Computación y Sistemas Vol. 5 No.1 pp. 54 - 66
© 2001, CIC - IPN. *ISSN 1405-5546* Impreso en México

Pattern-Based Simulation: Simulating the Actor Model Using the Active Object Behavioural Pattern

Simulación Basada en Patrones: Simulando el Modelo de Actores Utilizando el Patrón del Objeto Activo

Jorge Luis Ortega-Arjona and **Graham Roberts**
Departament of Computer Science, University College London
Gower Street, London, WC1E 6BT, U.K. Tel: +44 171419 36 79
e-mail : {J.Ortega-Arjona, G.Roberts}@cs.ucl.ac.uk

Article received on October 10, 1999, accepted on February 25, 2001

Abstract

This paper proposes the use of software patterns as a viable base tools for simulating and analysing expected attributes of specific software systems. Particularly, in this paper the Active Object pattern is considered, showing how its combination with a stochastic simulation technique produces a simulation model. This simulation model reflects the probably resulting behaviour in time of an active object, and thus, it can be used as reference for forecasting and analysing the performance behaviour of the active object through time. The simulation model has as input the Active Object Behavioral pattern and elements of Queuing Theory, and produces as output estimates about the active object's performance behaviour.

Keywords: Software Patterns, Simulation Models, Active Object, Queuing Theory, Performance Behaviour.

Resumen

Este artículo propone el uso de patrones de software como herramientas de base viables para simular y analizar atributos deseables de sistemas de software específicos. Particulamente, en este artículo consideramos el patrón de Objeto Activo, mostrando cómo su combinación con una técnica de simulación estocástica produce un modelo de simulación. Este modelo de simulación refleja el probable comportamiento resultante del objeto activo en el tiempo, y por tanto, puede ser usado como referencia para pronosticar y analizar el comportamiento de desempeño del objeto activo a través del tiempo. El modelo de simulación tiene como entrada el patrón de comportamiento del Objeto Activo y elementos de Teoría de Colas, y produce como salida estimaciones acerca del comportamiento de desempeño del objeto activo.

Palabras Clave: Patrones de Software, Modelos de Simulación, Objeto Activo, Teoría de Colas, Comportamiento de Desempeño.

1 Introduction

Concurrent software design is a complex activity, aiming for performance improvement while diminishing the costs of concurrent software development. It requires extra effort from the software designer, who has to balance between these two conflicting design issues. Thus, it would be highly valuable and advantageous for the software designer to count with a method or model that could assist him/her on estimating and forecasting the performance properties of a concurrent program.

Software Patterns are proposed as new techniques that describe useful solutions for software design. Moreover, they potentially present the criterion of simulatability: they might be a viable base to simulate the behaviour of a resulting software system, using its information about structure and behaviour of the solution. The motivation for this paper is precisely to test such criterion of simulatability for concurrent systems, in order to estimate the performance behaviour of a concurrent program as the important attribute of interest. Hence, this paper proposes a pattern-based simulation to estimate the performance properties of a concurrent program. More specifically, the Active Object pattern, which was initially proposed for the design and implementation of active objects (Lavender & Schmidt, 1996), is used as base to simulate the performance of active objects.

Nevertheless, in order to use the Active Object pattern to describe the temporal performance of an actor, it is necessary to express its behaviour in time terms. Queuing Theory and simulation techniques are used to accomplish this, introducing time parameters for the activities performed by the actor. Then, the performance of an actor is simulated by adding to the pattern description the information about the actor's behaviour parameters in time. The expected output of the simulation are estimates about the actor's performance while executing a task.

In general, there are no other clear approaches that intend to obtain performance forecasting or analysis based on software patterns information. The paper by Smith and Williams (1993) consider using software patterns for improving the design of a software system, but it does not precise or identify a relation between the performance of a software system and the software pattern used to design it.

The present paper is organised as follows: first, a brief introduction to the Actor Model is presented to outline its basic characteristics and components; second, the Active Object pattern is described, mentioning its principal elements, and commenting on the relation between its structure and behaviour with the Actor Model; third, an introduction to the Active Object Simulation Model is made, using concepts from stochastic and queuing modelling. Using these concepts and the behaviour of the Active Object pattern as a base, the implementation of a simulation model is developed. Finally, in order to validate this simulation model, an experiment is carried out, using a simple actor program as example.

2 An Introduction to the Actor Model

The objective of this section is to only provide an introduction to the basic concepts and characteristics of the Actor Model. These concepts and characteristics are used to understand the basic theory of concurrent programming using objects. They are re-taken later in the following section, when describing the Active Object pattern as an actual implementation of the Actor Model.

2.1 General Description of the Actor Model

Traditional objects encapsulate a state and an expected behaviour, and provide an interface defined as the names of procedures that are visible. These procedures, often called methods, manipulate the state of the object when invoked.

The interface is a representation of the functionality of the object. Interfaces representing the same functionality may be interchanged transparently.

The concept of actor (or active object) has been developed as an extension to traditional Object-Oriented (OO) programming, aiming to eliminate the limit imposed by considering programming as a sequence of actions. Commonly, OO languages are sequential due to they allow only one object to be active at a precise time during program execution. The behaviour of an object is then the result of a sequence of actions, which may be blocked by invoking methods in another object. Actors are proposed as the underlying basic building blocks for concurrent programming, due to they are a more natural representation of objects as computational elements.

Based on the Actor Model (Agha et al., 1993a; Agha et al., 1993b), an actor application consists of a collection of asynchronous objects that execute concurrently. Actors are autonomous and concurrent objects, executing at their own rate, and able to communicate by passing each other messages.

Since they are conceptually distributed, communications between them is asynchronous, preserving the available potential for concurrent activity: an actor sending a message asynchronously does not need to block until the receiver is ready to receive or process a message. If a sender object is required to block, the available potential concurrency is reduced.

2.2 Message Passing

Actors communicate exclusively by sending messages, invoking each other's methods. Message passing is the only means of inter-object communication; there is no shared memory notion between actors. Message passing between actors present the following characteristics (Frolund, 1996):

- Messages are asynchronous. Sending a message is a non-blocking operation.
- Messages are guaranteed to reach eventually their destination, but subject to arbitrary communication delays.
- Message ordering is not guaranteed; messages may not arrive in the order that they were sent.
- A message invokes a method in its destination object; we say that messages are dispatched into method invocations. Objects are reactive entities that execute their methods only in response to messages.

2.3 Structure of the Actor Model

Since each actor has one thread of control, which is used to execute methods in response to messages, at most one method can be executed at any time by the actor. The Actor Model, as originally described by Agha et al., (1993b), contains the notion of internal concurrency, allowing multiple methods to execute concurrently within an object. However, the semantics of the model guarantees serializability, and the overall effect of internal concurrency is made equivalent to execute the methods one at a time. Therefore, it may be considered that there is no concurrency within an object, focusing only on inter-object concurrency. As an actor is considered only to execute one method at a time, it can be modelled as a structure based on a message delivery mechanism, an input queue, a scheduling mechanism, one or several methods, and a state. Figure 1 illustrates a proposed basic structure of an actor with these characteristics.

Using this structure, an actor is composed of elements described as follows:

- *Message delivery*. A message passing mechanism that can be used as an interface to deliver and accept messages to be processed by the actor.

* *Input queue*. Messages received by the actor are queued in the actor's input queue, and stored until they can be dispatched.

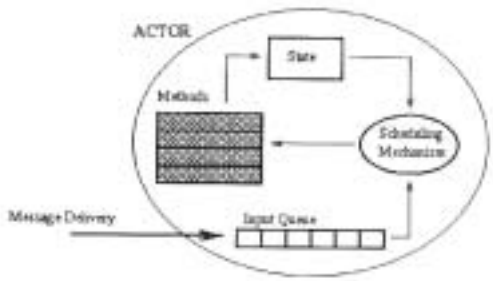


Figure 1: Components of an actor

* *Scheduling Mechanism*. In the original Actor Model (Agha et al., 1993b), the scheduling mechanism is simple: messages are executed in order of arrival by the current behaviour, and each behaviour nominates a replacement behaviour to execute the next message. The new behaviour can be nominated possibly before the nominating behaviour has completed its execution. Scheduling mechanism variations allow messages in the input queue to be executed based on criteria other than arrival order.
* *Methods*. The different behaviours defined for an actor's response to messages.
* *State*. The set of instance variables that collectively represent the state of an actor. As in the traditional object model, an actor's state is modified by its methods, reacting to messages received from other actors.

This completes a brief introduction to the basic concepts, characteristics, and components of the Actor Model. A more extensive and complete analysis of the description and behaviour of the Actor Model can be found in Agha et al., (1993a), Agha et al., (1993b), and Frolund (1996).

3 Software Patterns and the Active Object Pattern

3.1 Software Patterns

Software patterns are becoming popular in the OO programming community, proposing a particular language for software design. The software patterns practitioners, known as the Pattern Community, focus on develop and communicate the most successful practices of expert OO programmers, through an iterative and incremental method (Gamma et al., 1995; Buschmann et al. 1996; Fowler, 1996).

Briefly, a software pattern is considered to be a three element rule, relating a *problem* (in the form of a system of forces) and a *context* (in which the problem occurs) with a *solution* (a software configuration, described as a structure, with an associated behaviour, which allows the forces to resolve themselves) (Gabriel, 1996; Buschmann et al., 1996).

Software patterns are expressed using several forms. The most commonly used forms are the GoF ("Gang of Four") form (Gamma et al., 1995) and the POSA (Pattern-Oriented Software Architecture) form (Buschmann et al., 1996). This last form has become popular among software patterns practitioners with an engineering background, and expresses a software pattern usually in terms of sections such as *brief*, *context*, *problem*, *forces*, *solution*, *structure*, *participants*, *dynamics*, *consequences*, *known uses* and *related patterns*.

3.2 The Active Object Pattern

In the following sections, the Active Object pattern is presented, based on the original version proposed by Lavender and Schmidt (1996), and using the POSA form. This pattern is actually used as an OO design intended to implement actors. For our actual purposes of simulating the behaviour of an actor, we only present some of the relevant details about the Active Object pattern.

3.2.1 Brief

The Active Object pattern describes how to decouple method execution from method invocation in order to simplify synchronised access to a 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 modelled as a single object. A method can be executed in a thread of control separate from the one that originally invoked it, in contrast with passive objects, which execute in the same thread as the object that called a method on them (Lavender & Schmidt, 1996).

3.2.2 Context

Use the Active Object pattern during the design and implementation of a concurrent program (Lavender & Schmidt, 1996).

3.2.3 Problem

The execution of several objects is required in a concurrent program. Each object is expected to execute at its own rate, using an individual thread of control and communicating with other objects. As several threads of control are executed simultaneously, each object has to guarantee a synchronised execution of its methods, controlling the access to its state by methods invoked in different threads of control (Lavender & Schmidt, 1996).

Forces

The active object design resolves the following forces (Lavender & Schmidt, 1996):

* The design and implementation of a concurrent program can be simplified. Concurrent programs can often be simplified by decoupling the thread of control of objects that invoke a method from the thread of control of the object, which actually executes that method. Active objects are based on a message queue structure that allows operations to proceed concurrently. Operations are scheduled according with synchronisation constraints that guarantee a serialised access to a data resource, and depend on the state of the resource.
* Multiple threads of control require synchronised access to a data resource. The Active Object pattern helps to avoid dealing explicitly with low-level synchronisation mechanisms among multiple threads of control accessing a data resource. As an active object has an individual thread of control, it can block but messages can still be inserted onto its associated message queue. After completing its current activity, the active object dequeues the next message from its message queue, and continues.
* The order of method execution may be different from the order of method invocation. As message order is not guaranteed and messages may not arrive in the order that they were invoked, methods are usually scheduled and executed based on a synchronisation policy, and not on the order of invocation.
* Software programs based on active objects can take advantage of the inherent concurrency of multiprocessor platforms to improve performance.

3.2.4 Solution

A collection of objects is proposed to perform the activities of the Active Object pattern. When a message is issued to an active object, it is received through its *client interface*, which accepts messages to be processed by the active object. The *scheduler*, a scheduling mechanism, is in charge to queue incoming messages in the active object's associated *activation queue*. In this queue, messages are stored in the form of method objects until they can be dispatched. A *method object* is a representation of the method invoked by a message. In general, messages are dispatched based on an arrival criteria. When a message can be dispatched, the scheduler removes its associated method object from the activation queue and invokes the real method in the object resource representation.

This performs the expected behaviour in response to the message, manipulating the instance variables that represent the state of the active object (Lavender & Schmidt, 1996).

Class diagram and participants

The class diagram of the Active Object pattern proposed in (Lavender & Schmidt, 1996) is illustrated in figure 2, using the UML notation .

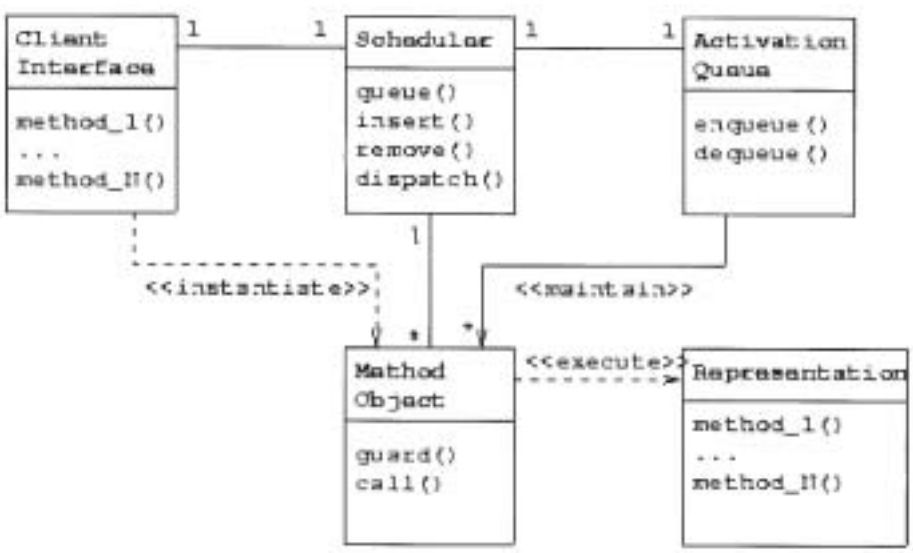


Figure 2: Class diagram of the components of a generic active object

The participants of the Active Object pattern are (Lavender & Schmidt, 1996):

- *Client interface*. The client interface is a method interface presented to client applications. When a method defined by the client interface is invoked, this triggers the construction and queuing of a method object.
- *Method objects*. A method object is constructed by the scheduler for any input message requiring a synchronised method execution. Each method object contains the context information necessary to execute an invoked method operation and return any result of that execution through the client interface.
- *Activation queue*. The activation queue is a priority queue, storing input messages as method invocations represented by method objects. The activation queue is controlled and managed by the scheduler.
- *Scheduler*. The scheduler is an object that manages the activation of method objects requiring execution. It is in charge of inserting and removing method objects from the activation queue, and deciding which method object is to be executed at certain time. The execution of a method object is based on mutual exclusion and condition synchronisation constraints.
- *Resource representation*. The resource representation object is the implementation of the methods defined in the client interface. It represents the resource modelled as an active object. It may also contain other methods used by the scheduler to compute runtime synchronisation conditions that determine the execution order.

When the Active Object pattern is used to implement actors, the actor's scheduling mechanism corresponds to the scheduler, the methods or behaviours defined in the actor correspond to the method objects, and the set of instance variables that collectively represent the state of an actor is the resource representation. The actor's input queue corresponds to the activation queue, and the client interface corresponds to the message delivery mechanism, as a strongly typed mechanism used to pass messages to the actor.

Dynamics

Figure 3 illustrates the behaviour of the Active Object pattern, described in terms of interactions among its components (Lavender & Schmidt, 1996).

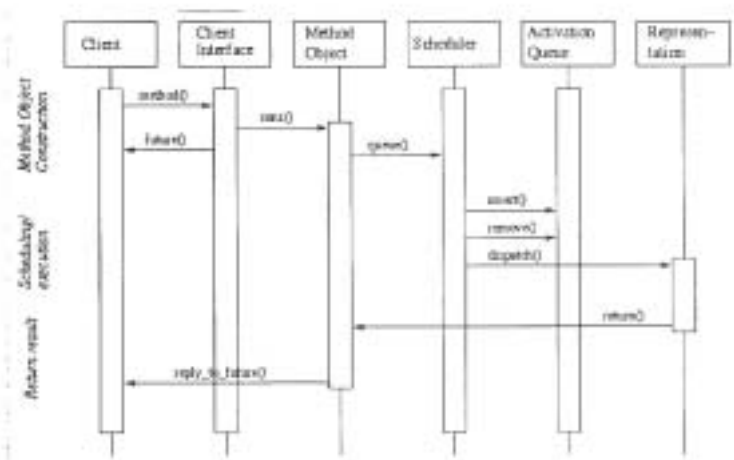


Figure 3. Interactions among the internal components of an active object

1. *Method object construction*. In this stage, the client application invokes a method defined by the client interface. This triggers the construction a method object, which maintains context information about the method as well as any other required to execute the method and return a result. After its creation, the method object requests the scheduler to be queued on the active object's activation queue, waiting for its eventual execution. The scheduler inserts it and a result handle, or future, is returned to the client.

2. *Scheduling/execution*. In this phase, the scheduler consults the activation queue to determine which method object matches established synchronisation constraints. The scheduler removes the method object from the activation queue, and calls the resource representation to dispatch it. An invocation is produced to the actual method of the resource representation with the information contained in the method object. The method is executed, accessing and updating the state of the resource representation to create a result.

3. *Return result*. Finally, the result value is returned when the method finishes executing. Using again the information contained in the method object, the result is passed to the future that returns it to the client. The future and method object involved will be garbage-collected when they are no longer needed.

4 Introducing Time Parameters

In order to develop a valid simulation model for the Active Object, it is necessary to precisely define the attribute of interest that the simulation model is supposed to reflect. In this case, the following definition of performance is considered: *"Performance refers to the responsiveness of the system - the time required to respond to stimuli (events) or the number of events processed in some interval time"* (Bass et al., 1998; Smith & Williams, 1993). A suitable simulation model for the performance of an active object, then, should reflect the active object's behaviour when responding to messages through time. Thus, the behaviour depends on the kinds of messages received by the active object, and by the messages exchanged among the participant objects of active object. For the simulation model, the message reception and exchange are simulated using stochastic models.

4.1 Stochastic Models

When trying to describe the time behaviour of an active object exceeds the simple parameters of structure (represented by the Active Object pattern), it is possible to address the underlying stochastic nature of message passing as part of an initial description. Requests for service arriving at the active object's message queue often can be modelled as a random process. The amount of computation required by a process (or job) can also be commonly modelled as a random variable.

Queuing Theory is an area of mathematics that encompasses the set of analytical models that most adequately describe this kind of systems (Law & Kelton, 1991; Lazowska et al., 1984). The Queuing Theory analyses queuing structures in many areas where real systems are very complex mechanisms and tractable mathematical models must often be simplified approximations to the real system.

This section analyses a fundamental queuing model that has application to the analysis of software systems, and may form the basis for a more advanced active object simulation model. Therefore, from the Actor Model and the Active Object pattern descriptions, it is clear that a queuing model used for simulating active objects should present the following elements (Law & Kelton, 1991; Stone et al., 1975):

* *An arrival mechanism.* In general, this is a stochastic process that generates requests to be serviced by the active object, representing the time between the arrival of different requests, or inter-arrival time. The model discussed here assumes that the interarrival times are random variables, drawn from an arrival-time probability distribution function.
* *Service mechanisms.* After a request arrives, the primary objective of the active object is to service it. This service requires some time, and like interarrival times, the time to service a request or service time can also be modelled as random variables with a service-time probability distribution function.
* *Queuing discipline.* When requests for service arrives at an active object faster that they can be serviced, a line or queue forms, and a policy is needed to determine the order in which outstanding requests will be processed.

In the following section, we present the modelling of each one of these elements, in order to later develop a simple enough model that represents the active object's behaviour.

4.1.1 Interarrival - Time Distribution. The Poisson Arrival Process

The simplest arrival mechanism to mathematically model interarrival times is the Poisson (completely random) arrival process. The most important property of this process is that events are taken from a very large population, where each member is independent of the others. This means that, for our purposes, the arrival at a present instant does not depend on the arrival or non-arrival at past or future instants. This lack of dependence on the past and future is commonly called *Markovian or memoryless property* (Law & Kelton, 1991; Lazowska et al., 1984). The simplicity of the mathematical analysis of the Poisson arrival process relies precisely on this property.

To analyse a Poisson process, we begin by letting λ be the average arrival rate of the Poisson process. The fundamental assumption that during a gap of time δt an arrival is independent of all other arrivals can be stated with the following two postulates (Lazowska et al., 1984; Stone et al., 1975):

* The probability of an arrival between the epochs t and $t+\delta t$ is $\lambda\delta t + o(\delta t)$, where $o(\delta t)$ denotes a quantity of smaller order of magnitude than δt.
* The probability of more than one arrival between epochs t and $t + \delta t$ is $o(\delta t)$.

From these postulates, it is possible to mathematically derive a function for $P_n(t)$, the probability of n arrivals during an interval of duration t. The procedure to obtain this expression is detailed by Stone et al. (1975).

$$P_n(t) = \frac{(\lambda \cdot t)^n}{n!} \cdot e^{-\lambda \cdot t}$$

From this expression, it can be observed that the exponential function is the only function that can be used to model the required Markovian property of the Poisson arrival process.

4.1.2 Service-Time Distributions. The Exponential Service-Time Distribution

The same consideration, about the memoryless or Markovian property that the Poisson process enjoys, is present in Service-

Time distributions. Let μ be the average rate of service completions by an active object. Making a similar assumption to the one used in the Poisson arrival process, consider f(t) as the probability of the completion of service between epochs t and t+δt be $\mu\delta t + o(\delta t)$ (Lazowska et al., 1984; Stone et al., 1975). Therefore, an expression representing such service probability presents the following form:

$$f(t)=\mu\cdot e^{-\mu\cdot t} \quad t>0$$

The procedure to obtain this expression is also presented by Stone et al. (1975). Again, it can be observed that using the exponential function as the Service-Time distribution of an active object maintains the required Markovian property.

4.1.3 Simple Queuing Structure: a Single Active Object with Exponential Interarrival and Service Times

The queue structure proposed to simply model the time behaviour of an active object is simple: a Poisson arrival process, and a single active object with exponential service time. Figure 4 represents this case.

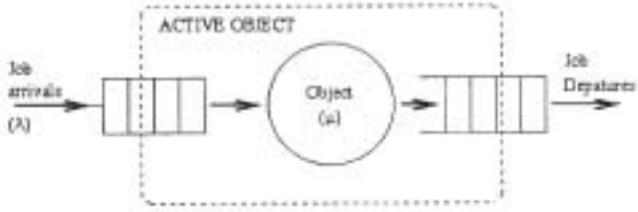


Figure 4: Simple Queuing Structure: a Single Active Object with Exponential Interarrival and Service Times

The fact that the model is simple does not imply it is of little use. On the contrary, this model should be considered as a good initial approximation to the modelling of an active object.

The basic assumptions are: the arrival of messages forms a Poisson process with and average arrival rate of λ messages per second; the processing time per message is an exponentially distributed random variable with a service time average μ messages per second; and for simplicity, a first-in, first-out (FIFO) queue discipline is considered. Using this information, the model must answer a number of questions when describing an active object time behaviour, for instance:

- How much time can the active object spend processing a number of messages?
- What fraction of time will the actor be idle?
- What is the average response time seen that requests are handled by the active object?

To analyse this simple queuing structure, and answer the previous questions, it is necessary to develop an expression (in terms of λ and μ) for the probability of an active object to be in certain state. This expression should be based on the basic considerations of the Poisson process and the exponential service-time distribution. Let p be the probability of the active object being in state E at epoch t, that is, having n messages in service or waiting for service. Considering the long-term or steady-state behaviour of the active object and normalising to obtain the probability in terms exclusively of λ and μ, the following expression for p is obtained. The detailed procedure to obtain this expression is developed by Stone et al. (1975).

$$p_n=\rho^n\cdot(1-\rho) \quad n=1,2,3...$$

It is possible to make some observations about this result. For instance, observe that this equation is undefined when ρ=1. Furthermore, as the analysis of this queuing structure is performed for the steady state behaviour, this equation is meaningful only for ρ<1: as ρ is defined as the relation λ/μ, when λ>μ requests arrive at a faster rate than the active object can service them. This means that there is no steady state solution for ρ>1, because the arrival process is considered to saturate the active object, and its message queue grows without bound.

The ratio ρ has an important role in general Queuing Theory, and it is commonly referred as the traffic intensity of the queuing system.

Applying this result to the questions about the active object's performance, it can be observed that the expression for p is considered to directly answer the first question about the probable time required to process a number of messages. The second question can be answered considering that the active object is idle when $n = 0$. Since p indicates the active object has no outstanding messages to process it is idle probably for (1-ρ) of the time. The third question can be answered by the important fact that the actor is NOT idle with probability ρ. Thus, in any single active object queuing structure, the average response or utilisation of the active object equals to the ratio of the arrival rate to the service rate.

This completes out an introduction to the stochastical models for a simple active object modelling. However, many of the assumptions made here seem to be quite restrictive, and there is the question of whether they can be considered approximations to the behaviour of real active object systems. Other possible assumptions that can be taken into consideration are details about the platform (hardware and software) on which the model is executed, the number and complexity of the operations that the active object is supposed to execute, and the programming language used for coding. Nevertheless, as our objective is to test the ability of the Active

Object pattern to simulate the Actor Model, for our actual purposes these other assumptions are considered fixed.

5 Simulation Model

The use of simulation techniques to analyse a system is certainly advantageous, especially when the system is not simple enough to be analysed exclusively with queuing models. Simulation techniques are used to study phenomena, ranging from flight dynamics of aircrafts to theories of cognitive processes.

The type of simulation considered to be appropriate for the study and modelling of software systems is the discrete-event simulation (Law & Kelton, 1991). Its most important feature is precisely that time is not considered a continuous variable incremented by uniform intervals. The execution of an event in the model is represented by only updating the state of the simulation to reflect the occurrence of the event. After the event is occurred, the simulation is advanced to the time of the next event, and the process is repeated. Usually, when simulating a queue structure, its parameters (inter-arrival time of jobs, length of processing time required by jobs, and so on) are considered as random variables. Therefore, an important point for a discrete-event simulation of a queue structure is to generate a random variate from an arbitrary distribution. For our purposes, due to the nature of interarrival and service times, it is important to generate exponentially distributed random variables. Let {e} be a sequence of number randomly distributed with the distribution function F(x):

$$F(x) = 1 - e^{-\lambda \cdot x}$$

If {v} is a sequence of uniformly distributed random variables obtained from a random generator, then it is possible to generate {e} as a sequence of exponentially distributed random variables with distribution function F(x) with the following relation:

$$e_i = -\lambda^{-1} \cdot \ln(v_i)$$

5.1 Implementation

In general, a discrete-event simulation used to model a queuing structure is composed of the following steps (Stone et al., 1975):

* Generate random variates.
* Create, modify and generally describe processes (jobs) that move through the simulation
* Delimit and sequence the phases of a process
* Facilitate the queuing of processes
* Collect, generate and display summary statistics

In order to illustrate these points, and based on the description provided by the Actor Model and the Active Object pattern, let us consider a simple active object, and model it as an Active Object Simulation Model.

The Active Object Simulation Model should allow to represent at most `M` jobs to be serviced simultaneously and if jobs arrive for service and there are already `M` jobs being serviced, the arriving jobs enter a FIFO queue. Once a job is being processed, it should generate a "burst" of processing time and communication time. The simulation model should also reflect that a job receives "service" from the active object or, if the active object is busy, it is put to idle. Upon generating a total amount of processing, communicating and idling time, it should leave the active object, triggering another job to be serviced if the job queue is not empty.

In order to further specify the simulation model, the following assumptions, which will allow us to discuss the details of its structure, are used:

* The arriving jobs form a Poisson process with mean arrival rate `lambda`.
* The maximum degree of multiprogramming is `M`.
* The computation time required by a job is a random exponentially distributed variable with mean `mu`.
* The communication channel is able to transfer a result to other active object describing a Poisson process with an average time `transfertime`.
* All queuing in model is FIFO.

From the previous modelling considerations described above, a class `ActiveObjectSimulator` is used for the simulation model in C++, based on the behaviour of the Active Object pattern. Only the interface of this class is presented in figure 5. The most important data structures that define the attributes or current status of the model are described as follows:

* `clock` is a real that indicates the current epoch in time being simulated.
* `actorbusy` and `channelbusy` are boolean variables that are true if the actor and channel, respectively, are busy servicing a job.
* `arrival` is an array of reals representing the times when each job arrived to the active object for service.
* `activeobjecttime` is an array of reals that stores the total amount of processing time required to complete servicing each job.
* `eventtime` is an array of reals representing times in which the event associated with the job is scheduled to occur. Some jobs will not have an event time if they are queued waiting to begin service on the active object.
* `type` is an array which specifies the state for jobs that have an event pending.
* `linkof` is an array used to link the state of each job with the next state.
* The job queue `jobQ`, active object queue `activeobjectQ`, and channel queue `channelQ` are the three FIFO queues. Internally in the class there are

pointers to their head as well as their tail to facilitate respectively the addition and deletion of jobs, from these queues.

* The free queue `freeQ` is simply a list of unused job descriptions. Jobs are taken from this queue when they are scheduled to arrive at the computer system and are added to the free queue upon departure from the system. For simplicity, this queue is implemented here as a stack, or last-in, first-out queue.
* The class `Markovian` represents the stochastical and memoryless behaviour of message passing in an active object, and its members are presented as follows (Figure 6): The methods `NegExponential()` and `Lognormal()` generate random variates with the indicated distributions. The kernel of each of these procedures is a uniform random number generator `Random()`. The random number generator is not defined since it is generally machine dependent. In the case that an event-driven simulation is required, this class can be simply replaced by another with a set of procedures that allow to read the duration of compute intervals, locate the input/output requests, and so on, from a trace information.
* Other remaining variables declared are needed to define several useful constants and parameters for the simulation and collect statistics and results.

During execution, these data structures are affected by the following methods of the class:

* The method `Simulate()` constitutes the main loop of the simulation model, initialising the state of the simulation and beginning the simulation by scheduling the arrival of the first job. The major operations of the simulation are called from this method. For each event, the simulation updates the state of the model, schedules a future event or queues a request when the unit is busy, and collects summary statistics.
* The following methods are used to control the execution of the simulation, reflecting the different events when the active object is processing, communicating or idling, and considering how time is spent on each event. There are six types of events: `NewJob()` represents the arrival of a new job to the active object; `JobComplete()` expresses that the job has completed all its processing time, and the active object searches for more queued jobs or it remains idle waiting for new jobs; `RequestCommunication()` represents the event when the object has requested to send an output communication; `StartCommunication()` considers the time spent when sending an output result; and `CommunicationComplete()` represents that the current message being sent has completed its transfer. The special method `Idle()` can be invoked in any situation in which the active object is not found in a processing or communicating state.
* A set of procedures to facilitate the maintenance of the queues in the model. The `Schedule()` method adds a job to the queue of pending events; the `Queue()` method adds a job to one of the three FIFO queues `jobQ`, `activeobjectQ` and `channelQ`. The `Dispatch()` method simulates the processing of a job on the processor.

```
class ActiveObjectSimulator {
public:
  ActiveObjectSimulator(); // Constructor
  ActiveObjectSimulator(double,double,double);
    // Constructor accepting values for
lambda,
    // mu and transfertime
  ~ActiveObjectSimulator(); // Destructor
  void Simulate(); // Main Active Object
                   // simulation method
private:
  // Simulation of Method Objects activity
  void NewJob();
  void Idle();
  void JobComplete();
  void RequestCommunication();
  void StartCommunication();
  void CommunicationComplete();

  // Scheduler (Method Object Scheduler)
  void Schedule(int);
  void Queue(int*,int);
  void Dispatch();
  . . .

  // Objects used during simulation
  Markovian markovian;
  . . .

  // Define simulation parameters
  double lambda;
  double mu;
  double transfertime;

  // Mayor state variables for simulation
  double clock, activeobjectmark, channelmark;
double arrival[maxN], activeobjecttime[maxN],
    eventtime[maxN+1];
  int type[maxN], linkof[maxN];
  int freeQ[2], jobQ[2], activeobjectQ[2]
    channelQ[2];
  int activeobjectbusy, channelbusy;
  int active, current, M, I, next, nextevent;
  . . .
}
```

Figure 5: Class interface for the Active Object Simulator

```
class Markovian {
public:
  Markovian(); // Constructor
  ~Markovian(); // Destructor

  double NegExponential(double);
  double LogNormal(double, double);
  . . .
private:
  double Random(); // returns a pseudo-random
    // value between 0 and 1
  const double pi = 3.141592;
}
```

Figure 6: Class interface for the Markovian

6 Experimentation

The objective of experimentation is to actually confirm that the simulation model described previously yields a possible behaviour close to the real behaviour of the simulated active object, when the right time parameters are considered. This is tested as follows: first, the parameters `lambda`, `mu` and `transfertime` are obtained from a real active object application program; second, using these parameters, a discrete-event simulation is produced varying the number of processes or jobs to be executed by the active object; and third, a comparison is made between the results of the simulation with the behaviour of the real program when the number of processes or jobs vary. These activities are performed using a simple active object program, as the experimental subject.

6.1 A Simple Active Object Example

In order to obtain input parameters for the experiment, let us consider the behaviour of a very simple active object program to be simulated. A counter active object program, originally developed in UC++ (Winder et al., 1995) and executed using PVM (Geist et al., 1994), is used for example. The behaviour of this counter active object is quite simple: it concurrently accepts requests to increment and display a value. For illustrative purposes, the measurement takes advantage of the simplicity of this program and its execution in the PVM environment to obtain the parameters for the simulation.

However, this not exclusive for this example, but the procedure can be used as well for actor programs with a complex or larger number of actors and function members.

6.2 Obtaining the Parameters for the Simulation

The parameters `lambda`, `mu`, and `transfertime`, required to validate our a pattern-based simulation, are obtained from the active counter example program. The arrival, service and result communication times are directly measured, generating a sample set for each one of them. The distribution of each parameter is chosen through observing a graphical representation of the distribution of measured times. The parameters are actually calculated by statistically obtaining the mean and standard deviation from the sample set of each parameter (Law & Kelton, 1991). Table 1 shows the observed distribution, mean, standard deviation and confidence interval used for each measured time parameter. The measurements are taken and adjusted using the *t-test* statistical technique for small samples, and considering sample sets of 10 measurements, considering an accuracy of 85%.

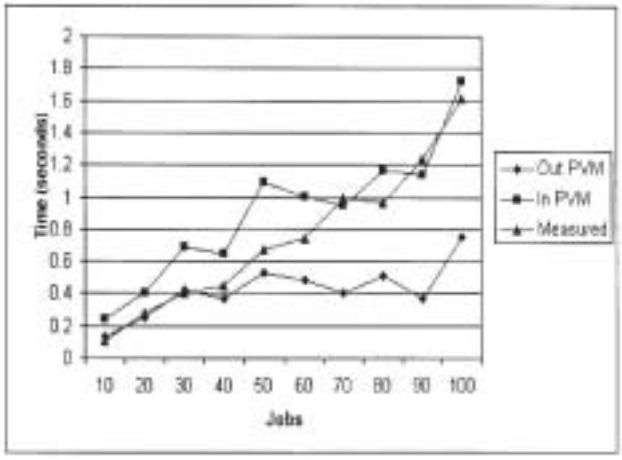
	Distribution	Mean	Standard Deviation	Confidence Interval
`lambda`	LogNormal	0.0187	0.0247	0.0006
`mu`	Neg Exponential	0.0139	0.0181	0.0004
`transfer time`	LogNormal	0.0068	0.0127	0.0003

Table 1: Input parameters for the Active Object Simulation

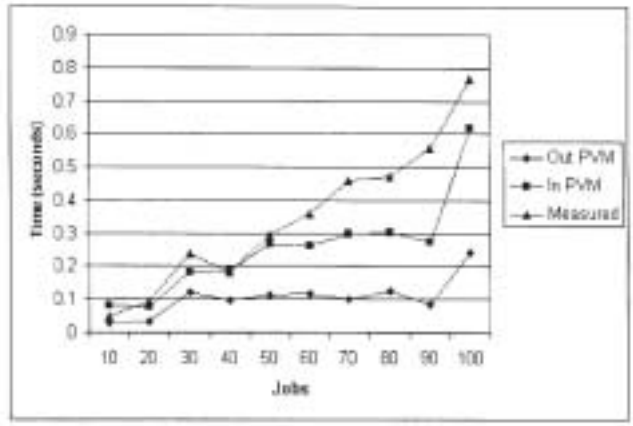
6.3 Simulation results

Using the measured parameters as input for the developed simulation model, the estimation times for the active object's execution are obtained by simply running the model. The estimates for processing, communicating and idling times are produced directly by the model, and collected using variables as simulation results. Since each execution yields each time different values for processing, communicating and idling times, the *t-test* statistical technique is used again, now for comparing the simulated and measured times. The experiment considers samples from 10 simulations and an accuracy of 85%. The average simulated processing, communicating and idling times are obtained against the number of request to the counter active objects. Figure 7 shows comparisons between simulated and measured processing, communicating and idling times for the counter active objects example. Simulated times are obtained considering the simulation execution cases outside and inside the PVM environment.

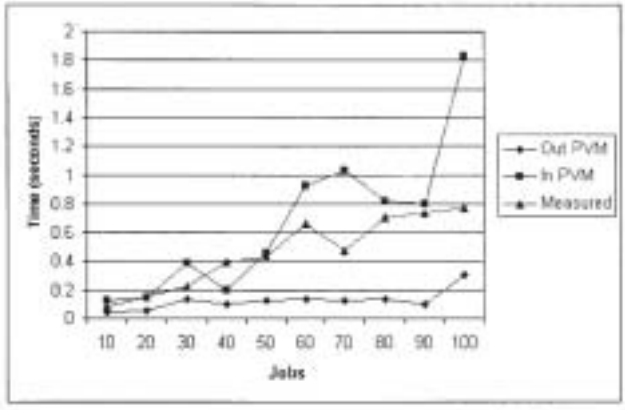
Some observations can be made from the comparisons in Figure 7. Consider the estimated processing and communicating times obtained from simulation (Figure 7a and 7b). Both estimations inside and outside the PVM environment present a similar tendency, as relatively good approximations to the real measured value of processing and communicating times. Furthermore, the simulation performed inside the PVM environment seems closer to the real measured values. In contrast, the idling times tendency exposes an erratic behaviour, which seems to have little to do with the real measured idling time value (Figure 7c). However, this does not mean that the simulation model is faulty. Let us remember that the simulation model was made based on parameters related to processing and communicating times.



(a) Processing



(b)Communicating



(c) Idling

Figure 7: Comparison between simulated and measured times for (a) processing, (b) communicating, and (c) idling of the counter active objects case example. Simulated values are obtained considering the simulation execution cases outside and inside the PVM environment

Precisely, the main objective of dividing the total execution time into processing, communicating and idling times is that, during execution, the influence of processing and communication estimations are balanced in the average case by the idling estimation, aiming to compensate errors and providing the simulation model with certain degree of flexibility. Table 2 shows the real and estimated total execution times obtained for both inside and outside PVM cases, and a calculation of the error for each case.

Jobs	Real Program (seconds)	Simulation Model, Out PVM (seconds)	Percentage of Error, Out PVM	Simulation Model, In PVM (seconds)	Percentage of Error, In PVM
10	0.232	0.388	67.15%	0.304	30.95%
20	0.517	0.357	30.86%	0.698	35.04%
30	0.874	0.995	13.96%	0.956	9.49%
40	1.028	0.935	9.03%	1.379	34.22%
50	1.4	1.676	19.66%	1.688	20.6%
60	1.766	1.98	12.11%	2.192	24.08%
70	1.932	2.28	18.04%	2.527	30.84%
80	2.159	1.964	9.0%	2.837	31.4%
90	2.533	2.197	13.26%	3.03	19.63%
100	3.151	3.867	22.72%	3.381	7.31%

Table 2: Total execution times obtained from measuring the real program and simulation estimations

Finally, Figure 8 shows the comparison between the estimated total times (again, executing inside and outside PVM) and the real total average execution time obtained from measuring the real program execution. These comparison are used to validate the output of the simulation program.

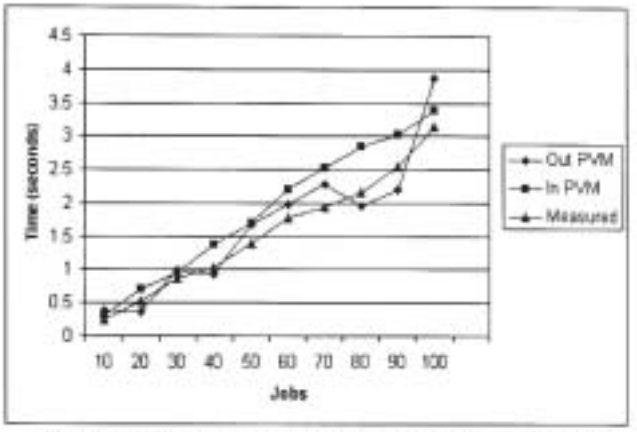


Figure 8: Active Object simulated and real times for different workloads

Even though a simulation does not produce an exact prediction on how a system will act, it does produce an outcome related with how the system will perform on average. This is, to some degree, predictable. In Figure 8, a correspondence between what the simulator estimates based on stochastic operations and the form in which the counter active object actually performs can be observed. The fact of executing inside and outside the PVM environment provides different average errors depending on the case. When considering the result of the simulation executing without using the PVM environment, the average error obtained is 27.57%. An improvement of the simulation output can be observed when executing the simulation inside a PVM environment, showing a more marked tendency to emulate the shape of the real counter active object execution, and obtaining a smaller average error for this case, equal to 24.35%. We consider that this error can be further decreased by considering other statistic details to improve its precision, like the standard deviation, and by using other distributions that may well provide better approximations to the temporal behaviour of the actor, for instance, the Weibull distribution (Law & Kelton, 1991). At this level, as the simulator and the real active object share the same structure, the simulator can be considered perhaps as a good initial prototype, useful for an initial concurrent program.

7 Summary and Conclusion

This paper proposes that a software pattern can be used not only to aid the design of a software system, but also as a base to simulate, with certain accuracy, its resulting behaviour. More concretely, it is proposed that the performance of an actor can be estimated through a pattern-based simulation, which combines the information contained in the Active Object pattern with stochastic and simulation techniques.

In summary, the Actor model, the Active Object pattern and the Active Object Simulation Model are presented and described, explaining their structural and behavioural characteristics and relations among them. It is particularly important the structural relations between the abstract elements of the Actor Model and their expression as design elements of the Active Object pattern, and between the design elements of the Active Object pattern and their implemented representation as code constructions in the Active Object Simulation Model. Finally, an example of a simple actor program is developed in order to validate the simulation model. From this example, measurements and experimentation show that the performance behaviour of an active object can be precisely modelled using a pattern-based model.

Two main conclusions can be drawn from the present experience. First, it is possible to say that the performance behaviour of an actor can be modelled with a certain precision, based on the information from the Active Object pattern in combination with stochastic and simulation techniques. Second, it is demonstrated that in general the structure and behaviour description of a software pattern can be used as a viable source of information about the value of a possible attribute of interest of a software based on such pattern. Depending on the attribute, different approaches or models can be constructed to observe with certain degree of accuracy, the effect of using the pattern on the expected attributes of the software system.

As a next step in this research work, it is proposed to experiment with more complex and larger concurrent systems, composed of a number of active objects, to test if a prediction can be made based on simulation using the Active Object pattern, time parameters, and the pattern of structure of cooperation between active components.

References

Agha, G., Frolund, S., Kim, W.Y., Panwar, R., Patterson, A., and Sturman, D. (1993a). "Abstraction and Modularity Mechanisms for Concurrent Computing". In Gul Agha, Peter Wegner and Akinori Yonezawa, (eds.) *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press.

Agha, G., Mason, I.A., Smith, S.F., and Talcott, C.L. (1993b). "A Foundation for Actor Computation". *Journal of Functional Programming*, Vol. 1, No. 1. Cambridge University Press.

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley.

Fowler, A. (1996) *Analysis Patterns. Reusable Object Models.* Addison-Wesley Object Technology Series.

Frolund, S. (1996). *Coordinating Distributed Objects, An Actor-based Approach to Synchronization.* The MIT Press.

Gabriel, R. (1996). "Pattern definitions". In http://hillside.net/patterns/definition.html.

Gamma, E., Helm, R., Johnson, R. Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam. V., (1994). *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press.

Lavender, R.G. and Schmidt, D.C. (1996). "Active Object: An Object Behavioral Pattern for Concurrent Programming". In Vlissides, J.M., Kerth, N.L., and Coplien, J.O. (eds.) *Pattern Languages of Program Design 2.* Reading, MA: Addison-Wesley.

Law, A.M. and Kelton, W.D. (1991). *Simulation Modeling & Analysis.* Second edition. McGraw-Hill International Editions.

Lazowska, E.D. Zahorjan, J., Graham, G.S., and Sevcik, K.C. (1984). *Quantitative System Performance. Computer Systems Analysis using Queueing Network Models.* Prentice-Hall, Inc.

Smith, C.U. and Williams, L.G. (1993). "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives". *IEEE Transactions on Software Engineering*, Vol. 19, No. 7, July 1993.

Stone, H.S. (editor), Chen, T.C., Flynn, M.J., Fuller, S.H., Lane, W.G., Loomis Jr., H.H., McKeeman, W.M., Magleby, K.B., Matick, R.E., and Whitney, T.M. (1975). *Introduction to Computer Architecture.* Science Research Associates, Inc.

Winder, R., Roberts, G., and Poole, J. (1995). "The UC++ Project". In http://www.dcs.kcl.ac.uk/UC++/.

Jorge Luis Ortega Arjona *was born in México, D.F. He obtained a Bachelo'sr in Science degree in Electronics Engineering from the Universidad Nacional Autónoma de México (UNAM), a Master's in Science degree in Computer Science from the UNAM and a PhD from the University College London (UCL) Computer Science department. He was a customer service representative at IBM, México from 1991 to1192. He was a lecturer in the Engineering Faculty at the UNAM from 1993 to 1995. He was a research assistant at the Parallel Processing Laboratory at the DEA-IIMAS, UNAM from 1994 to 1996. He held a postgraduate teaching assistant position at the UCL from 1997 to 1999. His research interests are in software architecture and design, software patterns, object-oriented programming and parallel processing.*

Graham Roberts *is a lecturer in the Department of Computer Science at University College London. He is graduated with a BSc. hons degree (1st) in Computer Science (1984), a MSc in advanced computer science and a PhD (thesis title: SmallType - A Type Declaration and Type Checking System for Smalltalk-80) from Queen Mary College, University of London. Currently, he teaches courses on Java, C++ and object-oriented software engineering, while doing research into object-oriented systems, patterns and parallel C++.*