Concurrent Object-Oriented Programming in Python with ATOM

Michael Papathomas
Computing Department
Lancaster University
Lancaster LA1 4YR, UK
michael@comp.lancs.ac.uk

Anders Andersen*
NORUT IT
Tromsø Research Park
9005 Tromsø, Norway
aandersen@acm.org

Abstract

Object-oriented mechanisms, such as classes and inheritance, and concurrency mechanisms, such as threads and locks, provide two separate software structuring dimensions. The development of concurrent object-oriented software requires both dimensions to be taken into account simultaneously. Previous research has identified that substantial effort is required by programmers to avoid clashes in structuring software along these separate dimensions. This has led to the design of concurrent object-oriented programming models that integrate object-oriented features with concurrent execution and synchronization, eliminating the need to consider two separate dimensions when developing concurrent object-oriented software. Although several issues that have to be addressed by such programming models have been identified, there is no consent on a programming model that addresses all these issues simultaneously. In addition, little experience has been gained from the use of proposals addressing these issues. We have used Python to prototype and experiment with the use of a novel concurrent object-oriented programming model called ATOM. In this paper we present the model's main features and illustrate their use for concurrent programming in Python. We also provide information on a free prototype implementation of the model. Taking advantage of Python's extensibility we were able to prototype the model without undergoing a lengthy development effort and with no need to change the Python language or its interpreter.

*This work has been supported by a NATO Science Fellowship through The Norwegian Research Council and was carried out while being a visiting student at the Computing Department, Lancaster University.

1 Introduction

Object-oriented mechanisms, such as classes and inheritance, and concurrency mechanisms, such as threads and locks, provide two separate software structuring dimensions. The development of concurrent object-oriented software requires both dimensions to be taken into account simultaneously. Previous research has identified that substantial effort is required by programmers to avoid clashes in structuring software along these separate dimensions. This has led to the design of concurrent object-oriented programming models that integrate object-oriented features with concurrent execution and synchronization eliminating the need to consider two separate dimensions when developing concurrent object-oriented software [1].

There has been substantial research on Concurrent Object-Oriented Programming (COOP) models and several issues that have to be addressed by such models have been identified. However, research in the area are concentrated on the design of features that address particular issues in isolation and there is no consent on a programming model that addresses all these issues simultaneously. Furthermore, few languages incorporating the proposed features are widely available and little experience is reported from their use.

In this paper we present a COOP model, ATOM, and show how it can be used for concurrent programming in Python. ATOM incorporates a number of novel features that aim to address simultaneously all issues identified by previous research in concurrent object-oriented programming. ATOM has been incorporated and implemented in Python and we have already used it in the development of concurrent object-oriented software. Section 2 provides some background on concurrent object-oriented programming and the issues that have to be addressed by COOP models. In section 3, we present ATOM's main features. Section 4, shows how ATOM can be used for concurrent object-oriented programs in Python. In section 5, we discuss how ATOM features address concurrent programming issues, the benefits from the use of Python and our plans for future work. Finally, we present our conclusions in section 6.

2 Background and Motivation

2.1 Previous Work and Motivation

The purpose of COOP models is to integrate in a single programming model object-oriented features, such as classes and inheritance, and features that support concurrent execution and synchronization. In this section we provide a brief overview of the work in this area, the motivation behind such models and the issues they have to address. There has been abundant literature on COOP so it is impossible to present all the work in this paper. A more extensive survey of work in the area can be found in [2].

2.1.1 Early Work on COOP Models

In a concurrent program objects are shared by concurrent threads. In this case the execution of their methods needs to be synchronized in order to provide mutually exclusive access to the objects' state as well as to coordinate the use of an object by concurrent threads. Synchronization can be expressed using low-level synchronization mechanisms used independently from the object-oriented constructs. However, this approach does not scale well if objects are to be reused in programs which have a different thread structure. Early work on COOP [3, 4, 5, 6] concentrated on the design of features that integrate objects with concurrency features providing adequate expressive power for classical concurrent programming problems. For instance, in some proposals objects are identified with message passing processes and asynchronous message passing variants are introduced. In other proposals, objects are similar to monitors. They are provided with mutual exclusion and some features are provided to coordinate the execution of invoking threads.

2.1.2 Synchronization Constraints

Early COOP models either did not provide support for inheritance or the use of inheritance required substantial rewriting of inherited code to synchronize inherited methods with those defined in subclasses. This issue has been discussed in previous research [3, 7, 8, 9] and the term *inheritance anomaly* introduced by Matsuoka [10] is now often used to refer to related issues. The approach commonly adopted to avoid such problems is to synchronize method execution by the specification of synchronization constraints on the acceptance of messages. Synchronization constraints are specified separately from the code of methods. As methods do not contain any synchronization code, it is easier to reuse them in subclasses without modification. Various approaches for specifying, inheriting and combining inherited synchronization constraints have been proposed [11, 12, 13, 14, 15, 16, 17, 18, 19, 20].

2.1.3 Object Coordination

In a concurrent object-oriented program coordination between cooperating objects is traditionally expressed through the use of concurrency constructs embedded in the implementation of the objects. The main motivation behind the work on object coordination is to allow coordination patterns among several objects to be specified separately from the implementation of individual objects. The benefits of such an approach is that it is possible to coordinate objects in ways that were not anticipated when the objects were implemented and to allow the reuse of the coordination patterns themselves [21, 22, 9].

2.2 Further Work on COOP Models

COOP models have to provide adequate expressive power for coping with general concurrent programming problems. This requires a choice of appropriate programming constructs for thread creation, message passing and synchronization. The choice of such features was the main concern of early COOP models. Since it was identified that these models had difficulties of taking advantage of inheritance, most work in the area concentrated on proposals for specifying and reusing through inheritanced synchronization constraints for method invocation. However, the proposals for the specification of synchronization constraints are based on oversimplified COOP models that fail in providing adequate expressive power. In particular: (i) synchronization constraints are not compatible with the specification of objects that have internal activities and (ii) these proposals are based on the assumption that once a message is accepted, method execution proceeds to completion. This overconstrains the message processing patterns that are expressible.

2.3 The Motivation Behind ATOM

The motivation behind ATOM is the design of a COOP model that simultaneously supports (i) the specification and reuse of synchronization constraints, (ii) adequate expressive power and (iii) object coordination. We also wanted to be able to use the model in the development of concurrent software and evaluate and gradually refine its features. However, we wanted to avoid a major language development effort. This was achieved to a large extent by the implementation and incorporation of a prototype of the model using the dynamic features of Python (see section 5.2).

3 The ATOM Concurrent Object Model

3.1 Overview of the Computational Model

ATOM objects are active entities that resemble multi-threaded servers that accept and process messages in an order that is most suitable to them. Messages are processed by the creation of a new thread within the object. In addition, threads may also be created spontaneously at the creation of an object for executing internal activities. Only a single thread may be active at a time within the object and the execution of threads is non-preemptive; another thread may run only when the current thread suspends its execution. Threads are associated with *activation conditions* (discussed in section 3.3) which determine the states of the object that are compatible with their execution.

Central to the ATOM model are the novel features of *abstract states*, *state predicates* and *state notification*. These features are integrated with thread scheduling and message passing in such a way that adequate expressive power for COOP, support for inheritance and the specification and reuse of coordination algorithms are addressed simultaneously.

An object in ATOM may be either executing some thread or it may be waiting for an event, such as a message invocation (other events will be discussed later), that will resume the execution of a suspended thread. When no thread is running or the current thread is suspended, the object is at a *stable state*. The execution of an object can be represented by a sequence of stable states as illustrated in the right part of Figure 1. When the object is at stable state, a *ready* thread is scheduled for execution using a simple round-robin algorithm. A thread is ready if it is not waiting for an event, such as the reply to a message or a state notification event (explained later), and the activation conditions associated with the thread hold at that particular state.

3.2 Abstract States and State Predicates

Activation conditions are expressed in terms of *abstract states*, which represent properties of the object's state at a level of abstraction that hides implementation details. Abstract states are defined by the programer and their definitions may be inherited in subclasses. 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.

A *state predicate* is an evaluator of abstract states associated with an object. State predicates are objects, defined by the programmer, that are used by the ATOM run-time to determine if an abstract state is true at a given stable state. The ATOM run-time interacts with state predicates following a message protocol that has to be supported by such objects. State predicates may be associated with an object statically or dynamically at run-time. It is also possible for a state predicate to be shared among several objects. This last feature is used to support object coordination.

3.3 Activation Conditions

Activation conditions are used to constrain method acceptance and more generally the execution of the object's threads. Methods are associated with an activation 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 its class definition, or dynamically to a particular instance at run-time. It is also possible to associate a condition with a particular message sent to an object. Static activation conditions defined in a class are inherited by its subclasses and support is provided for defining generic activation conditions. Inherited activation conditions are by default conjoined with the ones defined in subclasses.

3.4 Synchronization Actions and Variables

The programmer can define a number of actions to be executed when certain events, such as the receipt of a message, the invocation and completion of a method and the suspension and resumption of a thread, occur. In these *synchronization actions* the programmer may use

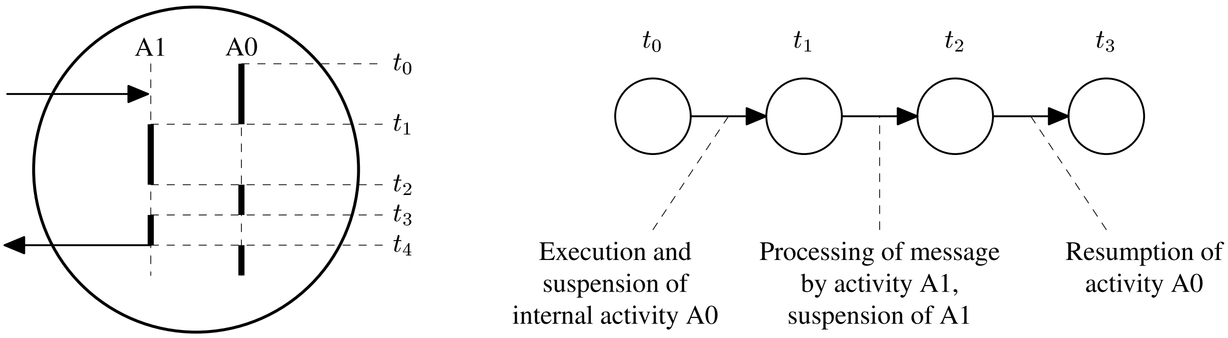

Figure 1 ATOM execution model

synchronization variables especially defined for this purpose to keep track of the occurrence of such events. This mechanism in combination with abstract states and state predicates can be used to define abstract states that capture aspects of the history of object execution that are not represented in the object's state. Synchronization actions defined in a class are inherited by its subclasses. Thus they provide more expressive power for the specification of generic synchronization constraints than mixins.

3.5 Abstract State Notification

State notification is one of the features provided in ATOM for supporting object coordination. It can be used by an object to monitor and synchronize with abstract state changes of other objects. An object that wants to be notified when another object reaches an abstract state first issues a state notification request to the source object. This returns a notification event object that can be used to suspend one of its threads. State notification may be asynchronous or synchronous. The latter ensures that when the thread in the notified object resumes, the source object is still at the requested state.

It is important to note that state notification does not require the explicit collaboration of the source object. This supports object coordination in a way that promotes a high degree of code reuse at two levels: on the one hand, objects are more reusable because they are coded without specifying coordination constraints. On the other hand, the coordination algorithms are more reusable as they are specified separately from the implementation of the involved objects. In addition, as coordination is expressed on abstract information, it is possible to specify generic coordination algorithms that can be reused for objects with different interfaces but similar abstract behavior.

3.6 Message Passing

The *message passing* facilities are integrated with thread scheduling to provide adequate expressive power in the way objects process messages, how and when they receive replies and how they reply to messages. The following ways of sending messages are provided:

- *Blocking remote procedure call:* when this type of message passing is used, the thread and its object are blocked until the message is received by the destination and a reply is returned. The ordinary Python method invocation on ATOM objects has this semantics.
- *Non-blocking remote procedure call:* The difference between non-blocking and blocking remote procedure call is that while the sending thread in a non-blocking remote procedure call waits for the reply, other threads may run in the object. The suspended thread is resumed after the reply has been received if there is no other active thread in the object and the object is at a state where the method associated with the thread can be run. The non-blocking designation should be understood with respect to the object. The calling thread itself is blocked.
- *Asynchronous message passing:* the calling thread sends a message and proceeds with its execution without waiting for a reply. However, it is possible to specify that another object, which should an instance of the class `Reply`, will receive the reply. `Reply` objects have an abstract state `Ready` and a method `getResult` which is accepted at this state to retrieve the reply later. State notification may be used on the `Ready` state of a `Reply` object to avoid blocking an object if the result is not yet available. The use of these features is illustrated in section 4.4.2.

4 Concurrent Programming with ATOM

In this section we show how ATOM can be used for concurrent programming in Python and in particular how its features address the issues discussed in section 2. Due to space limitations we do not describe all of ATOM's features in detail. However, we believe that the examples in this section provide a good idea on the use of ATOM in Python. A detailed description can be found in [23].

4.1 Using ATOM in Python

To create an ATOM object, also called an *active object* in the rest of the paper, one first defines a class that inherits the class `ActiveObjectSupport`. Methods defined in this class can be used in subclasses to access ATOM's concurrency features such as sending messages asynchronously and suspending the object's threads.

4.1.1 Creating ATOM Objects

An *active object class* is created by applying the function `ActiveObject`[1] to a class that inherits from `ActiveObjectSupport`. The Python class returned from `ActiveObject` is used to create active objects with the ordinary Python object creation syntax. For example, the `Buffer` class in Figure 2 can be used to create an active object class, `BoundedBuffer`, which is then instantiated to create active objects:

```
# Create an active object class
BoundedBuffer=ActiveObject(Buffer)
# Create a buffer with size 10
aBoundedBuffer=BoundedBuffer(10)
```

4.1.2 Special Attributes

A number of attributes defined in a class, used to create an active object class, are interpreted in a special way. These are shown in Table 1. None of these has to be defined to create an active object class. However, if the `methods` attribute is not defined, no method of the active of object may be invoked by other objects. The methods attribute specifies the list of methods that may be called by other objects. The other methods defined in the class are considered "private". The other special attributes are discussed when used in the following sections.

[1] In the next release of ATOM it is no longer needed to invoke the `ActiveObject` function provided that the `__init__` method of `ActiveObjectSupport` is invoked.

Attribute	Description
`states`	abstract states
`methods`	public methods
`state_predicates`	state predicates
`conditions`	activation conditions
`pre_actions`	pre actions
`post_actions`	post actions
`receipt_actions`	receipt actions
`activities`	internal activities

Table 1 Special attributes for active object classes

4.2 Synchronization Constraints

4.2.1 Simple Activation Conditions

In the definition of the class `Buffer` in Figure 2, the `methods` variable specifies that a buffer accepts messages for executing its `put` and `get` methods. The `states` variable defines the abstract states of the class. In the `Buffer` class the abstract states `empty` and `full` are defined for representing the corresponding states of a buffer. The association between abstract states and state predicates is implicit. A default state predicate object is created implicitly and the methods `empty` and `full` are used to determine whether or not the bounded buffer is at any of these abstract states. The `conditions` dictionary associates messages with activation conditions. Activation conditions are Boolean functions that take a single argument. A number of methods can be invoked on this argument to obtain information on the abstract state of the object. In the `Buffer` class, the messages `put` and `get` are associated with activation conditions that constrain the acceptance of these messages at the abstract states where the buffer is not full and not empty respectively. The method `atState` is used in activation conditions to refer to abstract states of the object.

4.3 Generic Activation Conditions

Figure 3 shows the possibility to define generic synchronization constraints as mixins. The class `lockMixin` defines an abstract state `locked`. The activation conditions defined in the class specify that when the object is at the abstract state `locked` no method but `unlock` may be accepted. The set of methods constrained by the activation condition is specified by a function that evaluates to a list of methods. This function uses the predefined method `allMethodsExcept` on its argument to obtain the set of all methods ex-

```
class Buffer(ActiveObjectSupport):

  # Special attributes
  states=['empty','full']
  methods=['put','get']
  conditions={
    'put':
    (lambda o:
     not o.atState(('full',)),),
    'get':
    (lambda o:
     not o.atState(('empty',)),)}

  # Initialization
  def __init__(self,size):
    self.inbuffer=0
    self.limit=size
    self.store=[]

  # Used by default state predicate
  def empty(self,state):
    return self.inbuffer==0
  def full(self,state):
    return self.inbuffer==self.limit

  # Methods
  def put(self,data):
    self.store.append(data)
    self.inbuffer=self.inbuffer+1
  def get(self):
    self.inbuffer=self.inbuffer-1
    data=self.store[0];
    del self.store[0]
    return data
```

Figure 2 Simple message acceptance constrains

cept `unlock`. When this class is inherited by some other class, `allMethodsExcept` will return[2] the set all methods of the subclass except `unlock`. Thus, the activation condition will constrain the execution of all methods of the classes that inherit the lock mixin.

4.4 Expressive Power

4.4.1 Use of Internal Activities

Figure 4 illustrates the specification and use of internal activities in ATOM. The class `adaptBuffer` specializes the behavior of the bounded buffer so that the size allocated for the buffer is increased, in a background activity, if the buffer is full most of the time. The `activities` variable is used to specify that the method `extendSize` is run in a new thread at the creation of the object. This thread is run, as specified by the conditions variable, only when the buffer is at the abstract state `oftenFull`. When the thread is activated it increases the allowable buffer size, `limit`, by a fixed amount, `incr`, then suspends its execution to allow other threads to run. The method `oftenFull` is used to determine if the buffer is at the associated state by examining if the ratio of `put` messages accepted when the buffer was full is greater than `factor`. This is only examined after it has received a number of `put` messages greater than `freq` since last `extendSize` was done. The information needed by `oftenFull` is maintained in the synchronization variables[3] `put_count` and `full_count` and is updated using a synchronization action, `monitorPut`, which is executed, as specified by `receipt_action`, whenever a `put` message is received.

```
class LockMixin:

  # Special attributes
  states=['locked']
  methods=['lock','unlock']
  conditions={
    lambda o:
    o.allMethodsExcept(['unlock']):
     (lambda o:
      not o.atState(('locked',)),)}

  # Instance variables
  islocked=0

  # State function
  def locked(self,state):
    return self.islocked

  # Methods
  def lock(self):
    self.islocked=1
  def unlock(self):
    self.islocked=0
```

Figure 3 Generic message acceptance constrains

4.4.2 Flexible Object Interactions

As we discussed above, a widely accepted approach for overcoming the inheritance anomaly is to keep synchronization code separate from the code of the methods. This works well for a pattern of processing messages where once a message is accepted the requested method

[2] This function is called just once at the creation of the active object class.

[3] Attributes used for synchronization purposes; updated in synchronization actions and red by state predicates.

```
class adaptBuffer(Buffer):

  # Special attributes
  states=['oftenFull']
  activities=['extendSize']
  conditions={
    'extendSize':
    (lambda o:
     o.atState(('oftenFull',)),)}
  receipt_actions={
    'put': 'self.monitorPut'}

  # Synchronization variables
  self.put_cunt=0;
  self.full_count=0

  # Abstract state
  def oftenFull(self,state):
    return (
      self.put_count > self.freq and
      self.full_count/self.put_count
      > self.factor)

  # Synchronization actions
  def monitorPut(self):
    self.put_count=self.put_count+1
    if self.atState(('full',)):
      self.full_count=self.full_count+1

  # Internal activity
  def extendSize(self):
    while 1:
      self.lim=self.lim+self.incr
      self.put_count=0
      self.full_count=0
      self.suspend()
```

Figure 4 Internal activities

executes to completion without any further need for synchronization. However, such an approach fails to provide adequate expressive power for other message processing patterns. In particular, it is hard to address the concurrent programming issues known as *nested monitors calls* [24] and *remote delays* [25].

These issues are illustrated in a concurrent programming pattern known as the administrator [26]. In this programming pattern an object, we call it the *server*, accepts requests from clients and makes a number of (sub)requests to other objects (the *second-level servers*) to process the client's request. While a second-level server processes a (sub)request, the server accepts and processes other clients' requests. The server resumes the processing of a client request when the replies from second-level servers are available. This programming pattern, discussed in [25] and [26], is important for the following reasons:

```
class Server(ActiveObjectSupport):

  # Special attributes
  methods=['request']

  # Initialization
  def __init__(self,server2):
    self.server2=server2

  # Method
  def request(self,date):
    # Create a reply object
    myReply=Reply()
    self.send(
      target=self.server2,
      key='aMethod',args=(),
      replyTo=myReply)
    # Suspend until reply is ready
    self.SuspendUntil(
      myReply.atState(('Ready',)))
    return myReply.getResult()
```

Figure 5 Administrative pattern

- First, if the server can accept further requests while a second-level server processes a subrequest, the server, its clients and the other second-level servers do not have to stay idle waiting for it to complete the subrequest. Other clients' requests that use different resources in the server and need the cooperation of different second-level servers can be processed concurrently.

- Second, the use of this pattern prevents a situation, known in concurrent programming research as *nested monitor call* [24], that may lead to a deadlock. This occurs when a second-level server may only be reached through the server and the second-level server delays a server message.

Figure 5 illustrates a class `Server` that processes `request` messages according to the administrator pattern. In its `request` method, the server first creates an object of class `Reply` that is used to receive the reply to an asynchronous message. Then an asynchronous message is sent to the second-level server, `server2`, using `send`. The `replyTo` argument of send specifies that the reply should be sent to the object `myReply`. Then state notification is used to suspend the thread until `myReply` is at the abstract state `Ready`. At this point the thread is suspended and other ready threads, if any, may be run. The suspended thread will resume after `myReply` is at the state `Ready`, no other threads are running within the object and the object is at a state compatible with the execution of the request method.

```
class Philo(ActiveObjectSupport):

  # Special attributes
  states=['Hungry','Eating']
  methods=[
    'getForks','eat','releaseForks']
  activities=['eatActivity']
  conditions={
    'eatActivity':
    (lambda o: a.atState(('Hungry',)),)}
  post_actions={
    'getForks': ["self.eating=1"],
    'releaseForks': ["self.eating=0"]}

  # State function
  def Eating(self, state):
    return self.eating

  # Activities
  def eatActivity(self):
    while 1:
      self.callAndSuspend(
        target=self.interface,
        key='getForks')
      while self.atState(('Hugry',)):
        self.eat()
      self.interface.releaseForks()
```

Figure 6 Philosopher

When the thread is resumed it retrieves the result from `myReply` and returns it to the client.

4.5 Object Coordination

Object coordination can be supported by dynamically associating abstract states, state predicates and an activation conditions with an object. In this case the state predicate can be an independent object that is used to constrain the invocation of the methods of an existing objects. Another possibility is offered by the state notification mechanism.

4.5.1 Coordination using Shared State Predicates

We illustrate this approach for coordinating the object execution with a version of the dining philosophers problem. The behavior of philosophers is defined in the `Philo` class shown in Figure 6. In its `eatActivity` a philosopher waits until she is `Hungry`. Then, she tries to get the forks by calling its `getForks` method. When she has done it successfully, she eats until she is not `Hungry` anymore, and then releases the forks.

```
class PhiloCoord(ActiveObjectSupport):

  # Initialization
  def __init__(self, philo_list):
    self.philo_list = philo_list
    for philo in self.philo_list:
      philo.newPredObject(
        self.interface, ['MyTurn'])
      philo.addCondidtion(
        {'getForks':
         lambda o:
         o.atState(('MyTurn',))})

  # Reply, but wait until philosopher
  # is eating (has grabbed both forks)
  def evalState(self, object, state):
    self.reply(1)
    self.waitUnitil(
      object.atState('Eating',))
```

Figure 7 Philosophers' coordinator

In its `getForks` method (not shown) a philosopher attempts to grab its left and right forks sequentially. A deadlock can occur if a philosopher gets one fork and the other fork is not available.

The `PhiloCoord` object, shown in Figure 7, coordinates philosophers to provide atomicity in getting the forks: no other philosopher can get a fork before the one trying at the moment has acquired both her forks and started eating. The coordinator calls the `newPredObject` method of each philosopher to define dynamically an abstract state, `MyTurn`, and to specify itself as the state predicate object to be called for telling whether or not the abstract state is true. It also calls the method `addCondition` to associate a new activation condition, that the state `MyTurn` is true, with the `getForks` method of each philosopher.

When a philosopher calls its `getForks` method[4], the ATOM run-time, before accepting the message, calls the `evalState`[5] method of the coordinator to check if the abstract state `MyTurn` for the philosopher is true. In the `evalState` method, the coordinator allows the message to be accepted by replying 1. It then waits until the philosopher (the object argument of `evalState`) is at the state `Eating` before accepting any further calls. This ensures that the actions of acquiring the forks are executed atomically with respect to other philosophers. In fact, the coordinator could use a more complex algorithm to decide if the philosopher should get the forks allowing to schedule the philosophers' actions.

[4] The call is made using non-blocking remote procedure call to avoid the nested monitor call problem.

[5] This is part of the protocol supported by state predicate objects.

The coordination of the philosophers provided by `PhiloCoord` is completely transparent to the philosophers. The dining philosopher example shows that the ATOM model provides synchronization mechanisms which can be added dynamically and used to synchronize existing objects without changing their implementation.

4.5.2 State Notification

The use of state notification was illustrated in section 4.4.2 to coordinate the execution of the server's thread with the `Ready` state of a `Reply` object. More examples illustrating the use of this feature are provided in [23].

5 Discussion

5.1 Achievements of the ATOM Model

5.1.1 Inheritance and Expressive Power

Previous COOP models had limitations with respect to the use of inheritance and sacrificed the expressive power for the sake of inheriting synchronization constraints. In our model synchronization constraints are combined through the use of abstract states with the synchronization of internal activities and message passing features that allow a more flexible method processing pattern. As we have shown in section 4 these features are compatible with the use of inheritance and subsume previous proposals for the specification of synchronization constraints.

5.1.2 Object Coordination

State predicates and the possibility to associate abstract states and activation with an object dynamically provide support for coordination using the same features as used for the specification of synchronization constraints. State notification provides an alternative approach for object coordination which departs from previous work [21, 22] where coordination is based on constraining the acceptance of messages. Coordination may also be based on changes in the state of objects. This approach is more suitable for coordinating the execution of objects in cases where objects encapsulate activities that are not convenient to express as method invocation sequences. For instance, an object that encapsulates a device. In addition, the objects to be coordinated do not have to provide the same message interface. Coordination is based on an abstract view of object behavior in terms of abstract state changes.

5.2 On the Use of Python

The use of Python, and especially the more dynamic features of the language such as the possibility to inspect and modify class attributes, allowed us to develop a prototype of the model without undergoing a major language development effort. The `ActiveObject` function discussed in section 4.1.1 goes through the class hierarchy to construct a class that has the semantics of the ATOM object model. Currently, ATOM is entirely written in Python and it is structured as a class framework. This made it easier to experiment with variations of ATOM's features. For instance, it was possible to experiment with various ways of combining the definition of abstract states and activation conditions of superclasses. We have also tried different approaches for scheduling threads and supporting state notification.

The availability of Python on several platforms and the large number of modules provided in the Python library makes it attractive to experiment with ATOM for the development of concurrent applications. However, the current implementation has performance overhead that are due both to ATOM features and the way threads synchronization is handled in the Python interpreter. We are investigating ways of overcoming these limitations. We expect that the use of some "free threading" Python extension and the use of extension classes for implementing most of ATOM code in C will provide us with major performance improvements. Another issue in using Python in the way discussed in this paper, is how to extend the language with new constructs. In other languages such as Smalltalk the ability of defining blocks of code that can be passed as arguments makes it easy to define new control structures. The presence of such a feature in Python would simplify the specification of activation conditions. We are currently searching for ways of achieving something similar in Python.

6 Conclusion

Although there has been substantial research in COOP and several issues that have to be addressed by COOP models have been identified, most research has addressed particular issues in isolation. We have presented a COOP model, ATOM, that incorporates a number of novel features that aim to address simultaneously several concurrent object-oriented programming issues that have been identified by previous research: synchronized method invocation, reuse of synchronization constraints through inheritance, support for object-coordination and adequate expressive power for concurrent programming.

ATOM has been incorporated in Python and we have shown examples of how it can be used for concurrent object-oriented programming in Python. A detailed presentation of all features of ATOM can be found in [23] and a free prototype implementation is available through anonymous ftp from `fidji.imag.fr:/pub/michael/atom`. The current version does not require any modification to the Python language or interpreter.

The use of Python allowed us to develop a prototype and experiment with the early design of ATOM features in developing example applications. The ATOM prototype is itself implemented as an object-oriented framework in Python. This made it easier to experiment with variations to ATOM's features. The fact that ATOM is in a sense an extension of Python makes it possible to take advantage of the rich set of modules available in the Python library to use the ATOM model to develop concurrent applications in various application domains.

After experimenting with ATOM in the development of simple multimedia programming environment based on active objects, we are now working on a new version. The main aspects under consideration in the new version are: improved performance and distributed execution. This raises the issue of whether we should modify the Python interpreter, construct a preprocessor or search for other possibilities for extending Python with features such as those of ATOM.

References

[1] P. Wegner, "Dimension of object-based language design," in *Proceedings OOPSLA'87*, vol. 22(12) of *SIGPLAN Notices*, (Orlando, Florida), pp. 168–182, ACM, Dec. 1987.

[2] M. Papathomas, "Concurrency in object-oriented programming languages," in *Object-Oriented Software Composition* (O. Nierstrasz and D. Tsichritzis, eds.), Prentice Hall, 1995.

[3] P. America, "Inheritance and subtyping in a parallel object-oriented language," in *Proceedings ECOOP '87* (J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, eds.), vol. 276 of *Lecture Notes in Computer Science*, pp. 234–242, Springer Verlag, 1987.

[4] O. Nierstrasz, "Active objects in Hybrid," in *Proceedings OOPSLA'87* (N. Meyrowitz, ed.), vol. 22(12) of *SIGPLAN Notices*, pp. 243–253, ACM, Dec. 1987.

[5] Y. Yokote and M. Tokoro, "Experience and evolution of concurrent Smalltalk," in *Proceedings OOPSLA '87*, vol. 22(12) of *SIGPLAN Notices*, (Orlando, Florida), pp. 168–182, ACM, Dec. 1987.

[6] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda, "Modelling and programming in an object-oriented concurrent language ABCL/1," in *Object-Oriented Corcurrent Programming* (A. Yonezawa and M. Tokoro, eds.), pp. 55–89, Cambridge, Massachusetts: The MIT Press, 1987.

[7] D. G. Kafura and K. H. Lee, "Inheritance in actor based concurrent object-oriented languages," in *Proceedings ECOOP 89* (S. Cook, ed.), British Computer Society Workshop Series, Cambridge University Press, 1989.

[8] D. Caromel, "Concurrency and reusability: From sequential to parallel," *Journal of Object-Oriented Programming*, pp. 34–42, September/October 1990.

[9] M. Papathomas, G. Blair, and G. Coulson, "A model for active object coordination and its use for distributed multimedia applications," in *ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution* (A. Yonezawa, O. Nierstrasz, and P. Ciacarini, eds.), vol. 924 of *Lecture Notes in Computer Science*, (Bologna, Italy), Springer-Verlag, July 1994.

[10] S. Matsuoka, K. Wakita, and A. Yonezawa, "Analysis of inheritance anomaly in concurrent object-oriented languages (extended abstract)," in *Research Directions in Concurrent Object-Oriented Programming* (G. Agha, P. Wegner, and A. Wonezawa, eds.), MIT Press, 1993.

[11] C. Atkinson, S. Goldsack, A. D. Maio, and R. Bayan, "Object-oriented concurrency and distribution in DRAGOON," *Journal of Object-Oriented Programming*, March/April 1991.

[12] L. Bergmans, *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[13] S. Crespi Reghizzi, G. G. de Paratesi, and S. Genolini, "Definition of reusable concurrent software components," in *Proceedings of ECOOP'91, Geneva, Switzerland*, vol. 512 of *Lecture Notes in Computer Science*, pp. 148–166, Springer Verlag, July 1991.

[14] S. Frølund, "Inheritance of synchronization constraints in concurrent object-oriented programming languages," in *Proceedings ECOOP 92* (O. L. Madsen, ed.), vol. 615 of *Lecture Notes in Computer Science*, (Utrecht), pp. 185–196, Springer Verlag, June/July 1992.

[15] S. Krakowiak, M. Meysembourg, H. N. Van, M. Riveill, C. Roisin, and X. R. de Pina, "Design and implementation of an object-oriented strongly typed language for distributed applications," *Journal of Object-Oriented Programming*, vol. 3, pp. 11–22, September/October 1990.

[16] P. Löhr, "Concurrency annotations for reusable software," *Communications of the ACM*, vol. 36, pp. 8–89, September 1993.

[17] C. McHale, *Synchronization in Concurrent Object-Oriented Langauges: Expressive Power, Genericity and Inheritance*. PhD thesis, University of Dublin, Trinity College.

[18] S. Matsuoka, K. Taura, and A. Yonezawa, "Highly efficient and encapsulated re-use of synchronisation code in concurrent object-oriented languages," in *Proceedings OOPSLA'93*, vol. 28(10) of *ACM SIGPLAN Notices*, pp. 109–129, October 1993.

[19] F. Sanchez *et al.*, "Issues in composability of synchronization constraints in concurrent object-oriented languages," in *Workshop Reader of the 10th. European Conference on Object-Oriented Programming, ECOOP96* (M. Muhlhauser, ed.), Special Issue in Object-Oriented Programming, (Linz, Autria), July 1996.

[20] C. Tomlinson and V. Singh, "Inheritance and synchronization with enabled sets," in *ACM SIGPLAN Notices, Proceedings OOPSLA'89*, pp. 103–112, Oct. 1989.

[21] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa, "Abstracting object interactions using composition filters," in *Prooceedings of the ECOOP'93 Workshop on Object-Based Distributed Programminng* (R. Guerraoui, O. Nierstrasz, and M. Riveill, eds.), vol. 791 of *Lecture Notes in Computer Science*, pp. 152–184, Springer Verlag, 1994.

[22] S. Frølund and G. Agha, "A language framework for multi-object coordination," in *Proceedings ECOOP 93*, vol. 707 of *Lecture Notes in Computer Science*, pp. 346–360, Springer Verlag, July 1993.

[23] M. Papathomas, "ATOM: An active object model for enhancing reuse in the development of concurrent software," Research Report RR 963-I-LSR-2, IMAG-LSR, Grenoble, France, November 1995. (Available through anonymous ftp at fidji.imag.fr:/pub/michael/atom/atom-report.ps.gz).

[24] A. Lister, "The problem of nested monitor calls," *Operating Systems Review*, pp. 5–7, July 1977.

[25] B. Liskov, M. Herlihy, and L. Gilbert, "Limitations of synchronous communication with static process structure in languages for distributed computing," in *Proceedings of the 13th ACM symposium on Principles of Programming Languages*, (St. Petersburg, Florida), 1986.

[26] W. Gentleman, "Message passing between sequential processes: the reply primitive and the administrator concept," *Software-Practice and Experience*, vol. 11, pp. 435–466, 1981.