Yet Another Hierarchical State Machine

by Stefan Heinzmann

Most of you are probably familiar with state machines. Finite state machines (FSMs) are widely used in both digital electronics and programming. Applications include communication protocols, industrial control or GUI programming. UML includes a more flexible variant in the form of statechart diagrams. Here, states can contain other states, making the corresponding state machine hierarchical.

Hierarchical state machines (HSMs) can be converted to ordinary (flat) state machines, so most implementors concentrate on the implementation of FSMs, and many implementations exist. The conversion however tends to lose the original structure of the HSM, making it difficult to make the connection between a statechart and its implementation. Clearly, it would be nicer if the HSM could be implemented directly in source code.

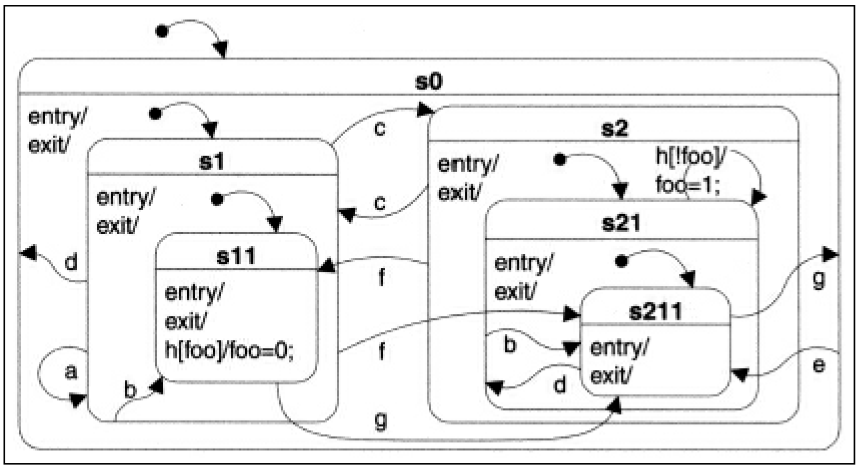
Direct implementations of HSMs seem to be comparatively rarely published. Miro Samek provides a prominent example in his book [1]. See the sidebar (next page) for some more information about his approach, which provided the motivation for the approach I will present here. UML tools are available that generate source code directly from statechart diagrams (example: Rhapsody [5]). More information about HSMs and Statecharts can be found in [1],[2],[3] and – of course – through Google.

The implementation I'm presenting here allows you to code your HSM in C++ directly, without the need for special code generation tools or wizards, although such a wizard could ease development further by converting the statechart automatically. Template techniques are used to enable the compiler to optimize the code through inlining, and I believe the resulting code ranks among the fastest you can get, provided you have a good up-to-date optimizing compiler (which you'll need anyway because of the templates).

Only a subset of UML statecharts are supported at present. If you need advanced features like concurrent states or history pseudo-states, you need to expand the solution yourself (tell me if you do so, it would be nice to have a complete implementation).

The TestHSM Example

It is best to describe how a hierarchical state machine works when you have an example. I lifted the statechart shown below from Miro's book. It specifies the state machine for the test example he implemented in his book ([1] page 95). We'll implement the same here. The example is artificial and only serves the purpose of providing a number of test cases for checking the correct implementation of the state machine. Its advantage is that it is quite small and therefore well suited for demonstration. It is driven by keypresses on the keyboard.



An HSM is a state machine where states can be grouped into a composite state. Actions defined for such a composite state then apply automatically to all states contained therein. This allows a considerable simplification of the state diagram. In our diagram, you can easily see this. Each state is drawn as a box with rounded edges and bears a unique name. Composite states are those that contain other states. States that contain no other states are leaf states. The only leaf states in our example are `s11` and `s211`. Arrows represent the transitions between the states, they are labeled with an event that causes this transition to be taken. In the example, the transitions are annotated with the key that needs to be pressed to invoke the transition.

If a transition originates from a composite state, it is taken whenever a substate does not handle the corresponding event itself. A state can thus pass on the handling of a specific event to its enclosing state. For example, pressing the `e` key causes a transition to state `s211` regardless of the currently active state. Rather than cluttering the diagram with numerous transition arrows, it suffices to introduce an all-encompassing top-level state `s0` and handle the `e` key there.

This does not just simplify the diagram, it also points out where code reuse is possible. A statechart implementation should use this opportunity. We therefore need a possibility to pass unhandled events to the encompassing state. If no state handles the event, it is ultimately discarded. Each state can have special exit and entry actions associated with it. Those actions are executed whenever a transition leads out of or into a state, respectively. This is called an external transition. By contrast, an internal transition does not execute any exit and entry actions. Our state machine implementation needs to do the necessary bookkeeping to call the actions in the right order. In particular, the actions associated with a transistion are executed after all exit actions and before any entry action associated with the relevant states are executed.

During operation the state machine is always in a leaf state. So transitions ultimately lead from one leaf state to another. If a transition is drawn to target a composite state, the composite state needs to specify one of its substates as the initial state. In our example, the initial state of composite state `s0` is specified to be state `s1`. As this state is also composite, it needs an initial state, too, in this case `s11`. The effect is that any transition targeting state `s0` is in reality targeting state `s11`. A composite state that does not specify an initial state can not be the target of a transition. Our example diagram only contains two action specifications. In our code we will additionally print out trace messages for illustration, but the diagram does not show this. The action specifications shown are:

- The transition from `s21` to itself (a self-transition). This is an example of a transition that has a guard (in brackets `[]`) and an associated action (after the slash `/`). The guard is a condition that must evaluate to true to enable the transition. If it evaluates to false, the transition is not taken and none of the actions are executed. A self-transition exits and reenters the state, hence the associated exit and entry actions are executed.

Miro Samek's HSM Implementation

If coding state machines is one of your favourite pastimes you will surely have come across Miro Samek's book "Practical Statecharts in C/C++" [1]. Chris Hills reviewed it for ACCU quite favourably a few months ago. I can second this, yet I'm still in the game for new state machine designs. Why is that?

Well, you may have noticed that Miro's way of implementing state machines isn't typesafe and requires quite a few typecasts, neatly tucked away in a set of convenience macros. His implementation of hierarchical state machines isn't the fastest either, because of his way of handling entry and exit actions. There is a strong reason for this: His implementation works with just about anything that calls itself a C++ compiler, even ancient versions like VC++1.5. That means he completely avoids the "newer" C++ features like templates. If you are programming for embedded systems this is a good thing because "full" C++ compilers are only slowly gaining ground here.

State machines are more widely applicable than that, however, and even in embedded systems you may have the luck to use a compiler that attempts to support the full language, for example g++. Hence I believe there is a "market" for state machine designs that use the full language in order to address the deficiencies of Samek's design. This is what motivated me.

Miro's implementation represents the current state with a member function pointer that points to the state handler function for this state. This is an efficient representation, but it means that the handler function has to do double duty in that it also handles the entry and exit actions. For this, special reserved event codes are used, and a transition leads to potentially quite a large number of function calls through member function pointers. This is especially annoying when you realize that a large fraction of those will do little or no real work.

It also restricts your freedom in the way in which you can represent events. You are forced to use the predefined event class defined by Miro's framework, and some events/signals are predefined. The code presented here assumes nothing about the event representation. This aspect is left entirely to the concrete case you're concerned with.

Another difference is that Miro needs a number of typecasts, which are mostly hidden in convenience macros. This is because of the C++ restrictions in automatic conversion of member function pointer types. Miro's code works efficiently, but lacks type safety.

Miro works out which entry and exit actions are to be called in a function `tran()`, which does the work at runtime. This is very flexible as it allows dynamic transitions that can change at runtime. This comes at a cost, however, as there are potentially many handler functions that must be called without much work being done in them. As most transitions are static, he implemented an optimization that does the calculation of the transition chain only once and stores the result in a static variable. The code presented here only supports static transitions and calculates the chain at compile time, allowing inlining the entire chain. The result is typically both faster and uses less storage than Miro's code. Also, Miro found it hard to obey the UML rule that the actions associated with a transition are executed after any exit actions and before any entry actions are executed. His implementation executes the transition actions as the first thing, followed by all exit and entry actions. This makes exit actions a lot less useful. My code avoids these drawbacks.

The flip side is that Miro's code is more portable because the demands on the compiler are low. This is most welcome in embedded systems, where compilers often don't even attempt to implement the whole C++ standard. His solution is thus more widely applicable than mine.

Both implementations lack support for some more advanced features of UML statecharts, such as concurrent states or history pseudo-states. It is as yet an open question how difficult they are to add to the solution that I presented here. If you find you need such features and have an idea how to include them in the code presented here, I'd be interested to hear from you.

- The internal transition inside `s11` is not drawn with an arrow. It merely specifies an action that is to be taken when a certain event occurs, but no transition to another state occurs, and no exit or entry actions are performed. In our case the internal transition has a guard, so the associated action (`foo = 0`) is only executed when the `h` key is pressed while `foo` evaluates to `true`.

Note the difference between a self-transition and an internal transition. The latter never changes the active state and doesn't execute any exit or entry actions. Note also that internal transitions can be specified in a composite state, too, although this isn't shown in our example.

Representing the State

Flat state machines often represent the current state using a variable of `enum` type. Other implementations use a function pointer that points to the handler function for the current state. This handler function is called whenever an event occurs. Still other implementations represent states with objects, so the current state is represented by a pointer to the current state's object instance. This latter implementation is suggested by the State design pattern [6]. This is also the approach taken here, with the additional feature that all states have unique types to allow compile-time algorithms based on the state hierarchy. Only instances of leaf states need to be present, as they are the only states that can be active at any time. Composite states only exist as types, they are abstract and can therefore not be instantiated. The relationship between a composite state and its substates is modelled through inheritance. A composite state is the base class of its substates. All states derive from a top-level state class.

Often, entry or exit actions are empty or consist of trivial statements. I wanted to use the benefits of inlining as much as possible to allow the compiler to optimize away the overhead associated with functions that don't do much. I was prepared to dismiss the possibility of determining the target state at run time. If all transitions go from a source state to a target state, both of which are known at compile time, the compiler can figure out the entry and exit functions that need to be called and inline all of it into a single optimized string of code. My goal was to use templates to implement this compile-time task.

I chose therefore to represent the states as instances of class templates. Leaf states and composite states have separate templates. Each different state in the diagram is thus represented by a different instantiation of a predefined class template. Implementing state handlers and entry/exit actions is done by specializing member functions from this class template. If you don't specialize it, an empty default is automatically taken.

Here's the definition of the `CompState` and `LeafState` class templates:

```
template<typename H>
struct TopState {
  typedef H Host;
  typedef void Base;
  virtual void handler(Host&) const =0;
  virtual unsigned getId() const =0;
};
template<typename H, unsigned id,
         typename B> struct CompState;
template<typename H, unsigned id,
       typename B=CompState<H,0,TopState<H> > >
struct CompState : B {
  typedef B Base;
  typedef CompState<H,id,Base> This;
  template<typename X> void handle(H& h,
      const X& x) const { Base::handle(h,x); }
  static void init(H&); // no implementation
  static void entry(H&) {}
  static void exit(H&) {}
};
template<typename H>
struct CompState<H,0,TopState<H> > :
                                TopState<H> {
  typedef TopState<H> Base;
  typedef CompState<H,0,Base> This;
  template<typename X> void handle(H&,
                             const X&) const {}
  static void init(H&); // no implementation
  static void entry(H&) {}
  static void exit(H&) {}
};
template<typename H, unsigned id,
      typename B=CompState<H,0,TopState<H> > >
struct LeafState : B {
  typedef B Base;
  typedef LeafState<H,id,Base> This;
  template<typename X> void handle(H& h,
      const X& x) const { Base::handle(h,x); }
  virtual void handler(H& h) const
                          { handle(h,*this); }
  virtual unsigned getId() const { return id; }
  static void init(H& h) { h.next(obj); }
                       // don't specialize this
  static void entry(H&) {}
  static void exit(H&) {}
  static const LeafState obj;
};
template<typename H, unsigned id, typename B>
const LeafState<H, id, B> LeafState<H, id,
                                    B>::obj;
```

And here's how you use this to specify the states of our example:

```
typedef CompState<TestHSM,0>     Top;
typedef CompState<TestHSM,1,Top>  S0;
typedef CompState<TestHSM,2,S0>     S1;
typedef LeafState<TestHSM,3,S1>       S11;
typedef CompState<TestHSM,4,S0>     S2;
typedef CompState<TestHSM,5,S2>       S21;
typedef LeafState<TestHSM,6,S21>        S211;
```

I used indentation to indicate state nesting. Each state bears a unique numeric ID code, starting with `0` for the top-level state which is outside of the all-encompassing `s0` state of our example. The ID code ensures that all states are distinct types. Except for the top-level state you have to specify the enclosing state for each state. This is how the hierarchy is defined. It leads to a corresponding class inheritance pattern, i.e. `Top` is a base class for all other states.

The `TestHSM` class is where the current state is held (it corresponds loosely to Miro's `QHsmTst` class). This class hosts the state machine. Actions are typically implemented as member functions of this class. As the states all have different types, we can only represent the current state through a pointer to the top-level state, from which all states are derived. Dispatching an event to the current state handler calls the `handler()` member function of the current state through that pointer. The `handler()` member function thus needs to be virtual. All states that can actually be the current state, that is all leaf states, contain nothing but a vtbl-Pointer. So, ironically, they are objects without state.

The current state of the state machine is represented by a pointer to the corresponding state object.

```
const TopState<TestHSM>* state_;
```

Invoking the handler of the current state in response to an event is called dispatching, and it is done simply like this (assuming we're in a member function of `TestHSM`):

```
state_->handler(*this);
```

Note that only `LeafState` has a static member `obj`; `CompState` does not need it because it can't be instantiated anyway, as it does not implement the pure virtual functions inherited from `TopState`. The `LeafState` and `CompState` templates provide empty implementations for entry and exit actions. If you provide specialized functions yourself, they will be taken instead. This is how you implement your own entry and exit actions. More on this later.

Representing Events

An event is something that triggers actions and state transitions in the state machine. Without events, the state machine is sitting still and doing nothing. State machines are reactive systems. Events are not represented by anything predefined in this state machine implementation. You are essentially free to provide what you see fit for this task. The only thing you need to do is to call the event dispatcher shown above whenever an event happens. In order for the state handlers to determine what happened, you will also need to provide access to data associated with the event. For example, you could store an event ID-code in a member variable of the state machine's host class and have the state handler functions interrogate it to find out about the particular event at hand. Here's how our `TestHSM` class does it:

```
enum Signal { A_SIG,B_SIG,C_SIG,D_SIG,
              E_SIG,F_SIG,G_SIG,H_SIG };
class TestHSM {
public:
  TestHSM();
  ~TestHSM();
  void next(const TopState<TestHSM>& state)
                         { state_ = &state; }
  Signal getSig() const { return sig_; }
  void dispatch(Signal sig)
       { sig_ = sig; state_->handler(*this); }
  void foo(int i) { foo_ = i; }
  int foo() const { return foo_; }
private:
  const TopState<TestHSM>* state_;
  Signal sig_;
  int foo_;
};
```

Here, the event is represented by `enum` values corresponding to the actual key pressed on the keypad. On each keypress, the surrounding system needs to call `dispatch()` to invoke the dispatcher. In our example, we do it like this:

```
int main() {
  TestHSM test;
  for(;;) {
    printf("\nSignal<-");
    char c = getc(stdin);
    getc(stdin); // discard '\n'
    if(c<'a' || 'h'<c) {
      return 0;
    }
    test.dispatch((Signal)(c-'a'));
  }
}
```

You can see how the state machine is driven from the outside by calling `dispatch()` repeatedly. This is the essence of driving a reactive system. You call it each time something interesting happens. This is also easy to integrate with the message pump or event loop of a typical GUI, although I don't show this here (I would have to commit to a specific GUI, making it more difficult for you to try the code if you use a different system).

Your representation of events may be completely different from that in the example, and it is neither necessary to store it in a single member variable nor indeed do you need to store it in the state machine host class at all. You just need to make sure the handler functions can somehow get at it. This is easiest when it is stored in the host class, as a reference to the host object is always passed to the handlers.

Handlers and Actions

Implementing the handler functions is the central element of implementing the statechart. Here are the handler functions for our example. You may want to cross-check with the diagram while browsing through this source code. We implement a function template specialization for each state.

```
template<> template<typename X> inline void
    S0::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case E_SIG: { Tran<X,This,S211> t(h);
                  printf("s0-E;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
    S1::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case A_SIG: { Tran<X,This,S1> t(h);
                  printf("s1-A;"); return; }
    case B_SIG: { Tran<X,This,S11> t(h);
                  printf("s1-B;"); return; }
    case C_SIG: { Tran<X,This,S2> t(h);
                  printf("s1-C;"); return; }
    case D_SIG: { Tran<X,This,S0> t(h);
                  printf("s1-D;"); return; }
    case F_SIG: { Tran<X,This,S211> t(h);
                  printf("s1-F;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
   S11::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case G_SIG: { Tran<X,This,S211> t(h);
                  printf("s11-G;"); return; }
    case H_SIG: if(h.foo()) {
                  printf("s11-H;");
                  h.foo(0); return;
                } break;
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
    S2::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case C_SIG: { Tran<X,This,S1> t(h);
                  printf("s2-C;"); return; }
    case F_SIG: { Tran<X,This,S11> t(h);
                  printf("s2-F;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
    S21::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case B_SIG: { Tran<X,This,S211> t(h);
                  printf("s21-B;"); return; }
    case H_SIG: if(!h.foo()) {
                  Tran<X,This,S21> t(h);
                  printf("s21-H;"); h.foo(1);
                  return;
                } break;
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
  S211::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case D_SIG: { Tran<X,This,S21> t(h);
                  printf("s211-D;"); return; }
    case G_SIG: { Tran<X,This,S0> t(h);
                  printf("s211-G;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
```

This is about as straightforward as is gets. Let's look at the last handler: `S211::handle()` as an example. If you check with the diagram, you can see that the `s211` state handles transitions associated with two events: Pressing `d` causes a transition to state `s21`, while pressing `g` causes a transition to state `s0`. Each of the transitions print a log message. The function `S211::handle()` implements this behaviour, and you should have no trouble making the connection between the diagram and the code. This simple handler function illustrates 3 points:

1. The event (key code) is retrieved from the host object using the `getSig()` function. A `switch` discriminates amongst the different events that are relevant for this state. The default case forwards the unhandled event types to the parent state. The `CompState`/`LeafState` class templates contain helpful typedefs to make this convenient. If no state handles the event, it ends up in the handler for the top state, where it is silently discarded by default. If you want a different behaviour, you may specialize the `handle()` function template for the `Top` state.
2. Actions are implemented as ordinary function calls, for example to member functions of the host class. In our example handler, the action is simply a call to `printf()`, which prints a log message.
3. Transitions are managed by yet another class template: `Tran`. The details of this are explained later, suffice to say that a `Tran` object is created on the stack in much the same way as a scoped lock object, and it is destroyed automatically at the end of the scope. At construction time all relevant exit actions associated with the state transition are called, and at destruction the relevant entry actions are performed. Also, the host object's state pointer is made to point at the new state. In between construction and destruction of this `Tran` object you can call any actions that are associated with this particular transition.

The UML statechart formalism allows a few more variations. It allows conditional transitions, that is transitions that are only executed when a guard condition holds true. This can be acommodated easily by testing the guard condition with an `if`-statement inside the corresponding `switch` case. The handler function `S21::handle()` illustrates this in `case H_SIG`. For an internal transition, you don't construct a `Tran` object. This is what is done in `case H_SIG` of `S11::handle()`.

The implementation of exit and entry actions is similarly straightforward:

```
// entry actions
template<> inline void S0 ::entry(TestHSM&)
                { printf("s0-ENTRY;"); }
template<> inline void S1 ::entry(TestHSM&)
                { printf("s1-ENTRY;"); }
// and so on...

// exit actions
template<> inline void S0 ::exit(TestHSM&)
                { printf("s0-EXIT;"); }
template<> inline void S1 ::exit(TestHSM&)
                { printf("s1-EXIT;"); }
// and so on...
```

Can it get any simpler? Here we just call the action routine that needs to be executed whenever a state is exited/entered. Again, we just print a log message, but anything could be done here. The only thing missing is the init routines, which are necessary for each state that has an initial transition. This initial transition may have an associated action, but usually just points to a substate.

```
// init actions (note the reverse ordering!)
template<> inline void S21 ::init(TestHSM& h)
        { Init<S211> i(h); printf("s21-INIT;"); }
template<> inline void S2 ::init(TestHSM& h)
        { Init<S21> i(h); printf("s2-INIT;"); }
// and so on...
```

As before, the action is the printing of a log message. Another special template `Init` is used to specify the transition to the initial substate. Please crosscheck with the diagram. In most practical cases, action routines will be members of the host class. This is hinted at in our example with the function `foo()`. This is where you put the actual code that implements the actions. The handlers only have the task of selecting the right action and state transition and invoke them in the right order. Try to keep detailed action code out of the handlers.

The Magical Tran Template

The most interesting part is the last: the `Tran` template that figures out which entry and exit actions to call:

```
template<typename C, typename S, typename T>
// Current,Source,Target
struct Tran {
  typedef typename C::Host Host;
  typedef typename C::Base CurrentBase;
  typedef typename S::Base SourceBase;
  typedef typename T::Base TargetBase;
  enum { // work out when to terminate
         // template recursion
    eTB_CB = IsDerivedFrom<TargetBase,
                           CurrentBase>::Res,
    eS_CB = IsDerivedFrom<S,CurrentBase>::Res,
    eS_C = IsDerivedFrom<S,C>::Res,
    eC_S = IsDerivedFrom<C,S>::Res,
    exitStop = eTB_CB && eS_C,
    entryStop = eS_C || eS_CB && !eC_S
  };
  // We use overloading to stop recursion. The
  // more natural template specialization
  // method would require to specialize the
  // inner template without specializing the
  // outer one, which is forbidden.
  static void exitActions(Host&, Bool<true>) {}
  static void exitActions(Host& h, Bool<false>){
    C::exit(h);
    Tran<CurrentBase,S,T>::exitActions(h,
                            Bool<exitStop>());
  }
  static void entryActions(Host&, Bool<true>) {}
  static void entryActions(Host& h,Bool<false>){
    Tran<CurrentBase,S,T>::entryActions(h,
                           Bool<entryStop>());
    C::entry(h);
  }
  Tran(Host& h) : host_(h)
       { exitActions(host_,Bool<false>()); }
  ~Tran() { Tran<T,S,T>::entryActions(host_,
          Bool<false>()); T::init(host_); }
  Host& host_;
};
```

It uses a gadget described in Herb Sutter's GotW #71 [4]. It is used to test at compile time whether a class `D` is derived from a class `B` either directly or indirectly. This is an important ingredient in the mechanism that figures out the exit/entry actions to call. Here it is:

```
template<class D, class B>
class IsDerivedFrom {
private:
  class Yes { char a[1]; };
  class No { char a[10]; };
  static Yes Test( B* ); // undefined
  static No Test( ... ); // undefined
public:
  enum { Res = sizeof(Test(static_cast<D*>(0)))
             == sizeof(Yes) ? 1 : 0 };
};
```

So how does `Tran` work? I explained already that all the exit actions are called when a `Tran` object is constructed and all entry actions are called when it is destructed again. Our states are all different types, so `Tran` needs to be a template. Its template parameters are the type of the current state (which is always a leaf state), the source state (where the transition arrow originates, which may be a composite state that contains the leaf state either directly or indirectly) and the type of the target state.

`Tran` now needs to walk up in the inheritance hierarchy of the current state (C) until it finds the common base class of current and target state (C and T), but it must not stop before the source state (S) was reached. From there it needs to descend the hierarchy down to the target state T. While ascending, it needs to call the exit actions of the states along the way, and when descending it needs to call the entry actions of the states along the way.

Ascending uses template recursion in `exitActions()`, and descending uses a similar recursion in `entryActions()`. The additional `Bool` parameter of these functions is used to terminate the recursion at the right point via overloading. Finding the right point is where Herb's gadget enters the picture.

The point where the recursion needs to terminate is at the first state that is common to both source and target state, in other words a common base class of both states. So when you are ascending from the source state you will eventually encounter a base class of the target state, and there's where the recursion must end. Similarly when ascending from the target state you will eventually encounter a state that is a base class of the source state.

So we know how to ascend from both ends towards the common base class, but we actually need to descend towards the target class once we have ascended from the source state, so it appears as if we've got it the wrong way up. But this is not a problem, as we can ensure the correct order of the entry routines by just swapping the recursion point with the invocation of the action as seen in `entryActions()`. The effect is that in `exitActions()`, the actual exit actions are invoked as we drill recursively into the inheritance hierarchy, while in `entryActions()` the entry actions are invoked as we work ourselves back out of the hierarchy.

You can now also see why there is a member function template `handle()` in the `CompState`/`LeafState` class template. Since `Tran` needs to know the current state in order to work out which entry and exit actions to invoke, it is necessary to pass it up the inheritance hierarchy in the default case of each handler function's `switch` statement. If we didn't do that, transitions handled in the handler for a composite state would miss the exit actions of its substates.

Finally, the handling of the initial state of a composite state deserves explanation. Remember that targetting a composite state with a transition leads you to the initial state specified within the composite state. The init action of a target state is always executed after executing the entry action. The init action of a leaf state is to announce itself to the host class as the new state. This behaviour shouldn't be changed. A composite state by default has no init action. So if you target a composite state with a transition, you will get a compile-time error, unless you specifically provide an init function for this composite state. Inside such an init function, you use the following `Init` class template to specify the initial substate.

```
template<typename T>
struct Init {
  typedef typename T::HostClass Host;
  Init(Host& h) : host_(h) {}
  ~Init() { T::entry(host_); T::init(host_); }
  Host& host_;
};
```

A Test Run

When you compile all the code for our example, you may run a little test to see whether the actions are called in the right order. Here's what I got:

```
top-INIT;s0-ENTRY;s0-INIT;s1-ENTRY;s1-INIT;s11-ENTRY;
Signal<-a
s11-EXIT;s1-EXIT;s1-A;s1-ENTRY;s1-INIT;s11-ENTRY;
Signal<-e
s11-EXIT;s1-EXIT;s0-EXIT;s0-E;s0-ENTRY;
                     s2-ENTRY;s21-ENTRY;s211-ENTRY;
Signal<-e
s211-EXIT;s21-EXIT;s2-EXIT;s0-EXIT;s0-E;
            s0-ENTRY;s2-ENTRY;s21-ENTRY;s211-ENTRY;
Signal<-a

Signal<-h
s211-EXIT;s21-EXIT;s21-H;s21-ENTRY;s21-INIT;
                                        s211-ENTRY;
Signal<-h

Signal<-x
```

You'll notice that Miro's implementation renders a different result (page 100 in [1]). Notably, the actions associated with a transition are executed before any exit actions in Miro's version. This violates the UML rules, but Miro explains that this was deliberate, as obeying this rule would have made his implementation significantly more complicated. My code obeys the rule with no noticeable hit on code performance.

Furthermore, note that when pressing the `e` key, `s0` is exited and reentered. This is not immediately obvious from the way the diagram is drawn. In fact, Miro's code doesn't show this behaviour. The UML definition seems to specify the behaviour of my code, although this isn't entirely clear to me. It certainly is more consistent with the behaviour of self-transitions. If my interpretation turned out to be correct, it would be clearer to draw transition arrows in UML statecharts such that they always leave the source state boundary towards the outside, and also enter it from the outside. Hence the exit action of the source state is always called, even when the target is a substate of the source state. Likewise the entry action of the target state is called even when the source state is one of its substates.

Efficiency of the Generated Code

With such a lot of templates, you might worry about the kind of code generated. Templates are still being accused of causing code bloat, a reason why embedded programmers in particular still hesitate to use them. If used wisely, however, templates in conjunction with inlining can actually reduce the amount of code produced. So how does the system presented here fare in this respect?

Given a good quality compiler and a sensible setting of compiler switches, all `handle()`, `init()`, `entry()` and `exit()` functions will be inlined into the virtual `handler()` function for a state. As a result, you get as many handler functions as there are leaf states. A good compiler will also be able to fuse the `switch` statement of a `handle()` function with those from the `handle()` functions of the base classes, so that you effectively get a larger `switch` incorporating all cases that need to be considered in a state. The result is the same as if you had converted the hierarchical state machine into a flat one, taking all entry and exit actions into account, and implemented the handler function for each flat state manually. The code generated literally is exactly the same. The templates flatten the hierarchical state machine into a simple state machine and generate the code for that.

In particular, empty actions don't produce any code at all, not even a call to an empty procedure. If the entry and exit actions associated with a particular transition are also empty, the transition simmers down to a single assignment to the host class's state variable, which on many processors is just one or two instructions.

The result is probably about as fast as it gets, but there is no code sharing between states. This is the reason why you should not include a lot of action code in the handlers, but rather call a corresponding action function in the host class. This is particularly true for handler functions of composite states.

Afterword

Templates can be used to advantage here in order to allow the compiler to thoroughly optimize the code. It is even fairly readable and requires neither liberal casting nor preprocessor macros as the solution described by Miro Samek. It does require a dose of template metaprogramming, and this may challenge your compiler.

So we've got a balance of advantages and drawbacks:

+ Transitions are worked out at compile time, allowing generation of very efficient inlined code
+ Malformed statecharts are caught at compile time
+ Stylized code lends itself well to automatic code generation
+ The code is typesafe and doesn't need casts
+ Complete flexibility in representing events
– We need full template support in the compiler
– All transitions must be static (known at compile time)
– Only a subset of the functionality of UML statecharts is supported

[concluded at foot of next page]

boost::fsm

The well-known boost library [7] is about to acquire statechart support. Andreas Huber has developed a library that aims to cover the entire functionality of UML statecharts, and it should appear in one of the next official releases of boost. Until it is accepted, you may have a look in boost's sandbox: `http://boost-sandbox.sourceforge.net/libs/fsm/doc/index.html` is the entry point to the documentation accompanying `boost::fsm`.

Some of the design goals of `boost::fsm` match mine. Both facilitate direct coding of statecharts in C++ without the aid of special code generation tools. Such tools ought to be pretty straightforward in both cases, however. Both solutions are type safe and detect malformed statecharts at compile time.

The support of `boost::fsm` for the complete UML semantics makes it less efficient, although it should still surpass the efficiency of Miro Samek's implementation in many cases. In particular, entering a state is done through construction of a state object. It gets destructed again when exiting the state. As states are allocated using `operator new`, the heap manager is excercised unless you overload `operator new` and `operator delete`. This is done so that you can include your own extra data members with a state.

No dynamic allocation happens in the solution I introduced here. As noted, the resulting code should be fast and consume little memory. The downside is of course that some major facilities of UML statecharts aren't supported.

The benefit is yours: You've got a choice between a restricted, efficient solution and a flexible, universal solution.

`[continued from previous page]`

If you find this approach useful, have improvements or comments on offer or bugs to fix, I'd like to hear from you.

I'd like to thank Miro Samek and Andreas Huber for discussion and advice as well as for their work on HSM implementations. My work wouldn't exist without theirs. Thanks also to the Overload reviewers.

Stefan Heinzmann

References

[1] Miro Samek, *Practical Statecharts in C/C++*, CMP Books 2002. There's a companion website with additional information: `http://www.quantum-leaps.com`

[2] Booch, Rumbaugh, Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley

[3] Rumbaugh, Jacobson, Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley 1999

[4] `http://www.gotw.ca`

[5] `http://www.ilogix.com`

[6] Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley 1995

[7] `http://www.boost.org`