State-Oriented Programming

Implementing hierarchical state machines doesn't mean you have to use code-synthesizing tools. Here are some techniques for simple, efficient, and direct mapping of state machines to C or C++.

The Unified Modeling Language (UML) provides a number of conceptual and graphical views for capturing and conveying designs. Among these views, hierarchical state machines (HSMs), based on Harel statecharts, are of key importance and provide the foundation for automatic code generation from object models.[1] Unfortunately, this creates the impression that the methodology is only accessible through use of code synthesizing tools. This is similar to the common belief that object-oriented programming (OOP) is only possible with object-oriented (OO) languages. However, the key OO concepts of encapsulation, inheritance, and polymorphism may be implemented as design patterns[2] in a non-OO language such as C. Similarly, hierarchical state machines can be viewed as another such fundamental pattern.

From a more abstract perspective, one may view hierarchical state machines as a meta-pattern, in that various structured uses become design patterns (behavioral patterns[3]) in their own right. This is analogous to OO design patterns[7] built on meta-patterns of inheritance and polymorphism. Following this analogy to OOP, we propose the term *state-oriented programming* (SOP) to describe a programming style based on HSMs.

The primary goal of this article is to present a simple and efficient implementation of the HSM design pattern. By providing easy-to-use C and C++ recipes for generating HSMs, we hope to make the major benefits of the technology more accessible to the software community. The proposed implementation techniques are valuable in that they raise the level of abstraction and allow for straightforward mapping of UML statecharts to compact and efficient code in C or C++. We have used the technique extensively in deeply embedded, hard real-time RF receiver applications where both high speed and small memory footprint were crucial.

In an effort to maximize efficiency and minimize implementation complexity, many of the more advanced features of UML statecharts have been omitted. The implemented features form a proper subset of UML statecharts and include:

State-oriented programming raises the level of abstraction and allows for straightforward mapping of UML statecharts to compact and efficient code in C or C++.

- Nested states with proper handling of group transitions and group reactions
- Guaranteed execution of entry/exit actions upon entering/exiting states
- Straightforward implementation of conditional event responses (guards)
- Design that enables inheriting and specializing state models

More advanced features of UML statecharts (such as history mechanisms and orthogonal regions) can be added as behavioral patterns built on top of the implementation presented here.

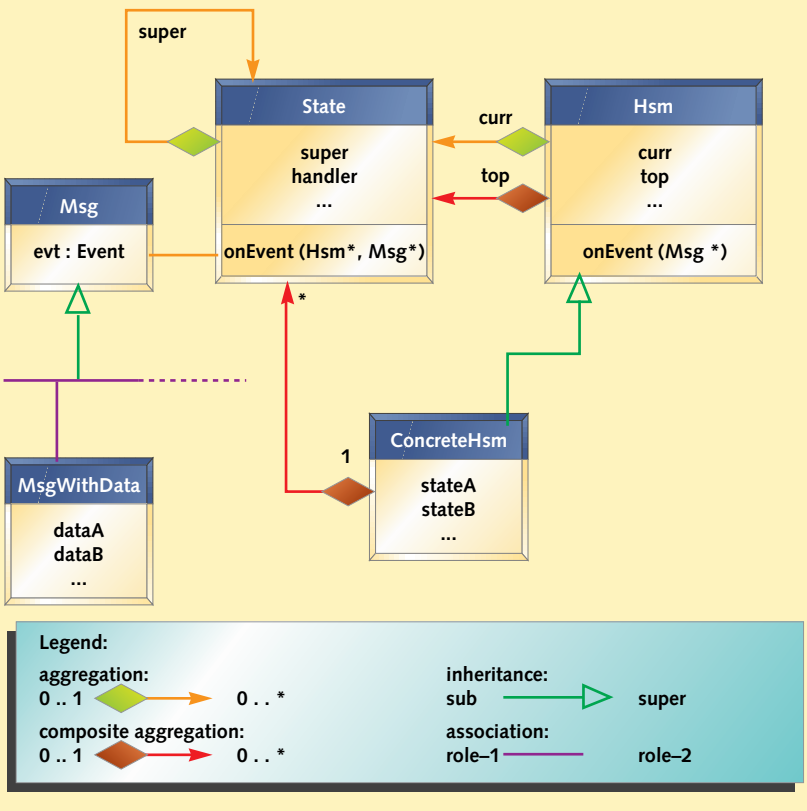
We begin with a brief summary of approaches that have been documented in the relevant literature or implemented in commercial products. We then describe our implementation using UML class diagrams and provide a complete implementation in both C and C++. Taking the example of a simple digital watch, we demonstrate how to map a UML state diagram to code and how to use most features of the HSM implementation in a concrete fashion. We briefly discuss how the HSM pattern, when combined with RTOS facilities, can be used to build a powerful real-time framework. We conclude by drawing a comparison between SOP and OOP and propose some modifications and extensions to UML statecharts. We assume that the reader is familiar with basic concepts of state machines and UML notation.[3,11]

Standard approaches

Typical implementations of state machines in C/C++ include:

- Doubly nested `switch` statements with a scalar "state variable" used as the discriminator in the first level of the switch and event-type in the second.[4] This is, perhaps, the most common technique and works well for classical "flat" state machines and is widely employed by automatic code synthesizing tools.[5] Manual coding of entry/exit actions and nested states is, however, cumbersome, mainly because code pertaining to one state becomes distributed and repeated in many places, making it difficult to modify and maintain when the topology of state machine changes
- Action-state tables containing typically sparse arrays of actions and transitions for each state.[4] Actions (including entry/exit, state reactions, and actions associated with transitions) are most commonly represented as pointers to functions. Representing state hierarchy in a flat action-state table is cumbersome. Also, this approach requires a large (and consequently wasteful) action/state array and many fine-granularity functions representing actions
- Generic "state machine interpreters" driven by typically complex data structures that represent the hierarchy of states together with entry/exit actions and transitions.[6] This is a generalized action-state table approach that attempts

FIGURE 1 Structure of HSM design pattern



RUPERT ADLEY

LISTING 1a hsm.h—C header file

```
typedef int Event;
typedef struct {
  Event evt;
} Msg;

typedef struct Hsm Hsm;
typedef Msg const *(*EvtHndlr)(Hsm*, Msg const*);

typedef struct State State;
struct State {
  State *super;                              /* pointer to superstate */
  EvtHndlr hndlr;                          /* state's handler function */
  char const *name;
};

void StateCtor(State *me, char const *name,
               State *super, EvtHndlr hndlr);
#define StateOnEvent(me_, ctx_, msg_)\
  (*(me_)->hndlr)((ctx_), (msg_))

struct Hsm {                    /* Hierarchical State Machine base class */
  char const *name;                          /* pointer to static name */
  State *curr;                                        /* current state */
  State *next;               /* next state (non 0 if transition taken) */
  State top;                                   /* top-most state object */
};

void HsmCtor(Hsm *me, char const *name, EvtHndlr topHndlr);
void HsmOnStart(Hsm *me);                /* enter and start the top state */
void HsmOnEvent(Hsm *me, Msg const *msg);                /* "HSM engine" */

/* protected: */
unsigned char HsmToLCA_(Hsm *me, State *target);
void HsmExit_(Hsm *me, unsigned char toLca);
#define STATE_CURR(me_) (((Hsm *)me_)->curr)
#define STATE_START(me_, target_) \
  (assert((((Hsm *)me_)->next == 0),\
    ((Hsm *)me_)->next = (target_))

#define STATE_TRAN(me_, target_) if (1) { \
  static unsigned char toLca_ = 0; \
  assert(((Hsm *)me_)->next == 0);\
  if (toLca_ == 0) \
      toLca_ = HsmToLCA_((Hsm *)(me_), (target_));\
  HsmExit_((Hsm *)(me_), toLca_);\
  ((Hsm *)(me_))->next = (target_);\
} else ((void)0)

#define START_EVT ((Event)(-1))
#define ENTRY_EVT ((Event)(-2))
#define EXIT_EVT  ((Event)(-3))
```

to represent HSMs with a more efficient data structure than an array. Techniques in this category require each action to be coded as a separate function. They perform relatively poorly if the state machine is complex

- Object-oriented "State" design pattern based on delegation and polymorphism.[7,3] States are represented as subclasses implementing a common interface (each method in this interface corresponds to an event). A context class delegates all events for processing to the current state object. State transitions are accomplished by changing the current state object (typically re-assigning one pointer). This pattern is elegant and relatively efficient but is not hierarchical. Accessing context attributes from state methods is indirect (cannot use an implicit `this` pointer) and breaks encapsulation. The addition of new states requires subclassing and the addition of new events requires adding new methods to the common interface

Implementation

Our implementation of the HSM pattern is, to some degree, a combination of the techniques itemized above. The structure of the pattern is shown in Figure 1. This structure is greatly simplified relative to the standard full-featured UML design.[8]

States are represented as instances of the `State` class, but unlike the "State" pattern as described by Gamma et al.[7], the `State` class is not intended for subclassing but rather for inclusion as is. Accordingly, our approach requires state machines to be constructed by composition rather than by inheritance. The most important attributes of `State` class are the event handler (to describe behavior specific to the state) and a pointer to superstate (to define nesting of the state).

Messages are represented as instances of `Msg` class or its subclasses.

LISTING 1b hsm.h—C++ header file

```
typedef int Event;
struct Msg {
  Event evt;
};

class Hsm;                                        /* forward declaration */
typedef Msg const *(Hsm::*EvtHndlr)(Msg const *);

class State {
  State *super;                                  /* pointer to superstate */
  EvtHndlr hndlr;                               /* state's handler function */
  char const *name;
public:
  State(char const *name, State *super, EvtHndlr hndlr);
private:
  Msg const *onEvent(Hsm *ctx, Msg const *msg) {
    return (ctx->*hndlr)(msg);
  }
  friend class Hsm;
};

class Hsm {                      /* Hierarchical State Machine base class */
  char const *name;                             /* pointer to static name */
  State *curr;                                          /* current state */
protected:
  State *next;                /* next state (non 0 if transition taken) */
  State top;                                     /* top-most state object */
public:
  Hsm(char const name, EvtHndlr topHndlr);                        /* Ctor */
  void onStart();                        /* enter and start the top state */
  void onEvent(Msg const *msg);                /* "state machine engine" */
protected:
  unsigned char toLCA_(State *target);
  void exit_(unsigned char toLca);
  State *STATE_CURR() { return curr; }
  void STATE_START(State *target) {
    assert(next == 0);
    next = target;
  }
# define STATE_TRAN(target_) if (1) {\
     static unsigned char toLca_ = 0;\
     assert(next == 0);\
     if (toLca_ == 0)\
         toLca_ = toLCA_(target_);\
     exit_(toLca_);\
     next = (target_);\
   } else ((void)0)
};

#define START_EVT ((Event)(-1))
#define ENTRY_EVT ((Event)(-2))
#define EXIT_EVT  ((Event)(-3))
```

All messages carry event-type (attribute `evt` inherited from `Msg`) and possibly arbitrary data (added by subclassing).

Events are handled uniformly by *event handlers*, which are member functions of `Hsm` class (`typedef EvtHndlr`). As shown in Figure 1, a state machine consists of at least one state—the `top`-level state inherited from `Hsm`. Concrete state machines are built by inheriting from `Hsm` class, adding an arbitrary number of states (plus other attributes), and defining event handlers.

Because event handlers are methods of `Hsm` or its subclasses, they have direct access to attributes via the implicit `this` pointer (in C++) or the explicit `me` pointer (in C). Within event handlers, only one level of dispatching (based on event-type) is necessary. Typically this is achieved using a single-level `switch` statement. Event handlers communicate with the state machine engine (see Listing 2) through a return value of type `Msg*`. The semantic is simple: if an event is processed, the event handler returns 0 (NULL pointer); otherwise it returns ("throws") the message for further processing by higher-level states. To be compliant with UML statecharts, the returned message is the same as the received message, although return of a different message type can be considered. As we discuss later, returning the message provides a mechanism similar to "throwing" exceptions.

Entry/exit actions and default transitions are also implemented inside the event handler in response to the pre-defined events `ENTRY_EVT`, `EXIT_EVT`, and `START_EVT`. The state machine engine generates and dispatches these events to appropriate handlers upon state transitions. An alternative approach would be to represent entry/exit and start actions as separate methods, but this would require specification and maintenance of three additional (fine granularity) methods and three additional function pointers in each state.

LISTING 2a hsm.c—C implementation

```
static Msg const startMsg = { START_EVT };
static Msg const entryMsg = { ENTRY_EVT };
static Msg const exitMsg  = { EXIT_EVT };
#define MAX_STATE_NESTING 8
                                                        /* Hsm ctor */
void HsmCtor(Hsm *me, char const *name, EvtHndlr topHndlr) {
  StateCtor(&me->top, "top", 0, topHndlr);
  me->name = name;
}
                                      /* enter and start the top state */
void HsmOnStart(Hsm *me) {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  me->curr = &me->top;
  me->next = 0;
  StateOnEvent(me->curr, me, &entryMsg);
  while (StateOnEvent(me->curr, me, &startMsg), me->next) {
    for (s = me->next, trace = entryPath, *trace = 0;
         s != me->curr; s = s->super)
      *(++trace) = s;                          /* trace path to target */
    while (s = *trace )                     /* retrace entry from source */
      StateOnEvent(s, me, &entryMsg);
    me->curr = me->next;
    me->next = 0;
  }
}
                                            /* state machine "engine" */
void HsmOnEvent(Hsm *me, Msg const *msg) {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  for (s = me->curr; s; s = s->super) {
    if ((msg = StateOnEvent(s, me, msg)) == 0) {;
      if (me->next) {                        /* state transition taken? */
        for (s = me->next, trace = entryPath, *trace = 0;
             s != me->curr; s = s->super)
          *(++trace) = s;                      /* trace path to target */
        while (s = *trace )                   /* retrace entry from LCA */
          StateOnEvent(s, me, &entryMsg);
        me->curr = me->next;
        me->next = 0;
        while (StateOnEvent(me->curr, me, &startMsg),
               me->next) {
          for (s = me->next->super, trace = entryPath,
               *trace = 0; s != me->curr; s = s->super)
            *(++trace) = s;                   /* record path to target */
          while (s = *trace )                    /* retrace  the entry */
            StateOnEvent(s, me, &entryMsg);
```

The topology of a state machine is determined upon construction. The constructor of the concrete HSM is responsible for initialization of all participating `State` objects by setting the `super`-state pointers and the event handlers. An example of a state machine and corresponding data structures is shown in Figure 2. In our approach we do not distinguish between composite-states (states containing substates) and leaf states. All states are potentially composite.

State transitions are implemented as macros: `STATE_START()` and `STATE_TRAN()` (see Listing 2). The first macro (inline member function in C++) handles start transitions (transitions originating from a "black dot" pseudostate). The second macro `STATE_TRAN()` implements a regular state transition and is slightly more complex.

Due to the UML-specified order of invocation, all exit actions must precede any actions associated with the transition, which must precede any entry actions associated with the newly entered state(s). To discover which exit actions to execute, it is necessary to first find the *least common ancestor* (LCA) of the source and target states. For example, the LCA of the transition triggered by event `e4` in Figure 2 is `s2` and the LCA for the transition triggered by event `e1` is `top`.

Finding the LCA can be expensive (see method `HsmToLCA_()` in Listing 2). However, for any given transition the LCA needs to be calculated only once. Method `HsmToLCA_()` returns the number of levels from the current state to the LCA rather than a pointer to the LCA state itself. The former is the same for all instances of a given HSM, that is, it is characteristic of the `Hsm` (sub)class rather than individual state machine objects. For this reason it can be stored in a `static` variable shared by all instances.

The `STATE_TRAN()` macro ensures that all exit actions to the LCA will be executed. The user must then explicitly invoke any actions associated with

LISTING 2a, cont'd. hsm.c—C implementation

```
            me->curr = me->next;
            me->next = 0;
          }
        }
        break;                                    /* event processed */
      }
    }
  }
                    /* exit current states and all superstates up to LCA */
void HsmExit_(Hsm *me, unsigned char toLca) {
  register State *s;
  for (s = me->curr; toLca > 0;  toLca, s = s->super)
     StateOnEvent(s, me, &exitMsg);
  me->curr = s;
}
                           /* find # of levels to Least Common Ancestor */
unsigned char HsmToLCA_(Hsm *me, State *target) {
  State *s, *t;
  unsigned char toLca = 1;
  for (s = me->curr->super; s != 0; ++toLca, s = s->super)
    for (t = target; t != 0; t = t->super)
      if (s == t)
        return toLca;
  return 0;
}
```

the transition and return from the event handler. The framework will then correctly execute any required entry actions.

The state machine engine (method `Hsm::onEvent()` from Listing 2) is small, due mostly to the simple data representation employed. To minimize stack use and maximize performance we were careful to replace potential recursion (natural in hierarchical state machines) with iteration.

Perhaps the weakest part of the implementation lies in execution of entry actions during state transitions. Entry actions must be executed in order from the least deeply nested to the most deeply nested state.[11] This is opposite to the "natural" navigability in our data structure (see Figure 2). This problem is solved by first recording the entry path from the LCA to the target, then "playing it backwards" with execution of entry actions. By applying a technique similar to that described previously for LCA calculation, it is possible to record an entry path only once and avoid repetitive calculation. This optimization trades memory and additional complexity for speed improvement.

Sample application

To illustrate the use of the HSM pattern, consider a simple digital watch (Figure 3). The watch has two buttons—which generate external events—and an internally generated tick event. The different events are handled differently depending upon the mode of operation. The basic watch operates as follows:

- In `timekeeping` mode, the user can toggle between displaying date or current time by pressing the "mode" button
- Pressing the "set" button switches the watch into `setting` mode. The sequence of adjustments in this mode is: hour, minute, day, month. Adjustments are made by pressing the "mode" button, which increments the chosen quantity by one. Pressing the "set" button while adjusting month puts the watch back into timekeeping mode
- While in `setting` mode the watch ignores tick events
- Upon return to `timekeeping` mode the watch displays the most recently selected information, that is, if date was selected prior to leaving `timekeeping` mode, the watch resumes displaying the date, otherwise it displays the current time

A state diagram for the specification given above is depicted in Figure 3b and its partial implementation is shown in Listing 3. We apply the HSM pattern according to the following recipe:

1. Declare a new class, inheriting from `Hsm` class (the `Watch` class)
2. Put into this new class all states (`State` class instances) and other attributes (`tsec`, `tmin`, `thour`, and so on)
3. Declare an event handler method (member function) for every state. Don't forget to declare event handlers for inherited states, like `top`, whose behavior you intend to customize
4. Define the state machine topology (nesting of states) in the new class (the `Watch` class) constructor
5. Define events for the state machine (for example, as enumeration). You can use event-types starting from 0, because the pre-defined events use the upper limit of the `Event` type range (see Listing 1)
6. Define event handler methods. Code entry/exit actions and start-up transitions as response to pre-defined events `ENTRY_EVT`,

`EXIT_EVT`, and `START_EVT`, respectively. Provide code for other events using `STATE_TRAN()` macro for state transitions. Remember to return 0 (NULL pointer) if you handle the event and the initial message pointer if you don't

7. Execute the initial start transition by invoking `Hsm::onStart()`
8. Arrange to invoke `Hsm::onEvent()` for each incoming event

Real-time framework

The HSM design pattern is particularly suited for implementing behavior associated with *active objects* (objects that are the roots of threads-of-control in a multitasking system). The concept of active objects exists in many modeling languages under different names: active objects stereotype in UML[9], active instance in Schlaer-Mellor[10], and actor in ROOM[6]. We use the term "actor" because it's most compact.

The typical structure of a framework based on active objects is shown in Figure 4. The `Actor` class inherits

FIGURE 2 a) State diagram and b) corresponding data structure

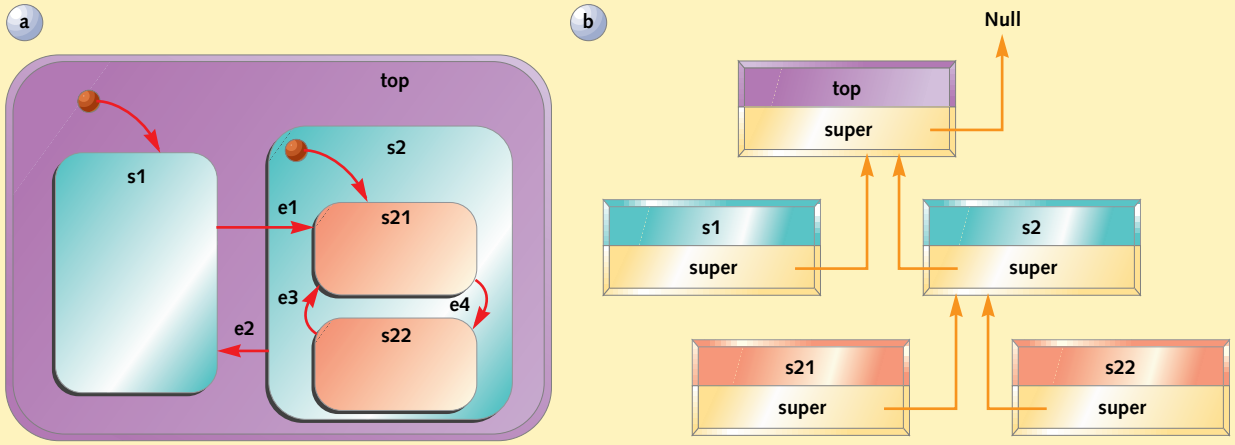
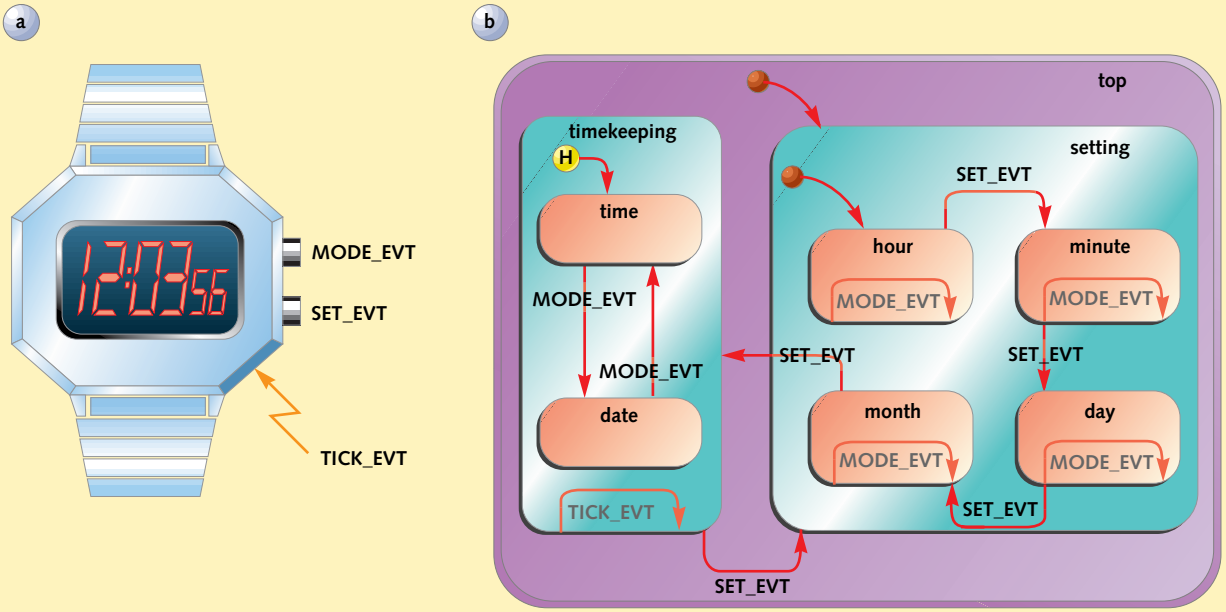


FIGURE 3 a) Simple digital watch events and b) corresponding state diagram



LISTING 2b hsm.cpp—C++ implementation

```
static Msg const startMsg = { START_EVT };
static Msg const entryMsg = { ENTRY_EVT };
static Msg const exitMsg  = { EXIT_EVT };
#define MAX_STATE_NESTING 8
                                                        /* Hsm ctor */
Hsm::Hsm(char const *name, EvtHndlr topHndlr)
  : top("top", 0, topHndlr) { this->name = name; }
                                      /* enter and start the top state */
void Hsm::onStart() {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  curr = &top;
  next = 0;
  curr->onEvent(this, &entryMsg);
  while (curr->onEvent(this, &startMsg), next) {
    for (s = next, trace = entryPath, *trace = 0;
         s != curr; s = s->super)
      *(++trace) = s;                          /* trace path to target */
    while (s = *trace )                     /* retrace entry from source */
      s->onEvent(this, &entryMsg);
    curr = next;
    next = 0;
  }
}
                                             /* state machine "engine" */
void Hsm::onEvent(Msg const *msg) {
  State *entryPath[MAX_STATE_NESTING];
  register State **trace;
  register State *s;
  for (s = curr; s; s = s->super) {
    if ((msg = s->onEvent(this, msg)) == 0) {              /*processed?*/
      if (next) {                           /* state transition taken? */
        for (s = next, trace = entryPath, *trace = 0;
             s != curr; s = s->super)
          *(++trace) = s;                      /* trace path to target */
        while (s = *trace )                   /* retrace entry from LCA */
           s->onEvent(this, &entryMsg);
        curr = next;
        next = 0;
        while (curr->onEvent(this, &startMsg), next) {
          for (s = next->super, trace = entryPath, *trace=0;
               s != curr; s = s->super)
            *(++trace) = s;                    /* record path to target */
          while (s = *trace )                      /* retrace the entry */
            s->onEvent(this, &entryMsg);
          curr = next;
          next = 0;
        }
      }
```

HSM functionality from `Hsm` and adds to it a thread of execution (for example, via `taskId` attribute and task's `run()` method) and a message queue (via `msgQ` attribute). Actors can only communicate with each other by sending each other messages via message queues. The messages are processed by the HSM in *run-to-completion* steps. Run-to-completion ensures that actors don't have to deal with concurrency issues internally, thereby eliminating a whole class of difficult time-domain problems by design.

SOP vs. OOP

OOP introduces two fundamental types of inheritance: implementation (class) inheritance and interface inheritance.[7] Implementation inheritance defines an object's implementation in terms of another object's implementation. In contrast, interface inheritance enforces only object interface compatibility regardless, of implementation.

Hierarchical state machines introduce a third type of inheritance that is equally fundamental. We call this *behavioral inheritance*. To understand why hierarchy introduces inheritance and how it works, consider an empty (or transparent) substate nested within an arbitrary superstate. If such a substate becomes active it behaves in exactly the same way as its superstate, that is, it *inherits* the superstate's entire behavior. This is analogous to a subclass which does not introduce any new attributes or methods. An instance of such a subclass is indistinguishable from its superclass because, again, everything is inherited exactly.

This property of HSMs is fundamental because it requires only the *differences* from the superstate's behavior to be defined. One observes that all OO design principles (for example, the Liskov Substitution Principle) hold in HSM designs because one deals with inheritance in yet another form. The concept of behavioral inheritance describes the "is-a" ("is-in") relationship between substates

and superstates and should not be confused with inheritance of entire state models.[3]

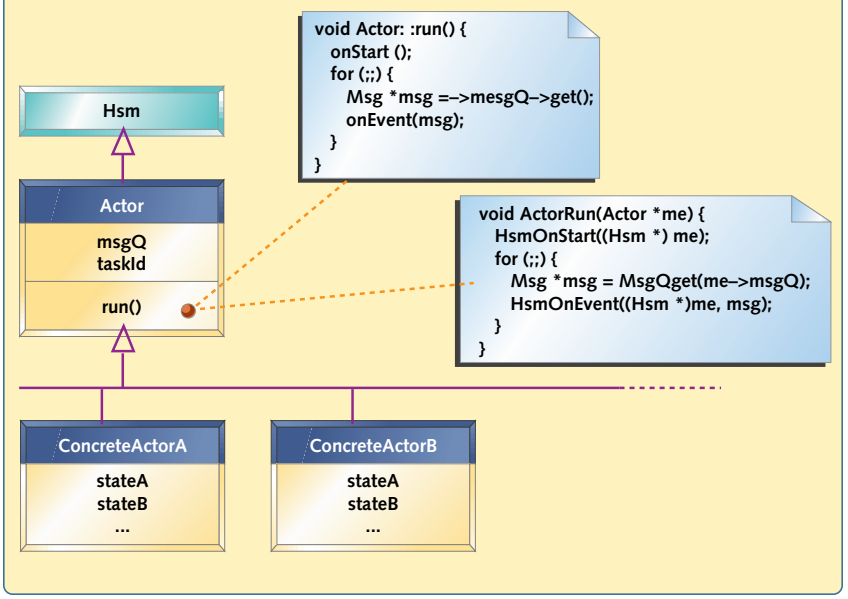
The analogy between SOP and OOP goes further. Class instantiation and finalization is similar to entering and exiting a state. In both cases special methods are invoked: constructors and destructors for objects, entry and exit actions for states. Even the order of invocation of these methods is the same: constructors are invoked starting from most remote ancestor classes (destructors are invoked in reverse order), and entry actions are invoked starting from the topmost superstate (exit actions are invoked in reverse order).

A final similarity between OOP and SOP lies in the way they are most efficiently implemented. Although polymorphism can be implemented in many ways, virtually all C++ compilers implement it in the same way: by using function pointers grouped into virtual tables. In view of the deep analogy between SOP and OOP, it is therefore not surprising that arguably the most efficient implementation of HSMs is also based on function pointers grouped into states. These simple state objects define both behavior and hierarchy but are not specific to any particular instance of a state machine. The same holds for virtual functions, which are characteristics of the whole class rather than specific to any particular object instance. For this reason we observe that state objects could (and probably should) be placed in v-tables and be supported as a native language feature.

LISTING 2b, cont'd. hsm.cpp—C++ implementation

```
      break;                                        /* event processed */
    }
  }
}
                  /* exit current states and all superstates up to LCA */
void Hsm::exit_(unsigned char toLca) {
  register State *s;
  for (s = curr; toLca > 0;  toLca, s = s->super)
    s->onEvent(this, &exitMsg);
  curr = s;
}
                           /* find # of levels to Least Common Ancestor */
unsigned char Hsm::toLCA_(State *target) {
  State *s, *t;
  unsigned char toLca = 1;
  for (s = curr->super; s != 0; ++toLca, s = s->super)
    for (t = target; t != 0; t = t->super)
      if (s == t)
        return toLca;
  return 0;
}
```

FIGURE 4 Structure of real-time framework based on HSM



Improvments to UML?

UML state diagrams do not provide any graphical representation of state reactions, which are reactions to events not causing state transitions. In practice however, reactions are common and sometimes the whole state hierarchy is most naturally designed to reuse group reactions rather than group transitions. In these cases a UML state diagram cannot be properly understood because it is simply incomplete.

We propose to include reactions in state diagrams as directed lines starting and finishing in the same state and lying entirely *within* that state. (An example of this notation is shown in Figure 3 for reactions to `TICK_EVT` and `MODE_EVT`.) Please note that this nota-

LISTING 3a Simple watch HSM: C implementation

```
#include "hsm.h"

typedef struct Watch Watch;
struct Watch {
  Hsm super;                                          /* superclass */
  int tsec, tmin, thour, dday, dmonth;
  State timekeeping, time, date;
  State setting, hour, minute, day, month;
  State *tkeepingHist;
};

enum WatchEvents {
  MODE_EVT,
  SET_EVT,
  TICK_EVT
};
                                        /* timekeeping state handler */
Msg const *WatchTimekeepingHndlr(Msg const *msg) {
  switch (MsgGetEvt(msg)) {
    case START_EVT:
      STATE_ENTER(me->tkeepingHist ? me->tkeepingHist :
                  &me->time);
      return 0;
    case SET_EVT:
      STATE_TRAN(&me->setting);
      printf("Watch::timekeeping-SET;");
      return 0;
    case TICK_EVT:
      WatchTimeTick(me);
      return 0
    case EXIT_EVT:
      me->tkeepingHist = STATE_CURR(me);
      return 0;
  }
  return msg;
}
/*... other state handlerds ... */
                                                /* Watch constructor */
void WatchCtor(Watch *me) {
   HsmCtor((Hsm *)me, "Watch", (EvtHndlr)Watch_top);
   StateCtor(&me->timekeeping, "timekeeping",
      &((Hsm *)me)->top, (EvtHndlr)Watch_timekeeping);
      StateCtor(&me->time, "time", &me->timekeeping,
            (EvtHndlr)Watch_time);
      StateCtor(&me->date, "date", &me->timekeeping,
            (EvtHndlr)Watch_date);
   StateCtor(&me->setting, "setting", &((Hsm *)me)->top,
            (EvtHndlr)Watch_setting);
      StateCtor(&me->hour, "hour", &me->setting,
            (EvtHndlr)Watch_hour);
```

tion is different from self-transitions, which also begin and end in the same state, but lie entirely *outside* the state. Self-transitions are different from reactions because they cause execution of entry and exit actions.

UML statecharts distinguish composite-states (states with substates) from leaf-states (states without substates) in that they only allow leaf-states to be active. This is an unnecessary limitation, which occasionally creates degenerate (empty) substates. To extend the analogy with OOP, this is like saying that a class with subclasses must necessarily be abstract. Our HSM implementation does not distinguish between composite and leaf states; it allows any state to be active.

In our experience with receiver-applications we frequently encountered the following situation: in response to a given event, we wished to perform an action (for example, update a digital filter) and then—depending on the state of the filter—potentially take a (conditional) state transition. We did not want to treat the filter update as part of the guard condition because this would add a side effect to the guard (typically a bad idea[11]). The only alternative is to treat the filter update as an action.

UML requires that actions associated with transitions be executed *after* all exit-actions from the source state but before any entry-actions to the target state.[11] At this point however, it is not yet know if the transition will be required at all. In general, this aspect of UML semantics makes it difficult to mix conditional execution of reactions and transitions. A UML-compliant solution would require specification of a pure reaction (update of the digital filter in this case) and then conditional propagation of *another* event to "self," specifically to trigger a pure state transition.

Because our implementation performs state transitions using the

LISTING 3a, cont'd. Simple watch HSM: C implementation

```
        StateCtor(&me->minute, "minute", &me->setting,
              (EvtHndlr)Watch_minute);
        StateCtor(&me->day, "day", &me->setting,
              (EvtHndlr)Watch_day);
        StateCtor(&me->month, "month", &me->setting,
              (EvtHndlr)Watch_month);
    me->tsec = me->tmin = me->thour = 0;
    me->dday = me->dmonth = 1;
    me->timekeepingHist = NULL;
}

void main() {                                          /* test harness */
    Watch watch;
    WatchCtor(&watch);
    HsmOnStart((Hsm *)&watch);
    for (;;)  {
        Msg *msg = getEvt();                /* block until event arrives */
        HsmOnEvent((Hsm *)&watch, msg);
    }
}
```

LISTING 3b Simple watch HSM: C++ implementation

```
#include "hsm.h"

class Watch : public Hsm {
  int tsec, tmin, thour, dday, dmonth;
  timeTick();                 /* process tick event in the 'time' state */
protected:
  State timekeeping, time, date;
  State setting, hour, minute, day, month;
  State *tkeepingHist;                              /* to record history */
  Msg const *topHndlr(Msg const *msg);
  Msg const *timekeepingHndlr(Msg const *msg);
  Msg const *settingHndlr(Msg const *msg);
  Msg const *hourHndlr(Msg const *msg);
  /*... other state handler methods */
public:
  Watch();                          /* ctor for defining state hierarchy */
};

enum WatchEvents {
  MODE_EVT,
  SET_EVT,
  TICK_EVT
};
```

`STATE_TRAN()` macro, which causes execution of all exit-actions up to least common ancestor, the order of execution is left to the designer. The UML-compliant implementation requires invoking the `STATE_TRAN()` macro *before* other actions associated with the transition. With our approach, the designer may choose a different order (for example, reaction-guard-transition), if it would simplify the problem at hand.

A final (proposed) extension to UML statecharts is associated with event handling at different levels of a hierarchical state machine. UML statecharts require that the same event be presented for processing to all levels of the hierarchy, starting with the active state. We propose to allow an event to change as it propagates upward through the state hierarchy. In our implementation this is simple to achieve. This feature would facilitate "throwing" an exception to a higher scope, which could in turn either handle or "throw" it again. Because outer states of an HSM are typically behavioral generalizations of inner states, this technique for handling exceptions is natural and arguably, makes more sense than the traditional exception handling technique of unwinding the call stack.

The benefits

The HSM design pattern allows hierarchical state machines to be directly and efficiently implemented in C or C++ without code synthesizing tools. The event-handler methods provide a concise textual representation of the state model and allow high-level structure and low-level details to be accessed easily. The simplicity of the event-handlers leads to "housekeeping" code[3]—portions of software that can be automatically generated by tools—that is trivial to write by hand.

The HSM pattern is flexible, allowing even fundamental changes

LISTING 3b, cont'd. Simple watch HSM: C++ implementation

```
                                          /* timekeeping state handler */
Msg const *Watch::timekeepingHndlr(Msg const *msg) {
  switch (msg->getEvt()) {
    case START_EVT:
      STATE_ENTER(tkeepingHist ? tkeepingHist : &time);
      return 0;
    case SET_EVT:
      STATE_TRAN(&setting);
      printf("Watch::timekeeping-SET;");
      return 0;
    case TICK_EVT:
      timeTick();
      return 0
    case EXIT_EVT:
      tkeepingHist = STATE_CURR();
      return 0;
  }
  return msg;
}
                                    /*... other state handlerds ... */
                                                /* Watch ctor */
Watch::Watch()
  :  Hsm("Watch", (EvtHndlr)topHndlr),
  timekeeping("timekeeping", &top,
    (EvtHndlr)timekeepingHndlr),
  time("time", &timekeeping, (EvtHndlr)timeHndlr),
  date("date", &timekeeping, (EvtHndlr)dateHndlr),

  setting("setting", &top, (EvtHndlr)settingHndlr),
  hour("hour", &setting, (EvtHndlr)hourHndlr),
  minute("minute", &setting, (EvtHndlr)minuteHndlr),
  day("day", &setting, (EvtHndlr)dayHndlr),
  month("month", &setting, (EvtHndlr)monthHndlr)
{
  tsec = tmin = thour = 0;
  dday = dmonth = 1;
  tkeepingHist = NULL;
}
                                                /* test harness */
void main() {
  Watch watch;
  watch.onStart();
  for (;;) {
    Msg *msg = getEvt();                  /* block until event arrives */
    watch.onEvent(msg);
  }
}
```

in state machine topology to be accomplished easily, even late in the process. Due to their hierarchical nature, models can be developed in incremental steps and remain executable throughout the development cycle. By requiring only specialization of behavior to be coded at nested levels of the state machine, common policy mechanisms (for example, exception handling) can be handled naturally. The state machine engine can easily be instrumented (an example is available in code accompanying this article at *www.embedded.com/code.html*) to produce execution trace, message sequence charts, or even animated state diagrams. In practice, however, we found it most useful to use a standard debugger to step through interesting parts of the code.

This implementation of the HSM pattern is no more complex than an internal implementation of inheritance or polymorphism in C++ and, in fact, has many similarities (both are based on function pointers). Given the many parallels, it seems reasonable to suggest that state-oriented programming should be directly supported by a (state-oriented) programming language in the same way that OOP is supported by object-oriented languages. We see this as beneficial for a couple of reasons. First, the compiler could place state objects in a virtual table and perform memory and performance optimizations at compile time (for example, computation of LCAs for state transitions). Second, the compiler could check consistency and well-formedness of the state machine, thereby eliminating many errors at compile time. In our view this is one direction in which C/C++ could evolve to better support future real-time applications. **esp**

Miro Samek is lead software architect at IntegriNautics Corp. He holds a PhD in

physics from Jagiellonian University in Cracow, Poland. Miro previously worked at GE Medical Systems where he developed real-time software for diagnostics X-ray equipment. He may be reached at miro@integrinautics.com.

Paul Y. Montgomery is software group leader at IntegriNautics Corp. He holds a PhD in Aero/Astro engineering from Stanford University, specializing in control systems applications based on the Global Positioning System. He may be reached at paulm@integrinautics.com.

Resources

1. Bell, Rodney, "Code Generation form Object Models," *Embedded Systems Programming*, March 1998, p. 74.
2. Samek, Miro, "Portable Inheritance and Polymorphism in C," *Embedded Systems Programming*, December 1997, p. 54.
3. Douglass, Bruce Powell. *Doing Hard Time*. Reading MA: Addison-Wesley Longman, 1999.
4. Douglass, Bruce Powell. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Reading, MA: Addison-Wesley, 1998.
5. *www.i-logix.com*
6. Selic, Bran, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. New York City: John Wiley & Sons Inc., 1994.
7. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
8. *www.omg.org/docs/97-96-02.pdf*, UML 1.1 Alpha—Behavioral Elements: State Machines.
9. Douglass, Bruce Powell and Srini Vason, "Temporal Models in UML," *Dr. Dobbs Journal*, December 1999.
10. Levkoff, Bruce, "A Schlaer-Mellor Architecture for Embedded Systems," *Embedded Systems Programming*, November 1999, p. 88.
11. Douglass, Bruce Powell, "UML Statecharts," *Embedded Systems Programming*, January 1999, p. 22.