Operating System Scenarios as Use Case Maps*

Edward A. Billard
Department of Math and Computer Science
California State University, Hayward
Hayward, CA 94542
billard@csuhayward.edu

ABSTRACT

This paper summarizes the analysis, design, implementation, and performance analysis of an object-oriented operating system. The analysis applies Use Case Maps (UCMs) to provide a high-level abstraction of the behavior scenarios for state transition, character output, network access, and disk access. The UCM for state transitions is converted into a queueing network for simulation study of I/O-bound versus CPU-bound systems. An overview of the later stages emphasizes UML for architecture and detailed collaboration, as well as Java examples. The performance of the disk subsystem is analyzed by instrumenting the Java code, and the results show that design choices can affect the throughput.

1. INTRODUCTION

Operating systems have traditionally presented a challenge in terms of software design [22]. These systems have a high degree of concurrency and often exhibit very subtle behaviors which emerge from the underlying implementation. A Use Case Maps (UCM) [1], [2], [7], [9] is a good candidate for abstracting this type of behavior, especially at the early analysis stage of development. A UCM provides a visual display of causal paths that weave through subsystems and transcend subsystems. UCMs have also been used in the context of agents [8], [10] and patterns [15].

The anticipated performance of a UCM can be analyzed at an early stage by translation to a queueing network [4] or, more specifically, a Layered Queueing Network (LQN) [12], [17], [18], [21]. Alternatively, a Unified Modeling Language (UML) [6], [14] presentation can be converted to a LQN [16].

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOSP 04, January 14-16, 2004, Redwood City, CA.
Copyright 2004 ACM 1-58113-673-0/04/0001 ...$5.00.

This allows the developer to investigate issues and problems with performance at an early stage of development [3], [13], an important consideration for performance-sensitive software such as an operating system. Other performance studies examine concurrency and real-time issues [19], [20] state machines [25], and predictive modeling [23], [24].

This study summarizes the life cycle of the development of an operating system. The system is object-oriented but is similar to the function-oriented XINU system [11], which is, itself, similar to UNIX. Previously, this author implemented a graphical user interface to a XINU-like operating system [5], intended to display the high-degree of concurrency and state transitions present in an operating system.

The goal of the current study is to apply modern techniques such as UCMs, UML, object-oriented programming (Java) to the classic domain of operating system software. Also, simulation and instrumentation provide for performance analysis of such systems.

The paper is organized as follows. Section 2 provides a subset of the UCM notation sufficient to present classic operating system scenarios in Section 3, as well as details of the network subsystem and the disk subsystem. In Section 4, the designs of these two subsystems are presented in UML, followed by Java implementations in Section 5. Section 6 shows the results of a queueing simulation for the state transitions in I/O-bound versus CPU-bound operating systems. The performance of the disk subsystem is analyzed by applying instrumentation to the Java implementation. Most disk subsystems use a form of an elevator algorithm to minimize disk arm movement, and the results show that the choice of algorithm can affect the throughput of the subsystem.

2. A SHORT UCM TUTORIAL

Figure 1 shows a subset of UCM notation sufficient to understand the maps presented in this paper. The full notation [7] is still relatively straight-forward but is not needed for this exposition. (Note that all of the UCMs in this study were created using UCM Navigator [1].) A start point represents the beginning of a causal path, and one should imagine this point as a source of "tokens" (with some distribution, say, uniform) that follow along the path. The path may lead through one or more components which have responsibilities. The path then terminates, and the direction of the tokens can usually be inferred from the start to end point but an arrow can be used as a guide.

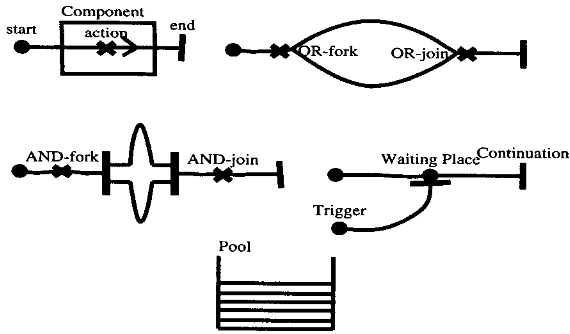

Figure 1: A subset of Use Case Map notation illustrating a casual path flow through forks and joins, along with possible waiting places. A pool is a useful notation for an operating system.

In essence, a path represents a high-level abstraction of the emergent behavior of the component hardware/software at runtime that results from the underlying implementations and protocols. The path is intended to reflect this emergent behavior at an early stage of analysis of the problem, even before the underlying implementations are attempted.

The paths become interesting because of AND-forks/joins and OR-forks/joins. An OR-fork represents a probabilistic alternative in which a token may travel one way or another (the figure only shows a two-way split but it can be n-way). Two or more paths may come together in an OR-join. The example shows two paths joining which originated from an OR-fork; this need not be the case. Tokens on different segments can flow into an OR-join, at different times, and then each will travel along the same path.

A token arriving at an AND-fork generates multiple, concurrent tokens, one for each output path. An AND-join requires that a token wait until a token arrives at each of the other joining paths and, again, it is not necessary for the paths, which lead to an AND-join, to have been created by an AND-fork. A path may reach a waiting place, which halts a token until another token arrives along a trigger path. At this time, the original token moves along the continuation path.

Finally, a pool is a data structure that stores arbitrary components and, in the case of operating systems, can be used to represent a queue, an important and prevalent feature of any operating system.

This is sufficient notation for the scenarios demonstrated in the next section. The full notation includes, among other things, subcomponents and stubs [7].

3. OS SCENARIOS AS UCMS

This section presents a sequence of UCMs to capture the behavior scenarios for different aspects of the operating system. The goal is to show how UCMs abstract, at a high-level, the difficult concepts.

First, all successful operating systems rely on a state-based model [22], that is, a process in the system is in one state at a time, and crucial events make the process transit to another

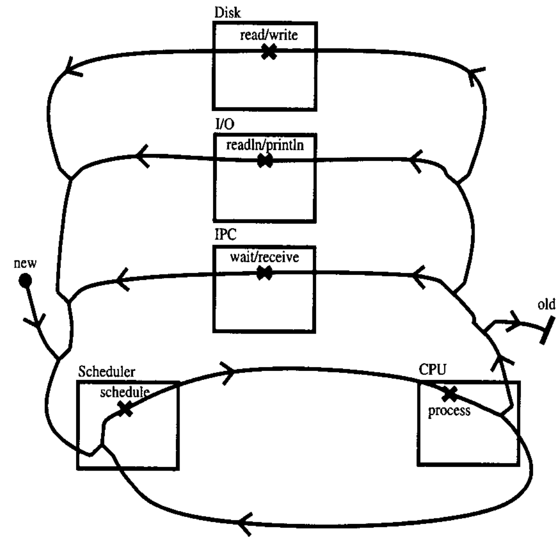

Figure 2: A state transition diagram as a UCM, with processes leaving the CPU for rescheduling or for the disk, I/O, or IPC subsystems.

clearly-defined state. Figure 2 shows the state transitions for the system, where each transition moves along a causal path to a component, or subsystem, which is responsible for the behavior of that particular state.

The scheduler moves a process from the scheduling, or ready, state onto the CPU. If the process requires more processing, it returns to the scheduler; otherwise, the path is to one of three other subsystems: Interprocess Communication (IPC), Input/Output (I/O), or disk. A subsystem handles the arriving process, and then returns the process to the scheduling state again. New processes also enter this state, and old ones leave the CPU.

One of the important problems in operating systems is mutual exclusion to a critical section of code, where only one concurrent process executes the code at any particular time. The problem (not the solution) is illustrated in Fig. 3. Both processes have an OR-fork which results in three possible scenarios.

First, each process may take the fork which leads to a waiting place. Note that the trigger to this waiting place is a path that results from an AND-fork from the other process after it successfully completes access to the critical section. That means that both processes will deadlock, waiting for the other process to cause a trigger event.

Second, each process may take the non-waiting path and both might enter the critical section, hence, violating mutual exclusion.

The third scenario results when one process takes the waiting path and the other process enters the critical section. After the access is complete, the process triggers the other process to enter the section. This is the desired scenario but

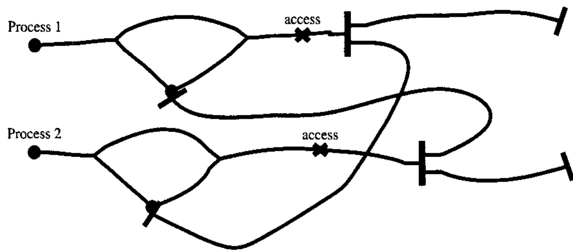

Figure 3: Three scenarios result from this critical section UCM: mutual exclusion, violation of mutual exclusion, or deadlock.

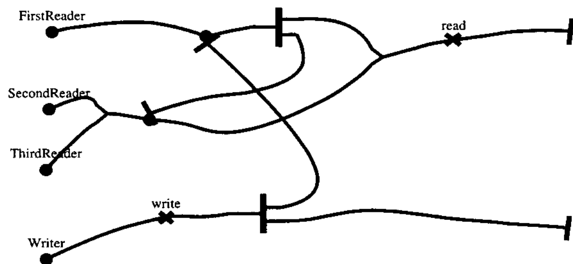

Figure 4: Readers have shared access to a file but must compete against a sole writer. Only the first reader does the actual contention and, afterwards, unsuspends the other readers.

does not show "how" to implement the code; this is not the main point of the analysis stage, and the UCM provides a good way of abstracting the problem.

Another important classical problem in operating systems is the Readers-Writer problem, see Fig. 4. Whereas the critical section problem requires mutual exclusive access, hence low concurrency, the access to a sharable resource, such as a file, allows for more concurrency. Readers of a file are allowed this type of shared access but must compete for this access against a sole writer to the file. The UCM shows a first reader competing for the file against a writer. The particular scenario shown has the reader block at a waiting place while the writer actually writes to the file. Other "piggy-back" readers enter the scenario but are blocked at a different waiting place: only the first reader needs to compete for the resource. After the writer is finished, it unblocks the first reader, who then unblocks the other piggy-back readers. All readers can then access the resource.

The previous UCMs analyze the system from an outside perspective, but now the inner behaviors of the system are investigated, in particular, the three main subsystems: I/O, IPC, and disk.

Figure 5 shows the classic bounded-buffer solution to the output subsystem. The scenario shows the behavior of printing a string of characters. The OutProducer component represents the "upper-half" of the device driver and it must

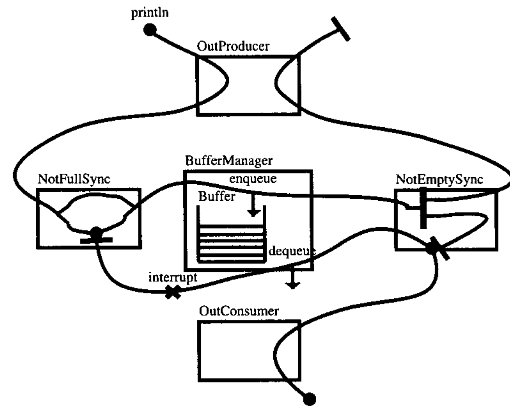

Figure 5: Output subsystem scenario. The producer blocks when the buffer is full; the consumer blocks when the buffer is empty.

first synchronize to be sure that it does not overwrite the buffer of output characters. The OR-fork in the UCM may block at a waiting place, or avoid this place and move on to the buffer manager.

The buffer manager puts a new character into the buffer pool, and then the path goes to another synchronizing mechanism. This mechanism is a trigger to a waiting place where the "lower-half" OutConsumer is blocked. (Device drivers typically have these upper- and lower-half components because of the differential in speeds: the upper-half runs, as the application writes strings, at the speed of the CPU, while the lower-half runs at the speed of the output/console controller.)

The scenario shows that it is important to block the OutConsumer because the buffer is initially empty. It is only when the OutProducer has put some data into the buffer that the OutConsumer can run, dequeue data from the buffer, and then unblock the OutProducer waiting because of a full buffer. It is the responsibility of the lower-half to generate an interrupt so that the device controller initiates the actual display of data.

A typical implementation of a lower-half usually involves a process with an infinite loop. The UCM here does not show that and, instead, relies on the token source nature of the start point to process more data. Again, this demonstrates that the analysis does not always directly match, nor should it, a design or implementation of the scenario. The input system has the same structure as the ouput scenario, with the exception that the roles are reversed with respect to the buffer.

Interprocess communication allows processes to coordinate their activities and is one of the most important features of an operating system. A semaphore may be used to control access to shared data or a critical section of code. Message passing is an alternative form of IPC. Messages may pass along a port within a computer system or, as shown in Fig. 6, across a network to a port on another system. The

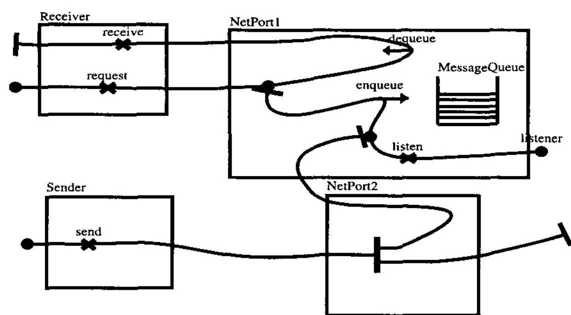

Figure 6: A sender transmits a message across the network, with a listening process at the other end of the connection. When the message arrives, it is placed into the queue and a blocked receiver can retrieve the message.

sender interacts with a port of the current system, the result is an AND-fork which sends the message on the network, and continues the behavior of the current process. On the receiver side, the act of requesting a message causes the path to go into a waiting place. This scenario results when the receiver requests a message but no messages are available in the message queue, or inbox.

The port on the receiver side also has another concurrent path which blocks on a waiting place, listening for messages coming in from the network. When a message arrives from the sender, this path unblocks, continues along the path to put the new message in the queue, and then triggers the blocked receiver to dequeue the new message, hence returning it to the receiver.

The last major subsystem is the disk, Figs. 7 and 8. The scenario to read a block of data from the disk is shown in Fig. 7, where it is assumed that the read is "synchronous", that is, the read must suspend in a waiting place until the completion of the disk read. The reader provides a buffer in which the device driver fills the data. This disk operation takes a long time compared to the speed of the reader, i.e., CPU speed. If the reader were to continue, without a suspension, then it would be eligible to read from a buffer with non-existent data.

On the other hand, the disk write scenario, Fig. 8, assumes an "asynchronous" write, that is, the write does not have to suspend. Instead, a write can continue on a path and go about doing other work, but usually this other work involves more writing to the disk. In particular, this work could involve reusing the same buffer that was provided in the previous write operation. Since the disk write operation takes a long time in comparison, the writer would, in effect, write over the original data. For this reason, an asynchronous writer needs to allocate an extra buffer in which to copy the data. This is the data that will be written to the disk. The writer is then free to reuse the original buffer over again. The only problem is that operating systems do not allocate any more memory after boot time because that could lead to a complete drain on memory by over-demanding disk writers. Instead, only a fixed amount of these "copy" buffers

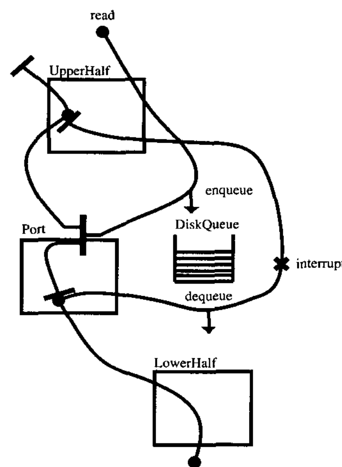

Figure 7: A disk read places a request into the queue and wakes up the lower-half process. After processing the request, the lower-half unblocks the "synchronous" read.

are allocated at boot time, and an over-demanding write will have to suspend if these are all consumed. When the system finishes processing a previous write request, a buffer will be returned, and a suspended writer can then reuse this as the copy buffer.

Now for the details of both the read and write scenarios as presented in the UCMs. A reader first interacts with the upper-half of the device driver. The upper-half enqueues a disk request on a disk queue and then sends a message on a port, thus signally the lower-half to unblock and dequeue the disk request. Note that this disk queue is probably not a First-In First-Out (FIFO) queue because the disk requests would tend to make the disk arm move back and forth while processing very few requests. Typically, an elevator algorithm [11], [22] would be used so that many requests can be processed as the disk arm moves in one direction.

After the lower-half removes a disk request, an interrupt is generated for the disk controller to process the request. After the processing occurs, the reading process, which has been suspended in the upper-half, is now unsuspended.

The disk write scenario, Fig. 8, is more complicated because it is asynchronous and requires the additional copy buffer mechanism. First, the writer needs to acquire a copy buffer, which may or may not block depending on the availability of buffers. Either immediately, or after the suspended writer unblocks, a disk request is placed into the disk queue and a message sent on the port. This message unblocks the lower-half, telling it to extract from the disk queue. After processing the request, the copy buffer is returned to the buffer pool manager, which then unblocks a suspended writer. Note that a writer only suspends in the case of an empty buffer pool; otherwise, the writer is free to continue with its activities at the same time as the device driver processes requests.

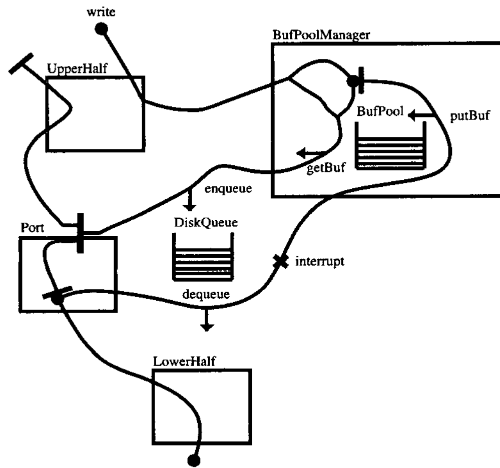

Figure 8: An "asynchronous" disk write attempts to allocate a copy buffer from the buffer pool. If empty, it blocks; otherwise, the write places a request into the queue and wakes up the lower-half. Because of the copy buffer, the write does not need to wait until the request is processed.

In conclusion, the UCMs are able to abstract this lengthy exposition into a visually concise format.

4. UML DESIGN

This section presents a short summary of the UML design. (The full UML design contains about 125 diagrams and is available at http://www.sci.csuhayward.edu/~billard/ooos/. The design is currently used in a graduate operating systems course.) Figure 9 shows the overall package layout with the public classes (+) representing the Application Programming Interface (API) for applications such as Dining Philosophers and Readers-Writer [22], both good examples of concurrent processes which use semaphores for interprocess communication. This package layout illustrates an important design principle for operating systems: layer the software. The upper layer is for application code, the middle layer for subsystems (IPC, I/O, Disk), and the lowest layer is for the core functionality of the system, in particular, the scheduling of processes. The following subsections provide details of the UML design of semaphores, network subsystem, and disk subsystem.

4.1 Semaphore Design

In Section 3, UCM scenarios for mutual exclusion (Fig. 3) and the Readers-Writer problem (Fig. 4) illustrated the need for semaphores to perform interprocess communication. A semaphore is used to protect "critical sections" of code where only one concurrent process should operate at a time. Entry to a critical section must be accompanied by a wait method call, which will suspend the process if another process is already in the section of code. When leaving a critical section, a process needs to do a signal call in case there are waiting processes. Figures 10 and 11 show the UML collaboration diagrams for both methods. The main point is that a counter determines whether a process needs to be suspended in a wait, or resumed in a signal, and, if neces-

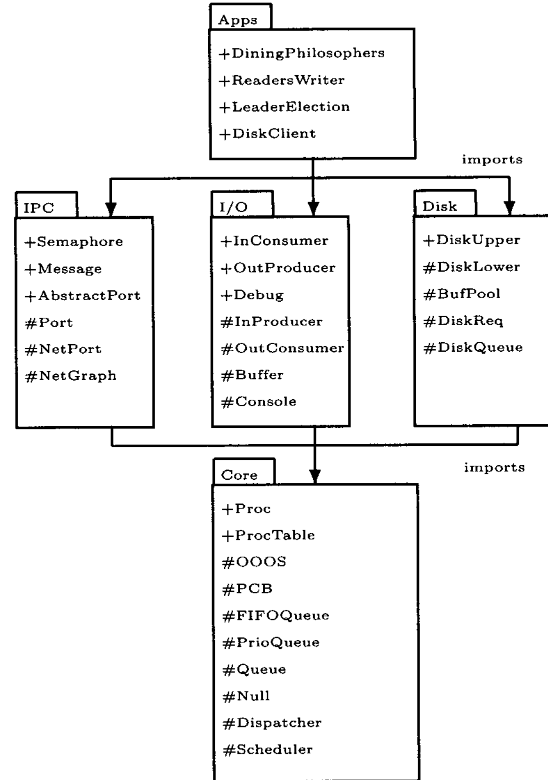

Figure 9: Application Programming Interface (API) specified as UML packages with subsystems in the middle layer.

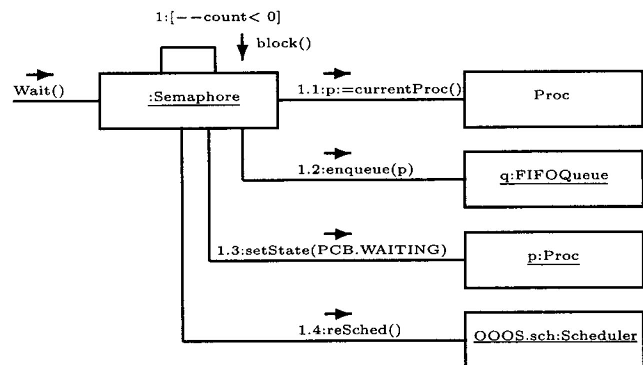

Figure 10: Collaboration diagram of a wait on a semaphore. A negative counter causes a requester to block and another process scheduled for the CPU.

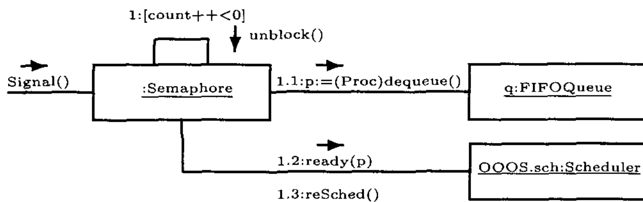

Figure 11: Collaboration diagram of a signal on a semaphore. A negative counter indicates the existence of waiting process, which should then be dequeued from the semaphore's queue and placed into the ready queue for the CPU.

sary, the scheduler will select a new process from the ready queue and install it on the CPU.

4.2 Network Subsystem Design

The design of the port subsystem, analyzed earlier as a Use Case Map (Fig. 6), is presented in Figs. 12, 13, and 14. The architecture design, Fig. 12, shows a superclass, called AbstractPort, which maintains both the queue of unreceived messages and the queue of processes waiting to receive a message. This class is able to generalize an implementation for the receive method: just return the head of the message queue, if not empty.

However, the send method depends upon the type of port. Communication within a computer system does not require network access and can be performed in-memory. Communication between two computer systems requires the network. Both of these ports, therefore, require specialization in subclasses of the abstract superclass, with only the network port shown in the figure. The network port requires the host and port number with which to communicate on the other end of the network connection. The network port implements the Runnable interface and is, therefore, an asynchronous thread. The thread blocks as it listens on the network for incoming messages.

Figure 13 shows the detailed design, as a UML collaboration diagram, of the send method. The main point is that the message can be serialized to a string, and then written to the network using Socket and BufferedWriter objects.

Figure 14 shows the design of the listener at the other end of the network connection. The run method uses Socket and BufferedReader objects to read from the network. Afterwards, the new message is placed in the message queue and, if the queue of waiting processes is not empty, then the longest waiting process is placed in the ready queue.

4.3 Disk Subsystem Design

The design of the disk scheduling subsystem is presented in Figs. 15 and 16. Disk subsystems, as well as I/O subsystems, are usually designed with upper-half and lower-half components. The upper-half object has a shared data structure with the lower-half, a queue of disk requests sorted based on an elevator algorithm [11], [22]. A port object is used to *communicate* between the two halves. The upper-half enqueues new disk requests (read/write) at the request of the

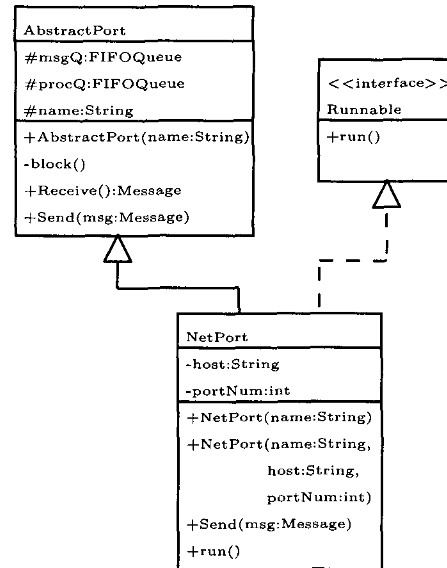

Figure 12: Network port inheritance structure in UML. The abstract superclass maintains the queues and implements the receive (just dequeue from the head of the queue). The subclass, a concurrent thread, sends messages on a port and also has a run method which listens for incoming messages on the network.

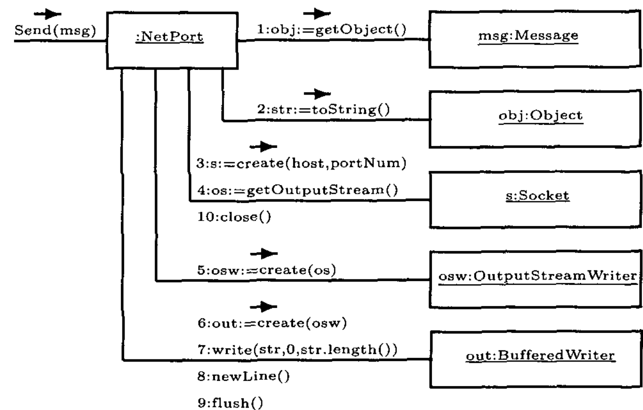

Figure 13: UML collaboration diagram for NetPort's send method. The message is written as a string to the network using a socket.

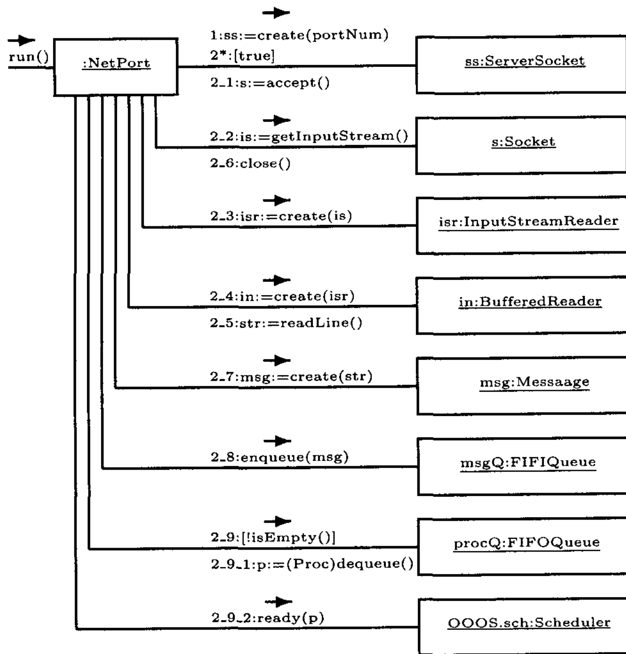

Figure 14: UML collaboration diagram for NetPort's run method. This asynchronous method listens on a server socket, unblocks, and reads a string from the port. After an enqueue of the new message, a receiving process can be unblocked and readied for the CPU.

application and the lower-half, an asynchronous process, dequeues and processes the requests.

The requests include a buffer in which to read data or write data, and the design calls for any reading application to block until the read operation is done ("synchronous read"). For any writing application, an "asynchronous write" is permitted by doing a copy to a new buffer allocated from a buffer pool, before the calling application continues on with its computation. The pool has only a finite number of copy buffers and a semaphore is used to block the requesting application if all buffers have been allocated. The lower-half removes from the head of the queue, seeks to the block, and writes the data from this copy buffer.

5. JAVA IMPLEMENTATION

The object-oriented operating system was implemented in Java. Details of the network and disk subsystems follow.

5.1 Network Subsystem Implementation

The analysis and design of the network port lead to an implementation in Java, Fig. 17. The code closely follows, as it should, the detailed UML collaboration diagrams in Figs. 13 and 14.

5.2 Disk Subsystem Implementation

In this subsection, the Java code is presented for the key parts of the disk subsystem, which forms the basis for the performance evaluation in the next section. The Java code for the pool of pre-allocated copy buffers, for asynchronous

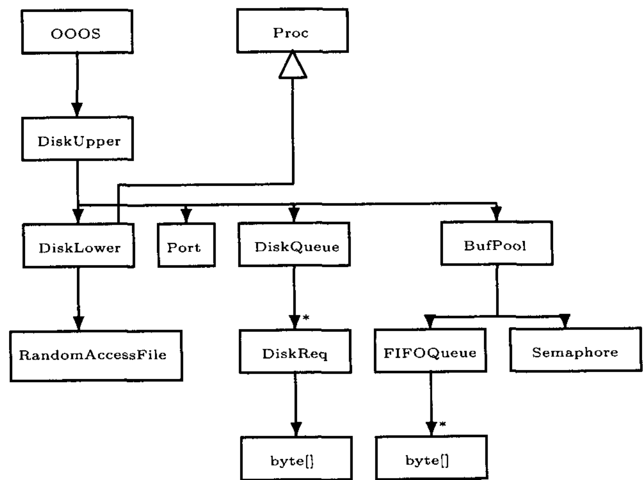

Figure 15: Structure of the disk scheduler. The upper-half of the device places new disk requests into a shared queue so that the lower-half of the device can extract, and process, at its own speed. A message port is used to control synchronization to the shared queue, and a buffer pool is available for acquiring "copy" buffers for asynchronous disk write requests.

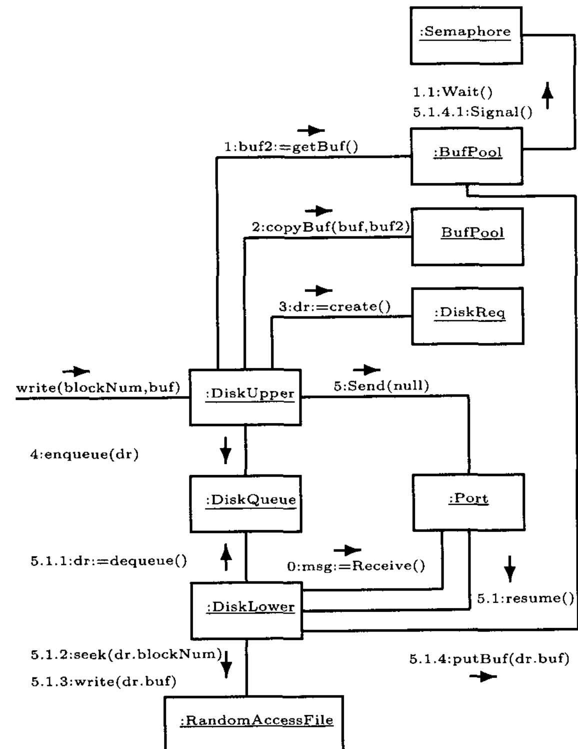

Figure 16: Collaboration diagram of disk scheduling interaction. Disk writes require a "copy" buffer before the enqueue of a new request. The lower-half blocks on a message receive until a new request is placed in the queue. After processing the write request, the buffer can be returned to the pool.

```
public class NetPort extends AbstractPort
                     implements Runnable {
  private String host;
  private int portNum;
  public NetPort (String name, String host,
    int portNum) {
    this(name);
    this.host = host;
    this.portNum = portNum;
    Thread t = new Thread(this);
    t.start();
  }
  public void Send(Message msg) {
    Object obj = msg.getObject();
    String msgStr = obj.toString();
    try {
      Socket s = new Socket(host, portNum);
      BufferedWriter
        out = new BufferedWriter(
                new OutputStreamWriter(
                  s.getOutputStream()));
      out.write(msgStr,0,msgStr.length());
      out.newLine();
      out.flush();
      s.close();
    }
    catch(IOException e) {}
  }
  public void run() {
    try {
      ServerSocket
        server = new ServerSocket(portNum);
      while (true) {
        Socket incoming = server.accept();
        BufferedReader
          in = new BufferedReader(
                 new InputStreamReader(
                   incoming.getInputStream()));
        String s = in.readLine();
        incoming.close();
        msgQ.enqueue(new Message(s));
        if (!procQ.isEmpty()) {
          Proc waiting = (Proc) procQ.dequeue();
          OOOS.sch.ready(waiting);
        }
      }
    } catch(IOException e) {}
  }
}
```

Figure 17: Java code for the NetPort class. There is almost a one-to-one relationship between the Java code and the detailed UML collaboration diagrams.

```
public class BufPool {
  public static int NUMBUFS = 20;
  FIFOQueue bufPool = new FIFOQueue();
  Semaphore notEmpty = new Semaphore(NUMBUFS);
  protected BufPool() {
    for (int i=1; i<=NUMBUFS; i++)
      bufPool.enqueue(new byte[512]);
  }
  protected byte[] getBuf() {
    notEmpty.Wait();
    return (byte[]) bufPool.dequeue();
  }
  protected void putBuf(byte[] buf) {
    bufPool.enqueue(buf);
    notEmpty.Signal();
  }
  protected static void copyBuf(byte[] buf1,
                                byte[] buf2) {
    for (int i=0; i<OOOS.BUFSIZE; i++)
      buf2[i] = buf1[i];
  }
}
```

Figure 18: Java code for pool of pre-allocated copy buffers, with protected access via a semaphore.

writes, is shown in Fig. 18. The constant NUMBUFS determines the number of pre-allocated copy buffers, which are placed in a First-In, First-Out queue. This constant becomes a variable in the performance evaluation and turns out to be important to the disk throughput. A semaphore is provided to block requesters of copy buffers when the pool is empty. This has the effect of stopping a burst of asynchronous write requests, hence affecting the throughput.

Figure 19 shows the Java code for the upper-half device driver. The application makes calls to the public read/write methods to access disk blocks. Note that an asynchronous write access requires the allocation of a copy buffer for the data, whereas the synchronous read is suspended. The upper- and lower-halves use messages on ports to control synchronization to the shared queue of disk requests.

In this study, the queueing and processing of disk requests has two implementations. First, the C-LOOK algorithm [22] is a fairly simple form of an elevator algorithm:

> *Move the disk arm from low to high blocks, processing any requests in the queue which occur along the way. Upon servicing the highest requested block, reset the arm to the lowest requested block in the queue, and start scanning toward higher blocks again.*

Second, the XINU [11] algorithm is another form of elevator algorithm:

> *When adding a request for block B to the existing queue of requests, schedule it to be performed between requests for i and $i + 1$ if the disk arm*

```
public class DiskUpper {
  private Port diskPort = new Port();
  private DiskQueue diskQ = new DiskQueue();
  private BufPool bufPool = new BufPool();
  public DiskUpper() {
    new DiskLower(diskPort,diskQ,bufPool).Start();
  }
  private void diskEnqueue(int ioType,int blockNum,
                           byte[] buf, Proc p) {
    DiskReq dr =new DiskReq(ioType,blockNum,buf,p);
    diskQ.enqueue(dr);
    diskPort.Send(null);
  }
  public void write(int blockNum, byte[] buf) {
    byte[] buf2 = bufPool.getBuf();
    BufPool.copyBuf(buf,buf2);
    diskEnqueue(DiskReq.WRITE,blockNum,buf2,null);
  }
  public void read(int blockNum, byte[] buf) {
    Proc me = Proc.currentProc();
    diskEnqueue(DiskReq.READ,blockNum,buf,me);
    me.Suspend();
  }
}
```

Figure 19: Java code for the upper-half device driver. Write requests require a "copy" buffer and read requests must be suspended.

> *will pass over block B on its way from i to i + 1. If no such pair i and i + 1 exist, add the new request to the end of the list.*

Consider the arrival of requests for blocks 50, 70, 60, 40, 80, 20, with block 50 undergoing immediate processing. The C-LOOK algorithm forms a queue (50 60 70 80) (20 40) and processes the first batch before resetting to the second batch for another upward swing. Note that both batches are sorted in ascending order and that a binary search can be performed on a particular batch in order to determine the position for inserting a new request. The queue is implemented with a Java vector, hence, the insertion operation is quick. The XINU algorithm forms a queue (50 60 70 40 80 20), which is a "fairer" order in terms of arrivals. The problem is that the queue is not in ascending order, implying that a sequential search is required, rather than a binary search. This has performance implications which are analyzed in the next section.

The lower-half device driver is shown in Fig. 20. The process blocks unless there are messages in the shared port from the upper-half device driver. The number of outstanding messages is equal to the number of outstanding disk requests in the disk queue. The lower-half extracts a disk request and either writes the block of data to the disk (and returns the pre-allocated copy buffer) or reads the block of data from the disk (and unsuspends the currently blocked application process).

The run method of the lower-half driver is the object of instrumentation in the performance evaluation. The total

```
public class DiskLower extends Proc {
  private RandomAccessFile raf;
  private DiskQueue diskQ;
  private Port    diskPort;
  private BufPool bufPool;
  protected DiskLower(Port diskPort,
                      DiskQueue diskQ,
                      BufPool bufPool) {
    this.diskPort = diskPort;
    this.diskQ    = diskQ;
    this.bufPool  = bufPool;
  }
  public void run() {
    while (true) {
      Message msg = diskPort.Receive();
      DiskReq dr = (DiskReq) diskQ.dequeue();
      diskIO(dr);
    }
  }
  private void diskIO(DiskReq dr) {
    raf.seek(OOOS.BUFSIZE * dr.getBlockNum());
    if (dr.getType() == DiskReq.READ) {
      raf.read(dr.getBuf());
      dr.getProc().Resume();
    } else {
      raf.write(dr.getBuf());
      bufPool.putBuf(dr.getBuf());
    }
  }
}
```

Figure 20: Partial Java code for the lower-half device driver. Write requests return the "copy" buffer and read requests are unsuspended.

number of processed disk requests divided by the time spent in the body of code determines the throughput of requests per unit of time.

6. PERFORMANCE ANALYSIS

Two performance studies are summarized in this section: a queueing network simulation of the state transitions and instrumentation of the disk subsystem.

6.1 State Transition Simulation

The state transition diagram, in Fig. 2, can be analyzed for anticipated performance using a queueing network for simulation. In this study, a tool implemented by the author [4] was used for simulation but a Layered Queueing Network (LQN) would be an alternative [17], [21].

Figure 21 shows a queueing network for Fig. 2, with new job arrival rate λ, probabilities p_i for rescheduling on the CPU, exiting the system, or entering one of the subsystems (I/O, Disk, IPC). The service rate for the CPU is μ_1 and for each of the subsystems μ_2.

A closed-form analytic solution is determined as follows. The conservation of job flow establishes a set of simultaneous equations for λ_i, the flow at any point in the network. The number of visits to each point is the ratio $V_i = \lambda_i/\lambda$.

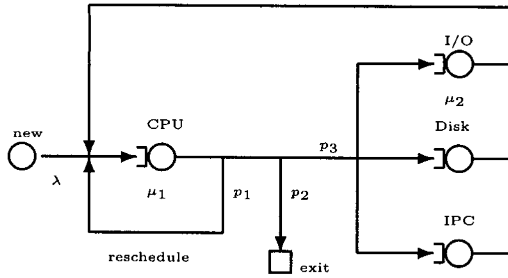

Figure 21: Queueing network for state transitions from CPU to CPU, I/O, Disk, or IPC.

Service Rates			
CPU (μ_1)	100 jobs/sec		
I/O, Disk, IPC (μ_2)	10 jobs/sec		
Probabilities			
Exit (p_2)	0.1		
	CPU-Bound	IO-Bound	Balanced
Reschedule (p_1)	0.9	0.0	0.45
I/O, Disk, IPC (p_3)	0.0	0.90	0.45

Table 1: Experimental design of CPU-bound versus I/O-bound systems.

Assuming an M/M/1 queueing system at each server, the response time is $R_i = 1/(\mu_i - \lambda_i)$. From this, the overall system response (turnaround) time is:

$$R_{system} = \sum_{i=1}^{4} V_i R_i = \frac{1}{p_2\mu_1 - \lambda} + \frac{1 - p_1 - p_2}{3p_2\mu_2 - (1 - p1 - p2)\lambda}$$

The simulation experimental design of service rates and probabilities is presented in Table 1. The main feature is that the probabilities are adjusted to reflect the nature of the input job stream. I/O-bound jobs do not return to the CPU directly but, instead, go to the I/O (or disk or IPC) subsystem. CPU-bound jobs always return to the CPU for more processing (unless the job exits), and never do I/O.

Figure 22 shows the difference in performance of I/O-bound versus CPU-bound job input streams as a function of the new job arrival rate. Given the faster service rate of the CPU, the CPU-bound system has a faster turnaround time.

6.2 Disk Subsystem Instrumentation

The instrumentation in the lower-half device code measures the throughput of disk requests, as shown in Fig. 23. The test application creates 5000 disk write requests. To analyze the effect of bursty disk requests, time slicing is not permitted. The only thing preventing an entire burst of 5000 requests from filling up the queue is the fact that the disk writes are asynchronous, necessitating the allocation of copy buffers. The semaphore in the buffer pool is designed to block this type of request if all pre-allocated buffers have

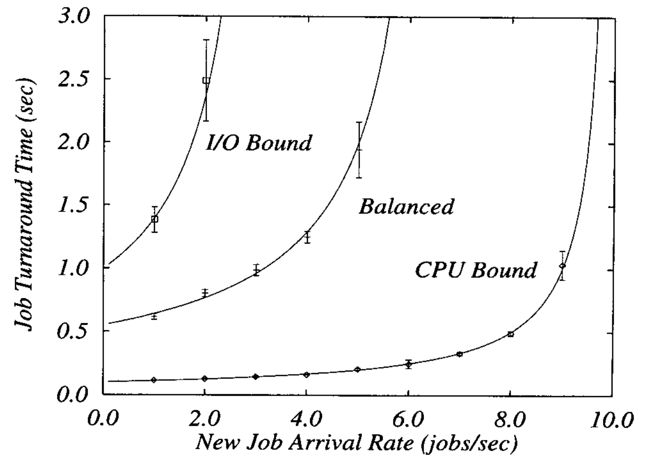

Figure 22: Simulation results (5 runs each) show the performance of I/O-bound versus CPU-bound job systems. Turnaround time is the average time until a job exits the system. The solid line represents an analytic result assuming M/M/1 queueing characteristics at each processor.

been assigned. The number of pre-allocated buffers is the x-axis in Fig. 23, each data point for the throughput (y-axis), is the average of 5 runs.

With a small number of copy buffers, the bursty behavior is controlled because the application blocks on the semaphore, and waits for a copy buffer to be returned by the lower-half device driver after processing a write request. With a large number of copy buffers, the bursty behavior of the application is only encouraged, and the queue of disk requests is large.

The C-LOOK algorithm shows a constant throughput mainly because the binary search on the sorted queue, even on a long queue, is efficient in determining the insertion point. In the XINU algorithm, the sequential scan of the disk queue, to find the appropriate position, slows the system down as the number of pre-allocated copy buffers increases. (Note that a very small number of copy buffers, say 1, reduces the throughput because of the context switching back-and-forth between the upper-half and lower-half drivers.)

Figure 24 shows the average response (or service) time to process a single disk write request. This is the time delta between inserting into the queue by the upper-half and actual processing by the lower-half driver. The C-LOOK algorithm shows an increase in this response time as the size of the disk queue increases (according to the number of copy buffers). This is to be expected because, although insertion-location is quicker, there are still a larger number of requests in the queue. As for XINU, the response time is worse because the insertion-location time is also longer.

7. CONCLUSIONS

This paper has summarized the life cycle development of an operating system. The Use Case Maps for describing operating system scenarios illustrate the subtle characteristics

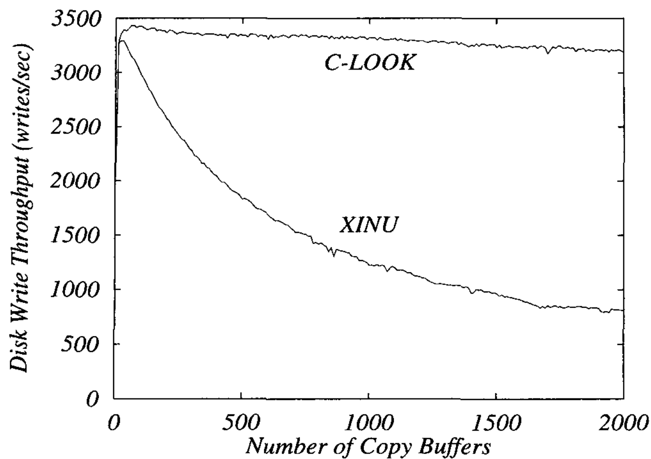

Figure 23: Disk write throughput as a function of the number of copy buffers. The XINU algorithm suffers because of bursty asynchronous disk write requests.

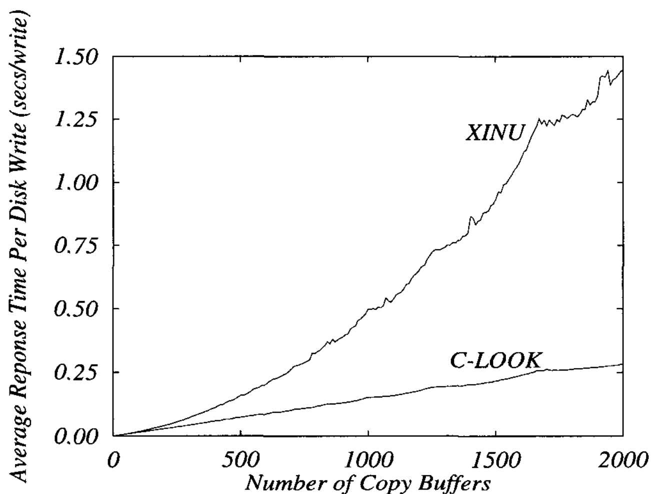

Figure 24: Average response time as a function of the number of copy buffers. The XINU algorithm, again, suffers because of bursty asynchronous disk write requests.

found in such systems, in particular, the concurrent processes which block and unblock. Also, the UCMs are able to show the behaviors which transcend subsystems, mainly the upper-half and lower-half interactions of the I/O and disk subsystems. The UCM notation for a pool is applicable for the wide variety of queue structures found in such systems.

The later part of the life cycle, the UML design and Java implementation, is distinct from the early UCM analysis, which tends to support a more open-ended, wide-ranging consideration of the problems at hand.

Although operating system design has been traditionally a function-oriented view, all of the concepts readily translate to an object-oriented view, with UML presentations describing both the organization of the subsystems and the details of the interactions. Class representations make clear the concepts of semaphores, messages, ports, processes, process control blocks, CPU schedulers, dispatchers, queues, as well as upper- and lower- half components for input, output, and disk schedulers. Operating systems have always presented challenges in terms of effective designs and the UML portion of this case study helps to deal with the complexities of such systems.

With respect to performance, queueing network simulation allows for the early investigation of performance issues, rather than waiting until completion of the implementation. The instrumentation of the disk subsystem shows the interesting (and paradoxical) result that more system resources - in this case, copy buffers - actually leads to reduced performance using the XINU elevator algorithm. The C-LOOK algorithm, with the binary search on the sorted queue, performs well. The XINU algorithm could be improved with the introduction of a mechanism, say a yield, which would artificially slow down the burst and let the lower-half driver process disk requests, hence, reducing the queue length.

8. REFERENCES

[1] Use Case Maps Web Site: http://www.UseCaseMaps.org.

[2] D. Amyot, R. J. A. Buhr, T. Gray, and L. Logrippo. Use case maps for the capture and validation of distributed systems requirements. In *Fourth International Symposium on Requirements Engineering*, pages 44–53, 1999.

[3] S. Balsamo, P. Inverardi, and C. Mangano. An approach to performance evaluation of software architectures. In *First International Workshop on Software and Performance*, pages 178–190, 1998.

[4] E. Billard and A. Riedmiller. Q-Sim: A GUI for a queueing simulator using Tcl/Tk. *ACM Software Engineering Notes*, 19(4):82–85, 1994.

[5] E. Billard and A. Riedmiller. A GUI for a manager of lightweight processes. *ACM Software Engineering Notes*, 20(5):48–50, 1995.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[7] R. J. A. Buhr. Use Case Maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–115, 1998.

[8] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. High level, multi-agent prototypes from a scenario-path notation: A feature-interaction example. In *Third Conference on Practical Application of Intelligent Agents and Multi-Agent Technology*, 1998.

[9] R. J. A. Buhr and R. S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice Hall, 1996.

[10] R. J. A. Buhr, M. Elammari, T. Gray, and S. Mankovski. Applying Use Case Maps to multi-agent systems: A feature interaction example. In *31st Annual Hawaii International Conference on System Sciences*, 1998.

[11] D. Comer and T. Fossum. *Operating System Design: The XINU Approach.* Addison-Wesley, 1988.

[12] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and C. M. Woodside. Performance analysis of distributed server systems. In *Sixth International Conference on Software Quality*, pages 15–26, 1996.

[13] C. Hrischuk, J. Rolia, and C. M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 399–409, 1995.

[14] C. Larman. *Applying UML and Patterns.* Prentice Hall, 2002.

[15] G. Mussbacher and D. Amyot. A collection of patterns for Use Case Maps. In *First Latin American Conference on Pattern Languages of Programming*, 2001.

[16] D. Petriu, C. Shousha, and A. Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Trans. Soft. Eng.*, 26(11):1049–1065, Nov 2000.

[17] D. Petriu and M. Woodside. Analysing software requirements specifications for performance. In *Third International Workshop on Software and Performance*, 2002.

[18] S. Ramesh and H. G. Perros. A multi-layer client-server queueing network model with synchronous and asynchronous messages. In *First International Workshop on Software and Performance*, pages 107–119, 1998.

[19] W. C. Scratchley and C. M. Woodside. Evaluating concurrency options in software specifications. In *International Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 330–338, 1999.

[20] B. Selic, G. Gullickson, and P. T. Ward. *Real-time Object-Oriented Modeling.* Wiley, 1994.

[21] K. H. Siddiqui and M. Woodside. Performance-aware software development (PASD) using resource demand budgets. In *Third International Workshop on Software and Performance*, 2002.

[22] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts.* Addison-Wesley, 1998.

[23] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley, 2001.

[24] L. G. Williams and C. U. Smith. Performance evaluation of software architectures. In *First International Workshop on Software and Performance*, pages 164–177, 1998.

[25] C. M. Woodside, C. Hrischuk, B. Selic, and S. Bayarov. A wideband approach to integrating performance prediction into a software design environment. In *First International Workshop on Software and Performance*, pages 31–41, 1998.