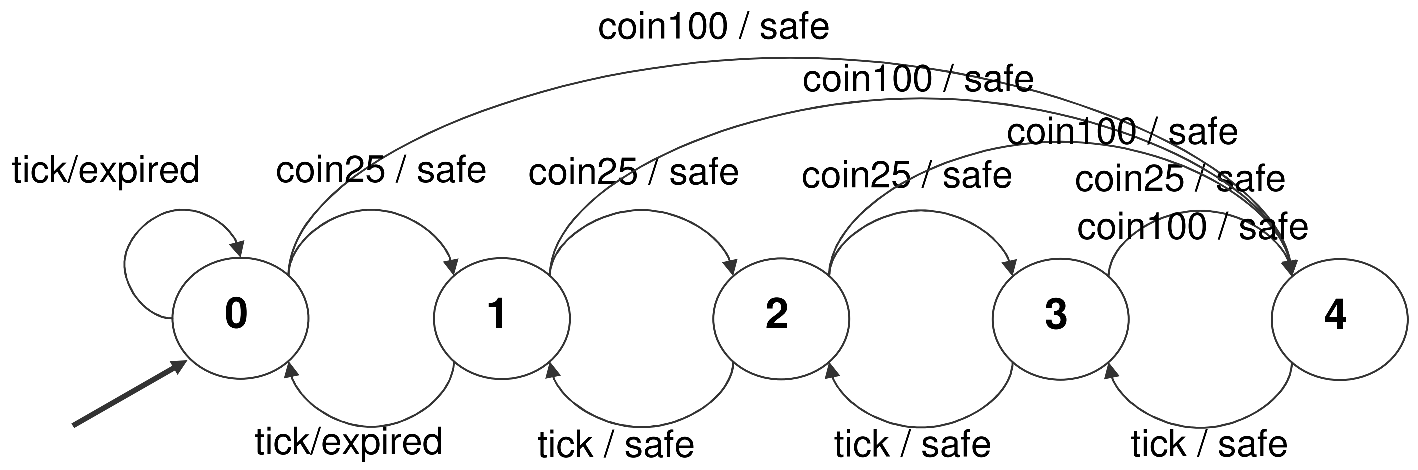
Implementation of FSM

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, State 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

Constructors and destructors

- The constructors take the me argument when they initialize preallocated memory, and return the pointer to the initialized object when the attribute structure can be initialized properly, or NULL when the initialization fails.
- The destructor takes only the me argument and returns void.

- As in C++, you can allocate objects statically, dynamically (on the heap), or automatically (on the stack).
- However, because of C syntax limitations,
  - Can't initialize objects at the definition point
  - For static objects, you can't invoke a constructor at all, because function calls aren't permitted in a static initializer.
  - Automatic objects must all be defined at the beginning of a block. At this point, you generally do not have enough initialization information to call the constructor; therefore, you often have to divorce object allocation from initialization.
  - It's a good programming practice to explicitly call destructors for all objects when they become obsolete or go out of scope.

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

allow writing a C-language pseudo-class like this

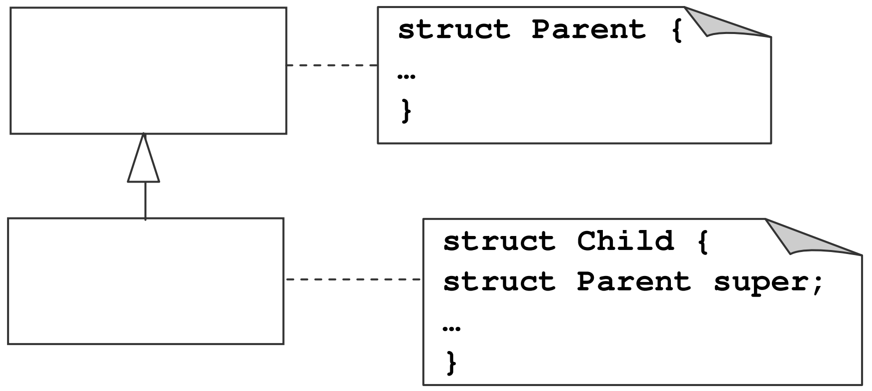
```
CLASS(Hsm)
   State state_;
   State source_;
METHODS
   Hsm *HsmCtor_(Hsm *me, PState initial);
   void HsmXtor_(Hsm *me);

   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
```

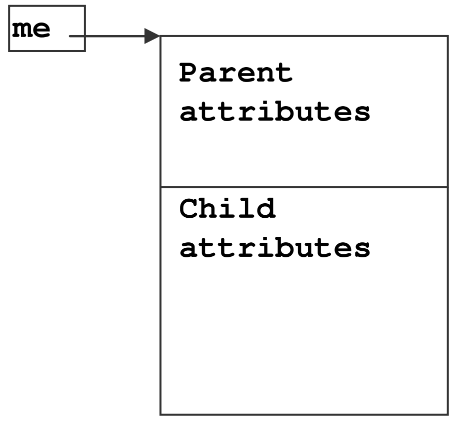
Inheritance

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

Constructors and destructors

- Inheritance adds responsibilities to class constructors and the destructor.
- The child constructor must initialize the portion controlled by the parent through an explicit call to the parent's constructor.
- To avoid potential dependencies, the superclass constructor should be called before initializing the attributes.
- Exactly the opposite holds true for the destructor. The inherited portion should be destroyed as the last step.

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
```

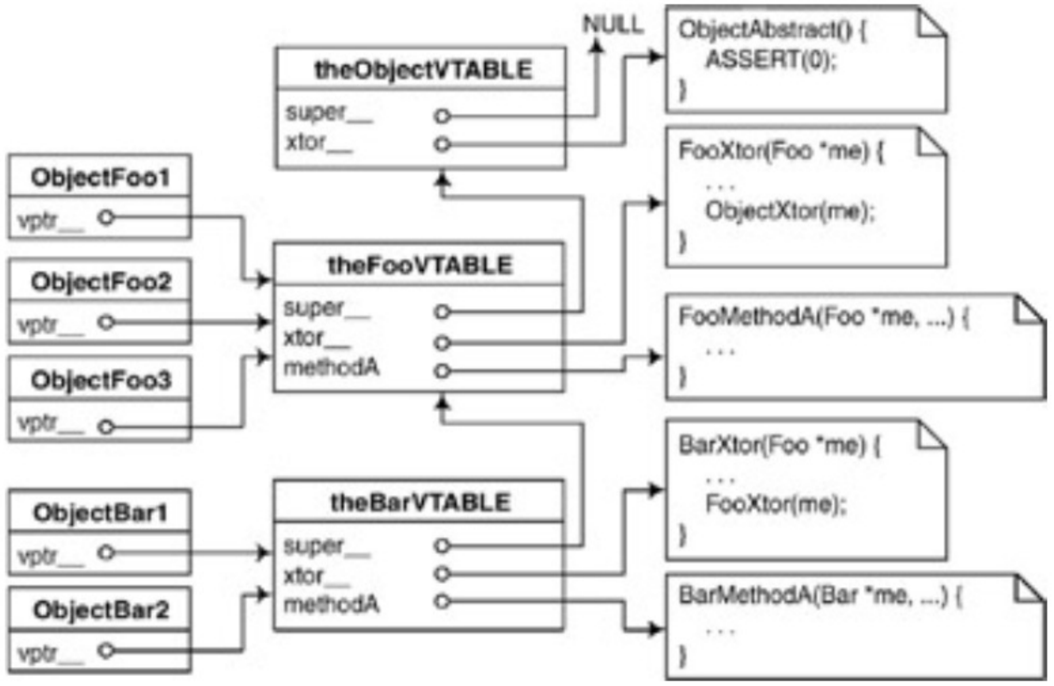
Polymorphism

(*this section is really not necessary and can be skipped*)

- a class can override behavior defined by its parent class by providing a different implementation of one or more inherited methods.
- With polymorphism, in general, the association between an object and its methods cannot be established at compile time. Binding must happen at run time and is therefore called *dynamic binding*.
- dynamic binding always involves a level of indirection in method invocation.

Polymorphism: dynamic binding

- In C, this indirection can be provided by function pointers grouped into virtual tables (VTABLE).
- The function pointer stored in the VTABLE represents a method (a virtual method in C++), which a subclass can override.
- All instances (objects) of a given class have a pointer to the VTABLE of that class (one VTABLE per class exists).
  - This pointer is called the *virtual pointer* (VPTR).

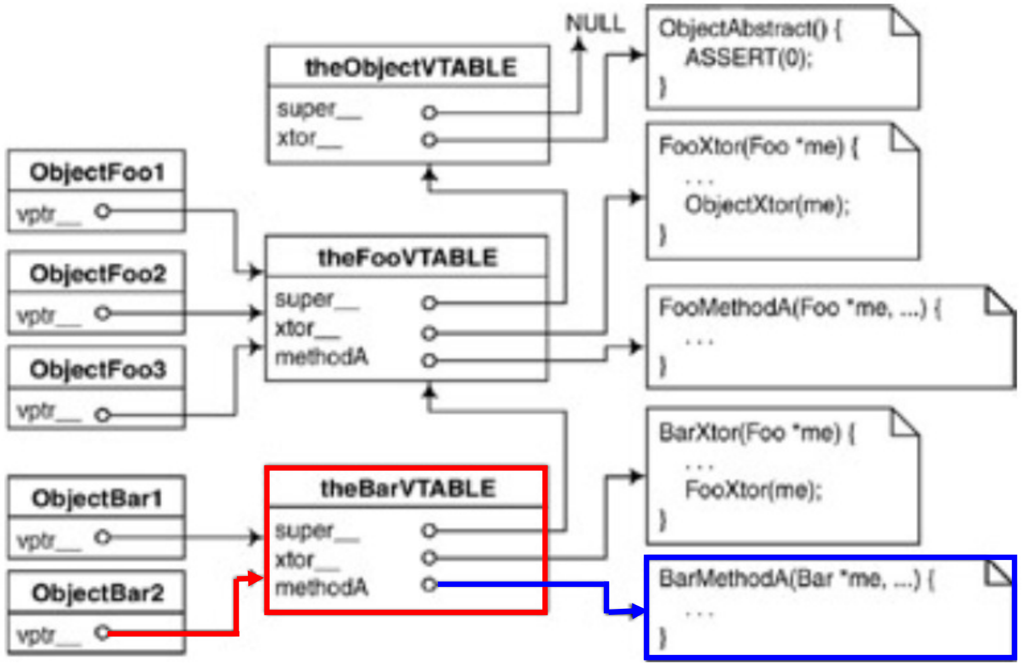


Polymorphism: dynamic binding

Late binding is a two–step process of

(1) dereferencing the VPTR to get to the VTABLE, and

(2) dereferencing the desired function pointer to invoke the specific implementation.



Polymorphism: dynamic binding

- Each object involved in dynamic binding must store the VPTR to the VTABLE of its class.
- One way to enforce this condition is to require that all classes using polymorphism be derived, directly or indirectly, from a common abstract base class, Object (a loaner from Java).
- The VTABLEs themselves require a separate and parallel class hierarchy, because the virtual methods, as well as the attributes, need to be inherited.
- The root abstract base class for the VTABLE hierarchy is the ObjectVTABLE class.

Polymorphism: dynamic binding

```
CLASS(ObjectVTABLE)
   ObjectVTABLE *super__;  /* pointer to superclass' VTABLE */
   void (*xtor)(Object *); /* public virtual destructor */
METHODS
END_CLASS

extern ObjectVTABLE theObjectVTABLE; /* Object class VTABLE */
```

- The purpose of the ObjectVTABLE class is to provide an abstract base class for the derivation of VTABLEs.
- The private attribute super__ is a pointer to the VTABLE of the superclass.
- The second attribute is the virtual destructor, which subsequently is inherited by all subclasses of ObjectVTABLE.
- There should be exactly one instance of the VTABLE for any given class. The VTABLE instance for the Object class (theObjectVTABLE) is declared in the last line.

Polymorphism: dynamic binding

```
CLASS(Object)
  struct ObjectVTABLE *vptr__; /* private vpointer */
METHODS
/* protected constructor 'inline'... */
  # define ObjectCtor_(_me_) ((_me_)->vptr__= &theObjectVTABLE,(_me_))
/* destructor 'inline'... */
  # define ObjectXtor_(_me_) ((void)0)
/* dummy implementation for abstract methods */
  void ObjectAbstract(void);
/* RTTI */
  # define ObjectIS_KIND_OF(_me_, _class_) \
  ObjectIsKindOf__((Object*)(_me_), &the##_class_##Class)
  int ObjectIsKindOf__(Object *me, void *class);
END_CLASS

```

Polymorphism: dynamic binding

```
SUBCLASS (touHMI, HMI)          typedef struct touHMI touHMI;\
                                struct touHMI {               ...
                                  HMI super;
VTABLE(touHMI, HMI)             };
                                typedef struct touHMIVTABLE touHMIVTABLE;
                                extern touHMIVTABLE thetouHMIVTABLE;
                                struct touHMIVTABLE {
                                   HMIVTABLE super_;
  void (*myMet)(Object *);         void (*myMet)(Object *);
METHODS                          };

BEGIN_VTABLE(touHMI, HMI)       touHMIVTABLE  thetouHMIVTABLE;
                                static ObjectVTABLE *touHMIVTABLECtor(HMI *t) {
                                  register touHMIVTABLE *me = &thetouHMIVTABLE;
                                  *(HMIVTABLE *)me = *(HMIVTABLE *)((Object *)t)->vptr__;

VMETHOD(touHMI, xtor) =           ((touHMI *)me)->xtor = (void (*)(Object *))touHMIXtor_;
(void (*)(Object *))touHMIXtor_;
VMETHOD(touHMI, myMet) =          ((touHMI *)me)->myMet = (void (*)(Object *))touHMImyM_;
(void (*)(Object *))touHMImyM_;

END_VTABLE                        ((ObjectVTABLE*)me)->super__ = ((Object*)t)->vptr__;
                                  return (ObjectVTABLE *)me; \
                                }
```

Polymorphism: dynamic binding

The hierarchies of the attribute classes (rooted in the Object class) and VTABLEs (rooted in the ObjectVTABLE class) must be exactly parallel.

The declaration of the VTABLE hierarchy and the VTABLE singletons can be encapsulated in the VTABLE() macro.

```
#define VTABLE(class_, superclass_) }; \
typedef struct class_##VTABLE class_##VTABLE; \
extern class_##VTABLE the##class_##VTABLE; \
struct class_##VTABLE { \
  superclass_##VTABLE super_;
```

```
SUBCLASS (touHMI, HMI)          ...
  VTABLE(touHMI, HMI)         };
                              typedef struct touHMIVTABLE touHMIVTABLE;
                              extern touHMIVTABLE thetouHMIVTABLE;
                              struct touHMIVTABLE {
                                HMIVTABLE super_;
METHODS                       };
```

Polymorphism: dynamic binding

VTABLE singletons, as with all other objects, need to be initialized through their own constructors, which the preprocessor macros can automatically generate.

The body of the VTABLE constructor can be broken into two parts:

(1) copying the inherited VTABLE and

(2) initializing or overriding the chosen function pointers.

The first step is generated automatically by the macro BEGIN_VTABLE().

```
#define BEGIN_VTABLE(class_, superclass_) \
class_##VTABLE the##class_##VTABLE; \
static ObjectVTABLE *class_##VTABLECtor(class_ *t) \
register class_##VTABLE *me = &the##class_##VTABLE; \
*(superclass_##VTABLE *)me = \
*(superclass_##VTABLE *)((Object *)t)->vptr__;
```

Polymorphism: dynamic binding

```
1 #define BEGIN_VTABLE(class_, superclass_) \
2 class_##VTABLE the##class_##VTABLE; \
3 static ObjectVTABLE *class_##VTABLECtor(class_ *t) \
4 register class_##VTABLE *me = &the##class_##VTABLE; \
5 *(superclass_##VTABLE *)me = \
6 *(superclass_##VTABLE *)((Object *)t)->vptr__;
```

```
CLASS(touHMI)                         };
  VTABLE(touHMI, HMI)                   typedef struct touHMIVTABLE touHMIVTABLE;
                                        extern touHMIVTABLE thetouHMIVTABLE;
                                        struct touHMIVTABLE {
                                          HMIVTABLE super_;
  BEGIN_VTABLE(touHMI, HMI)               touHMIVTABLE  thetouHMIVTABLE;
                                          static ObjectVTABLE *touchHMIVTABLECtor(HMI *t)
                                          register touchHMIVTABLE *me = &thetouchHMIVTABLE;
                                          *(HMIVTABLE *)me = *(HMIVTABLE *)((Object *)t)->vptr__;
```

This macro first defines the ClassVTABLE instance (line 2), then starts defining the static VTABLE constructor (line 3).

- First, the constructor makes a copy (by value) of the inherited VTABLE (lines 5, 6).
- After adding the attributes or methods to the superclass, no manual changes to the subclasses are required. You only have to recompile the subclass code.

Polymorphism: dynamic binding

```
1 #define BEGIN_VTABLE(class_, superclass_) \
2 class_##VTABLE the##class_##VTABLE; \
3 static ObjectVTABLE *class_##VTABLECtor(class_ *t) { \
4 register class_##VTABLE *me = &the##class_##VTABLE; \
5 *(superclass_##VTABLE *)me = \
6 *(superclass_##VTABLE *)((Object *)t)->vptr__;
```

```
METHODS
  BEGIN_VTABLE(touchHMI, HMI)
  /* other methods  */
  /* VMETHOD macro */
```

```
};
  touchHMIVTABLE  thetouchHMIVTABLE;
  static ObjectVTABLE *touchHMIVTABLECtor(HMI *t) {
  register touchHMIVTABLE *me = &thetouchHMIVTABLE;
  *(HMIVTABLE *)me = *(HMIVTABLE *)((Object *)t)->vptr__;
```

This macro first defines the ClassVTABLE instance (line 2),

then starts defining the static VTABLE constructor (line 3).

- **First, the constructor makes a copy (by value) of the inherited VTABLE (lines 5, 6).**
- After adding the attributes or methods to the superclass, no manual changes to the subclasses are required. You only have to recompile the subclass code.

Of course, if a class adds its own virtual functions, the corresponding function pointers are not initialized during this step.

Polymorphism: dynamic binding

The second step of binding virtual functions to their implementation is facilitated by the VMETHOD() macro.

```
#define VMETHOD(class_, meth_) ((class_##VTABLE *)me)->meth_
```

This macro is an lvalue, and its intended use is to assign to it the appropriate function pointer as follows.

```
VMETHOD(Object, xtor) = (void (*)(Object *))touchHMIXtor_;
```

```
METHODS
 BEGIN_VTABLE(touchHMI, HMI)
 /* other methods      */
 /* VMETHOD macro */
 VMETHOD(touchHMI, xtor) =
  (void (*)(Object *))touchHMIXtor_;
```

```
};
 touchHMIVTABLE  thetouchHMIVTABLE;
 static ObjectVTABLE *touchHMIVTABLECtor(HMI *t) {
 register touchHMIVTABLE *me = &thetouchHMIVTABLE;
 *(HMIVTABLE *)me = *(HMIVTABLE *)((Object *)t)->vptr;
((touchHMI *)me)->xtor = (void (*)(Object *))touchHMIXtor_;
```

Generally, to avoid compiler warnings, you must explicitly upcast the function pointer to take the superclass me pointer (Object* in this case) rather than the subclass pointer (touchHMI* in this case).

The explicit upcasting is necessary, because the C compiler doesn't know that touchHMI is related to Object by inheritance and treats these types as completely different.

Polymorphism: dynamic binding

If you don't want to provide the implementation for a given method because it is intended to be abstract (a pure virtual in C++), you should still initialize the corresponding function pointer with the ObjectAbstract() dummy implementation.

An attempt to execute ObjectAbstract() aborts the execution through a failed assertion, which helps detect unimplemented abstract methods at run time.

Polymorphism: dynamic binding

The definition of every VTABLE constructor opened with BEGIN_VTABLE() must be closed with the following END_VTABLE

```
#define END_VTABLE\
((ObjectVTABLE*)me)->super__ =
((Object*)t)->vptr__; \
return (ObjectVTABLE *)me; \
}
```

```
END_VTABLE      ((ObjectVTABLE*)me)->super__=((Object*)t)->vptr__;
                return (ObjectVTABLE *)me;
               }
```

Polymorphism: dynamic binding

The attribute and virtual method class hierarchies are coupled by the VPTR attribute, which needs to be initialized to point to the appropriate VTABLE singleton.
The appropriate place to set up this pointer is the attribute constructor, after the superclass constructor call to set the VPTR to point to the VTABLE of the superclass.
If the VTABLE for the object under construction is not yet initialized, the VTABLE constructor should be called.
These two steps are accomplished by invoking the VHOOK() macro.

```
#define VHOOK(class_) \
if (((ObjectVTABLE *)&the##class_##VTABLE)->super__== 0) \
((Object *)me)->vptr__ = class_##VTABLECtor(me); \
else \
((Object *)me)->vptr__ = \
(ObjectVTABLE *)&the##class_##VTABLE
```

Polymorphism: dynamic binding

```
#define VHOOK(class_) \
if (((ObjectVTABLE *)&the##class_##VTABLE)->super__== 0) \
((Object *)me)->vptr__ = class_##VTABLECtor(me); \
else
((Objec         )->vptr__ = \
(ObjectV          the##class_##VTABLE
```

The macro determines whether the VTABLE has been initialized. It checks the super__ attribute. If it is NULL (initialization value in C) …

Polymorphism: dynamic binding

```
#define VHOOK(class_) \
if (((ObjectVTABLE *)&the##class_##VTABLE)->super__== 0) \
((Object *)me)->vptr__ = class_##VTABLECtor(me); \
els  \
((Obj     *)me)->vptr__ = \
(Objec       *)&the##class_##VTABLE
```

If it is NULL (initialization value in C) … then the VTABLE constructor must be invoked before setting up the VPTR;

Polymorphism: dynamic binding

```
#define VHOOK(class_) \
if (((ObjectVTABLE *)&the##class_##VTABLE)->super__== 0) \
((Object *)me)->vptr__ = class_##VTABLECtor(me); \
else \
((Object *)me)->vptr__ = \
(ObjectVTABLE *)&the##class_##VTABLE
```

otherwise, just the VPTR must be set up

Polymorphism: dynamic binding

```
#define VHOOK(class_) \
if (((ObjectVTABLE *)&the##class_##VTABLE)->super__== 0) \
((Object *)me)->vptr__ = class_##VTABLECtor(me); \
else \
((Object *)me)->vptr__ = \
(ObjectVTABLE *)&the##class_##VTABLE
```

```
VHOOK(touHMI)    if (((ObjectVTABLE*)&thetouHMIVTABLE)->super__== 0)
                    ((Object *)me)->vptr__ = touHMIVTABLECtor(me);
                 else
                    ((Object *)me)->vptr__ = (ObjectVTABLE *)&thetouHMIVTABLE
```

Polymorphism: dynamic binding

Note that because VHOOK() is invoked after the superclass constructor, the VTABLE of the superclass is already initialized by the same mechanism applied recursively, so the whole class hierarchy is initialized properly.

Finally, after all the setup work is done, you are ready to use dynamic binding. For the virtual destructor (defined in the Object class), the polymorphic call takes the form

```
(*obj->vptr__->xtor)(obj);
```

where obj is assumed to be of Object* type. Note that the obj pointer is used in this example twice: once for resolving the method and once as the me argument.

Polymorphism: dynamic binding

In the general case, you deal with Object subclasses rather than the Object class directly. Therefore you have to upcast the object pointer (on type Object*) and downcast the virtual pointer vptr__ (on the specific VTABLE type) to find the function pointer. These operations, as well as double−object pointer referencing, are encapsulated in the macros VPTR(), VCALL(), and END_CALL.

```
#define VPTR(class_, obj_) \
((class_##VTABLE *)(((Object *)(obj_))->vptr__))
#define VCALL(class_, meth_, obj_) \
(*VPTR(class_, _obj_)->meth_)((class_*)(obj_)
#define END_CALL )
```

Polymorphism: dynamic binding

The virtual destructor call on behalf of object touHMI of any subclass of class Object takes the following form.

```
VCALL(Object, xtor, touHMI)
END_CALL;
```

If a virtual function takes arguments other than me, they should be sandwiched between the VCALL() and END_CALL macros.

The virtual function can also return a result.

```
result = VCALL(Foo, myMethod, obj), 2, 3, END_CALL;
```

obj points to a Foo class or any subclass of Foo, and the virtual function myMethod() is defined in touHMIVTABLE.

Note the use of the comma after VCALL().

Design options

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

Design options

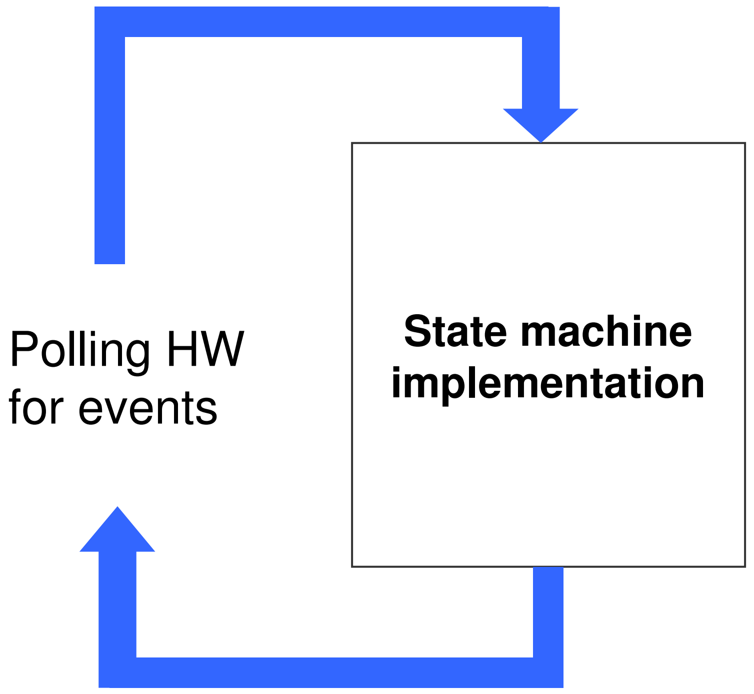
- Design decisions and trade-offs
  - How do you represent events? How about events with parameters?
  - How do you represent states?
  - How do you represent transitions?
  - How do you dispatch events to the state machine?
- When you add state hierarchy, exit/entry actions and transitions with guards,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

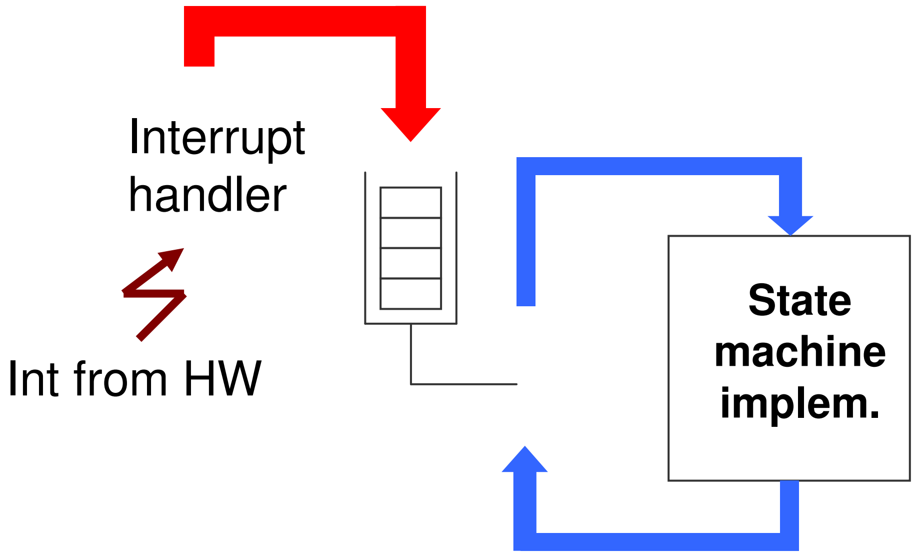
Typical implementations

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

Typical implementations

- Implementation of a FSM is not enough!
- State machine implementations are typically coupled with a concurrency model and an event dispatching policy
- More about this later !!





Typical implementations

- The simplest interface for an FSM implementation 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

Or nest the switches first by event and then by state

This seems to be the option of embedded coder …
(and there may be good reasons)

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:
        ManageState2(sig);
        break;
   ...
}
```

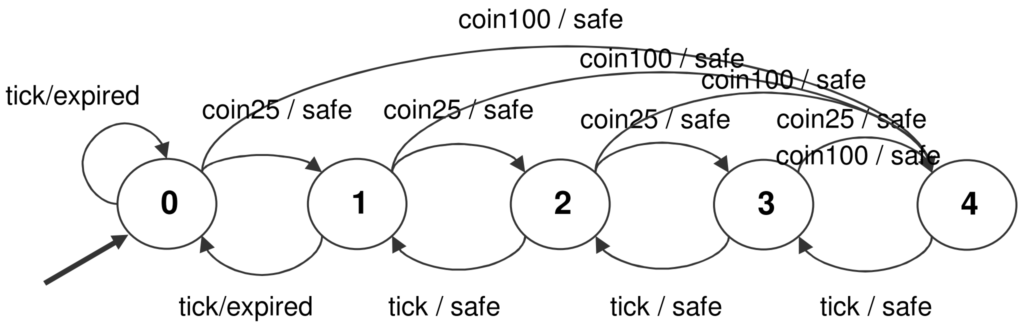
Nested switch method

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

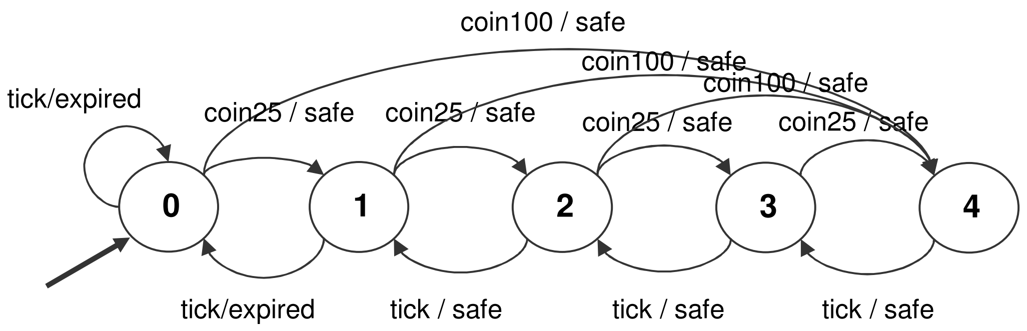
The example ....

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

enum Display {
   EXPIRED, SAFE
};
```



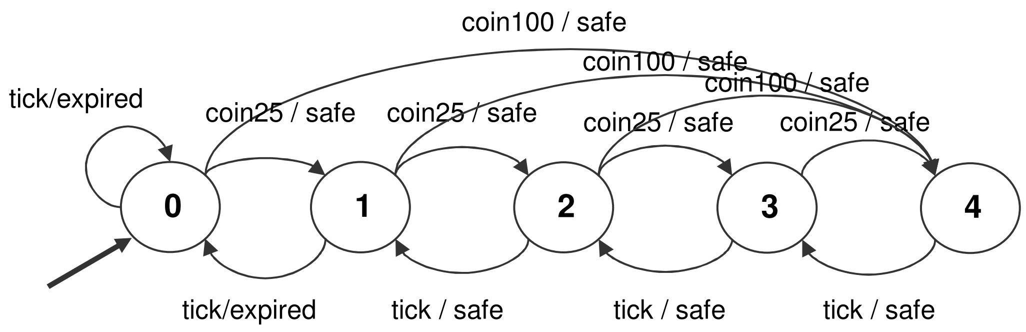
The example ….

```
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
```



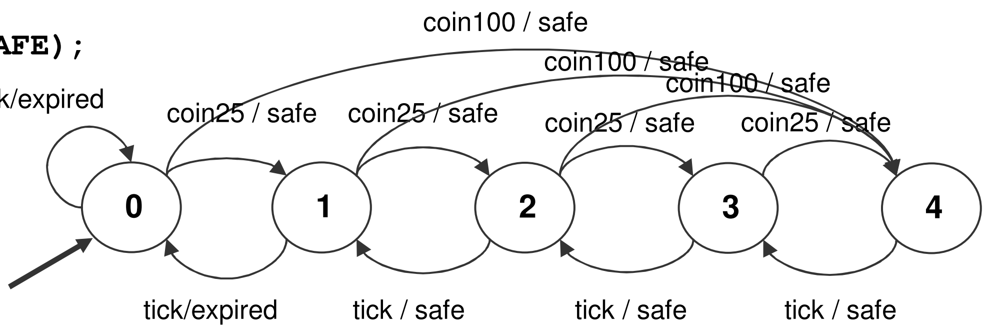
The example ….

```
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

type Tran is the type of the table cell

```
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;
};
```

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

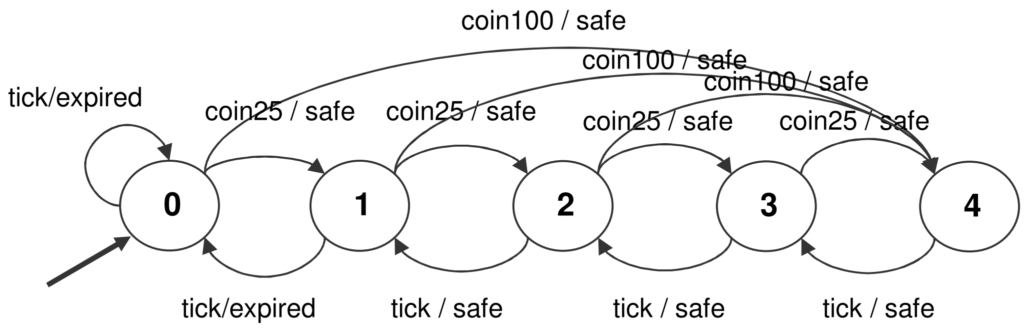
Typical implementations

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

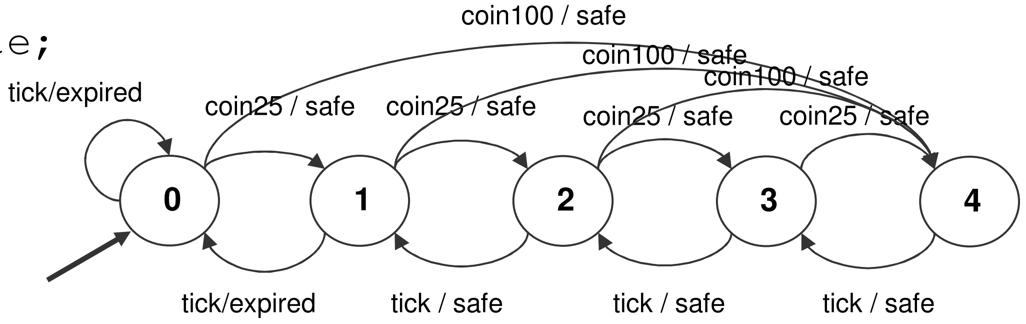
The example: basic types

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

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



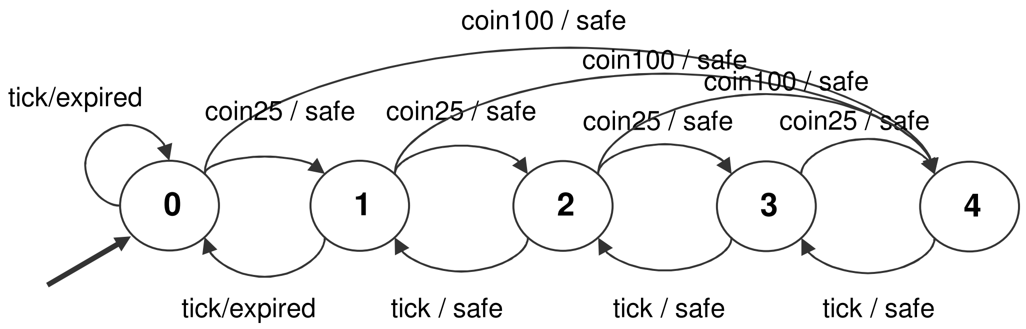
The Example: the State Table "class"

```
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 = me->myTable__ +
   me->myState_*me->myNsignals__ + sig;
        t->action();
        myState_ = t->nextState;
   }
   void doNothing() {};
END_CLASS
```



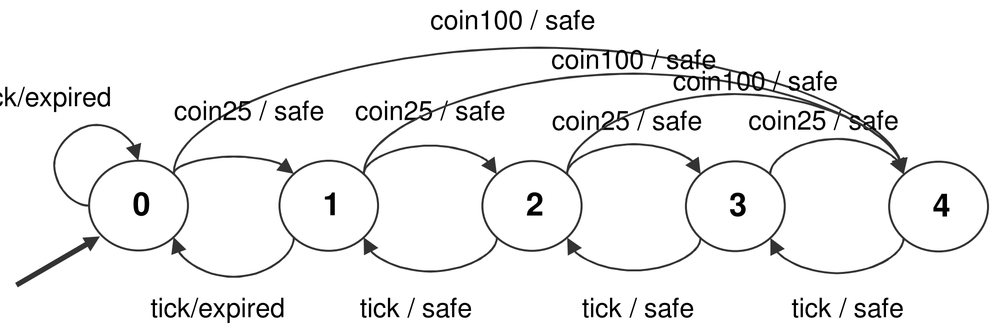
The example: preparing for PMeter

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



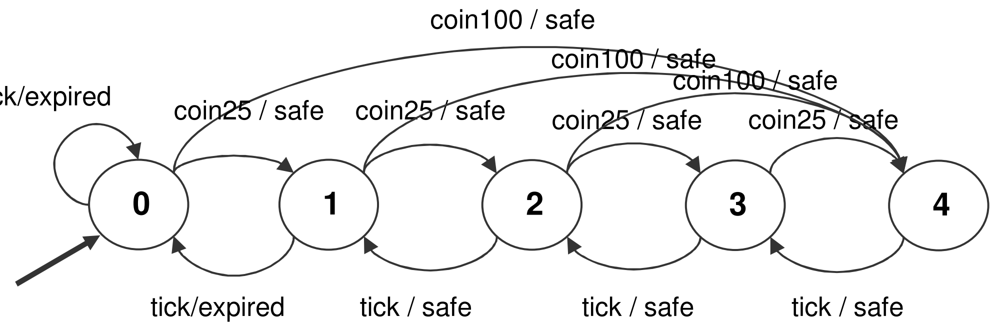
The Example: the PMeter “class”

```
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
```



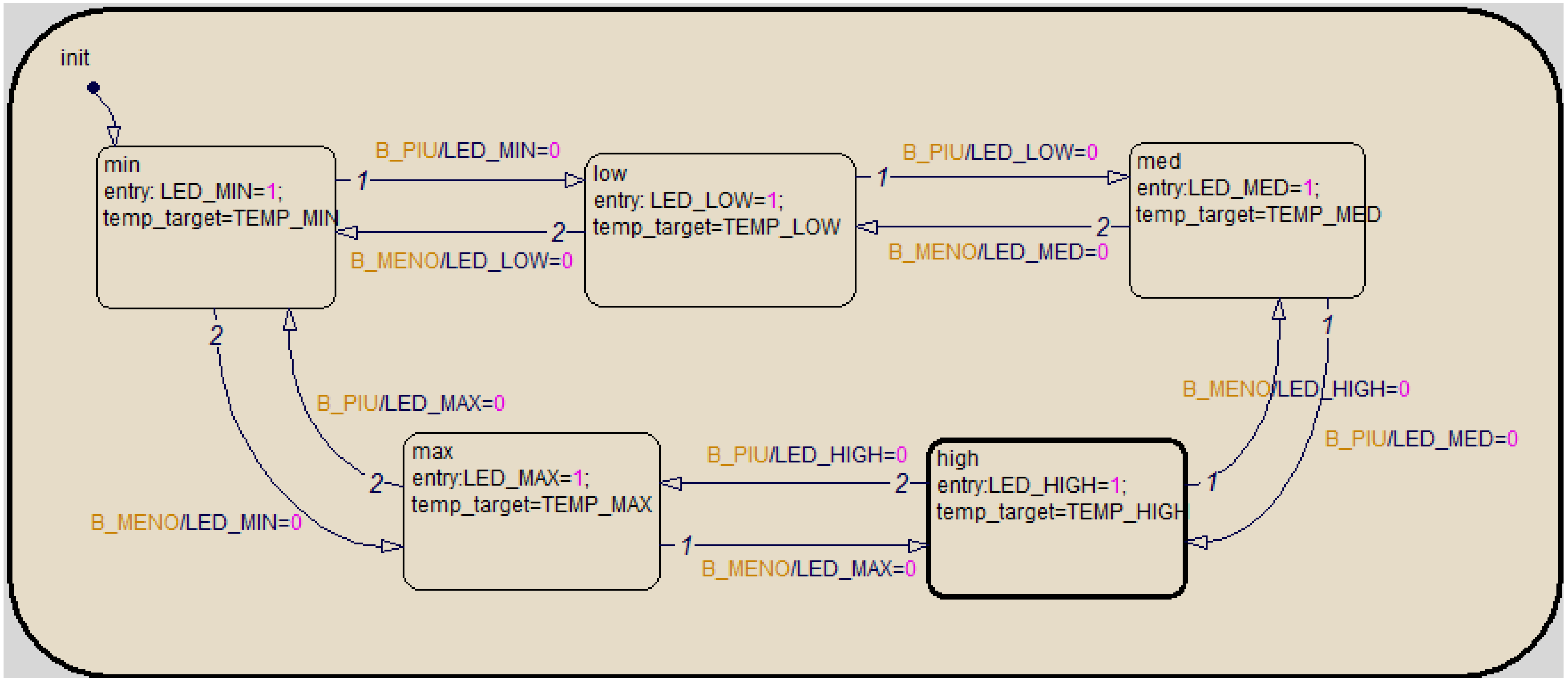
An example

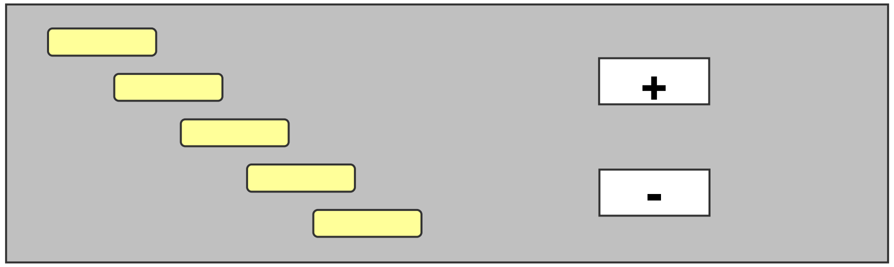
```
Tran const myTable[MAX_STATE][MAX_SIGNAL] = {
   {{ &doNothing, S_0},
    { &PMeterShowSafe, S_1},
    { &PMeterShowSafe, S_4}},
   {{ &PMeterShowExpired, S_0},
    { &doNothing, S_2},
    { &doNothing, S_4}},
   {{ &doNothing, S_1},
    { &doNothing, S_3},
    { &doNothing, S_4}},
   {{ &doNothing,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



Typical implementations: State pattern

Led setup example

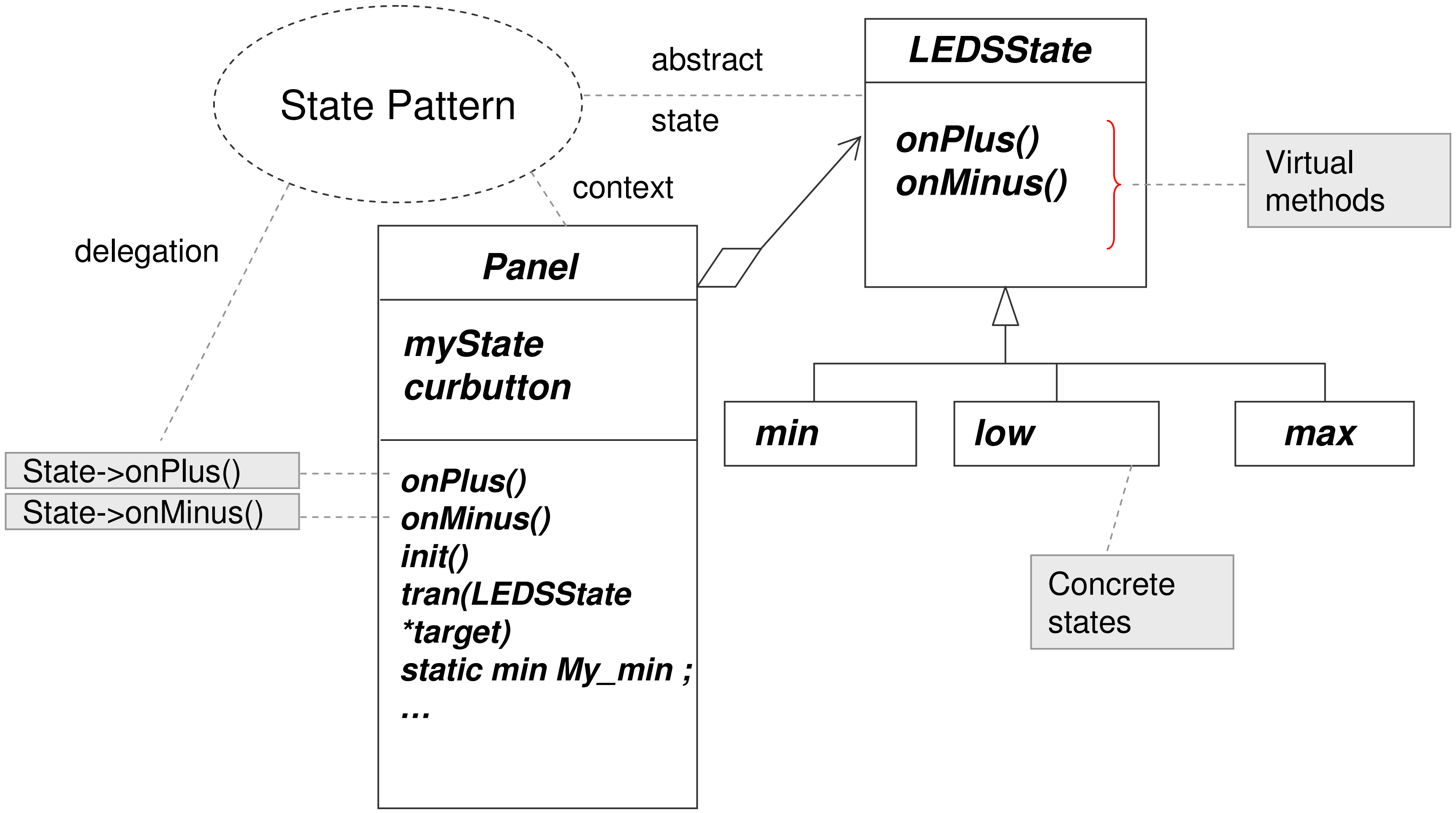




Typical implementations

State Pattern

- OO approach



Typical implementations: State pattern

- States are represented as subclasses of an abstract state class (LEDSState), which defines a common interface for handling events (each event corresponds to a virtual method).
- A context class (Panel) performs the processing and delegates all events for processing to the current state object (designated by the myState attribute).
- State transitions are accomplished by reassigning the myState pointer.
- Adding new events requires adding new methods to the abstract state class (and to the concrete classes actually hendling them)
- Adding new states requires subclassing the abstract state class .

Typical implementations: State pattern

- The LEDSState class provides the interface for handling events as well as the default (do–nothing) implementation for the actions associated with these events.
- The ***min***, ***low***, ***med***, ***high***, ***max*** states are defined as concrete subclasses of the abstract LEDSState class.
- State subclasses override only event–handler methods corresponding to events that are handled in these states.
- Class Panel plays the role of the context class from the pattern. It grants friendship to all state classes and also declares all concrete states as static members.

Typical implementations: State pattern

- Example

```
class PanelState { // abstract State
 public:
   virtual void onPlus(Panel *context) {}
   virtual void onMinus(Panel *context) {}
};
class min : public PanelState { // concrete State min
 public:
   virtual void onPlus(Panel *context);
   virtual void onMinus(Panel *context);
};
```

Typical implementations: State pattern

- Example

```
class Panel { // Context class
   friend class min;
   friend class low;
   ...
   friend class high;
   static min my_min;
   static min my_low;
   ...
   static min my_high;
   PanelState *myState;
 public:
   Panel(PanelState *initial) : myState(initial) {}
   void init() { AllLedsOff(); tran(&my_min); }
   void onPlus() { myState->onPlus(this); }
   void onMinus() { myState->onMinus(this); }
 protected:
   void tran(PanelState *target) { myState = target; }
};
```

Typical implementations: State pattern

- The context class ***Panel*** duplicates the interface of the abstract state class declaring a method for every signal event.
- The implementation of these methods is <u>fixed</u> as prescribed by the pattern.

```
void onEvent(args) { myState->onEvent(this, args); }
```

- The context class simply delegates to the appropriate methods of the state class, which are invoked polymorphically.
- The specific actions are implemented inside the event handler methods of the concrete LEDSState subclasses.

Typical implementations: State pattern

The State design pattern has the following consequences.

- It partitions state–specific behavior and localizes it in separate classes.
- It makes state transitions efficient (reassigning one pointer).
- It provides very good performance for event dispatching through the late binding mechanism (O(const), not taking into account action execution).
  - This performance is generally better than indexing into a state table plus invoking a method via a function pointer, as used in the state table technique.
  - However, such performance is only possible because the selection of the appropriate event handler is not taken into account. Indeed, <u>clients typically will use a switch statement to perform such selections</u>.

Typical implementations: State pattern

The State design pattern has the following consequences.

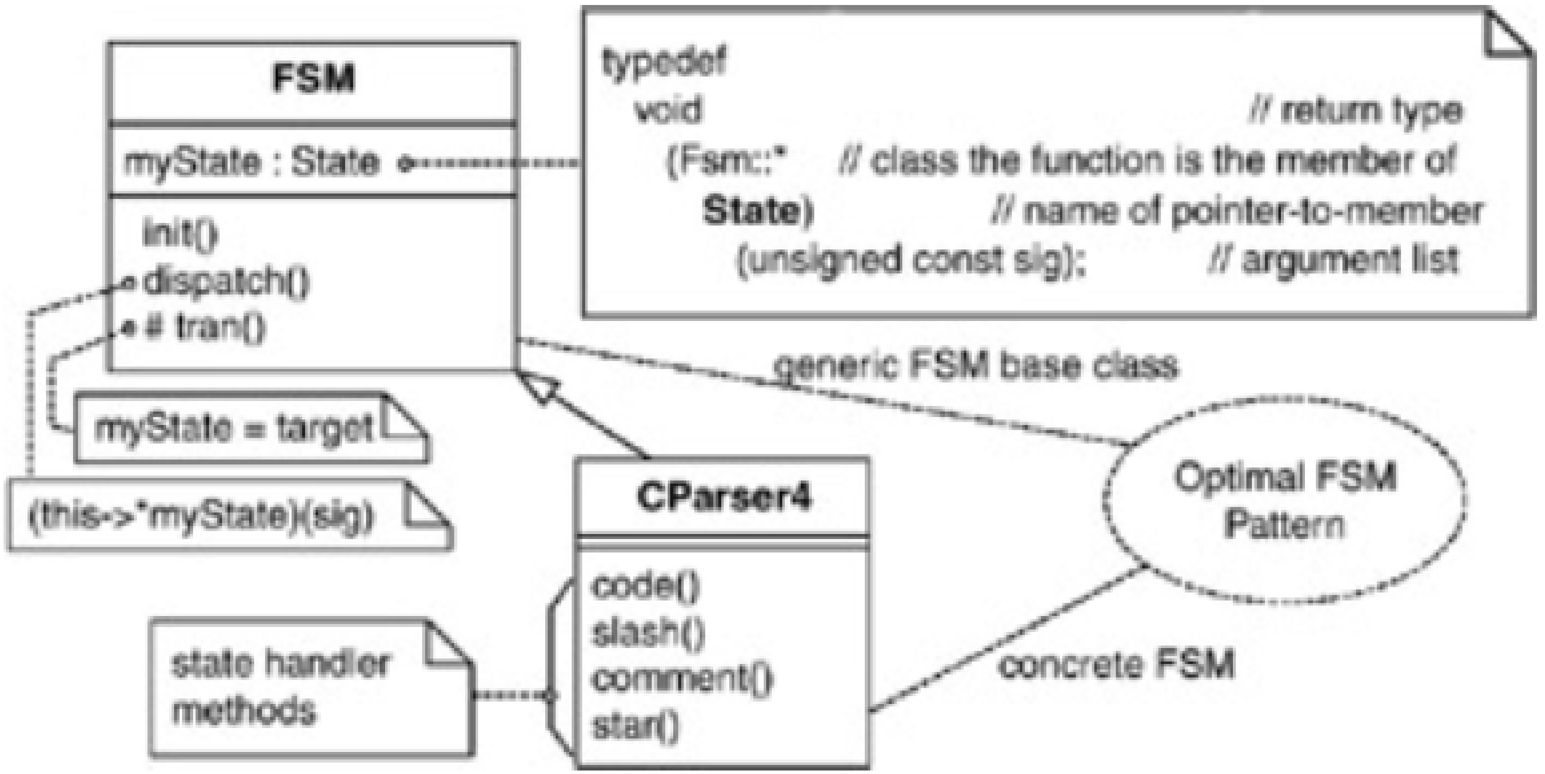
- It allows you to customize the signature of each event handler. Event parameters can be made explicit.
- It is memory efficient. If the concrete state objects don't have attributes (only methods), they can be declared static and shared among FSMs.
- It does not require enumerating states.
- It does not require enumerating events.
- It compromises the encapsulation of the context class, which requires granting friendship to all state classes.
- Adding states requires adding state subclasses.
- Handling new events requires adding event handlers to the state class interface.
- The event handlers are typically of fine granularity, as in the state table approach.
- It is not hierarchical.

Typical implementations: State pattern

- The standard State design pattern does not use the dispatch() method for performing RTC steps, but provides a specific (type–safe) event–handler method for every signal event.
- However, the pattern can be modified (simplified) by combining all event handlers of the state class into just one, generic state handler, dispatch().
  - The abstract state class then becomes generic, and its dispatch() method becomes the generic state handler. Demultiplexing events (by event type), however, must be done inside the dispatch() methods of the concrete state subclasses.

Samek’s proposal

•A mixed implementation



Typical implementations: Samek’s pattern

- The implementation combines elements from the nested switch statement, state table, and State design pattern.
- The design hinges on class ***Fsm***. This class plays a double role as the context class from the state pattern and the event processor from the table pattern.
- States are represented as function pointers, more exactly, as handlers to methods of ***Fsm*** (actually, its concrete subclasses like Panel).
    - This means that state handlers have immediate access to all attributes of the context class (via the this pointer) without breaking encapsulation.
- Like the context class, Fsm keeps track of the current state by means of the myState attribute of type Fsm::State, which is a pointer–to–member function of the Fsm class.

```
typedef void (Fsm::*State)(unsigned const sig);
```

Typical implementations: Samek’s pattern

- LED panel implementation

```
class Fsm {
public:
  typedef void (Fsm::*State)(unsigned const sig);
  Fsm(State initial) : myState(initial) {}
  virtual ~Fsm() {} // virtual xtor
  void init() { dispatch(0); }
  void dispatch(int sig) { (this->*myState)(sig); }
protected:
  void tran(State target) { myState = target; }
  #define TRAN(target_) tran(static_cast<State>(target_))
  State myState;
};
```

Typical implementations: Samek’s pattern

- LED panel implementation

```
enum Signal{ // enumeration for CParser signals
 B_PLUS, B_MINUS
};

class Panel : public Fsm {
public:
  Panel() : Fsm((FSM_STATE)initial) {} // ctor
private:
  void initial(int); // initial pseudostate
  void min(int sig); // state-handler
  void low(int sig); // state-handler
  void med(int sig); // state-handler
  void max(int sig); // state-handler
  void high(int sig); // state-handler
private:
  LedDevice *LEDS; // comment character counter
};
```

Typical implementations: Samek’s pattern

- LED panel implementation

```
void Panel::initial(int) {
   AllLedsOff();
   TRAN(&Panel::min); // take the default transition
}

void Panel::min(int sig) {
  switch (sig) {
  case B_PLUS:
    TRAN(&Panel::low); // transition to “low"
    break;
  case B_MINUS:
    TRAN(&Panel::max); // transition to “max"
    break;
  }
}
```

Typical implementations: Samek’s pattern

- The corresponding C implementation is straightforward ….

Typical implementations: Samek’s pattern

This FSM design pattern has the following characteristics.

- It is simple.
- It partitions state–specific behavior and localizes it in separate state handler methods.
- It provides direct and efficient access to state machine attributes from state handler methods and does not require compromising the encapsulation of the context class.
- It has a small memory footprint because only one state variable (the myState pointer) is necessary to represent a state machine instance.
- It promotes code reuse of an extremely small (trivial) and generic event processor implemented in the Fsm base class.
- It makes state transitions efficient (by reassigning the myState pointer).

more …

Typical implementations: Samek's pattern

This FSM design pattern has the following characteristics.

- It provides good performance for event dispatching by eliminating one level of switch from the nested switch statement technique and replacing it with the efficient function pointer dereferencing technique. However, the switch can be replaced by a look–up table in selected (critical) state handlers
- It is scalable and flexible. It is easy to add both states and events, as well as to change state machine topology, even late in the development cycle.
- It does not require enumerating states (only events must be enumerated).
- It is not hierarchical.

Implementing Guards, Junctions, and Choice Points

- Guards, Junctions and choice points map to plain structured code and are therefore easy to implement in those techniques that give program-level control of the target of a state transition, such as the nested switch statement, the State design pattern, and the optimal FSM pattern.
- A guard specified in the expression [guard]/action … maps simply maps to the if statement: if(guard()) { action(); …}.
- Conditional execution is much harder to use in the state table technique because the rigidly structured state table implicitly selects the targets of state transitions.
- In the absence of orthogonal regions, a junction pseudostate can have only one incoming transition segment and many outgoing segments guarded by nonoverlapping guard expressions. This construct maps simply to chained if–else statements:

```
if (guard1()) { action1();} else if (guard2()) { action2();}
```

Implementing Entry and Exit Actions

- The classical nonhierarchical state machines can also reap the benefits of a guaranteed initialization of the state context through entry actions and a guaranteed cleanup in the exit actions.
- One way of implementing entry and exit actions is to dispatch reserved signals (e.g., ENTRY_SIG and EXIT_SIG) to the state machine.
- The tran() method could dispatch the EXIT_SIG signal to the current state (transition source) then dispatch the ENTRY_SIG signal to the target as in.

```
void Fsm::tran(FsmState target) {
  (this->*myState)(EXIT_SIG); // EXIT signal to target
  myState = target;
  (this->*myState)(ENTRY_SIG); //ENTRY signal to source
}
```

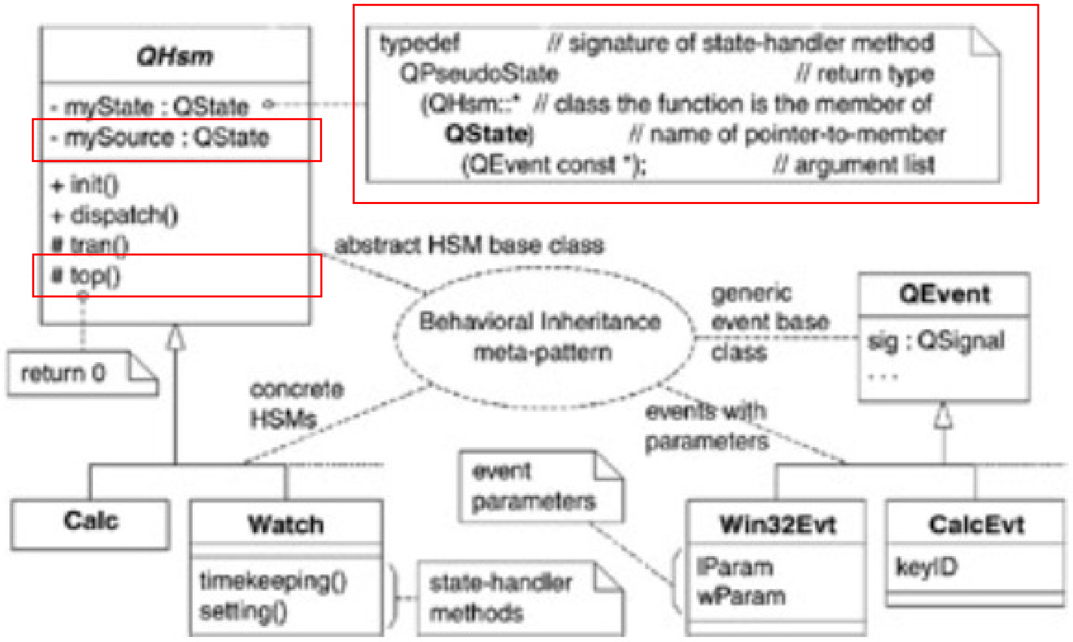
Implementing hierarchical behavior

The pattern for behavior inheritance addresses the following few essential elements of HSM implementation:

- hierarchical states with full support for behavioral inheritance,
- initialization and cleanup with state entry and exit actions, and
- support for specializing state models via class inheritance.

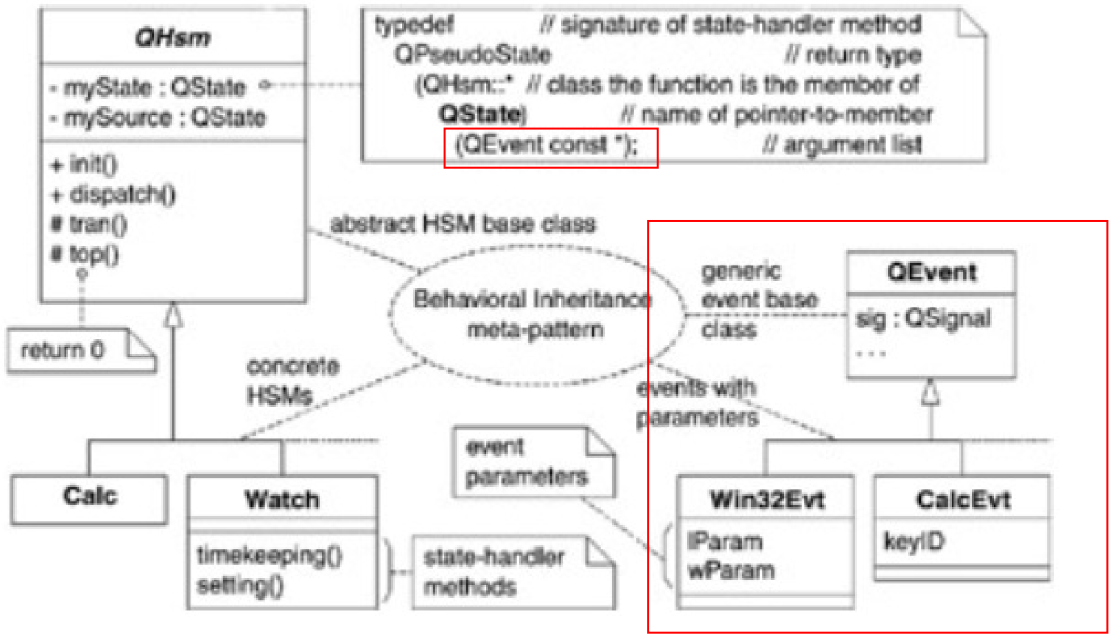
Next, we will see how to realize orthogonal regions, and transitions to history as patterns that build on top of the fundamental behavioral inheritance implementation.

Implementing hierarchical behavior



- As in the first Samek's FSM design …
  - the state QState (quantum state) is represented as a pointer–to–member function of the QHsm class.
  - class QHsm keeps track of the active state by means of the myState attribute.
- In addition, it uses another attribute (mySource) to keep track of the current transition source during a state transition
  - in HSMs, when a transition is inherited from a higher level superstate, the source of this transition is different from the active state.

Implementing hierarchical behavior



- On the right side, the facilities for representing events and event parameters.
- The QEvent class represents a Signal/Event, which can be used as–is (for events without parameters) or can serve as a base for subclassing (for events with parameters).
- QEvent relates to QHsm through the signature of the state handler method

Implementing hierarchical behavior

QHsm class, provides the familiar public methods: `init`() and `dispatch`(). You also can find the protected `tran`() method for executing transitions, but this method is not intended to be invoked directly by the clients.

Instead, QHsm provides three macros to execute state transitions:

- Q_INIT() exclusively for initial transitions,
- Q_TRAN() for regular state transitions, and
- Q_TRAN_DYN() for state transitions in which the target can change at run time.

Implementing hierarchical behavior

Every state hierarchy is a specific “chain of responsibility”, in which a request (event instance) is sent down (up?) a chain of state hierarchy in which more than one state has a chance to handle it.

Implementing hierarchical behavior

```
class Hsm {                     // Quantum Hierarchical State Machine
 public:
   typedef void (Hsm::*PseudoState)(Event const *);
   typedef PseudoState (Hsm::*State)(Event const*);
   #define STATE Hsm::State

   Hsm(PseudoState initial);         // Ctor
   virtual ~Hsm();                   // virtual Xtor
   void init(Event const *e = 0);    // execute initial transition
   void dispatch(Event const *e);    // dispatch event
   int isIn(State state) const;      // "is-in-state" query
   static char const *getVersion();

 protected:
   struct Tran {                    // protected inner class Tran
     State myChain[8];
     unsigned short myActions;   // action mask (2-bits/action)
   };
// CONTINUES ....
```

Implementing hierarchical behavior

```
  PseudoState top(Event const*) { return 0; }      // the "top" state
  State getState() const { return myState; }
  void tran(State target);                         // dynamic state transition
  void tranStat(Tran *t, State target);            // static state transition
  void init_(State target) { myState = target; }
  #define Q_INIT(target_) init_( Q_STATIC_CAST(State, target_))
  #define Q_TRAN(target_) if (1) { \
  static Tran t_; \
  tranStat(&t_, Q_STATIC_CAST(State, target_));\
  } else ((void)0)
  #define Q_TRAN_DYN(target_) tran(Q_STATIC_CAST(State, target_))

 private:
  void tranSetup(Tran *t, State target);

 private:
   State myState;                                  // the active state
   State mySource;                      // source state during a transition
};
typedef Hsm::PseudoState STATE;              // state-handler return type
```

INIT, TRAN, TRAN_DYN

```
#define Q_INIT(target_) init_( Q_STATIC_CAST(State, target_))

#define Q_TRAN(target_) if (1) { \
  static Tran t_; \
  tranStat(&t_, Q_STATIC_CAST(State, target_));\
  } else ((void)0)

#define Q_TRAN_DYN(target_) tran(Q_STATIC_CAST(State, target_))
```

The purpose of these macros is to add a (compiler dependent) static cast to the State type of the type PseudoQState returned by init, tran and tranStat.

In C you'll need similar casting …

Implementing hierarchical behavior

most of the techniques (in particular, the optimal FSM approach) require a uniform representation of events, which leaves essentially only two choices for passing events to the handler methods:

(1) passing the signal and generic event parameters separately

(2) combining the two into an event object.

```
typedef unsigned short Signal; // Signal
  struct Event {               // Event
  Signal sig;
  . . .
};
```

Implementing hierarchical behavior

In contrast to a basic flat state, a hierarchical state includes more than behavior. At a minimum, it must provide a link to its superstate to represent state nesting.

A state handler method can provide behavior and the needed structural link by returning the superstate.

(PseudoState is needed to avoid a recursive definition)

```
typedef void (Hsm::*PseudoState)(Event const *);
typedef PseudoState (Hsm::*State)(Event const *);
```

return type

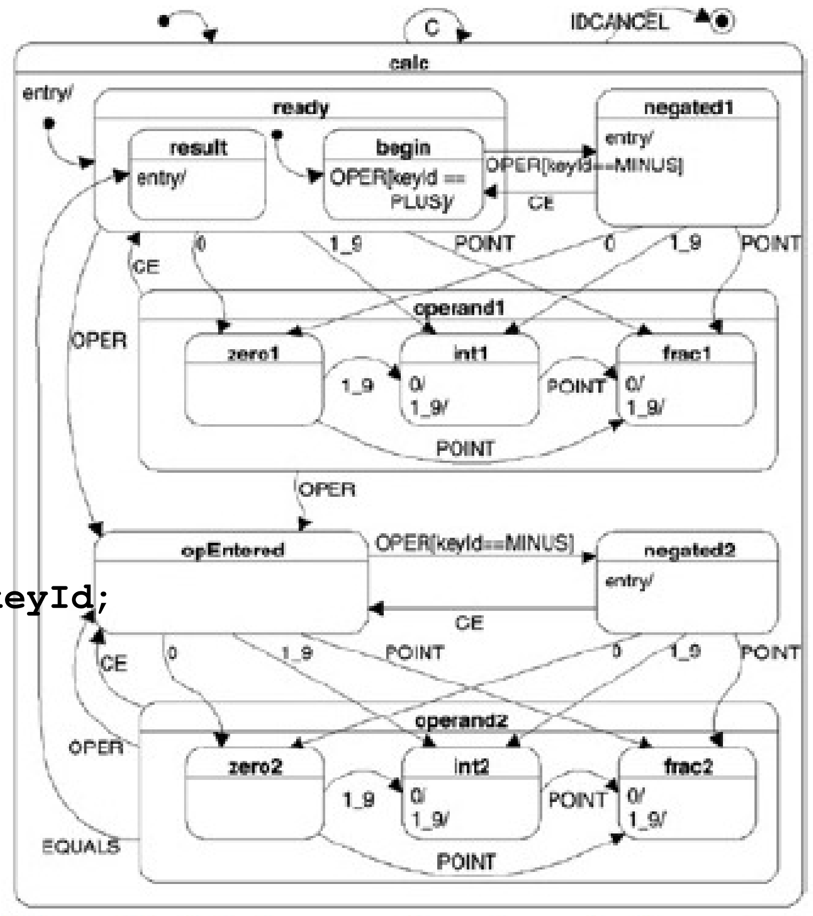
Class the function is member of

Name of pointer-to-member

Argument list

Implementing hierarchical behavior

```
STATE Calc::operand1(Event const *e) { // state-handler signature
  switch (e->sig) {
  case Q_ENTRY_SIG:
    dispState("operand1");
    return 0; // event handled
  case IDC_CE:
    clear();
    // transition to "begin"
    Q_TRAN(&Calc::begin);
    return 0; // event handled
  case IDC_OPER:
    sscanf(myDisplay, "%lf", &myOperand1);
    // downcast
    myOperator = (static_cast<CalcEvt *>(e))->keyId;
    // transition to "opEntered"
    Q_TRAN(&Calc::opEntered);
    return 0; // event handled
  }
  //event not handled, return superstate
  return (STATE)&Calc::calc;
}
```



Entry, exit and initial

```
enum {
  Q_ENTRY_SIG = 1,
  Q_EXIT_SIG,
  Q_INIT_SIG,
  Q_USER_SIG
};
```

- Define a set of reserved signals for handling entry, exit and init actions.
- These signals take up the lowest signal IDs, which are not available for clients.
- The public HSM interface contains the signal Q_USER_SIG, which indicates the first signal free for client use.
- Reserved signals start with a 1. Signal 0 is also reserved (the Empty signal Q_EMPTY_SIG), to force a state handler to return the superstate.

Entry, exit and initial

```
enum MySignals {
  MY_KEYPRESS_SIG = Q_USER_SIG,
  MY_MOUSEMOVE_SIG,
  MY_MOUSECLICK_SIG,
  . . .
};
```

- This is how user-defined signals are declared

Entry, exit and initial

- State handlers handle these signals by defining the appropriate cases in the switch statement.
- When the Q_INIT_SIG signal needs to be handled, state handlers should always invoke the Q_INIT() macro to designate the initial direct substate

```
STATE Calc::calc(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG:
    dispState("ready");          // entry action
    return 0;                    // entry action executed
  case Q_INIT_SIG:
    clear();
    Q_INIT(&Calc::ready);        // initial transition
    return 0;                   // initial transition taken
  . . .
  }
  return (STATE)&Calc::top;   // not handled, return superstate
}
```

Entry, exit and initial

A word of caution: the UML specification prescribes the following transition execution sequence:

(1) exit actions from the source state configuration,
(2) actions associated with the transition, and
(3) entry actions to the target state configuration.

- Instead, the Q_TRAN() macro executes only the exit actions from the source state configuration immediately followed by the entry actions to the target state configuration.
- This sequence does not include actions associated with the transition, which can either precede the change of state (if you define them before Q_TRAN()) or follow the change of state (if you define them after Q_TRAN()), meaning the Q_TRAN() macro performs an atomic change of state, which cannot be disrupted by any other actions.

The top state

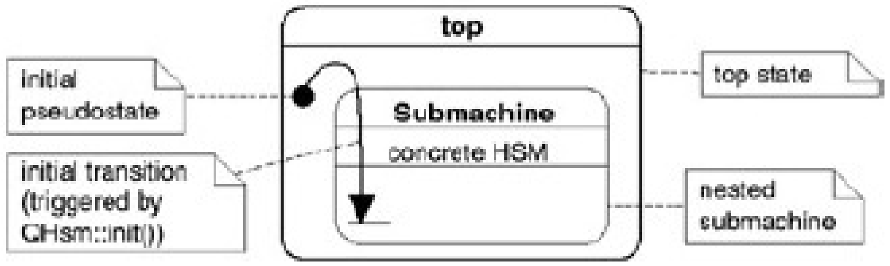
- Every HSM has the (typically implicit) top state, which contains all the other elements of the entire state machine.
- The Hsm class guarantees that the top state is available in every state machine by providing the protected Hsm::top() state handler inherited subsequently by all Hsm subclasses.
- The top state has no superstate, so the corresponding state handler always returns 0.
- Clients cannot override it (Hsm::top() is not virtual).
- The only purpose, and legitimate use, of the top state is to provide the ultimate root of a state hierarchy.

```
PseudoState top(Event const*) { return 0; } // the "top" state
```

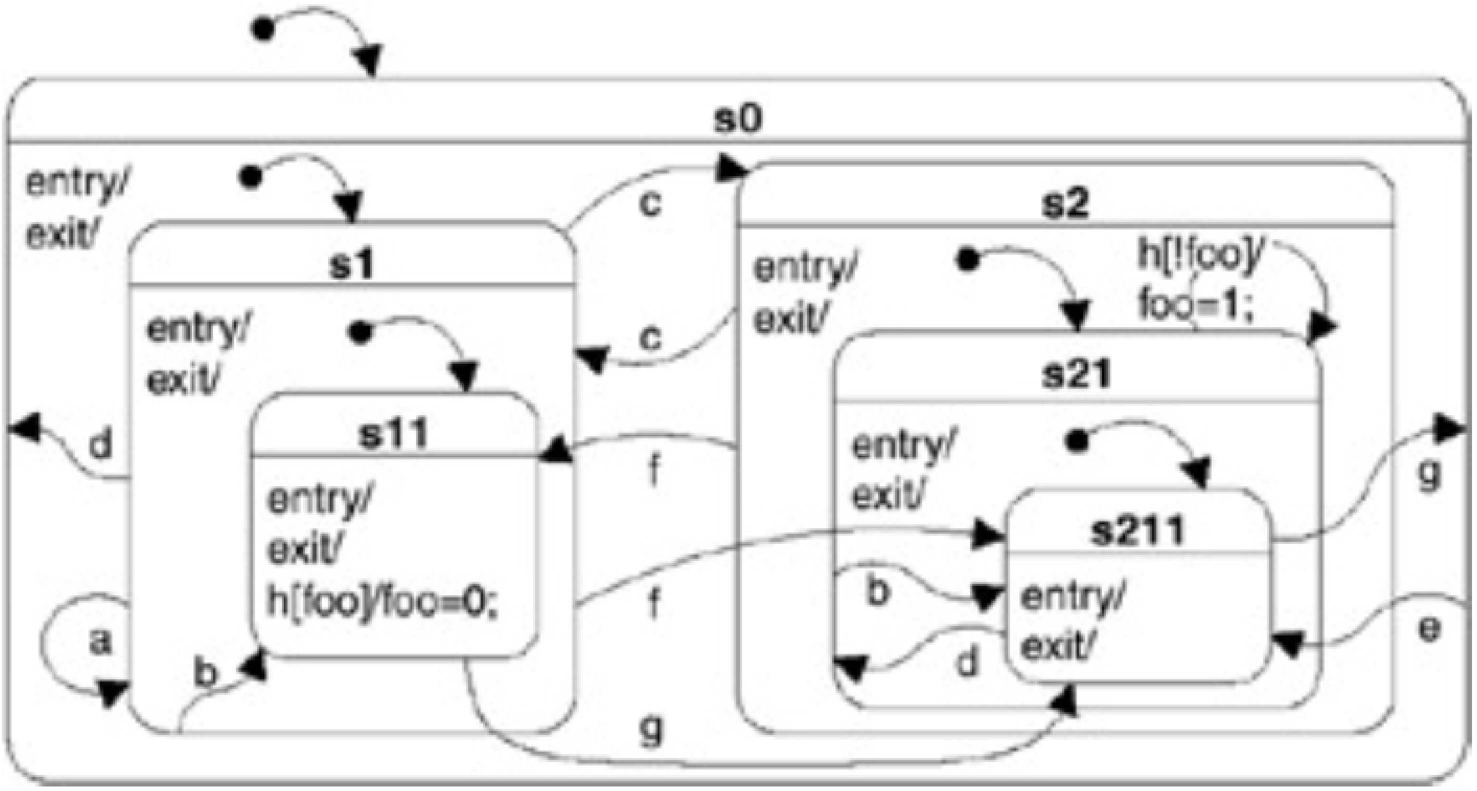
The top state

- The top state contains the initial transition. Clients must define the initial pseudostate handler for every state machine.
- The QHsm constructor requires a pointer to the initial pseudostate handler as an argument, which must designate the default state of the state machine nested inside the top state (via the Q_INIT() macro).
  - The initial transition can also specify arbitrary actions (typically initialization). The following code is an example

```
void Calc::initial (Event const *) {
  clear();                // perform initializations...
  Q_INIT(&Calc::calc); // designate the default state
}
```



Summarizing …. an example



An example

Step 1:

- The first step of the implementation consists of enumerating all signals. Remember: the user signals do not start from zero; but from Q_USER_SIG to leave space for the reserved signals (Q_ENTRY_SIG, Q_EXIT_SIG, and Q_INIT_SIG).

```
enum QHsmTstSignals {
  A_SIG = Q_USER_SIG, // user signals start
  B_SIG, C_SIG, D_SIG, E_SIG, F_SIG, G_SIG, H_SIG
};
```

An example

Step 2: Derive the concrete HSM by inheriting from QHsm

- Declare state handler methods with the predefined signature for all states in the statechart (The example statechart has six states).
- The initial pseudostate handler must be declared.

```
class QHsmTst : public Hsm { // QHsmTst derives from Hsm
public:
  QHsmTst() : Hsm((PseudoState)initial) {} // default Ctor
private:
  void initial(Event const *e); // initial pseudostate
  STATE s0(Event const *e); // state-handler
  STATE s1(Event const *e); // state-handler
  STATE s11(Event const *e); // state-handler
  STATE s2(Event const *e); // state-handler
  STATE s21(Event const *e); // state-handler
  STATE s211(Event const *e); // state-handler
private: // extended state variables...
  int myFoo;
};
```

An example

Step 3: Defining all state handler methods

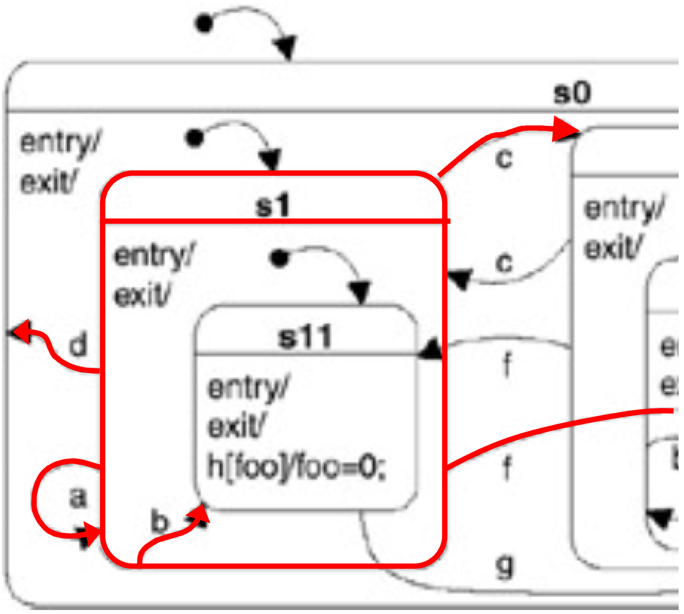
```
void QHsmTst::initial(Event const *) {
  printf("top-INIT;");
  myFoo = 0;            // initial extended state variable
  Q_INIT(&QHsmTst::s0); // initial transition
}

STATE QHsmTst::s0(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s0-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s0-EXIT;"); return 0;
  case Q_INIT_SIG: printf("s0-INIT;");
    Q_INIT(&QHsmTst::s1); return 0;
  case E_SIG: printf("s0-E;");
    Q_TRAN(&QHsmTst::s211);
    return 0;
  }
  return (STATE)&QHsmTst::top;
}
```

An example

```
STATE QHsmTst::s1(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s1-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s1-EXIT;"); return 0;
  case Q_INIT_SIG: printf("s1-INIT;"); Q_INIT(&QHsmTst::s11);
    return 0;
  case A_SIG: printf("s1-A;"); Q_TRAN(&QHsmTst::s1); return 0;
  case B_SIG: printf("s1-B;"); Q_TRAN(&QHsmTst::s11); return 0;
  case C_SIG: printf("s1-C;"); Q_TRAN(&QHsmTst::s2); return 0;
  case D_SIG: printf("s1-D;"); Q_TRAN(&QHsmTst::s0); return 0;
  case F_SIG: printf("s1-F;"); Q_TRAN(&QHsmTst::s211); return 0;
  }
  return (STATE)&QHsmTst::s0;
}
```

Look up the state in the diagram and trace around its state boundary.

You need to implement all transitions originating at this boundary, as well as all internal transitions enlisted in this state. Additionally, if an initial transition is embedded directly in the state, you need to implement it as well.



An example

Step 3: Defining state handler methods

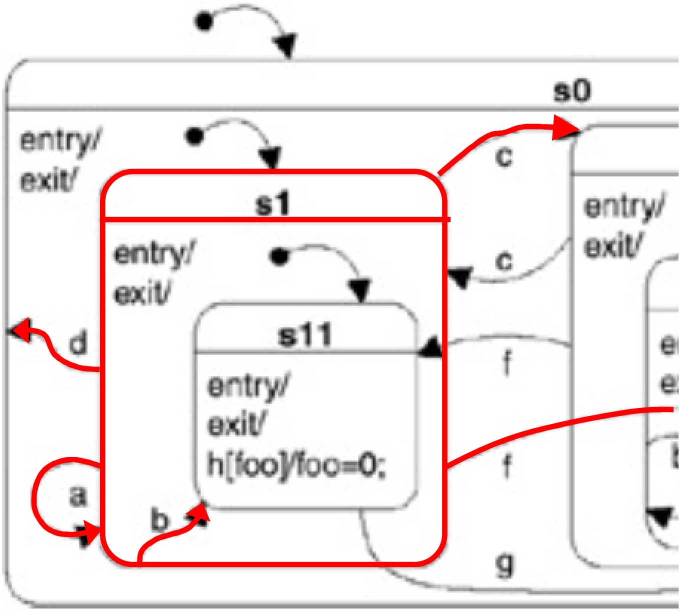
```
STATE QHsmTst::s1(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s1-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s1-EXIT;"); return 0;
  case Q_INIT_SIG: printf("s1-INIT;"); Q_INIT(&QHsmTst::s11);
    return 0;
  case A_SIG: printf("s1-A;"); Q_TRAN(&QHsmTst::s1); return 0;
  case B_SIG: printf("s1-B;"); Q_TRAN(&QHsmTst::s11); return 0;
  case C_SIG: printf("s1-C;"); Q_TRAN(&QHsmTst::s2); return 0;
  case D_SIG: printf("s1-D;"); Q_TRAN(&QHsmTst::s0); return 0;
  case F_SIG: printf("s1-F;"); Q_TRAN(&QHsmTst::s211); return 0;
  }
  return (STATE)&QHsmTst::s0;
}
```

An example

```
STATE QHsmTst::s1(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s1-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s1-EXIT;"); return 0;
  case Q_INIT_SIG: printf("s1-INIT;"); Q_INIT(&QHsmTst::s11);
    return 0;
  case A_SIG: printf("s1-A;"); Q_TRAN(&QHsmTst::s1); return 0;
  case B_SIG: printf("s1-B;"); Q_TRAN(&QHsmTst::s11); return 0;
  case C_SIG: printf("s1-C;"); Q_TRAN(&QHsmTst::s2); return 0;
  case D_SIG: printf("s1-D;"); Q_TRAN(&QHsmTst::s0); return 0;
  case F_SIG: printf("s1-F;"); Q_TRAN(&QHsmTst::s211); return 0;
  }
  return (STATE)&QHsmTst::s0;
}
```

For state s1, the transitions that originate at the boundary are transition c, d and f, internal transition b and self–transition a. In addition, the state has an entry action, an exit action, and an initial transition.

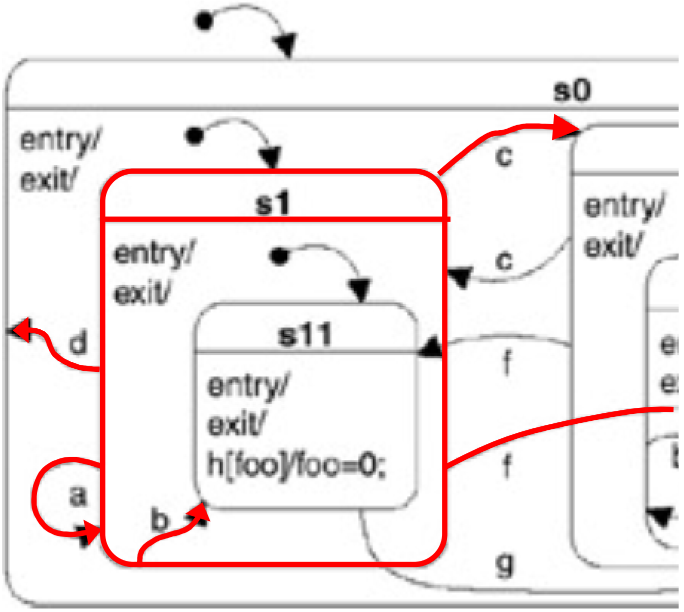
Coding of entry and exit actions is the simplest. intercept the reserved signals Q_ENTRY_SIG or Q_EXIT_SIG, enlist actions you want to execute, and terminate the lists with return 0.



An example

```
STATE QHsmTst::s1(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s1-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s1-EXIT;"); return 0;
  case Q_INIT_SIG: printf("s1-INIT;"); Q_INIT(&QHsmTst::s11);
    return 0;
  case A_SIG: printf("s1-A;"); Q_TRAN(&QHsmTst::s1); return 0;
  case B_SIG: printf("s1-B;"); Q_TRAN(&QHsmTst::s11); return 0;
  case C_SIG: printf("s1-C;"); Q_TRAN(&QHsmTst::s2); return 0;
  case D_SIG: printf("s1-D;"); Q_TRAN(&QHsmTst::s0); return 0;
  case F_SIG: printf("s1-F;"); Q_TRAN(&QHsmTst::s211); return 0;
  }
  return (STATE)&QHsmTst::s0;
}
```

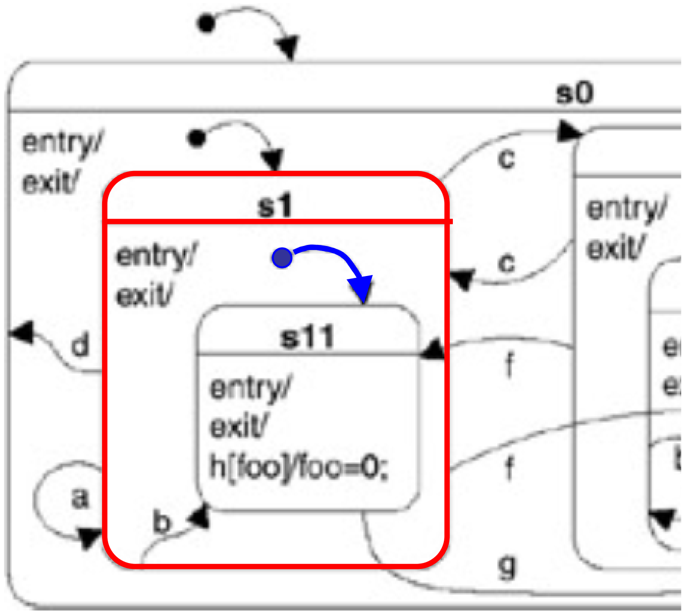
Regular transitions are coded in a very similar way, except that refer to custom–defined signals (e.g., B_SIG), and the Q_TRAN() macro is used to designate the target state. Again, the state handler is exited with return 0.



An example

```
STATE QHsmTst::s1(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s1-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s1-EXIT;"); return 0;
  case Q_INIT_SIG: printf("s1-INIT;"); Q_INIT(&QHsmTst::s11);
    return 0;
  case A_SIG: printf("s1-A;"); Q_TRAN(&QHsmTst::s1); return 0;
  case B_SIG: printf("s1-B;"); Q_TRAN(&QHsmTst::s11); return 0;
  case C_SIG: printf("s1-C;"); Q_TRAN(&QHsmTst::s2); return 0;
  case D_SIG: printf("s1-D;"); Q_TRAN(&QHsmTst::s0); return 0;
  case F_SIG: printf("s1-F;"); Q_TRAN(&QHsmTst::s211); return 0;
  }
  return (STATE)&QHsmTst::s0;
}
```

To code the initial transition, you intercept the reserved signal Q_INIT_SIG, enlist the actions, and then designate the target substate through the Q_INIT() macro, after which you exit the state handler with return 0.



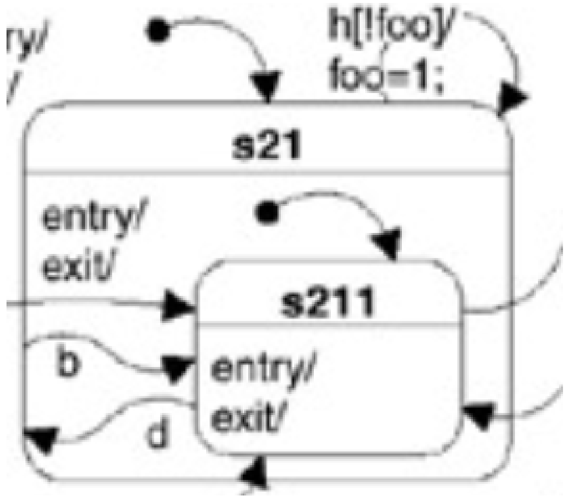
An example

Step 3: Defining state handler methods

```
STATE QHsmTst::s11(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s11-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s11-EXIT;"); return 0;
  case G_SIG: printf("s11-G;"); Q_TRAN(&QHsmTst::s211); return 0;
  case H_SIG: // internal transition with a guard
    if (myFoo) { // test the guard condition
      printf("s11-H;");
      myFoo = 0;
      return 0;
    }
    break;
  }
  return (STATE)&QHsmTst::s1;
}
```

An example

```
STATE QHsmTst::s21(Event const *e) {
  switch (e->sig) {
  case Q_ENTRY_SIG: printf("s21-ENTRY;"); return 0;
  case Q_EXIT_SIG: printf("s21-EXIT;"); return 0;
  case Q_INIT_SIG: printf("s21-INIT;"); Q_INIT(&QHsmTst::s211);
    return 0;
  case B_SIG: printf("s21-C;"); Q_TRAN(&QHsmTst::s211); return 0;
  case H_SIG: // self transition with a guard
    if (!myFoo) { // test the guard condition
      printf("s21-H;");
      myFoo = 1;
      Q_TRAN(&QHsmTst::s21); // self transition
      return 0;
    }
    break; // break to return the superstate
  }
  return (STATE)&QHsmTst::s2; // return the superstate
}
```



Transitions with guards are a little more involved. The custom signal (H_SIG), is intercepted and the guard condition is checked inside an if (…) statement. The transition actions, the call to Q_TRAN(), and return 0 are inside the TRUE branch.

When the if statement expression evaluates to FALSE the code breaks out of the switch and returns the superstate (to indicate that the event has not been handled).

The C+ implementation

`To be filled in`

An example of use (through testing)

```
static QHsmTst test; // instantiate the QHsmTst state machine
static Event const testEvt[] = { // static event instances
  {A_SIG, 0, 0}, {B_SIG, 0, 0}, {C_SIG, 0, 0}, {D_SIG, 0, 0},
  {E_SIG, 0, 0}, {F_SIG, 0, 0}, {G_SIG, 0, 0}, {H_SIG, 0, 0}
};

main() {
  printf("QHsmTst example, v.1.00, QHsm: %s\n", QHsm::getVersion());
  test.init();                         // take the initial transition
  for (;;) {                           // for-ever
    printf("\nSignal<-");
    char c = getc(stdin);
    getc(stdin);                       // discard '\n'
    if (c < 'a' || 'h' < c) {          // character out of range?
      return 0;                        // terminate
    }
    test.dispatch(&testEvt[c - 'a']); // dispatch event
  }
  return 0;
}
```

The event processor: INIT, dispatch and TRAN

init() must be called once for a given FSM before dispatching any events

(1) Triggers the initial transition defined in the initial pseudostate and

(2) Recursively "drills" into the state hierarchy until it reaches a leaf state.

```
void Hsm::init(Event const *e) {
  REQUIRE(myState == top && mySource != 0);  // HSM not executed yet

  register State s = myState;        // save myState in a temporary
  (this->*(PseudoState)mySource)(e); // top-most initial transition
                         // initial transition must go *one* level deep
  ASSERT(s == TRIGGER(myState, Q_EMPTY_SIG));
  s = myState;                                // update the temporary
  TRIGGER(s, Q_ENTRY_SIG);                         // enter the state
  while (TRIGGER(s, Q_INIT_SIG) == 0) {             // init handled?
    ASSERT(s == TRIGGER(myState, Q_EMPTY_SIG)); //1-level transition
    s = myState;
    TRIGGER(s, Q_ENTRY_SIG);                    // enter the substate
  }
}
```

The event processor: INIT, dispatch and TRAN

It uses the TRIGGER macro

```
#define TRIGGER(state_, sig_) \
Q_STATE_CAST((this->*(state_))(&pkgStdEvt[sig_]))
```

The goal of this macro is to present one of the reserved signals (Q_EMPTY_SIG, Q_ENTRY_SIG, Q_EXIT_SIG, or Q_INIT_SIG) to a given state handler, indicated by state_.

Handler method invocation is based on the pointer−to−member function ((this−>*state_)(…)). State is not recursive and the value returned by the state handler (of type pointer to PseudoState) must be cast onto QState, by the Q_STATE_CAST() macro.

- Q_STATE_CAST() is compiler dependent and should be defined as reinterpret_cast<QState>(…) for the C++ compilers that support the new type casts and as the C−style cast (QState)(…) for the C++ compilers that don't.

The event processor: INIT, dispatch and TRAN

dispatch() must scan the state hierarchy until some state handles the event (in which case, it returns 0) or the top state is reached (in which case, it also returns 0).

```
void QHsm::dispatch(QEvent const *e) {
  for (mySource = myState; mySource;
    mySource = Q_STATE_CAST((this->*mySource)(e)))
  {}
}
```

By using the mySource attribute, the current level of the hierarchy (potential source of a transition) is accessible to tran() (see next section).

The dispatch() method traverses the state hierarchy starting from the currently active state myState. It advances up the state hierarchy (i.e., from substates to superstates), invoking all the state handlers in succession.

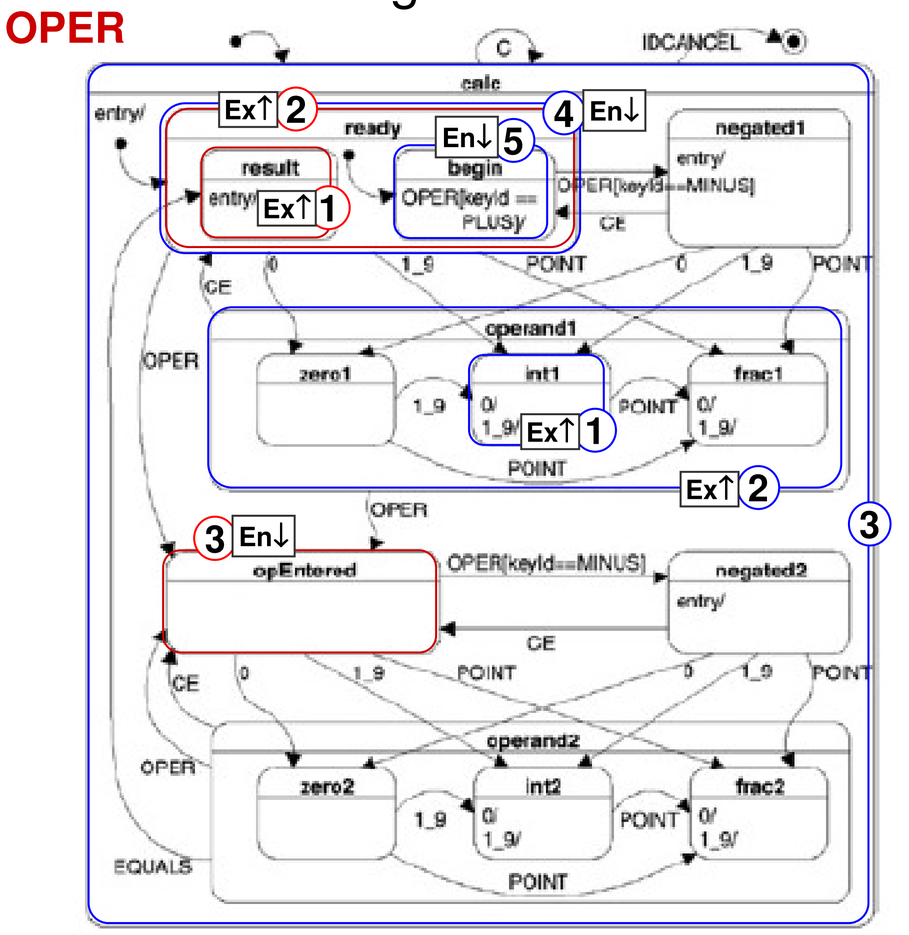
At each level of state nesting, it intercepts the value returned from a state handler to obtain the superstate needed to advance to the next level.

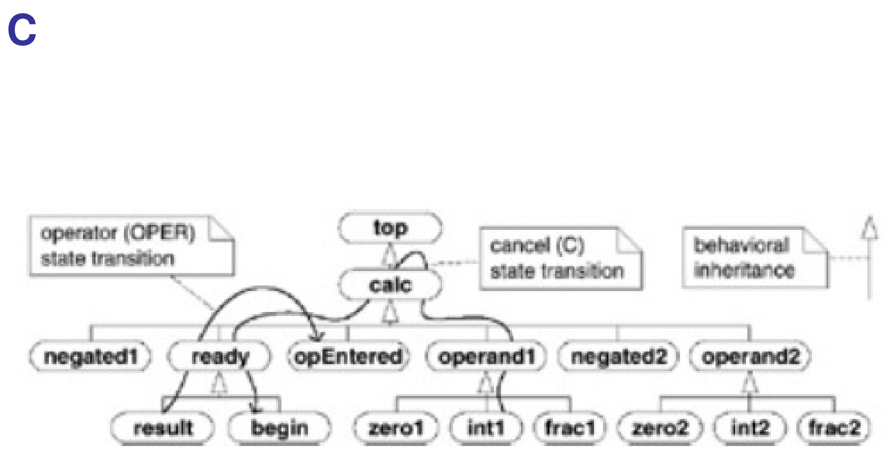
The event processor: INIT, dispatch and TRAN

- State transitions are performed by invoking the macro Q_TRAN(). At the heart of this macro is the protected tranStat() method, which actually drives state handlers to accomplish a static state transition.
- In “static” state transitions both the source and the target of the transition do not change at run time. This allows optimizing the execution of such transitions by precomputing and storing the transition chain (finding out which exit and entry actions and initial transitions to execute).
- Some other transitions need to change their targets at run time (e.g., transitions to history). For these, the QHsm class offers a dynamic state transition that you code with the Q_TRAN_DYN() macro, which invokes the protected method tran() rather than tranStat().

The event processor: INIT, dispatch and TRAN

- Executing state transitions is the most complex part of the HSM implementation. A transition execution sequence involves the exit of all states up to the least Common Ancestor (LCA), then recursive entry into the target state.





The event processor: INIT, dispatch and TRAN

- Exiting the current state configuration can be done by following the child-parent direction of navigation through the state hierarchy (state handlers return the superstate).
- The entry to the target requires navigating in the opposite direction.
- The solution to this problem is to first record the exit path from the target to the LCA, *without executing any actions*. This is done by dispatching the reserved empty signal (0), which causes every state handler to return the superstate without side effects.
- After the exit path has been recorded, it can be turned into the entry path by playing it backwards. Instead of rediscovering the entry path every time, we store it in a static object and subsequently reuse the path information.
- Dynamic transitions (coded with Q_TRAN_DYN()) must determine the transition execution sequence every time.

The event processor: INIT, dispatch and TRAN

- Hsm::tran() executes transition sequences (i.e., chains of exit and entry actions and initial transitions) by invoking the appropriate state handlers in the correct order using the appropriate standard signal for each invocation.
- Unlike init() and dispatch(), which are invoked directly by clients, tran() is protected and can be invoked only from dispatch().
- The tran() method consists of two major steps. In the first step, tran() performs a traversal of the state hierarchy similar to that of dispatch(), but with the objective to exit all states up to the level in which the transition is defined. This step covers the case of an inherited state transition — that is, the transition defined at a level higher than the currently active state.

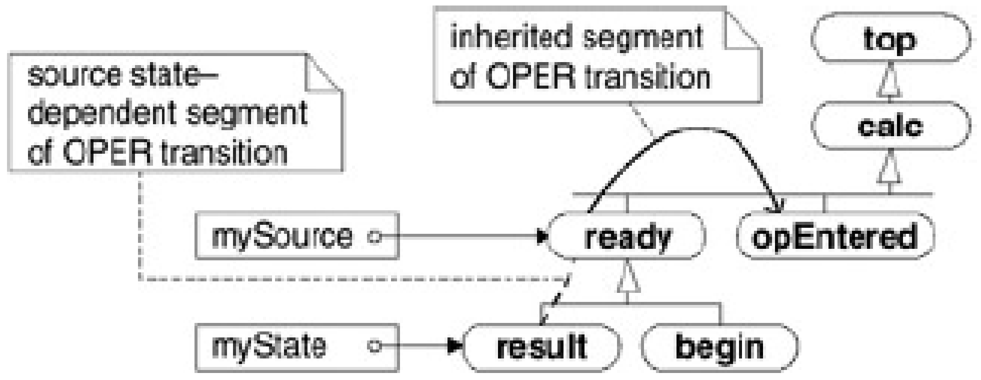
The event processor: INIT, dispatch and TRAN

In the case of OPER, this transition is defined at the level of **`ready`**, from which **`result`** and **`begin`** inherit.

When a client calls dispatch() with the OPER event, dispatch() invokes the currently active state first (result). This state does not "know" how to handle the OPER, so it returns the superstate. The dispatch() method then loops to ready, where OPER triggers the transition to opEntered.

However, the correct exit of the current state configuration must include exiting `result` and then exiting `ready` (dashed line).

The figure shows the myState and mySource pointers when dispatch() invokes tran(). myState still points to the previously active state (`result`), whereas mySource points to the state handler that invoked tran() (`ready`).



source state–dependent segment of OPER transition
inherited segment of OPER transition
top
calc
mySource
ready
opEntered
myState
result
begin

The event processor: INIT, dispatch and TRAN

```
void Hsm::tran(State target) {
  REQUIRE(target != top);           // cannot target "top" state
  State entry[8], p, q, s, *e, *lca;
  for (s = myState; s != mySource; ) {
    ASSERT(s);                      // we are about to dereference s
    QState t = TRIGGER(s, Q_EXIT_SIG);
    if (t) {     // exit action unhandled, t points to superstate
      s = t;
    } else {              // exit action handled, elicit superstate
      s = TRIGGER(s, Q_EMPTY_SIG);
    }
  } ...
```

In this first step tran() exits all states from the currently active state (myState) up to the level in which the transition is defined (mySource) to cover the case of an inherited state transition. While exiting the states, tran() must differentiate between the case in which the exit action is not handled (the state handler returns the superstate) and the case in which the exit action is executed (the state handler returns 0). In the latter case, the state handler is triggered again with the empty event to elicit the superstate.

The event processor: INIT, dispatch and TRAN

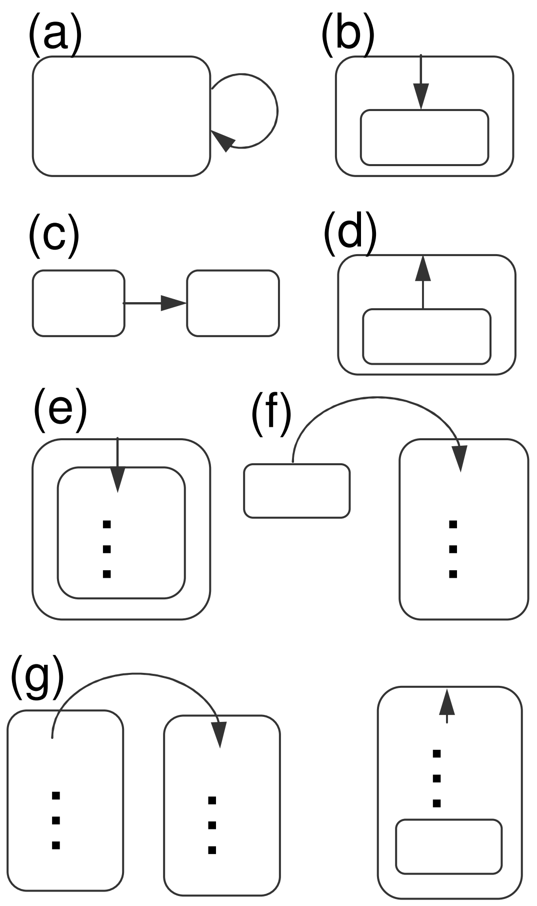
After exiting all states up to the source of the transition tran() proceeds with the second step, which is execution of the transition itself.

This step tries to optimize the workload by minimizing the number of "probing" invocations of state handlers with empty signals (i.e., with the purpose of eliciting the superstate).

The optimization relies on testing directly for all the simplest source–target state configurations, which are most likely to occur in practice. Moreover, the strategy is to order these configurations in such a way that the information about the state configuration obtained from earlier steps can be used in later steps.

The event processor: INIT, dispatch and TRAN

The figure shows such ordering of state transition topologies and the Table enlists the tests required to determine a given configuration



Step	Test	Description
a	`source == target`	(Self−transition) Can be checked directly without probing any superstates. Involves exit from source and entry to target (lines 18–22).
b	`source == target->super`	Requires probing the superstate of the `target` state. Involves only entry to source but no exit from target (lines 23–27).
c	`source->super == target->super`	(Most common transition topology) Requires additional probing of the superstate of the source state. Involves exit from source and entry to target (lines 28–33).
d	`source->super == target`	Does not require additional probing. Involves only exit from source but not entry to target (lines 34–39).
e	`source == any of target->super ...`	Requires probing the superstates of the target until a match is found or until the `top` state is reached. The `target` state hierarchy is stored in the automatic array `entry[]` (Figure 4.8) and subsequently is reused to retrace the entry in the reverse order (down the state hierarchy). This transition topology is the last that does not require exiting the original source state, so if the given transition does not fall into this category, the source must be exited (lines 40–50).
f	`source->super == any of target->super ...`	Requires traversal of the target state hierarchy stored in the array `entry[]` to find the LCA (lines 51–57). As shown in Figure 4.8, the subsequent entry proceeds from `lca-1` (line 54).
g	`any of source->super ... == any of target ...`	Requires traversal of the target state hierarchy stored in the array `entry[]` for every superstate of the source. Because every scan for a given superstate of the source exhausts all possible matches for the LCA, the source's superstate can be safely exited (lines 58–67). As shown in Figure 4.8, the subsequent entry proceeds from `lca-1` (line 62).

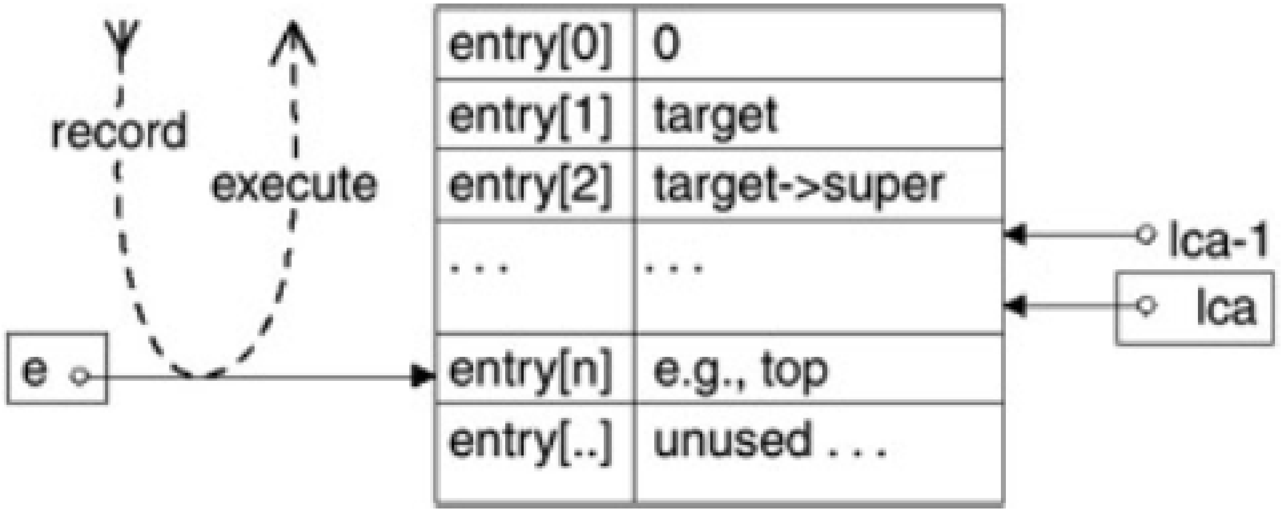
The event processor: INIT, dispatch and TRAN

In the second step, tran() executes all the actions associated with the change of state configuration.

Although this step is rather elaborate, the most frequently used source–target configurations are handled efficiently because only a small fraction of the code is executed.

The method uses the automatic array entry[] to record the entry path to the target in order to execute entry actions in the correct order.

```
State entry[8], p, q, s, *e, *lca;
```

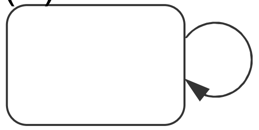


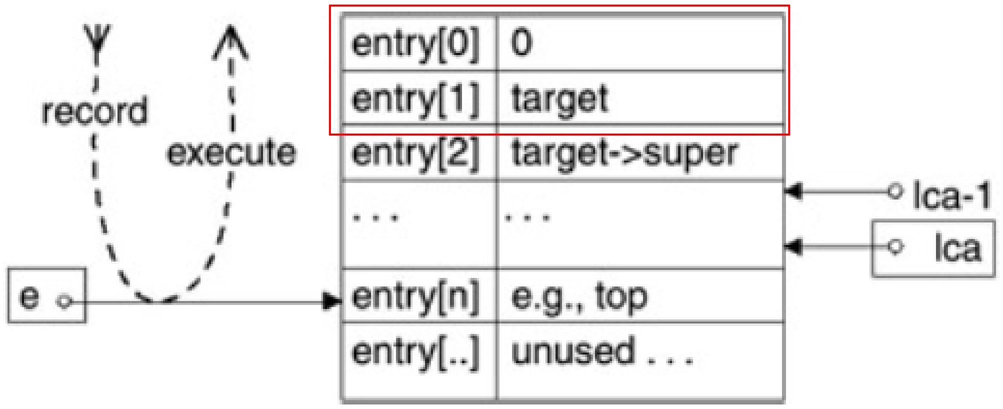
The event processor: INIT, dispatch and TRAN

Next step: tran() detects the state configuration and executes all necessary exit actions up to the LCA

```
*(e = &entry[0]) = 0;
*(++e) = target; // assume entry to target

// (a) check mySource == target (transition to self)
if (mySource == target) {
  TRIGGER(mySource, Q_EXIT_SIG); // exit source
  goto inLCA;
}
```

(a)

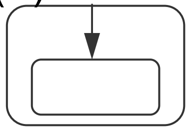




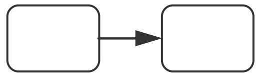
The event processor: INIT, dispatch and TRAN

```
// (b) check mySource == target->super
p = TRIGGER(target, Q_EMPTY_SIG);
if (mySource == p) {
  goto inLCA;
}
// (c) check
// mySource->super == target->super
// (most common)
q = TRIGGER(mySource, Q_EMPTY_SIG);
if (q == p) {
  TRIGGER(mySource, Q_EXIT_SIG); // exit source
  goto inLCA;
}
```

(b)



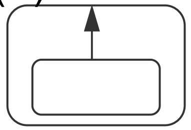
(c)



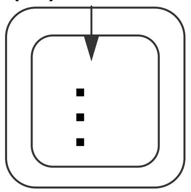
The event processor: INIT, dispatch and TRAN

```
// (d) check mySource->super == target
if (q == target) {
TRIGGER(mySource, Q_EXIT_SIG); // exit source
--e; // do not enter the LCA
goto inLCA;
}
// (e) check rest of
// mySource == target->super->super... hierarchy
*(++e) = p;
for (s = TRIGGER(p, Q_EMPTY_SIG); s;
         s = TRIGGER(s, Q_EMPTY_SIG)){
  if (mySource == s) {
    goto inLCA;
  }
  *(++e) = s;
}
TRIGGER(mySource, Q_EXIT_SIG); // all the other types start
                               // with an exit source
```

(d)

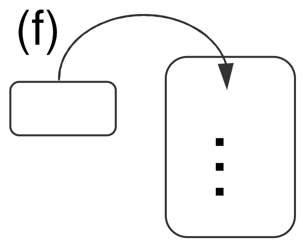


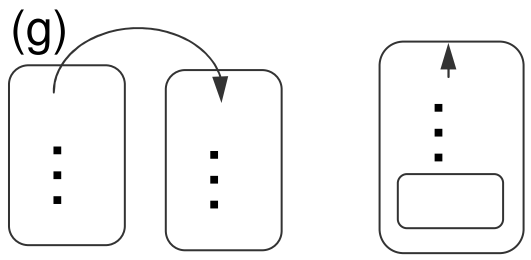
(e)



The event processor: INIT, dispatch and TRAN

```
// (f) check rest of mySource->super == target->super->super...
for (lca = e; *lca; --lca) {
  if (q == *lca) {
    e = lca - 1; // do not enter the LCA
    goto inLCA;
  }
}
// (g) check each mySource->super->super..for each target...
for (s = q; s; s = TRIGGER(s, Q_EMPTY_SIG)) {
  for (lca = e; *lca; --lca) {
    if (s == *lca) {
      e = lca - 1; // do not enter the LCA
      goto inLCA;
    }
  }
  TRIGGER(s, Q_EXIT_SIG); // exit s
}
ASSERT(0);          // malformed HSM - we should never get here
```





The event processor: INIT, dispatch and TRAN

Next step: tran() enters the target state configuration. Thanks to the entry path saved in step e, this is straightforward. The assertion checks that the automatic array entry[] does not overflow, which can happen if the transition chain has more than seven steps.

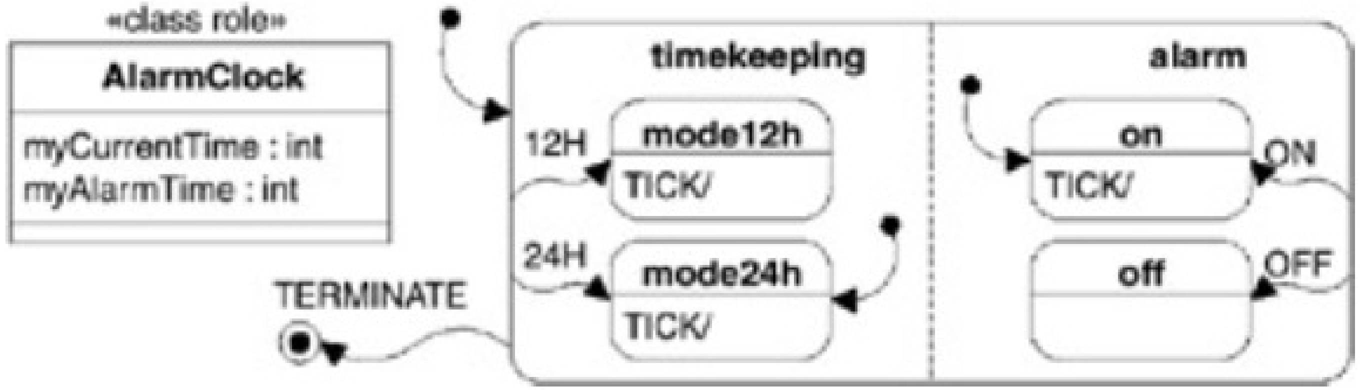
```
inLCA:          // now we are in the LCA of mySource and target
  ASSERT(e < &entry[DIM(entry)]); // entry array must not overflow
  while (s = *e--) {    // retrace the entry path in reverse order
    TRIGGER(s, Q_ENTRY_SIG);                 // call entry action on s
  }
  myState = target;                           // update current state
```

The event processor: INIT, dispatch and TRAN

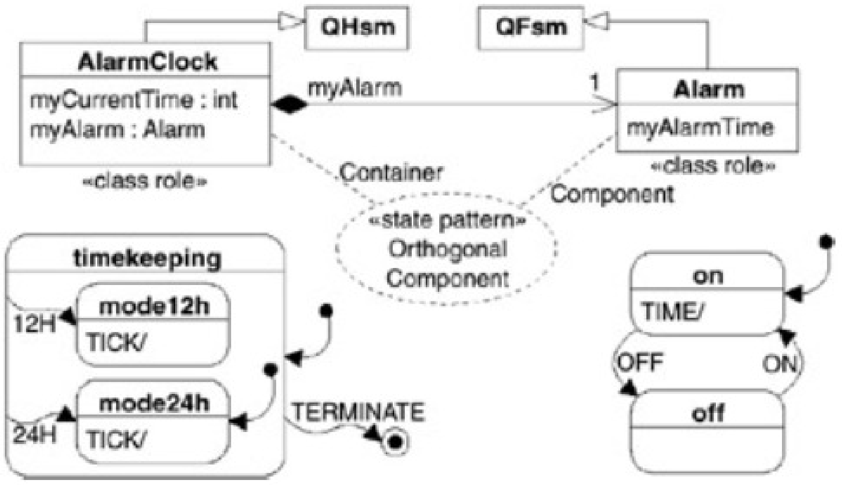
Finally, the target state can be composite and can have an initial transition. Therefore, tran() iterates until it detects a leaf state (the initial transition returns non–0) and executes initial actions (inside init).

```
  ...
  while (TRIGGER(target, Q_INIT_SIG) == 0) {
                    // initial transition must go *one* level deep
    ASSERT(target == TRIGGER(myState, Q_EMPTY_SIG));
    target = myState;
    TRIGGER(target, Q_ENTRY_SIG);                          // enter target
  }
}
```

Concurrent states



Proposed solution: use object composition



Concurrent states

The use of aggregation in conjunction with state machines raises questions.

1. How does the container state machine communicate with the component state machines?
2. How do the component state machines communicate with the container state machine?
3. What kind of concurrency model should be used?

The composite object interacts with its aggregate parts by synchronously dispatching events to them (by invoking dispatch() on behalf of the components).

To communicate in the opposite direction (from a component to the container), a component needs to post events (in the queue) to the container. A child cannot call dispatch() on behalf of the parent because it would violate RTC semantics.

Concurrent states

- The parent dispatches events synchronously (without queuing them) to the children, but the children must post events asynchronously (by queuing them) to the parent.
- This way of communication dictates a concurrency model in which a parent shares its execution thread with the children.
- The parent dispatches an event to a child by synchronously calling dispatch() on behalf of the child. Because this method executes in the parent's thread, the parent cannot proceed until dispatch() returns (i.e., until the child finishes its RTC step).
- In this way, the parent and children can safely share data without any concurrency hazards (data sharing is also another method of communication among them).

Concurrent states

The sample code demonstrates the typical code organization for the Orthogonal Component state pattern, in which the component (Alarm) is implemented in a separate module from the container (AlarmClock).

Common signals and events

```
enum AlarmClockSignals {
  TIME_SIG = Q_USER_SIG,
  ALARM_SIG, TERMINATE
};
struct AlarmInitEvt : public QEvent {
  HWND hWnd;
};
struct TimeEvt : public QEvent {
  unsigned currentTime;
};
```

Concurrent states

Alarm (component) FSM declaration

```
class Alarm : public Fsm {
 public:
   Alarm() : Fsm((FsmState)initial) {}
 private:
   void initial(Event const *e);
   void on(Event const *e);
   void off(Event const *e);
 private:
   unsigned myAlarmTime; // time to trigger the alarm
   HWND myHwnd;          // window handle
};
```

Concurrent states

Alarm (component) FSM implementation

```
void Alarm::initial(QEvent const *e) {
myHwnd = (static_cast<AlarmInitEvt const *>(e))->hWnd;
  . . .
  QFSM_TRAN(&Alarm::on);
}
void Alarm::on(QEvent const *e) {
  switch (e->sig) {
   case TIME_SIG:
     if ((static_cast<TimeEvt *>(e))->currentTime == myAlarmTime) {
       Beep(1000, 20);
       PostMessage(myHwnd, WM_COMMAND, ALARM_SIG, 0); // notify
     }
     return;
   case IDC_OFF:
     . . .
    QFSM_TRAN(&Alarm::off);
    return;
  }
}
```

Concurrent states

Alarm (component) FSM implementation

```
void Alarm::off(QEvent const *e) {
  char buf[12];
  unsigned h, m;
  switch (e->sig) {
   case IDC_ON:
     GetDlgItemText(myHwnd, IDC_ALARM, buf, sizeof(buf));
     if (...) { // does the user input represent valid alarm time?
     . . .
     QFSM_TRAN(&Alarm::on);
   }
   return;
  }
}
```

Concurrent states

AlarmClock (composite) HFSM implementation

```
#include "clock.h"
#include "alarm.h"

class AlarmClock : public Hsm {       // hierarchical state machine
 public:
   AlarmClock() : Hsm((QPseudoState)initial) {}
 private:
   void initial(QEvent const *e);              // initial pseudostate
   STATE timekeeping(QEvent const *e);         // state-handler
   STATE mode12hr(QEvent const *e);            // state-handler
   STATE mode24hr(QEvent const *e);            // state-handler
   STATE final(QEvent const *e);               // state-handler
 private:
   unsigned myCurrentTime;                 // current time (in minutes)
   Alarm myAlarm;                          // reactive component Alarm
   BOOL isHandled;
   friend class Alarm; // grant friendship to reactive component(s)
   ...
};
```

Concurrent states

AlarmClock HFSM implementation

```
void AlarmClock::initial(Event const *) {
  . . .
  AlarmInitEvt ie; // initialization event for the Alarm component
  ie.wndHwnd = myHwnd;
  myAlarm.init(&ie); // initial transition in the alarm component
  Q_INIT(timekeeping);
}

STATE AlarmClock::timekeeping(Event const *e) {
  switch (e->sig) {
   . . .
   case IDC_ON:
   case IDC_OFF:
     myAlarm.dispatch(e); // dispatch event to orthogonal component
     return 0;
  }
  return (QSTATE)top;
}
```

Concurrent states

AlarmClock HFSM implementation

```
STATE AlarmClock::mode24hr(Event const *e) {
  TimeEvt pe;                             // temporary for propagated event
  switch (e->sig) {
      . . .
   case WM_TIMER:
      . . .                               // update myCurrentTime
     pe.sig = TIME_SIG;
     pe.currentTime = myCurrentTime;
     myAlarm.dispatch(&pe);   //dispatch event to orthogonal component
     return 0;
   }
   return (STATE)timekeeping;
}
```

Concurrent states

The AlarmClock class container has several responsibilities toward its components.

- The initialization of the Alarm component's state machine, which is best accomplished in the initial transition of the container
- The explicit dispatching of events to the component(s) (lines 36, 50).
  - While (explicitly) dispatching event to orthogonal regions the container can even change the event type on the fly. For instance, AlarmClock dispatches events ON and OFF to its myAlarm component as they arrive (lines 34–37).
  - The WM_TIMER signal is handled differently (lines 46–51). In this case, AlarmClock synthesizes a TimeEvt event on the fly, furnishes the current time, and dispatches this event to the Alarm component. Note that TimeEvt can be allocated automatically (on the stack) because it is dispatched synchronously to the component.

Concurrent states

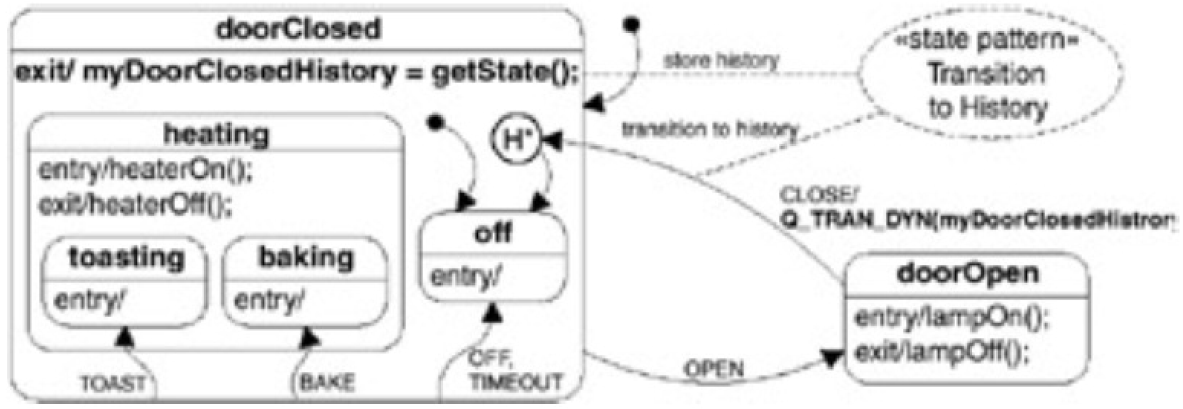
The Orthogonal Component state pattern partitions independent islands of behavior into separate reactive objects and has the following consequences.

- Partitioning introduces a container component (also known as parent–child or master–slave) relationship. The container implements the primary functionality and delegates other (secondary) features to the components. Both the container and the components are state machines.
- The container shares its execution thread with the components.
- Deals nicely with several components of the same type
- The container communicates with the components by directly dispatching events to them. The components notify the container by posting events to it, never through direct event dispatching.
- The components typically use the Reminder state pattern to notify the container (i.e., the notification events are invented specifically for the internal communication and are not relevant externally). If there are more components of a given type, then the notification events must identify the originating component (the component passes its ID in a parameter of the notification event).
- *More …*

Concurrent states

- More …
- The container and components can share data. Typically, the data is an attribute of the container (to allow multiple instances of different containers). The container typically grants friendship to the selected components.
- The container is entirely responsible for its components. In particular, it must explicitly trigger initial transitions in all components, as well as explicitly dispatch events to the components. Errors may arise if the container "forgets" to dispatch events to some components in some of its states.
- The container has full control over the dispatching of events to the components. It can choose not to dispatch events that are irrelevant to the components. It can also change event types on the fly and provide some additional information to the components.
- The container can dynamically start and stop components (e.g., in certain states of the component state machine).

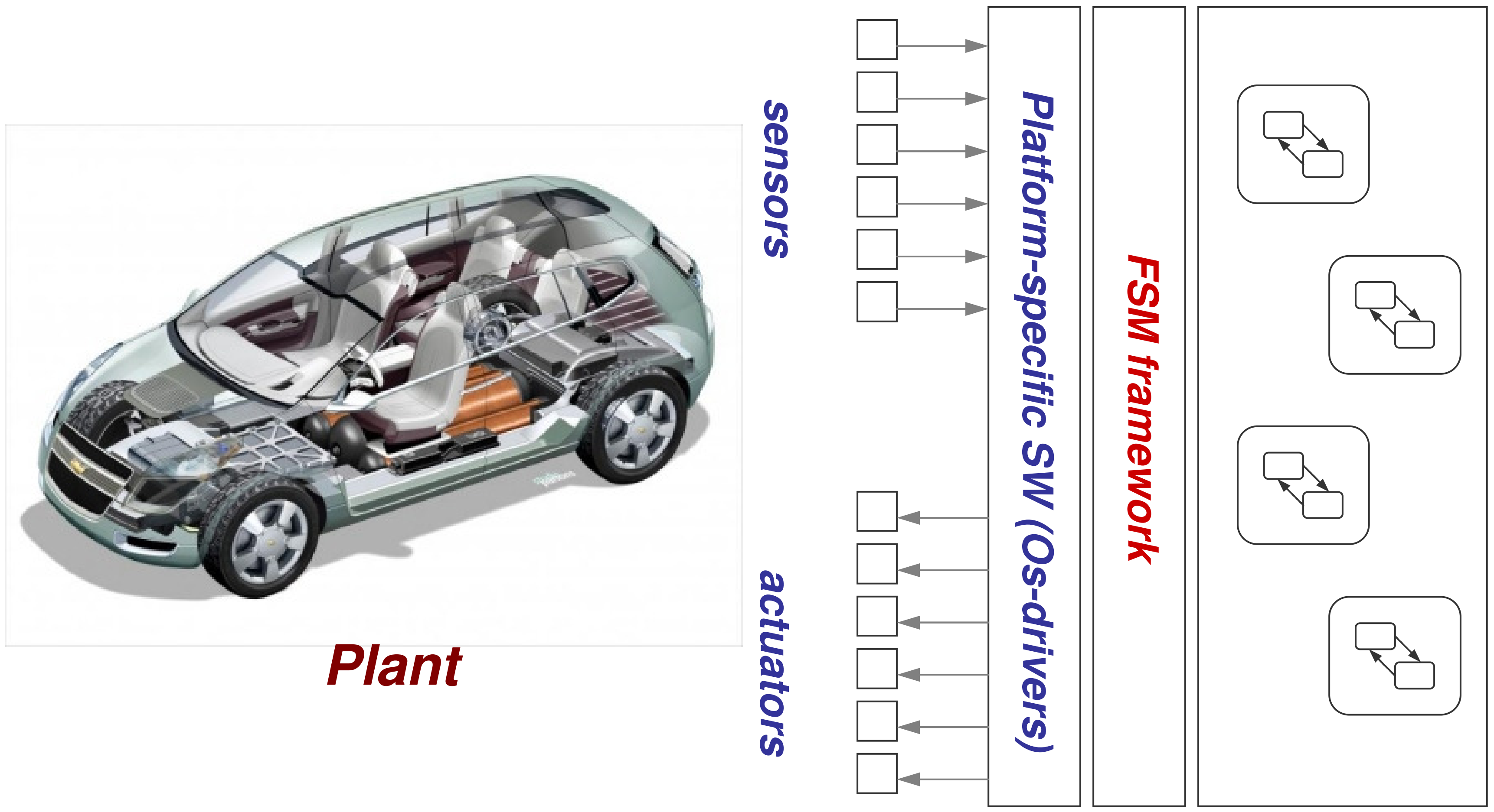
History states



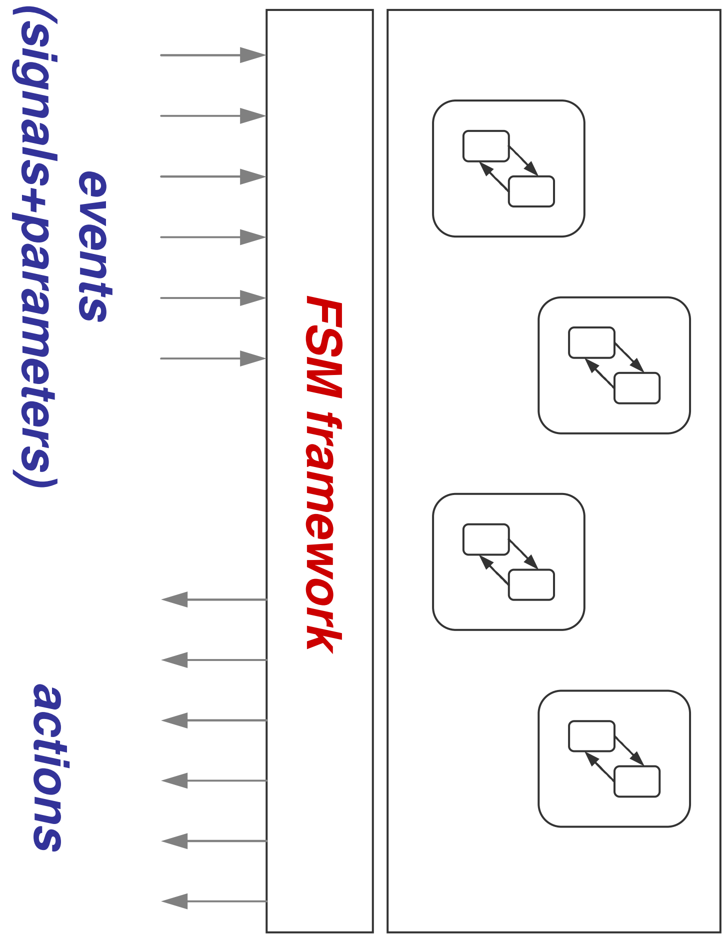
The solution consists in storing the most recently active substate of the doorClosed state in the dedicated attribute myDoorClosedHistory. The ideal place for setting this attribute is the exit action from the doorClosed state.

Subsequently, the transition to history of the doorOpen state (transition to the circled H*) uses the attribute as the target of the transition. Because this target changes at run time, it is crucial to code this transition with the Q_TRAN_DYN() macro, rather than the optimized Q_TRAN()

Outer framework



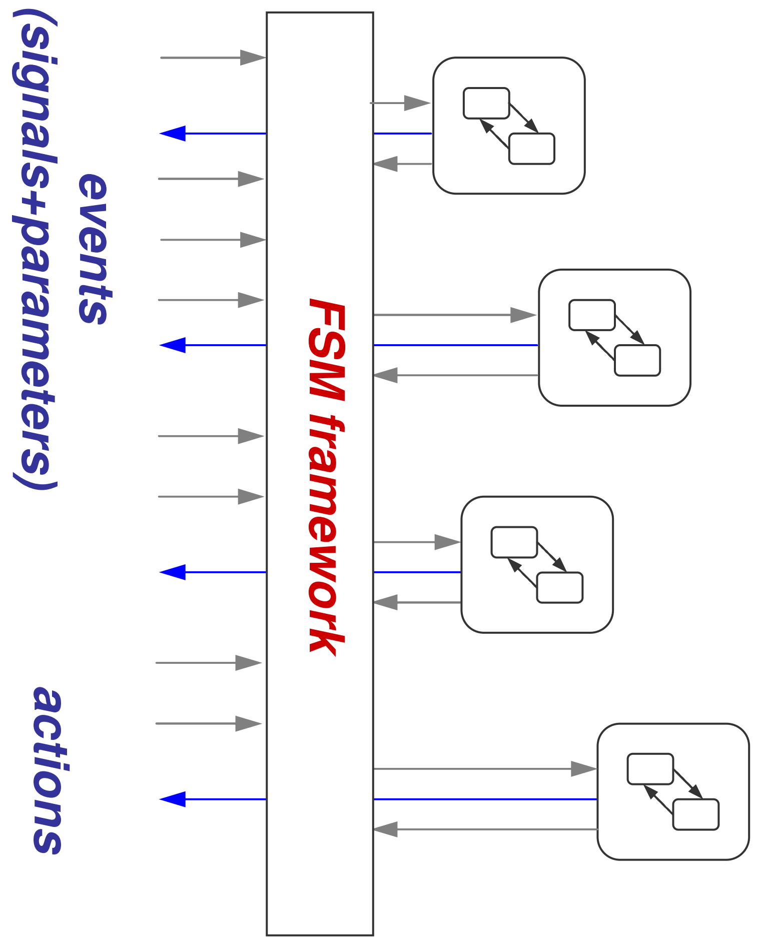
Outer framework



Issues:

- How to code events
- How to execute reactions
- How to forward events
- How to synchronize the execution of FSMs
- How to map into a task model
- How to solve issues wrt critical sections, timing, priority inversion

Outer framework: some models



The framework as an event broker

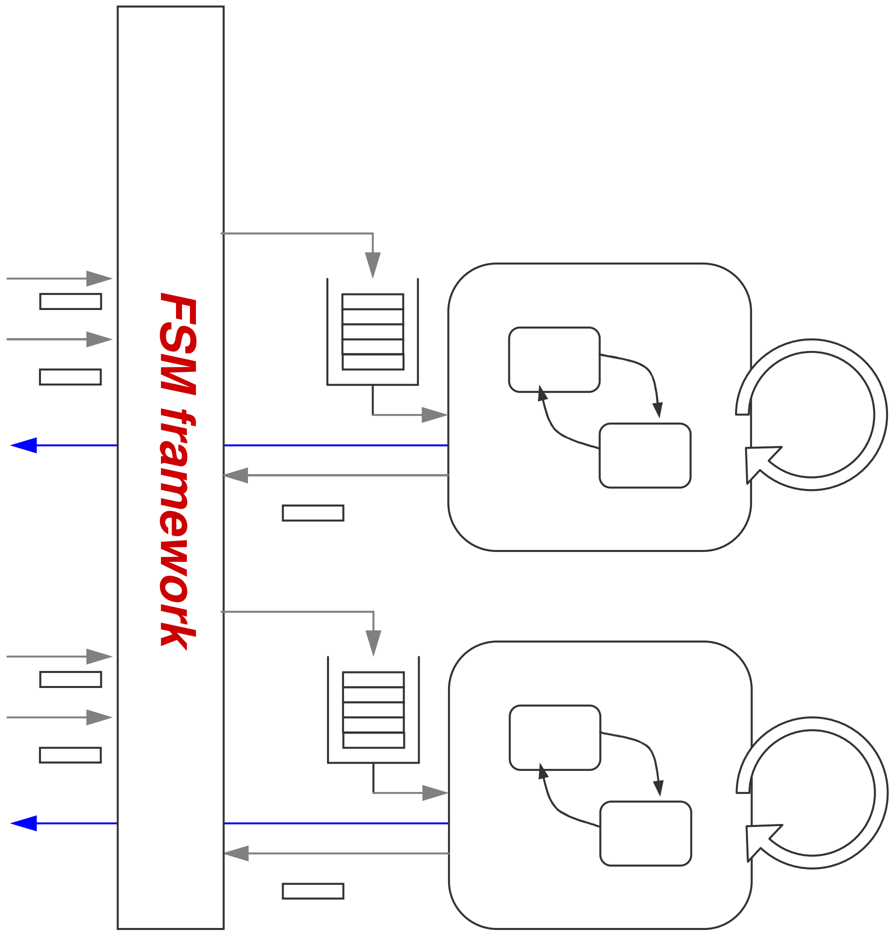
It gets events from the platform-dependent layer and from (FSM) subsystems

It forwards events to FSM that need to process them

Publish-subscribe model

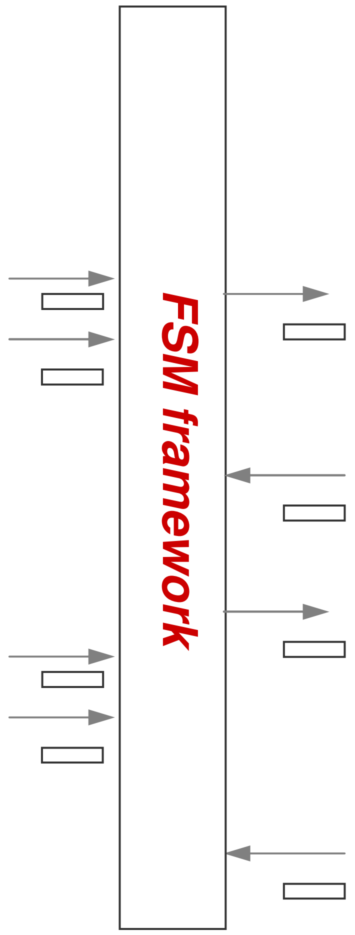
The framework is transparent wrt actions

Outer framework: Asynchronous task model



- One task for each FSM
- The task is cyclic and processes input events from a queue
- FSM communicate asynchronously (no synchronization among FSMs)
- Advantage: no need to protect state variables, run-to-completion semantics guaranteed by tasking model
- Problems (possible) with time behavior
  - Queue is typically FIFO
- No synchronization
- Framework must coordinate signal forwarding

Outer framework: Asynchronous task model



- The Framework:
- Events
  - Event definition/basic mngmt
    - Events are dynamic, they are created, managed and destroyed
  - Event forwarding
    - Collecting events from FSM components
    - Sending events to FSM components
  - Event processing by FSMs
    - Event queuing
    - Event processing
    - Event generation
- Execution of reactions, tasking model for FSMs

Outer framework: Asynchronous task model

(Samek’s model)

- Step 1: Event definition/basic management
- Events are structures consisting of a pure signal and a (variable, event-depending) set of data attributes
- Example: mouse click on the screen
  - Signal: mouse (left/right) button down
  - Attributes: (x,y) position on the screen (unsigned short, unsigned short)

```
typedef unsigned short Signal; // Quantum Signal
struct Event { // Quantum Event
  Signal sig;
  . . .
};
```

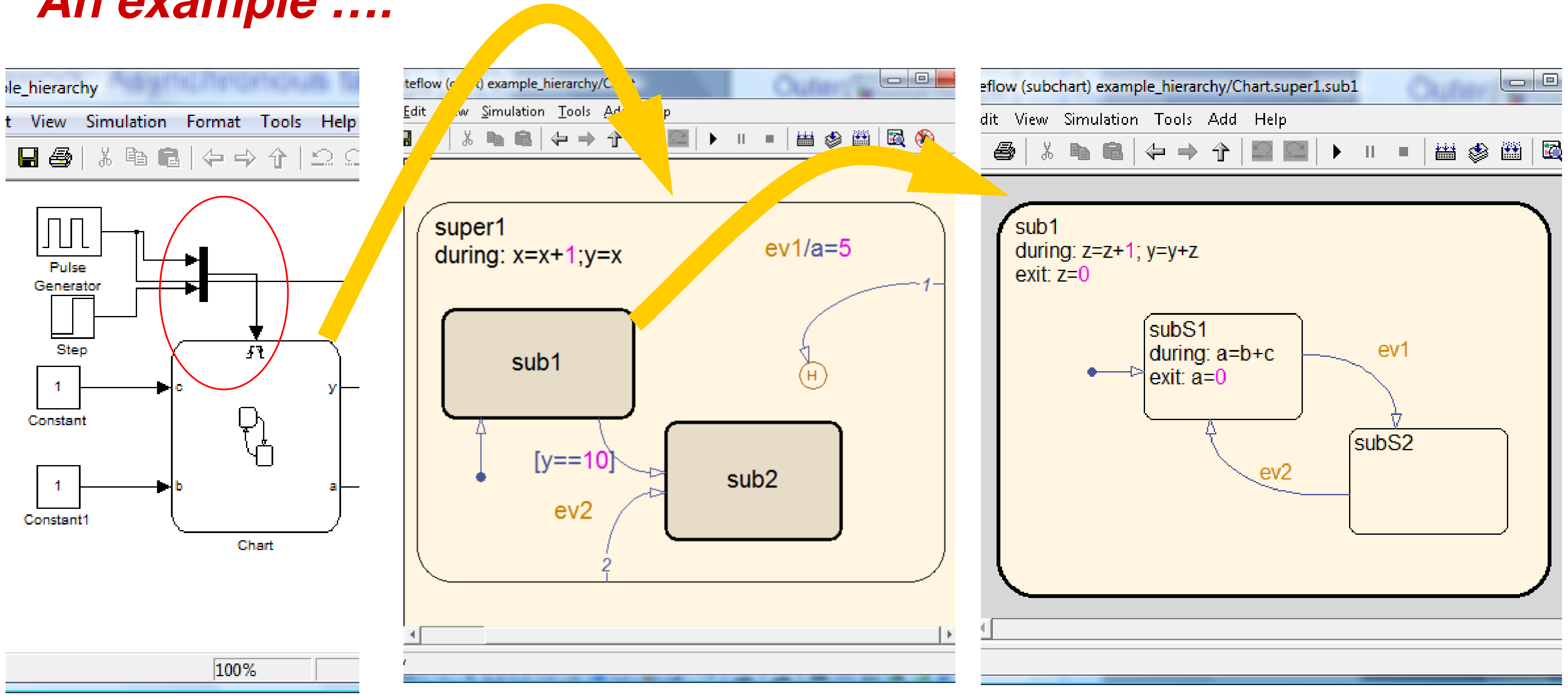
(Event may be subclassed for additional attributes)

Outer framework: Asynchronous task model

Alternative model (Simulink/stateflow)

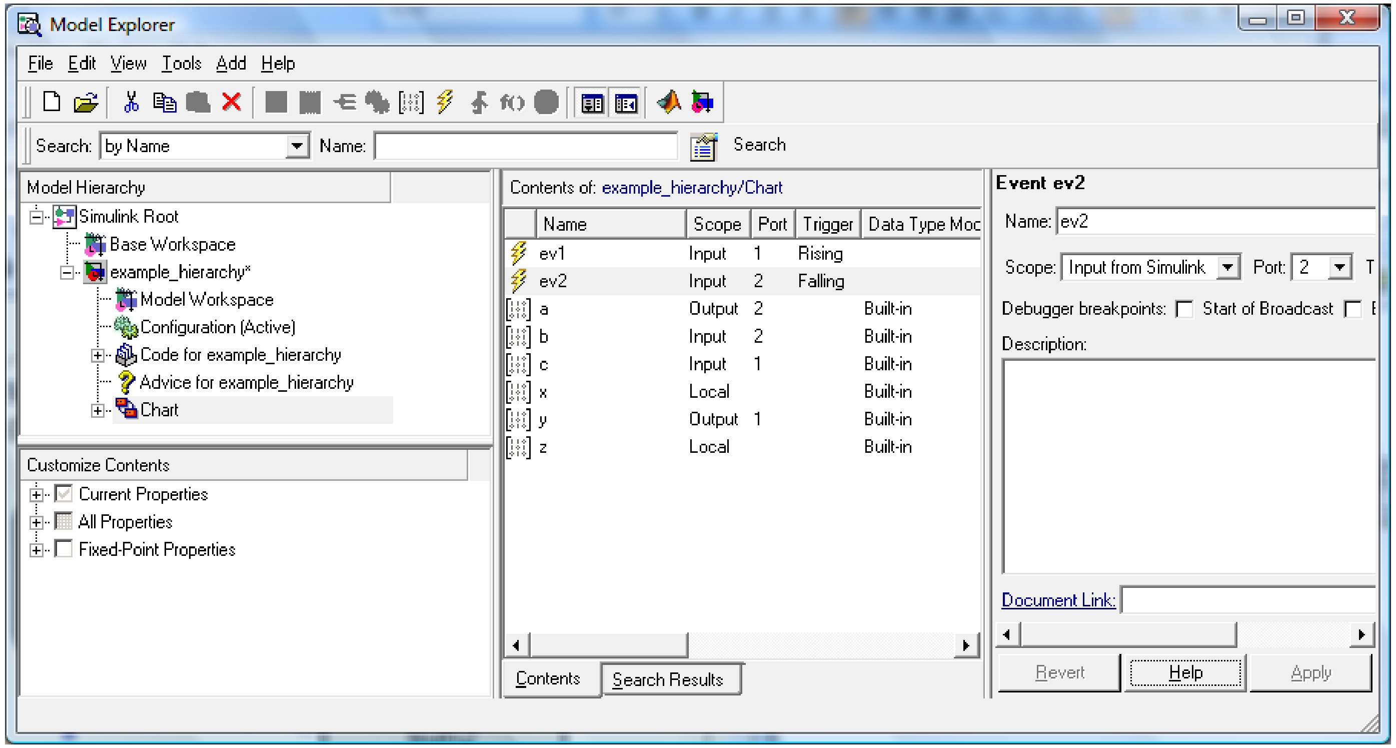
Coding of signals is separated from coding of data (events are “pure signals”), Signals are typically extracted from data ….

An example ….



Outer framework: Asynchronous task model

Signals are extracted from data ….
(example: from rising edge – ev1 – and falling edge – ev2 -)



Contents of: example_hierarchy/Chart

Name	Scope	Port	Trigger	Data Type Moc
ev1	Input	1	Rising	
ev2	Input	2	Falling	
a	Output	2		Built-in
b	Input	2		Built-in
c	Input	1		Built-in
x	Local			Built-in
y	Output	1		Built-in
z	Local			Built-in

Event ev2

Name: ev2

Scope: Input from Simulink   Port: 2   T

Debugger breakpoints: ☐ Start of Broadcast ☐ E

Description:

Document Link:

Outer framework: Asynchronous task model

Generated code: Chart.c

Signal/event encoding

```
/* Named constants for block: '<Root>/Chart' */
#define Chart_event_ev1                    (1U)
#define Chart_event_ev2                    (0U)
```

Outer framework: Asynchronous task model

Generated code: Chart.c

Main processing (extracting events from inputevents)

```
/* Output and update for trigger system: '<Root>/Chart' */
void Chart_Chart(void)
{
  /* local block i/o variables */
  int8_T rtb_inputevents[2];
  {
    ZCEventType trigEvent;
    ZCEventType zcEvents[2] ;
    /* subsystem trigger input */
    trigEvent = NO_ZCEVENT;
    zcEvents[0] = rt_ZCFcn(ANY_ZERO_CROSSING,&(Chart_PrevZCSigState.Chart_ZCE[0]),
      Chart_U.inputevents[0]);
    trigEvent = (zcEvents[0] == NO_ZCEVENT)? trigEvent : zcEvents[0];
    zcEvents[1] = rt_ZCFcn(ANY_ZERO_CROSSING,&(Chart_PrevZCSigState.Chart_ZCE[1]),
      Chart_U.inputevents[1]);
    trigEvent = (zcEvents[1] == NO_ZCEVENT)? trigEvent : zcEvents[1];

 /* conditionally execute */
    if (trigEvent != NO_ZCEVENT) {
      ...
```

Outer framework: Stateflow

Generated code: Main

```c
/* The example "main" function illustrates what is required by the application to
 * initialize, execute, and terminate the generated code. Attaching rt_OneStep to
 * a real-time clock is target specific. This example illustrates how to do this
 */
int_T main(int_T argc, const char_T *argv[])
{
  /* Initialize model */
  Controller_initialize();

  /* Attach rt_OneStep to a timer or interrupt service routine with
   * period 1.0 seconds (the model's base sample time) here.  The
   * call syntax for rt_OneStep is
   *
   * rt_OneStep();
   */
  while (rtmGetErrorStatus(Controller_M) == NULL) {
    /*  Perform other application tasks here */
  }

  /* Disable rt_OneStep() here */

  /* Terminate model */
  Controller_terminate();
  return 0;
}
```

Outer framework: Stateflow

Generated code: rt_OneStep

```
/* Associating rt_OneStep with a real-time clock or interrupt
 * service routine is what makes the generated code "real-time".
 * The function rt_OneStep is always associated with the base rate
 * of the model.  Subrates are managed by the base rate from
 * inside the generated code.  Enabling/disabling interrupts and
 * floating point context switches are target specific.  This
 * example code indicates where these should take place relative
 * to executing the generated code step function. Overrun behavior
 * should be tailored to your application needs.  This example
 * simply sets an error status in the real-time model and returns
 * from rt_OneStep.
 */

void rt_OneStep(void)
{
    ...
}
```

Outer framework: Stateflow

Generated code: rt_OneStep

```
void rt_OneStep(void)
{
  /* Disable interrupts here */
  /* Check base rate for overrun */
  /* Save FPU context here (if necessary) */
  /* Re-enable timer or interrupt here */

  /*
   * For a bare-board target (i.e., no operating system), the rates
   * that execute this base step are buffered locally to allow for
   * overlapping preemption.  The generated code includes function
   * Controller_SetEventsForThisBaseStep() which sets the rates
   * that need to run this time step.  The return values are 1 and 0
   * for true and false, respectively.
   */

  /* Set model inputs associated with base rate here */
  /* Get model outputs associated with base rate here */
  /* Check subrates for overrun */
  /* Set model inputs associated with subrates here */
  /* Get model outputs associated with subrates here */
  /* Disable interrupts here */
  /* Restore FPU context here (if necessary) */
  /* Enable interrupts here */
}
```

Outer framework: Stateflow

Generated code: rt_OneStep

```
static boolean_T OverrunFlags[2] = { 0, 0 };

void rt_OneStep(void)
{
 boolean_T eventFlags[2];             /* Model has 2 rates */

  /* Check base rate for overrun */
  if (OverrunFlags[0]++) {
    rtmSetErrorStatus(Controller_M, "Overrun");
    return;
  }
  Controller_SetEventsForThisBaseStep(eventFlags);
  /* Set model inputs associated with base rate here */
  Controller_step0();
  /* Get model outputs associated with base rate here */
  OverrunFlags[0]--;
  /* Check subrates for overrun */
  if (eventFlags[1]) {
    if (OverrunFlags[1]++) {
      rtmSetErrorStatus(Controller_M, "Overrun");
      return;
    }
    /* Set model inputs associated with subrates here */
    Controller_step1();
    /* Get model outputs associated with subrates here */
    OverrunFlags[1]--;
  }
}
```

Outer framework: Stateflow

Generated code: rt_OneStep

```
static boolean_T OverrunFlags[2] = { 0, 0 };

void rt_OneStep(void)
{
 boolean_T eventFlags[2];             /* Model has 2 rates */

  /* Check base rate for overrun */
  if (OverrunFlags[0]++) {
    rtmSetErrorStatus(Controller_M, "Overrun");
    return;
  }
  Controller_SetEventsForThisBaseStep(eventFlags);
  /* Set model inputs associated with base rate here */
  Controller_step0();
  /* Get model outputs associated with base rate here */
  OverrunFlags[0]--;
  /* Check subrates for overrun */
  if (eventFlags[1]) {
    if (OverrunFlags[1]++) {
      rtmSetErrorStatus(Controller_M, "Overrun");
      return;
    }
    /* Set model inputs associated with subrates here */
    Controller_step1();
    /* Get model outputs associated with subrates here */
    OverrunFlags[1]--;
  }
}
```

Outer framework: Stateflow

Generated code: Controller_step0

```
void Controller_step0(void)              /* Sample time: [1.0s, 0.0s] */
{
  ...// Data declarations
  {                                      /* Sample time: [1.0s, 0.0s] */
    rate_monotonic_scheduler();
  }
  ...// Model execution
}
/* This function updates active task flag for each subrate and rate transition flags for tasks
 * that exchagne data. The function assumes rate-monotonic multitasking scheduler.
 */
static void rate_monotonic_scheduler(void)
{
  /* Data is transfered at the priority of a fast task and the frequency of the slow task.
   The flags indicate when the data transfer happens.  That is, a rate interaction flag is set
   true when both rates will run, and false otherwise.
   */

  /* tid 0 shares data with slower tid rate: 1 */
  Controller_M->Timing.RateInteraction.TID0_1= (Controller_M->Timing.TaskCounters.TID[1]==0);

  /* Compute which subrates run during the next base time step. The subtask counter is reset
   when it reaches its limit (zero means run).
   */
  if (++Controller_M->Timing.TaskCounters.TID[1] == 10) {/* Sample time: [10.0s, 0.0s] */
    Controller_M->Timing.TaskCounters.TID[1] = 0;
  }
}
```

Outer framework: Stateflow

Generated code: Controller_step0

```
void Controller_step0(void)            /* Sample time: [1.0s, 0.0s] */
{
  ...

  /* DataTypeConversion: '<S2>/Data Type Conversion' */
  Controller_B.DataTypeConversion = (boolean_T)(rtb_Add != 0.0 ? 1U : 0U);

  /* DiscretePulseGenerator: '<S1>/Pulse Generator' */
  rtb_Add =
    (Controller_DWork.clockTickCounter < Controller_P.PulseGenerator_Duty &&
     Controller_DWork.clockTickCounter >= 0) ?
    Controller_P.PulseGenerator_Amp :
    0.0;
  if (Controller_DWork.clockTickCounter >= Controller_P.PulseGenerator_Period-1){
    Controller_DWork.clockTickCounter = 0;
  } else {
    (Controller_DWork.clockTickCounter)++;
  }

  /* DataTypeConversion: '<S1>/Data Type Conversion' */
  Controller_B.DataTypeConversion_p = (boolean_T)(rtb_Add != 0.0 ? 1U : 0U);

  /* trigger Stateflow Block: '<S3>/Stati_HMI' */
  Controller_Stati_HMI();

  ...
  /* Outport: '<Root>/LED_ONOFF' */
  Controller_Y.LED_ONOFF = rtb_Switch22;
```

Outer framework: Stateflow

Generated code: Controller_step0

```
void Controller_step0(void)            /* Sample time: [1.0s, 0.0s] */
{
  ...
  /* DataTypeConversion: '<S2>/Data Type Conversion' */
  Controller_B.DataTypeConversion = (boolean_T)(rtb_Add != 0.0 ? 1U : 0U);

  /* DiscretePulseGenerator: '<S1>/Pulse Generator' */
  rtb_Add =
    (Controller_DWork.clockTickCounter < Controller_P.PulseGenerator_Duty &&
     Controller_DWork.clockTickCounter >= 0) ? Controller_P.PulseGenerator_Amp : 0.0;
  if (Controller_DWork.clockTickCounter >= Controller_P.PulseGenerator_Period-1){
    Controller_DWork.clockTickCounter = 0;
  } else {
    (Controller_DWork.clockTickCounter)++;
  }

  /* DataTypeConversion: '<S1>/Data Type Conversion' */
  Controller_B.DataTypeConversion_p = (boolean_T)(rtb_Add != 0.0 ? 1U : 0U);

  /* trigger Stateflow Block: '<S3>/Stati_HMI' */
  Controller_Stati_HMI();
  ...
  /* Outport: '<Root>/LED_ONOFF' */
  Controller_Y.LED_ONOFF = rtb_Switch22;

  /* Outport: '<Root>/Resistenza' */
  Controller_Y.Resistenza = rtb_LogicalOperator_e;
}
```

Outer framework: Stateflow

Generated code: rt_OneStep

```
static boolean_T OverrunFlags[2] = { 0, 0 };

void rt_OneStep(void)
{
 boolean_T eventFlags[2];             /* Model has 2 rates */

  /* Check base rate for overrun */
  if (OverrunFlags[0]++) {
    rtmSetErrorStatus(Controller_M, "Overrun");
    return;
  }
  Controller_SetEventsForThisBaseStep(eventFlags);
  /* Set model inputs associated with base rate here */
  Controller_step0();
  /* Get model outputs associated with base rate here */
  OverrunFlags[0]--;
  /* Check subrates for overrun */
  if (eventFlags[1]) {
    if (OverrunFlags[1]++) {
      rtmSetErrorStatus(Controller_M, "Overrun");
      return;
    }
    /* Set model inputs associated with subrates here */
    Controller_step1();
    /* Get model outputs associated with subrates here */
    OverrunFlags[1]--;
  }
}
```

Outer framework: Stateflow

Generated code: rt_OneStep

```
static boolean_T OverrunFlags[2] = { 0, 0 };

void rt_OneStep(v
{
 boolean_T eventF

  /* Check base r
  if (OverrunFlag
    rtmSetErrorSt
    return;
  }
  Controller_SetEventsForThisBaseStep(eventFlags);
  /* Set model inputs associated with base rate here */
  Controller_step0();
  /* Get model outputs associated with base rate here */
  OverrunFlags[0]--;
  /* Check subrates for overrun */
  if (eventFlags[1]) {
    if (OverrunFlags[1]++) {
      rtmSetErrorStatus(Controller_M, "Overrun");
      return;
    }
    /* Set model inputs associated with subrates here */
    Controller_step1();
    /* Get model outputs associated with subrates here */
    OverrunFlags[1]--;
  }
}
```

```
void Controller_SetEventsForThisBaseStep(boolean_T *eventFlags)
{
  /* Task runs when counter=0, computed by rtmStepTask macro */
  eventFlags[1] = rtmStepTask(Controller_M, 1);
}

# define rtmStepTask(rtm, idx) ((rtm)->Timing.TaskCounters.TID[(idx)] == 0)
```

Outer framework: Asynchronous task model

(back to Samek's model)

- Step 1: Event definition/basic management
- Events are managed in pools. They are "created" (allocated in the pool), managed (see next) and "destroyed" (entry is returned to the pool)
- We don't see the event pool management, but this is the interface for event creation and destruction

```
QEvent *QF::create(unsigned evtSize, QSignal sig)
void   QF::annihilate(QEvent *e)
```

Outer framework: Asynchronous task model

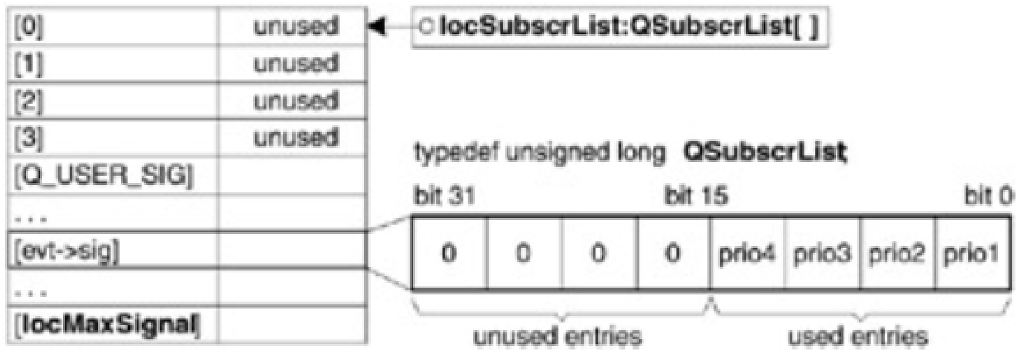
(Samek’s model)

- Step 2: Event forwarding
- A publish-subscribe model
- FSMs register for the events they are interested in to the framework (subscribe)
- Each time an event is produced and sent to the framework, the framework “publishes” it in the input queues of all the subscribers.
  - It actually publishes it in the input queue of the highest priority FSM (more later).
  - Each FSM has the obligation to process and then forward it to the next priority subscriber (if any)

Outer framework: Asynchronous task model

Details of publish-subscribe

A lookup table links signals to subscriber lists



A subscriber list is a list of active objects that have subscribed to a given signal. The list (typedef'd to QSubscrList) is a densely packed bit field storing unique priorities of active objects.

Consequently, the QF requires that clients assign a unique priority to each active object (through QActive::start()), even when the QF is based on an operating system that does not support thread priorities in the preemptive, priority–based sense

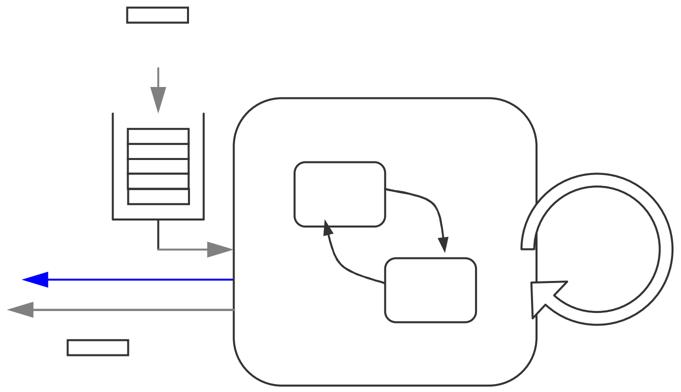
Outer framework: Asynchronous task model

```
void QF::subscribe(QActive *a, QSignal sig) {
  register unsigned char p = a->myPrio; // priority of active object
  REQUIRE(Q_USER_SIG <= sig && sig < locMaxSignal && // boundary chk
  p < QF_MAX_ACTIVE && pkgActive[p] == a); //consistency chk
  QF_PROTECT(); // enter critical section
  register QSubscrList sl = locSubscrList[sig];
  ASSERT((sl & 0xF0000000) == 0); //must have at least one free slot
  for (register int n = 0; n < 32; n += 4) { // find priority slot
    if (p > ((sl >> n) & 0xF)) { // found priority slot?
       sl = (sl & ~(~0 << n)) | // part of sl with priorities>p
       (p << n) | // insert p at bit n
       ((sl << 4) & (~0 << (n + 4))); // shifted rest of sl
       locSubscrList[sig] = sl; // update the subscriber-list
       break; // subscriber registered (attached to the list)
    }
  }
  QF_UNPROTECT(); // leave critical section
}
```

Outer framework: Asynchronous task model

```
void QF::unsubscribe(QActive *a, QSignal sig) {
  register unsigned char p = a->myPrio; // prio. of active object
  REQUIRE(Q_USER_SIG <= sig && sig < locMaxSignal && // bndary chk
  pkgActive[p] == a); // consistency check
  QF_PROTECT(); // enter critical section
  register QSubscrList sl = locSubscrList[sig];
  for (register int n = 0; n < 32; n += 4) { // find priority slot
    if (p == ((sl >> n) & 0xF)) { // found priority slot?
      sl = (sl & ~(~0 << n)) | ((sl >> 4) & (~0 << n));
      locSubscrList[sig] = sl; // update the subscriber-list
      break; // subscription canceled (removed from the list)
    }
  }
  QF_UNPROTECT(); // leave critical section
}
```

Outer framework: Asynchronous task model



Event queuing

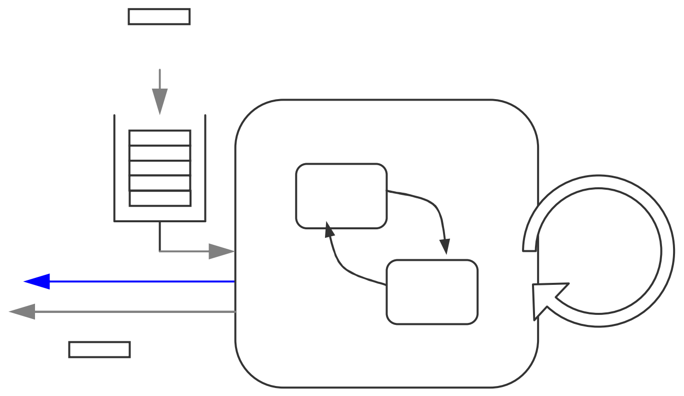
Event queues allow to reconcile the asynchronous production of events with the RTC semantics of their consumption.

An event queue makes the corresponding active object appear always to be receptive to events, even though the internal state machine can accept events only between RTC steps.

Additionally, the event queue provides buffer space that protects the internal statechart from bursts in event production that can, at times, exceed the available processing capacity.

Outer framework: Asynchronous task model

first signs of trouble …



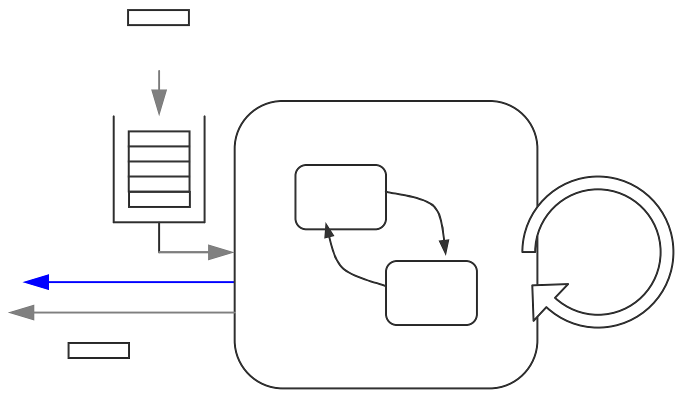
One end of the queue – the end where producers insert events – is obviously shared among many threads and must provide an adequate mutual exclusion mechanism to protect the internal consistency of the queue. The other end – the end from which the local thread extracts events – must provide a mechanism for blocking this thread when the queue is empty.

In addition, an event queue must manage a buffer of events, *typically organized in a FIFO structure*.

Outer framework: Asynchronous task model

Event processing: event processing

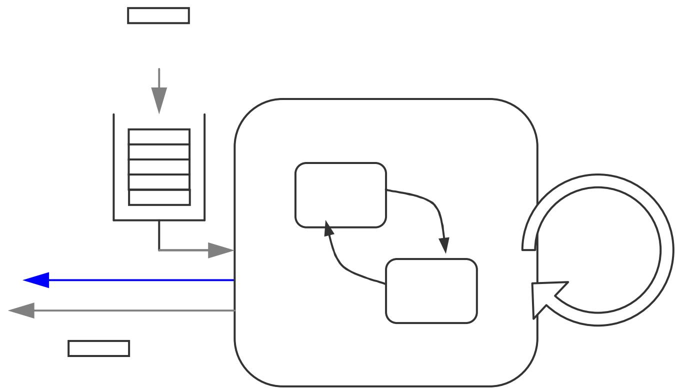
Every active object in the QF executes in its own thread of execution



```
void QActive::start(unsigned prio, QEvent*qSto[], unsigned qLen,
int stkSto[], unsigned stkLen) {
  myPrio = prio; // store the priority in the attribute
  QF::add(this); // register "this" active object by QF

// create event queue "myEqueue" of length "qLen"
// create execution thread "myThread" with priority "prio"
// and stack size "stkLen"

// postcondition: assert proper creation of event-queue and thread
}
```

Outer framework: Asynchronous task model

Event processing: event processing



Once started, all active objects execute the following thread routine (i.e., all active object threads share the following code).

```
/* return type, calling convention */ run(void *a, /*... */) {
   ((QActive *)a)->run();
}

void QActive::run() {
   QHsm::init(); // execute initial transition
   for (;;) { // for-ever
      QEvent *e = myEqueue->get(); // get event; block if queue empty
      dispatch(e); // dispatch event to the statechart
      QF::propagate(e); // propagate event to next subscriber
   }
}
```

Outer framework: Asynchronous task model

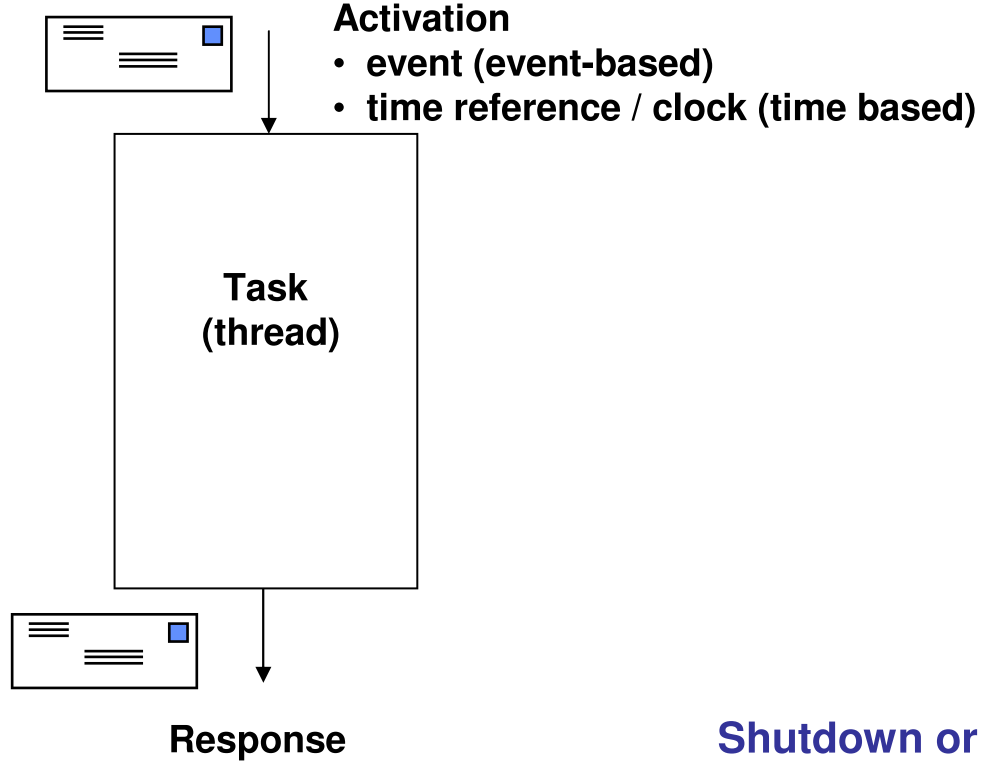
Event generation (publishing)

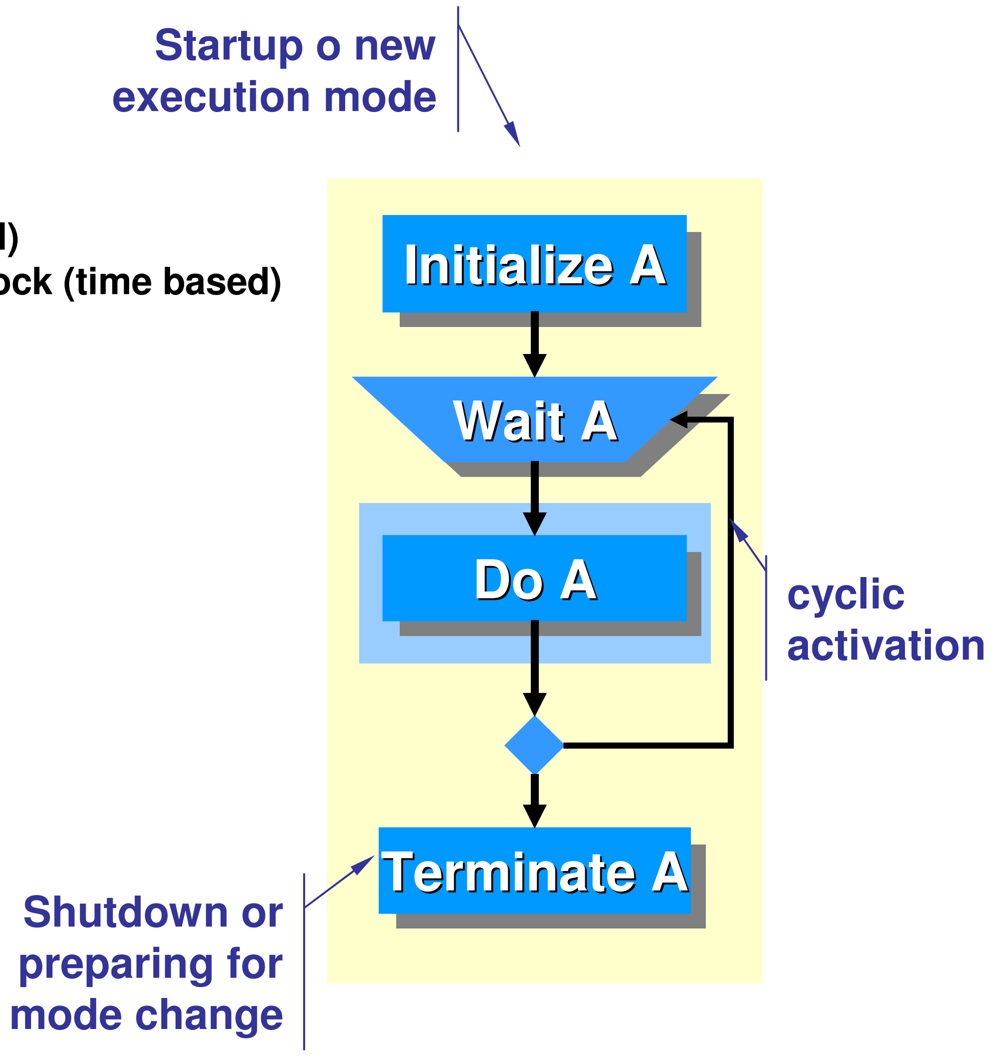
```
void QF::publish(QEvent *e) {
  REQUIRE(e->sig < locMaxSignal && // signal exceeding boundary
  e->useNum == 0); // event cannot be "in use"
  register QSubscrList sl = locSubscrList[e->sig]; //table look-up
  if (sl) { // any subscribers?
    register unsigned char p = (unsigned char)(sl & 0xF);
    e->useNum = 1; // the first use
    ASSERT(pkgActive[p]); // active object must have subscribed
    ALLEGE(pkgActive[p]->enqueue(e)); // queue cannot overflow!
  }
  else { // no subscribers
    QF::annihilate(e); // do not leak the event
  }
}
```

Task-centric vs. event-centric

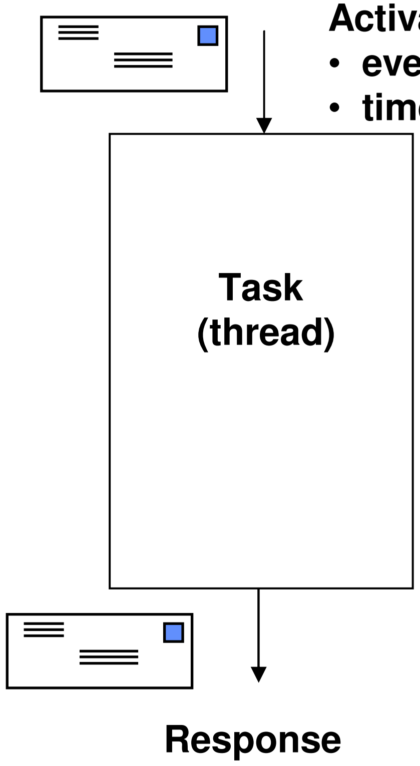
- Task are the main design entities
  - Scheduling operates on tasks
  - Application = set of tasks
  - Tasks activated by signals
  - model for the OMG SPT and used by all commercial tools
- Events are the main design entities
  - Scheduler operates on events
  - Application = Set of events and corresponding reactions
  - Events are handled by one or more tasks

Task centric model




Startup o new execution mode
Initialize A
Wait A
Do A
Terminate A
cyclic activation
Shutdown or preparing for mode change

Task centric model



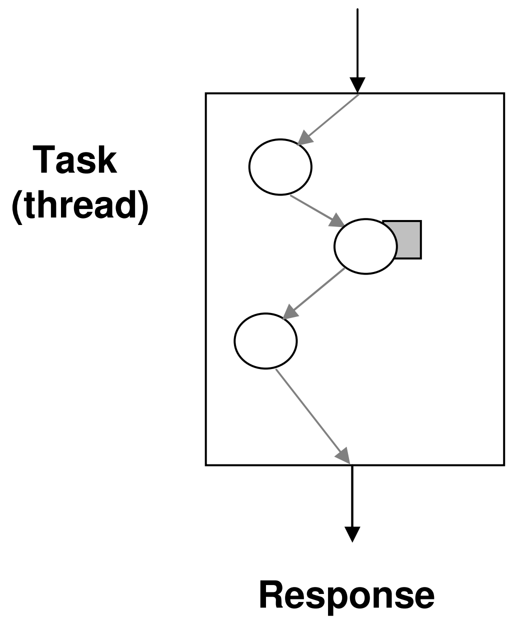
Activation
- **event (event-based)**
- **time reference / clock (time based)**

Main design parameters:

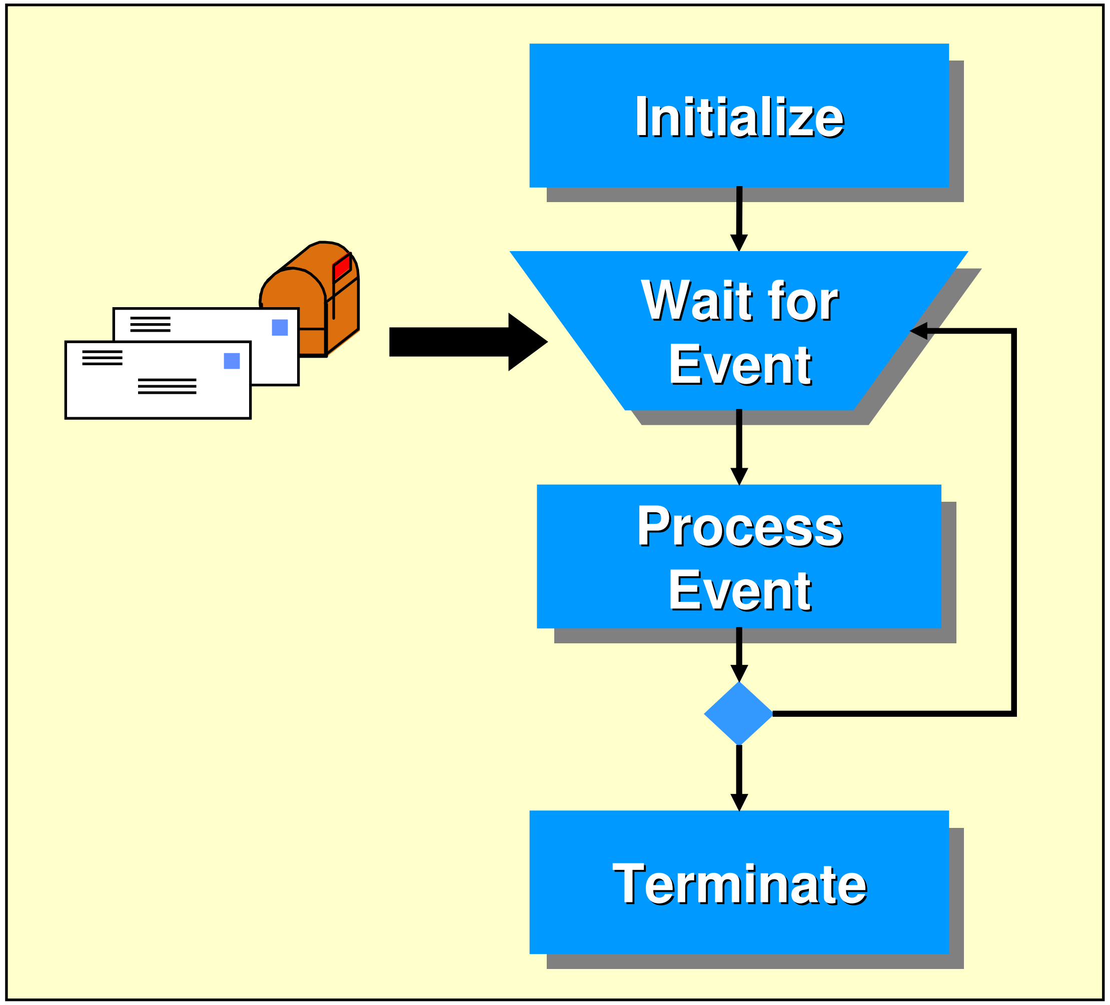
Worst case comp. Time (C)
Release time (r)
Period / Minimum interarrival time (T)
Deadline (D)

Task centric model

- A can contains invocation of passive (protected) objects only (HRT-HOOD/ADA Ravenscar profile model)



Event Centric Design



- **Event scheduling**
    - *Non-Preemptive*
    - *Priority based*
- **Event priorities**
    - *Under application control*

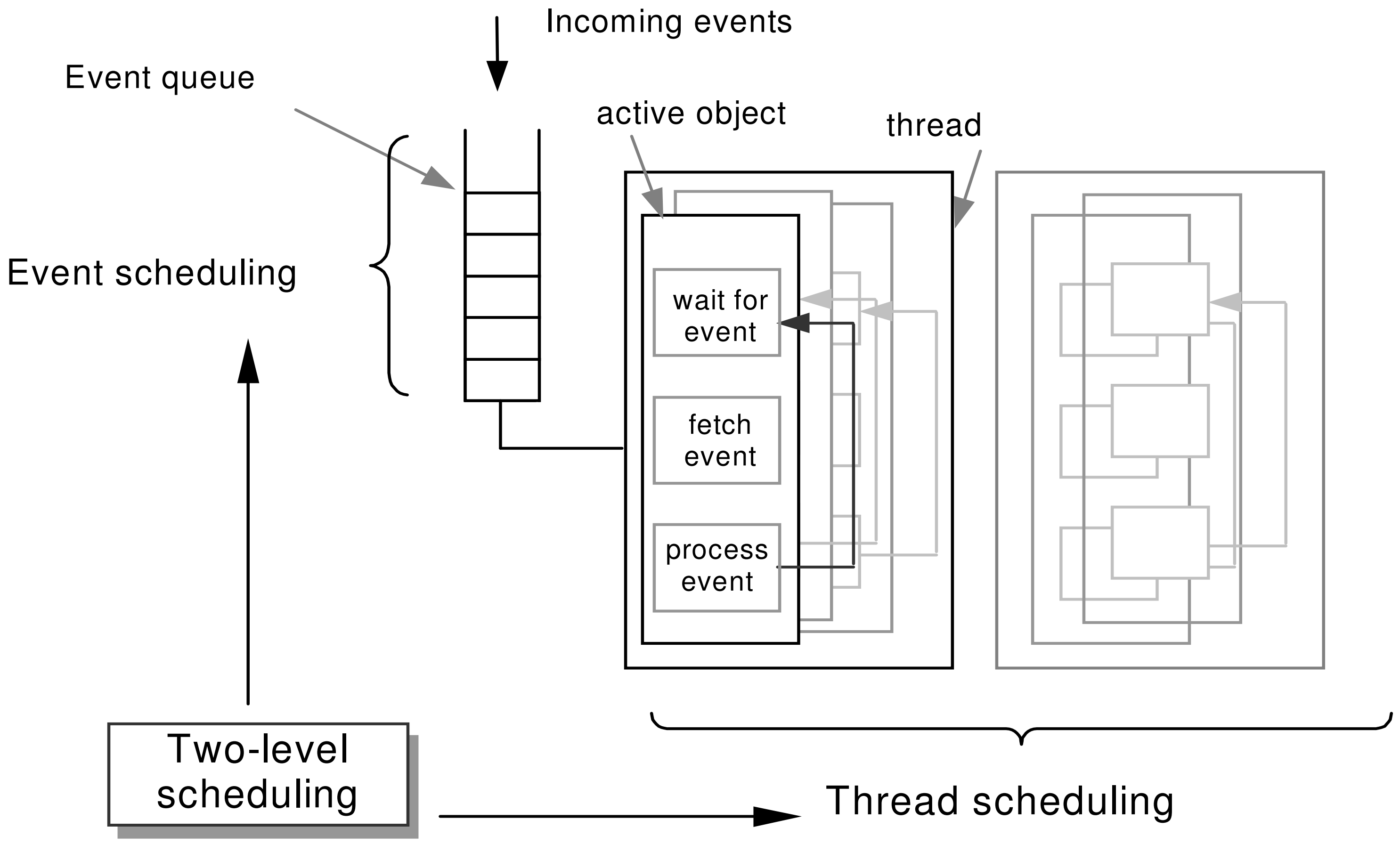
Events

- Schedulable entities
- Three types
  - External events
    - Generated outside the model
    - Introduced by the run-time support (OS/interrupts)
  - Time events
    - Caused by progression of time
    - Timer (periodic and one-shot)
  - Internal events
    - Generated inside the model
    - Asinchronous

Events and objects

- Event Consumer objects
  - Consuming events
  - Each event has a destination object consuming it
  - Define the application behavior
- Event Dispatching Tasks
  - Forwarding events to consumers
    - Control the events queue
    - Implementing event scheduling (Non-Preemptive)

Two-level Scheduling



Event-based scheduling

- Main schedulable objects
  - Events/Messagges in the task mailbox
- Scheduling algorithm
  - Non-preemptive
    - Run-to-completion
  - Priority based
    - Priorities associated to events
    - Tasks inherit events priorities

Single task implementation

- Analysis of the response time for single task implementation
  - Similar to the non-preemptive model for tasks

Multitask implementation

- Motivation
  - Improve response times for high priority events
- Events
  - Priority-based, Non-Preemptive scheduling
- Tasks
  - Preeemptive Scheduling, based on priorities
  - Level 1: RTOS chooses task according to its priority
  - Level 2: Event handler chooses event according ot its priority

Problems

- Tasks
  - How many?
  - What is the mapping relationship between consumer objects and tasks?
- Event scheduling
  - How should priorities be assigned to events?
  - How should events be scheduled?
- Task scheduling
  - what are task priorities?

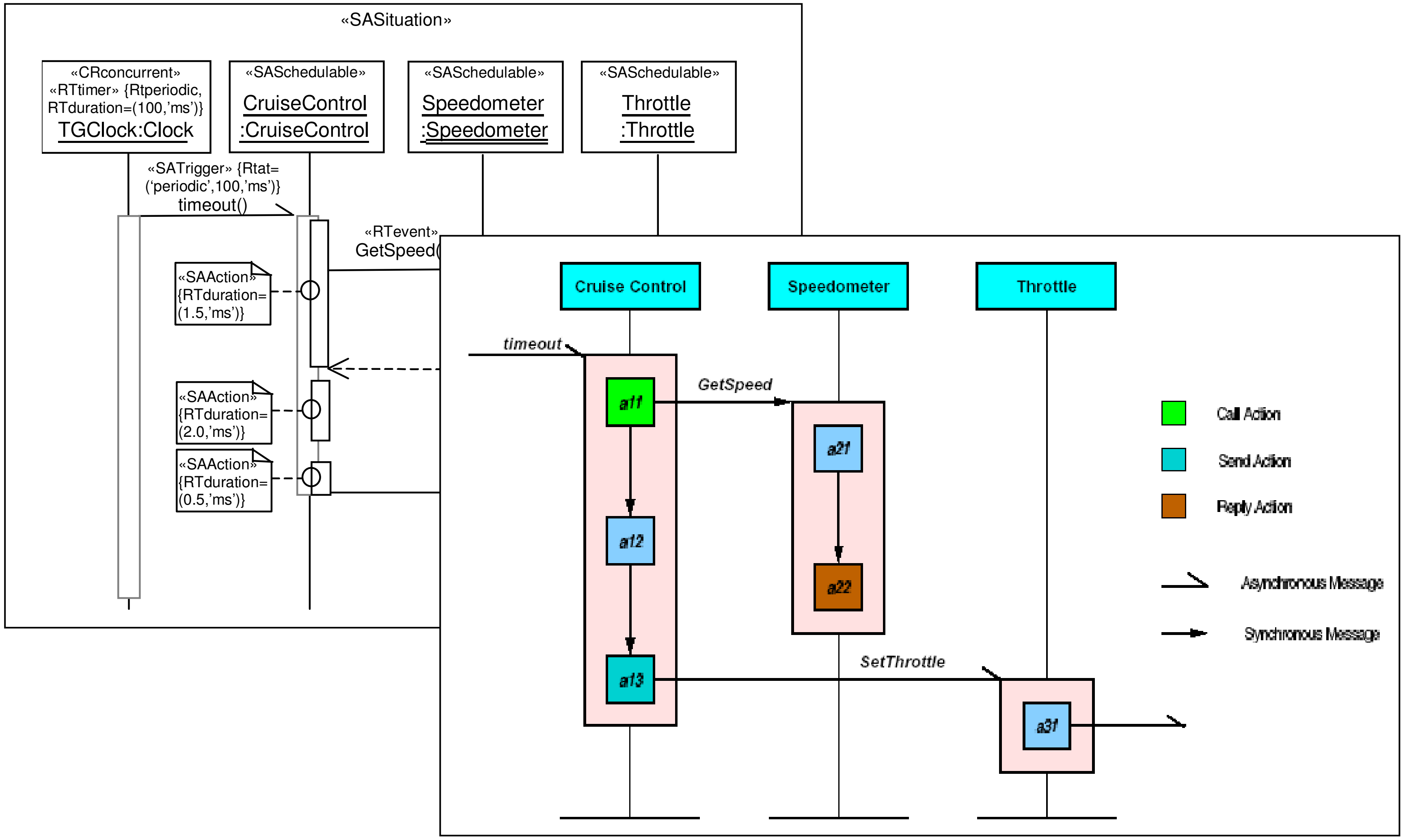
Static task priorities

- Fixed task priorities
  - The priority with which the event is processed depends on the priority of the task
- Problems
  - Counterintuitive
  - Difficult to assign priorities to tasks
  - the schedulability analysis problem is very difficult because of large priority inversions (pessimism)
- But …
  - This is the actual multitask implementation for most (all) commercial tools!

Dynamic task priorities

- Intuition: events are the elements that need processing
  - Tasks are processing actors
- Dynamic priorities for tasks
  - Task priority is inherited from the priority of the event being processed
- Task are processing “resources” shared by events
- Standard resource sharing protocols (PCP) can be applied
- Schedulability analysis is possible …

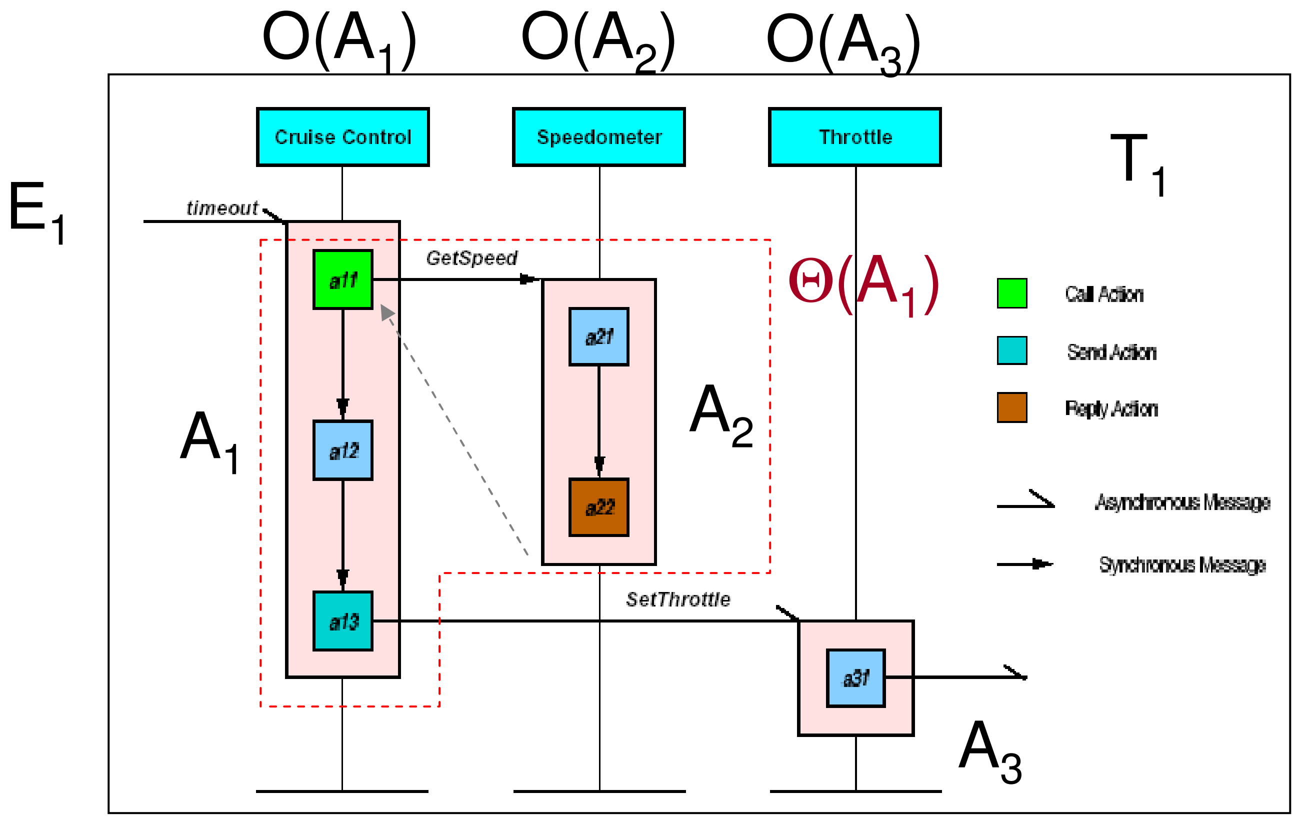
Schedulability analysis from behavioral diagrams


«SASituation»
«CRconcurrent» «RTtimer» {Rtperiodic, RTduration=(100,'ms')} TGClock:Clock
«SASchedulable» CruiseControl :CruiseControl
«SASchedulable» Speedometer :Speedometer
«SASchedulable» Throttle :Throttle
«SATrigger» {Rtat= ('periodic',100,'ms')} timeout()
«RTevent» GetSpeed(
«SAAction» {RTduration= (1.5,'ms')}
«SAAction» {RTduration= (2.0,'ms')}
«SAAction» {RTduration= (0.5,'ms')}
Cruise Control
Speedometer
Throttle
timeout
GetSpeed
SetThrottle
a11
a12
a13
a21
a22
a31
Call Action
Send Action
Reply Action
Asynchronous Message
Synchronous Message

Schedulability analysis: definitions

- let $\mathcal{E} = \{E_1, E_2, ..., E_n, E_{n+1}, ..., E_m\}$ be the set of all events in the system,
- E_1, E_2, ... E_n denote external (asynchronous) events (the remaining internal ones)
- Each external event E_i may originate a transaction T_i
- Associated with each event E_i is an action A_i.
- Each action is decomposed in subactions $A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,n}\}$
- All events and actions are part of a transaction.
- Action A^T and event E^T belong to transaction T.
- Each action A_i is characterized as either asynchronously triggered or synchronously triggered and executes within the context of an active object $O(A_i)$.
- A_i is characterized by a priority π_i, which is the same as the priority of its triggering event E_i.

Schedulability analysis from behavioral diagrams



Schedulability analysis: definitions

- Each action A_i (subaction $a_{i,j}$) is characterized by a computation time $C(A_i)$ (abbreviated as C_i).
- $\Theta(A_k)$ denotes the synchronous set of A_k, that is the set of actions that can be built starting from action A_k and adding all actions that are called synchronously from it.
- $C(\Theta(A_k))$ is the sum of the execution times of all the actions in $\Theta(A_k)$.
- Each external event stream E_i is characterized by a function $\Psi_i(t)$ that gives the maximum number of event arrivals in any interval $[x, x+t)$
- $\Psi^+_i(t)$ indicates the maximum number of event arrivals in any right-closed interval $[x, x+t]$.
- Ex: if the min. interarr. time is T, then $\Psi_i(t) = \lceil t/T \rceil$ and $\Psi^+_i(t) = \lfloor t/T \rfloor + 1$

Schedulability analysis: definitions

- Schedulability analysis of the general model is carried out by computing response times of actions.
- The response time of an action A^T_i is derived relative to the arrival of the external event that triggers the transaction T.
- The analysis is based on the standard concepts of critical instant and busy-period for task instances with generic deadlines (adapted to the transaction model.)

Schedulability analysis: definitions

- The analysis of the worst case response time of A^T_i requires computing the response times of all the instances of A^T_i inside the busy period.
- If $r^T_{i,q}$ is the release time of the instance *q* of the external event E^T_i, starting from the critical instant *t*=0, we need to compute $S^T_{i,q}$, the worst case start time for an instance *q* of A^T_i and its worst-case finishing time $F^T_{i,q}$.
- The worst-case response time of action A^T_i is given by:

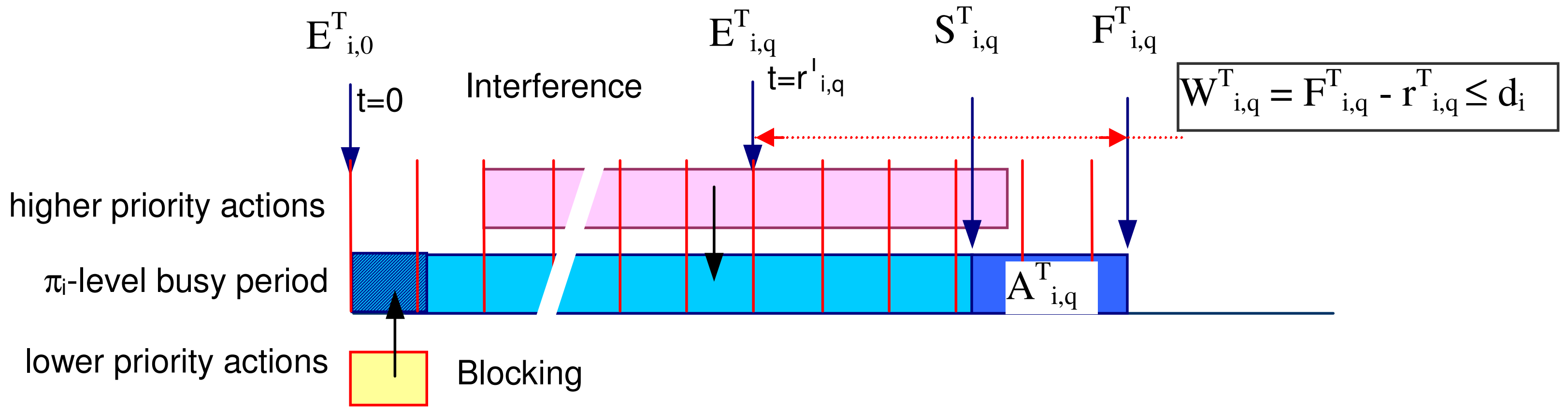
$$W_i^T = \max_{q \in [1,\ldots,m]} \{F_{i,q}^T - r_{i,q}^T\}$$

- If W^T_i is lower than the deadline of A^T_i, then the action is schedulable (Figure 12.11).

Schedulability analysis: assumptions

- Priorities are assigned to events
- Each action A^{T}_{i} inherits its priority from the triggering event.
- A synchronously triggered action inherits its priority from the caller.
- Every transaction is made up of actions with non-increasing priorities (if A_j is triggered by A_i then $\pi_i \geq \pi_j$)

Schedulability analysis: definitions



In computing the busy period we need to evaluate all the factors that contribute to it:

- the interference factor
- the computation time of the action A^T_i itself and
- the blocking factor (from events with priority < A^T_i).

The first term can be further specialized in an interference factor from actions belonging to the same transaction T and interference from actions belonging to transactions other than T.

Schedulability analysis: definitions

- Single thread
- *Blocking*
- Since intra-task preemption is not allowed, in single threaded implementations, any synchronous set that starts executing must be completed with no interruption. Hence, the worst case blocking time of an action is bound by the longest synchronous set of any lower priority action that started prior to *t*=0.

$$B(A_i) = \max_k \{ C(\Theta(A_k)) :: \pi_i > \pi_k \}$$

Schedulability analysis: definitions

- Single thread
- *Critical instant*
- All higher priority events arrive at the same time of E_i (t=0)
- The event generating the worst blocking term has just arrived (immediately prior to t=0)
- (All transactions –external events- arrive at their maximum rate)

Schedulability analysis: definitions

- Once A^T_i starts executing, no other action can interrupt it other than any synchronous calls that A^T_i makes. Consequently, the worst-case finish time for instance q of A^T_i is

$$F_i^T = S_{i,q}^T + C(\Theta(A_i^T))$$

- and schedulability can be guaranteed provided

$$W_i^T = \max_{q \in [1,\ldots,m]} \{F_{i,q}^T - r_{i,q}^T\} \leq d_i$$

Schedulability analysis: definitions

- We need to find an expression for $S^T_{i,q}$
- we need to consider the blocking term plus interference by all actions that can execute (arrive) in the interval
  $[0, S^T_{i,q}]$ before $A_{i,q}$

Schedulability analysis: definitions

- Single thread
- *Interference (1) From other transactions (T'≠T)*
- For each transaction T'≠T the highest arrival rate of external events is assumed and the sum of the computation times of all actions $A^{T'}_{l} \in T'$ with a priority higher than π_i is considered.

$$I^{T' \neq T}(A_i^T) = \sum_{T' \neq T} [\Psi^{+}_{T'}(S \quad) \cdot \sum_{l} (C_l^{T'} :: \pi_l \geq \pi_i)]$$

Schedulability analysis: definitions

- Single thread
- *Interference (2) From transaction* T (instances from 0 to q-1 and instances after q*)*
- for transaction T, we need to consider the term from all the higher priority actions in the previous *q*-1 instances and the interference from the actions belonging to the instances from *q*+1 to the last one that can be possibly activated before $S_{i,q}$.
- for instances from 0 to q-1 the interference term is the same as before.

$$I^T(A_i^T) = (q-1) \cdot \sum_l (C_l^T :: \pi_l \geq \pi_i)$$

Schedulability analysis: definitions

- Single thread
- *Interference (2) From transaction* T (instances from 0 to q-1 and instances after q*)*
- For all instances ≥*q*, only the higher priority actions that are not successors of $A_{i,q}$ contribute to the interference.
  - if $A_{i,q}$ has not started execution, neither can its successors !
- let $A_i^T \xrightarrow{path} A_l^T$ denote the condition for which there exists a path of (call or signal) events and actions that makes A^T_l causally dependent on A^T_i

$$(\Psi_T^+(S)-(q-1))\cdot\sum_l (C_l^T :: \neg(A_i^T \xrightarrow{path} A_l^T)\wedge \pi_l \geq \pi_i)$$

Schedulability analysis: definitions

- Single thread
- *Interference (2) Combined*

$$I^T(A_i^T) = (q-1)\cdot\sum_l (C_l^T :: \pi_l \geq \pi_i)$$

$$+(\Psi_T^+(S)-(q-1))\cdot\sum_l (C_l^T :: \neg(A_i^T \xrightarrow{path} A_l^T) \wedge \pi_l \geq \pi_i)$$

Schedulability analysis: definitions

- Single thread
- *Composing the terms …*

Once the blocking and the interference factors are known, the worst case start time can be computed with the iterative formula

$$S_{i,q} = \min S :: S = \quad B(A_i) + I^{T' \neq T}(A_i^T) + I^T(A_i^T)$$