) **alors** // événement le moins prioritaire

traitement\_start()

**fin\_si**

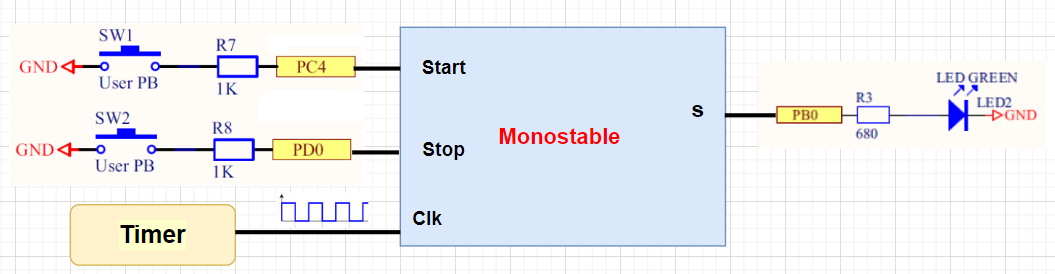
**fin\_itérer**

-------------------------------------------------------------------------------

**• Programme en micropython :**

Pour cette partie programmation, on a fait le choix de programmer le comportement du monostable en utilisant la carte de développement STM32WB55.

Pour les entrées binaires « start » et « stop » on utilisera deux des trois boutons poussoirs disponibles sur le carte. L’état de la sortie est visualisé une des trois LED de la carte. Un timer est utilisé pour générer le signal d’horloge.



''' Ce programme a été implanté sur une carte STM32WB55

L'entrée binaire "start" correspond au bouton SW1

L'entrée binaire "stop" correspond au bouton SW2

L'entrée binaire "clock" à un Timer virtuel

L'état de la sortie binaire "s" est visualisée sur la LED2 (bleue)

'''

# Déclaration des bibliothèques à importer

**import** pyb

**import** machine

'''

Déclarations et initialisations

Entrées, sorties, variables

'''

start **=** 0

start\_precedent **=** 0

stop **=** 0

stop\_precedent **=** 0

s **=** 0

ctr **=** 0

ctr\_precedent **=** 0

ctr\_max **=** 10

# Etats

idle **=** **False**

count **=** **False**

# Evènements

when\_ctr\_sup\_ou\_egal\_ctr\_max **=** **False**

front\_montant\_start **=** **False**

front\_montant\_stop **=** **False**

'''

Configurations des broches d'E/S digitales

On utilise les deux boutons poussoirs SW1 et SW2 pour les entrées start et stop

Chacun de ces boutons est relié à une broche du processeur et le 0 V

On activera donc pour chacun la résistance interne de pull-up de 10 Kohms

Si BP relaché => on a un 1 logique sur l'entrée du prcesseur

Si BP appuié = > on a un 0 logique sur l'entrée du prcesseur

SW1, SW2 sont connectés respectivement à PC4 et PD0

On utilisera une des trois diodes électroluminescentes disponibles sur la

carte STM32WB55 pour visualiser l'état de la sortie s du monostable

'''

BPstart **=** pyb**.**Pin**(**pyb**.**Pin**.**cpu**.**C4**,** pyb**.**Pin**.**IN**,** pyb**.**Pin**.**PULL\_UP**)**

BPstop **=** pyb**.**Pin**(**pyb**.**Pin**.**cpu**.**D0**,** pyb**.**Pin**.**IN**,** pyb**.**Pin**.**PULL\_UP**)**

LEDsortie **=** pyb**.**LED**(**2**)** # on choisit la LED2 verte

'''

Fonction appelée à chaque demande d'interruption accordée et de calcul des

occurrences des évènements liés

'''

**def** interruption\_clock**(**tim**):**

**global** ctr **;** **global** ctr\_precedent **;** **global** count

**global** ctr\_max **;** **global** when\_ctr\_sup\_ou\_egal\_ctr\_max

**if** count **==** **True** **:**

ctr **=** ctr **+** 1

# calcul et mémorisation des occurrences des évènements

**if** **((**ctr **>=** ctr\_max**)** **and** **(**ctr\_precedent **<** ctr\_max**)):**

when\_ctr\_sup\_ou\_egal\_ctr\_max **=** **True**

ctr\_precedent **=** ctr

'''

Fonction d'acquisition des entrées et de calcul des occurrences des

évènements liés

'''

**def** acquisition\_entrees**():**

**global** start **;** **global** stop **;** **global** start\_precedent **;** **global** idle

**global** front\_montant\_start **;** **global** stop\_precedent

**global** front\_montant\_stop

#lire une seule fois les entrées pour la cohérence des calculs

start **=** **not** BPstart**.**value**()** # appui sur BP SW1, BPstart = 0

stop **=** **not** BPstop**.**value**()** # appui sur BP SW2, BPstop = 0

# calcul et mémorisation des occurrences des évènements

**if** **(**start **==** 1**)** **and** **(**start\_precedent **==** 0**)** **:**

front\_montant\_start **=** **True**

**if** **(**stop **==** 1**)** **and** **(**stop\_precedent **==** 0**)** **:**

front\_montant\_stop **=** **True**

start\_precedent **=** start

stop\_precedent **=** stop

'''

Procédures de traitement des occurrences des évènements

'''

**def** traitement\_stop**():**

**global** count **;** **global** front\_montant\_stop **;** **global** idle **;** **global** s

# effacement de la mémorisation de l'occurrence de l'évènement

front\_montant\_stop **=** **False**

# traitement de l'occurrence de l'évènement

**if** count **==** **True** **:**

count **=** **False**

idle **=** **True**

s **=** 0

LEDsortie**.**off**()**

**def** traitement\_when\_ctr\_max**():**

**global** count **;** **global** when\_ctr\_sup\_ou\_egal\_ctr\_max **;** **global** idle **;** **global** s

# Effacement de la mémoristion de l'occurrence de l'évènement

when\_ctr\_sup\_ou\_egal\_ctr\_max **=** **False**

# Traitement de l'occurrence de l'évènement

**if** count **==** **True** **:**

count **=** **False**

idle **=** **True**

s **=** 0

LEDsortie**.**off**()**

**def** traitement\_start**():**

**global** count **;** **global** front\_montant\_start **;** **global** idle **;** **global** idle

**global** s **;** **global** ctr

# effacement de la mémorisation de l'occurrence de l'évènement

front\_montant\_start **=** **False**

# traitement de l'occurrence de l'évènement

**if** idle **==** **True** **:**

idle **=** **False**

count **=** **True**

ctr **=** 0

s **=** 1

LEDsortie**.**on**()**

'''

Transistion initiale de la machine

'''

idle **=** **True**

s **=** 0

LEDsortie**.**off**()**

# Simulation du signal horloge de période 1 seconde

tim **=** machine**.**Timer**(-**1**)**

tim**.**init**(**period**=**1000**,** callback **=** interruption\_clock**)**

'''

Programme principal

'''

**while** **True** **:**

# Acquisition des entrées et calcul des occurrences des évènements

acquisition\_entrees**()**

# Choix de l'occurrence de l'évènement compte tenu de l'ordre de priorité

# et appel de la fonction de traitement correspondante

**if** front\_montant\_stop **==** **True** **:**

traitement\_stop**()**

**elif** when\_ctr\_sup\_ou\_egal\_ctr\_max **==** **True** **:**

traitement\_when\_ctr\_max**()**

**elif** front\_montant\_start **==** **True** **:**

traitement\_start**()**

**Algorithmes pour quelques cas**

• Pour éviter des aléas, les occurrences des événements et les gardes sont calculées à partir d’une photographie (mémorisation) à un même instant des entrées et variables nécessaires. Voir ce qui précède.

• Notations :

- **si** (event = trigger\_A) : si l’événement (event) choisi dans la mémoire des événements (pool) est un déclencheur (trigger\_A) pour la transition A.

- **si** (guard\_A)  : si la condition de garde (guard\_A) de la transition A est vraie.

- **si** (state\_1 = **actif**) : si l’état 1 (state\_1)est actif.

|  |  |
| --- | --- |
| *Diagramme* | *Algorithme* |
| *Transition simple entre deux états simples* | **si** (event = trigger\_A) **alors**  **si** (guard\_A) **alors**  **si** (state\_1 = **actif**) **alors**  arrêter doActivity\_1  exit\_beh\_1  state\_1 := **inactif**  beh\_A  state\_2 := **actif**  entry\_beh\_2  démarrer doActivity\_2  **fin\_si**  **fin\_si**  **fin\_si** |
| *Transition simple externe « propre »*  *(i.e. état source = état cible)* | **si** (event = trigger\_A) **alors**  **si** (guard\_A) **alors**  **si** (state\_1 = actif) **alors**  arrêter doActivity\_1  exit\_beh\_1  state\_1 := **inactif**  beh\_A  state\_1 := **actif**  entry\_beh\_1  démarrer doActivity\_1  **fin\_si**  **fin\_si**  **fin\_si** |
| *Transition interne à un état* | **si** (event = event\_1) **alors**  **si** (guard\_1) **alors**  **si** (state\_1 = **actif**) **alors**  event\_beh\_1  **fin\_si**  **fin\_si**  **fin\_si** |
| *Transitions entre un état simple et un état composite avec une seule région* | **si** (event = tA) **alors**  **si** (gA) **alors**  **si** (state\_1 = **actif**) **alors**  arrêter doA\_beh\_1  exi\_beh\_1  state\_1 := **inactif**  bA  state\_2 := **actif**  ent\_beh\_2  démarrer doA\_beh\_2  state\_21 := **actif**  ent\_beh\_21  démarrer doA\_beh\_21  **fin\_si**  **fin\_si**  **fin\_si**  **--------------------------------------------------**  **si** (event = tB) **alors**  **si** (gB) **alors**  **si** (state\_2 = **actif**) **alors**  **si** (state\_21 = **actif**) **alors**  arrêter doA\_beh\_21  exi\_beh\_21  state\_21 := **inactif**  **fin\_si**  **si** (state\_22 = **actif**) **alors**  arrêter doA\_beh\_22  exi\_beh\_22  state\_22 := **inactif**  **fin\_si**  arrêter doA\_beh\_2  exi\_beh\_2  state\_2 := **inactif**  bB  state\_1 := **actif**  ent\_beh\_1  démarrer doA\_beh\_1  **fin\_si**  **fin\_si**  **fin\_si** |
| *Transitions avec un état composite à deux régions* | **si** (event = trigger\_A) **alors**  **si** (guard\_A) **alors**  **si** (state\_1 = **actif**) **alors**  arrêter doAct\_beh\_1  exit\_beh\_1  state\_1 := **inactif**  beh\_A  state\_2 := **actif**  entry\_beh\_2  démarrer doAct\_beh\_2  beh\_C  beh\_D  state\_21 := **actif**  entry\_beh\_21  démarrer doAct\_beh\_21  state\_22 := **actif**  entry\_beh\_22  démarrer doAct\_beh\_22  **fin\_si**  **fin\_si**  **fin\_si**  **--------------------------------------------------**  **si** (event = trigger\_B) **alors**  **si** (guard\_B) **alors**  **si** (state\_21 = **actif**) **et**  (state\_22 = **actif**) **alors**  arrêter doAct\_beh\_21  exit\_beh\_21  state\_21 := **inactif**  arrêter doAct\_beh\_22  exit\_beh\_22  state\_22 := **inactif**  arrêter doAct\_beh\_2  exit\_beh\_2  state\_2 := **inactif**  beh\_B  state\_3 := **actif**  entry\_beh\_3  démarrer doAct\_beh\_3  **fin\_si**  **fin\_si**  **fin\_si** |
| *Transition d’achèvement entre deux états simples*    Remarque : en présence de « doActivity\_beh\_2 », l’événement d’achèvement est généré après que « doActivity\_beh\_2 » soit terminé (cas non traité ici). | **si** (event = trigger\_A) **alors**  **si** (guard\_A) **alors**  **si** (state\_1 = **actif**) **alors**  arrêter doActivity\_1  exit\_beh\_1  state\_1 := **inactif**  beh\_A  state\_2 := **actif**  entry\_beh\_2  completion\_event\_2  //événement d’achèvement  **fin\_si**  **fin\_si**  **fin\_si**  **--------------------------------------------------**  **si** (event = completion\_event\_2) **alors**  **si** (guard\_B) **alors**  **si** (state\_2 = **actif**) **alors**  exit\_beh\_2  state\_2 := **inactif**  beh\_B  state\_3 := **actif**  entry\_beh\_3  démarrer doActivity\_3  **fin\_si**  **fin\_si**  **fin\_si** |

**Comportements d’entrée (entry/), de sortie (exit/), durant l’état actif (do/)**

• Les comportements d’entrée (entry/) et de sortie (exit/) sont de « courtes » durées et ne peuvent pas être interrompus.

• Le comportement durant l’état actif (do/) est effectué après le comportement d’entrée (entry/), avant le comportement de sortie (exit/), et tant que l’état est actif ou jusqu’à ce qu’il se termine par lui-même. En conséquence, un comportement non terminé durant l’état actif (do/) sera interrompu en cas de sortie de l’état.

* On ne traitera pas ici des comportements durant l’état actif (do/) de manière générale car l’écriture nécessiterait des notions plus avancées de programmation (comme les processus légers ou fils d’exécution, thread en anglais).
* Cependant, dans le cas où un comportement durant l’état actif (do/) est de « courte » durée, déclenché par des instructions exécutées une seule fois, il suffit d’insérer les instructions après celles du comportement d’entrée (entry/).
* Dans le cas où un comportement durant l’état actif (do/) nécessite de répéter des instructions ou d’exécuter des instructions en fonction d’un contexte évoluant, on pourra prévoir une partie de programme adaptée (et conditionnée à l’état actif considéré) dans le programme principal. Par exemple, pour la partie de comportement de doActivity\_beh\_2 concernée, on pourrait avoir :

-------------------------------------------------------------------------------

**si** (state\_2 = **actif**) **alors**

partie de comportement de doActivity\_behavior\_2

**fin\_si**

-------------------------------------------------------------------------------

**Etat composite**

• L’entrée dans un état composite provoque l’activation de toutes les régions (après avoir effectué le comportement d’entrée).

• L’entrée dans un état composite et les comportements associés sont effectués séquentiellement du niveau hiérarchique le plus élevé (le moins profond) – l’état composite lui-même – au niveau le plus faible (le plus profond) de chaque région.

• La sortie d’un état composite provoque la désactivation de toutes ses régions et donc de tous ses états actifs.

• La sortie d'un état composite et les comportements associés sont effectués séquentiellement du niveau hiérarchique le plus faible (le plus profond) au niveau le plus élevé (le moins profond) de chaque région, pour finir par la sortie de l’état composite lui-même.

**Pseudo-états historiques profond et superficiel**

• Il convient de tracer (avec une variable adaptée) la configuration d’état actif, c'est-à-dire la mémoriser au fur et à mesure de son évolution, afin de pouvoir y revenir le cas échéant. Voir ressource : éléments sur quelques diagrammes SysML.

**Priorité d’un événement d’achèvement**

• En SysML-UML, le concept d’événement toujours occurrent n’est pas prévu mais il existe le concept d’événement d’achèvement (completion event). Un événement d’achèvement possède la plus haute priorité.

**Evénement différé**

• Un événement peut être différé par un état. Pour connaitre le comportement de la machine, voir ressource : ***éléments sur quelques diagrammes SysML***.