SYSTÈMES À ÉVÈNEMENTS DISCRETS

1 Modélisation du comportement logique des systèmes

1.1 Description par processus et par système à événements discrets

Comme cela a déjà été abordé dans le cours de première année, les systèmes modernes comportent très souvent une commande numérique, réalisée par un ou plusieurs circuits logiques (type FPGA) ou calculateurs (type microcontrôleur et DSP). Les exemples vus en première année ont consisté à élaborer, à partir d'un cahier des charges simple du comportement attendu, le programme C correspondant, puis à le charger dans un microcontrôleur afin de vérifier le résultat.

Produire du code en langage C directement depuis un cahier des charges est tout à fait possible mais cela reste néanmoins limité à de petits programmes. Cette manière de faire bute sur plusieurs points :

— une analyse trop succincte du cahier des charges risque de conduire à un programme qui ne répond pas réellement à ce qui est arrêté.
— Les ajustements de fonctionnalités vont conduire à des petits bouts de codes correctifs éparpillés pour éviter de reprendre la totalité du programme. Le logiciel est alors une construction sur une base inadaptée, refaçonnée superficiellement pour satisfaire les exigences.
— le logiciel produit est déstructuré et devient une « usine à gaz » à maintenir et à faire évoluer.
— Il n'y a aucun outil de documentation systèmatique permettant de comprendre la structure du code. Toute modification nécessite au préalable d'analyser le code pour comprendre où intervenir. C'est un des gros travers dans le monde du développement.

Même si la première version s'avère fonctionnelle, il y a de fortes chances pour que le logiciel pose de gros problèmes à l'avenir (maintenance, évolution, etc). Les versions successives demanderont des modifications telles qu'il deviendra rapidement plus sage de repenser le programme dans sa totalité.

En règle générale, les systèmes modernes complexes intègrent beaucoup d'informatique. Pour saisir ce qu'il doit accomplir, il est nécessaire de modéliser le comportement à haut niveau (au niveau système) afin d'aboutir à un modèle simulable (on dit aussi exécutable) avant d'avoir produit la moindre ligne de code. C'est le rôle des diagrammes de séquences, d'activités et des machines d'états. Ce sont surtout ces deux derniers qui vont nous intéresser.

Dans sa vue la plus simple, le diagramme d'activités peut être assimilé à un algorigramme. C'est-à-dire qu'il décrit un déroulement qui, une fois engagé, va à son terme. C'est le cas par exemple de la résolution d'une équation du second degré (figure1). Les étapes sont connues et suivent un ordre bien précis. Il y a des alternatives possibles mais elle aboutit nécessairement à un résultat (deux racines réelles ou complexes ou une racine double).

Malgré une certaine ressemblance graphique, les machines d'états décrivent de manière différente le comportement. Elles permettent de représenter les différentes phases de fonctionnement traversées par le système en réaction à des événements discrets. Contrairement à un processus qui se poursuit presque indépendamment des évènements extérieurs, l'évolution de la machine d'états est conditionnée par les événements qu'elle reçoit.

1.1.0.1 Comment organiser la description du comportement d'un système ?

Fondamentalement la norme ne définit pas de hiérarchie entre les diagrammes. Leurs objectifs sont différents et leur utilisation dépend du contexte. Mais dans leur grande majorité, les systèmes sont avant tout pilotés par des

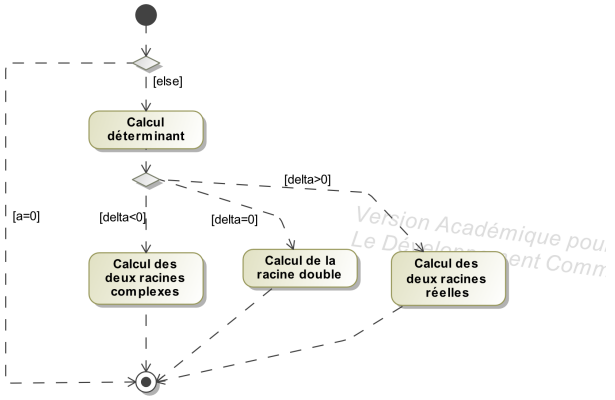


FIGURE 1 – Illustration avec un diagramme d'activités ?.

évènements, qui conduiront à différents traitements dans chaque mode de fonctionnement.

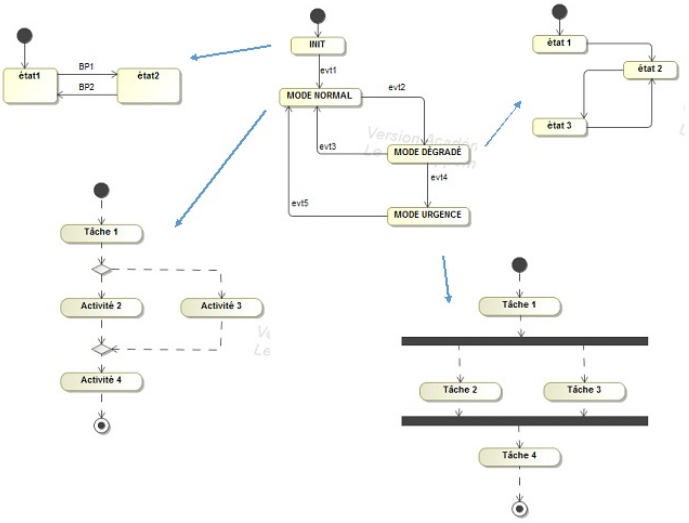


FIGURE 2 – Illustration avec machine d'états principale et sous-description.

L'organisation classique de la commande d'un système est du type décrit figure 2 : une machine d'états principale permet de représenter les différents modes de fonctionnement, puis chaque état est décrit par le diagramme approprié en fonction du cahier des charges. Cela peut être un diagramme d'activités ou une machine d'états de plus bas niveau, voire un état d'attente.

L'exemple qui servira de fil rouge par la suite est un système (simplifié) de trains d'atterrissage d'avion. Les trois trains d'atterrissage sont commandés simultanément par trois vérins hydrauliques alimentés par une pompe unique. Le pilote dispose d'une commande *rentrer train/sortir train* et d'un afficheur indiquant l'état du train. Des capteurs situés aux extrémités du mouvement des trains permettent de détecter les fin de courses. Une description simple du comportement par diagramme d'états est donnée à la figure 3.

1.2 Machine d'états simplifiée

Bien que la norme permette de décrire des fonctionnements complexes, une machine d'états simple est assez facile à comprendre. Elle possède peu d'éléments syntaxiques :

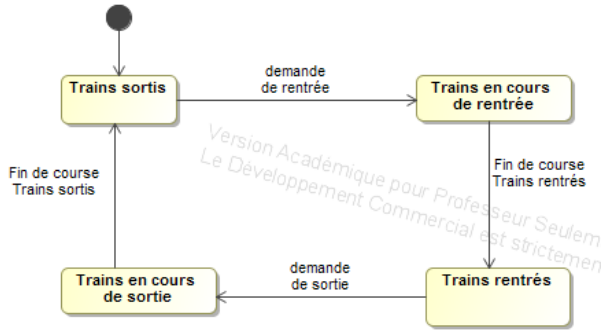


FIGURE 3 – Diagramme d'états décrivant le fonctionnement du train d'atterrissage.

— les *états* sont représentés pas un rectangle aux coins arrondis, ou parfois par des cercles ou des ovales (cela n'a pas une grande importance). Ils représentent l'état dans lequel se situe une entité (au sens large du terme mais elle est logicielle dans le cadre de notre propos.) au cours de son existence. Un seul état peut être actif à la fois.
— les *transitions* sont des liens orientés représentés par des flèches. La flèche part de l'état source et pointe vers l'état cible. Elles permettent de représenter les chemins possibles d'évolution de la machine d'états.
— les *événements* sont associés aux transitions. Ils sont notés à côté d'une transition, indiquant qu'une occurrence de l'événement déclenche automatiquement le franchissement de la transition. L'activité en cours dans l'état source est aussitôt interrompue et celle dans l'état cible est démarrée (s'il y en a une).

L'évolution de la machine d'états se fait uniquement sur la base d'événements reçus (il n'y a pas d'évolution possible sans évènement). Un exemple générique pourrait être celui de la figure 4.

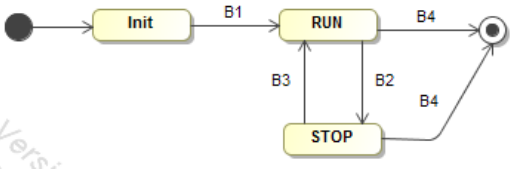


FIGURE 4 – Exemple élémentaire de machine d'états.

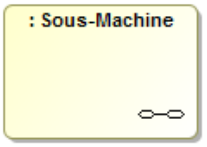
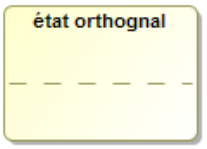
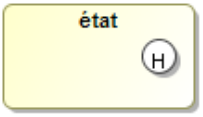
Le cercle noir marque le point de départ de la machine d'états. Une transition mène directement au premier état *Init*. Une transition sans événement est automatiquement franchie lorsque l'état source a fini son activité. Pour la suite, seul l'appui sur le bouton B1 permettra d'accéder à l'état *RUN*. Ensuite le système alterne entre les états *RUN* et *STOP* en fonction des appuis sur B2 et B3. Dans tous les cas, l'appui sur le bouton B4 amène à un état terminal signant la fin du fonctionnement.

2 Notation standard de la machine d'états

En plus des éléments décrits ci-dessus, la norme propose d'autres objets permettant de décrire des situations plus complexes.

2.1 États

Le tableau ci-dessous définit d'autres types d'états possibles.

Dénomination	Symbole Graphique	Commentaires
État composite		L'*état composite* permet d'encapsuler une sous-machine d'états dans un état. Un autre diagramme d'états permet de décrire comment fonctionne cette sous-machine d'états.
Concurrence		Un *état orthogonal* possède plusieurs régions (deux dans le schéma ci-contre), séparée par des pointillés. Chaque région possède sa propre description à états et toutes fonctionnent en parallèle. Elles sont dites en *concurrence*.
Historique		La désactivation d'un état noté *historique* fige l'activité en cours, qui sera reprise à la réactivation de l'état.
État de départ		Pseuso-état permettant de préciser où commence le fonctionnement de la machine d'états. Il ne peut y en avoir qu'un seul par diagramme. En tant que pseudo-état, il ne peut pas avoir d'activités internes ou de transitions internes.
État final		Pseudo-état de fin permettant d'indiquer la fin du fonctionnement de la machine d'états (état piège). Il peut y en avoir plusieurs où aucun (aucune fin particulière n'a été prévue).

2.2 Transitions

2.2.1 Transitions externes

Comme indiqué au paragraphe 1.2, une transition est notée comme une simple flèche associée à un événement. La norme définit par ailleurs d'autres éléments offrant plus de possibilités. L'écriture générique d'un événement est la suivante :

événement [garde]/effet.

2.2.1.1 Détail de la notation

2.2.1.2 Événement :

Pour un système, un évènement est l'occurrence d'un phénomène. L'évènement n'est pas caractérisé par une durée : il a seulement « eu lieu » ou pas. On trouvera des exemples basiques comme « départ opérateur » ou « arrivée d'une pièce ». La norme définit aussi des événements temporels. Ceux-ci sont de deux types :

— l'événement relatif, est noté *after(T)*. L'évènement se déclenche après le temps T passé dans l'état amont.
— l'événement absolu, noté *when(H)*. L'évènement se déclenche à la date H dans un référentiel de temps, dont l'origine correspond généralement au démarrage de la machine à états.

2.2.1.3 Garde :

Il s'agit d'une expression logique permettant de valider ou pas le franchissement d'une transition. Lorsque l'événement associé à la transition est effectif, la garde est évaluée et la transition franchie seulement si l'expression est vrai.

2.2.1.4 Effet :

un effet accompagne le franchissement d'une transition. Cela peut être n'importe quel action sans limite temporelle. En règle générale, un effet se limite à une action élémentaire.

Lorsqu'aucun évènement n'est explicitement indiqué à coté d'une transition, l'évènement implicite est la terminaison de l'activitée assurée dans l'état.

2.2.2 Transitions internes

Il existe des transitions dites *internes* car elles ne conduisent pas à un changement d'état. Elles sont notées dans l'état. Une transition bouclant sur un état existe aussi, mais déclenche d'autres traitements comme nous allons le voir ci-après.

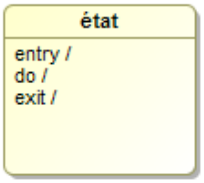


FIGURE 5 – Transitions internes dans un état.

Il est possible de créer des évènements internes quelconques dans les états. Nous nous limiterons cependant à 3 évènements internes proposés par la norme, et décrits sur la figure 6 :

— *entry*, évènement déclenché lors de l'entrée dans l'état. Il sert souvent d'initialisation à l'activité indiquée dans le *do*.
— *do*, évènement déclenché après l'événement interne *entry*. C'est ce que fait le bloc une fois "installé" dans l'état (son activité courante).
— *exit*, évènement déclenché lors de la sortie de l'état. Il permet d'effectuer les actions liées à la fin de l'activité située dans le *do* (sauvegarder des données par exemple).

L'occurrence d'un événement interne déclenche l'activité associée sans déclencher celles qui sont associées à *entry* et *exit* puisque l'état reste actif. Par contre, il est possible de tracer une transition bouclant sur un même état à la place d'un événement interne. Dans ce cas les activités liées à *entry* et *exit* sont exécutées (figure 7).

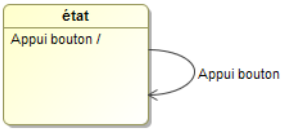


FIGURE 6 – Exemple de transition interne

2.3 Déroulement du fonctionnement d'une machine d'états

Le fonctionnement d'une machine d'état est relativement simple à comprendre et suit les étapes suivantes :

— il n'existe qu'un seul pseudo-état de départ indiquant le point de départ de la machine d'états.

— les transitions sont franchies en fonction des événements et des gardes. La suite des états actifs est donc fonction des événements qui se produisent.

— un seul état peut être actif à la fois. Attention, les états orthogonaux sont particuliers sur ce point car les sous-machines d'états faisant partie de chaque région ne possèdent pas les mêmes états que la machine d'états principale. Il est donc possible d'aboutir à plusieurs états actifs, mais chacun appartenant à une machine d'état distincte.

3 Démarche de développement

Le langage SysML est un outil d'ingénierie système, dont l'objectif premier est de décrire à haut niveau le système et son comportement, c'est-à-dire sans s'attacher aux détails de réalisation matériel et en donnant priorité au sens perçu par le client. Une description de haut niveau s'oppose à une description de bas niveau, qui permet de définir précisement les composants matériels et les détails d'implémentation du code.

Néanmoins, la description SysML de haut niveau tente d'accompagner les concepteurs tout au long du processus de conception. Si la description du système en début de projet sera très globale et peu précise, elle s'enrichie au fil du projet pour comporter de plus en plus de détails qui permettront de définir la solution technique réalisée.

Ainsi, une machine d'état en fin de phase de conception, juste avant la phase d'implémentation du code, sera très détaillée, au point de permettre une simulation du comportement du système pour validation du cahier des charges.

La machine d'état finale peut parfois servir de support pour une génération de code automatique. Ce n'est néanmoins pas l'objectif de la description SysML : la phase d'implémentation du code est hors du champ du langage SysML et elle sera traitée à part.

Dans les parties qui suivent nous allons nous attacher à mettre au point la description du comportement par machine d'états du système de déploiement du train d'atterrissage présenté au paragraphe 1.1. En partant de la formulation des exigences du client. À ce stade, l'architecture du système n'est pas encore définie.

3.1 Traduire les exigences client en diagramme d'états

Dans un premier temps, la description doit correspondre au besoin client tout en restant dans l'espace du problème. Le comportement doit être décrit sans penser aux solutions constructives adoptées par la suite. La description reste au niveau système, et conduit à une première version figure 8.

Cette première version fait apparaitre un cycle unidirectionnel, c'est-à-dire qu'il est nécessaire de sortir les trains complètement pour ensuite les rentrer. Bien évidemment, des modifications sont à apporter pour moduler le comportement. Une discussion plus précise avec le client amène la machine d'états figure 9 (toujours au niveau système).

Un état d'initialisation du système a été ajouté ainsi que la possibilité d'agir sur la sortie ou la rentrée du train d'atterrissage à n'importe quel moment. Les affichages ne nécessitent pas une description particulière dans l'immédiat. Par contre, nous allons détailler figure 10 les activités liées au déploiement et au repliement du train.

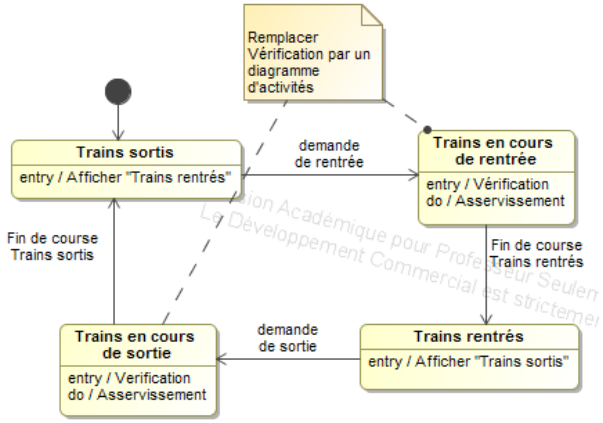


FIGURE 7 – Diagramme d'état traduisant ce que le client exprime initialement.

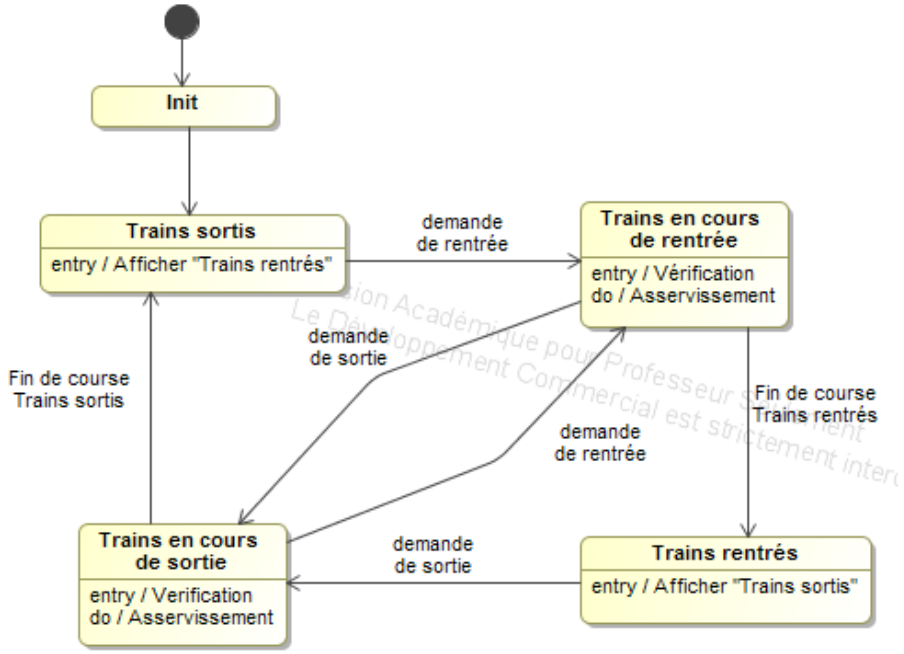


FIGURE 8 – Diagramme d'états obtenu après prise en compte des contraintes techniques.

3.2 Prise en compte du point de vue « métier » dans le diagramme d'états

Tout en restant dans le domaine du problème, les ingénieurs qui réaliseront le système apportent leur expertise métier. L'ingénieur soulève des problèmes que le client ne connait pas nécessairement puisque ce n'est pas son cœur de métier. Cela aboutira à la modification du modèle de base en accord avec le client. Dans notre exemple, le concepteur du système de train d'atterrissage souhaite prendre en compte le problème du déverrouillage du train lors de la sortie. Celui-ci peut en effet se bloquer. Pour prendre en compte cette possibilité, il a été convenu avec le client que si le déverrouillage ne se faisait pas, alors un message devait être envoyé au pilote afin que celui-ci acquitte le problème. Ceci fait, les trappes doivent être refermées pour revenir à l'état ou les trains sont rentrés. Cela donne le diagramme d'activités figure 11.

La machine d'états sera aussi modifiée pour prendre en compte le fait de ne pas rentrer le train alors que l'avion est encore au sol (figure 12).

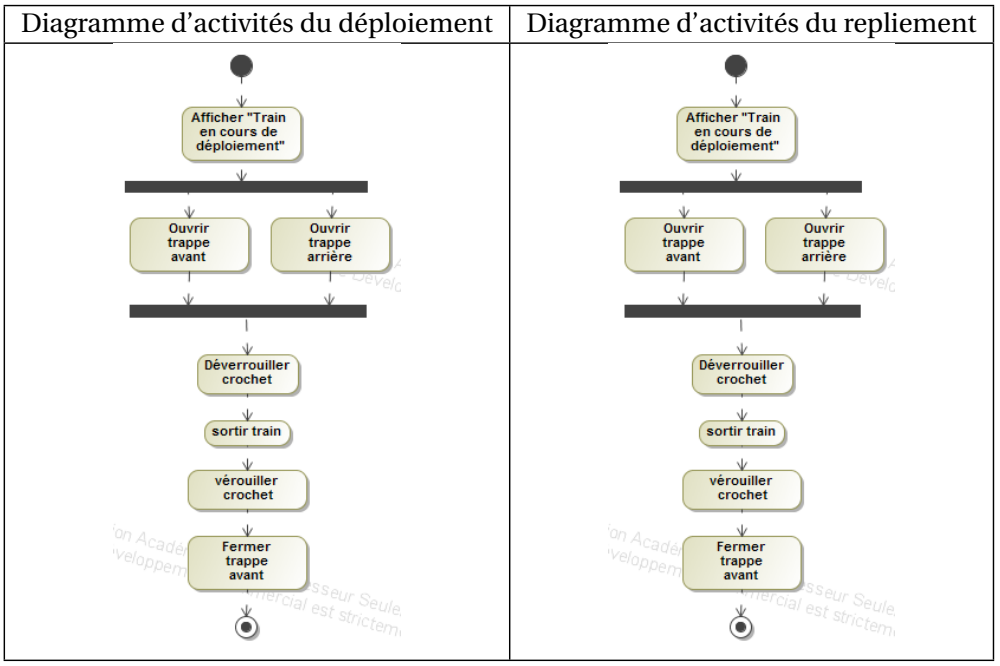


FIGURE 9 – Diagrammes d'activité de déploiement et de repliement du train.

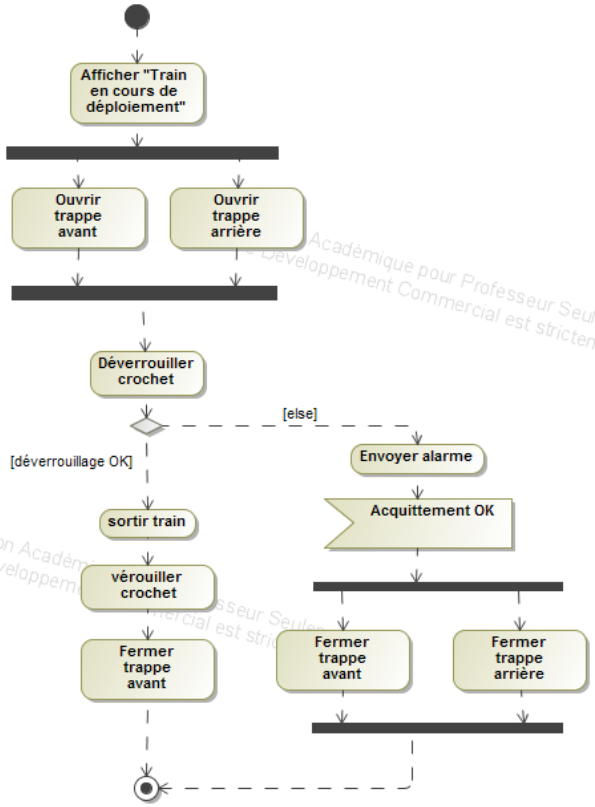


FIGURE 10 – Diagramme d'activité du déploiement du train, obtenu après prise en compte des contraintes techniques.

4 Implentation en C et Python d'une machine d'états

Une grande majorité des machines d'états sont codées dans un langage procédural. Il reste possible de la coder sur un réseau de portes logiques (type FPGA pour des systèmes de traitements très rapides) ou encore de façon

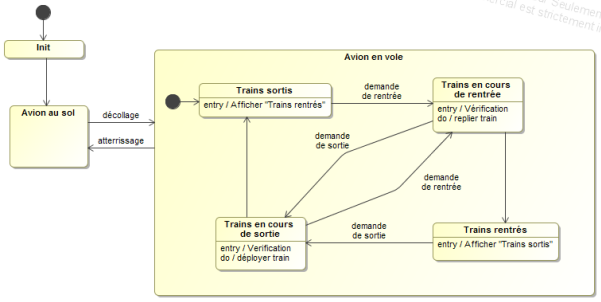


FIGURE 11 – Diagramme d'états obtenu après prise en compte des contraintes techniques.

matérielle mais c'est plus rare.

Le langage presque exclusif pour la programmation des microcontrôleurs embarquées est le C. Les cours d'informatique en CPGE étant pour partie réalisés en Python, l'implémentation d'une machine d'états sera illustrée parallèlement en C et en Python.

La formalisation d'une machine d'état simple [1] peut se résumer à :

— un ensemble fini d'états Q,
— un ensemble fini d'évènements d'entrée E,
— un ensemble fini de sorties S,
— une fonction de transitions qui à un état et un évènement d'entrée associe un nouvel état $Q \times E \longrightarrow Q$,
— une fonction de sortie qui à un état associe une sortie $Q \longrightarrow S$
— un état initial Q_0.

Cette description ne tient pas compte de l'ensemble des possibilités du langage SysML mais suffit à comprendre les principes généraux permettant de programmer une machine d'états.

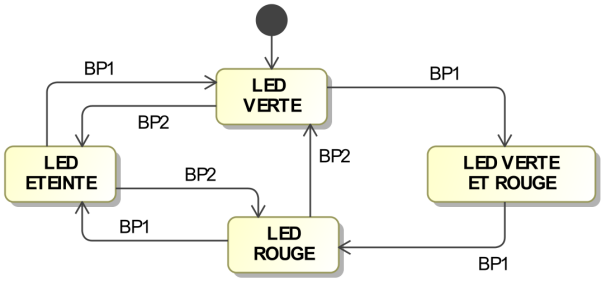


FIGURE 12 – Diagramme d'état à implémenter en C et Python.

L'exemple simple qui sera traité consiste en une machine d'état de 4 états et 7 transitions décrite figure 13. Deux entrées par boutons poussoirs et deux LED en sortie permettront de tester les algorithmes sous Python (à gauche) et C (à droite).

4.1 Organisation des états

La définition de la machine d'états commence par une liste des états possible permet de définir Q. De même, un liste des évènements d'entrée possibles permet de définir E.

En python, ces listes sont informatives car les clés des dictionnaires utilisés par la suite peuvent constituer ces ensembles. En C, la définition des énumérations est utile pour manipuler les états et évènements autrement que par

1. La description utilisée se rapproche d'une machine de Moore.

des indices de tableaux.

```
# Les états  sont stockés dans une liste
Etats=["DEPART",\
       "LED VERTE",\
       "LED ROUGE",\
       "LED ETEINTES",\
       "LED VERTE ET ROUGE"]

# définition  des évenements d'entrée E
Evenements=["VIDE","BOUTON_1",\
            "BOUTON_2"]
```

```
// Les  etats  sont  stockes  dans une  liste
enum Etats { DEPART, LED_VERTE,
   LED_ROUGE, LED_ETEINTES,
   LED_VERTE_ET_ROUGE,
   N_ET } etat;

// definition  des evenements d'entree E
enum Evenements { VIDE, BOUTON_1,
   BOUTON_2, N_EVS
   } evenement;

enum Etats transitions [N_ET][N_EVS];
```

4.2 Organisation des transitions

Les transitions dépendent des états et des évènements : pour chaque état, un certain nombre de transitions sont possible en fonction des évènements d'entrée. Le résultat de la fonction de transition est le nouvel état atteint par la machine d'états.

La table des transitions est donc une table d'autant de lignes que d'états et autant de colonnes que d'évènements. Elle contient l'état en sortie de transition.

En python, cette table est stockée sous forme de dictionnaire, ce qui permet de manipuler explicitement les clés et de vérifier facilement l'existence d'une transition.

```
# définition  des fonctions de transition
transitions={}

transitions["DEPART"]={}
transitions["DEPART"]["VIDE"] = "LED VERTE"

transitions["LED VERTE"]={}
transitions["LED VERTE"]["BOUTON_1"] = "LED VERTE ET ROUGE"
transitions["LED VERTE"]["BOUTON_2"] = "LED ETEINTES"

transitions["LED ROUGE"]={}
transitions["LED ROUGE"]["BOUTON_1"] = "LED ETEINTES"
transitions["LED ROUGE"]["BOUTON_2"] = "LED VERTE"

transitions["LED ETEINTES"]={}
transitions["LED ETEINTES"]["BOUTON_1"] = "LED VERTE"
transitions["LED ETEINTES"]["BOUTON_2"] = "LED ROUGE"

transitions["LED VERTE ET ROUGE"]={}
transitions["LED VERTE ET ROUGE"]["BOUTON_1"] = "LED ROUGE"
```

En C, cette table est stockée sous forme d'un tableau à deux dimensions. L'utilisation des *enum* permet de manipuler des clé explicites.

```
// definition  des  transitions
 transitions [DEPART][VIDE]=LED_VERTE;

 transitions [LED_VERTE][VIDE]=LED_VERTE;
```

```
transitions [LED_VERTE][BOUTON_1]=LED_VERTE_ET_ROUGE;
transitions [LED_VERTE][BOUTON_2]=LED_ETEINTES;

transitions [LED_ROUGE][VIDE]=LED_ROUGE;
transitions [LED_ROUGE][BOUTON_1]=LED_ETEINTES;
transitions [LED_ROUGE][BOUTON_2]=LED_VERTE;

transitions [LED_ETEINTES][VIDE]=LED_ETEINTES;
transitions [LED_ETEINTES][BOUTON_1]=LED_VERTE;
transitions [LED_ETEINTES][BOUTON_2]=LED_ROUGE;

transitions [LED_VERTE_ET_ROUGE][VIDE]=LED_VERTE_ET_ROUGE;
transitions [LED_VERTE_ET_ROUGE][BOUTON_1]=LED_ROUGE;
```

4.3 Organisation des fonctions de sortie

Les fonctions de sortie sont exécutées lorsqu'un état est actif, pour modifier les sorties (ici l'état des deux LED rouge et vertes).

Ces fonctions sont définies puis associées dans un tableau de fonctions (un dictionnaire en python et un tableau de pointeurs en C) pour qu'à chaque état corresponde une fonction de sortie.

```python
# définition des fonctions de sortie
def led_verte():
  print("VERT")

def led_rouge():
  print("ROUGE")

def led_eteintes():
  print("ETEINT")

def led_verte_et_rouge():
  print("VERT ET ROUGE")

# sous forme d'un dictionnaire
sorties={}
sorties["LED VERTE"] = led_verte
sorties["LED ROUGE"] = led_rouge
sorties["LED ETEINTES"] = led_eteintes
sorties["LED VERTE ET ROUGE"] = \
                    led_verte_et_rouge
```

```arduino
// definition des fonctions de sortie
void depart(){
  digitalWrite(8,LOW);
  digitalWrite(9,LOW); }
void led_verte(){
  digitalWrite(8,HIGH);
  digitalWrite(9,LOW); }
void led_rouge(){
  digitalWrite(8,LOW);
  digitalWrite(9,HIGH); }
void led_verte_et_rouge(){
  digitalWrite(8,HIGH);
  digitalWrite(9,HIGH); }
void led_eteintes(){
  digitalWrite(8,LOW);
  digitalWrite(9,LOW); }

void (*const sorties [N_ET]) (void) =
  {depart, led_verte, led_rouge,
   led_eteintes, led_verte_et_rouge};
```

4.4 Structuration et programmation de la machine d'états

Au cours du temps, la variable *état_courant* stock l'état actif de la machine d'état et la variable *évenement_courant* stock l'évènement à traiter.

Après avoir initialisé l'état courant à l'état DEPART, la machine d'état est lancée par une boucle infinie.

A chaque évènement relevé, l'appel au tableau *transition* permet de déterminer le nouvel état courant (après avoir éventuellement vérifié l'existence de la transition). L'appel à la fonction de sortie permet de modifier les sorties (l'activation des LED ici).

Le programme Python est conçu pour être testé dans une console, si bien qu'une entrée clavier et un test de la valeur entrée permet de déterminer l'évènement à prendre en compte.

Le programme C est conçu pour être mis en oeuvre sur Arduino, si bien qu'une phase préalable d'identification des fronts montants des boutons poussoirs est réalisée.

```
# état de départ :
etat="DEPART"
evenement="VIDE"


# BOUCLE DE LA MACHINE D'ÉTATS
while (True):
  // entrée clavier (0, 1, 2 ou rien)
  res=raw_input("Evenement -> ")
  evenement="VIDE"
  if res=="1": evenement="BOUTON_1"
  if res=="2": evenement="BOUTON_2"
  if res=="0": break

  if evenement in transitions[etat]:
    etat=transitions[etat][evenement]
  if etat in sorties:
    sorties[etat]()
```

```
void setup() {
  etat=DEPART;
  evenement=VIDE;
}
void loop() {
  // lecture des broches d'entrees
  bouton_1_prec=bouton_1;
  bouton_2_prec=bouton_2;
  bouton_1=digitalRead(2);
  bouton_2=digitalRead(3);
  // scrutation d'un front montant
  // pour identifier les evenements
  evenement=VIDE;
  if (bouton_1 && !bouton_1_prec) {
    evenement=BOUTON_1; }
  if (bouton_2 && !bouton_2_prec) {
    evenement=BOUTON_2; }

  etat=transitions[etat][evenement];
  sorties[etat]();
  delay(200);
}
```

Pour conclure, la mise en oeuvre d'une machine d'état simple dans un langage procédurale est tout à fait possible en organisant bien les données et les fonctions.

Pour des machines d'états élaborées, un outil graphique avec génération du code est indispensable. Néanmoins, l'interfaçage avec le matériel nécessite bien souvent de retoucher le code produit, ce qui ne peut se faire qu'avec une bonne compréhension du principe de fonctionnement de la machine d'états.