STUDIA UNIV. "BABEŞ–BOLYAI", INFORMATICA, Volume I, Number 1, March 1996

AN ACTIVE OBJECT MODEL FOR MULTIMEDIA PRESENTATIONS

M. PAPATHOMAS AND V.-M. SCUTURICI

Abstract. In this paper we propose an active object model that aims at a more comprehensive approach in integrating concurrent programming and object-oriented features. The model incorporates a number of previously proposed features with the novel features of abstract states, state predicates and state notification. We have started using the prototype for the development of multimedia programming environment based on active objects. A prototype of the model has been implemented in Python and the presentation in this paper is based on this implementation.

1. Introduction

Substantial research activity in the past few years concentrated on the design of languages and models for integrating concurrency and object-oriented features with the intention to enhance the potential for software reuse in the development of concurrent systems. Most of the work in the area has focused on the problem of combining inheritance with concurrency and more particularly specifying and reusing through inheritance synchronization constraints on the invocation of objects' methods. Currently, this is a widely recognised problem and several solutions have been proposed. However, most proposed solutions address this problem in isolation. More recent and less research has addressed the issue of coordinating the execution of a set of objects and of specifying and reusing coordination patterns separately from objects. Furthermore, few languages supporting the proposed features are widely available and relatively little experience has been gained from their use.

In this paper we propose an active object model that aims at a more comprehensive approach in integrating concurrent programming and object-oriented features. The model incorporates a number of previously proposed features with the novel features of *abstract*

Received by the editors: September 12, 1996.

1991 *Mathematics Subject Classification.* 68Q10, 68Q90.

1991 *CR Categories and Descriptors.* D.1.5 [Programming Techniques]: Object-oriented Programming; D.1.3 [Programming Techniques]: Concurrent Programming; D.2.6 [Software Engeneering]: Programming Environments; D.3.2 [Programming Languages]: Language Classifications – object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features – Concurrent programming structures; H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems.

states, *state predicates* and *state notification*. The model has been designed for addressing simultaneously the following reuse aspects in concurrent object-oriented programming:

- Support for self-contained objects that can be reused across applications.
- Reuse of methods through class inheritance.
- Reuse of synchronization constraints.
- Composition of active object behaviours.
- Object coordination.

A prototype of the model has been implemented in Python and the presentation in this paper is based on this implementation. Python is freely available on a large number of systems. This will allows us to make our prototype widely available and gain more experience with the use of its concurrent object-oriented features in the development of applications.

2. The Object Model

Objects are active entities that resemble server processes that accept requests from other object, they can delay requests and process them in an order that is most suitable to them. Requests are processed by threads that execute quasi-concurrently within an object. Threads may also be created spontaneously at the creation of an object.

The main aim in the design of this object model is to enhance the potential for reuse in concurrent object-oriented systems by integrating support for all the reuse issues discussed in section 2. For doing so, the model incorporates a number of message passing features that are combined with thread scheduling in such a way that allows objects to schedule the processing of requests and replies in a self-contained way. These features are integrated with the novel concepts of abstract states, state predicates and state notification. The use of these features avoids the problems caused by the use of inheritance in COOPLs, supports new ways to specify and inherit active object behaviours and also provides support for object coordination.

The model is general enough so that it may be incorporated in a variety of languages. The current version is implemented as an extension to the language Python. The linguistic constructs and examples used in this paper are based on the Python implementation.

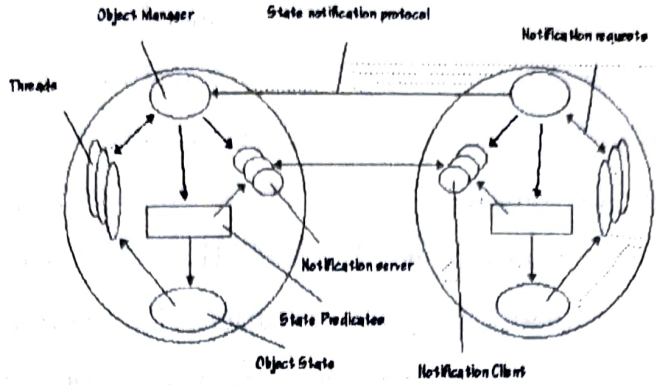

FIGURE 1. Conceptual view of the object execution model

2.1. **Language-Independent Description.** Figure 1 shows a conceptual view of object execution in our model. Each object is associated with an object manager that controls the actions that are executed by its object. The object manager is a conceptual entity that makes it easier to describe the behaviour of objects in our model.

The object manager instructs its object to carry out a number of actions such as to start or resume a thread that executes some methods of the object. The object executes the code of a method until the occurrence of an event, such as the completion or suspension of a method. The occurrence of such an event activates the object manager which decides what should be the next action to be executed by its object. The object can thus be either executing some thread or it may be waiting for the object manager to instruct it what it should to do next. In the latter case, we will say that the object is at a stable state.

The execution of an object can be seen as a graph where nodes represent stable states and edges are associated with tuples of the form (a,e) where a is the action that the object was instructed to execute in the previous stable state and e is the event that stopped the execution of a and made the object to move into the next stable state.

Apart from events that are generated from the execution of its object, the object manager may also be activated by external events that are generated by other object in program. The events that trigger the execution of the object manager are:

- *A method invocation request is received at the object.* In this case the object manager creates a new thread for running the method. If there is no thread already active within the object and if the object is at a state where it can execute the request method, the object manager runs the newly created thread.

- *A thread completes the execution of its method.* In this case the object manager chooses another waiting thread, if any, for execution.
- *A thread requests to be suspended until the occurrence of an event.* In this case the object manager removes the thread from the queue of threads that are ready to run and inserts it in a queue of threads that are waiting for an event. Then, it picks another thread, if any, for execution. There are several types of events a thread may wait for. For instance, it may wait for the reply to a request it has issued to another object or it may wait until the object reaches a certain state. These events will be discussed later in conjunction with the constructs generate them.
- *The occurrence of an external event of interest to one of its threads.* This any event awaited by some of the object's threads that is not generated by actions that are executed within the object. This can be for instance: the arrival of the reply to a request a thread has made to another object or the arrival of a state notification event (to be discussed later). In this case, the object manager removes the thread from the list of waiting threads and depending on whether or not the thread can be run immediately it resumes the execution of the thread.
- *The receipt of a state notification request.* This is a request by another object manager asking the object manager to generate a notification event when its object reaches a certain state. The object manager stores the request in a queue of notifications requests and when its object reaches the request state sends a notification event. State notification is a novel feature of our model that will be discussed in detail later.

In many of the situation discussed above, the object manager needs information about the object's state. In our model, however, the object manager does not see or access the object state directly. The concept of *abstract state* is used to represent properties of interest with respect to the object's state at a level of abstraction that hides implementation details. *State predicates* provide the mapping from abstract states to the concrete object state. In order to find out whether the object is at an abstract state, the object manager goes through a state predicate. Abstract states and state predicates are discussed in detail next. Figure 1 shows a conceptual view of object execution in our model illustrating the interactions of the object components of model.

2.1.1. *Abstract States and State Predicates.* An abstract state represents some aspect of the real state of execution of an object at a level of abstraction that hides implementation details. The state of execution is taken here in a broad sense. It may comprise not only the values of the object's instance variables but also the messages that are suspended at the object interface, the state of execution of the object's threads, etc.

At each stable state in the object's execution graph, the object's manager decides what action will be executed next based on abstract states. In our model abstract states are used to constrains the execution of the possible actions. This is accomplished by allowing the programmer to constrain the acceptance of methods and to express the suspension or resumption of threads using abstract states. The linguistic constructs that are provided for doing so are discussed in the remaining of the document.

State predicates provide an interpretation for abstract states. They are used to tell whether or not a property that is represented by an abstract state holds a concrete state. The mapping from abstract states to concrete states provided by state predicates may differ for different classes and thus supports polymorphism for abstract states.

A state predicate can be defined as a function P from CS X Asp to {*true, false*}, where CS is the set of concrete object states and Asp is a subset of the set of the object's abstract states AS. The following are some important properties of state predicates:

- A set of abstract states may be true at one object state.
- An object may be associated with a set of state predicates and each predicate is associated to a subset of the object's abstract states.
- The subsets of abstract states associated with different state predicates of an object are disjoint.

From a practical point of view, state predicates are objects that encapsulate the information necessary to determine whether or not the property described by the abstract state is true at a certain stable state in the execution of the object. They are defined by the programmer and used by the object manager to evaluate the conditions that constraint the execution of the object's actions.

2.1.2. *State Notification.* State notification allows active objects to monitor and synchronize with state changes, expressed in terms of abstract states, occurring in other objects. State notification is a protocol provided by object managers that allows a thread in one

object to synchronize it's execution with state changes expressed in terms of abstract states of another object.

Figure 1 shows a conceptual view of the architecture used to support state notification. The figure shows two active objects A and B. B's object manager has made a state notification request to A's object manager. Following this, A's object manager has created a local notification server object that represents the notification request. The object manager maintains a list of notification server objects (notification requests); each time the object state changes it goes through the list and activates each notification server. The notification server evaluates its associated abstract state expression by invoking the appropriate state predicate(s) and informs the notification client if appropriate. When the notification client is informed, it requests its object manager to take the appropriate action; for instance, schedule a suspended method for execution.

State notification may be *asynchronous* or *synchronous*. In the asynchronous case, when the object is at stable state, its object manager services all notification requests and then proceeds with the normal execution of the object's methods. The synchronous variant allows the object that requested the notification and the notifying objects to be synchronized in a way similar to "rendez-vous". This variant guarantees, by postponing the execution of the notifying object's methods, that the notified object will get the chance to invoke methods of the notifying object while it is still at the requested state.

Together with abstract states, state notification may be used to describe abstractly, as a sequence of abstract state changes, practically any activity encapsulated within an active object. Also, note that this is a general architecture and can thus be instantiated with different implementations of notification clients and servers. For example, support for real-time notification can be provided by specifying a time-out at the notification client and/or by requesting a bound on the notification delay.

2.2. **Linguistic Support.**

2.2.1. *Abstract States.* In the implementation of our object model in Python, abstract states are represented as tuples that contain at least one element. This first mandatory element is a string that represents the name of an abstract state. For example, ('empty',) and ('full',) are used to represent the abstract states of the bounded buffer. Apart from the first mandatory string element that represents the state name, a tuple that represents an abstract state may contain an arbitrary number of additional elements. In this case

the abstract state name may be used to represented a family of abstract states that are further qualified by the additional elements. The semantics of these additional elements is defined by the state predicate responsible for the abstract state. For instance, a set of abstract states ('contains',n), where n is a natural number, may be used to represent the states where a container object contains exactly n elements. In the buffer example it is the functions empty and full that are implicitly defined as state predicates for the states with the same name. Later we will see other ways for defining state predicates explicitly.

2.2.2. *State Predicates.* State predicates are objects that are associated with an object and are responsible for determining whether or not the associated object is at a given abstract state. An object may have a set of state predicates and each state predicate is associated with a disjoint subset of the set of abstract states defined for the object. When the object manager needs to determine if an object is at an abstract state, it calls the state predicate that is responsible for that particular abstract state.

If no state predicate is specified explicitly for an abstract state the object should have a method with the same name as the state that is used to determine whether or not the property associated with is abstract state is true.

State predicates may be associated with an object statically or dynamically. It is also possible for a state predicate to be shared among several objects. This feature can be used as it will be shown in section 4.4.1 to support object coordination.

In a class definition, the variable state_predicates is used to associate abstract states to state predicates. This variable should be set to a dictionary, entries of this dictionary should have as key a state predicate class and as value a list abstract states. For abstract states that are not associated explicitly with a state predicate in the state_predicates variable, a method with the same name as the abstract state should be provided for allowing to determine whether or not the object is at the associated state.

The method newPred supported by all active objects may be used to associate a state predicate to an object instance at run-time. This method takes as arguments a state predicate object and a list of the associated abstract states. The object should not define these any of these states already.

In order to be able to operate with the object manager, state predicates should define a method evalState that takes as argument a tuple that represents an abstract state and the object for which it is needed to find out if it is at the specified state. This

method should return None to indicate that the object is not at the requested state or a value different to None otherwise.

In order to determine whether or not its associated object is at a requested abstract state, the state predicate has to get some information about its object. It's part of the definition of the state predicate to state clearly the nature of this information and it's the responsibility of the programmer that defines the active object class that uses a particular state predicate to make sure that its class makes available the information required by the state predicate to determine whether or not the object is at a given state.

2.2.3. *Activation Conditions.* Activation conditions are used to constrain method invocations and more generally the execution of the object's threads. They associate methods with a condition, expressed in terms of abstract states, that has to be true in order to run a thread that executes the associated method. Activation conditions may be associated to an object either statically, in it's class definition, or dynamically to a particular instance at run-time.

In the definition of a class, the variable conditions is used to specify activation conditions. This variable should be set to a dictionary having as keys objects that designate a set of methods the acceptance of which is constrained by the condition, and as values, boolean functions that specify the conditions. The following types of objects may be used as keys:

- A string: in this case the string is the name of the method that is constrained by the condition.
- A list of strings: the strings have to be method names and in this case all methods in the list are constrained by the condition.
- A function: the function has to evaluate to a list of strings designating object methods. Such functions are evaluated at the creation of the active object class to determine the actual set of methods that is constrained by the condition. The function may use in its evaluation variables defined by the object that contain lists of method names as well as predefined functions that return such lists. For instance, the function allMethods may be used to return the list of all methods of the object.

Conditions are boolean functions that may use the predefined function atState that takes as argument an abstract state specification and returns the values true or false

depending on whether or not the object is at a state that matches it's argument. Abstract state specifications and the matching depend on the state predicate that is associated with a particular state.

Activation conditions may be associated with an instance at run-time by calling the method addCondition. This method is provided by the object manager and is supported by all active objects.

2.2.4. *Synchronization Actions.* The programmer can define a number of actions to be executed when certain events, such as the receipt of a request or the completion of a method execution, take place during the execution of the object. In these actions the programmer may use variables especially defined for this purpose to keep track of the occurrence of such events. These variables may then be used in the definition state predicates. This mechanism in combination with abstract states and state predicates, may be used to specify synchronization based on *synchronization counters.*

Some of these actions could be included directly in the code of object. However, this approach has several disadvantages. Most importantly, one can not anticipate all the information that will be needed by state predicates to be defined in subclasses when writing the methods of a class the first time. If more information is needed for defining the state predicates in a subclass, it will be necessary to redefine the methods inherited from the class. In addition, the code of the object's methods would become more complex as it would mix computations that have to carried out by the method as well as computations that are used to keep track of such events.

Such a feature has been presented in previous proposals [25][15]. In these, pre-actions and post-actions can be associated with methods and are executed before and after the execution of their associated methods. A more detailed discussion of the benefits of a such feature may be found in these references. Our proposal extends these previous proposals by the inclusion of further actions and provides more flexibility for the specification and inheritance of actions.

In addition to pre-actions and post-actions we introduce actions that are executed when a method invocation request is received by an object, actions that are executed when the execution of a method is suspended and resumed.

Synchronization actions are associated with methods by the definition of the dictionaries pre_action, post_actions, suspend_actions, resume_actions and receipt_actions.

The keys in the dictionary are either lists of method names or functions that evaluate to lists of method names. The values in the dictionary specify the actions associated with these various events in the execution of methods.

2.2.5. *Thread Creation and Synchronization. Thread Creation.* In addition to threads that are created to execute method invocation requests, it is possible for objects to have a number of threads that are created when an object is instantiated. These threads may execute quasi-concurrently with each other and with the threads that execute method invocation requests. The execution of these threads is constrained in the same way as the threads that execute requests.

The variable **activities** is used in the definition of a class to specify the list of methods that are to be executed by threads created spontaneously at the creation of an instance of the class. If the object's class defines a method with the name **Activity**, this is also executed in new thread at the creation of the object. The execution of such threads is constrained by activation conditions in the same way that threads that execute method invocation requests are. In the definition of a class it is possible to specify whether or not the activities of parent classes are inherited.

Synchronizing threads with abstract states of the object. A thread may suspend its execution until the object reaches a specified state by calling the method **suspendUntill** providing as argument an abstract state expression that specifies the requested state. This feature is particularly interesting when combined with the execution of background threads. Such a thread may loop waiting for the object to reach some state where it executes some background actions. The use of these features for defining and reusing active object behaviours is shown in the examples in section 4.

Synchronization with requests sent to other objects. The method **sendAndSuspend** may be used to issue a request to another object and suspend the calling thread until a reply is returned by the called object. This allows the calling object to do some other actions in a background thread or possibly accept other request while the called object processes its request. This feature provides some flexibility for reply scheduling and addresses the concurrent programming problems known as remote delays and nested monitor calls.

Asynchronous State Notification. An object that wants to be notified when another object reaches a state that satisfies an abstract state expression has to first issue a

notification request to the source object. The request returns an object that represents the notification event. This object may then be used to suspend a thread until the event occurs.

A notification request to an active object is made by calling the method **notifyRequest()** or the method **atState()**. This method takes as argument an abstract state expression and returns an object that represents the notification event. The method **suspendUntil()** and **waitUntil()** take as argument an object representing a notification event and suspend the calling thread until the event occurs. With the former, while the calling thread is suspended other threads in the object may be scheduled for execution. The execution of the suspended thread is resumed sometime after the event has occured and when there are no other active threads within the object. With the latter, no other threads may be scheduled while the thread waits for the event. In this case the thread is resumed immediately after the event occurrence.

It is also possible to wait for a combination of notification events. This is accomplished by the method **suspendUntilComplexEvent**. This method takes two arguments. The first argument is a dictionary that has keys strings used to tag the events and as values notification event object returned from calls to **notifyRequest**. The second argument may be one of the strings 'Any' or 'All'. In the first case, the complex event occurs when any of the notification events occurs. In the second case, the thread waits until all events have occured. The method returns the tag of the last event. This is useful in the case of 'Any' to determine which among a set of mutually exclusive events has occured.

Synchronous Notification. The syncBlock method allows the execution of a block of code in synch with one or more objects at a given state. For instance, the code below specifies that the code following 'do': will be executed when the object, anObject, is at the state: aState. The local variable theObject is bound to the object when the block of code following 'do': is executed.

```
self.SyncBlock(
  {'with': {'theObject': (anObject, ('aState',)),},
   'do': ''theObject.aMethod()''
   })
```

Once the object, anObject has reached the request state it will not accept any more calls at its interface. Method calls are only accepted through a privileged interface that is only known in the syncBlock code and is bound to a variable local to the block

- the variable theObject in the example. After the code block is executed the privileged interface is discarded and messages are again accepted at the ordinary object interface.

Another feature of the privileged interface used in the syncBlock is that activation conditions for methods are interpreted as assertions. That is, it is an error if the activation condition for a method is not satisfied whwn a method is called through this interface.

The reason for interpreting activation conditions as assertions is that it does not make sense to suspend a method call within a syncBlock statement. As the object will not accept any other calls on its ordinary interface its state may not change but only through calls executed within the syncBlock.

2.2.6. *Message Parsing Features. Issuing Requests.* Active object's methods may be invoked using the ordinary Python method invocation syntax which has in our model the semantics of a remote procedure call. However, the model provides other constructs allowing more flexibility in structuring object interactions. The message passing features provided are the following:

- Remote procedure call: this is supported by the ordinary method call syntax of Python. The calling object is blocked until the calling object replies
- Asynchronous method call: this is supported by the method send which, takes as argument a dictionary that represents the message to be sent. The field of this dictionary are explained below.
- Non-blocking remote procedure call: this is supported by the method sendAndSuspend that takes as argument a dictionary representing a message. The difference with remote procedure call is that while the thread that issues the request is suspended other threads may run in the object. The suspended thread is resumed after the reply has been received, there is no other active thread in the object and the object ia at a state where the method associated with thread can be run. The non-blocking designation assigned to this feature should be understood with respect to the object. The calling thread itself is blocked.

The dictionary that is used to represent the message to be sent by the Send and sendAndSuspend method contains the following fields:

- target: the object to which the message is sent
- key: a string specifyng the name of the method to be called
- args: a tuple with the arguments to be passed to the method

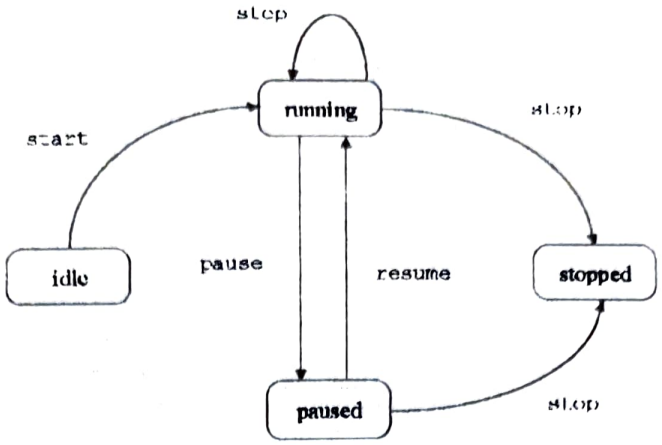

FIGURE 2. State diagram for the behavior of objects of class activity

- replyTo: this field is optional. Its value may be set to None. In either case, if it is present, the message is send asynchronously. In the case where it's set to a reply destination object other than None, the reply to the request is delivered to the caller associated with this reply destination.

3. Examples

3.1. **Combining Behavioural Patterns through inheritance.** Previous proposal for combining concurrency features with class inheritance, considered that, in order to be able to reuse methods in subclasses through inheritance, methods should not contain any synchronisation code. With the features provided in our model it is possible to use inheritance for reusing methods that contain synchronisation code. In this section we show, not only, that is not a problem but that in addition, this possibility also enhances reuse by allowing the definition and reuse of mixins that define behavioral patterns.

We define a set of abstract classes and mixins that specify and allow the reuse of the behavior of objects that representing continous activities.

A Basic Activity

The most general such behaviour is defined by the class Activity. The behavior of instances of this class is shown abstractly in the state diagram in Figure 3. Objects of class Activity may be at the states idle, running, paused and stopped. When first created, they are at the state idle where they can accept the method start and move at the state running. While they are in the state running they continously execute their method stepaction. This method should be redefined by subclasses and it corresponds to the actions to be executed by the activity at each step. At the state running objects accept calls to their pause and stop methods. The execution of the pause method moves

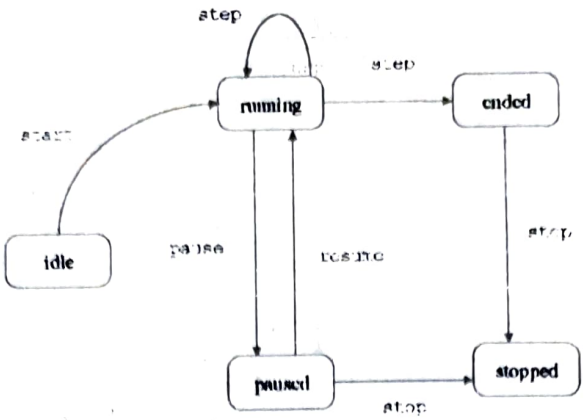

FIGURE 3. Behavior of an activity that ends

the object to the state paused where the execution of their activity is ceased temporarily. From the paused state an object may accept a resume method call and move back to the running state where it resumes the execution its activity. At the states running and paused, an object may accept a call to stop which moves the object into the stopped state. After moving to this state the object will not accept any further requests. Moving to this state triggers the execution of internal actions for freeing the resources used by the object.

Activities that Terminate after a Number of Steps. The class ActivityWithEnd refines the behavior of basic activities to specify the behavior of activities that terminate after a finite number of steps. The behavior of such an activity is illustrated by the state diagram in Figure 2. The state diagram includes a new state ended where an instance of ActivityWithEnd may move into after executing the last step of activity. Note that the state ended is different than the stopped state. Consider for instance an object encapsulating the playback of a video clip. The activity ends after the last frame of the clip fas been displayed by stepaction. However, the window showing the last frame may remain visible on the display. However, if the object moves into the stopped state, the window showing the last frame disapears.

The definition of the class ActivityWithEnd is shown in Figure 4. This class can be inherited by a class that also inherits Activity to obtain the behavior of an activity that ends after a finitenumber of steps.

This class defines two instance variables length and step. Length stores the number of steps of the activity and step stores the number of steps that were executed so far by the activity. The post-actions associated with stepaction is used by the state predicate to determined if the object is at the state ended. This function compares the number of

```
class SimpleActivity(ActiveObjectSupport):
  methods = ['pause', 'start', 'stop', 'resume', 'stepaction']
  states = ['idle','running', 'paused', 'stopped']
  conditions = {'start' : (lambda o: o.atState(('idle',)),),
                'stepaction' : (lambda o: o.atState(('running',)),)}
  state_predicates = {enum_states: [ 'idle','running', 'paused', 'stopped']}
  enum_states_afuncs = {'idle': 'isIdle', 'running' : 'isRunning',
                        'paused' : 'isPaused','stopped' : 'isStopped'}
  method_call_opt = ['stepaction']
  def __init__(self, stepDelay = 2):
    self.step = 0, self.running = 0, self.idle = 1, self.paused = 0
    self.active = 0, self.stopped = 0, self.steptime = stepDelay
  def isIdle(self,state): return self.idle
  def isRunning(self,state): return self.running
  def isPaused(self,state): return self.paused
  def isStopped(self,state): return self.stopped
  def Activity(self):
    #self.run = self.interface.notifyRequest(('running',))
    while not self.isStopped(1):
      self.step = self.step + 1
      self.sendAndSuspend({ 'target' : self.interface,
                            'key' : 'stepaction', 'args' : ()})
  def stepaction(self):
    print "executing step: %d ... \n" % self.step
    time.sleep(self.steptime)
  def pause(self): self.running = 0, self.paused = 1
  def start(self): self.active = 1, self.running = 1
  def resume(self): self.running = 1, self.paused = 0
  def stop(): self.running = 0, self.active = 0
class ActivityWithEnd(SimpleActivity):
  length = 10, states = ['ended']
  state_predicates = { enum_states : [ 'ended']}
  enum_states_afuncs = {'ended': 'isEnded'}
  conditions = {'stepaction': (lambda o: not o.atState(('ended',)), 'and')}
  def isEnded(self,state): return self.step >= self.length
class loopingActivity(ActivityWithEnd):
  loop = 0, states = ['looping']
  state_predicates = { enum_states : [ 'looping']}
  enum_states_afuncs = {'looping': 'inloop'}
  conditions = {'stepaction': (lambda o: not o.atState(('ended',))
                or o.atState(('looping',)),'and')}
  methods = ['looptoggle']
  def isEnded(self,state): return self.step >= self.length
  def inloop(self,state): return self.loop
  def looptoggle(self): self.loop = not self.loop
```

FIGURE 4. Continous activity with an end. Definition of a continous activity. Looping activity.

steps executed by the activity to the number of steps of the activity and returns true if they are equal.

This class defines a new activation condition that is combined with inherited activation conditions to constrain the execution of the method step action. This condition

ensures that the activity will not execute any more steps after it has reached its end - has executed its total number of its steps.

Activities that loop. The class loopingActivity can be combined with the previous classes to specify the behavior of activities that can loop. This class introduces a new abstract state looping and two new methods looptoggle and reset. The method looptoggle may be accepted at any abstract state of the object other than stopped and moves the objects from the abstract state looping to the abstract state not looping. This method does not affect the other abstract states of the object. If during its execution an object moves to a state corresponding to the abstract state ended and looping, the reset method is executed and the object moves to the state running where it starts executing its activity from the beginning. The reset method has to be defined in subclasses and it should include the actions needed for restarting the activity. For instance, if the object plays a video clip that is stored in a file, the reset action moves the file pointer to the beginning so that its stepaction method will redisplay the video frames from the beginning of the video clip.

Figure 4 shows the definition of the class loopingActivity. This class defines the function looping that is used to move into and out of this state. The method ResetAndEnd is executed by a thread created spontaneously at the creation of the object - this is specified by the definition of the activity variable. This thread waits until the object at a state where the abstract states looping and ended are true simultaneously. Then, it calls reset to execute the actions that will allow the activity to restart executing from the beginning. The post-action associated with reset sets the value of step to zero.

This has the effect that the abstract state ended is no longer true so that the method stepaction that was constrained, in activityWithEnd, by a condition starting that the object should not be at this state, may again be accepted.

3.2. Object coordination.

3.2.1. *Associating Audioeffects to Playback of a Video Clip.* In this example we use state notification and the dinamic definition of abstract states and state predicates to associate some audio effects to the play back a video clip.

In this example an instance of a videoPlayer class reads video frames from a file and displays them in a window on the screen. An instance of an AudioEffectManager class associates dynamically with the videoPlayer object a state predicate and a set of abstract states that "annotate" the video clip. The AudioEffectManager uses state notification to

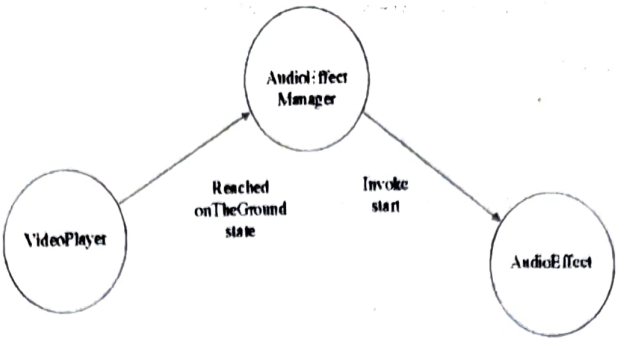

FIGURE 5. Adding audio effects to video

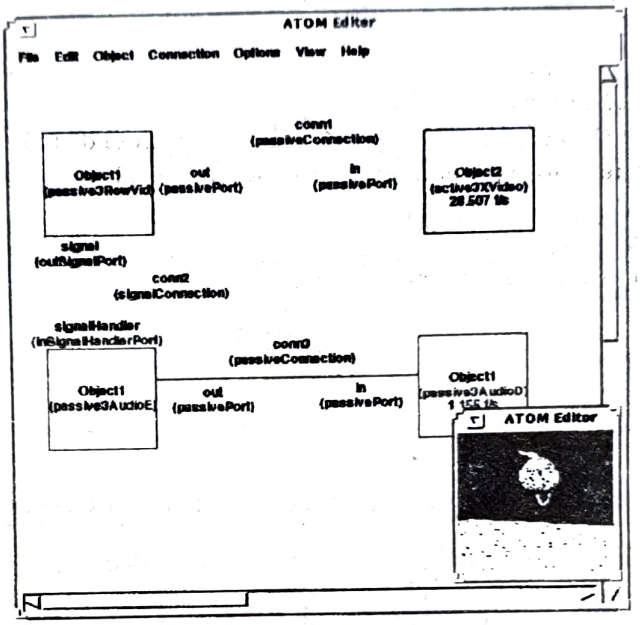

FIGURE 6. The Active Object Model Editor: Teapot example

wait for the occurrence of a visual event and activates an audioEffect object that plays the associated sound.

In this particular example the video shows a teapot that jumps, does a looping in the air and falls on the ground. A state predicate of class flightVideoAnnotations is associated with the videoPlayer object and defines the abstract states: inTheAir, onTheGround and highest. These states are associated with visual events in the playback of the video-clip. For instance, the videoPlayer object is at the abstract state onTheGround when it displays the video frames where the teapot is on the ground. In the example, the AudioEffectManager uses state notification to wait for the event that the videoPlayer is at the abstract state onTheGround to generate an impact sound. Figure 5 shows the program structure used for the example.

The state predicate, flightVideoAnnotation, that is used in this example to define abstract states that are associated with visual events is shown in Figure 7. The association

```
class flightVideoAnnotations:
  obj = None
  functs = {
    'highest': (lambda x: x.frame == 13),
    'inTheAir': (lambda x: 1 < x.frame < 26)
  }
  def __init__(self, object):
    self.obj = object
  def evalState(self, state, obj):
    return self.functs[state[0]](obj)
class AudioEffectManager(ActiveObjectSupport):
  def __init__(self, video, state, audioEffect):
    self.state = state, self.video = video
    self.audioEffect = audioEffect
    self.video.newPred(flightVideoAnnotations,
                       ['highest', 'onTheAir', 'onTheGround'],))
    self.fall = self.video.notifyRequest(('onTheGround',))
  def activity(self):
    while not self.atState(('stopped',)):
      self.suspendUntil(self.fall), self.audioEffec.start()
```

FIGURE 7

between the abstract states and visual events in the video clip is made by using the instance variable step of the videoPlayer that provides information about the frame that is displayed. The class defines three functions that establish the relationsheep between video frames and abstract states. A state predicate class such as videoflightVideoAnnotations may in fact be generated automatically by a tool that allows a user go view a video and associate interactively frames with abstract states.

The class AudioEffectManager is shown in figure. At its creation an AudioEffectManager object is acquitainted to a videoPlayer object and an AudioEffect object. using the newPred method, it associates dynamically with the VideoPlayer object the state predicate flightVideoAnnotation that defines the abstract states associated with the visual events of interest. Then, it requests by calling the method notifyRequest, to be notified when the videoPlayer object is at the abstract state onTheGround. The call to notifyRequest returns an object representing the notification event. After initialization the activity method of the AudioEffectManager is executed in a new thread. In this method the event object, returned by the call to notifyRequest, is used in the call to the suspendUntil method to suspend the execution of the method until the videoPlayer object reaches the abstract state ontheGround. When the state is reached the thread is resumed and it invokes the AudioEffect's start method to playback the audio effect.

To keep the example simple, the association of abstract states to audio effects is done statically in the code of the AudioEffectManager class that allows the association of different audio effects to several abstract states of the videoPlayer. Furthemore, the association of audio effects to abstract states can be specified in a list that is passed to the AudioEffectManager at initialization rather than having it hard-wired in the code of the class in the example.

4. Conclusion

We have presented an active object model that combines concurrency and object-oriented features. The model integrates concurrency and object-oriented features in such a way that alleviates several known problems for taking advantage of the software reuse potential of object-oriented features in the development of concurrent software. In addition, the model provides support for novel ways to combine concurrent object behaviors.

The model introduces the novel features of abstract states, state predicates and state notification that can be used to synchronize the actions of a single objects as well as coordinate the execution of sets of objects. This done in a way that is compatible with polymorphism and inheritance and provide novel ways to support reuse by combining active object behaviours.

The proposed model has been implemented as an extension to the programming language Python. We have started using the prototype for the development of multimedia programming environment based on active objects and have had very positive experiences. The implementation of the model in language that is freely available on different platforms this will allow us and other researchers to gain more experience with the model by using it for developing concurrent software and further refine the model's features.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, *Abstracting Object Interactions Using Composition Filters*, Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, ed. R. Guerraoui, O. Nierstrasz, M. Riveill.

[2] P. America, *Inheritance and Subtyping in a Parallel Object-Oriented Language*, Proceedings of the ECOOP '87, ed. J. Bezivin, J-M. Hullot, P. Cointe and H. Lieberman.

[3] C. Atkinson, S. Goldsack, A. di Maio, R. Bayan, *Object-Oriented Concurrency and Distribution in DRAGOON*, JOOP, March/April 1991.

[4] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution*, Addison-Wesley/ACM Press, 1991Lodewijk Bergmans, "Composing Concurrent Objects", Ph. D. Thesis, University of Twente, 1994.

[5] T. Bloom, *Evaluation Synchronization Mechanisms*, in 7th International ACM Symposium on Operating Systems Principles, 1977.

[6] D. Caromel, *Concurrency and Reusability: From Sequential to Parallel*, JOOP, Sept./Oct. 1990.

[7] S. Frolund, *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*, Proceedings ECOOP '92, ed. O. Lehrmann Madsen.

[8] S. Frolund, G. Agha, *A language Framework for Multi-Object Coordination*, Proceedings ECOOP '93.

[9] S. Matsuoka, K. Taura, A. Yonezawa, *Highly Efficient and Encapsulated Re-use of Synchronisation Code in Concurrent Object-Oriented Languages*, Proceedings OOPSLA '93.

[10] B. Meyer, *Systematic Concurrent Object-Oriented Programming*, Communications of the ACM, Sept. 1993.

[11] C. Neusius, *Synchronisation Actions*, Proceedings of ECOOP '91, July 1991.

[12] M. Papathomas, D. Konstantas, *Integrating Concurrency and Object-Oriented Programming: An Evaluation of Hybrid*, in Object Management, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1990.

[13] M. Papathomas, *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*, Ph. D. Thezis, University of Geneva, 1992.

[14] M. Papathomas, G. S. Blair, G. Coulson, *A model for Active Object Coordination and its use for Distributed Multimedia Applications*, ECOOP '94 Workshop on Coordination Models and Languages for Parallelism and Distribution, Bologna, Italy, July 1994.

[15] M. Papathomas, *Concurrency in Object-Oriented Programming Languages*, in Object-Oriented Software Composition, Prentice Hall, O. Nierstrasz and D. Tsichritzis eds.

[16] A. Yonezawa, E. Shibayama, T. Takada, Y. Honda, *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1*, in Object-Oriented Concurrent Programming, ed. M. Tokoro, MIT Press, 1987.

LSR-IMAG Grenoble, France

"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, RO-3400 Cluj-Napoca, Romania

E-mail address: scuty@cs.ubbcluj.ro