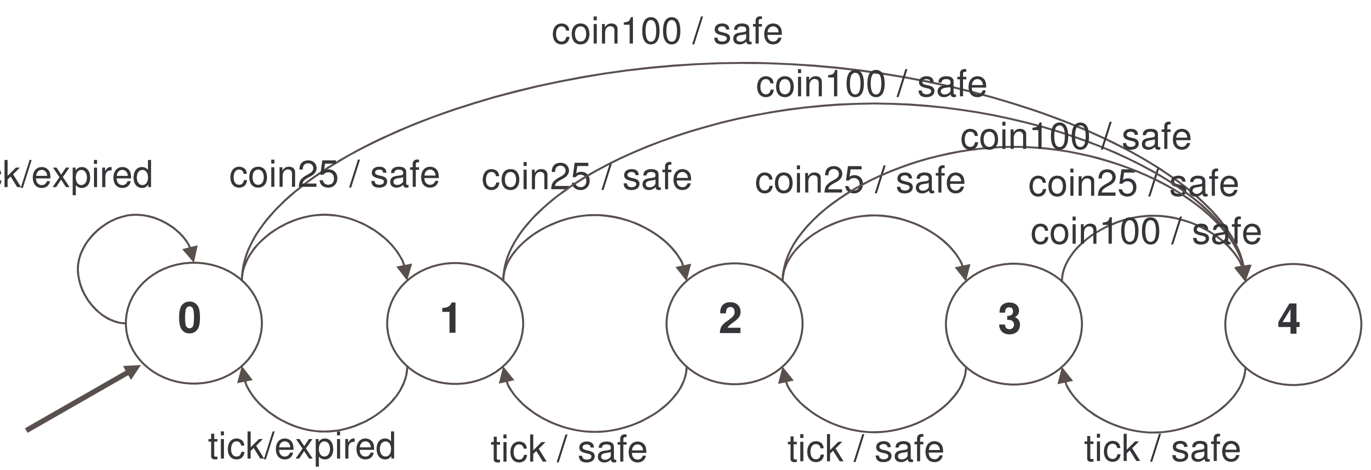
Implementation of FSM

Implementing (hierarchical) FSMs in C++/C

From Practical Statecharts in C/C++ by Miro Samek, CMPBooks

Implementation refers to the simple parking mater example (modified from the one in Lee-Varaiya book)



A C implementation of (some) OO programming

We will focus on a C implementation that provides support for

- Abstraction *joining data and functions operating on them, defining which functions are for public use (interface) and which for private use*
- Inheritance *defining new classes based on existing classes*
- Polymorphism *substituting objects with matching interfaces at run-time*

This is done by using a set of conventions and idioms

A C implementation of (some) OO programming

The Approach:
think of the `FILE` structure in ANSII C and of file-related ops (`open, close` ...)

- Attributes of the class are defined with a C struct
- Methods of the class are defined as C functions. Each function takes a pointer to the attribute structure as an argument
- Special methods initialize and clean up the attribute structure

Abstraction

An example: pseudo-class Hsm (Hierarchical state machine)

```
typedef struct Hsm Hsm;
struct Hsm {
   State state_;
   State source_;
};

Hsm *HsmCtor_(Hsm *me, PState initial);
void HsmXtor_(Hsm *me);

void HsmInit(Hsm *me);
void HsmDispatch(Hsm *me, Event const *e);
void HsmTran_(Hsm *me);

State Hsm_top(Hsm *me, Event const *e);
```

A C implementation of (some) OO programming

Constraints on constructors and destructors

```
struct Hsm {
...
}
```

A C implementation of (some) OO programming

Some helper macros

```
#define HsmGetState(me_) ((me_)->state_)

#define CLASS(class_) typedef struct class_ class_;\
                      struct class_ {
#define METHODS };
#define END_CLASS
```

A C implementation of (some) OO programming

allows writing

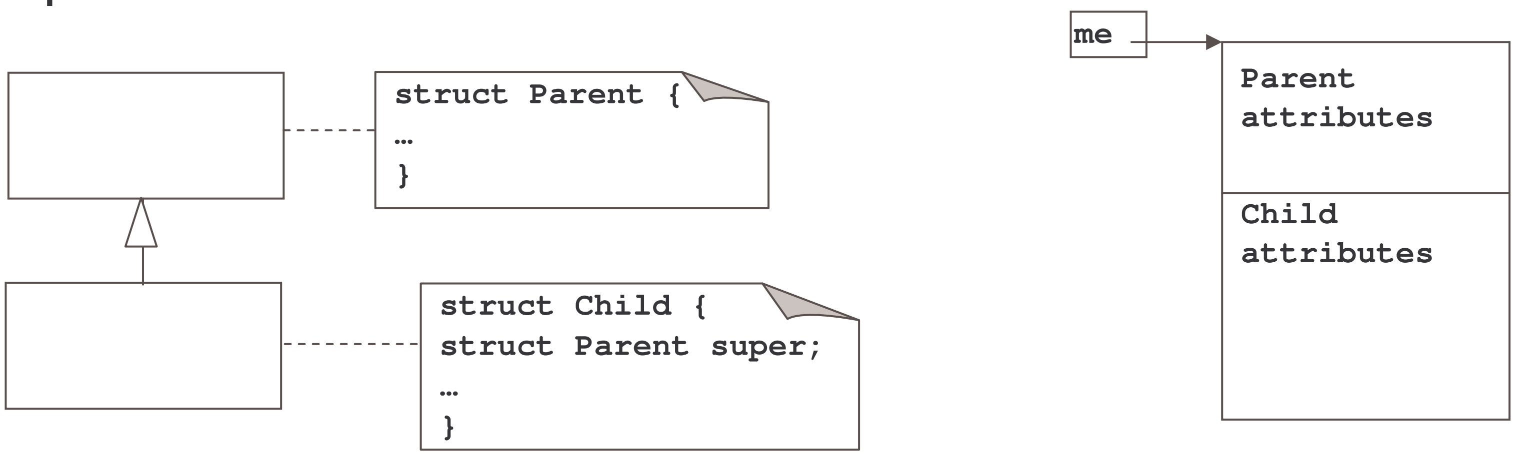
```
CLASS(Hsm)
   State state_;
   State source_;
METHODS
   Hsm *HsmCtor_(Hsm *me, PState initial);
   void HsmXtor_(Hsm *me);

   void HsmInit(Hsm *me);
   void HsmDispatch(Hsm *me, Event const *e);
   void HsmTran_(Hsm *me);

   State Hsm_top(Hsm *me, Event const *e);
END_CLASS
```

Inheritance

Extension by adding attributes and methods

(overriding not considered at this time)

- Inheritance can be implemented in a number of ways
- Single inheritance can be obtained by embedding the parent into the child



You can pass the child pointer to any function that expects a pointer to the Parent class (you should explicitly upcast the pointer)

A C implementation of (some) OO programming

Inheritance

An example: child of pseudo-class Hsm

```
struct ChildHsm {
struct Hsm super;
...
}
```

Uses

```
me->super->method

((Hsm *)me)->method
```

A C implementation of (some) OO programming

Constraints on constructors and destructors

```
struct ChildHsm {
struct Hsm super;
...
}
```

A C implementation of (some) OO programming

The macro

```
#define SUBCLASS(class_, superclass_) \
   CLASS(class_)                      \
      superclass_ super_;
```

allows writing

```
SUBCLASS(ChildHsm, Hsm)
METHODS
   void ChildHsmAdditional_(ChildHsm *me);
END_CLASS
```

Design options

- Now back at our original objective …
- Encoding FSMs in C++ and C

Design options

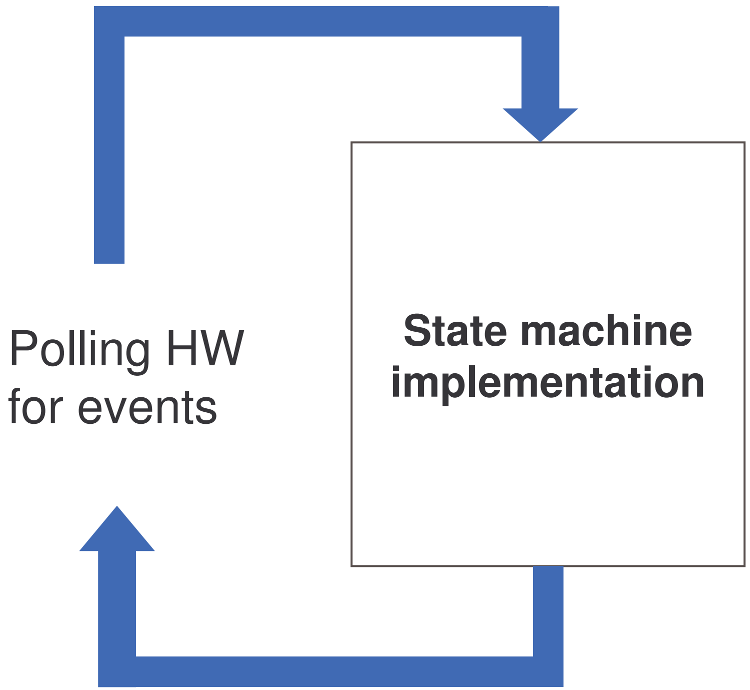
- Design decisions and trade-offs
  - How do you represent events? How about events with parameters?
  - How do you represent states?
  - How do you represent transitions?
  - How do you dispatch events to the state machine?
- When you add state hierarchy, exit/entry actions and transitions with guards, the design can become quite complex
- We are going ot deal with standard (i.e. not hierarchical) state machines

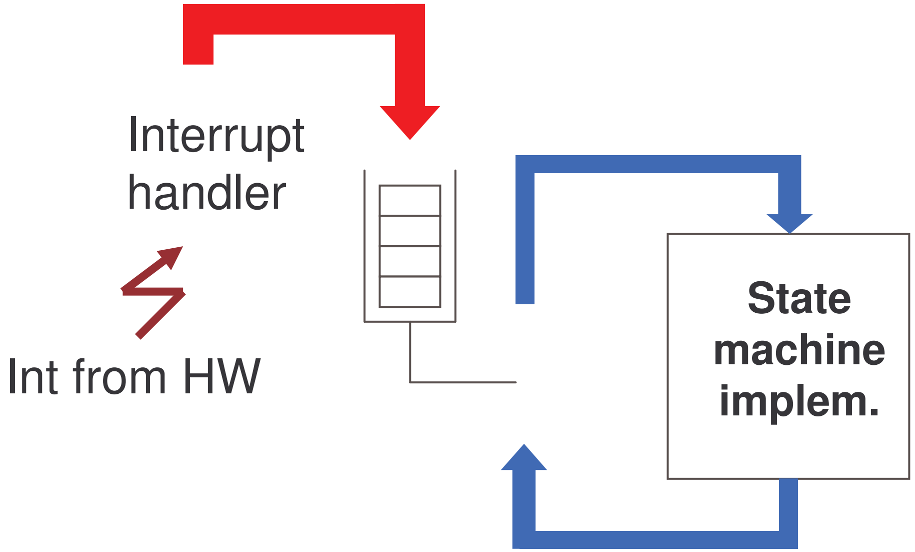
Typical implementations

- Typical implementations in the C or C++ language include
  - The nested switch statement
  - The state table
  - The object-oriented State design pattern and
  - Combinations of the previous

Typical implementations

- State machine implementations are typically coupled with a concurrency model and an event dispatching policy





Typical implementations

- Interface consisting of three methods
  - `init()` takes a top-level initial transition
  - `dispatch()` to dispatch an event to the state machine
  - `tran()` to make an arbitrary transition

Typical implementations

- Nested switch statement
- Perhaps the most popular technique
- 2 levels of switch statements
- 1st level controlled by a scalar state variable
- 2nd level controlled by an event signal variable

Nested switch implementation

Signals and states are typically represented as enumerations

```
enum Signal {
  SIGNAL_1, SIGNAL_2, SIGNAL_3, ...
};
enum State {
  STATE_X, STATE_Y, STATE_Z, ...
};

void init() {}
void dispatch(unsigned const sig) {}
void tran(State target)
```

Nested switch implementation

C++ (class based) implementation

```
class Hsm1 {
  private:
      State myState;
      ...
  public:
      void init();
      void dispatch(unsigned const sig);
      void tran(State target);
      ...
 }
```

Each instance tracks its own state

Nested switch implementation

```
void dispatch(unsigned const sig) {
   switch(myState) {
   case STATE_1:
        switch(sig) {
                case SIGNAL_1:
                        tran(STATE_X)
                        ...
                        break;
                case SIGNAL_2:
                        tran(STATE_Y)
                        ...
                        break;
        }
        break;
   case STATE_2:
        switch(sig) {
                case SIGNAL_1:
                ...
                break;
                ...
        }
        break;
   ...
}
```

Nested switch impl.: variations

Breaking up the event handler code by moving the second (signal) level into a specialized state handler function

```
void dispatch(unsigned const sig) {
    switch(myState) {
    case STATE_1:
         ManageState1(sig);
         break;
    case STATE_2:
         ManageState1(sig);
         break;
    ...
}
```

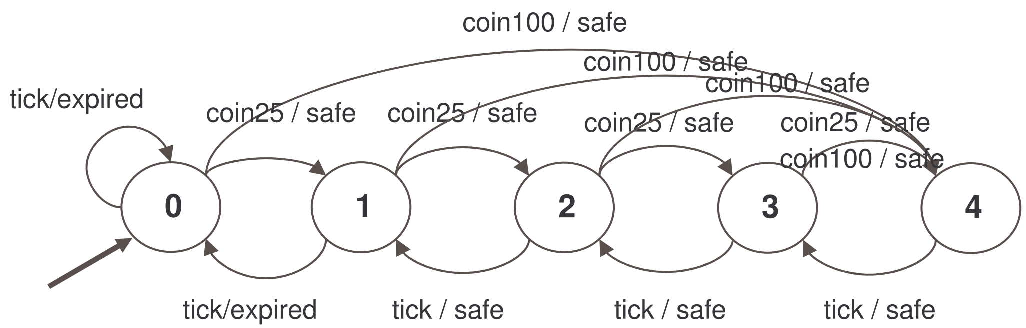
Nested switch method

The nested switch statement method:

- Is simple
- Requires enumerating states and triggers
- Has a small (RAM) memory footprint
  - 1 scalar variable required
- Does not promote code reuse
- Event dispatching time is not constant
  - Increases with the number of cases O(log n)
- Implementation is not hierarchical and manual coded entry/exit actions are prone to error and difficult to maintain against changes in the state machine. The code pertaining to one state (entry action) is distributed and repeated in many places (on every transition leading to that state)
  - This is not a problem for automatic synthesis tools

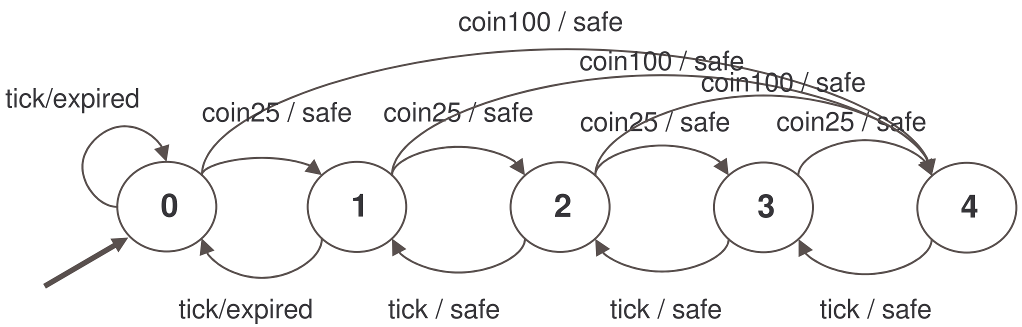
The example ....

```
enum Signal {
   TICK, COIN25, COIN100
};
enum State {
   S_0, S_1, S_2, S_3, S_4
};

enum Display {
   EXPIRED, SAFE
};
```



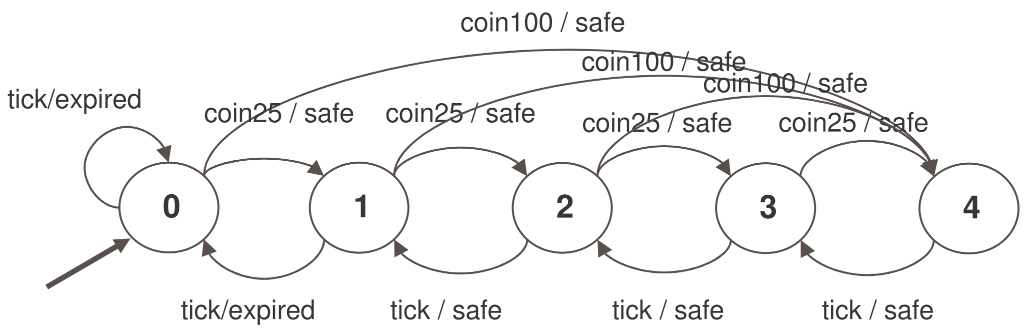
The example ….

```
CLASS(PMeter)
   State state_;
METHODS
   void PMeterInit(PMeter *me);
   void PMeterDispatch(PMeter *me, Signal const *e);
   void PMeterTran_(PMeter *me, PMeter dest);

   void PMeterShow(Display d);
END_CLASS
```



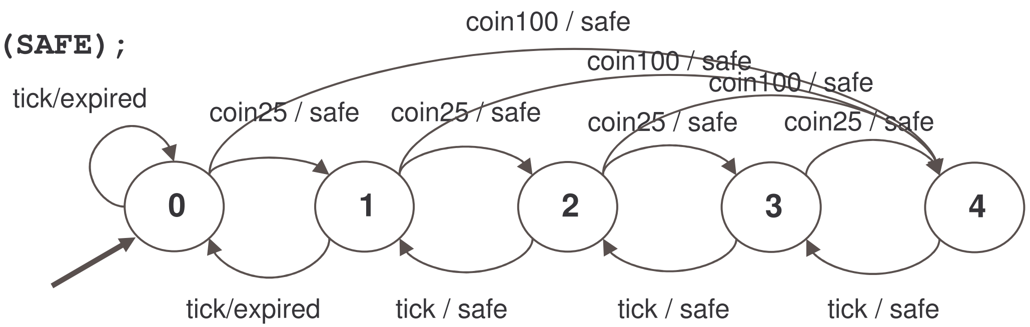
The example ….

```
void PMeterInit(PMeter *me)
{
  me->state_ = S_0;
}

void PMeterTran_(PMeter *me, PMeter dest)
{
  me->state_ = dest;
}
```



The example ….

```
void PMeterDispatch(PMeter *me, Signal const *s)
{
    switch(me->state_) {
    case S_0:
          switch(sig) {
                   case COIN25:
                            PMeterShow(SAFE);
                            tran(S_1)
                            break;
                   case COIN100:
                            PMeterShow(SAFE);
                            tran(S_4)
                            break;
          }
          break;
    case S_1:
          switch(sig) {
                   case TICK:
                            PMeterShow(EXPIRED);
                            tran(S_0)
                            break;
                   case COIN25:
                            tran(S_2)
                            break;
                   case COIN100:
                            tran(S_4)
                            break;
          }
          break;
```



The State Table approach

State tables containing arrays of transitions for each state

Signals→

States↓

	SIGNAL_1	SIGNAL_2	SIGNAL_3	SIGNAL_4
STATE_X				
STATE_Y				
STATE_Z	action1() STATEX			
STATE_A				

The content of the cells are transitions, represented as pairs {action, next state}

The class StateTable

```
class StateTable {
public:
    typedef void (StateTable::*Action)();
    struct Tran {
         Action action;
         unsigned nextState;
    };
    StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
    : myTable(table) myNsignals(nSignals), myNstates(nStates) {}
    virtual ~StateTable(){}
    void dispatch(unsigned const sig) {
         register Tran const *t = myTable + myState*myNsignals + sig;
         (this->*(t->action))();
         myState = t->nextState;
    }
    void doNothing() {}
protected:
    unsigned myState;
private:
    Tran const *myTable;
    unsigned myNsignals;
    unsigned myNstates;
};
```

type Action is a pointer to a member function of StateTable (*or a subclass*)

The class StateTable

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
        Action action;
        unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
   : myTable(table) myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable(){}
   void dispatch(unsigned const sig) {
        register Tran const *t = myTable + myState*myNsignals + sig;
        (this->*(t->action))();
        myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```

type Tran is the type of the table cell

The class StateTable

```
class StateTable {
public:
   typedef void (StateTable::*Action)();
   struct Tran {
        Action action;
        unsigned nextState;
   };
   StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
   : myTable(table) myNsignals(nSignals), myNstates(nStates) {}
   virtual ~StateTable(){}
   void dispatch(unsigned const sig) {
        register Tran const *t = myTable + myState*myNsignals + sig;
        (this->*(t->action))();
        myState = t->nextState;
   }
   void doNothing() {}
protected:
   unsigned myState;
private:
   Tran const *myTable;
   unsigned myNsignals;
   unsigned myNstates;
};
```

(initialization list parameter) constructor and destructor

The class StateTable

```
class StateTable {
public:
    typedef void (StateTable::*Action)();
    struct Tran {
         Action action;
         unsigned nextState;
    };
    StateTable(Tran const *table, unsigned nStates, unsigned nSignals)
    : myTable(table) myNsignals(nSignals), myNstates(nStates) {}
    virtual ~StateTable(){}
    void dispatch(unsigned const sig) {
         register Tran const *t = myTable + myState*myNsignals + sig;
         (this->*(t->action))();
         myState = t->nextState;
    }
    void doNothing() {}
protected:
    unsigned myState;
private:
    Tran const *myTable;
    unsigned myNsignals;
    unsigned myNstates;
};
```

(simple) dispatch function

Declaring an object, the events, states and table

```
Enum Event{
   SIGNAL1, SIGNAL2, ..., MAX_SIG
};

Enum State {
   STATE_X, STATE_Y, ..., MAX_STATE
};

class Hsm : public StateTable {
public:
   Hsm() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() {myState=STATE_X;}
   ...
private:
   void action1();
   void action2();
   ...
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   ...
};
```

Needed for detecting the array size

Declaring an object, the events, states and table

```
Enum Event{
   SIGNAL1, SIGNAL2, ..., MAX_SIG
};

Enum State {
   STATE_X, STATE_Y, ..., MAX_STATE
};

class Hsm : public StateTable {
public:
   Hsm() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() {myState=STATE_X;}
   ...
private:
   void action1();
   void action2();
   ...
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   ...
};
```

Initialize with the table and table size

Declaring an object, the events, states and table

```
Enum Event{
   SIGNAL1, SIGNAL2, ..., MAX_SIG
};

Enum State {
   STATE_X, STATE_Y, ..., MAX_STATE
};

class Hsm : public StateTable {
public:
   Hsm() : StateTable(&myTable[0][0], MAX_STATE, MAX_SIG) {}
   void init() {myState=STATE_X;}
   ...
private:
   void action1();
   void action2();
   ...
private:
   static StateTable::Tran const myTable[MAX_STATE][MAX_SIG];
   ...
};
```

myTable is a static constant table (one for all the objects crated from this class) with elements of type Tran

The state transition table

```
StateTable::Tran const Hsm::myTable[MAX_STATE][MAX_SIG] = {
    {{ &StateTable::doNothing, STATEX},
     { static_cast<StateTable::Action>(&Hsm::action2), STATEY},
     { static_cast<StateTable::Action>(&Hsm::action3), STATEX}},
    {{ static_cast<StateTable::Action>(&Hsm::action4), STATEZ},
     { &StateTable::doNothing, STATE_ERR},
     { static_cast<StateTable::Action>(&Hsm::action5), STATEZ}},
};
```

State Table implementation

Dispatch performs three steps:

- it identifies the transition to take as a state table lookup
- It executes the action
- it changes the state

```
void dispatch(unsigned const sig) {
   register Tran const *t = myTable + myState*myNsignals + sig;
   (this->*(t->action))();
   myState = t->nextState;
}
```

Typical implementations

The state table is divided into a generic and reusable processor part and an application-specific part

The application-specific part requires enumerating states and signals

Subclassing StateTable

Defining the action functions

Initializing the transition table

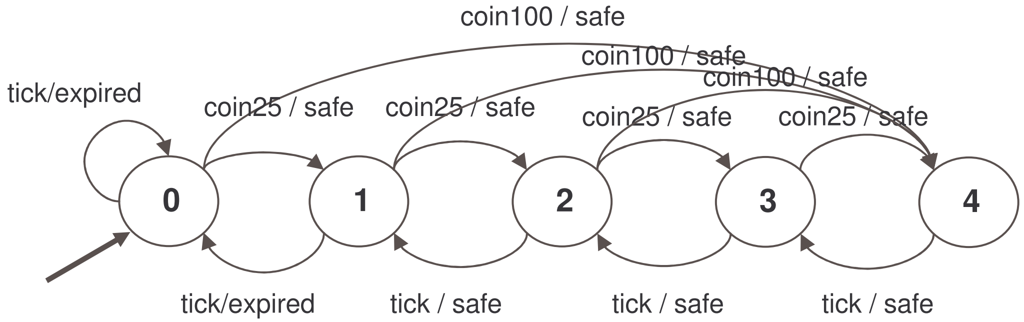
Typical implementations

The state table implementation has the following consequences

- it maps directly to the highly regular state table representation of a state machine
- it requires the enumeration of triggers and states
- It provides relatively good performance for event dispatching O(1)
- It promotes code reuse of the event processor
- It requires a large state table, which is typically sparse and wasteful. However, the table con be stored in ROM
- It requires a large number of fine grain functions representing actions
- It requires a complicated initialization
- It is not hierarchical
  - the state table can be extended to deal with state nesting, entry/exit actions and transition guards by hardcoding into transition actions functions

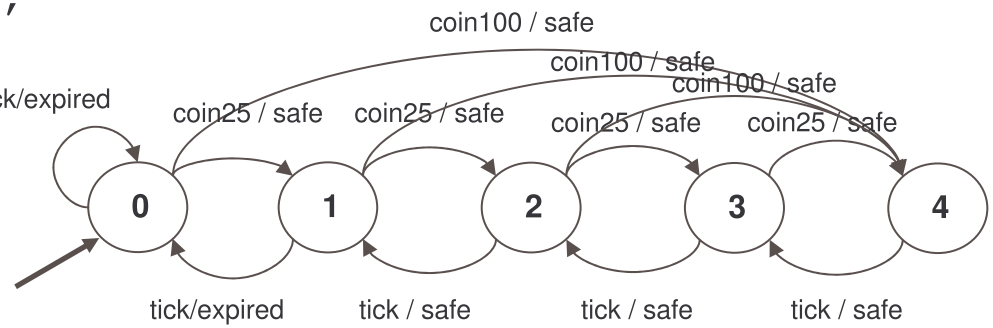
The example: basic types

```
typedef int (*Action)(StateTab *me);
Typedef struct Tran {
   Action action;
   unsigned nextState;
};

CLASS (StateTab)
   ...
METHODS
   ...
END_CLASS
```



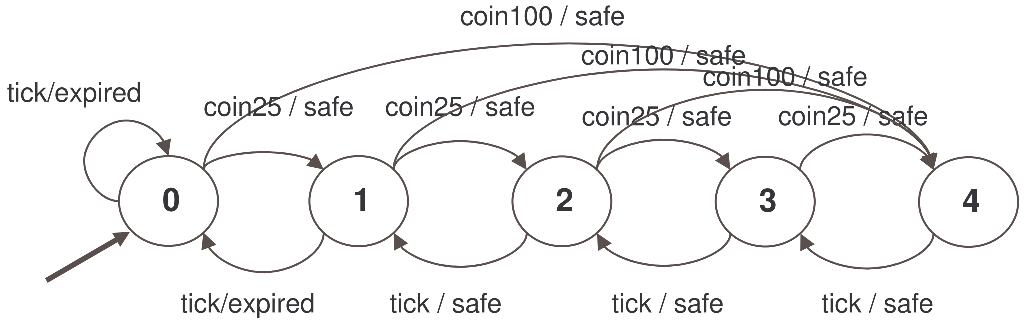
The Example: the State Table “class”

```
CLASS (StateTab)
   State myState_;
   Tran const *myTable__;
   unsigned myNsignals__;
   unsigned myNstates__;
METHODS
   StateTab *StateTabCTor(StateTab *me, Tran const *table,
   unsigned nStates, unsigned nSignals) {
        me->myTable__ = table;
        me->myNstates__ = nStates;
        me->myNsignals__ = nSignals;
   }
   void dispatch(StateTab *me, unsigned const sig) {
        Tran const *t = myTable__ + myState_*myNsignals__ + sig;
        t->action();
        myState_ = t->nextState;
   }
   void doNothing() {};
END_CLASS
```



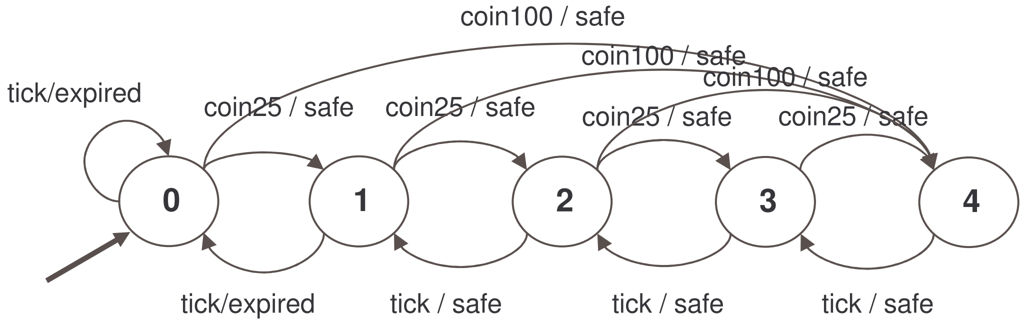
The example: preparing for PMeter

```
enum Signal {
   TICK, COIN25, COIN100
};
enum State {
   S_0, S_1, S_2, S_3, S_4
};
```



The Example: the PMeter "class"

```
SUBCLASS(PMeter, StateTab)
METHODS
   void PMeterCtor(PMeter *me) {
        StateTabCtor(me, &myTable[0][0], MAX_STATE, MAX_SIGNAL);
   }
   void PMeterinit(PMeter *me) {me->myState_ = S_0;};
   void PMeterShowSafe();
   void PMeterShowExpired();
END_CLASS
```



An example

```
Tran const myTable[MAX_STATE][MAX_SIGNAL] = {
   {{ &doNothing, S_0},
    { &PMeterShowSafe, S_1},
    { &PMeterShowSafe, S_4}},
   {{ &PMeterShowExpired, S_0},
    { &doNothing, S_2},
    { &doNothing, S_4}},
   {{ &doNothing, S_1},
    { &doNothing, S_3},
    { &doNothing, S_4}},
   {{ &doNothing, S_2},
    { &doNothing, S_4},
    { &doNothing, S_4}},
   {{ &doNothing, S_3},
    { &doNothing, S_4},
    { &doNothing, S_4}},
};
```