Qt - Module IHM
TP n°6 - Machine à états

par Thierry Vaira © v.1.0

Sommaire

Les TP d'acquisition des fondamentaux visent à construire un socle de connaissances de base, à appréhender un concept, des notions et des modèles qui sont fondamentaux. Ce sont des étapes indispensables pour aborder d'autres apprentissages. Les TP sont conduits de manière fortement guidée pour vous placer le plus souvent dans une situation de découverte et d'apprentissage.

Les objectifs de ce tp sont de découvrir la programmation d'un automate fini (ou machine à états finis) sous Qt.

Automate fini ou machine à états finis

Présentation

Un **automate fini** ou **machine à états finis** (*finite state machine*) est un modèle mathématique de calcul utilisé dans de nombreux domaines (conception de programmes informatiques, protocoles de communication, contrôle des processus, analyse linguistique, ...).

Lire : fr.wikipedia.org/wiki/Automate_fini

Un automate fini est susceptible d'être :

- dans un nombre fini d'états,
- dans un seul état à la fois.

L'état où il se trouve est appelé l'« **état courant** ». Le passage d'un état à un autre est dirigé par un **évènement** (ou une condition) appelé une « **transition** ».

Un automate sera défini par la liste de ses états et par les conditions des transitions.

On rencontre couramment des automates finis dans de nombreux appareils qui réalisent des actions déterminées en fonction des évènements qui se présentent.

Exemples : Un exemple est un distributeur automatique de boissons qui délivre l'article souhaité quand le montant introduit est approprié, un autre les ascenseurs qui savent combiner les appels successifs pour s'arrêter aux étages intermédiaires, ou les feux de circulation capables de s'adapter aux voitures en attente, ou encore des digicodes qui analysent la bonne suite de chiffres, ou aussi la gestion de menu des IHM qui s'affichent en fonction des choix de l'utilisateur.

Terminologie

Un **état** est la description de la configuration d'un système en attente d'exécuter une transition.

Une **transition** est un ensemble d'actions à exécuter lorsqu'une condition est remplie ou lorsqu'un évènement est reçu.

Exemple : une chaîne audio peut être dans l'état « CD » et recevoir l'évènement « suivant ». Elle passe alors à la piste suivante du CD.

Dans certaines représentations de machines finies, il est possible d'associer des actions à un état :

- action d'entrée : réalisée lorsque l'on « entre » dans l'état,
- action de sortie : réalisée lorsque l'on « quitte » l'état.
- action de transition : réalisée lors d'une transition

Exemple : le portillon d'accès

Un exemple très simple d'un mécanisme que l'on peut modéliser par un automate fini est un **portillon d'accès**.

Un portillon, utilisé dans certains métros ou dans d'autres établissements à accès contrôlés est une barrière avec trois bras rotatifs à hauteur de la taille. Au début, les bras sont verrouillés et bloquent

l'entrée, et empêchent les clients de passer. L'introduction d'une pièce de monnaie (ou d'un jeton dans une fente du portillon ou la présentation d'un ticket ou d'une carte) débloque les bras et permet le passage d'un et un seul usager à la fois. Une fois le client entré, les bras sont à nouveaux bloqués jusqu'à ce qu'un nouveau jeton est inséré.

Un portillon d'accès

Un portillon peut être vu comme un automate fini à deux états : **verrouillé** (« *locked* ») et **déverrouillé** (« *unlocked* »).

Deux "entrées" peuvent modifier l'état : la première si l'on insère un jeton dans la fente (entrée **jeton**) et la deuxième si l'on pousse le bras (entrée **pousser**).

Dans l'état verrouillé, l'action de pousser n'a aucun effet : quel que soit le nombre de fois que l'on pousse, l'automate reste verrouillé. Si l'on insère un jeton, c'est-à-dire si l'on effectue une "entrée" jeton, on passe de l'état verrouillé à l'état déverrouillé. Dans l'état déverrouillé, ajouter des jetons supplémentaires n'a pas d'effet, et ne change pas l'état. Mais dès qu'un usager tourne le bras du portillon, donc fournit un pousser, la machine retourne à l'état verrouillé.

État courant	**Entrée**	**État suivant**	**Sortie**
verrouillé	jeton	déverrouillé	Déverrouille le portillon pour qu'un usager puisse passer
verrouillé	pousser	verrouillé	Rien
déverrouillé	jeton	déverrouillé	Rien
déverrouillé	pousser	verrouillé	Quand l'usager est passé, verrouille le portillon

Graphe orienté

On peut représenter l'automate par un **graphe orienté**.

Chaque état est représenté par un **sommet** (visualisé par un cercle). Les **arcs** (représentés par des flèches) montrent les transitions d'un état à un autre. Chaque **flèche** porte une entrée qui déclenche la transition. Un **point noir** sert à indiquer que c'est état est l'état initial, au début de la modélisation.

Les données qui ne provoquent pas de changement d'état sont représentées par des arcs circulaires (des boucles) qui tournent autour de l'état.

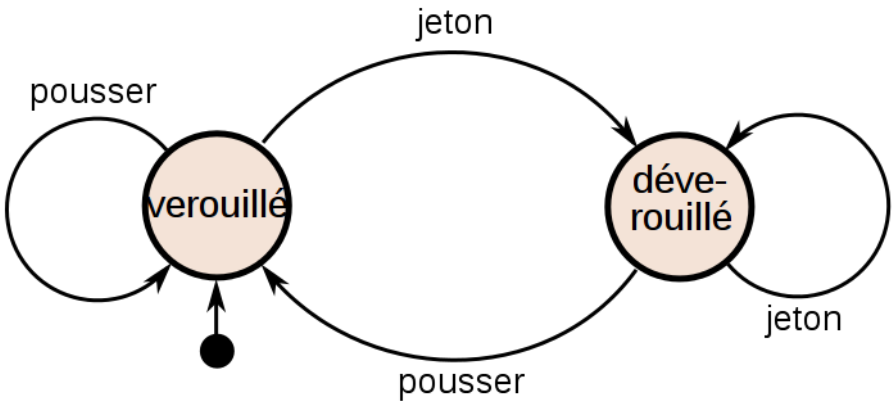


Table états-transitions

Plusieurs types de tables de transition d'état sont utilisées. Ces diagrammes sont très populaires en **UML** notamment (cf. diagramme états-transitions).

La représentation la plus courante est illustrée ci-dessous :

État / Entrée	jeton	pousser	bouton init	bouton arrêt
initial	déverrouillé	verrouillé	initial	final
verrouillé	déverrouillé	verrouillé	initial	final
déverrouillé	déverrouillé	verrouillé	initial	final
final	final	final	final	final

Explication : La combinaison de l'état courant (par exemple « verrouillé ») et d'une entrée (par exemple « jeton ») montre l'état suivant (dans l'exemple « déverrouillé »).

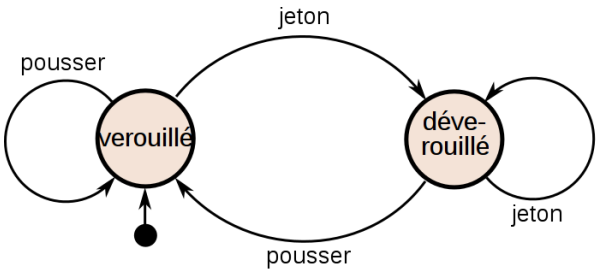
L'information complète associée à une action n'est pas décrite dans la table et peut être ajoutée par des annotations.

Codage sous Qt

Sous Qt, la classe `QStateMachine` fournit une machine à états finis.

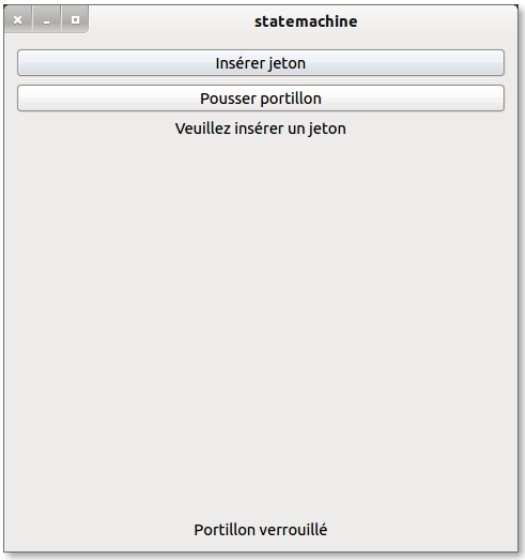
Une machine à états gère un ensemble d'**états** (classes qui héritent de `QAbstractState`) et des **transitions** (descendants de `QAbstractTransition`) entre ces états. Une fois qu'un graphe orienté a été construit, la machine à états pourra l'exécuter. L'algorithme d'exécution de `QStateMachine` est basé sur l'algorithme *State Chart XML* (SCXML).

On reprend l'exemple du portillon d'accès (cf. page 2) :



On va réaliser une GUI pour tester cet exemple :

```
// La GUI
btInsererJeton = new QPushButton(QString::fromUtf8("Insérer jeton"), this);
btPousser = new QPushButton(QString::fromUtf8("Pousser portillon"), this);
labelUtilisateur = new QLabel(this);
labelUtilisateur->setAlignment(Qt::AlignCenter);
labelPortillon = new QLabel(this);
labelPortillon->setAlignment(Qt::AlignCenter);
QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(btInsererJeton);
layout->addWidget(btPousser);
layout->addWidget(labelUtilisateur);
layout->addStretch();
layout->addWidget(labelPortillon);
setLayout(layout);
```



On commence par créer la **machine à états** :

```
// La machine à état
QStateMachine *machine = new QStateMachine();
```

On peut maintenant définir les **états**. On utilise la méthode `addState()` pour ajouter un état à la machine à états. Les états sont supprimés avec `removeState()` (la suppression des états pendant la mise en marche de la machine est déconseillée).

```
// Les états (voir graphe)
QState *etatVerrouille = new QState();
etatVerrouille->assignProperty(labelPortillon, "text", QString::fromUtf8("Portillon
    verrouillé"));
QState *etatDeverrouille = new QState();
etatDeverrouille->assignProperty(labelPortillon, "text", QString::fromUtf8("Portillon
    déverrouillé"));

machine->addState(etatVerrouille);
machine->addState(etatDeverrouille);
```

Remarque : Les états seront simplement visualisés par un ***text*** *dans un* ***QLabel****.*

On ajoute ensuite les **transitions** :

```
// Les transitions (voir graphe)
etatVerrouille->addTransition(btInsererJeton, SIGNAL(clicked()), etatDeverrouille);
etatDeverrouille->addTransition(btPousser, SIGNAL(clicked()), etatVerrouille);
```

Il est possible d'associer des **actions** à un état :

- action d'entrée : réalisée lorsque l'on « entre » dans l'état (signal `entered()`)
- action de sortie : réalisée lorsque l'on « quitte » l'état (signal `exited()`)

```
// Les actions
connect(etatVerrouille, SIGNAL(entered()), this, SLOT(afficherInviteInsererJeton()));
connect(etatVerrouille, SIGNAL(exited()), this, SLOT(afficherInvitePassage()));
```

Avant de démarrer la machine, l'état initial doit être réglé avec `setInitialState()`. L'état initial est l'état d'entrée de la machine au démarrage. On peut ensuite démarrer la machine à états avec `start()`. Le signal `started()` est émis lorsque on entre dans l'état initial.

```
machine->setInitialState(etatVerrouille);
machine->start();
```

Remarque : La machine à états émet le signal ***finished()*** *lorsqu'on entre dans l'état final. On peut également arrêter la machine à états avec* ***stop()*** *et elle émet alorts le signal* ***stopped()*** *dans ce cas.*

Télécharger le code source fourni : http ://tvaira.free.fr/info1/qt-portillon.zip

Voir aussi en Annexe : un exemple de codage en C page 8.

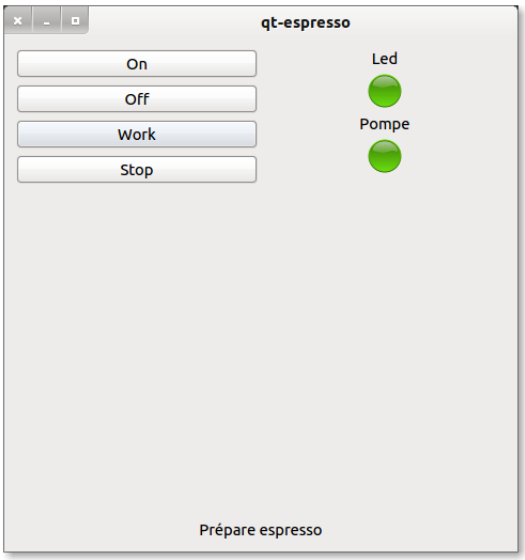
Travail demandé

Chaque jour, nous sommes tous confrontés à des appareils et machines qui peuvent être considérés comme des automates parmi lesquels on trouve les distributeurs automatiques de boissons ou de billets de banque, les machines à laver et tant d'autres.

Vous devez développer une **machine à café espresso** pouvant se trouver dans trois états différents : elle peut être éteinte (*OFF*), allumée et en attente (*STANDBY*) ou en train de pomper et chauffer de l'eau pour préparer un espresso (*WORKING*). L'interface de contrôle de la machine comporte quatre boutons : *turnOn*, *turnOff*, *work*, et *stop*.

Itération 1. Représenter l'automate par un graphe orienté.

Itération 2. Coder l'automate sous Qt.



Annexe : code source en C

On commence par déclarer les **entrées** :

```
// Les entrées
#define ENTREE_JETON         0
#define ENTREE_POUSSER       1
#define ENTREE_INIT          2
#define ENTREE_ARRETER       3
#define ENTREE_INVALIDE     -1
```

Puis pour gérer les **états**, on traduira la table états-transitions précédentes par un tableau à deux dimensions en C/C++ :

```
// Les différents états de l'automate
#define ETAT_INITIAL     0
#define ETAT_VERROUILLE  1
#define ETAT_DEVERROUILLE 2
#define ETAT_FINAL       3
#define ETAT_RIEN       -1

// Table des transitions
int transitions[4][4] = { ETAT_DEVERROUILLE, ETAT_VERROUILLE, ETAT_INITIAL,    ETAT_FINAL
                         ,ETAT_DEVERROUILLE, ETAT_VERROUILLE, ETAT_INITIAL,    ETAT_FINAL
                         ,ETAT_DEVERROUILLE, ETAT_VERROUILLE, ETAT_INITIAL,    ETAT_FINAL
                         ,ETAT_FINAL,        ETAT_FINAL,      ETAT_FINAL,      ETAT_FINAL
                       };
```

Il est possible maintenant de créer une table pour les **actions** sur le même modèle que le tableau précédent :

```
// Les actions
#define INITIALISER    0
#define VERROUILLER    1
#define DEVERROUILLER  2
#define ARRETER        3
#define AUCUNE        -1

// Table des actions
int actions[4][4]   = {  DEVERROUILLER, VERROUILLER, INITIALISER, ARRETER
                        ,DEVERROUILLER, AUCUNE,      INITIALISER, ARRETER
                        ,AUCUNE,        VERROUILLER, INITIALISER, ARRETER
                        ,AUCUNE,        AUCUNE,       AUCUNE,      AUCUNE
                       };
```

Remarque : il a été décidé ici de ré-initialiser le système à chaque fois que l'on appuie sur le bouton init.

Il suffit ensuite de gérer l'**automate** en créant un « **interpréteur de table** » :

```
Initialiser l'état courant
TANT QUE l'etat final n'est pas atteint
   Lire l'entrée
   Executer l'action associée
   Changer d'état
FINT TANT QUE
```

Soit en C/C++ :

```
// Initialiser l'état courant
etat = ETAT_INITIAL;

// TANT QUE l'état final n'est pas atteint
while (etat != ETAT_FINAL)
{
     // Lire l'entrée
     entree = lireEntree();

     if(entree != ENTREE_INVALIDE)
     {
        // Executer l'action associée
        executer(actions[etat][entree]);

        // Changer d'état
        etat = transitions[etat][entree];
     }
}
```

```
//============================================================
// Programme : portillon.c
//
// Role : Exemple simple d'automate fini
//
//============================================================

#include <stdio.h>

// Les différents états de l'automate
#define ETAT_INITIAL     0
#define ETAT_VERROUILLE  1
#define ETAT_DEVERROUILLE 2
#define ETAT_FINAL       3
#define ETAT_RIEN       -1

// Les actions
#define ACTION_INITIALISER   0
#define ACTION_VERROUILLER   1
#define ACTION_DEVERROUILLER 2
#define ACTION_ARRETER       3
#define ACTION_AUCUNE       -1

// Les entrées
#define ENTREE_JETON         0
#define ENTREE_POUSSER       1
#define ENTREE_INIT          2
#define ENTREE_ARRETER       3
#define ENTREE_INVALIDE     -1

// Fonctions de gestion de l'automate
int  lireEntree();
void executer(int action);
```

```
// Les actions
void initialiser();
void deverrouiller();
void verrouiller();
void arreter();

// Débuggage
void afficherEtat(int etat);

// Programme principal
int main()
{
   int etat = ETAT_INITIAL; // état courant
   int entree;            // l'évènement d'entrée

   // Table des transitions
   int transitions[4][4] = { ETAT_DEVERROUILLE, ETAT_VERROUILLE, ETAT_INITIAL,
      ETAT_FINAL
                           ,ETAT_DEVERROUILLE,  ETAT_VERROUILLE,  ETAT_INITIAL,
                              ETAT_FINAL
                           ,ETAT_DEVERROUILLE,  ETAT_VERROUILLE,  ETAT_INITIAL,
                              ETAT_FINAL
                           ,ETAT_FINAL,         ETAT_FINAL,       ETAT_FINAL,
                              ETAT_FINAL
                          };
   // Table des actions
   int actions[4][4]   = {  ACTION_DEVERROUILLER, ACTION_VERROUILLER, ACTION_INITIALISER,
      ACTION_ARRETER
                           ,ACTION_DEVERROUILLER, ACTION_AUCUNE,  ACTION_INITIALISER,
                              ACTION_ARRETER
                           ,ACTION_AUCUNE,      ACTION_VERROUILLER, ACTION_INITIALISER,
                              ACTION_ARRETER
                           ,ACTION_AUCUNE,      ACTION_AUCUNE,    ACTION_AUCUNE,
                              ACTION_AUCUNE
                          };

   printf("Gestion PORTILLON\n\n");

   // démarrage du système
   executer(actions[etat][ENTREE_INIT]);
   etat = transitions[etat][ENTREE_INIT];
   afficherEtat(etat);

   // tant que l'etat final n'est pas atteint
   while (etat != ETAT_FINAL)
   {
         // Lecture de l'entrée
         entree = lireEntree();
         if(entree != ENTREE_INVALIDE)
         {
            // Action associée
            executer(actions[etat][entree]);
            // Changement d'etat
```

```
            etat = transitions[etat][entree];
        }
        // Affichage de la transition (débuggage)
        afficherEtat(etat);
   }

   return 0;
}

int lireEntree()
{
   char saisieEntree;
   int  entree = ENTREE_INVALIDE; // invalide

   printf("0 - Initialiser\n");
   printf("1 - Jeton\n");
   printf("2 - Pousser\n");
   printf("3 - Arreter\n");

   // on avale les sauts de ligne
   do
      saisieEntree = fgetc(stdin);
   while((saisieEntree == 0x0a) || (saisieEntree == 0x0d));

   if (saisieEntree == '0')     entree = ENTREE_INIT;
   else if (saisieEntree == '1') entree = ENTREE_JETON;
   else if (saisieEntree == '2') entree = ENTREE_POUSSER;
   else if (saisieEntree == '3') entree = ENTREE_ARRETER;
   else                          entree = ENTREE_INVALIDE;

   return entree;
}

void executer(int action)
{
   switch (action)
   {
   // Initialisation du portillon
   case ACTION_INITIALISER : initialiser();
           break;
   // Déverrouiller
   case ACTION_DEVERROUILLER : deverrouiller();
           break;
   // Verrouiller
   case ACTION_VERROUILLER : verrouiller();
           break;
   // Arreter
   case ACTION_ARRETER :     arreter();
           break;
   }
}

// Les actions (TODO)
```

```
void initialiser()
{
    printf("-> action : %s\n", __FUNCTION__);
}

void deverrouiller()
{
    printf("-> action : %s\n", __FUNCTION__);
}

void verrouiller()
{
    printf("-> action : %s\n", __FUNCTION__);
}

void arreter()
{
    printf("-> action : %s\n", __FUNCTION__);
}

void afficherEtat(int etat)
{
    switch (etat)
   {
   // Initialisation du portillon
   case ETAT_INITIAL :    printf("Etat : INITIALISE\n");
           break;
   // Déverrouiller
   case ETAT_DEVERROUILLE : printf("Etat : DEVERROUILLE\n");
           break;
   // Verrouiller
   case ETAT_VERROUILLE : printf("Etat : VERROUILLE\n");
           break;
   // Arreter
   case ETAT_FINAL :      printf("Etat : ARRET\n");
           break;
   }
   printf("\n");
}
```