Class 345: Implementing Unified Modeling Language(UML) Statechart Diagrams

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 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 accesses the cell using the index of the current state and the received event. The value in the cell determines the next state of the active object. The execution framework executes the action associated with a 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 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 to for implementing UML Statechart Diagrams associated with UML classes.

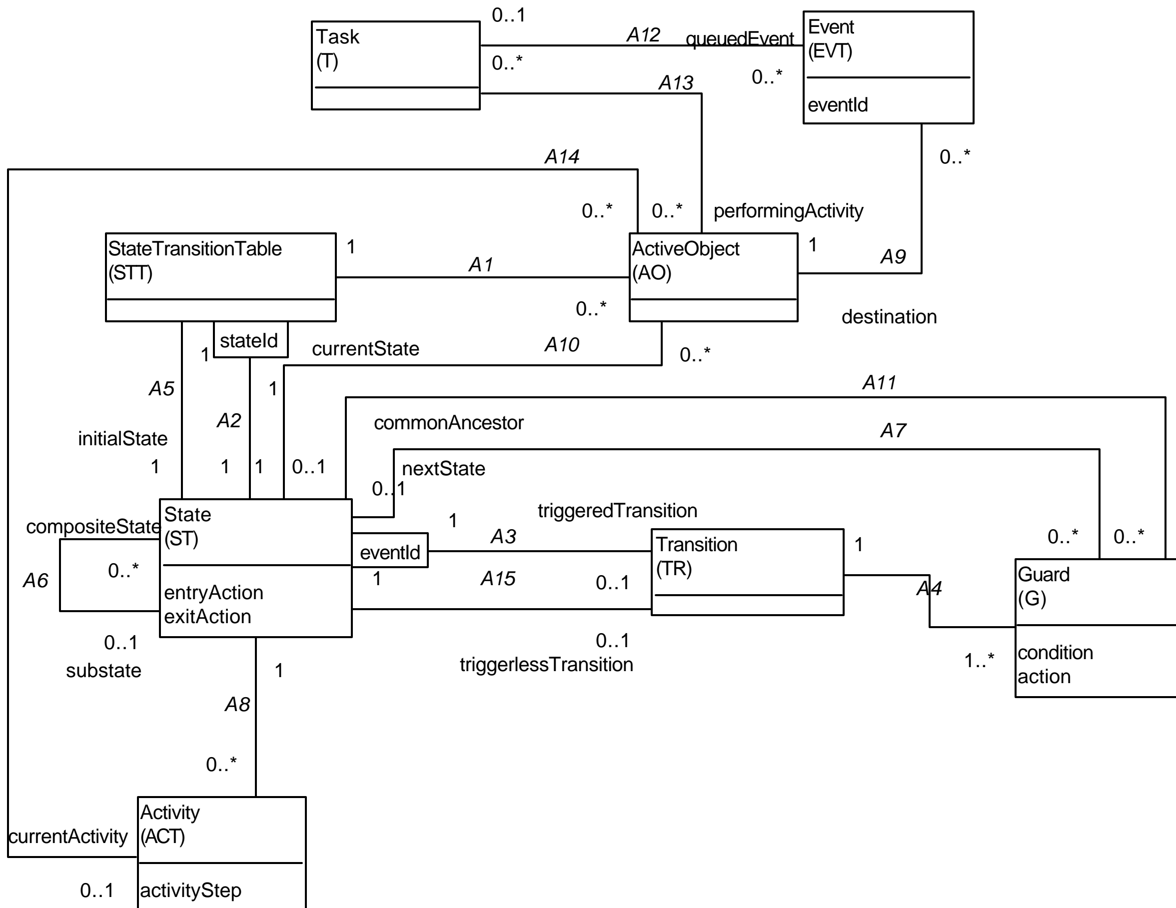

Figure 1: UML Class Diagram of Statechart Mechanisms

2 Statechart Diagram Classes

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 class. The receipt of the event causes the active object to transition to a new state. As a result of the transition the active object may perform a series of actions or 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.

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 causes by the event. Each event carrying parameters is derived from the Event class. The event subclasses hold 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 current event in the queue, 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 class defines the states and transitions supported by all instances of an active object subclass. One instance of the StateTransitionTable specification class is shared by all instances of a particular active object subclass. The ActiveObject stores the current state of the instance and the activity it is currently executing.

The StateTransitionTable consists of a set of States defined by the ActiveObject subclass. The StateTransitionTable also keeps the initial state to enter when a new instance of an ActiveObject is created. Each State has a set of Transitions out of the state triggered by events and a single Transition that is not triggered by any event. In addition each state has an ordered set of activities that are performed when the active object is in this state. A State has an association to the composite state in which it is nested.

A State may have many outgoing Transitions on the same event or many outgoing triggerless Transitions as long as the transitions have different guard conditions. Each guarded transition may go to a different next state. A transition is taken only if its guard condition evaluates to true. The Guard stores the next state as well as the common ancestor state. The common ancestor state is the innermost composite state that contains both the source and destination substates. When taking a transition, the statechart mechanism must execute the exit actions for all states from the current state of the active object up to but not including the common ancestor state. Then it must execute the action on the current transition. Finally it executes the entry actions from the common ancestor state down to the next state defined by the transition. Triggered and untriggered transitions without any guards are represented as a Transition instance associated with a single Guard instance. The condition on the guard instance is a condition that always 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 {
      // make enterState a friend of ActiveObject, don't allow
      // unauthorized clients to set the state
      friend void State::enter(ActiveObject* object,
            const Event* event, const State* common_ancestor) const;
public:
      ActiveObject(State* initial_state);
      void takeEvent( Event*  event);
      bool doActivity();
      const State* currentState() const { return m_currentState;}
protected:
      void performTriggerlessTransitions();
private:
      // hide setState from clients execept ActiveObject
      void setState(const State*    current_state);
      const State*         m_currentState;
      ActivityIterator     m_currentActivity;
};

// reactive class on Class Diagram that has a Statechart Diagram
class MyActiveObject : public ActiveObject {

public:
      enum events {
            EVENT_E1,
            EVENT_E2,
            // … rest of events
            NUM_EVENTS
      };
      // initialize attributes and enter INITIAL_STATE
      MyActiveObject();
      // initialize m_stateTable at startup
      static void initializeSTT();
      // state entry actions
      void enterStateWaiting(const Event* event);
      // state exit actions
      void exitStateWaiting(const Event* event);
      //transition actions
      void transitionFromAToBOnE1(const Event* event);
      // guard conditions
      void checkCondition(const Event* event);
      // activity functions
      void performActivity();
private:
      // identifiers for states
      enum states {
            INITIAL_STATE,
            STATE_A,
            STATE_B,
            // … rest of states
            NUM_STATES
```

```
    };

    static StateTransitionTable m_stateTable;
    // … member variables for each attribute
    // … other member functions

};
```

Each subclass will have a static member variable m_stateTable to store the set 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 member variable is initialized at system startup by the static member function initializeSTT. In addition each ActiveObject subclass has a constructor that enters the initial state defined by the StateTransitionTable.

The ActiveObject subclass defines a number of member functions called by the execution mechanisms when the ActiveObject enters or exits a state, takes a transition, evaluates a guard condition, or performs an activity. Each type of member function has a different signature. The States and Guards store pointers to these member functions. The State and Transition classes invoke the member functions on an ActiveObject instance when a transition is taken, a guard is evaluated or an activity starts.

Entry, exit, and transition action functions take an event parameter and do not return any value. The entry, exit and transition functions have access to the attributes of the instance as well as any parameters of the event that triggered the transition. Since all entry, exit and transition action functions must have the same signature, the type of the event argument to the action function is Event*. If the action function needs to access the parameters associated with the event, the action function will need to do a dynamic_cast or some other form of safe downcast to the event type that is expected in this state or on this transition. The event argument may be NULL if the transition was untriggered so an action on an untriggered transition or a state that has incoming untriggered transitions should not attempt to access event parameters. In MyActiveObject the function enterStateWaiting is an entry action, exitStateWaiting is an exit function, and transitionFromAToBOnE1 is a transition action function.

Guard functions take an event parameter and return a boolean value. The guard function returns true if the transition should be taken and false if the transition should not be taken. The event parameter will be NULL for an untriggered transition. In MyActiveObject the function checkCondition is a guard function.

Activity functions are not triggered by events so they do not have any inputs. Activity functions return a boolean status of true if the activity is complete or false if the activity should continue. When an activity completes, the active object will attempt to take any untriggered transitions out of the current state. In MyActiveObject the function performActivity is an activity function.

3.1 Initializing the StateTransitionTable

The four steps to initializing an active object's StateTransitionTable specification are:

- define state and event identifiers
- add the states
- set the initial state
- add the transitions

3.1.1 Defining the State and Event Identifiers

The ActiveObject subclass must define an enumerated type to identify all of the states in the Statechart Diagram and another enumerated type to identify all the events received by the active object. The enumerated values will be used to identify states and events when intializing the StateTransitionTable. 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 in the initializeSTT function. Start the state identifier enumerated

type with the name INITIAL_STATE. This will represent the initial state on the Statechart Diagram. The initial state will become the current state of the active instance when it is created. Add one enumerated value for each 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.

Pass the NUM_STATES and NUM_EVENTS values to the constructor for the StateTransitionTable when defining the m_stateTable member as follows:

```
StateTransitionTable
      MyActiveObject::m_stateTable(NUM_STATES,NUM_EVENTS);
```

3.1.2 Adding the States

The initializeSTT function of MyActiveObject initializes the state transition specification information for the class. The initialize function adds all the states and transitions and sets the initial state. The StateTransitionTable class declaration follows:

```
class State;
typedef int StateId;
typedef void (ActiveObject::*ActionFunction)(const Event* event);
typedef bool (ActiveObject::*ActivityFunction)();

extern const StateId NULL_STATE;

class StateTransitionTable
{
public:
      StateTransitionTable(int num_states, int num_events);
      ~StateTransitionTable();
      void addState(StateId id, StateId parent_composite,
            ActionFunction entry_action = 0,
            ActionFunction exit_action = 0);
      // add an untriggered transition
      void addTransition(StateId source_state, StateId common_id,
            StateId next_state, ActionFunction action = 0,
            GuardFunction condition = 0);
      // add a triggered transition
      void addTransition(EventId id, StateId source_state,
            StateId common_id, StateId next_state,
            ActionFunction action = 0, GuardFunction condition = 0);
      void addActivity(StateId source_state,
            ActivityFunction activity);
      void setInitialState(StateId initial_state)
            { m_initialState = m_states[initial_state]; }
      State* initialState() const
            { return m_initialState;}
private:
      State* lookupState(StateId id) const;
      State*                m_initialState;
      std::vector<State*>   m_states;
      int                   m_numEvents;
```

```
	int			m_lastState;
};
```

The constructor creates a vector of the appropriate size to accommodate the number of states, sets the initial state to NULL and stores the number of events. The number of events will be used to initialize the triggered transition vectors for each state.

```
StateTransitionTable::StateTransitionTable(int num_states,
                                           int num_events):
   m_states(num_states), m_initialState(0), m_numEvents(num_events)
{
}
```

To add a state to the state transition diagram, call the addState function specifying the identifier of the state to be created, the identifier of the parent composite state, a pointer to the entry member function, and a pointer to the exit member function. Set the entry or exit action to 0 if the state does not have an entry or exit action. If the state does not have a parent composite state, specify the constant NULL_STATE as the parent composite state value.

Nested states must know their parent composite state upon initialization. Therefore parent composite states must be added to the StateTransitionTable before adding their nested states are added. The addState function locates the parent composite state in the state vector and attaches it to a new state. The new state is entered into the state vector using the state identifier as an index:

```
void StateTransitionTable::addState(StateId id, StateId composite,
                                    ActionFunction entry_action,
                                    ActionFunction exit_action)
{
	m_states[id] = new State(m_numEvents,
		lookupState(superstate), entry_action, exit_action);
}
```

The addState function uses the lookupState function to find the pointer to the state with a given identifier. If the state identifier is the special state NULL_STATE, lookupState returns NULL.

```
State* StateTransitionTable::lookupState(StateId id) const
{
	if (id == NULL_STATE)
		return 0;
	else
		return m_states[id];
}
```

After adding a state, call addActivity to attach activity functions to be performed while in the state. The addActivity function locates the state performing the activity using the source_id as an index to the states vector. Once the State is located, addActivity adds the activity function to the list of activities associated with the state.

```
typedef bool (ActiveObject::*ActivityFunction)();

void StateTransitionTable::addActivity(StateId source_id,
                                       ActivityFunction activity)
{
	m_states[source_id]->addActivity(activity);
}
```

3.1.3 Setting the Initial State

After the initial state has been added to the STT, call the setInitialState function with the identifier of the initial state. The setInitialState function locates a pointer to the initial state and sets the m_initialState member variable:

```
void setInitialState(StateId initial_state)
{
      m_initialState = m_states[initial_state];
}
```

3.1.4 Adding the Transitions

Add the triggered and untriggered transitions out of each state. The set of transitions out of a state includes the transitions specifically out of this state plus the transitions out of all of its parent composite states. The addTransition function is overloaded. The signature accepting an event identifier supports adding triggered transitions and the other untriggered transitions:

```
// add a triggered transition
void StateTransitionTable::addTransition(EventId id,
                                         StateId source_id,
                                         StateId common_id,
                                         StateId next_id,
                                         ActionFunction action,
                                         GuardFunction condition)
{
      State*     source_state = lookupState(source_id);
      State*     common_state = lookupState(common_id);
      State*     next_state = lookupState(next_id);
      source_state->addTransition(id, common_state, next_state,
                                  action, condition);
}

// add a triggered transition to a state
void State::addTransition(EventId id, State* common_ancestor,
                          State* next_state, ActionFunction action,
                          GuardFunction condition)
{
      if (!m_triggeredTrans[id])
      {
            m_triggeredTrans[id] = new Transition;
      }
      m_triggeredTrans[id]->addGuard(m_triggeredTrans[id],
                                     common_ancestor, next_state,
                                     action, condition);
}

// add an untriggered transition
void StateTransitionTable::addTransition(StateId source_id,
                                         StateId common_id,
                                         StateId next_id,
                                         ActionFunction action,
                                         GuardFunction condition)
{
      State*     source_state = lookupState(source_id);
      State*     common_state = lookupState(common_id);
```

```
	State*     next_state = lookupState(next_id);
	source_state->addTransition(common_state, next_state, action,
	                            condition);
}

// add an untriggered transition to a state
void State::addTransition(State* common_ancestor, State* next_state,
            ActionFunction action, GuardFunction condition)
{
	if (!m_untriggeredTrans)
	{
		m_untriggeredTrans = new Transition;
	}
	m_untriggeredTrans->addGuard(m_untriggeredTrans, common_ancestor,
	                             next_state, action, condition);
}
```

The addTransition function locates the source, destination and common ancestor states for the transition and calls State::addTransition for the source state. The addTransition function locates the appropriate transition. If adding a triggered transition, addTransition checks the value of the transition vector at the index of the event identifier. If adding an untriggered transition, addTransition checks the value of the m_untriggeredTrans member variable. If a transition does not already exist, a new one is created. AddTransition adds the specified guard condition and action to the transition. If a transition is unguarded, set the condition parameter should be NULL.

Set the next state of the transition to the innermost state that the transition enters. For example, if a transition goes from a state to a composite state, set the next state of the transition to the initial state of the composite rather than the composite itself. Transitions on an event out of a nested state override transitions on the same event out of the composite state.

Find the common ancestor by determining the innermost composite state that is a parent of both the source and the next state. If there is no common ancestor between the two states, pass NULL_STATE as the common ancestor parameter. If a state has a transition to self, specify the parent composite state as the common ancestor or NULL_STATE if the state is not nested. Note that the State class could calculate the common ancestor at run time using the parent composite state member, but for efficiency purposes, we pre-calculate the common ancestor for each transition.

Internal transitions and deferred events are special cases of transitions. An internal transition executes the action on the transition without changing state. For an internal transition, specify the state itself as the common ancestor. Since the transition algorithm executes any entry and exit actions up to the common ancestor, setting the state to itself as the common ancestor will bypass any entry and exit actions of the state.

Deferred events are created with addTransition. The action function is 0 and the common ancestor and next states are the NULL_STATE. When the transition algorithm sees a guard with no next state specified, it defers the event by placing the event back onto the event queue.

Figure 2 contains a sample State model followed by an example of how to initialize the sample state model:

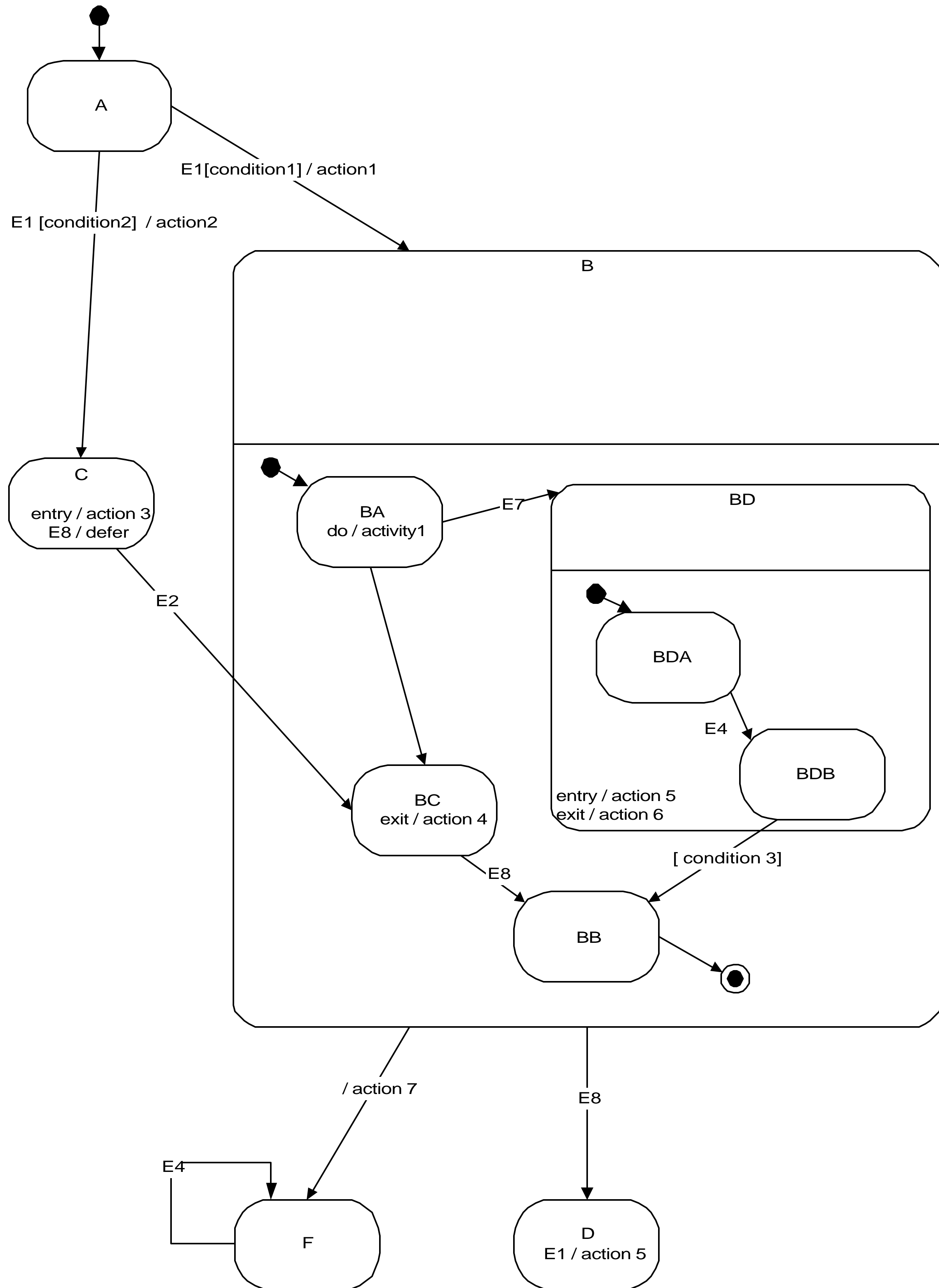

Figure 2 : Sample Statechart Diagram for MyActiveObject

```
class MyActiveObject1 : public ActiveObject
{
public:
      enum events{
            EVENT_E1,
            EVENT_E2,
            EVENT_E7,
            EVENT_E8,
            EVENT_E4,
            NUM_EVENTS
      };
      MyActiveObject1();

      // guard functions
      bool condition1(const Event*);
      bool condition2(const Event*);
      bool condition3(const Event*);

      //action functions
      void action1(const Event*);
      void action2(const Event*);
      void action3(const Event*);
      void action4(const Event*);
      void action5(const Event*);
      void action6(const Event*);
      void action7(const Event*);

      // activity functions
      bool activity1();
      static void initializeSTT();
private:
      enum states {
            INITIAL_STATE,
            STATE_A,
            STATE_B,
            STATE_BA,
            STATE_BB,
            STATE_BC,
            STATE_BD,
            STATE_BDA,
            STATE_BDB,
            STATE_C,
            STATE_D,
            STATE_F,
            NUM_STATES
      };
      static StateTransitionTable m_stateTable;
};

void MyActiveObject1::initializeSTT()
{
      // add the states - state id, parent, entry action, exit action
      m_stateTable.addState(INITIAL_STATE, NULL_STATE);
      m_stateTable.addState(STATE_A, NULL_STATE);
      m_stateTable.addState(STATE_B, STATE_B);
      m_stateTable.addState(STATE_BA, STATE_B);
      // add activity to state BA
```

```
m_stateTable.addActivity(STATE_BA,
      (ActivityFunction)&MyActiveObject1::activity1);
m_stateTable.addState(STATE_BB, STATE_B, 0,
      (ActionFunction)&MyActiveObject1::action4);
m_stateTable.addState(STATE_BC, STATE_B);
m_stateTable.addState(STATE_BD, STATE_B,
      (ActionFunction)&MyActiveObject1::action5,
      (ActionFunction)&MyActiveObject1::action6);
m_stateTable.addState(STATE_BDA, STATE_BD);
m_stateTable.addState(STATE_BDB, STATE_BD);
m_stateTable.addState(STATE_C, NULL_STATE,
      (ActionFunction)&MyActiveObject1::action3, 0);
m_stateTable.addState(STATE_D, NULL_STATE);
m_stateTable.addState(STATE_F, NULL_STATE);

// set intial state
m_stateTable.setInitialState(INITIAL_STATE);

// add transitions
// guarded transition into a composite state
m_stateTable.addTransition(EVENT_E1, STATE_A, NULL_STATE,
      STATE_BA, (ActionFunction)&MyActiveObject1::action1,
      (GuardFunction)&MyActiveObject1::condition1);
m_stateTable.addTransition(EVENT_E1, STATE_A, NULL_STATE,
      STATE_C, (ActionFunction)&MyActiveObject1::action2,
      (GuardFunction)&MyActiveObject1::condition2);
// unguarded transition
m_stateTable.addTransition(EVENT_E2, STATE_C, NULL_STATE,
      STATE_BC, 0, 0);
// defer event E8 in state C
m_stateTable.addTransition(EVENT_E8, STATE_C, NULL_STATE,
      NULL_STATE);
m_stateTable.addTransition(EVENT_E7, STATE_BA, STATE_B,
      STATE_BDA);
// add triggerless transition from BA to BC
m_stateTable.addTransition(STATE_BA, STATE_B, STATE_BC);
m_stateTable.addTransition(EVENT_E4, STATE_BDA, STATE_BD,
      STATE_BDB);
// add guarded triggerless transition from BDB to BB
m_stateTable.addTransition(STATE_BDB, STATE_B, STATE_BB, 0,
      (GuardFunction)&MyActiveObject1::condition3);
// E8 out of state BC overrides the E8 out of the composite state
m_stateTable.addTransition(EVENT_E8, STATE_BC, STATE_B,
      STATE_BB);
// add a triggerless transition out of BB to F since BB
// is a final state
m_stateTable.addTransition(STATE_BB, NULL_STATE, STATE_F,
      (ActionFunction)&MyActiveObject1::action7);

// propagate E8 to all other states nested in B
m_stateTable.addTransition(EVENT_E8, STATE_BA, NULL_STATE,
      STATE_D);
m_stateTable.addTransition(EVENT_E8, STATE_BDA, NULL_STATE,
      STATE_D);
m_stateTable.addTransition(EVENT_E8, STATE_BDB, NULL_STATE,
      STATE_D);
m_stateTable.addTransition(EVENT_E8, STATE_BB, NULL_STATE,
```

```
            STATE_D);
    // take an internal transition on E1 in state D
    m_stateTable.addTransition(EVENT_E1, STATE_D, STATE_D, STATE_D,
            (ActionFunction)&MyActiveObject1::action5);
    // take a self transition on E4 in state F
    m_stateTable.addTransition(EVENT_E4, STATE_F, NULL_STATE,
            STATE_F);
}
```

4 Event and Activity Scenarios

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;
};
```

To 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 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)
    {
          // if there are events in the queue to process
          if (m_eventQueue.size() != 0)
          {
                // dispatch event to destination active object
                next_event = m_eventQueue.front();
                m_eventQueue.pop_front();
                next_event->dispatch();
          }
          if (m_activityQueue.size() > 0 && !m_stopProcessingEvents)
          {
                // do the current activity
                ActiveObject*       object = *m_currentActiveObject;
                if (object->doActivity())
                {
                      if (object == *m_currentActiveObject)
                      {
                            // this is the last step in the activity
                            // 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();
                            }
                      }
                }
          }
    }
}
```

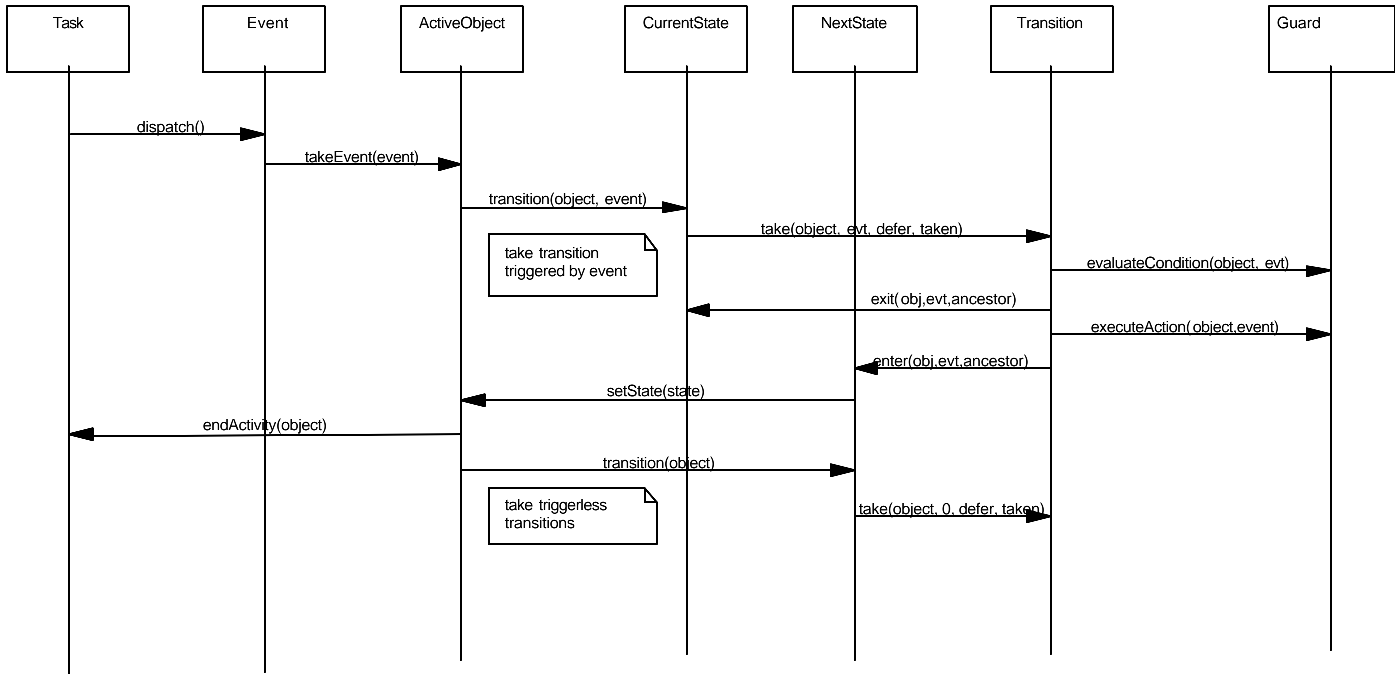

Figure 3: Message Sequence Chart for Dispatch Event Scenario

4.1 Dispatching an Event

Figure 3 shows a Message Sequence for an event dispatch. The scenario is a transition out state CurrentState to next state NextState. NextState has a triggerless transition out of it and does not have any activities. The Task removes the next event from the queue and calls the dispatch function on it. Dispatch calls the takeEvent function on the destination active object:

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

The take event function looks for a triggered transition out of the current state for the event received. If the event triggers a transition and the current state does not have activities, the active object performs any triggerless transitions out of the next state:

```
void ActiveObject::takeEvent(Event* event)
{
      // take a triggered transition
      bool    transition_taken =
                  m_currentState->transition(this, event);

      // keep taking triggerless transitions out of current state if
      // there are no activities to perform
      if (transition_taken)
```

```
	{
		performTriggerlessTransitions();
	}
}
```

The transition function of the State locates the appropriate transition for this event identifier in the m_triggeredTrans vector. If there is no transition out of the state defined for this event, the transition function ignores the event. If the event is ignored, the transition function returns false so that no triggerless transitions will be taken. If a transition is defined for this event, the transition function calls take on the transition associated with this event. The transition function returns a boolean indicating if the event was deferred and another boolean indicating if any of the guard conditions evaluated to true causing a transition. If the event was not deferred, the transition function cleans up the event. If the event is deferred, the event was placed back on the event queue so transition should not delete it.

```
bool State::transition(ActiveObject* object, Event* event) const
{
	bool transition_taken = false;
	bool event_deferred = false;
	if (m_triggeredTrans[event->eventId()])
	{
		m_triggeredTrans[event->eventId()]->take(object,
				event, event_deferred, transition_taken);
	}
	if (!event_deferred)
	{
		delete event;
	}
	return transition_taken;
}
```

The take function finds the first guard condition that evaluates to true. If a guard condition evaluates to true, the take function checks to see if the guard has a next state. If the guard does not define a next state, take defers the event by placing the event back on the event queue. If the event is not deferred, the take function exits the current state, executes the action on the transition, and enters the next state.

```
void Transition::take(ActiveObject* object, Event* event,
			bool& event_deferred,
			bool& transition_taken) const
{
	event_deferred = false;
	transition_taken = false;
	Guard*	true_guard = findFirstTrueGuard(object, event);
	if (true_guard)
	{
		if (true_guard->nextState())
		{
			object->currentState()->exit(object, event,
				true_guard->commonAncestor());
			true_guard->executeAction(object, event);
			true_guard->nextState()->enter(object, event,
				true_guard->commonAncestor());
			transition_taken = true;
		}
		else
		{
			// event is deferred, place it back on the queue
```

```
                        event_deferred = true;
                        Task::singleTask()->enqueueEvent(event);
                }
        }

}
```

The findFirstTrueGuard function iterates through the vector of Guards for this Transition and evaluates guard conditions returning when a condition evaluates to true.

```
typedef vector<Guard*>::const_iterator  GuardIter;

Guard* Transition::findFirstTrueGuard(ActiveObject* object,
                                      const Event* event) const
{
        GuardIter       g_iter;
        for(g_iter = m_guards.begin(); g_iter != m_guards.end();
            g_iter++)
        {
                if ((*g_iter)->evaluateCondition(object, event))
                {
                        return *g_iter;
                }
        }
        return 0;
}
```

The evaluateCondition function of the Guard determines if the guard has a condition associated with it. If there is no condition, the guard automatically evaluates to true. A guard has no condition function if the transition is unconditional. If the guard has a condition function, evaluateCondition executes the evaluation function pointer and returns the boolean result of the condition function.

```
bool Guard::evaluateCondition(ActiveObject* object,
                              const Event* event) const
{
        if (m_condition)
        {
                return (object->*m_condition)(event);
        }
        return true;
}
```

When a guard condition evaluates to true, the transition executes the exit actions for the current state and for all parent composite states up to the common ancestor state. The exit function determines which exit functions to execute and the correct order by starting at the current state and navigating up the state hierarchy using the m_parent member variable until it reaches the common ancestor. The common ancestor may be a composite state or it may be the NULL_STATE if the two states do not have a common ancestor. Note that when a transition is an internal transition, the common ancestor is the same as the current state so the exit condition exiting_state != common_ancestor evaluates to true the first time it is tested. Thus no exit actions are executed. In a self transition, the common ancestor is the parent of the state so only the exit action for the current state is called.

```
void State::exit(ActiveObject* object, const Event* event,
                 const State* common_ancestor) const
{
```

```
    const State*     exiting_state = this;
    while(exiting_state != common_ancestor)
    {
          if (exiting_state->m_exitAction)
          {
              (object->*(exiting_state->m_exitAction))(event);
          }
          exiting_state = exiting_state->m_parent;
    }
}
```

After exiting the previous state, the transition executes the action for the true guard by calling executeAction. ExecuteAction calls the action member function pointer associated with this transition.

```
void Guard::executeAction(ActiveObject* object,
                          const Event* event) const
{
    if (m_action)
    {
          (object->*m_action)(event);
    }
}
```

The active object is now ready to enter the next state. When entering the next state, the active object must execute all the state entry actions hierarchically starting at the common ancestor and ending with the next state. The enter function traverses the nested states from the next state up to the common ancestor storing the path in a vector. Iterating the vector in the reverse direction gives the correct order of entry action execution from common ancestor to next state. After the entry actions have been executed, enter sets the current state of the active object. Although setState is a private member function of ActiveObject, enterState is allowed to call setState because enterState is a friend of ActiveObject. We restrict access to the setState function because the state should only be set by the execution framework classes and not by other unauthorized clients.

```
typedef vector<const State*>::reverse_iterator ReverseStateIter;

void State::enter(ActiveObject* object, const Event* event,
                  const State* common_ancestor) const
{
    // determine the order of the entry actions
    vector<const State*>     enter_path;
    const State*             entering_state = this;
    while(entering_state != common_ancestor)
    {
          if (entering_state->m_entryAction)
          {
                enter_path.push_back(entering_state);
          }
          entering_state = entering_state->m_parent;
    }
    // execute the entry actions
    ReverseStateIter   rstate_iter;
    for(rstate_iter = enter_path.rbegin();
        rstate_iter != enter_path.rend(); rstate_iter++)
    {
          (object->*((*rstate_iter)->m_entryAction))(event);
    }
```

```
	object->setState(this);
}
```

The setState function sets the current state to the newly entered state. If the state has activities, the active object sets the current activity iterator to the first activity and notifies the task that it has activities. If the state does not have any activities, the state notifies the task to end any activities associated with the active object.

```
void ActiveObject::setState(const State* current_state)
{
	m_currentState = current_state;
	// set activity to the first activity in the list
	if (m_currentState->hasActivities())
	{
		// set up to execute the first activity in the state
		m_currentActivity = m_currentState->firstActivity();
		// add to activity list
		Task::singleTask()->beginActivity(this);
	}
	else
	{
		Task::singleTask()->endActivity(this);
	}
}
```

The beginActivity function adds the active class to the end of the activity queue. If this is the first activity, 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 activity, set the current active object
	if (m_activityQueue.size() == 1)
	{
		m_currentActiveObject = m_activityQueue.begin();
	}
}
```

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 affecting 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);
        }
    }
}
```

After taking a triggered transition, if the new state of the active object is not performing any activities, the active object attempts to take a tiggerless transition out of the new state. PerformTriggerlessTransitions will continue to process available triggerless transitions while the current state does not have activities. If the current state has activities, triggerless transitions cannot be taken until the activity completes.

```
void ActiveObject::performTriggerlessTransitions()
{
    while (!m_currentState->hasActivities() &&
            m_currentState->transition(this))
        ;
}
```

The transition function that handles triggerless transitions determines if the state has any triggerless transitions. If the state does not have any triggerless transitions, the transition function returns false. If the state does have triggerless transitions, the transition function calls the take function with a NULL event pointer. If any of the guard conditions evaluate to true and an untriggered transition is taken, the transition function returns true.

```
bool State::transition(ActiveObject* object) const
{
    // see if any of the guard conditions on any of the triggerless
    // transitions evaluates to true
    bool transition_taken = false;
    bool event_deferred = false;
    if (m_untriggeredTrans)
    {
        m_untriggeredTrans->take(object, 0, event_deferred,
                                 transition_taken);
    }
    return transition_taken;
```

```
}
```

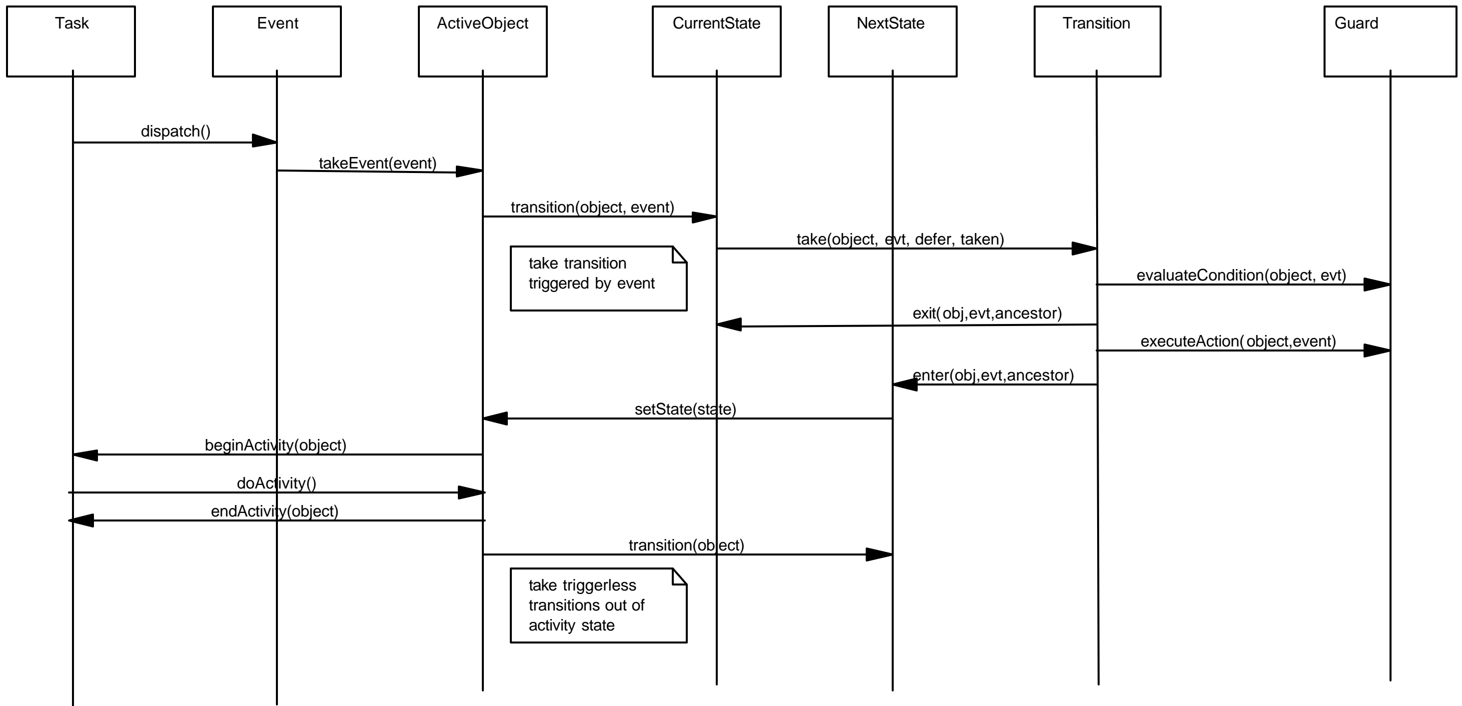

Figure 4: Message Sequence Chart for Activity Scenario

4.1.1 Performing Activities

Figure 4 shows a Message Sequence for an event dispatch. After dispatching an event, the task checks the activity queue. If there is an active object in the activity queue, the task calls the doActivity function on the active object:

```
bool ActiveObject::doActivity()
{
      // if the activity is complete, perform any triggerless
      // transitions out of the state and remove the activity
      // from the task
      ActivityFunction   activity_fun = *(m_currentActivity);
      bool activity_done = (this->*activity_fun)();
      bool last_act_for_obj = false;

      if (activity_done)
      {
            Task::singleTask()->endActivity(this);
            performTriggerlessTransitions();
      }
      else
      {
            m_currentActivity++;
            if (m_currentActivity == m_currentState->lastActivity())
            {
```

```
                m_currentActivity = m_currentState->firstActivity();
                last_act_for_obj = true;
            }
        }
        return last_act_for_obj;
}
```

The doActivity function executes the current activity for the active object. If the activity function returns true indicating that the activity is complete, the active object removes itself from the activity queue by calling Task::endActivity. After ending the activity, the active object performs any triggerless transitions out of the current state. If the activity is not complete, doActivity moves the current activity iterator to the next activity for the current state. If the iterator is at the end of the activities for this state, doActivity wraps to the first activity for the state and returns true. Returning true from doActivity will cause the Task to advance its current active object iterator to the next object in the activity queue. This allows all objects performing activities to get equal time.

5 System Startup

5.1 Constructing Active Objects

The constructor for an active object subclass locates the initial state in the StateTransitionTable for the subclass and calls the constructor for ActiveObject. The constructor for ActiveObject sets the current state pointer. Once the data for the active object subclass is initialized, the constructor calls performTriggerlessTransitions to transition the class out of the initial state. According to the rules of construction in C++, base class constructors are called before calling the body of the constructors for any derived classes. PerformTriggerlessTransitions is called in the constructor for MyActiveObject rather than in the constructor for ActiveObject because any actions executed as a result of the transitions will expect the member variables to be fully initialized. When the ActiveObject constructor is executed, the members of MyActiveObject are uninitialized.

```
MyActiveObject::MyActiveObject() :
            ActiveObject(m_stateTable.initialState()),
            m_activityCounter(0)
            // initialize other member variables
{
      // insert any subclass specific initialization here
      // transition out of the initial state
      performTriggerlessTransitions();
}

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

5.2 The Main Function

The main function initializes all StateTransitionTables, 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()
{
	// initialize state machines
	MyActiveObject::initializeSTT();

	// 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 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.

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.