Class 265: Implementing UML Statechart Diagrams

Embedded Systems Conference

April 2001

Carolyn Duby
Pathfinder Solutions
carolynd@pathfindersol.com

1 Introduction

The UML Statechart Diagram is a powerful tool for specifying the dynamic behavior of reactive objects. Reactive objects are objects that respond to events sent from other objects. The response of the reactive object to an event depends on what state the object is in at the time that the event occurred. Implementing a simple Moore State Model, a state model with actions only on state entry is fairly easy. Commonly state models are stored in a two-dimensional array indexed by state and event. When an active object is in a state and receives an event, the execution framework gets the next state of the object by accesses the array element at the index of the current state and the received event. Then the execution framework executes the action associated with the next state.

The UML Statechart Diagram extends Moore state machines to offer many convenient modeling features such as composite states, exit actions, actions on transitions, and guards. These advanced features allow developers to model more complex behavior in a compact form. Although convenient for modeling, these features can be quite daunting when you consider how to implement them. This paper introduces a set of Standard C++ classes that implement UML Statechart Diagrams associated with UML classes.

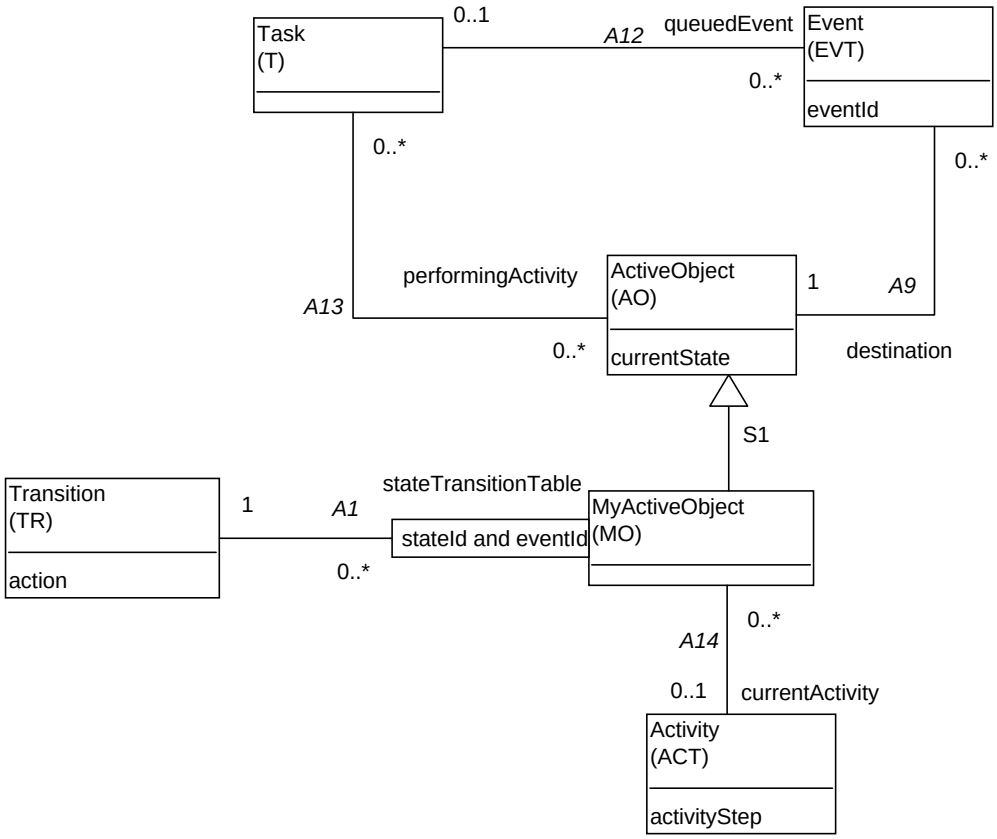

Figure 1: UML Class Diagram of Statechart Mechanisms

2 Statechart Mechanisms

The mechanisms implementing UML Statechart Diagrams are shown in the UML Class Diagram in Figure 1. The source code for the corresponding C++ files may be found in the demo section for this paper. A reactive system is composed of a set of ActiveObject instances that respond to events. Each class

in the system that defines a Statechart Diagram derives from the ActiveObject base class. The receipt of an event causes the active object to transition to a new state, perform a series of actions, and start activities.

An action is an atomic computation that changes the state of the active object. Actions are performed upon entry or exit from a state or upon the traversal of a transition. The active object cannot accept any events while it is executing an action. Thus an action is assumed to have a short duration.

In contrast, an activity is performed repeatedly while the active object is in the state defining the activity. An activity generally takes longer to complete than an action and has defined points at which it can be interrupted by events. When an event causes a transition out of the state, the activity terminates and a new activity associated with the destination state may begin.

Automated translation of UML Statechart Diagrams into an implementation language requires that the actions and activities to be formally specified in an analysis-level action language or in the target implementation language. Specifying actions in an action language rather than the target-implementation language hides the details of how modeling constructs such as associations and events are implemented and makes the models implementation-language independent. UML does not currently specify a standard action language however a group of Object Management Group (OMG) members is currently investigating a standard for action semantics.

Each event carries the identifier of the event and a pointer to the active object that will receive the event. Events may optionally carry parameters accessible to actions executed as a result of a transition caused by the event. Events carrying parameters derive from the Event class adding member variables that store the data associated with the event. The Event class defines the member functions that dispatch the event to its destination active object.

The Task is a singleton class that controls the execution path of the system. The main flow of control in a reactive system is based on events rather than function calls. When an active object sends an event to another active object, the event is placed on the Task's event queue. The main function in the Task is a continuous loop that reads the next event in the queue and dispatches the event to its destination active object.

The task also keeps a list of active objects that are currently executing activities. After the Task's main loop delivers the next queued event, the main loop executes the activity for the next active object in the activity list. Switching between dispatching events and executing activities allows the other active objects in the system to proceed and also allows the activities to be interrupted by incoming events.

The stateTransitionTable is a two-dimensional array of pointers to transition action functions indexed by state and event. When an object receives an event, the execution mechanisms look up and execute the transition action function array element at the state and event index. One instance of the stateTransitionTable specification is shared by all instances of a particular active object subclass. Each ActiveObject instance will potentially dwell in a different state so the ActiveObject stores the current state of the instance and the activity it is currently executing in attribute values.

The stateTransitionTable implements transitions triggered by events. After executing the actions for the triggered transition, the transition action function executes the actions for any untriggered transitions out of the destination state.

A State may have many outgoing Transitions on the same event or many outgoing untriggered Transitions as long as the transitions have different guard conditions. The transition action function for a guarded transition is an if statement that evaluates the guard conditions and executes the actions for the transition for the first guard condition that evaluates to TRUE.

3 Subclassing ActiveObject

All classes that define Statechart Diagrams will inherit from the ActiveObject class. The ActiveObject class and an example subclass follow:

```
class ActiveObject {
```

```
      public:
          ActiveObject(State initial_state);
          // derived type determines event handling
          virtual bool takeEvent( Event*  event) = 0;
          // redefined in all subclasses that use activities
          virtual void doActivity();
          State currentState() const { return m_currentState;}
      protected:
          void setState(State    current_state);
          bool deferEventAction(Event* event);
      private:
          State         m_currentState;
};

// reactive class on Class Diagram that has a Statechart Diagram
class MyActiveObject : public ActiveObject
{
public:
      // perform next step of current activity
      virtual void doActivity();
      // accept an event
      virtual bool takeEvent(Event* event);

      // transition action functions for triggered transitions
      bool fromAOnE1(Event* event);
      bool fromCOnE2(Event* event);
      bool fromEOnE3(Event* event);
      bool fromFOnE1(Event* event);
      bool fromFOnE4(Event* event);
      bool fromFOnE5(Event* event);
      bool fromGOnE6(Event* event);
      bool fromHOnE7(Event* event);
      bool fromIOnE2(Event* event);

      // transition functions for untriggered transitions
      void fromBUntriggered();
      void fromIUntriggered();

      enum events {
            EVENT_E1,
            // rest of events…
            NUM_EVENTS
      };
      MyActiveObject();

      // state entry and exit actions
      void enterStateA(const Event*);
      void exitStateA();
      void enterStateB(const Event*);
      void exitStateB();
      void enterStateC(const Event*);
```

```
      // activity steps
      void doActivityI1();
      void doActivityI2();

private:
      // omit activity implementation if no states start activities
      // pointer to function implementing next step of activity
      typedef void (MyActiveObject::*Activity)();
      // begin and end activity
      void beginActivity(Activity act);
      void endActivity();
      // activity currently being executed
      Activity  m_currentActivity;

      enum states {
            STATE_A,
            STATE_B,
            // rest of states…
            NUM_STATES
      };
      // pointer to transition action function
      typedef bool (MyActiveObject::*Transition)(Event* event);
      // state transition table
      static Transition m_stateTable[NUM_STATES][NUM_EVENTS];
};
```

Each subclass has a static member variable m_stateTable to store the array of legal states and transitions for all instances of the subclass. We cannot specify the static m_stateTable member variable as a part of the ActiveObject class because each ActiveObject subtype will define its own states and transitions. The m_stateTable array is initialized at compile time. In addition each ActiveObject subclass has a constructor that enters the initial state.

The ActiveObject subclass defines member functions called by the execution mechanisms when the ActiveObject enters or exits a state, takes a transition, or performs an activity.

3.1 State Entry and Exit Functions

An active object subclass has a member function for each state with an entry action or an activity. The member function has the following signature:

```
void ActiveObjectName::enterStateName(const Event* event);
```

The state entry function executes the actions specified by the entry action and starts any activities associated with the state. To begin an activity, the member function sets the m_currentActivity function pointer to the activity function and adds the active object to the Task's activity queue.

An active object subclass has a member function for each state with an exit action or an activity. The member function has the following signature:

```
void ActiveObjectName::exitStateName();
```

The state exit function executes the actions specified by the exit action and ends any activities associated with the state. To end an activity, the member function sets the m_currentActivity function pointer to NULL and removes the active object from the Task's activity queue.

Entry functions read data carried by the event causing the entry by dereferencing the event pointer argument.

3.2 Transition Action Functions

Triggered transition action functions call the collection of actions that execute as a result of taking a transition. Create a triggered action function for each explicit transition out of a non-nested state and if the non-nested state does not have any outgoing unguarded untriggered transitions or the state has activities, create a triggered transition action function for each triggered transition inherited from the parent composite state. A triggered transition action function has the following signature:

```
bool ActiveObjectName::fromStateNameOnEventName(Event* event);
```

The boolean return value indicates whether the transition consumed the event. A true return value indicates that the event was consumed and should be deleted. A false return value indicates that the event was deferred and should not be deleted.

The event parameter allows the action function to access the data associated with the event causing the transition. Transition action functions are entered into a function pointer array so each transition function must have the same signature. To access the parameters associated with the event, use a dynamic_cast or some other form of safe downcast to the specific event type that caused the transition.

Triggered transition action functions exit the current state, execute the actions for the transition, enter the destination state, and update the m_currentState member variable. If the destination state does not have activities, the action function calls from*StateName*Untriggered to execute any untriggered transitions out of the destination state.

To determine the order of entry and exit action execution for nested states, calculate the common ancestor state, the innermost composite state that is a parent of both the source and the next state. The action execution rules of UML specify that transitions must: 1) exit all the states and parent composite states starting at the current state up to but not including the common ancestor 2) execute the action on the transition 3) enter all the states starting at one nesting level below the common ancestor down to the destination state.

If a triggered transition is guarded, the action function tests each guard condition and takes the transitions when it finds the first guard condition that evaluates to TRUE.

If the destination of a transition is a composite state, determine the destination by finding the initial state of the composite.

For an internal transition, the transition action function executes the transition action only. It does not exit or enter the state or perform any untriggered transitions.

Untriggered transition action functions call the collection of actions that execute as a result of taking an untriggered transition. Create an untriggered transition action function when a non-nested state has an outgoing untriggered transition or when a non-nested state is the final state inside a composite state and the parent composite state has outgoing untriggered transitions. An untriggered transition action function has the following signature:

```
void ActiveObjectName::fromStateNameUntriggered();
```

Untriggered transition action functions are implemented exactly as triggered transition functions. The NULL pointer is passed to any entry functions expecting an event parameter.

3.3 Activity Functions

Activity functions execute the processing associated with the current step of an activity and set the current activity pointer to the next step in the activity. Activity functions have the following signature:

void *ActiveObjectName*::do*ActivityStepName*()

When an activity completes or it is interrupted by an event, the activity function removes the active object from the Task's activity queue and takes any untriggered transitions out of the current state.

3.4 Initializing the State Transition Table

The four steps to initializing an active object's State Transition Table specification are:

- define state and event identifiers
- initialize array of transition function pointers

3.4.1 Defining the State and Event Identifiers

The ActiveObject subclass must define an enumerated type to identify the states in the Statechart Diagram and another enumerated type to identify all the events received by the active object. The enumerated values are offsets into the stateTransitionTable array. The states and events enumerated values are nested inside the class to prevent name collisions with other classes that have the same state name. Include the state identifier enumerated type in the private section of the subclass since it will only be used to identify the current state. Add one enumerated value for each non-nested state in the Statechart Diagram. After including all of the states of the class, add a final enumerated value called NUM_STATES. The NUM_STATES value will represent the total number of states for the class.

Add an enumerated type that shows all of the events received by the class in the public section followed by the value NUM_EVENTS. The events enumerated values must be publicly accessible to other active objects since other active objects will be sending events to this subclass.

Use the NUM_STATES and NUM_EVENTS values to specify the dimensions of the state transition table:

```
typedef bool (MyActiveObject::*Transition)(Event* event);

Transition MyActiveObject::m_stateTable[NUM_STATES][NUM_EVENTS];
```

3.4.2 Initialize Array of Transition Function Pointers

Use a C++ array initializer to specify the action function to be called when an event is received while in a particular state.

- Ignored Event – If there is no legal transition from the state or from any of its parent composite states, set the array entry to NULL. The active object will ignore the event.
- Deferred Event – If the event is deferred, set the array entry to the function ActiveObject::deferEventAction. This function adds the event to the end of the event queue and returns false so the event is not deleted.
- Legal Transition – If there is a legal transition from the state or from a parent composite state, set the array entry to MyActiveObject::from*StateName*on*EventName* where *StateName* is the name of the state and *EventName* is the name of the event causing the transition.

See the file act_obj_init.cpp in the demo file section to see an example how to initialize the Statechart Diagram in Figure 2:

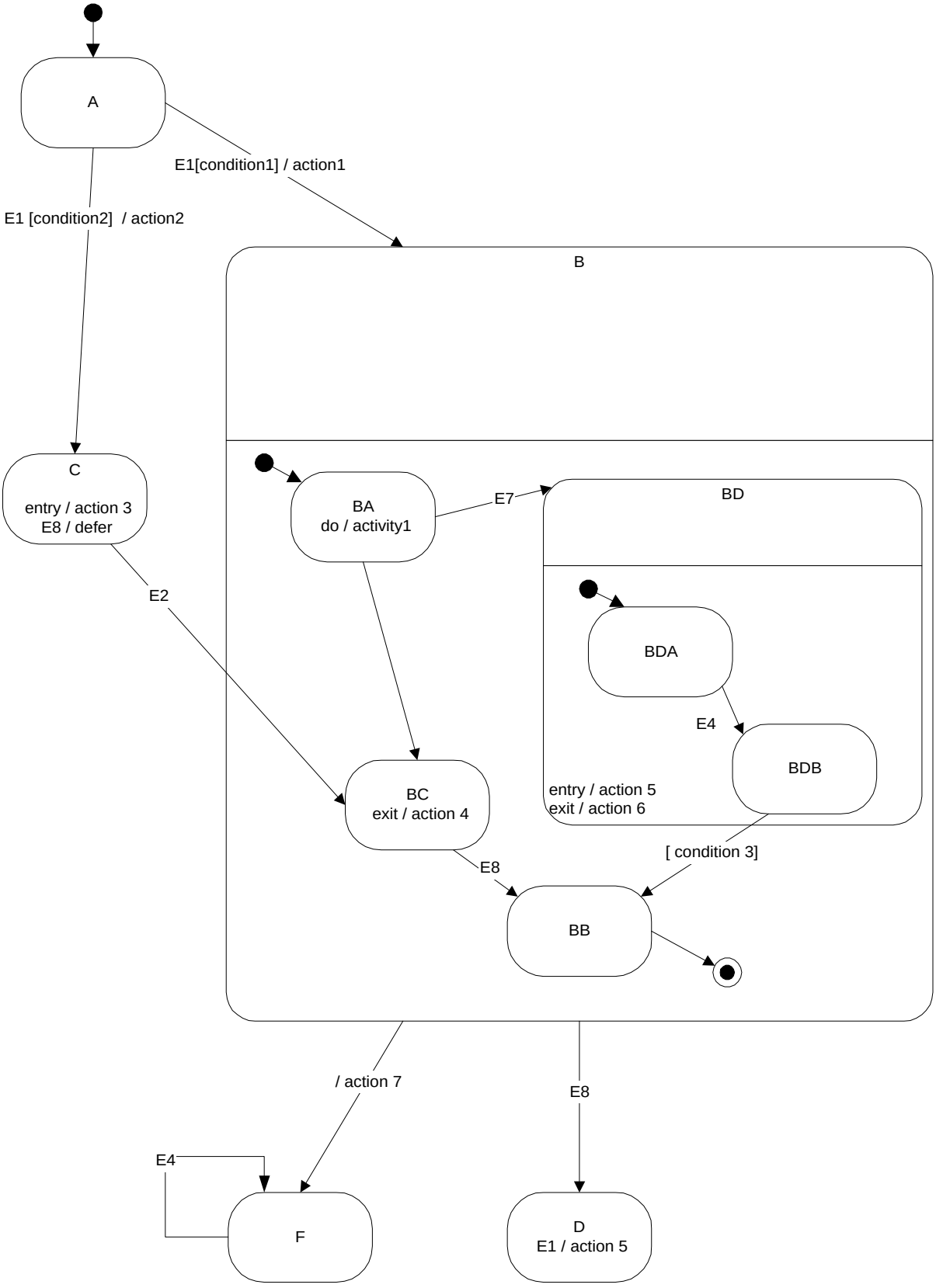

Figure 2 : Sample Statechart Diagram for MyActiveObject

4 Event and Activity Scenarios

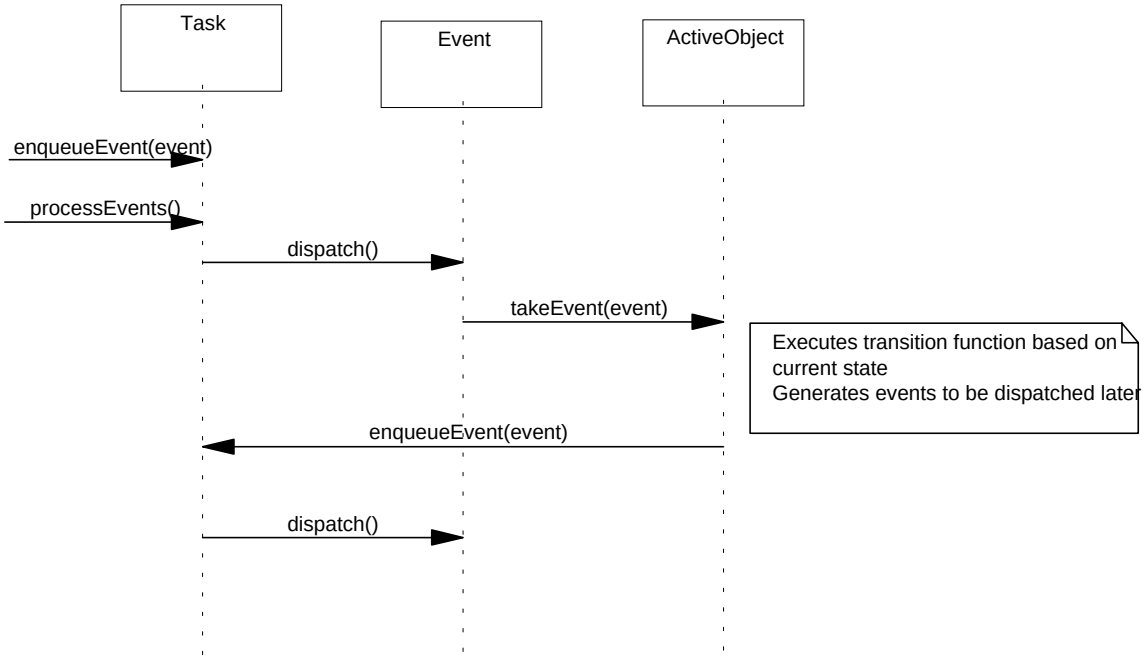

Figure 3: Message Sequence Chart for Dispatch Event Scenario

4.1 Dispatching an Event

The Task class implements the main event and activity loop. It supports enqueuing an event, starting and ending an activity, and shutting down the event loop. The task class is a singleton, meaning there is one instance of the Task per process. The single instance of the Task may be accessed by calling the static member function Task::singleTask that returns a pointer to the static member variable m_singleTask.

```
class Task {
public:
      void processEvents();
      void enqueueEvent(Event* event);
      void beginActivity(ActiveObject* object);
      void endActivity(ActiveObject* object);
      static Task* singleTask() { return &m_singleTask;}
      void shutdown() { m_stopProcessingEvents = true; }
private:
      Task();
      std::deque<Event*>                  m_eventQueue;
      std::list<ActiveObject*>            m_activityQueue;
      std::list<ActiveObject*>::iterator  m_currentActiveObject;
      bool                                m_stopProcessingEvents;
      static Task                         m_singleTask;
};
```

The enqueueEvent function takes an event allocated by new and adds it to the end of the event queue:

```
void Task::enqueueEvent(Event* event)
{
      m_eventQueue.push_back(event);
}

// enqueue event E1 directed at object
Task::singleTask()->enqueueEvent(
           new Event(MyActiveObject1::EVENT_E1, object));
```

The main loop is implemented in the Task by the function processEvents. The main loop continues processing events and activities while the event and activity queues are non-empty and until an action or activity calls the shutdown function setting the m_stopProcessingEvents member variable to true. The main loop alternates between dispatching events and performing activities. The loop checks the event queue first. If there is at least one event in the queue, the task pops the event from the front of the queue and dispatches the event to the destination active object.

The Task checks the activity list next. If there is at least one active object in a state that performs activities, the main loop calls the doActivity function on the current active object.

```
void Task::processEvents()
{
      Event*    next_event;
      while(!m_stopProcessingEvents && (m_eventQueue.size() != 0 ||
               m_activityQueue.size() != 0))
      {
            // if there are events in the queue to process
            if (m_eventQueue.size() != 0)
            {
                  next_event = m_eventQueue.front();
                  m_eventQueue.pop_front();
                  // delete event if it was consumed during dispatch
                  if (next_event->dispatch())
                  {
                        delete next_event;
                  }
            }
            if (m_activityQueue.size() > 0 && !m_stopProcessingEvents)
            {
                  // do the current activity
                  ActiveObject*       object = *m_currentActiveObject;
                  object->doActivity();
                  // advance to the next active object
                  m_currentActiveObject++;
                  // if we are at the end of the activity queue,
                  // wrap back to the beginning
                  if (m_currentActiveObject == m_activityQueue.end())
                  {
                        m_currentActiveObject =
                              m_activityQueue.begin();
                  }
            }
      }
}
```

Dispatching an event, delegates the event to its destination active object :

```
bool  Event::dispatch()
{
      return m_destination->takeEvent(this);
}
```

The take event function looks for a triggered transition action function in the state transition table given the current state and the event received. If the state transition table entry is non-NULL, takeEvent invokes the member function pointer for the transition action function:

```
bool MyActiveObject::takeEvent(Event* event)
{
      Transition    trans =
                        m_stateTable[currentState()][event->eventId()];
      bool  event_consumed = true;
      if (trans)
      {
            // invoke transition action function
            event_consumed = (this->*trans)(event);
      }
      return event_consumed;
}
```

The action functions executed as a result of taking the transition will add more events to the event queue keeping the event loop going.

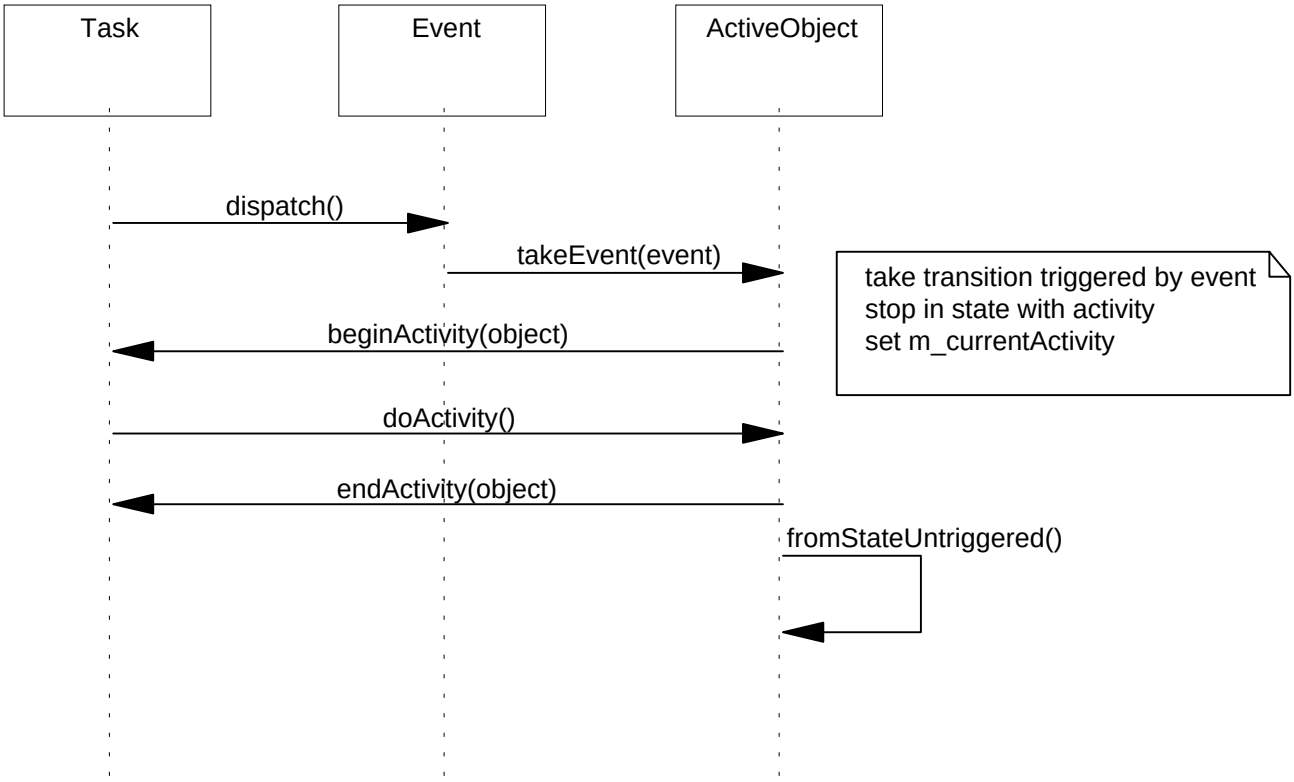

Figure 4: Message Sequence Chart for Activity Scenario

4.2 Transitioning to a State with Activities

The task dispatches an event to an active object. The takeEvent function invokes a transition action function that executes an entry action into a state that starts an activity. The state entry action sets the current ActiveObject subclass m_currentActivity function pointer to the function that executes the activity.

Then the state entry action calls beginActivity to add the active object to the activity queue. If this is the first active object in the activity queue, beginActivity initializes the current active object iterator to the newly added active object.

```
void Task::beginActivity(ActiveObject* object)
{
      m_activityQueue.push_back(object);
      // if this is the first active object in queue,
      // set the current active object iterator
      if (m_activityQueue.size() == 1)
      {
            m_currentActiveObject = m_activityQueue.begin();
      }
}
```

The transition action function completes and returns control to the Task's event/activity loop. The task calls the doActivity function for the active object subclass to invoke the current activity function:

```
void MyActiveObject::doActivity()
{
	if (m_currentActivity)
	{
		(this->*m_currentActivity)();
	}
}
```

The activity either completes or the object receives an event that interrupts the activity at a defined interruption point. The active object sets the m_currentActivity function pointer to NULL and calls endActivity to remove the activity from the Task's activity queue. The endActivity function checks to see if there are any active objects in the activity queue. If there are no active objects, the function returns. Otherwise endActivity determines if the current active object iterator is at the active object to be removed. If the iterator is at the activity to be removed, endActivity must update the iterator. If the activity being removed is not at the current iterator position, the active object may be removed without updating the iterator.

```
void Task::endActivity(ActiveObject* object)
{
	if (m_activityQueue.size() > 0)
	{
		// if removing the currently active object
		if (*m_currentActiveObject == object)
		{
			// update the current iterator position
			m_currentActiveObject =
				m_activityQueue.erase(m_currentActiveObject);
			if (m_currentActiveObject == m_activityQueue.end())
			{
				m_currentActiveObject =
					m_activityQueue.begin();
			}
		}
		else
		{
			// otherwise just remove the active object
			m_activityQueue.remove(object);
		}
	}
}
```

5 System Startup

5.1 Constructing Active Objects

The constructor for an active object subclass sets the current state to the default initial state and sets the current activity to NULL. Once the data for the active object subclass is initialized, the constructor calls the entry action function for the default initial state and the transition action function for any untriggered transitions out of the default initial state.

```
MyActiveObject::MyActiveObject() : ActiveObject(STATE_A),
m_currentActivity(NULL)
{
      enterStateA(NULL);
}

ActiveObject::ActiveObject(State initial_state):
            m_currentState(initial_state)
{
}
```

5.2 The Main Function

The main function creates any pre-existing instances, primes the event queue by generating any events necessary to start the system, and finally calls the processEvents loop to handle events and activities.

```
void main()
{
      // create pre-existing instances
      MyActiveObject      activeObject;
      // prime events
      Event* event = new Event(MyActiveObject::EVENT_E1,
                        &activeObject);
      Task::singleTask()->enqueueEvent(event);
      // handle events and activities
      Task::singleTask()->processEvents();
}
```

6 Performance

6.1 Implementation As Is

The implementation outlined is optimized for speed. In all cases, the amount of time to find the transition function is constant. The state transition table represented as a two-dimensional array uses one integer for each combination of state and event:

MU = S X E

MU = Amount of Memory Used

S = Number of States

E = Number of Events

The state transition table does not change during the execution of the program so it does not need to be stored in RAM. For example, the table could be stored in FLASH.

6.2 Optimize Array for Sparse Statecharts

A more efficient way to store sparse statecharts is to use an array of pointers to arrays of structures that map events to transition functions. See act_obj_opt1.cpp in the demo code. Although each transition uses twice as much memory, we save memory overall on sparse statecharts because we don't store every transition. Each transition table uses one integer array element per state plus two integer array elements to represent each transition. Each state with outgoing transitions requires a sentinel array element to signal the end of the event transition function pairs. Assuming the worst case where each state has at least one outgoing transition, the memory usage for the sparse array optimization is:

MU = 3 X S + 2 X T

MU = Amount of Memory Used

S = Number of States

T = Number of Transitions

The sparse optimization will use less space when:

$T < S \times (E - 3)/2$

This optimization saves space but it takes more time to find the transition function when an event is received. When looking for a transition given a particular state and event, the execution engine will have to search through the list of all the transitions out of a state in the worst case. The maximum number of transitions out of a state is the number of events. This optimization takes E comparisons in the worst case to find the transition function.

6.3 Switch Statement for Optimizing Data Space

Replacing the state transition table with a switch statement uses no memory in the data space, but it does require more code increasing the size of the program segment. See act_obj_opt2.cpp in the demo code. The performance of the switch statement depends upon the efficiency of the code generated by your compiler. In the worst case, the code locates the last transition out of the last state. Assume the worst case where the switch compares the value of the current state to each of the state identifiers and then compares the value of the event identifier to each of the possible event identifiers. This state transition table optimization will take S + E comparisons to find the transition function in the worst case.

7 References for Further Study

Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

Deitel, H. M. and P. J. Deitel. *C++ How to Program*. 2nd edition. Upper Saddle River, NJ: Prentice Hall, 1998.

Douglass, Bruce Powel. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Reading, MA: Addison-Wesley, 1998.

Fowler, Martin. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1997.

Gomez, Martin. "Embedded State Machine Implementation". *Embedded Systems Programming*, December 2000

Rumbaugh, James, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Selic, Brian, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. New York: John Wiley and Sons, Inc., 1994.

Stroustrup, Bjarne. *C++ Programming Language*. 3rd edition. Reading, MA: Addison-Wesley, 1997.