Active Object

an Object Behavioral Pattern for
Concurrent Programming

R. Greg Lavender
G.Lavender@isode.com
ISODE Consortium Inc.
Austin, TX

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis

An earlier version of this paper appeared in a chapter in the book "Pattern Languages of Program Design 2" ISBN 0-201-89527-7, edited by John Vlissides, Jim Coplien, and Norm Kerth published by Addison-Wesley, 1996.

Abstract

This paper describes the Active Object pattern, which decouples method execution from method invocation in order to simplify synchronized access to a shared resource by methods invoked in different threads of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. A broad class of producer/consumer *and* reader/writer *problems are well-suited to this model of concurrency. This pattern is commonly used in distributed systems requiring multi-threaded servers. In addition, client applications (such as windowing systems and network browsers), are increasingly employing active objects to simplify concurrent, asynchronous network operations.*

1 Intent

The Active Object pattern decouples method execution from method invocation in order to simplify synchronized access to a shared resource by methods invoked in different threads of control.

2 Also Known As

Concurrent Object, Actor, Serializer.

3 Motivation

To illustrate the Active Object pattern, consider the design of a connection-oriented Gateway. A Gateway decouples cooperating components in a distributed system and allows them to interact without having direct dependencies among each other [1]. For example, the Gateway shown in Figure 1 routes messages from one or more source processes to one or more destination processes in a distributed system

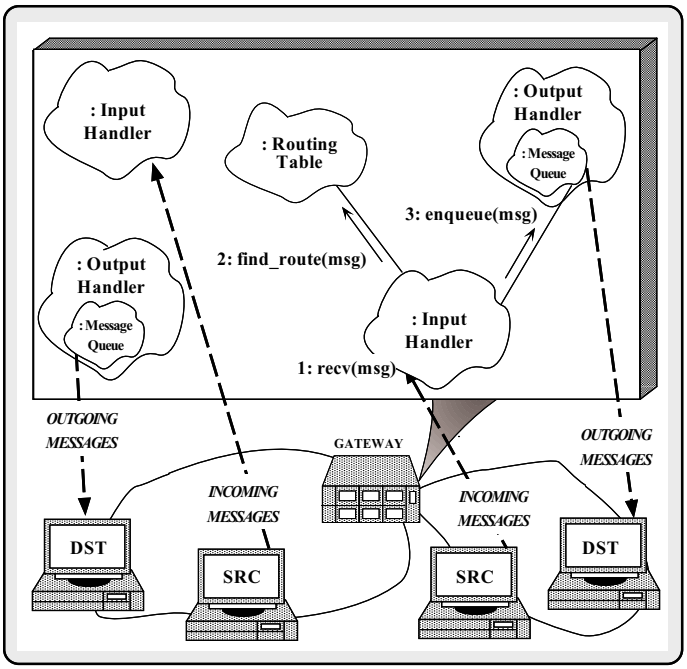


Figure 1: Connection-Oriented Gateway

[2]. Sources and destinations communicate with the Gateway using TCP connections. Internally, the Gateway contains a set of `Input` and `Output Handler` objects. `Input Handlers` receive messages from sources and use address fields in a message to determine the appropriate `Output Handlers` associated with the destination. The `Output Handler` then delivers the message to the destination.

Since communication between the sources, destinations, and Gateway use TCP, `Output Handlers` may encounter flow control from the transport layer. Connection-oriented protocols like TCP use flow control to ensure that a fast source does not produce data faster than a slow destination (or slow network) can buffer and consume the data.

To reduce end-to-end delay, an `Output Handler` object must not block the entire Gateway waiting for flow control to abate on any single connection to a destination. One way to ensure this is to design the Gateway as a single-

threaded reactive state machine that uses asynchronous network I/O. This design typically combines the Reactor pattern [3], non-blocking sockets, and a set of message queues (one per `Output Handler`). The Reactor pattern and the non-blocking sockets provide a single-threaded cooperative event loop model of programming. The Reactor demultiplexes "ok to send" and "ok to receive" events to multiple `Input` and `Output Handler` objects. These handlers use non-blocking sends and receives to prevent the Gateway from blocking. The message queues are used by `Output Handlers` to store messages in FIFO order until they can be delivered when flow control abates.

It is possible to build robust single-threaded connection-oriented Gateways using the approach outlined above. There are several drawbacks with this approach, however:

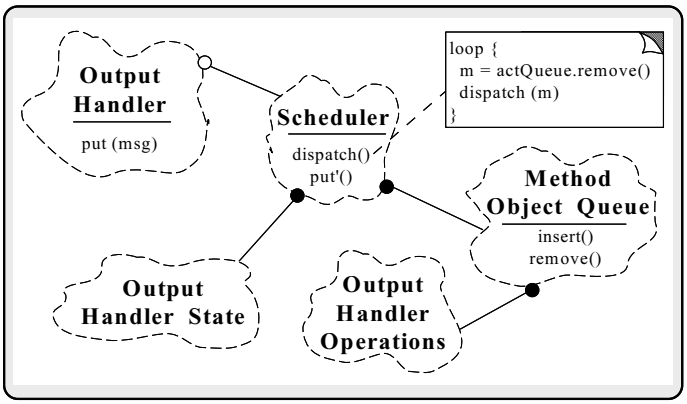
- *Complicated concurrent programming* – subtle programming is required to ensure that `Output Handlers` in the Gateway never block while routing messages to their destinations. Otherwise, one misbehaving output connection can cause the entire Gateway to block indefinitely.
- *Does not alleviate performance bottlenecks* – the use of single-threading does not take advantage of parallelism available from the underlying hardware and software platform. Since the entire Gateway runs in a single thread of control it is not possible to transparently alleviate performance bottlenecks by running the system on a multi-processor.

A more convenient, and potentially more efficient, way to develop a connection-oriented Gateway is to use the *Active Object pattern*. This pattern enables a method to execute in a different thread than the one that invoked the method originally. In contrast, passive objects execute in the same thread as the object that called a method on the passive object.

Implementing `Output Handlers` as active objects in the Gateway enables them to block independently, without adversely affecting each other or the `Input Handlers`. The active object Gateway design resolves the following forces:

- *Simplify flow control* – since an `Output Handler` active object has its own thread of control, it can block waiting for flow control to abate. If an `Output Handler` active object is blocked due to flow control, `Input Handler` objects can still insert messages onto the message queue associated with the `Output Handler`. After completing its current send, an `Output Handler` active object dequeues the next message from its queue. It then sends the message across the TCP connection to its destination.
- *Simplify concurrent programming* – The message queue used by the `Output Handler` active objects allows `enqueue` and `dequeue` operations to proceed concurrently. These operations are subject to synchronization constraints that (1) guarantee serialized access to a shared resource and (2) depend on the state of the resource (*e.g.,* full vs. empty vs. neither). The Active Object pattern makes is simple to program this class of "producer/consumer" application.
- *Take advantage of parallelism* – the Gateway can transparently take advantage of the inherent concurrency between `Input` and `Output Handler` to improve performance on multi-processor platforms. For example, the processing at `Output Handlers` can execute concurrently with `Input Handlers` that pass them messages to be delivered.

The structure of the Gateway application implemented using the Active Object pattern is illustrated in the following Booch class diagram:



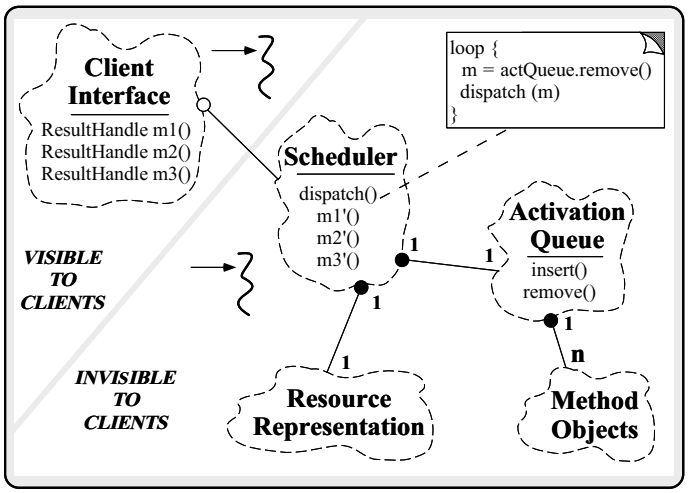
4 Applicability

Use the Active Object pattern when:

- *The design and implementation of a concurrent program can be simplified* – concurrent programs can often be simplified if the thread of control of an object O that executes a method can be decoupled from the thread of control of objects that invoke methods on O.
- *Multiple threads of control require synchronized access to shared data* – the Active Object pattern shields applications from low-level synchronization mechanisms, rather than having them acquire and release locks explicitly.
- *The order of method execution can differ from the order of method invocation* – methods invoked asynchronously are executed based on a synchronization policy, not on the order of invocation.
- *The operations on a shared object are relatively coarse-grained* – in contrast, if operations are very fine-grained the synchronization, data movement, and context switching overhead of active objects may be too high [4].

5 Structure and Participants

The structure of the Active Object pattern is illustrated in the following Booch class diagram:

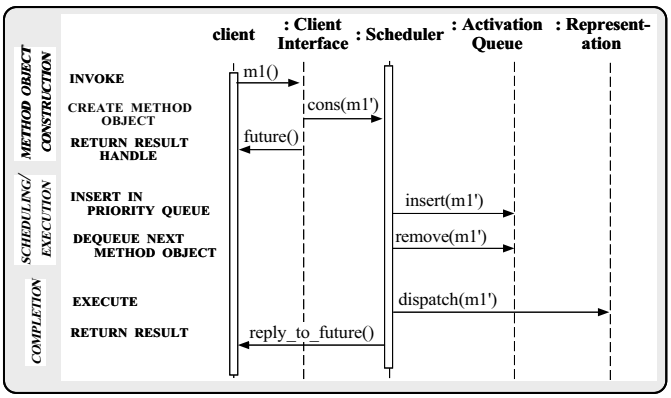


The key participants in the Active Object pattern include the following classes shown below:

- **Client Interface** (`Output Handler Interface`)
  - The Client Interface is a Proxy that presents a method interface to client applications. The invocation of a method defined by the Client Interface triggers the construction and queueing of a Method Object (see next bullet).
- **Method Objects** (`Output Handler Operations`)
  - A Method Object is constructed for any method call that requires synchronized access to a shared resource managed by the Scheduler. Each Method Object maintains context information necessary to (1) execute an operation following a method invocation and (2) to return any results of that invocation through the Client Interface.
- **Activation Queue** (`Method Object Queue`)
  - Maintains a priority queue of pending method invocations, which are represented as Method Objects created by the Client Interface. The Activation Queue is managed exclusively by the Scheduler (see next bullet).
- **Scheduler** (`Method Object Scheduler`)
  - A Scheduler is a "meta-object" that manages an Activation Queue containing Method Objects requiring execution. The decision to execute an operation is based on mutual exclusion and condition synchronization constraints.
- **Resource Representation** (`Output Handler Implementation`)
  - Represents the shared resource that is being modeled as an Active Object. The resource object typically defines methods that are defined in the Client Interface. It may also contain other methods that the Scheduler uses to compute run-time synchronization conditions that determine the scheduling order.
- **Result Handle**
  - When a method is invoked on the Client Interface, a Result Handle is returned to the caller. The Result Handle allows the method result value to be obtained after the Scheduler finishes executing the method.

6 Collaborations

The following figure illustrates the three phases of collaborations in the Active Object pattern:



1. *Method Object construction* – in this phase the client application invokes a method defined by the Client Interface. This triggers the creation of a Method Object, which maintains the argument bindings to the method, as well as any other bindings required to execute the method and return a result. For example, a binding to a Result Handle object returned to the ' caller of the method. A Result Handle is returned to the client unless the method is "oneway," in which case no Result Handle is returned.
2. *Scheduling/execution* – in this phase the Scheduler acquires a mutual exclusion lock, consults the Activation Queue to determine which Method Object(s) meet the synchronization constraints. The Method Object is then bound to the current Representation and the method is allowed to access/update this Representation and create a Result.

3. *Return result* – the final phase binds the Result value, if any, to a *future* [5, 6] object that passes return values back to the caller when the method finishes executing. A future is a synchronization object that enforces "write-once, read-many" synchronization. Subsequently, any readers that rendezvous with the future will evaluate the future and obtain the result value. The future and the Method Object will be garbage collected when they are no longer needed.

7 Consequences

The Active Object pattern offers the following benefits:

- *Enhance application concurrency while reducing synchronization complexity* – especially if objects only communicate via messages.
- *Leverage parallelism available from the hardware and software platform* – if the hardware/software platform supports multiple CPUs this pattern can allow multiple active objects to execute in parallel (subject to their synchronization constraints).

The Active Object pattern has the following drawbacks:

- *It potentially increases context switching, data movement, and synchronization overhead* – depending on how the Scheduler is implemented (*e.g.,* in user-space vs. kernel-space) overhead may occur to schedule and execute multiple active objects
- *It may be difficult to debug programs containing active objects due to the concurrency and non-determinism of the Scheduler* – moreover, many debuggers do not adequately support concurrent programs.

8 Implementation

The Active Object pattern can be implemented in a variety of ways. This section discusses several issues that arise when implementing the Active Object pattern. Section 9 illustrates the steps involved in using the Active Object pattern to implement the connection-oriented Gateway described in Section 3.

- *Separate interface, implementation, and synchronization policies* – A common way to implement a shared resource (such as a message queue) uses a single class whose methods first acquire a mutual exclusion (mutex) lock. The code then proceeds to access the resource, subject to conditional synchronization constraints (*e.g.,* the `dequeue` operation in Section 9 cannot execute when the message queue is empty and the `enqueue` operation cannot execute when the message queue is full).

```
class Message_Queue
{
public:
  // Enqueue message.
  int enqueue (Message *new_msg)
  {
    mutex_.acquire ();

    while (is_full ()) {
      // Release the lock_ and wait for
      // space to become available.
      notFullCond_.wait ();

    // Enqueue the message here...

    mutex_.release ();
  }

private:
  Mutex mutex_;
  Condition<Mutex> notFullCond_;
  // ...
```

A drawback to using this technique is that it embeds code representing the synchronization policy into methods that access the message queue representation. This tight coupling often inhibits the reuse of the resource implementation by derived classes that require specialized or different synchronization policies. This problem is commonly referred to as the *inheritance anomaly* [7, 8, 9, 10].

A more flexible implementation is to decouple the explicit synchronization policy code from the methods that access and update the shared resource. This decoupling requires that the Client Interface be defined separately. It is used solely to cause the construction of a Method Object for each method invocation.

A Method Object is an abstraction for the context (or *closure*) of an operation. This context includes argument values, a binding to the Resource Representation that the operation is to be applied to, a result object, and the code for the operation. Method Objects are constructed when a client application invokes a method on a Client Interface proxy.

Each Method Object is enqueued on a method Activation Queue. A Scheduler that enforces a particular synchronization policy on behalf of a shared resource will compute whether or not a Method Object operation can execute. Predicates can be defined as part of the resource implementation that represent the different states of the resource. Section 9 illustrates this decoupled implementation approach.

- *Determine rendezvous and return value policies* – A rendezvous policy is required since active objects do not execute in the same thread as callers that invoke their methods. Different implementations of the Active Object pattern choose different rendezvous and return value policies. Typical choices include the following:
  - *Synchronous waiting* – block the caller synchronously at the Client Interface until the active object accepts the method call.

– *Synchronous timed wait* – block only for a bounded amount of time and fail if the active object does not accept the method call within that period. If the timeout is zero this scheme is often referred to as "polling."
– *Asynchronous* – queue the method call and return control to the caller immediately. If the method produces a result value then some form of future mechanism must be used to provide synchronized access to the value (or the error status if the method fails).

In the context of the Active Object pattern, a *polymorphic future* pattern may be required [11] for asynchronous invocations that return a value to the caller. A polymorphic future allows parameterization of the eventual result type represented by the future and enforces the necessary synchronization. When a Method Object computes a result, it acquires a write lock on the future and updates the future with a result value of the same type as that used to parameterize the future. Any readers of the result value that are currently blocked waiting for the result value are awakened and may concurrently access the result value. A future object is eventually garbage collected after the writer and all readers no longer reference the future.

- *Leverage off other patterns that support the implementation of the Active Object pattern* –

  The Active Object pattern requires a set of related patterns for different forms of synchronization (such as mutual exclusion, producer-consumer, and readers-writers) and reusable mechanisms for implementing them (such as mutexes, semaphores, and condition variables). A current area of work is to define a collection of reusable building block synchronization patterns to complement the use of the Active Object pattern in a wide set of circumstances.

9 Sample Code

This section presents sample code that illustrates an implementation of the Active Object pattern. The following steps define an Active Object for use as a message queue by `Output Handler` objects in the Gateway described in Section 3.

1. *Define a non-concurrent queue abstraction that implements a bounded buffer* – using an internal representation like a circular array or linked list. This implementation is not concerned with mutual exclusion or condition synchronization. The following `MessageQueueRep` class presents the interface for this queue:

```
// The template parameter T corresponds to
// the type of messages stored in the queue:

template<class T>
class MessageQueueRep
{
public:
  void enqueue (T x);
  T dequeue (void);

  bool empty (void) const;
  bool full (void) const;

private:
  // Internal resource representation.
};
```

   The methods in the `MessageQueueRep`'s representation should not include any code that implements synchronization or mutual exclusion. A key goal of using the Active Object pattern is to ensure that the synchronization mechanisms remain external to the representation. This approach facilitates the specialization of the class representing the resource, while avoiding the inheritance anomaly described in Section 8. The two predicates `empty` and `full` are used to distinguish three internal states: empty, full, and neither empty nor full. They are used by the Scheduler to evaluate synchronization conditions prior to executing a method of a resource instance.

2. *Define a Scheduler that enforces the particular mutual exclusion and condition synchronization constraints* – The Scheduler determines the order to process methods based on synchronization constraints. These constraints depend on the state of the resource being represented. For example, if the `MessageQueueRep` is used to implement an `Output Handler`, these constraints would indicate whether the queue was empty, full, or neither.

   The use of constraints ensures fair shared access to the `MessageQueueRep`. Each method of a `MessageQueueRep` is represented by a class derived from a `MethodObject` base class. This base class defines pure virtual `guard` and `call` methods that must be redefined by a derived class. The type parameter `T` defined in the `MessageQueueScheduler` template is the same type of message that is inserted and removed from the `MessageQueue`.

```
template<class T>
class MessageQueueScheduler
{
protected:
  class Enqueue : public MethodObject
  {
  public:
    Enqueue (MessageQueueRep<T> *rep, T arg)
      : rep_ (rep), arg_ (arg) {}

    virtual bool guard (void) const {
      // Synchronization constraint
      return !rep_->full ();
    }

    virtual void call (void) {
      // Insert message into message queue
      rep_->enqueue (arg_);
    }

  private:
```

```
  MessageQueueRep<T> *rep_;
  T arg_;
};

class Dequeue : public MethodObject
{
public:
  Dequeue (MessageQueueRep<T> *rep, Future<T> &f)
    : rep_ (rep), result_ (f) {}

  bool guard (void) const {
    // Synchronization constraint.
    return !rep_->empty ();
  }

  virtual void call (void) {
    // Bind the removed message to the
    // future result object.
    result_ = rep_->dequeue ();
  }

private:
  MessageQueueRep<T> *rep_;
  // Future message result value
  Future<T> result_;
};
```

Instances of the `MethodObjects` derived classes `Enqueue` and `Dequeue` are inserted into an `ActivationQueue` according to synchronization constraints, as follows:

```
public:

  ... // constructors/destructors, etc.,

  void enqueue (T x) {
    MethodObject *method = new Enqueue (rep_, x);
    queue_->insert (method);
  }

  Future<T> dequeue (void) {
    Future<T> result;

    MethodObject *method = new Dequeue (rep_, result);
    queue_->insert (method);
    return result;
  }

  // These predicates can execute directly since
  // they are "const".

  bool empty (void) const {
    return rep_->empty ();
  }
  bool full (void) const {
    return rep_->full ();
  }

protected:
  MessageQueueRep<T> *rep_;
  ActivationQueue *queue_;
```

The `MessageQueueScheduler` object executes its `dispatch` method in a thread of control that is separate from client applications. Within this thread the `ActivationQueue` is searched. The Scheduler selects a `MethodObject` whose `guard` (corresponding to a condition synchronization constraint) evaluates to "true." This `MethodObject` is then executed.

As part of method execution, a Method Object receives a run-time binding to the current representation of the `MessageQueueRep` object (this is similar to providing a "`this`" pointer to a sequential C++ method). The method is then executed in the context of that representation. The following code illustrates how the `MessageQueueScheduler` dispatches Method Objects:

```
  virtual void dispatch (void) {
    for (;;) {
      ActivationQueue::iterator i;

      for (i = queue_->begin();
           i != queue_->end();
           i++) {
        // ...
        // Select a Method Object 'm'
        // whose guard evaluates to true.
        m = queue_->remove ();
        m->call();
        delete m;
      }
    }
  }
};
```

In general, a Scheduler may contain variables that represent the synchronization state of the shared resource. The variables defined depend on the type of synchronization mechanism that is required. For example, with reader-writer synchronization, counter variables may be used to keep track of the number of read and write requests. In this case, the values of the counters are independent of the state of the shared resource since they are only used by the scheduler to enforce the correct synchronization policy on behalf of the shared resource.

3. *Define a Client Interface called* `MessageQueue` – A `MessageQueue` is a `MethodObject` factory that constructs instances of methods that are sent to the `MessageQueueScheduler` for subsequent execution.

   If the synchronization conditions enforced by the `MessageQueueScheduler` prohibit the execution of a `MethodObject` when a method is invoked, the object is queued for later activation. In some cases, an operation may not create a `MethodObject` if it is not subject to the same synchronization constraint as other operations (*e.g.,* the "const" methods `empty` and `full` shown above). Such operations can be executed directly without incurring synchronization or scheduling overhead.

   If a method in the Client Interface returns a result `T`, a `Future<T>` is returned to the application that calls it. The caller may block immediately waiting for the `Future` to complete. Conversely, the caller may evaluate the `Future`'s value at a later point by using either an implicit or explicit type conversion of a `Future<T>` object to a value of type `T`.

```
template <class T>
class MessageQueue
{
public:
  enum { MAX_SIZE = 100 };
  MessageQueue (int size = MAX_SIZE) {
```

```
    sched_ = new MessageQueueScheduler<T> (size);
  }

  // Schedule enqueue to run as an active object.
  void enqueue (T x) { sched_->enqueue (x); }

  // Return a Future<T> as the ``future'' result
  // of an asynchronous dequeue operation.

  Future<T> dequeue (void)  {
    return sched_->dequeue ();
  }

  bool empty (void) const { sched_->empty (); }
  bool full (void) const { sched_->full (); }

private:
  MessageQueueScheduler<T> *sched_;
};
```

A `Future<T>` result can be evaluated immediately by a client, possibly causing the caller to block. For example, a Gateway `Output Handler` running in a separate thread may choose to block until new messages arrive from `Input Handlers`.

```
// Make an MessageQueue specialized for the Gateway.
  typedef MessageQueue<RoutingMessage>
          MESSAGE_QUEUE;

  MESSAGE_QUEUE messageQueue;

  // ...

  // Type conversion of Future<Message> result
  // causes the thread to block pending result
  // of the dequeue operation.
  Message msg = messageQueue.dequeue ();

  // Transmit message to the destination.
  sendMessage (msg);
```

Alternatively, the evaluation of a return result from an Active Object method invocation can be delayed. For example, if no messages are available immediately an `Output Handler` can store the `Future<T>` return value from `messageQueue` and perform other "bookkeeping" tasks (such as exchanging "keepalive messages" to make sure its destination is still active). When it's done with these various tasks it may choose to block until a message arrives from an `Input Handler`, as follows:

```
  // Does not block
  Future<Message> future = messageQueue.dequeue ();

  // Do something else here...

  // Evaluate future by implicit type conversion --
  // may block if the result is not yet available.
  Message msg = future;
```

10 Known Uses

The Active Object pattern is commonly used in distributed systems requiring multi-threaded servers. In addition, the Active Object pattern is used in client applications such as windowing systems and network browsers that employ multiple active objects to simplify concurrent programs that perform non-blocking network operations.

The Gateway example from Section 3 is based on the communication services portion of the Motorola Iridium project. `Output Handler` objects in Iridium Gateways are implemented as active objects to simplify concurrent programming and improve performance on multi-processors. The active object version of the Iridium Gateway uses the pre-emptive multi-tasking capabilities provided by Solaris threads [12]. An earlier version of the Iridium Gateway [2] used a reactive implementation described in Section 3. The reactive design relied on a cooperative event loop-driven dispatcher within a single thread. This design was more difficult to implement and did not perform as well as the active object version on multi-processor platforms.

The Active Object pattern has also been used to implement Actors [13]. An Actor contains a set of instance variables and behaviors that react to messages sent to an Actor by other Actors. Messages sent to an Actor are queued in the Actor's message queue. In the Actor model, messages are executed in order of arrival by the "current" behavior. Each behavior nominates a replacement behavior to execute the next message, possibly before the nominating behavior has completed execution. Variations on the basic Actor model allow messages in the message queue to be executed based on criteria other than arrival order [14]. When the Active Object pattern is used to implement Actors, the Scheduler corresponds to the Actor scheduling mechanism, Method Objects correspond to the behaviors defined for an Actor, and the Resource Representation is the set of instance variables that collectively represent the state of an Actor [15]. The Client Interface is simply a strongly-typed mechanism used to pass a message to an Actor.

11 Related Patterns

The Mutual Exclusion (Mutex) pattern is a simple locking pattern that can occur in slightly different forms (such as a spin lock or a semaphore) and can have subtle semantics (such as recursive mutexes and priority mutexes).

The Consumer-Producer Condition Synchronization pattern is a common pattern that occurs when the synchronization policy and the resource are related by the fact that synchronization is dependent on the state of the resource.

The Reader-Writer Condition Synchronization pattern is a common synchronization pattern that occurs when the synchronization mechanism is not dependent on the state of the resource. A readers-writers synchronization mechanism can be implemented independent of the type of resource requiring reader-writer synchronization.

The Future pattern describes a typed future result value that requires write-once, read-many synchronization. Whether a caller blocks on a future depends on whether or not a result value has been computed. Hence, the future pattern is a hybrid pattern that is partly a reader-writer condition

synchronization pattern and partly a producer-consumer synchronization pattern.

The Half-Sync/Half-Async pattern [16] is an architectural pattern that decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency. This pattern typically uses the Active Object pattern to implement the Synchronous task layer, the Reactor pattern [3] to implement the Asynchronous task layer, and a Producer/Consumer pattern to implement the Queueing layer.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[2] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[3] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[4] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[5] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 501–538, Oct. 1985.

[6] B. Liskov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 260–267, June 1988.

[7] P. America, "Inheritance and Subtyping in a Parallel Object-Oriented Language," in *ECOOP'87 Conference Proceedings*, pp. 234–242, Springer-Verlag, 1987.

[8] D. G. Kafura and K. H. Lee, "Inheritance in Actor-Based Concurrent Object-Oriented Languages," in *ECOOP'89 Conference Proceedings*, pp. 131–145, Cambridge University Press, 1989.

[9] S. Matsuoka, K. Wakita, and A. Yonezawa, "Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages," *OOPS Messenger*, 1991.

[10] M. Papathomas, "Concurrency Issues in Object-Oriented Languages," in *Object Oriented Development* (D. Tsichritzis, ed.), pp. 207–245, Centre Universitaire D'Informatique, University of Geneva, 1989.

[11] R. G. Lavender and D. G. Kafura, "A Polymorphic Future and First-Class Function Type for Concurrent Object-Oriented Programming in C++," in *Forthcoming*, 1995. http://www.cs.utexas.edu/users/lavender/papers/futures.ps.

[12] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[13] G. Agha, *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[14] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled-Sets," in *OOPSLA'89 Conference Proceedings*, pp. 103–112, Oct. 1989.

[15] D. Kafura, M. Mukherji, and G. Lavender, "ACT++: A Class Library for Concurrent Programming in C++ using Actors," *Journal of Object-Oriented Programming*, pp. 47–56, October 1992.

[16] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, September 1995.