Predicting the Performance of Software Systems

Jerome Alexander Rolia

Technical Report CSRI-260
January 1992

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Abstract

With the advent of distributed and multi-processor computing, systems of cooperating processes have become an attractive alternative to monolithic programs. In such software systems it is often possible to alter the degree of parallelism within a program, and for the parallel threads to synchronize and communicate. There is a likelihood that processes will incur delays for both hardware and software resources. The resulting performance behaviour can be difficult to understand without the use of performance modelling tools.

The purpose of this thesis is to develop analytical performance modelling techniques that can be used to study software systems. In particular, fixed sets of processes and several communication and synchronization mechanisms are considered.

Predicting the Performance of Software Systems

by

Jerome Alexander Rolia

A Thesis submitted in conformity with the
requirements for the Degree of
Doctor of Philosophy
in the University of Toronto

January, 1992

©Copyright by Jerome Alexander Rolia 1992

Abstract

Jerome Alexander Rolia

University of Toronto

Department of Computer Science

January, 1992

With the advent of distributed and multi-processor computing, systems of cooperating processes have become an attractive alternative to monolithic programs. In such software systems it is often possible to alter the degree of parallelism within a program, and for the parallel threads to synchronize and communicate. There is a likelihood that processes will incur delays for both hardware and software resources. The resulting performance behaviour can be difficult to understand without the use of performance modelling tools.

The purpose of this thesis is to develop analytical performance modelling techniques that can be used to study software systems. In particular, fixed sets of processes and several communication and synchronization mechanisms are considered.

Acknowledgements

I would like to thank my supervisor Ken Sevcik for the inspiration, guidance, and friendship offered throughout my studies. My years at the University were enriched by his discussion and detailed criticism of the ideas presented in this thesis.

My thanks to the members of my committee: Mart Molle, Vassos Hadzilacos, Songnian Zhou, Dave Wortman, and Eugene Fiume. Their many insightful comments improved this thesis in countless ways. I am grateful to Connie Smith who was the external examiner. Her contributions to software performance engineering influenced my studies, and her support made my work all the more satisfying. Dr. M. Posner was kind to participate in my Senate Oral.

My family and friends contributed greatly to my work with their unending encouragement. In particular, I thank my father for listening, my mother for her desire to help, and my girlfriend Elvira for her love and care. My dear friends from Victor Avenue kept my spirits high!

Financial assistance was provided by the University of Toronto, the Department of Computer Science of the University of Toronto, and the National Science and Engineering Research Council of Canada. The IBM Canada Laboratory provided computing facilities for the development and execution of the applications considered in this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems are being developed in which software processes share resources and cooperate to accomplish overall system goals. Examples include transaction processing systems and data acquisition systems. In such systems there is a likelihood that processes will incur delays due to contention for both hardware and software resources. The resulting system performance behaviour often defies intuition. The purpose of this thesis is to develop predictive performance modelling techniques that can be used to help understand such systems.

The performance issues that arise are of particular importance in distributed and multi-processor systems. In such systems, **software processes** can act as both **customers** and **software servers** while sharing **hardware servers**. The requests for service amongst processes and for devices can be described as a **software process architecture**.

It is possible for a software process architecture to have a major impact upon system performance. It is even possible for a software server to become a system bottleneck. This occurs when the server has a higher utilization than any single device. Though increasing the power of the hardware can increase the throughput of the system, it may be more economical to alter the software or to choose a software process architecture that will meet its performance requirements in the first place.

Performance modelling is a method of discovering which of a system's many features determine performance by representing them in a predictive model. For the systems that are considered in this thesis, each process' **resource demands**, such as CPU time requirements, are quantified and used to define a model. Solution techniques have been developed that estimate contention for resources and process throughputs. Some of the performance measures that are predicted include:

- process throughputs and response times,
- process utilizations,
- device queue lengths, and utilizations.

A change in model parameters permits the prediction of system performance under modified circumstances. Using the model, a range of system scenarios can be examined without actually altering the system under study.

Performance models have been used as tools by computer system capacity planners for almost two decades. With existing performance analysis packages [BGS 82, QSP 82], a system's expected behaviour can be forecast with new workloads and hardware configurations. The packages have gained acceptance in industry and have demonstrated the effectiveness of such predictive performance tools.

It is hoped that the techniques developed in this thesis will further the goal of using analytical modelling techniques to consider performance issues when designing, implementing, and maintaining software systems.

In section 1.1, several software environments are considered. The behaviours of communication primitives that are used in the environments are compared and a common set of performance prediction problems is chosen for analysis. In the following sections of chapter 1, background is given for the techniques that are used to analyse the problems.

Performance models for software processes are considered in section 1.2. The techniques can be used to characterize a program's resource requirements. A program can be analysed to discover the average number of times each of the program's statement blocks is expected to be executed. This information can be used to estimate service demands at devices and the frequency with which a process communicates and synchronizes with other processes.

When many programs compete for access to shared resources such as processors, contention delays can arise. A program that requires five units of processor time may take twelve time units to acquire the service. Analytical methods to predict such contention delays, in particular Queueing Network Modelling and Performance Petri Nets, are introduced in section 1.3.

In section 1.4, the performance modelling of software process architectures is considered. Several primitives including the *fork*, *join*, and *rendezvous* are presented and their effects on process response times and throughputs are discussed. The behaviour caused by such primitives exists in most multi-tasking environments and capture important aspects of inter-process interactions. The techniques considered serve as a starting point for the work done in the rest of the thesis.

Finally, outlines for the chapters of the thesis are provided in section 1.5.

1.1 Software Environments and Performance Issues

In this section, three different software environments are described. Each provides its users with the capability to have processes interact with one another using a set of communication primitives. The primitives can affect the performance behaviour of the processes that are involved. The purpose of this discussion is to find a common set of performance features that exist in the environments. A thorough treatment of concurrent programming languages and the primitives used to implement them can be found in the literature [AndSh 83, Wegner 83].

To begin, a **sequential program** specifies the sequential execution of a list of statements; its execution is called a **process**. A **concurrent program** specifies two or more sequential programs that may be executed concurrently as **parallel processes**. A concurrent program can be executed on one or more processors.

The terms process and **task** are considered synonymous. A concurrent program can also be called a **multi-tasking** or **multi-threaded** program.

Cooperation amongst processes implies that they affect each other's execution. **Communication** is described as the sharing or passing of information from one process to another. **Synchronization** implies that processes are constrained to satisfy a required ordering of statements.

In some environments, processes must block when they communicate. Synchronization occurs when each of the involved processes reaches a specific statement in its execution. After the communication occurs, the processes continue their independent execution. When a process reaches its synchronization statement, the time before the other involved processes reach their synchronization statements is defined as a **queueing** or **synchronization delay** for the process.

It is generally accepted that there are three classes of software environments that support concurrent programming: procedure oriented, message oriented, and operation oriented. Languages in the same class provide the same basic kinds of mechanisms for process interaction and have similar attributes, and each class has roughly the same expressive power [AndSh 83]. For example, each environment provides the capability for **mutual exclusion** in the use of shared resources. That is, the number of processes that can simultaneously access a shared resource is restricted. In figure 1.1, an example is given where two processes share a portion of code, a **critical section**, that only one process may execute at a time. The operations **wait** and **signal** are indivisible operations on a semaphore that are used to guarantee that only one process can pass into the critical section at a time.

In sections 1.1.1 through 1.1.3, the characteristics of each environment are described and a sample language is given. Several features from each language are discussed. In section 1.1.4, a set of performance problems common to each of the environments is described.

```
var mutex : semaphore initial(1)

Process P1
loop
  wait(mutex)
  critical section statements
  signal(mutex)
  non-critical statements
end loop

Process P2
loop
  wait(mutex)
  critical section statements
  signal(mutex)
  non-critical statements
end loop
```

Figure 1.1: A critical section protected by a semaphore.

1.1.1 Procedure Oriented Languages

Procedure oriented languages base process interaction upon shared variables which can be protected using mutual exclusion. This is the case with languages that use monitors. **Monitors** can be described as shared data that can be accessed via a set of procedures. The procedures define the operations that can be performed upon the monitor's data. Alternatively, the procedures can be regarded as services provided by the monitor.

In Brinch Hansen's Distributed Processes model [Brinch73], any number of processes can make a request for service from the monitor, but only one process can execute within the monitor's procedures at a time. In this way a monitor provides mutual exclusion over the shared data. Processes that use the procedures of a monitor are queued in a single external queue with a first-come-first-served scheduling discipline. A process within the monitor is permitted to use *suspend* and *resume* operations to suspend itself and resume another process that had previously suspended itself, respectively.

```
a_monitor : monitor;

{ monitor operations }
procedure do_critical;
begin
  critical section statements
end;

begin
  initialization statements
end; { monitor }

Process P1;
begin
  repeat
    a_monitor.do_critical;
    non-critical statements;
  until false;
end;

Process P2;
begin
  repeat
    a_monitor.do_critical;
    non-critical statements;
  until false;
end;
```

Figure 1.2: A monitor used to protected a critical section.

In figure 1.2, an example is shown where a critical section is protected using a monitor.

Concurrent Pascal [Brinch75] is an example of a procedure oriented language that uses Brinch Hansen's monitors.

1.1.2 Message Oriented Languages

Message oriented languages provide process communication via *send* and *receive* type primitives. Processes do not share data at all; communication takes place through the content of messages alone. A process can provide mutual exclusion over a resource by managing the number of requests it serves and the order in which it serves them.

In general, the primitives do not imply any form of blocking. However a *receive* can only be successful when there is some information to be received. Some languages use bounded message queues that allow a process to send to a queue until the queue is full. Thereafter, a sending process remains blocked until a queue position becomes available.

Communicating Sequential Processes (CSP) [Hoare 78] is an example of a message passing language. A blocking form of *send* and *receive* are used in CSP. Before information can be communicated between two processes, one must execute an *output* statement and the other must execute an *input* statement. Thus, the two processes must synchronize. Once the communication occurs, each process can continue its execution. Each *input* and *output* statement must name the process that is being input from or output to, respectively. A more general form of *input* statement, named *Pr*, exists that can choose amongst a set of possible outputing processes in a first-come-first-served manner [AndSh 83]. Thus it is possible for a queue of outputing processes to develop while waiting for synchronization with the corresponding inputing process. The corresponding *output* statement is named *Ps*.

The input-output sequence is often referred to as a **CSP rendezvous**. Other than the effort required to exchange data there is no **rendezvous service**. That is, neither the sender nor receiver is blocked while the other executes statements. All processing initiated due to the exchange of information occurs after the rendezvous and is defined as **post-rendezvous service**.

In figure 1.3, an example is shown where a process is used to protect a critical section. Processes P1 and P2 queue for service at process Server. The Server process will only execute the critical section on behalf of one caller at a time. The structured use of *Pr* and *Ps* statements in the Server and its callers emulates both rendezvous and post-rendezvous service.

```
Process Server;
begin
  loop
    Pr?Request; { input an output of type Request }
    critical section statements (Rendezvous)
    Ps!Result; { output a result of type Result }
    non-critical statements (Post-Rendezvous)
  end loop;
end;

Process P1;
begin
  loop
    Ps!Request; { output a request of type Request }
    Pr?Result;  { input an output of type Result }
    non-critical statements
  end loop;
end;

Process P2;
begin
  loop
    Ps!Request; { output a request of type Request }
    Pr?Result;  { input an output of type Result }
    non-critical statements
  end loop;
end;
```

Figure 1.3: A CSP process is used to protect a critical section.

1.1.3 Operation Oriented Languages

Operation oriented languages are similar to message oriented languages except that processes may share data and the receiving process can execute statements while the sending process remains blocked. In this way, the receiving process acts as a **software server**.

Ada [DOD 83] is an example of an operation oriented language. In Ada, processes are called **tasks**. Serving tasks provide different services via a set of entry points that can be called, like procedures, by requesting tasks. A serving task executes an *accept* statement to indicate that it is ready to synchronize with a caller. The *accept* statement has a body that may contain statements. The execution of the statements is defined as rendezvous service. Afterwards both the caller and server continue processing independently. Those statements in the server that are executed after the *accept* body provide post-rendezvous service. The call-accept-release interaction is referred to as an **Ada Rendezvous**.

The Ada style rendezvous is more general than the CSP rendezvous because it can provide both rendezvous and post-rendezvous service. The CSP rendezvous only directly enables post-rendezvous service. The **remote procedure call** is a rendezvous that permits only rendezvous service. For the remainder of the thesis, the Ada rendezvous primitive is referred to as the **rendezvous**.

In figure 1.4, an example is shown where an Ada task is used to provide mutual exclusion over a critical section. Tasks T1 and T2 queue for service at the Server task. The Server task only provides service to one caller at a time.

1.1.4 Common Performance Features

Using the examples of procedural Concurrent Pascal with monitors, the message passing CSP, and the operation oriented Ada, several performance issues common to the environments are now considered. It is beyond the scope of this thesis to consider all of the similarities. The set of common features serves as the motivation for the analysis

```
Task Server;
begin
  loop
    accept Request do
      critical section statements (Rendezvous)
    end Request;
    non-critical statements (Post-Rendezvous)
  end loop;
end;

Task P1;
begin
  loop
    Server.Request;
    non-critical statements
  end loop;
end;

Task P2;
begin
  loop
    Server.Request;
    non-critical statements
  end loop;
end;
```

Figure 1.4: An Ada task is used to protect a critical section.

developed in subsequent chapters of this thesis.

First, consider **hierarchical software systems**. In Concurrent Pascal, monitors can make calls to other monitors. It is possible for the calling monitor to be suspended, causing all of its users to be delayed. This has been called the problem of **nested monitor calls** [AndSh 83]. In CSP a similar situation can arise. Consider processes A, B, and C. Process A may wish to output to B, B may wish to output to C, and C may be doing some other processing. A must wait for B to synchronize with C before it can synchronize with B. In Ada, it is possible for a serving task to make requests for service from other tasks. The serving task can become blocked while waiting for service. This delay could be suffered by the initial caller as well. For both CSP and Ada this situation is defined as **nested rendezvous**. The Ada tasks, CSP processes, and Pascal monitors form hierarchies of tasks, processes, and monitors, respectively.

An example of a hierarchical software process architecture, written in Ada, is shown in figure 1.5. In figure 1.6, a diagram is shown that portrays the software process architecture of the program given in figure 1.5. Each parallelogram describes a task and an arrow from one task to another indicates requests for service [Buhr 83]. Such diagrams provide a quick description of the structure of an application's software and are used extensively throughout this thesis.

In general, each environment allows processes to provide services in a **fixed** or a **non-fixed** order. In the terminology of Ada, consider a task with entries A and B. The task can provide services in a fixed order by alternating its execution of *accept* A and *accept* B statements. The task can provide services in an non-fixed order by using a statement that allows the acceptance of a customer at either entry A or B depending upon which has an available customer. CSP has the *Ps* and *Pr* statements to achieve this effect. Monitors provide non-fixed ordered services by default, but can use *suspend* and *resume* operations to implement fixed ordered services. In figure 1.7, examples are shown of Ada code that implements servers that provide fixed ordered and non-fixed ordered service.

```
Task body C1 is
begin
  loop
    accept Request do
      critical statements
    end Request;
  end loop;
end;

Task body B1 is
begin
  loop
    accept Request do
      C1.Request;
    end Request;
  end loop;
end;

Task body B2 is
begin
  loop
    accept Request do
      C1.Request;
    end Request;
  end loop;
end;
```

```
Task body A1 is
begin
  loop
    B1.Request;
    B2.Request;
  end loop;
end;

Task body A2 is
begin
  loop
    B1.Request;
    B2.Request;
  end loop;
end;

Task body A3 is
begin
  loop
    B1.Request;
    B2.Request;
  end loop;
end;
```

Figure 1.5: A hierarchical software system written in Ada.

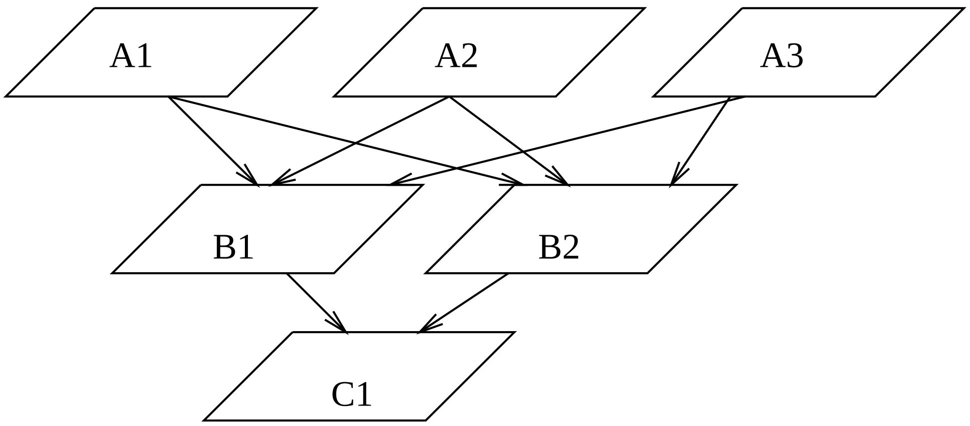


Figure 1.6: A visual portrayal of a hierarchical software system.

```
 Task body B is -- Implemented as a Fixed Ordered Server
begin
  loop -- alternate acceptance of callers of entries Request1 and Request2
    accept Request1 do
      rendezvous statements for entry 1
    end Request;
    post-rendezvous statements for entry 1

    accept Request2 do
      rendezvous statements for entry 2
    end Request2;
    post-rendezvous statements for entry 2
  end loop;
end;

Task body B is -- Implemented as a Non-Fixed Ordered Server
begin
  loop
    select -- accept an available caller from either entry
      accept Request1 do
        rendezvous statements for entry 1
      end Request;
      post-rendezvous statements for entry 1
    or
      accept Request2 do
        rendezvous statements for entry 2
      end Request2;
      post-rendezvous statements for entry 2
    end select;
  end loop;
end;

Task body A1 is
begin
  loop
    B.Request1;
    B.Request2;
  end loop;
end;

Task body A2 is
begin
  loop
    B.Request1;
    B.Request2;
  end loop;
end;
```

Figure 1.7: Fixed and Non-Fixed Ordered Servers written in Ada.

It is also possible to condition the acceptance of an entry upon some aspect of the system's state. This is often referred to as **condition synchronization** [AndSh 83]. For example, Ada uses **guards**; a buffer task with entries named *acquire* and *release* may accept either of the entries but only accept an *acquire* request when there are empty buffer elements available. The same functionality can be implemented using **conditional critical sections** in monitors and with guards in CSP. The producer-consumer problem is a standard condition synchronization problem [Holt 78].

In figure 1.8, a producer-consumer problem is shown. The Buffer task maintains data structures that contain empty and full buffer elements. The producer and consumer each acquire a buffer element, operate on the element, and then return it to the buffer. A **hold** time is the time required for producers and consumers to operate on an element before returning it to the buffer. Implementations of producer-consumer relationships that do not use a buffer task are also possible. Mutual exclusion over pertinent data structures must still be guaranteed.

Another interesting type of behaviour arises with the **nested acceptance** of callers. This can be used to force the explicit synchronization of several processes. In Ada, *accept* bodies can be nested. For instance, a task with entries A and B may accept a caller from entry A and then, before the body completes, accept a caller from entry B. The callers of A and B are synchronized along with the serving task within B's *accept* body. The callers must be released in a last-in-first-out (LIFO) order. Thus the caller of entry B is released before the caller of entry A. CSP could implement the same functionality using an appropriate sequence of *input* and *output* statements. Monitors can use their *suspend* and *resume* operations to accomplish the same behaviour.

An example of nested acceptance is shown in figure 1.9. Task B must accept two callers before it executes its rendezvous statements. The two callers synchronize and remain synchronized until the end of the nested accept statement that corresponds to entry Request2.

Some languages provide the facility for **dynamic process creation**. Each of

```
 Task body Buffer is
 Empty_Element_Avail : Boolean;
 Full_Element_Avail : Boolean;
 begin
   loop
     select
       when Empty_Element_Avail => -- Guards evaluate expressions
         accept Get_Empty_Element(Buf : out Element_Type) do
           rendezvous statements
         end Get_Empty_Element;
         post-rendezvous statements
         Set Empty_Element_Avail to False if none are available
     or
       when Full_Element_Avail =>
         accept Get_Full_Element(Buf : out Element_Type) do
           rendezvous statements
         end Get_Full_Element;
         post-rendezvous statements
         Set Full_Element_Avail to False if none are available
     or
       accept Put_Empty_Element(Buf : in Element_Type) do
         rendezvous statements
       end Put_Empty_Element;
       post-rendezvous statements
       Set Empty_Element_Avail to True
     or
       accept Put_Full_Element(Buf : in Element_Type) do
         rendezvous statements
       end Put_Full_Element;
       post-rendezvous statements
       Set Full_Element_Avail to True
    end select;
  end loop;
end;

Task body Producer is
Element : Element_Type;
begin
  loop
    Buffer.Get_Empty_Element(Element);
    statements that fill the buffer (Determines Producer Hold Time)
    Buffer.Put_Full_Element(Element);
    other statements
  end loop;
end;

Task body Consumer is
Element : Element_Type;
begin
  loop
    Buffer.Get_Full_Element(Element);
    statements that empty the buffer (Determines Consumer Hold Time)
    Buffer.Put_Empty_Element(Element);
    other statements
  end loop;
end;
```

Figure 1.8: A Producer-Consumer problem implemented using guards in Ada.

```
Task body B is
begin
  loop
    accept Request1 do
      accept Request2 do -- must accept two
                         -- callers before
                         -- the statements
                         -- can be executed
        rendezvous statements
      end Request2;
    end Request1;
    post-rendezvous statements
  end loop;
end;

Task body A1 is
begin
  loop
    B.Request1;
    statements
  end loop;
end;

Task body A2 is
begin
  loop
    B.Request2;
    statements
  end loop;
end;
```

Figure 1.9: An example of nested acceptance written in Ada.

```
Task body Dynamic is
atask : Task_Type;
begin
  loop
    statments
    if necessary
    then
      atask := new(Task_Type); -- create another instance of a
                               -- task of type Task_Type
    end if;
    exit when done;
  end loop;
  -- wait for all created tasks to terminate before completing
end;
```

Figure 1.10: An example of an Ada task that dynamically creates other tasks.

the languages considered in this thesis has a mechanism for creating and terminating processes during a program's execution. In each language, a process is initiated within a scope bounded by a *begin* and *end* statement. The scope can not complete until all processes within the scope have finished executing. CSP and Concurrent Pascal only allow processes to be initiated at the beginning of the scope. Ada permits tasks to be created within the scope as well. It also permits the creation of tasks that belong to an enclosing scope; enclosing scopes are not permitted to complete until the newly created tasks that belong to them have finished executing.

In figure 1.10, the Dynamic task creates instances of a task type as necessary. All of the created tasks must terminate before the task named Dynamic is permitted to terminate.

None of the languages considered provide an explicit primitive that allows **multiple serving processes** to share a common queue of customers. However, this feature can be implemented. In each of the languages considered, a process can act as a scheduler that indicates to the caller the identity of the next available server.

In figure 1.11, a multi-server is implemented in Ada. A task named Multi-Server manages a set of keys that can be used to access an array of tasks of type Server called Servers. The serving tasks indicate to Multi-Server when they are available to provide service. Customers get an identifier, named Key, for an available server from the Multi-Server and then call the appropriate server. This emulates the sharing of a single queue of customers by the servers. Unfortunately two rendezvous are required.

This completes the description of several features that are common to each of the environments. Now, a subset is chosen for analysis within this thesis. The analysis is concerned with the prediction of queueing delays that arise from the use of communication primitives. In the performance models, the system dependent overhead that is required to implement the primitives must be included in the resource demands of individual processes.

The problems to be considered are the modelling of:

- hierarchical software systems,
- the Ada style rendezvous,
- non-fixed ordered (first-come-first-served) services implemented using the Ada style rendezvous,
- condition synchronization (a producer-consumer example is considered),
- nested acceptance, and finally,
- multiple serving processes.

In the analysis of non-fixed ordered services, it is assumed that the serving process's callers are served in a first-come-first-served manner. This differs from Ada which only guarantees that each entry's callers are served in a first-come-first-served manner. A select statement can have accept statements for several entries. Ada does not define the order in which the entry queues will be examined. Therefore, a caller could be accepted that is the first in an entry's queue but not the first to be queued at the task.

```
Servers : array(1..N) of Server;        -- multiple servers

Task body Multi_Server is
Num_Server_Avail : Integer := 0;
Key_Stack : Type_Key_Stack;
begin
  for Server_Key in 1..N loop
   Servers(Server_Key).Get_My_Key(Server_Key);
  end loop;
  loop
    select
      when Num_Server_Avail > 0 =>  -- Guards evaluate Boolean expressions
      accept Get_Server_Key(Key : out Key_Type) do
        rendezvous statements
        pop(Key,Key_Stack);
        Num_Server_Avail := Num_Server_Avail - 1;
      end Get_Server_Key;
      post-rendezvous statements
    or
      accept Put_Server_Key(Key : in Key_Type) do
        rendezvous statements
        push(Key,Key_Stack);
        Num_Server_Avail := Num_Server_Avail + 1;
      end Put_Server_Key;
      post-rendezvous statements
    end select;
  end loop;
end;

Task body Server is
My_Key : Key_Type;
begin
  accept Get_My_Key(Key : IN Key_Type) do
    My_Key := Key;
  end Get_My_Key;
  loop
    Multi_Server.Put_Server_Key(My_Key); -- indicate this server is available
    accept Request do                    -- await a request for service
      server statements
    end Request;
  end loop;
end;


Task body Customer is -- sample customer body
begin
  loop
    Multi_Server.Get_Server_Key(Key);    -- get a key for a free server
    Servers(Key).Request;                -- call the appropriate server
    other statements
  end loop;
end;
```

Figure 1.11: A Multi-Server implemented in Ada.

Dynamic process creation has been considered in the literature [Heid 82, Heid 83]. An examination of the effects of fixed ordered acceptance upon performance is a topic for future research.

1.2 Performance Models of Software Processes

In this section, a method is introduced that can be used to create performance models for independent software processes. A process is described in such a way that it is possible to compute its resource requirements. The technique can be applied to processes in software process architectures as well.

The performance of independent processes has been examined by several authors [Beizer 78, Triv 82, Beizer 84, Booth 86, Qin 86, Smith 90]. As input, a process' algorithm is described in a manner that makes it possible to compute the average number of visits made by the process to each statement block or subroutine it visits. The techniques can be applied to any structured language, and even to those that contain the GOTO statement. Each of the techniques requires data dependent variables to be described statistically. For instance, the expected number of times a loop is to be executed or the probability a loop will terminate at the start or end of an iteration must be stated. Similarly the selection probability must be given for each branch of a conditional statement. Such information can be obtained from the program's designer, programmer, or from execution profiling information.

In general, a program is described using a state model. The states correspond to statement blocks. Two examples of statement blocks are the statements between the *then* and *else* in an *if-then-else-endif* statement, and the statements between the *while* and *endwhile* in a *while-loop* statement. A transition matrix can be constructed with a row and column for each state. Let a be the matrix, and let i and j indicate states. Define $a_{i,j}$ as the probability of moving from state i to state j. The sum over any one row of the $a_{i,j}$ is one. With this information it is possible to define a set of linear equations that, when solved, leads to the average number of visits to each statement block [Triv 82]. From this, the average number of visits to devices and the frequency with which processes communicate and synchronize with other processes can be derived.

When timing information is introduced, the structure of the software can also be used to estimate the mean and variance of a process' response time [Triv 82]. The time required to execute each statement block can be estimated by summing the expected time of each statement within the block. This has been called program micro-analysis [Beizer 78, Beizer 84].

In general, communication primitives are simply statements in statement blocks. The average number of times each of the primitives is executed is the number of times the statement block is executed. However, the timing delays that arise from the use of such primitives must still be considered. The techniques developed in this thesis are used to predict such delays. Once this is done, the delay times associated with communication primitives can be included in the model and the mean and variance of process response times can be estimated.

An assumption of the technique is that each conditional expression has a probability of evaluating to true, and that this probability does not change throughout the execution of the program. The assumption is unlikely to hold for any real programs but is required by this straightforward analysis. The extent that this assumption captures the true behaviour of a program affects the accuracy of the predicted response time variance and, in general, the mean response times themselves. For detailed studies of implemented systems, response time variance can be measured and used as input parameters to performance models.

1.3 Performance Models That Include Contention For Resources

Assuming that a program only uses one resource at a time, its average **response time** is the sum of its average demands for service at resources, such as CPUs and disks, and the average total queueing delays spent waiting for access to the resources. When resources are shared, queueing delays can have a significant impact upon program response times. For example, a program that requires one unit of service at the CPU may be queued behind other programs and have to wait several time units before it can receive service. Performance modelling techniques exist for predicting the performance measures of programs that share resources. Given the average demands of customers for service at the resources, the measures include:

- the throughput and average response time of each program class,
- the average queue length at each resource,
- the utilization of each resource.

In this section, two methods are described that can be used to predict the performance metrics of models. They are Queueing Network Modelling and Performance Petri Net Modelling. An overview of the computational effort required to determine the performance metrics of the models is also given. The models are discussed in detail in sections 1.3.1 and 1.3.2, respectively. Finally in section 1.3.3 some conclusions are offered regarding the applicability of the two approaches for software modelling.

Queueing network models represent computer systems as networks of queues with customers that move amongst the queues. In general, the queues are associated with resources. Classes of customers are composed of programs that have independent, but statistically identical behaviour. In general, many classes of customers can share resources.

An example of a queueing network model is shown in figure 1.12. Customers move from queue to queue along the directed arcs. Queues are represented as open rectangles and the resources as circles.

Methods exist that can be used to determine the exact performance measures for a basic class of such models that is defined in section 1.3.1. The class is called **separable models** and has properties that permit efficient solution techniques to be used to determine performance measures.

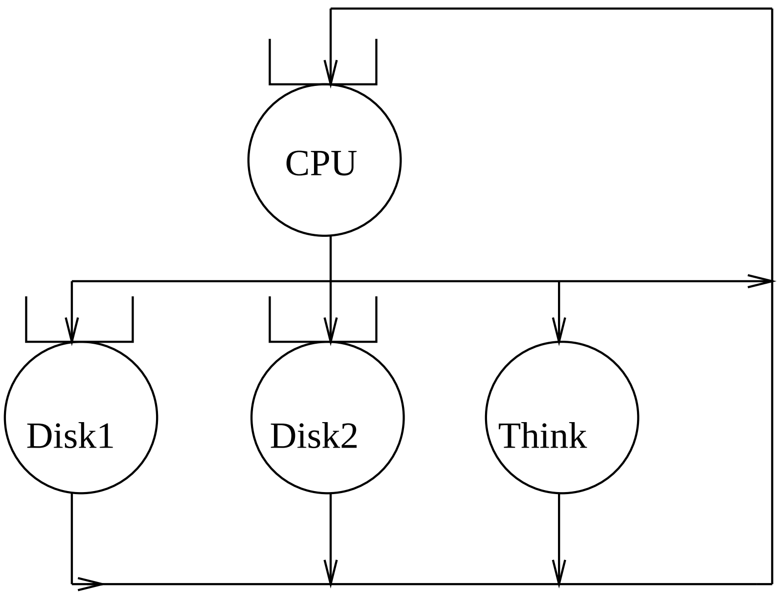


Figure 1.12: A Sample Queueing Network Model.

Unfortunately, there are many aspects of a system's behaviour that are not represented in separable queueing network models. For example, a customer may have to compete for pages of memory while it uses the CPU. Queueing network models have been extended to describe such complex behaviour. However, finding the exact performance metrics for the extended models requires much more computation than for separable models. Techniques that approximate the exact solution of such extended models exist and require much less computation than the exact approaches. They are discussed in section 1.3.1.

Petri Nets provide a general notation for expressing the relationships amongst concurrent processes. A Petri Net is a directed bipartite graph composed of places and transitions. Using directed arcs, the places are connected to transitions and transitions are connected to places. Tokens reside in places and move from place to place via arcs according to rules associated with transitions. In general, the transitions represent resources and synchronization points. Places represent queues, and tokens represent programs that synchronize and share resources. An example of a Petri Net is shown in figure 1.13. The places, transitions, and tokens are denoted as circles, rectangles, and discs (i.e. smaller circles), respectively.

Any queueing network or extended queueing network model can be represented using a Petri Net model. The converse is not true [Vernon 86]. Unfortunately, finding the performance measures of the models can require a great deal of computation. Though some classes of Petri Nets exist with low solution costs, they are no more general than the separable class of queueing network models.

Several techniques have been developed to approximate the exact results of queueing network and Petri Net models. They permit models to contain behaviour that is not present in the class of separable models. Some of the techniques are discussed in section 1.3.2.

A comparison of the space and time complexities of the modelling techniques is given in table 1.1. The comparison is based upon separable models. The queueing network modelling technique that approximates the exact performance metrics of models typically requires many iterations before the model's predicted performance metrics converge to a fixed point. The time complexity is given for one iteration of the technique, the space is reused for successive iterations.

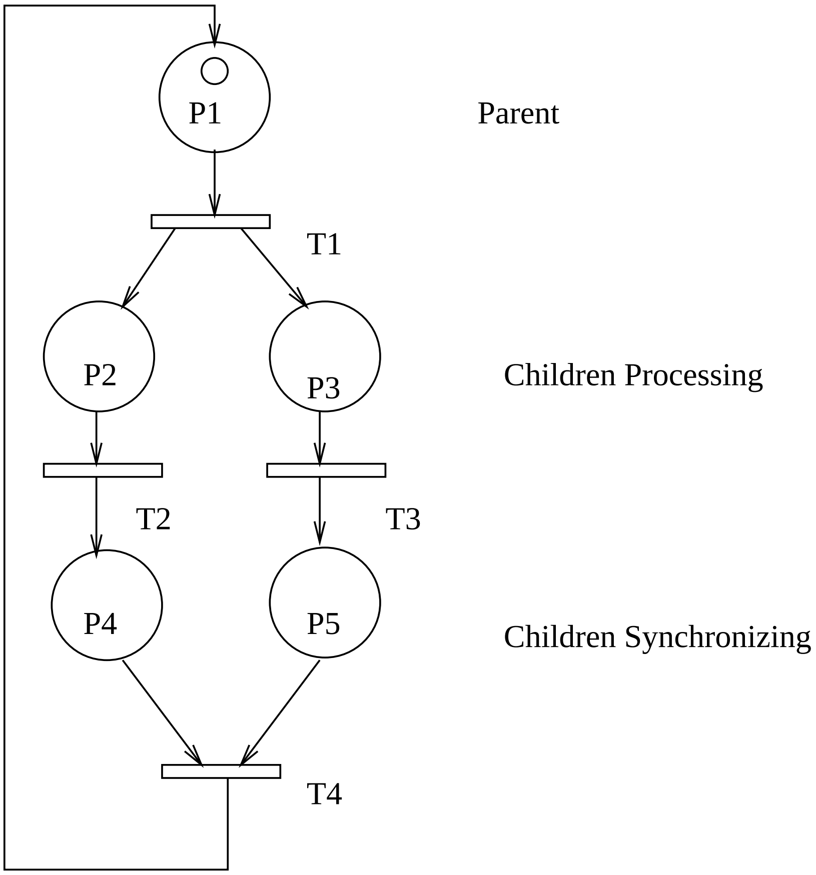


Figure 1.13: A Petri Net model of the fork and join primitive.

Method	Space Complexity	Time Complexity
Queueing Network Modelling Exact Sample Approximate QNM	$O(K\ \prod_{i=1}^{C} (N_i+1))$ $O(C^3\ K)$	$O(C\ K\ \prod_{i=1}^{C} (N_i+1))$ $O(C^2\ K)$ per iteration
Petri Net Modelling Exact	$O(\prod_{i=1}^{C} \binom{N_i+K-1}{K-1})$	$O((\prod_{i=1}^{C} \binom{N_i+K-1}{K-1})^2)$

Table 1.1: Space and Time complexities of performance prediction techniques for separable models. C is the number of classes, N_i is the number of customers in class i, and K is the number of resources.

1.3.1 Queueing Network Models

In this section an abstraction from computer systems to queueing network models is presented. The parameters that define a model are given and the techniques used to determine exact and approximate performance estimates for the models are discussed.

Computer systems are often modelled as networks of queues. For example, a program may be queued at a CPU, or at a disk while attempting to complete an I/O operation. Queueing network models can be used to describe the competition of programs for devices. Programs are the customers, and devices are the resources or **servers**.

One type of routing pattern commonly used in computer system models is the **central server** model. A customer alternates between visits to one server, usually the CPU, and the other servers. The queueing network model shown in figure 1.12 on page 26 is a central server model. Each customer visits the CPU, and then one of the disks or thinks (becomes idle). If represented in the model, passage directly from one CPU visit to another is usually interpreted as the termination of one program and the activation of a statistically identical one. Otherwise, passage from the CPU to the think resource represents a termination, and passage from the think resource to the CPU represents an activation. The pattern repeats itself indefinitely and represents **invocations** of the program. A customer is queued at exactly one server at a time and is considered to be in the queue when it receives service. The relative number of visits to each server is determined with respect to each invocation by a model's input parameters. Customers can suffer queueing delays at the servers.

Queueing network models are defined by servers, customer classes, and a description of how the classes use the servers. A **customer class** contains one or more customers that have independent yet statistically identical behaviour. A customer class can also be referred to as a **group**. If there is a fixed number of customers of the class in the model at all times, it is called a **closed class**. Closed class customers alternate between queueing for various resources and being in a queue for idle customers. The idle time is referred to as a **think time** because computer system users often initiate some system function, receive a result, then think about the result before making another request for work. Idle periods can be of duration zero. A model consisting only of closed classes is a **closed model**. If customers are better described as arriving at some rate, satisfying their service requirements, and then leaving the system, the class is called an **open class**. The arrival of an open class customer is an invocation of its corresponding program. A model consisting only of open classes is an **open model**. If both open and closed classes are present, the model is called a **mixed model**.

The following are parameters for a mixed queueing network model.

K The number of servers.

C The number of closed classes (numbered from 1 to C).

O The number of open classes (numbered from $C+1$ to O).

λ_i The arrival rate of open class i.

N_i The population of closed class i.

Z_i The average think time of closed class i.

$V_{i,k}$ The average number of visits of a class i customer to server k per invocation.

$S_{i,k}$ The average service time of a class i customer when visiting server k.

$D_{i,k}$ The average service demand of a class i customer at server k (which is the total service required by a customer at a server). By definition, $D_{i,k} = V_{i,k}\, S_{i,k}$.

The following performance measures are determined by solution techniques.

$R_{i,k}$ The average residence time of class i customers at server k. It includes both queueing and service time.

R_i The average response time of class i customers. It is defined as $\sum_{k=1}^{K} R_{i,k}$.

X_i The total throughput of class i customers (the number of closed class customers that complete their service requirements per unit time). It is defined as $\frac{N_i}{R_i+Z_i}$.

$Q_{i,k}$ The average queue length of class i customers at server k.

Q_k The average total number of customers at server k. It is defined as $\sum_{i=1}^{C} Q_{i,k} + \sum_{i=C+1}^{O} Q_{i,k}$.

$U_{i,k}$ The utilization of server k by class i customers.

U_k The total utilization of server k. It is defined as $\sum_{i=1}^{C} U_{i,k} + \sum_{i=C+1}^{O} U_{i,k}$.

It has been shown that many aspects of computer systems performance behaviour can be predicted using queueing network models that satisfy separability constraints [Reiser 78]. The accuracy of the predicted performance results seem to be robust, even when the separability assumptions are not completely valid for the system being modelled [Lazow 84]. Four queueing network model server descriptions that satisfy separability constraints have been given by Baskett et al. [Bask 75]. The first three are **queueing servers** and the last is a **DELAY server**. They are:

- The scheduling discipline is first-come-first-served (**FCFS**); all customers have the same service time distribution at the service center, and the service time distribution is a negative exponential. The service rate can be dependent on the number of customers at the server.

- The scheduling discipline is processor sharing (**PS**) (i.e., when there are n customers in the service center each is receiving service at a rate of $1/n$ sec/sec), and each class of customers may have a distinct service time distribution.

- The scheduling discipline is preemptive-resume last-come-first-served (**LCFS**), and each class of customers may have a distinct service time distribution.

- The server behaves as a service center with the number of servers in the service center being greater than or equal to the maximum number of customers that can be present at this center in any feasible state, so customers never need to wait for service. Each class of customer may have a distinct service time distribution. This scheduling discipline is called **DELAY**.

An explanation of why these servers lead to separable queueing networks is provided by Chandy et al. [Chandy 77].

The scheduling discipline of a server k is denoted as ψ_k.

One method that is used to solve for the exact solution of separable queueing network models is **exact mean value analysis** (MVA) [Reiser 78]. The method exploits the relationship amongst average queue lengths $Q_{i,k}$ and average service demands $D_{i,k}$ to compute the expected residence time $R_{i,k}$ of each customer class. **Little's law**

$$Q_{i,k} = X_{i,k} R_{i,k}$$

can be used to describe the relationship between $Q_{i,k}$ the number of class i customers in a queue k, class throughputs $X_{i,k}$ at the server, and class residence times $R_{i,k}$ at the server [Klein 75].

The following MVA algorithm computes performance measures for separable models with closed customer classes. Algorithms exist for models that contain both open and closed classes [Lazow 84].

Algorithm MVA For Closed Queueing Network Models [Reiser 78, Lazow 84]

Let $\vec{n}$ be a population vector that describes the population levels of the closed classes. For example, n_1 is the population of closed class 1, $\vec{n_1}$ is the number of class one customers in population vector $\vec{n}$, $Q_k(\vec{n})$ is the average total queue length at server k for customer population vector n, and $Q_k(\vec{n} - 1_c)$ is the average total queue length at server k for customer population vector $\vec{n}$ with one class c customer removed. A population vector is feasible if and only if $\vec{n_i} \leq n_i$ for all i.

for $k \leftarrow 1$ to K do $Q_k(\vec{0}) \leftarrow 0$

for $p \leftarrow 1$ to $\sum_i^C n_i$ do

for each feasible population $\vec{n}$ with a total of p customers do

begin

for $c \leftarrow 1$ to C do

for $k \leftarrow 1$ to K do

$$R_{c,k} \leftarrow \begin{cases} V_{c,k}\ S_{c,k} & \text{Delay Server} \\ V_{c,k}\ S_{c,k}(1 + Q_k(\vec{n} - 1_c)) & \text{Queueing Server} \end{cases}$$

for $c \leftarrow 1$ to C do

$X_c \leftarrow n_c/(Z_c + \sum_{k=1}^{K} R_{c,k})$

for $k \leftarrow 1$ to K do

$Q_k(\vec{n}) \leftarrow \sum_{c=1}^{C} X_c\ R_{c,k}$

end

End algorithm

To better understand the MVA algorithm, consider the case where there is only one customer class in the model. MVA makes use of the following **residence time** expression for server satisfying separability constraints:

$$R_k(n) = D_k(1 + A_k(n)).$$

$R_k(n)$ is the residence time of a customer at server k when there are n customers in the model. D_k is the service demand for customers at server k. Finally, $A_k(n)$ is the average number of customers that are ahead of an arriving customer, at server k, at the instant of its arrival.

The **Arrival Instant Theorem** (AIT) enables the computation of $A_k(n)$. In open models $A_k(n) = Q_k(n)$. In closed models $A_k(n) = Q_k(n-1)$ [Sevcik 81]. As a consequence, the analysis of a closed model requires the enumeration of each customer population. In the multiple class case, the algorithm considers queue lengths when there are no customers in the model and then adds one customer from a class at a time until the full customer population level is reached for each class. All possible customer population vectors are considered.

Unfortunately, the cost of solving models becomes prohibitive quickly as the number of classes and their respective populations grow. A model with six customer classes, each with nine customers, and ten servers requires over one hundred million operations. Many systems treated by analytic modelling have several job classes and high populations. If predicting the performance of the models requires too much time, it limits an analyst's ability to consider a system's performance for a wide range of circumstances. This is why techniques that approximate the exact performance metrics of models but require less space and time are needed.

Approximate MVA algorithms provide accurate performance estimates quickly for queueing network models. Typically, such algorithms approximate the exact solution of separable systems by estimating the solution of the residence time expression near and at the full customer population levels [Chandy 82, Zahor 88].

The solution complexities of the approximating algorithms are determined by the number of classes, not the class populations. The loss in accuracy from moving to techniques that only approximate the exact solution of a model is usually much lower than the error inherent in representing a system using an analytical model. The algorithms are iterative in nature and terminate when the predicted measures converge to a fixed point. Though there is no proven bound on the number of iterations required to reach convergence, empirically, the computation required by such techniques is significantly lower than for exact techniques when there are more than a few customers or servers. Each iteration's complexity, or the **iteration-wise time complexity**, is polynomially bounded with respect to the number of customer classes in the model. In each case, the space complexity is polynomially bounded with respect to the number of customer classes in the model.

The most significant differences amongst the approximating techniques are the range of customer populations that are considered and the methods used to predict the arrival instant queue lengths. Approximate MVA techniques are typically initialized by assuming a uniform distribution of customers across the queues. With the initial queue length estimates, the residence time expressions are evaluated. New customer response time estimates are found. Using Little's Law, new estimates for queue lengths are obtained. The technique can be applied iteratively until successive residence time or queue length estimates differ by less than some specified tolerance.

Approximate MVA (AMVA) [Lazow 84] is based upon the Bard-Schweitzer analysis [Bard 79, Schwe 79] which considers the model at full population, and at each population that differs from the full population by having one fewer customer in exactly one class. The Aggregate Queue Length algorithm (AQL) [Zahor 88] considers the model at the full population, and at each population where one or two customers are removed. The algorithm maintains aggregate queue lengths.

The methods presented within this thesis have been developed using Linearizer [Chandy 82]. The techniques could be adapted to other MVA algorithms as well. However, an exact technique would no longer be exact.

Linearizer estimates the arrival instant queue length for each server at the customer populations $\vec{N}$, $\vec{N}-1_c$, $\vec{N}-1_c-1_d$, where c and d are customer classes. The algorithm is presented here and uses the following notation.

$F_{c,d,s}(\vec{N})$ Represents the change in the fraction of class c customers at server s resulting from the removal of a class d customer.

$Q_{c,s}(\vec{N})$ The number of class c customers in queue s at population $\vec{N}$.

$(\vec{N}-1_d)_c$ Represents the population of class c after the removal of one class d customer from the system.

I Linearizer iteration index.

J Core subroutine iteration index.

The constants within the algorithm are as suggested by Chandy and Neuse [Chandy 82].

Algorithm Linearizer

Inputs: Closed MVA input parameters.

Outputs: $R_c(\vec{N})$.

Subroutine Core
Inputs: $C, K, \vec{N}, F_{c,d,s}(\vec{N}) \; \forall \; c, d, s, Q_{c,s}(\vec{N}) \; \forall \; c, s, D_{c,s} \; \forall \; c, s, Z_c \; \forall \; c$
Outputs: $Q_{c,s}(\vec{N}) \; \forall \; c, s, R_{c,s}(\vec{N}) \; \forall \; c, s,$ and $X_c(\vec{N}) \; \forall \; c$
Note: $Q_{c,s}(\vec{N} - 1_d) \; \forall \; c, d, s,$ are local storage to this routine.
Begin Core

$J \leftarrow 0$
REPEAT
  $J \leftarrow J + 1$
  $Q_{c,s}(\vec{N} - 1_d) \leftarrow (\vec{N} - 1_d)(Q_{c,s}(\vec{N})/N_c + F_{c,d,s}(\vec{N})) \; \forall \; c, d, s$
  Update Throughputs and Utilizations for population $\vec{N}$
  Core Support (For thesis use.)
  FOR ALL s CASE ψ_s OF
    Queueing : $R_{c,s}(\vec{N}) \leftarrow D_{c,s}(1 + Q_{c,s}(\vec{N} - 1_c)) \; \forall c$
    DELAY : $R_{c,s}(\vec{N}) \leftarrow D_{c,s} \; \forall c$
    Others: other residence time expressions
  ENDCASE
  $Q_{c,s}(\vec{N}) \leftarrow N_c \; R_{c,s}(\vec{N})/(\sum_{k=1}^{K} R_{c,k}(\vec{N}) + Z_c) \qquad \forall \; c, s$
UNTIL $max_{\forall \; c,s} |Q_{c,s}(Iter_J) - Q_{c,s}(Iter_{J+1})|/N_c \; < J/(4000 + 16 \; |\vec{N}|)$
$X_{c,s}(\vec{N}) \leftarrow Q_{c,s}(\vec{N})/R_{c,s}(\vec{N}) \qquad \forall \; c, s$

End Subroutine Core

Begin Linearizer

$Q_{c,s}(\vec{N}) \leftarrow N_c/K \qquad \forall \; c, s$
$Q_{c,s}(\vec{N} - 1_d) \leftarrow (\vec{N} - 1_d)_c/K \qquad \forall \; c, d, s$
$F_{c,d,s}(\vec{N}) \leftarrow 0 \qquad \forall \; c, d, s$
Initialize (For thesis use.)
FOR $I \leftarrow 1$ to 3 DO
  call core subroutine with $C, K, \vec{N}, F(\vec{N}), Q(\vec{N}),$
    returns $Q(\vec{N}), R(\vec{N}), X(\vec{N})$.
  IF $I \neq 3$
  THEN
    call core subroutine with $C, K, \vec{N} - 1_c, F(\vec{N}), Q(\vec{N} - 1_c),$
    returns $Q(\vec{N} - 1_c), R(\vec{N} - 1_c), X(\vec{N} - 1_c)$.
    $F_{c,d,s}(\vec{N}) \leftarrow Q_{c,s}(\vec{N} - 1_d)/N_c - Q_{c,s}(\vec{N})/N_c \qquad \forall \; c, d, s$
  END IF
END FOR
$R_c(\vec{N}) \leftarrow \sum_{s=1}^{K} R_{c,s}(\vec{N}) + Z_c \qquad \forall \; c$

End Algorithm Linearizer

In Linearizer, $F_{c,d,s}(\vec{N})$ is assumed to be equal to $F_{c,d,s}(\vec{N}-1_e)$ where e is any customer class. This is where the name Linearizer comes from. It implies that the fraction of class c customers at server s is a linear function of population. The throughputs and utilizations are updated using the standard formulas. Two statements have been included (Initialize and Core Support) that will be used later in the thesis. They are not part of the original Linearizer algorithm but will contain the modifications necessary to support the analysis developed in this thesis.

The algorithm presented has an iteration-wise time complexity of $O(C^3\ K)$ but with some minor changes that do not affect its performance estimates, it can be reduced to $O(C^2\ K)$ [Silva 90]. The termination criteria for the *repeat* loop in the core subroutine was chosen so that the algorithm would not terminate too early if convergence to a fixed point is proceeding slowly. Methods used to recognize slow convergence and speed up convergence for approximate MVA have been considered by Zahorjan et al. [Zahor 88].

Table 1.2 summarizes the complexities of MVA and three approximate MVA techniques. Each time complexity gives an indication of the range of customer populations that is being considered. For the approximating techniques, it has been observed that the greater the space and time complexity the more accurate the results with respect to exact results.

Method	Space Complexity	Time Complexity
Exact MVA	$O(K\ \prod_{i=1}^{C}(N_i+1))$	$O(C\ K\ \prod_{i=1}^{C}(N_i+1))$
Bard-Schweitzer AMVA	$O(C\ K)$	$O(C\ K)$ per iteration
AQL	$O(C\ K)$	$O(C^2\ K)$ per iteration
Linearizer	$O(C^3\ K)$	$O(C^2\ K)$ per iteration

Table 1.2: Space and Time complexities of closed separable model solution techniques. C is the number of classes, N_i is the number of customers in class i, and K is the number of servers.

The availability of approximate MVA techniques makes MVA an attractive technology for the study of separable models. Unfortunately, systems may exhibit behaviour that is not well predicted under separability assumptions. Various techniques have been developed that extend the class of systems to which MVA techniques can be applied. The following characteristics have been considered:

- Customer classes have different average service times at a first-come-first-served server.
- One customer class has priority over another at one or more servers.
- A customer must control several servers at a time, for example during data transfers.
- Customers must acquire memory before competing for other resources.

In analytical terms, these examples correspond to the following situations: first-come-first-served scheduling with distinct and possibly non-exponential service time distributions, prioritized scheduling policies, simultaneous resource possession, and memory constraints.

Methods have been developed to predict non-separable behaviour within the queueing network modelling framework. Often the MVA residence time expression is manipulated to reflect the behaviour under study. At other times the results of several models that are assumed to satisfy separability constraints are combined to provide performance estimates for the original model.

Hierarchical modelling has been developed to assist in approximating the exact solution for networks that contain non-separable behaviour [Chandy 75, Lazow 84]. The servers of a model can be partitioned into a tree of submodels that are more readily solved than the original model. Each submodel can be represented as a **flow equivalent** service center (FESC) that reflects the combined behaviour of a set of servers. The term flow equivalence implies that, for a given set of customers, the service times for the customers at the FESC are the same as the average interdeparture times for the customers at the set of servers it replaces. In the tree of submodels, the root submodel describes the behaviour of the entire system, and the leaves represent submodels that no longer contain FESCs and thus have performance estimates that can be measured, simulated, or easily computed. When solving the hierarchical model, the behaviour described by the leaves is used to determine the performance estimates for their parent submodels. The techniques used to do this are applied recursively until the metrics for the root, and hence the entire model, are known [Lazow 84].

Reiser [Reiser 79] provides an analytical expression that is used to model first-come-first-served scheduling disciplines with general service time distributions. The technique is also applicable when modelling multiple class first-come-first-served scheduling with distinct mean service times. The method alters the residence time expression to estimate a server's unfinished work based upon its expected queue length and each of the customer classes' service requirements and the utilization of the server. The amount of unfinished work determines the arriving customer's queueing delay.

Eager and Lipscomb consider the use of priorities in multiple class closed models. Several priority server heuristics that had been developed for use with exact MVA [Bryant 84] are converted for use with AQL [Eager 88]. Note that the heuristics, even when used with exact MVA, only approximate exact results. The accuracy of the methods are compared with exact results and with the results of heuristics when used with exact MVA. It is shown that approximate MVA based priority approximations have accuracy and behaviour comparable to priority approximations used with exact MVA approaches.

In many systems a customer may require more than one resource at a time. Jacobson and Lazowska [Jacob 82] introduce the **method of surrogate delays**. The method involves using multiple models to analyse such systems. For example, disks in an I/O system may have to compete for the use of a channel to complete their data transfers to or from memory. In such a situation, two models are used. The first represents the disks as delay centers and the channel as a queueing center; the second represents the channel as a delay center and the disks as queueing centers. The method iterates between the two models with the queueing delay estimates in one model being used to parameterize the delay centers in the other model.

Memory management can also have a significant impact upon performance. Customers may compete for simultaneous access to both memory and the other resources they require. Lazowska et al. [Lazow 84] describe several techniques for modelling systems in which memory contention is a factor.

Lavenberg provides a survey of analytical performance modelling techniques for computer systems [Laven 89].

1.3.2 Performance Petri-nets

The Petri Net (PN) notation [Petri 66] permits the representation of very general forms of concurrency and synchronization. Several types of PN models can be used for performance modelling. A Petri Net model of the *fork* and *join* construct is provided as an example. Unless constrained, the behaviour of a Petri Net model must be analysed using computationally expensive techniques. Several methods that attempt to increase the solution efficiency for certain classes of models will be discussed here.

Petri Net models are a collection of **places**, denoted by circles, and **transitions**, denoted by bars, that are linked together by **directed arcs**. Arcs connect places to transitions, and transitions to places. A set of **tokens**, denoted by small discs, is partitioned across the places. Token movement is governed by the transitions. When a token is available on each input place of a transition, the transition is said to be **enabled**, the transition **fires**, the tokens are consumed and a token is deposited in each of the output places of the transition. Synchronization is modelled using transitions with more than one input place. A **marking** of a Petri Net describes the number of tokens in each place. Given an initial marking, the set of reachable markings can be enumerated to define the Petri Net state space.

In **Performance Petri Nets**, transitions can be associated with time. After a transition is enabled, some time passes before the transition fires. The time is a sample from a service time distribution that is associated with the transition.

There are many classes of performance Petri Net models. Each has its own area of application. Two examples are Stochastic Petri Nets (SPN) [Molloy 82] and Generalized Stochastic Petri Nets (GSPN) [Marsan 84]. Both are capable of representing timing and synchronization. SPNs have transitions with firing times that are exponentially distributed. In GSPNs transitions have firing delays that are either deterministically equal to zero (that is, **immediate**), or are exponentially distributed (that is, **timed**). The immediate transitions have priority over timed transitions. For some GSPN models it is also necessary to specify the priority amongst immediate transitions. The class of Petri Net models that is most appropriate depends upon the modelling features that are required for an accurate description of the system under study.

The following example illustrates an SPN of the *fork* and *join* construct. The initial system state was shown in figure 1.13 on page 28. Since the single token is in place $P1$ the parent process is receiving service. The parent process requires a certain amount of service before it forks into two child processes. After the fork, two tokens exist, one in place $P2$ and one in place $P3$. Each of the tokens represents a child process. A child token advances to its next place once its service requirement is satisfied and then waits for synchronization. Once both have finished their requirements, synchronization occurs and control is passed back to the parent in the form of a single token.

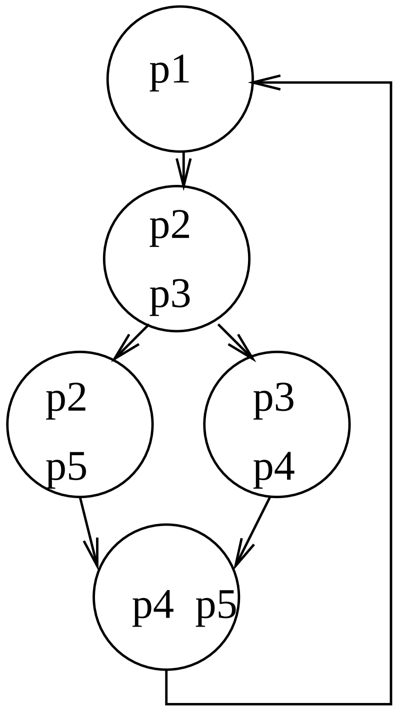


Figure 1.14: A State Model of the Petri Net in figure 1.13.

Often, places are used to represent the context of a token that resides at that place. A state space can be enumerated using one state for each feasible placement of the set of tokens (marking) in the net. The state space for the *fork* and *join* Petri Net shown in figure 1.13 is shown in figure 1.14. A system of linear equations can be constructed that uses timing information to determine the rate of movement from one state to the next. The equations can be solved to discover the equilibrium probability of residing in each state [Molloy 82]. Such solution techniques are called **Global Balance** techniques because they balance the flow into and out of states.

The state residence probabilities are used to find performance measures for the net that are similar to those provided for queueing network models. Some examples are resource utilizations, queue lengths, and customer response times.

The state space must be enumerated for this solution technique; hence the solution cost can be prohibitively large even for small nets. Let T be the maximum number of tokens in the net, and P be the number of places. For the solution of both SPNs and GSPNs, the space and time complexities required are, respectively,

$$O\left(\begin{pmatrix} T+P-1 \\ P-1 \end{pmatrix}\right),$$

and

$$O\left(\begin{pmatrix} T+P-1 \\ P-1 \end{pmatrix}^2\right).$$

To explain the time complexity, consider the set of linear equations that corresponds to the state space of size n. Solving the system of n equations requires at least $O(n^2)$ operations.

Some researchers use classes of Petri Net models that have efficient solution techniques or for which algorithms exist that approximate the exact solution. Smith [Smith 85] and Balbo [Balbo 86] each present hybrid techniques that decompose models into parts that can be solved efficiently and other parts that must be solved using the global balance techniques. The methods approximate the exact results of the model.

Smith [Smith 85] uses deterministically timed PNs to study pipelined and systolic processor architectures. In the specific class PNs that were considered, a transition can remove and add more than one token from its input and to its output places, respectively. The number of tokens to remove and add are specified as input parameters for the PN. The PNs are decomposed into three classes of nets: acyclic, cycles with data flow balance within the cycle (all tokens added by a transition are removed by the next transition), and cycles with global balance (the tokens added by a transition need not be removed by the next transition but are eventually removed later in the cycle). Efficient algorithms are provided that can be used to solve for the maximum throughput of the acyclic parts and the parts with cycles that have data flow balance within the cycle. Global balance techniques for stochastic PNs are used to estimate the throughputs of the remaining parts.

Balbo et al. [Balbo 86] use a combination of Queueing Network and Petri Net models to examine a class of systems that have software serialization delays. The GSPNs are used to represent features of the model that cannot be treated directly by means of separable queueing networks. Unfortunately, the GSPN model is still solved using global balance techniques, so it is possible to examine only small models using this technique.

In chapter 5 of this thesis it is shown that a class of GSPNs can be solved using the approximate MVA techniques developed in this thesis. The new method does not enumerate a state space yet provides accurate performance estimates for the models presented in the paper by Balbo et al. [Balbo 86].

1.3.3 Approach Used In This Thesis

Petri Nets offer the ability to express many complex forms of synchronization. Unfortunately, the procedure for predicting the performance of the models considered in this thesis requires a Global Balance technique that prohibits the solution of all but the smallest models. For this reason, a Mean Value Analysis approach has been used for all of the analysis presented in this thesis.

1.4 Performance Models Of Software Process Architectures

With the advent of distributed and multi-processor computing, systems of cooperating processes have become an attractive alternative to monolithic programs. In such systems it is often possible to alter the degree of parallelism within a program, and for the parallel threads to synchronize and communicate. The structure of process creation, termination, and synchronization describes a system's software process architecture. Amongst the possible process creation and synchronization strategies are the *fork*, *join*, and *rendezvous* primitives. Though the primitives may not explicitly exist in all systems environments, the behaviour they can cause occurs in each of the software environments described in section 1.1. In particular this behaviour includes:

- software synchronization and/or queueing delays (due to the semantics of primitives)
- systematic changes in the sets of processes that compete for devices (due to the interactions amongst primitives)

In this section, the primitives are described and a review of approximate mean value analysis modelling techniques for the primitives is given.

A process that issues a **fork operation** dynamically creates one or more child processes and, depending on the implementation, can then suspend itself using the **join operation**. The process is the parent of the child processes. All of the child processes execute concurrently and when they complete the parent (if suspended by a *join*) can continue its execution. In Concurrent Pascal and CSP the same behaviour is caused by *cobegin* and *coend* statements. In Ada, master scopes [DOD 83] synchronize the completion of tasks.

With the **rendezvous**, processes can act both as customers and servers. A process requests service from another process that acts as a server. However, the serving process might not be able to provide service to a requester at the time of the request.

Thus, requesters may suffer from queueing delays while waiting for service. The serving process is said to have rendezvous and post-rendezvous service. The requester must wait until the server provides the rendezvous service before it can continue its own execution. When the server completes its post-rendezvous processing, it is ready to provide service to another requester. A server is idle if it is ready to provide service but no requester is available.

There are two major performance issues introduced by interprocess communication. The first is due to the **semantic effects** of the primitive being used. For instance with the *fork* and *join*, a process may spawn several processes at the same time. The expected time for all processes to complete must be found. Similarly, with the rendezvous it is necessary to predict the queueing delays a customer will incur while attempting to obtain service from a shared software server. The next problem is due to the **interaction effects** of such primitives. Parallelism and synchronization amongst processes constrains the groups of processes that compete for system resources, such as processors, at the same time. This can have a significant effect upon the amount of contention each of the processes encounters when accessing system resources. The degree of contention may be different for each process and can vary depending upon the state of other processes.

These effects are often addressed in the solution techniques for models of systems by partitioning the models into submodels whose performance estimates can be found using efficient algorithms. The results of the submodels can then be combined to provide performance estimates for the entire model. Initial estimates for the input parameters of each submodel are used to solve the submodels separately. The performance estimates from one submodel can often be used to provide better estimates for the input parameters of the other submodels. Using this approach, the submodels can be solved iteratively until the input parameters of each submodel are consistent with the output parameters of the other submodels. When this occurs, the performance estimates for the submodels and the entire model have reached a fixed point and performance estimates can be reported.

There are many examples of this methodology being used in the literature and several are discussed below. Convergence properties are rarely discussed in detail but are affected by the following factors:

- the accuracy of the performance estimates for submodels and the convergence properties of their solution techniques.
- the combined accuracy of the techniques used to provide refined estimates for submodel input parameters based on submodel output parameters.
- the rate at which the performance estimates move towards a fixed point and the termination criteria for the iteration.

Empirical evidence is frequently used to assert that an iterative algorithm converges. Most of the techniques have not been proven to be guaranteed to converge.

The *fork* primitive is studied by Heidelberger and Trivedi [Heid 82] using modified queueing network models. In addition to the standard input and output parameters for mixed class QNMs, a closed parent customer class is associated with an open child customer class that represents processes that are spawned by parent customers. This relationship affects the output performance measures for the model. In this model, the parent does not suspend itself; no join statement is issued. The throughput of a child process class is determined by the throughput of its parent class. Since all of the processes compete for system resources, the throughputs must be computed using an iterative algorithm. The algorithm terminates when the throughput of the parent workload and the rate at which the child processes are spawned match.

Heidelberger and Trivedi propose the **decomposition approximation method** to consider models of systems that contain the *fork* and *join* primitives [Heid 83]. The method uses a closed queueing network model and a Markov chain. The parent and child processes are both represented as closed customer classes in a QNM with standard parameters. The Markov model is used to represent the effects of the *fork* and *join* and to help determine the performance estimates for the model.

Using a single application of exact MVA, a queueing network model is solved to determine performance measures for all populations of closed classes. A Markov model is created in which states represent populations of closed class customers that may compete for system resources. The state space is determined by the range of populations that are feasible given the *fork* and *join* relationships between customer classes. The transition rates amongst states are estimated by the results of the MVA at the state's corresponding population. The Markov model is solved using Global Balance techniques to determine the portion of time that each feasible population competes for devices. Finally, the results of the MVA are weighted by the portion of time that each population competes for devices to provide system performance estimates.

To avoid the need for exact MVA and Markov models, Heidelberger and Trivedi propose the **method of complementary delays**. The method uses a closed queueing network model that is augmented by a fictitious server. The parent and child processes make demands at the server equal to the times that the parent and child customer classes do not spend competing for non-fictitious servers. The method is similar to the method of surrogate delays. The total idle time of each process is estimated. In the case of parent processes, idle time includes time waiting for the child processes to complete. For a child, idle time is the average time between its completion and its parent's next fork command. The results of the queueing network model are used to provide better estimates for process idle times. The method is applied until successive process average response time estimates reach a fixed point.

The decomposition approximation appears to provide results that are much more accurate with respect to simulation than those of the method of complementary delays. However for large systems only the latter technique remains feasible.

The approaches developed by Heidelberger and Trivedi for dealing with the fork and join can be used to model dynamic process creation as described in section 1.1.4. The analysis addresses the problems of waiting for children to complete their execution and the interaction effects of changing rates at which children receive service. These problems are not considered further in this thesis.

The analysis developed in this thesis deals with synchronization and queueing delays that occur among a fixed set of processes. Rendezvous type interactions change the nature of systems of processes. With the *fork* and *join*, spawned processes provide dedicated service to their parents. In the rendezvous environment, processes can act as both customers and servers. As a result, a process that requests service from another process may suffer from a queueing delay. Queueing delays in such systems must be quantified if process response times are to be accurately modelled.

Woodside [Woods 86] proposes the **Stochastic Rendezvous Network** (SRVN) algorithm for predicting the throughput of systems of tasks that each have dedicated devices and request service from one another using the rendezvous. The model input parameters are similar to those of closed queueing network models with each task being represented by a customer class. The model input parameters are augmented to consider the following.

- The task service demand descriptions must be specified for both rendezvous and post-rendezvous service phases if they exist (the number of phases of service in a task i is denoted as P_i). The phases are central server systems. The QNM parameters for the service requirements of a task are subscripted with a p to indicate the phase to which they apply.

- Tasks can make visits (request service) to other tasks (service times are not specified, they are computed by the algorithm). As a result tasks can act as both customers and servers. For example, $V_{p,g,h}$ is the average number of visits by task g in its phase p to task h.

- All devices are assumed to be DELAY centers.

The output for the model is the set of task throughputs. The SRVN method uses a form of mean value analysis that can be compared with the Bard-Schweitzer approximate

MVA. The time complexity of each iteration is $O(N^3)$, where N is the number of processes.

Consider an software process architecture with N tasks that is being modelled using a SRVN. Assume that there are no cycles of requests for service. That is, a task that requests service from another task must never, directly or indirectly, provide service to that task. Tasks that do not provide service to any other tasks are defined as reference tasks and are numbered 1 to R. Those tasks that do provide service are numbered $R+1$ to N.

Woodside recognizes that, because of the synchronization of tasks, serving task throughputs satisfy the following set of equations:

$$X_j = \sum_{i=1}^{N} \sum_{p=1}^{P_i} V_{p,i,j} \, X_i \qquad j = R+1 \text{ to } j = N.$$

This can be reduced to:

$$X_j = \sum_{i=1}^{R} a_{i,j} \, X_i \qquad j = R+1 \text{ to } j = N,$$

where the $a_{i,j}$ can be found using Gaussian elimination or simply row reduction. Thus serving task throughputs are dependent on reference task throughputs and can be assigned in terms of reference task throughputs.

The throughput of a reference task is defined as the inverse of the sum of its phase response times $R_{p,i}$. The $R_{p,i}$ are defined as the average time required for one invocation of the phase:

$$X_i = \frac{1}{\sum_{p=1}^{P_i} R_{p,i}}.$$

The following method is used to compute the phase response times of tasks in the model. Since there is always one task per class there is no notion of intermediate population vectors.

The residence time expression for a visit by task i in its phase p to task j is:

$$R_{p,i,j} = w1 + w2 + w3$$

where

$w1$	$=$	the mean waiting time (if any) for the current phase of task j to complete
$w2$	$=$	waiting of task i for other tasks in task j's queue, following $w1$
$w3$	$=$	the service time of task j's rendezvous phase

The $w1$ term is the mean residual life of the task in service at the arrival of task i. The term takes into account the fact that the caller is more likely to arrive when the serving task is visiting a server with a long service time than a short one. It also includes the time that is spent by task i waiting for its own post-rendezvous service to complete.

The phase p response time of task i is

$$R_{p,i} = \sum_{j=R+1}^{N} R_{p,i,j}.$$

The SRVN approach is similar to the Bard-Schweitzer analysis because the $w2$ term is computed with the arriving task's contribution to server usage removed from the equation. Other mean value analysis algorithms such as Linearizer introduce into the analysis effects caused by the removal of a customer on other customers' use of the server. This has the effect of improving the estimates for customer response times and server utilizations.

For each iteration, the values of $R_{p,i}$ are computed for tasks in order 1 through N. These values lead to improved throughput estimates for the reference tasks 1 through R. Afterwards, better throughput estimates for the non-reference tasks $R+1$ to N are computed using their relationship, $a_{i,j}$, with the reference tasks. In this way the throughput estimates remain consistent with the response time estimates. The computation of the $R_{p,i}$ and then the throughput of the non-reference tasks is repeated until the throughput estimates for the reference tasks converge to a fixed point.

Devices can be included in the model by representing them as other tasks. The residence time expressions are updated to reflect the special nature of the device. For example, processor-sharing, and priority-preemptive-resume [Bryant 84] have been adapted for use with SRVNs [Mier 89].

When a task has multiple entries [Mier 88], there is no special treatment of the high service time variation at the task. A task's residence time at a multiple-entry server is the weighted sum, based on visit ratios, of the residence time expressions for each entry. The weight with which customers visit each entry of the multiple entry task determines the multiple entry task's service requirements when it acts as a customer.

Rolia [Rolia 87] uses closed queueing networks to analyse a restricted set of rendezvous networks that permit shared devices. The parameters for the model are the same as for the SRVN model but the solution technique differs. The algorithm is called the **Method of Layers**[1] (MOL). Models are made of processes that request and provide service to one another. The method decomposes a given model into several two-level models that can be solved using MVA techniques. The initial implementation did not support post-rendezvous processing, but this limitation has subsequently been removed and is discussed in chapter 3.

The given system of processes is mapped onto a hierarchical model in which processes request service only from processes one level lower in the hierarchy. Each pair of successive levels in the hierarchy defines a submodel. The response time of a process that is considered as a customer in one submodel defines its service time when it is considered as a server in its alternate submodel. The solution of all of the submodels represents one iteration of the algorithm. The algorithm is applied iteratively, until the changes in estimated average response times for processes between successive iterations is below a specified tolerance.

Linearizer is used as the MVA technique for solving the queueing network models. The time complexity of the MOL software submodel solutions is $O(L\ N^3)$ per iteration, where L is the number of software levels and N is the number of processes. The time complexity is higher than the SRVN technique by a factor of L. This is because the

[1] The method was originally named the Lazy Boss Method

MOL uses Linearizer instead of a Bard-Schweitzer like AMVA routine. If the MOL used AMVA, the time complexity would only be $O(L\ N^2)$ which is lower than SRVN. Since the time complexity of Linearizer is greater than the Bard-Schweitzer algorithm by a factor of N, and since L is usually much lower than N, the MOL's divide and conquer approach is allowing the use of more accurate MVA software for a small increase in time complexity.

When solving the hierarchy of models, it is assumed that each process has dedicated devices so only software contention is considered. After the algorithm terminates, a second model, in which software contention is ignored, is created to estimate the responsiveness of devices for each process.

A method similar to the method of complementary delays is used to introduce device contention into models [Rolia 88]. Each process is included in a single device contention queueing network model. The amount of time that a process does not spend competing for devices is used as a think time in the model. The iteration-wise complexity of the device responsiveness model is $O(N^2\ K)$, where N is the number of processes and K is the number of devices[2].

With new estimates of device responsiveness, another solution for the software contention model is found. The MOL alternates between software and device contention models until the estimates for average response times of non-serving processes differ by less than some tolerance. The convergence properties of the technique are discussed in chapter two.

1.5 Thesis Structure

This thesis consists of seven chapters. The first has provided a review of the current software modelling literature and an introduction to the different types of behaviour that are to be considered.

In chapter two the Method of Layers (MOL) technique is presented for predicting

[2] If the interaction effects are significant and a more accurate approach is required, the decomposition approximation method [Heid 83] is appropriate.

the performance of hierarchical software systems. Examples are shown in which devices have processor-sharing and priority-preemptive-resume (PPR) scheduled CPUs. At a PPR device, customers with the same priority are served in a first-come-first-served manner, but are preempted by higher priority customers. The accuracy of the MOL is compared with the SRVN method of Miernik et al. for the models considered in their paper [Mier 89]. The software modelling techniques that are developed in subsequent chapters can also be used in conjunction with the MOL.

In chapters three and four, techniques are presented that model the following features: the Ada rendezvous, multiple entries (first-come-first-served), a multi-server, nested accept statements, and a producer-consumer server. Each of the techniques has been validated with respect to simulation for a large range of model parameters. The accuracy of the MOL with the rendezvous, and multiple entry server techniques is compared with that of Woodside's et al. technique for the models considered in their papers [Woods 86, Mier 88].

The generality of the MOL approach is demonstrated in chapter five as it is used to provide performance estimates for a class of Generalized Stochastic Petri Nets. The techniques are then applied to two Ada applications in chapter six. The applications were implemented using Ada/6000 [3] on an IBM RS/6000 running the AIX operating system.

Finally, chapter seven offers conclusions about the advances that have been made and the problems that remain in carrying further the work originated in the thesis.

[3] Ada/6000, IBM, and AIX are all registered trademarks of the International Business Machines Corporation of Armonk New York, U.S.A

Chapter 2

Systems With Software Servers

Hierarchical software systems have been developed that contain one or more layers of software servers. Processes within such systems not only suffer contention delays for shared hardware but also at the software servers. The Method Of Layers (MOL) [Rolia 87, Rolia 88] is proposed to provide performance estimates for such systems. The MOL uses mean value analysis (MVA) to assist in predicting model performance measures.

The input parameters for the models are very similar to those for standard closed queueing network models. The difference is that processes can act as both customers and servers so that it is possible for one process to visit another. Thus the average number of visits a process makes to other processes must also be specified. When features studied in subsequent chapters require new parameters they are introduced with the analysis.

The performance measures that are predicted are the same as for closed QNMs. In addition each process has the following new measures.

- Average residence time: the average time that passes between the time a process requests service from another process and the time it can continue processing.

- Utilization: the percentage of time that the process is involved in computation.

The requests for service amongst processes can be described as a graph. Processes that do not request service from other processes are at the lowest levels and other processes are at higher levels. It is assumed that there are no cycles in the graph so that the model's requests for service are **acyclic**. Models with cyclic request graphs have not been considered.

The average response times of processes at one level of the model are estimated by viewing the processes at the next lower level as servers in a separable queueing network model. Processes that request service are considered as customer classes and those that provide service are represented as servers. The response time for process communication via the remote procedure call primitive has been estimated as the residence time at a first-come-first-served server [Rolia 87]. This representation captures the possible queueing delays incurred by the processes requesting service if the serving process is busy doing work on behalf of another calling process. Another queueing network model is used to determine queueing delays at devices. In this model, each process in the system is represented as a customer, and each device as a server. The results of the software and device contention models are combined, in a manner similar to the method of complimentary delays, to provide performance estimates for the entire system.

Processes that do not provide service to other processes are defined as **non-serving** processes, other processes are defined as **serving** processes. Each serving process and each device is represented as a server in exactly one submodel. All the callers of the server are also represented in the same submodel, along with the callers' other servers.

Any of the device scheduling and service time distributions considered in the MVA literature are permitted when describing a system. The MVA techniques can be used without modification. They include:

- Delay and Queueing Centers [Bask 75].
- First-come-first-serve scheduling with general service time distributions (HVFIFO) [Reiser 79].
- Priority-Preemptive-Resume servers at which groups are assigned fixed priorities

and queue for service in order of priority. Processes with the same priority are served in a first-come-first-served manner (PPR) [Bryant 84].

In subsequent chapters, several new techniques for representing process interactions are introduced. They can be used in conjunction with the MOL to model the behaviour of software systems. Each of the techniques is implemented as a type of server that addresses a specific interaction present in software systems. The MVA algorithm is modified and extended to include appropriate residence time expressions for each new type of server. They are the Rendezvous, Multiple-Entry, Multi-Server, SYNC, and SYNCDEL servers. The MOL can be used to predict the performance of software systems that can be described in terms of the servers that are available.

To summarize, the structure of the MOL algorithm is as follows:

```
The response time estimates for processes are initialized assuming no
device or process contention.
WHILE successive process response time estimates have not reached a fixed point DO
  WHILE successive process response time estimates
        have not reached a fixed point DO
        FOR each software submodel DO
            Solve the submodel using Linearizer with the following residence
            time expressions:
               FIFO, Rendezvous, Multiple-Entry, Multi-Server,
               SYNC, SYNCDEL, and DELAY.
        END FOR
  ENDWHILE
  Solve the device contention submodel using Linearizer with the
  following residence time expressions:
        PS, FIFO, HVFIFO, LCFS, DELAY, and PPR.
  Update the process response time estimates.
END WHILE
```

In section 2.1 the MOL algorithm is described. A definition of the error measure to be used in the thesis is given in section 2.2. In section 2.3, several models from the literature are considered. The accuracy of the MOL's performance estimates for the models are compared with the accuracy of SRVN's performance estimates for the same models. In section 2.4, the rate of convergence for the MOL on the models is considered for a range of convergence tolerances. Finally, conclusions are offered in section 2.5.

2.1 The Method Of Layers

The Method of Layers is used to predict the performance measures of models of systems containing software processes. The models are called **Layered Group Models** (LGM). Processes that can be said to have statistically identical behaviour form a group or class. Groups use devices but they can also request services from and provide services to other groups. The notion of a group has been introduced to enforce the notion that processes can both provide and receive service.

A sample LGM is shown in figure 2.1. Each parallelogram is a group of one or more processes and each circle is a device. Directed arcs indicate requests for service from the calling group to the serving group or device.

The MOL requires groups of processes to be associated with levels in a hierarchy. Groups that do not request service from other groups are at low numbered levels, and the groups that visit them are at higher levels. In general, it is assumed that groups at level l only visit groups at level $l-1$, each group is at exactly one level, and that there are no groups at level 0. The groups form a hierarchy with $|L|$ levels, $1 \cdots L$. A topological sort [Aho 83] can be used to choose levels for the groups.

Naturally, general software systems can have groups that can request service from any other group regardless of the notion of levels. Systems with acyclic dependencies that have requests for service that span more than one level can also be mapped onto LGMs but require the introduction of flow equivalent service centers (see page 38) into the model [Rolia 88].

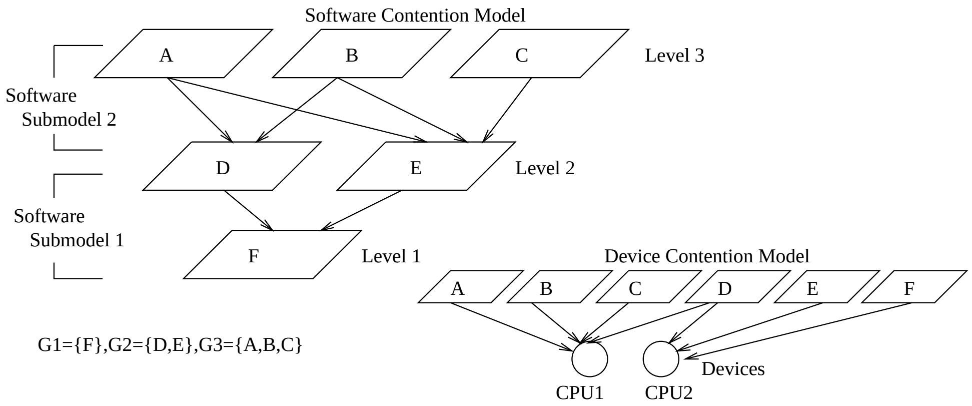


Figure 2.1: The decomposition of an LGM into submodels.

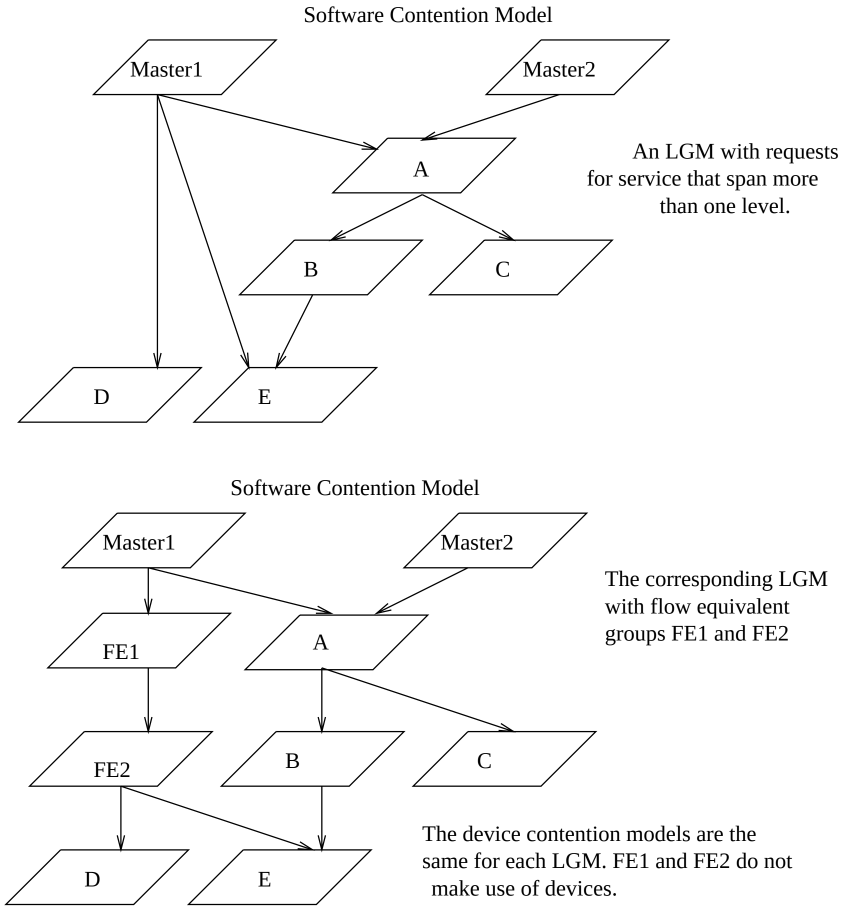


Figure 2.2: The introduction of flow equivalent groups into an acyclic LGM.

Consider the case where a group g visits a group h that is two levels below. The LGM can be augmented with a flow equivalent group that is constructed to act as a performance neutral place holder. It represents the lower level serving groups' behaviour when considered as a server, and the higher level requesting group's behaviour when considered as a requester. It is required to ensure that the queueing delays at each serving process are computed in exactly one of the submodels with all of its customers present or represented by flow equivalent groups.

A single flow equivalent group can be used to represent many serving groups. Processes within the requesting group do not suffer queueing delays when competing for access to the flow equivalent group, it represents the combined behaviour of the processes in the requesting group. If a level n process requests service from a level one group, one dummy group is required for each of the $n-2$ intermediate levels. An example is shown in figure 2.2. Examples of this technique are given in sections 3.1.2 and 3.2.2.

The input parameters for an LGM are a superset of those required for closed separable queueing network models. The input parameters for an LGM are:

$G,\ K$		Set of groups and devices
L		The number of software levels in the LGM hierarchy
G_n	$\forall\ n \in 1 \cdots L$	The set of groups at level n of the hierarchy
N_g	$\forall\ g \in G$	Population of group g
$V_{g,h}$	$\forall\ g, h \in G$	The average number of visits from group g to group h Group g is assumed to be one level higher than group h
$V_{g,k}$	$\forall\ g \in G\ \ \forall\ k \in K$	The average number of visits from group g to device k
$S_{g,k}$	$\forall\ g \in G\ \ \forall\ k \in K$	The average service time of a visit by group g at device k
Z_g	$\forall\ g \in G$	The think time for group g
ψ_j	$\forall\ j \in G \cup K$	The queueing discipline of group or device j

The think time of a group is the average time between the completion time of a process in the group and its next starting time, including periods of duration zero caused by contiguous computations. For serving groups, the input parameter Z_g is assumed to have value zero.

In addition to the metrics provided by MVA for device performance, some of the values that are computed by the MOL are:

R_g	$\forall\, g \,\epsilon\, G$	The average response time of a group g process
$R_{g,h}$	$\forall\, g, h \,\epsilon\, G$	The average residence time of a group g process when visiting group h
U_g	$\forall\, g \,\epsilon\, G$	The utilization of a group g process
R_g^{GRP}	$\forall\, g \,\epsilon\, G$	The average time a group g process spends acquiring service from other groups
R_g^{DEV}	$\forall\, g \,\epsilon\, G$	The average time a group g process spends acquiring service from devices
R_g^{IDL}	$\forall\, g \,\epsilon\, G$	The average time a group g process spends idle between invocations

Note that $R_g = R_g^{GRP} + R_g^{DEV}$ for all groups, $R_g^{IDL} = Z_g$ for non-serving groups, and R_g^{IDL} is computed for serving groups. The throughput of one of the group g processes is denoted as X_g^1 and is its average number of completions per unit time and is simply

$$X_g^1 = \frac{1}{(R_g^{GRP} + R_g^{DEV} + R_g^{IDL})} = \frac{1}{(R_g + R_g^{IDL})}.$$

The superscript of 1 indicates that the value corresponds to one customer of the group. The same notation is used for utilization as well. The average idle period for a process of group g is its average intercompletion time less its average response time and is simply (by rewriting the expression for X_g^1)

$$R_g^{IDL} = 1/X_g^1 - R_g.$$

Using the utilization law [Lazow 84], the utilization of each process in group g is $U_g^1 = X_g^1 R_g$. Thus the average idle time of a group g process can be expressed as follows

$$R_g^{IDL} = R_g/U_g^1 - R_g.$$

Given the parameters of an LGM, the R_g^{IDL} and U_g are not known for serving groups.

The purpose of the MOL is to find a fixed point where the predicted values for R_g^{IDL} and U_g are consistent with respect to all of the submodels. At that point, the results of the MVA calculations approximate the performance measures for the system under consideration. Intuitively, this is the point at which predicted group idle times and utilizations are balanced so that the group has the same throughput whether it is considered as a customer class or as a server class (the rate of completions of the serving group equals the rate of requests for its service), and the average service time required by callers of the group equals its average response time.

The MOL divides an LGM into two types of models. The software contention model describes the software relationships in the process architecture and contains information that is used to predict software contention delays. The **device contention** model describes each group's device usage. It is used when predicting device contention delays. The results of the two models are combined to provide performance estimates for the system. As in the method of complementary delays [Jacob 82], the two models are solved alternately, with the solution of one helping to determine some of the input parameters of the other.

Initially, with the assumption that there is no device contention, performance estimates are found for the software model. The method then alternates between device contention and software contention models until response time estimates for non-serving groups differ by less than some tolerance. In the following paragraphs, the device contention model is described, the software contention model is discussed, and then the relationship between the two models is explained. The parameters of the queueing network model are described in terms of the LGM parameters.

There is only one instance of the device contention model. It is a closed queueing network model in which each group is represented as a customer class, and each device as a server. In this queueing network model, the think time Z_g of group g is time that the group's processes do not spend competing with other groups for devices,

$$Z_g = R_g^{GRP} + R_g^{IDL}.$$

The values R_g^{GRP} and R_g^{IDL} are obtained from the software contention model. The remaining parameters required to complete the device contention queueing network model's description are: N_g, $V_{g,k}$, $S_{g,k}$, and ψ_k, and are obtained directly from the LGM's parameters. The performance measures for the model are predicted using MVA.

The software contention model is mapped onto a sequence of $L-1$ two level submodels that are used to estimate the software contention between successive levels of groups. A submodel is a closed queueing network that is solved using MVA. Groups that request service at the higher level are considered as customer classes, and groups that provide service at the lower level are represented as servers. There are only $L-1$ submodels because groups at level 1 do not visit other groups, so they are never considered as customer classes in a software contention submodel. The submodels are ordered from $L-1$ down to 1, with groups at level l representing customer classes in the $l-1$st submodel. The software submodels are solved in submodel order $L-1$ down to 1. See figure 2.1 on page 59 for an illustration of the difference between submodel number and software level number.

Initially, it is assumed that there is no device or software contention in the models. The response time of a group is simply the sum of the resource demands required by the group. The initial response time estimates for the groups of the LGM are computed software level by software level in order 1 to L. Note that G_0 is the empty set because there are no groups at level zero.

$$\begin{array}{lll} R_g^{DEV} = & \sum_{k \,\epsilon\, K} V_{g,k}\, S_{g,k} & \forall\, g \,\epsilon\, G \\ R_g^{GRP} = & \sum_{h \,\epsilon\, G_{l-1}} V_{g,h}\, R_h & \forall\, g \,\epsilon\, G_l \text{ solved in order } l = 1 \text{ up to } l = L \\ R_g = & R_g^{DEV} + R_g^{GRP} & \forall\, g \,\epsilon\, G_l \end{array}$$

Consider the software submodel $l-1$ in which groups g in G_l are considered as customer classes, and groups h in G_{l-1} are servers. In the submodel's corresponding queueing network model, the think time Z_g is the time that the group's processes do not spend competing for other groups,

$$Z_g = R_g^{DEV} + R_g^{IDL}.$$

The R_g^{DEV} are obtained from the device contention model. The values R_g^{IDL} are input parameters of non-serving groups of the LGM, and are computed using the relation $R_g^{IDL} = R_g/U_g^1 - R_g$ for serving groups. The parameters N_g, $V_{g,h}$ and ψ_h are given in the LGM's input parameters. To complete the queueing network model's description, the service time of group g per visit to group h is required.

- $S_{g,h}$ is the average service time requested by a customer of group g at server h.

$S_{g,h}$ is not an input parameter to the model. It is estimated using intermediate results from the MOL.

$$S_{g,h} = R_h^{GRP} + R_h^{DEV} = R_h.$$

Linearizer is used to predict the software contention between the two levels represented by the submodel. The results provide new estimates for R_g^{GRP} for all g in G_l and U_h for all h in G_{l-1}. The utilization U_h is used to predict R_h^{IDL} and hence the think time for the group h when it is considered as a customer class in the next submodel.

The set of submodels are solved repeatedly, in submodel order $L-1$ down to 1, until the differences in successive non-serving group response time estimates differ by less than some tolerance. At this stage, the software model is assumed to have reached a fixed point. At least $L-1$ iterations are used to ensure that the effects of submodel 1 can affect submodel $L-1$.

Now consider the relationship amongst the software and device contention models. The software contention model provides estimates for R_g^{GRP} and R_g^{IDL}. These values are used to specify group think times in the device contention model, which is then solved to provide new estimates for R_g^{DEV} for all g in G. Once this is done response time estimates for the groups in the software contention model can be computed as follows. The computation is done by software level, in order 1 to L.

$$\begin{array}{lll} R_g^{DEV} = & \sum_{k=1}^{K} V_{g,k}\, R_{g,k} & \forall\, g \,\epsilon\, G \\ R_g^{GRP} = & \sum_{h \,\epsilon\, G_{l-1}} V_{g,h}\, R_h & \forall\, g \,\epsilon\, G_l \text{ solved in order } l = 1 \text{ to } l = L \\ R_g = & R_g^{DEV} + R_g^{GRP} & \forall\, g \,\epsilon\, G_l \end{array}$$

The MOL alternates between software and device contention submodels until the differences in non-serving group response time estimates, after solving the device contention model, are less than some tolerance. At this stage, the performance metrics for the LGM have reached a fixed point. This completes the description of the MOL.

Algorithm Method of Layers

Parameters

- Model Inputs: an LGM.
- Model Outputs: Performance measures: R_g, U_g, $R_{g,h}$, $U_{g,h}$, $R_{g,k}$, $U_{g,k}$.
- Model Intermediate Results: $Step4R_g$ and $Step5R_g$ are copies of response time estimates for the previous iteration. The step 4 loop is used to find a fixed point for the software submodels. Step 5 is used to find a fixed point for the combined software and device contention models.
- Notation: $\downarrow$ and $\uparrow$ indicate an input and output parameters, respectively.
  $\Delta(R, StepR)$ compares two arrays of response time estimates, returns the maximum relative error with respect to R, and copies R to $StepR$. τ is the required tolerance for convergence of device and software contention loops. The models in this thesis have been solved with $\tau = 0.005$.

Begin

Step 1.
Initialize intermediate result vectors
$Step4R_g,\ \ Step5R_g \leftarrow 0\ \forall$ non-serving groups $g \epsilon G$
$L_0\ \leftarrow \emptyset$

Step 2.
Initialize assuming no device contention

$$\begin{array}{lll}
R_g^{DEV} \leftarrow & \sum_{k\ \epsilon\ K} V_{g,k}\ S_{g,k} & \forall\ g\ \epsilon\ G \\
R_g^{GRP} \leftarrow & \sum_{h\ \epsilon\ G_{l-1}} V_{g,h}\ (R_h^{DEV} + R_h^{GRP}) & \forall\ g\ \epsilon\ G_l \quad \text{solved in order } l = 1 \text{ to } l = L \\
R_g \leftarrow & R_g^{DEV} + R_g^{GRP} & \forall\ g\ \epsilon\ G \\
R_g^{IDL} \leftarrow & Z_g & \forall \text{ non-serving groups } g\ \epsilon G
\end{array}$$

Loop Device Contention
$iter \leftarrow\ 0$

Loop Software Contention

Step 3.
$iter = iter + 1$
For $l \leftarrow L$ downto 2 do
Apply MVA ($\downarrow$Customer classes $g\ \epsilon\ G_l$,
$\downarrow$Customer populations N_g for $g\ \epsilon\ G_l$,
$\downarrow$Customer think times $Z_g = R_g^{DEV} + R_g^{IDL}$ for $g\ \epsilon\ G_l$,
$\downarrow$Servers $h\ \epsilon\ G_{l-1}$,
$\downarrow$Visits $V_{g,h}$ from customers g at
level G_l to Servers h at level G_{l-1},
$\downarrow$Service time $R_h = R_h^{DEV} + R_h^{GRP}$ of groups $h\ \epsilon\ G_{l-1}$,
$\downarrow$Scheduling Discipline ψ_h of groups $h\ \epsilon\ G_{l-1}$,
$\uparrow$Residence time $R_{g,h}$ of groups $g\ \epsilon\ G_l$ at servers $h\ \epsilon\ G_{l-1}$,
$\uparrow$Utilization U_h of groups $h\ \epsilon\ G_{l-1}$)
$R_g^{GRP} \leftarrow \sum_{h\ \epsilon\ G_{l-1}} V_{g,h}\ R_{g,h} \forall\ g\ \epsilon\ G_l$
$R_g \leftarrow R_g^{DEV} + R_g^{GRP}\ \ \forall\ g\ \epsilon\ G_l$

$R_h^{IDL} \leftarrow R_h/U_h^1 - R_h \ \ \forall \ h \ \epsilon \ G_{l-1}$

End For.

Step 4. Exit when $\Delta(R_g, Step4R_g) < \tau$ $\forall$ non-serving groups $g \ \epsilon \ G$ and $iter > (L-1)$.

End Loop Software Contention

Step 5. Exit when $\Delta(R_g, Step5R_g) < \tau$ $\forall$ non-serving groups $g \ \epsilon \ G$

Step 6. Apply MVA ($\downarrow$Customer classes $g \ \epsilon \ G$, $\downarrow$Customer populations N_g for $g \ \epsilon \ G$,
$\downarrow$Customer think times $Z_g = R_g^{GRP} + R_g^{IDL}$ for $g \ \epsilon \ G$,
$\downarrow$Servers $k \ \epsilon \ K$,
$\downarrow$Visits $V_{g,k}$ from customers $g \ \epsilon \ G$ to servers $k \ \epsilon \ K$,
$\downarrow$Service time $S_{g,k}$ of groups $g \ \epsilon \ G$ while at $k \ \epsilon \ K$,
$\downarrow$Scheduling Discipline ψ_h of servers $k \ \epsilon \ K$,
$\uparrow$Residence time $R_{g,k}$ of groups $g \ \epsilon \ G$ at servers $k \ \epsilon \ K$,
$\uparrow$Utilization U_k of device $k \ \epsilon \ K$)

$R_g^{DEV} \leftarrow \sum_{k \ \epsilon \ K} V_{g,k} \ R_{g,k} \forall \ g \ \epsilon \ G$

$R_g^{GRP} \leftarrow \sum_{h \ \epsilon \ G_{l-1}} V_{g,h} \ (R_h^{DEV} + R_h^{GRP}) \forall \ g \ \epsilon \ G_l$ solved in order $l = 1$ to $l = L$

$R_g \leftarrow R_g^{DEV} + R_g^{GRP} \ \ \forall \ g \ \epsilon \ G$

End Loop Device Contention

End Algorithm Layers.

The Method of Layers has been used to study systems of software servers that communicate using the Remote Procedure Call while sharing devices [Rolia 87, Rolia 88]. Consider the time complexity of each iteration of the method. Let $|G|$ be the number of groups, and $|K|$ be the number of devices. When Linearizer is used as the MVA software, the time complexity of steps 1 through 6 are as follows: $O(|G||K|)$, $O(|G|^2|K|)$, $O(L|G|^3)$, $O(|G|)$, $O(|G|)$ and $O(|G|^2|K|)$.

The value of L is a function of the system that is described; it does not depend upon the number of groups $|G|$. Thus, the total complexity is considered to be $O(max(|L||G|^3, |G|^2|K|))$.

A sample solution for an LGM is given in appendix A.

2.2 Error

Unless otherwise stated, the relative error in estimated average process response time R_g with respect to the simulated average process response time is used as the standard measure of error for the models in this thesis:

$$\text{error } \% = \frac{\text{simulated response time - estimated response time}}{\text{simulated response time}} \times 100.$$

The error is reported for non-serving tasks since they are assumed to be the most important tasks in the models. All simulated average response times have a 95% confidence interval within $\pm 5\%$ of the reported result.

When models are validated with respect to simulation or exact results, errors in estimated response time of less than $\pm 10\%$ are considered to be good for non-serving tasks. In these cases the parameters for the models are the same for the layered group model and the simulation or exact model. The error shows how well the technique predicts exact results.

For the models that are validated with respect to measured system values, errors of less than $\pm 20\%$ are considered to be good. This is the case for mean value analysis software in general. The error in estimating parameters for the model and not considering some system features is likely to cause a significant portion of the error.

Relative error is used instead of absolute error because it makes it easier to compare the accuracy of the different techniques when applied to their respective sets of models. The response times for processes in the models can vary dramatically. The relative error in estimated process response time is used instead of the relative error in estimated residence times. This is because the ratio of time away from the server to time at the server is an important factor in how significant an error is. At low server utilizations a large part of the time is spent away from the server, at high utilizations most of the time is spent at the server.

When the MOL is compared with the results of Woodside and Miernik et al., the error measures they define are used. Their error estimates are based on reference (non-serving) task throughput.

2.3 The Accuracy Of The Technique

The accuracy of the MOL has been considered [Rolia 87, Rolia 88]. The MOL tends to overestimate average process response times with respect to simulation when the service demands in submodels are balanced. Errors in higher numbered software submodels have a greater effect upon total error than similar errors in lowered numbered software submodels. This is because the higher numbered submodels are solved first and errors propagate down. The relative resource utilization among levels of the model does not have a significant effect on error. Models with up to seven layers have been studied.

In this section, the MOL is compared with the Stochastic Rendezvous Network Model technique of Miernik et al. for the set of models considered in their paper [Mier 89]. The models describe hierarchical software systems that share devices. On average, the MOL and SRVN have the same error characteristics.

The models describe software systems with processor-sharing, and priority-preemptive-resume (PPR) CPUs. Processes are assigned fixed priorities and queue at the priority CPUs in order of priority. If several processes have the same priority, they are served in a first-come-first-served manner. If a higher priority process arrives, the process receiving service is preempted. The MOL has also been applied to software systems that contain priority scheduled devices [Rolia 88]. Both of the SRVN and MOL approaches use the priority server technique introduced by Bryant et al. [Bryant 84]. It provides a residence time expression that can be used to model a PPR server.

The processes in the software systems are referred to as **tasks**. In the LGM, the tasks are represented using groups with one task per group. Some notation for describing a task's priority is required.

$\pi(T_i)$ is defined as the priority of task T_i, where lower numbers correspond to lower priorities.

The following three models with a total of four sets of test cases are studied in the following subsections.

- Example 1: a real-time data base application with three tasks running on two processors.

- Example 2: a user-server system executing on two and three processors (examples 2a and 2b respectively) with a strong concentration of traffic to one server task and with processor utilization from very light to over 0.9.

- Example 3: Distributed software executing on four processors with more diversified task interactions and two execution phases.

The models have been simulated by Miernik et al. and the accuracy of the analytical technique is evaluated by comparing the simulated performance measures with the results of analysis. Miernik et al. use the following combined error measure to describe the percentage of error associated with each test case:

$$E_{tol} = \frac{\sum_{c=1}^{C} |X_c(\text{Simulated}) - X_c(\text{Approx.})|}{\sum_{c=1}^{C} X_c(\text{Simulated})} \times 100,$$

where C is the number of non-serving tasks. The equation relates errors to total throughput, which prevents errors in small throughputs from having too much influence on the perceived accuracy of the approximations. The percentage of error is defined as **tolerance** error. It must be noted that large errors in small throughputs are ignored if the throughputs of tasks differ significantly.

The following other error measures are also used. They are all based on relative errors in estimated process throughputs with respect to simulated process throughputs. Let T be the number of test cases and C_t be the number of non-serving tasks in test case t.

- $\bar{E_{tol}}$ The average tolerance error for T test cases.

$$\bar{E_{tol}} = \frac{\sum_{t=1}^{T} E_{tol_t}}{T}$$

- E_{arel} The mean of the absolute percentage relative error for all non-serving classes in each test case. It is used to show the average magnitude of the percentage relative errors. Smaller is better.

$$E_{arel} = \frac{\sum_{t=1}^{T}\sum_{c=1}^{C_t}[|X_c(\text{Simulated}) - X_c(\text{Approx.})|/X_c(\text{Simulated})]}{\sum_{t=1}^{T} C_t} \times 100$$

- E_{rel} The average of percentage relative errors for all non-serving classes in each test case. It is used to show what weighted signed error value the signed percentage relative errors are centered around. A value close to zero is best.

$$E_{rel} = \frac{\sum_{t=1}^{T}\sum_{c=1}^{C_t}[(X_c(\text{Simulated}) - X_c(\text{Approx.}))/X_c(\text{Simulated})]}{\sum_{t=1}^{T} C_t} \times 100$$

- E_{pdev} The population deviation of errors with respect to the average of percentage relative errors. It is a measure of how close the errors are to the average. Lower is better.

$$E_{pdev} = \sqrt{\frac{\sum_{t=1}^{T}\sum_{c=1}^{C_t}[\frac{X_c(\text{Simulated}) - X_c(\text{Approx.})}{X_c(\text{Simulated})} \times 100 - E_{rel}]^2}{\sum_{t=1}^{T} C_t}}$$

Several of the examples that are presented in this section describe serving task resource requirements in terms of phases. Phase 1 and phase 2 refer to rendezvous and post-rendezvous processing, respectively. The MOL server that is used to represent servers with post-rendezvous processing is presented in chapter 3. All of the models considered in the paper by Miernik et al. are presented here for completeness.

2.3.1 Example 1

In the first example, parameters are chosen so that there are significant changes in service time requirements across test cases at the shared processor. The relative priorities of the tasks sharing the processor are altered. The software process architecture corresponding to example 1 is shown in figure 2.3. The model parameters for the test cases are shown in table 2.1.

Table 2.1 provides the model parameters for example 1. In table 2.1a), the service demands are given for each task. The average number of visits and average service times are given as numeric values when they do not change for models in the set of test cases. Otherwise they are described as symbols whose values are specified in the test case descriptions found in table 2.1d). For example, each phase of task $T3$ visits the CPU2 once per invocation and has an average service request of one time unit. The average number of visits from task T1 to task T3 differs in the test cases. It is described as the symbol $V_{T1,T3}$. In table 2.1b), the processor scheduling disciplines are given. The serving tasks are all assumed to be first-come-first-served rendezvous servers with the same average service times for each caller. Table 2.1c) states which tasks reside on each processor and the task scheduling disciplines. Finally, in table 2.1d), the parameters for the individual test cases are specified using numeric values. Also, the task priorities are specified. The same method is used to describe each of the models.

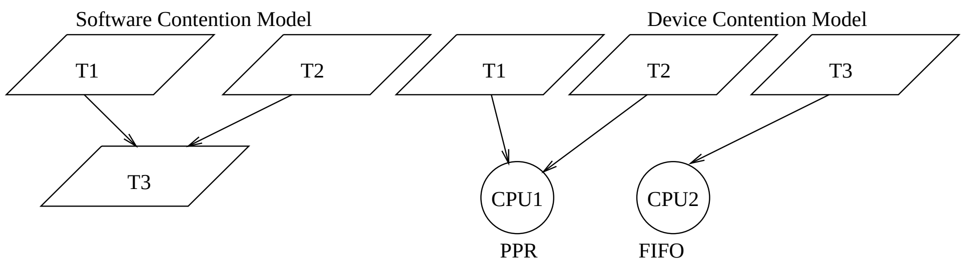


Figure 2.3: Miernik Priority Example 1

Entity	V_{T3}	V_{CPU}	S_{CPU}
T1 Phase 1	$V_{T1,T3}$	$V_{T1,T3}+1$	$S_{T1,CPU}$
T2 Phase 1	$V_{T2,T3}$	$V_{T2,T3}+1$	$S_{T2,CPU}$
T3 Phase 1	-	1	1
T3 Phase 2	-	1	1

Table 2.1: a) Service Demands Per Invocation for Example 1.

Processor Name	Scheduling Discipline
Processor 1	Priority-Preemptive-Resume (PPR)
Processor 2	First-In-First-Out (FIFO)

Table 2.1: b) Processor Descriptions for Example 1.

Entity	Processor Number	Task Scheduling Discipline
T1	1	Non-Serving
T2	1	Non-Serving
T3	2	Multiple-Entry Server

Table 2.1: c) Processor Allocation and Task Scheduling Disciplines for Example 1.

Case	$S_{T1,CPU}$	$S_{T2,CPU}$	$V_{T1,T3}$	$V_{T2,T3}$	$\pi(T1)$	$\pi(T2)$
1	0.667	1.333	2	2	1	1
2	0.667	1.333	2	2	1	2
3	0.667	1.333	2	2	2	1
4	2	1.333	1	2	1	1
5	2	1.333	1	2	1	2
6	2	1.333	1	2	2	1
7	4	1.333	1	2	1	1
8	4	1.333	1	2	1	2
9	4	1.333	1	2	2	1
10	8	1.333	1	2	1	1
11	8	1.333	1	2	1	2
12	8	1.333	1	2	2	1
13	20	1.333	1	2	1	1
14	20	1.333	1	2	1	2
15	20	1.333	1	2	2	1

Table 2.1: d) Miernik et al. Test Cases For Priority Example 1.

Case	Task	Simulation	Miernik	E_{tol}	MOL	E_{tol}
1	Task 1	0.1063	0.1147		0.1346	
	Task 2	0.1020	0.1013	4.37	0.1013	13.92
2	Task 1	0.0777	0.0741		0.0818	
	Task 2	0.1208	0.1199	2.27	0.1259	4.61
3	Task 1	0.1521	0.1489		0.1534	
	Task 2	0.0740	0.0797	3.94	0.0876	6.56
4	Task 1	0.1342	0.1296		0.1257	
	Task 2	0.0835	0.0828	2.43	0.0988	10.91
5	Task 1	0.0736	0.0655		0.0714	
	Task 2	0.1408	0.1351	6.44	0.1373	2.67
6	Task 1	0.1835	0.1851		0.1850	
	Task 2	0.0364	0.0201	8.14	0.0281	4.48
7	Task 1	0.0874	0.0853		0.085703	
	Task 2	0.0576	0.0580	1.72	0.0578	1.31
8	Task 1	0.0426	0.0397		0.0423	
	Task 2	0.1418	0.1407	2.17	0.1410	0.59
9	Task 1	0.1076	0.1092		0.1090	
	Task 2	0.0199	0.0051	12.86	0.0086	9.95
10	Task 1	0.0517	0.0507		0.0508	
	Task 2	0.0358	0.0348	2.29	0.0346	2.43
11	Task 1	0.0230	0.0222		0.0233	
	Task 2	0.1453	0.1448	0.77	0.1434	1.28
12	Task 1	0.0582	0.0586		0.0586	
	Task 2	0.0103	0.0013	13.72	0.0024	12.12
13	Task 1	0.0230	0.0229		0.0229	
	Task 2	0.0166	0.0155	3.03	0.0155	3.14
14	Task 1	0.0098	0.0095		0.0099	
	Task 2	0.1474	0.1477	0.38	0.1452	1.50
15	Task 1	0.0243	0.0244		0.0244	
	Task 2	0.0042	0.0002	14.39	0.0004	13.59
$\bar{E}_{tol}$				5.26		5.94
E_{arel}				12.60		11.92
E_{rel}				11.27		6.62
E_{pdev}				26.82		24.98

Table 2.2: Miernik Example 1. The non-serving task throughputs X are summarized for the test cases.

The throughput estimates for non-serving tasks T1 and T2 are shown for each of the test cases in table 2.2. The MOL produces a slightly higher average tolerance and a slightly lower worst case tolerance error $\bar{E_{tol}}$ than Miernik et al. for the set of test cases. None of the models have unusually large E_{tol} errors. However when task throughputs are significantly different (see cases six, nine, twelve, and fifteen), the task 2 throughputs have large errors. Even so, the MOL is a little more accurate than the method of Miernik et al. for these cases. This is why the E_{arel} and E_{pdev} values are lower for the MOL. It is conjectured that the lower error for the MOL is due to the fact that the MOL uses Linearizer as its MVA software and SRVN is more similar to the less accurate Bard-Schweitzer MVA algorithm. Finally, the MOL has an E_{rel} closer to zero than SRVN which indicates that it is less pessimistic for these cases.

2.3.2 Example 2a

In example 2a, server tasks are on the same processors as the user tasks. The software process architecture for example 2a is shown in figure 2.4 and test case parameters are described in table 2.3. The throughput estimates for tasks T1, T2, and T3 are shown in table 2.4.

The models have more complex behaviour than in the first example because customer and server tasks reside on the same processor. For example, when $T4$ provides rendezvous service to $T1$, the two tasks are synchronized. $T4$ need not compete with $T1$ for service at the shared CPU. When a serving task provides rendezvous service to a requesting task that shares the same processor, the serving task does not have to compete for service with the requesting task at the CPU. The MOL does not consider this in the analysis. The SRVN approach does, but there does not appear to be any advantage in doing so. In both example 2a and 2b, the techniques have identical error characteristics.

In test cases 1, 4, and 7 priority service is not used. The errors are associated with the MOL are very low and the errors associated with the SRVN are much lower than they are in other cases. This implies that for these models, the interactions amongst requesting and serving tasks that share a CPU does not have a large impact upon the accuracy of the analysis. For the other test cases, there are up to five distinct priorities. Both Miernik et al. and the MOL use an approximation based upon the work of Bryant et al. [Bryant 84] which has been shown to be inaccurate for low priority customers when high priority customers have relatively large service demands or high utilization at the priority server [Eager 88]. It is most likely that the high error is attributed to the priority server approximation. If an improved PPR server technique were discovered it could be used to improve the results.

The E_{arel}, E_{rel}, and E_{pdev} values indicate that there is little difference in the accuracy of SRVN and the MOL for these test cases. Both E_{rel} values are near zero which shows that neither method is too optimistic or pessimistic. But, the magnitude of the relative errors E_{arel} are high in the 17% range, and the E_{pdev} values indicate that the errors are widely distributed.

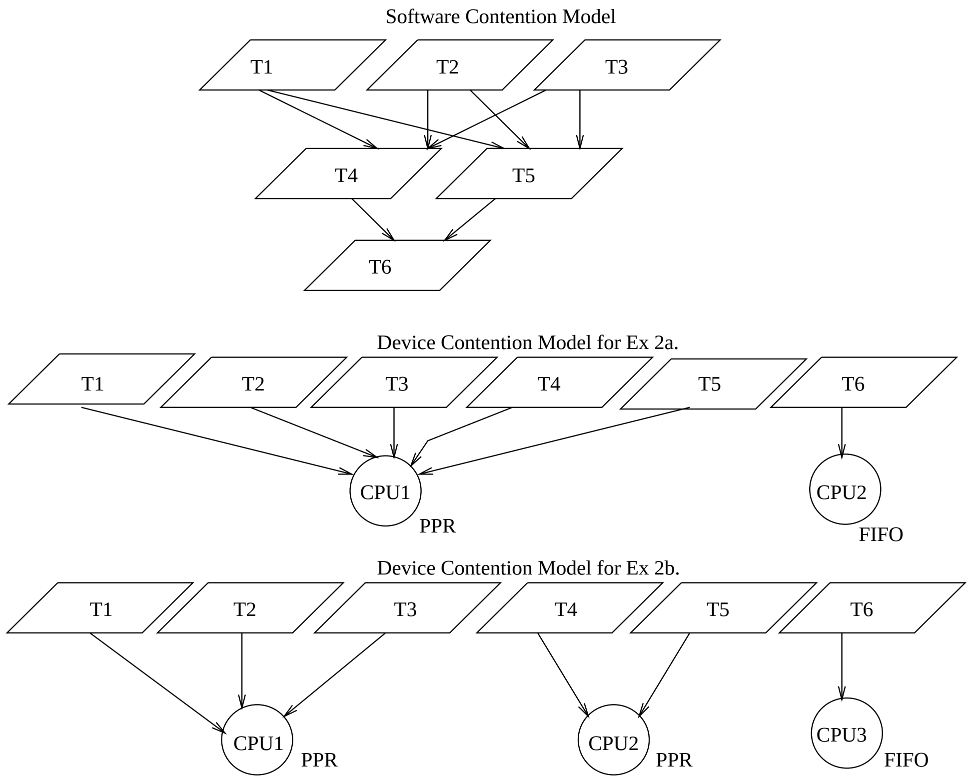


Figure 2.4: Miernik Priority Example 2a and Example 2b.

Entity	V_{T4}	V_{T5}	V_{T6}	V_{CPU}	S_{CPU}
T1	1	1	-	3	1
T2	1	1	-	3	1
T3	1	1	-	3	1
T4	-	-	1	2	$S_{T4,CPU}$
T5	-	-	1	2	$S_{T5,CPU}$
T6	-	-	-	1	$S_{T6,CPU}$

Table 2.3: a) Service Demands Per Invocation for Example 2a.

Processor Name	Scheduling Discipline
1	Priority-Preemptive-Resume (PPR)
2	First-In-First-Out (FIFO)

Table 2.3: b) Processor Descriptions for Example 2a.

Entity	Processor Number	Task Scheduling Discipline
T1	1	Non-Serving
T2	1	Non-Serving
T3	1	Non-Serving
T4	1	Multiple-Entry Server
T5	1	Multiple-Entry Server
T6	2	Multiple-Entry Server

Table 2.3: c) Processor Allocation and Task Scheduling Disciplines for Example 2a.

Case	$S_{T4,CPU}$	$S_{T5,CPU}$	$S_{T6,CPU}$	$\pi(T1)$	$\pi(T2)$	$\pi(T3)$	$\pi(T4)$	$\pi(T5)$
1	0.5	0.5	1	1	1	1	1	1
2	0.5	0.5	1	1	2	3	4	5
3	0.5	0.5	1	1	2	3	3	3
4	0.25	0.25	1	1	1	1	1	1
5	0.25	0.25	1	1	2	3	4	5
6	0.25	0.25	1	1	2	3	3	3
7	0.25	0.25	0.5	1	1	1	1	1
8	0.25	0.25	0.5	1	2	3	4	5
9	0.25	0.25	0.5	1	2	3	3	3

Table 2.3: d) Miernik et al. Test Cases For Priority Example 2a.

Case	Task	Simulation	Miernik	E_{tol}	MOL	E_{tol}
1	Task 1	0.0806	0.0751		0.0769	
	Task 2	0.0807	0.0751		0.0769	
	Task 3	0.0805	0.0751	6.82	0.0769	4.64
2	Task 1	0.0508	0.0636		0.0651	
	Task 2	0.0717	0.0753		0.0753	
	Task 3	0.1351	0.0838	26.28	0.0845	26.41
3	Task 1	0.0504	0.0623		0.0619	
	Task 2	0.0700	0.0737		0.0705	
	Task 3	0.1322	0.0897	23.00	0.0925	20.43
4	Task 1	0.1014	0.0936		0.0981	
	Task 2	0.1019	0.0936		0.0981	
	Task 3	0.1019	0.0936	7.99	0.0981	3.60
5	Task 1	0.0680	0.0825		0.0898	
	Task 2	0.0930	0.0983		0.1014	
	Task 3	0.1683	0.1088	24.08	0.1115	26.42
6	Task 1	0.0681	0.0811		0.0867	
	Task 2	0.0910	0.0958		0.0966	
	Task 3	0.1634	0.1109	21.80	0.1133	23.07
7	Task 1	0.1340	0.1256		0.1308	
	Task 2	0.1330	0.1256		0.1308	
	Task 3	0.1340	0.1256	6.03	0.1308	2.13
8	Task 1	0.0643	0.0831		0.0877	
	Task 2	0.1097	0.1374		0.1288	
	Task 3	0.2619	0.1799	29.48	0.1748	29.73
9	Task 1	0.0649	0.0826		0.0862	
	Task 2	0.1083	0.1305		0.1172	
	Task 3	0.2547	0.1815	26.43	0.1769	25.24
$\bar{E}_{tol}$				19.10		17.96
E_{arel}				17.52		16.76
E_{rel}				1.76		0.01
E_{pdev}				21.22		21.94

Table 2.4: Miernik Example 2a. The non-serving task throughputs X are summarized for the test cases.

2.3.3 Example 2b

In example 2b, server tasks are no longer on the same processors as the user tasks. Also, the requests for rendezvous amongst tasks are less balanced. The asymmetry increases the effect of software bottlenecks upon performance estimates and decreases the influence of the priority approximation. The software process architecture for example 2b is shown in figure 2.4 and test case parameters are described in table 2.5. The throughput estimates for tasks T1, T2, and T3 are shown in table 2.6.

The average error estimates are much lower than for example 2a. The most significant factor affecting this decrease in error is the low utilization of processor 1 which (except for test cases 7, 8, and 9) is the only priority server. From more detailed results, the processor has a utilization ranging from 2% to 20% for the various test cases. The priority-preemptive-resume server is expected to be accurate under such scenarios [Bryant 84, Eager 88].

Entity	V_{T4}	V_{T5}	V_{T6}	V_{CPU}	S_{CPU}
T1	1	2	-	4	0.25
T2	2	2	-	5	0.4
T3	1	1	-	3	0.167
T4	-	-	1	2	1
T5	-	-	1	2	0.6665
T6	-	-	-	1	$S_{T6,CPU}$

Table 2.5: a) Service Demands Per Invocation for Example 2b.

Processor Name	Scheduling Discipline
1	Priority-Preemptive-Resume (PPR)
2	Priority-Preemptive-Resume (PPR)
3	First-In-First-Out (FIFO)

Table 2.5: b) Processor Descriptions for Example 2b.

Entity	Processor Number	Task Scheduling Discipline
T1	1	Non-Serving
T2	1	Non-Serving
T3	1	Non-Serving
T4	2	Multiple-Entry Server
T5	2	Multiple-Entry Server
T6	3	Multiple-Entry Server

Table 2.5: c) Processor Allocation and Task Scheduling Disciplines for Example 2b.

Case	$S_{T6,CPU}$	$\pi(T1)$	$\pi(T2)$	$\pi(T3)$	$\pi(T4)$	$\pi(T5)$
1	10.	1	1	1	1	1
2	1.0	1	1	1	1	1
3	0.1	1	1	1	1	1
4	10.	2	1	1	1	1
5	1.0	2	1	1	1	1
6	0.1	2	1	1	1	1
7	10.	2	1	1	1	2
8	1.0	2	1	1	1	2
9	0.1	2	1	1	1	2
10	10.0	2	3	1	1	1
11	1.0	2	3	1	1	1
12	0.1	2	3	1	1	1

Table 2.5: d) Miernik et al. Test Cases For Priority Example 2b.

Case	Task	Simulation	Miernik	E_{tol}	MOL	E_{tol}
1	Task 1	0.0104	0.0082		0.0086	
	Task 2	0.0075	0.0062		0.0063	
	Task 3	0.0152	0.0125	18.70	0.0127	16.76
2	Task 1	0.0498	0.0476		0.0475	
	Task 2	0.0365	0.0344		0.0329	
	Task 3	0.0737	0.0713	4.19	0.0689	6.67
3	Task 1	0.0661	0.0601		0.0634	
	Task 2	0.0488	0.0434		0.0428	
	Task 3	0.0990	0.0905	9.30	0.0908	7.93
4	Task 1	0.0104	0.0082		0.0086	
	Task 2	0.0071	0.0062		0.0063	
	Task 3	0.0160	0.0125	19.73	0.0127	17.74
5	Task 1	0.0502	0.0478		0.0476	
	Task 2	0.0335	0.0338		0.0327	
	Task 3	0.0788	0.0721	5.78	0.0692	7.99
6	Task 1	0.0671	0.0604		0.0636	
	Task 2	0.0441	0.0423		0.0423	
	Task 3	0.1094	0.0921	11.70	0.0915	10.51
7	Task 1	0.0107	0.0082		0.0086	
	Task 2	0.0070	0.0062		0.0063	
	Task 3	0.0159	0.0126	19.67	0.0127	17.94
8	Task 1	0.0573	0.0504		0.0490	
	Task 2	0.0333	0.0319		0.0312	
	Task 3	0.0781	0.0678	11.03	0.0660	13.29
9	Task 1	0.0824	0.0680		0.0652	
	Task 2	0.0419	0.0384		0.0380	
	Task 3	0.1038	0.0831	16.92	0.0818	18.91
10	Task 1	0.0103	0.0082		0.0086	
	Task 2	0.0080	0.0062		0.0063	
	Task 3	0.0145	0.0124	18.26	0.0127	16.00
11	Task 1	0.0498	0.0474		0.0474	
	Task 2	0.0383	0.0348		0.0332	
	Task 3	0.0695	0.0702	4.19	0.0682	5.56
12	Task 1	0.0655	0.0598		0.0632	
	Task 2	0.0522	0.0441		0.0435	
	Task 3	0.0929	0.0887	8.55	0.0892	6.98
$\bar{E}_{tol}$				12.34		12.19
E_{arel}				11.95		11.76
E_{rel}				11.85		11.76
E_{pdev}				6.95		6.26

Table 2.6: Miernik Example 2b. The non-serving task throughputs X are summarized for the test cases.

2.3.4 Example 3

In example 3, a software system with seven tasks is executed on four processors. The software process architecture is shown in figure 2.5 and test case parameters are shown in table 2.7. Parameters have been chosen so that there is not a great deal of processor contention. Effectively, there are three server tasks executing on different processors. The priorities amongst tasks are altered to create the set of test cases for this example. The throughput estimates for tasks T1 and T2 are shown in table 2.8.

The $\bar{E_{tol}}$ and E_{arel} values for the MOL are slightly lower than for SRVN. The MOL has a lower E_{rel} which indicates that it is also less pessimistic. However, the MOL has a higher E_{pdev} value which indicates its errors are more widely distributed.

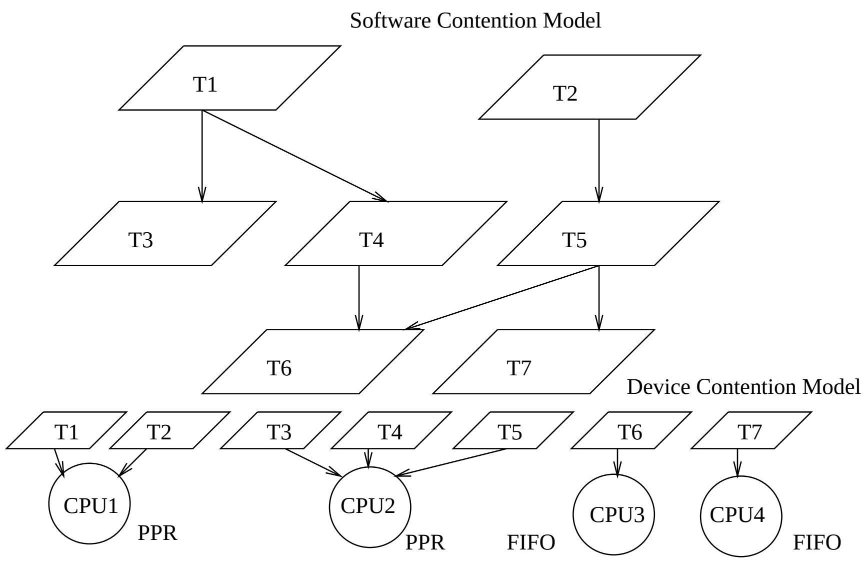


Figure 2.5: Miernik Priority Example 3.

Entity	V_{T3}	V_{T4}	V_{T5}	V_{T6}	V_{T7}	V_{CPU}	S_{CPU}
T1 Phase 1	1	1	-	-	-	3	0.667
T2 Phase 1	-	-	2	-	-	3	1.333
T3 Phase 1	-	-	-	-	-	1	2
T4 Phase 1	-	-	-	1	-	2	0.25
T4 Phase 2	-	-	-	0	-	1	0.5
T5 Phase 1	-	-	-	1	0	2	0.5
T5 Phase 2	-	-	-	0	1	2	0.25
T6 Phase 1	-	-	-	-	-	1	1
T6 Phase 2	-	-	-	-	-	1	0.2
T7 Phase 1	-	-	-	-	-	1	2
T7 Phase 2	-	-	-	-	-	1	1

Table 2.7: a) Service Demands Per Invocation for Example 3.

Processor Name	Scheduling Discipline
1	Priority-Preemptive-Resume (PPR)
2	Priority-Preemptive-Resume (PPR)
3	First-In-First-Out (FIFO)
4	First-In-First-Out (FIFO)

Table 2.7: b) Processor Descriptions for Example 3.

Entity	Processor Number	Task Scheduling Discipline
T1	1	Non-Serving
T2	1	Non-Serving
T3	2	Multiple-Entry Server
T4	2	Multiple-Entry Server
T5	2	Multiple-Entry Server
T6	3	Multiple-Entry Server
T7	4	Multiple-Entry Server

Table 2.7: c) Processor Allocation and Task Scheduling Disciplines for Example 3.

Case	$\pi(T1)$	$\pi(T2)$	$\pi(T3)$	$\pi(T4)$	$\pi(T5)$
1	1	1	1	1	1
2	2	1	1	1	2
3	2	1	2	2	1
4	1	2	2	2	1
5	1	2	1	1	2

Table 2.7: d) Miernik et al. Test Cases For Priority Example 3.

Case	Task	Simulation	Miernik	E_{tol}	MOL	E_{tol}
1	Task 1	0.1362	0.1224		0.1353	
	Task 2	0.0616	0.0583	8.65	0.0640	1.69
2	Task 1	0.1498	0.1151		0.1150	
	Task 2	0.0676	0.0678	16.05	0.0681	16.21
3	Task 1	0.1703	0.1452		0.1399	
	Task 2	0.0444	0.0373	15.00	0.0365	17.84
4	Task 1	0.1357	0.1207		0.1233	
	Task 2	0.0567	0.0469	12.89	0.0430	13.58
5	Task 1	0.1166	0.0860		0.0932	
	Task 2	0.0792	0.0747	17.93	0.0741	14.58
$\bar{E}_{tol}$				14.10		12.78
E_{arel}				12.99		12.40
E_{rel}				12.94		11.47
E_{pdev}				8.23		10.46

Table 2.8: Miernik Example 3. The non-serving task throughputs X are summarized for the test cases.

2.3.5 Summary

In table 2.9 the error measures are aggregated for non-serving classes in all tests cases in all of the models considered. In general, the MOL has about the same accuracy as the SRVN approach.

The inclusion of priority servers in the models is the cause for much of the error in the estimated performance measures. It is the weakest point in the analytic solution for both the MOL and SRVN.

	SRVN	MOL
E_{tol}	11.45	11.24
E_{arel}	13.70	13.18
E_{rel}	9.14	7.16
E_{pdev}	18.49	17.98

Table 2.9: Summary of Results for Miernik Examples.

2.4 Iterations versus Convergence Tolerance

In this section, the tolerance used in the convergence tests of the MOL is altered to see what the effect is on the amount of computation needed to reach a fixed point.

Consider the following terms.

τ the tolerance for iterative convergence. If the relative difference in successive response time estimates for non-serving tasks in the MOL differ by less than tolerance τ then the response time estimates are assumed to have reached to a fixed point. The convergence criteria for Linearizer is as defined in the paper describing Linearizer [Chandy 82]. The termination criteria takes into account the possibility that the model being solved is approaching its fixed point slowly. This aspect of the Linearizer algorithm has not been altered and is not affected by τ.

S the number of times the software contention model *for* loop is executed. Each of the software submodels must be solved for each iteration of this loop.

D the number of times the device contention *for* loop is executed.

M the total number of times Linearizer is applied. It includes calls to Linearizer from both the software contention and device contention submodels.

$I(\tau)$ the total number of residence time evaluations for all applications of MVA at tolerance τ. The count reflects the most computationally intensive step of each residence time expression and accounts for the fact that some residence time expressions contain *for* loops.

$R(\tau)$ the ratio $\frac{I(\tau)}{I(0.01)}$ is a measure of the increase in the number of iterations due to a decrease in tolerance.

In tables 2.10 and 2.11, two sets of models are considered. The first is Woodside's set of 18 models that are presented in section 3.1.2 [Woods 86]. There is no device contention in the models. The second is a set of 46 models from papers by Miernik et al. [Mier 89, Mier 88]. They are presented in sections 2.3 and 3.2.2. The models contain both software and device contention. A large portion of the models have priority scheduled processors. The values for S, D, I, and R, are given for a range of tolerances $\tau = \{0.01, 0.005, 0.001, 0.0001, 0.00001\}$.

The iteration information for Woodside's models is shown in table 2.10. Since there are three levels of tasks, there are two submodels in the software contention model so Linearizer gets applied twice per iteration (therefore $M = 2\ S$). At $\tau = 0.005$, each of the software contention submodels is solved approximately four times. These submodels must be solved at least twice so that performance estimates from the first submodel affect the second submodel. It takes an average of two further iterations to reach a fixed point.

The iteration information for Miernik's models is shown in table 2.11. Most of the models in the test set have three levels, though some have four. Dividing the tolerance τ in the MOL by 1000 causes 3 times as many total iterations. The average number of times the device contention submodels models must be solved per model is $\frac{D}{46}$. Comparing the value at $\tau = 0.01$ and $\tau = 0.00001$, the number of device contention submodels being solved per model increases from 4.3 to 8.3. This defines a device contention iteration ratio R_D for the range of τ as 1.9. Similarly, R_S is defined as a ratio for the range of τ of how many times the software contention submodels must be solved before they reach a fixed point in the software contention loop. The average number of times that each software submodel must be solved $\frac{S}{D}$ increases from 3.2 to 4.4 which sets R_S as 1.4. The device contention model loop is more sensitive to changes in tolerance.

As in Woodside's models, these models have at least three levels. The submodels must be solved at least twice so that performance estimates from the first submodel affect the second submodel. It takes an average of two or three further iterations of the software contention loop to reach a fixed point, depending on the value of τ.

In Woodside's models that do not have device contention, dividing the tolerance by 1000 only causes a doubling in iterations. In the models with device contention, dividing the tolerance by 1000 triples the number of iterations. This should be expected because $R(\tau)$ is determined by R_D and R_S. For these sets of models, the number of iterations grows at a rate proportional to $\log_{10}(R(0.00001))$ which equals 3. This suggests that the MOL is a stable algorithm that tends to approach a fixed point.

τ	S	D	M	$I(\tau)$	$R(\tau)$
0.01	60	0	120	39600	1.0
0.005	68	0	136	44880	1.13
0.001	76	0	152	50160	1.27
0.0001	92	0	184	60720	1.53
0.00001	106	0	212	69960	1.77
Avg per model at 0.005	3.8	0	7.6	2493	-

Table 2.10: MOL Iterations for Woodside's 18 Models at given Tolerance.

τ	S	D	M	$I(\tau)$	$R(\tau)$
0.01	627	199	1460	994010	1.0
0.005	696	216	1624	1275918	1.28
0.001	929	252	2171	1693412	1.70
0.0001	1257	302	2969	2253117	2.27
0.00001	1674	383	3973	2966104	2.98
Avg per model at 0.005	15.0	4.70	35.3	27737	-

Table 2.11: MOL Iterations for Miernik's 46 Models at given Tolerance.

2.5 Conclusions

The MOL can be used to predict the performance of systems that contain software servers. The technique has been demonstrated in conjunction with priority scheduling at devices and has comparable accuracy with the work of Miernik et al. on the examples from their paper [Mier 89]. Significant errors have arisen only in cases where the priority server technique [Bryant 84] is expected to be inaccurate [Eager 88].

In the MOL, the LGM submodels are solved using a version of Linearizer that has been enhanced to support several new residence time expressions. Linearizer is a well known MVA algorithm and the MOL benefits from its good accuracy and convergence properties. In addition, standard MVA software permits customers with statistically identical behaviour to be represented using a single customer class. This makes it possible to represent non-serving processes that have statistically identical behaviour as a single group in an LGM. The SRVN approach does not allow this.

An alternative approach to partitioning a model of a system into layers and layered models, would be to associate each server with a submodel. This is similar to the SRVN approach and has the benefit of not requiring the introduction of flow-equivalent groups into LGMs. However such changes may affect the accuracy and convergence properties of the solution approach.

For example, good estimates for the $F_{c,d,s}(\vec{N})$ values contribute to Linearizer's accuracy. The $F_{c,d,s}(\vec{N})$ values for a customer class c are affected by its customers' residence time at all of its servers, and are computed after the customers' residence time at each of its servers has been estimated. In the MOL, the contention for each server is computed in exactly one submodel; and all of a customer's software servers are represented in the same submodel. It is a topic for future study to determine whether this provides better estimates for the $F_{c,d,s}(\vec{N})$ values than if a separate submodel were to be used for each server and whether this has any significant impact on the overall accuracy of Linearizer when used with LGMs.

In subsequent chapters, several new techniques for representing process interactions are introduced. They can be used in conjunction with the MOL to model the behaviour of software systems. Each of the techniques is implemented as a type of server that addresses a specific interaction present in software systems. The MVA algorithm is modified and extended to include appropriate residence time expressions for each new type of server. The MOL can be used to predict the performance of software systems that can be described in terms of the servers that are available.

Chapter 3

The Rendezvous Server, Multiple-Entry Server, and Multi-Server

The Ada-style rendezvous, first-come-first-served multiple-entry servers, and multi-servers are each considered in this chapter. New residence time expressions are developed that can be used to predict the performance behaviour resulting from such features. Throughout the chapter, processes and tasks are considered to be identical concepts.

In chapter 2, the Method of Layers was presented. The software servers are modelled as first-come-first-served servers that have exponentially distributed service times. Each of the tasks that call a server has the same mean service time at the server. Each of the servers is represented in the layered group model as a group containing one task. In this chapter, these restrictions are relaxed by introducing the **Rendezvous**, **Multiple-Entry**, and **Multi-Server** residence time expressions for Linearizer.

In section 3.1 the rendezvous primitive is considered. Software servers that synchronize and communicate via the rendezvous primitive are similar to the first-come-first-served servers presented in chapter 2. The only difference is that a rendezvous server can have both rendezvous and post-rendezvous service periods. It is still assumed that the service times at serving tasks and devices are exponentially distributed with the same mean for each of the task's callers and that there is one task per serving group.

The multiple-entry server is described in section 3.2. A technique is presented that considers the effects of service time distributions when predicting the performance measures of software servers. Serving tasks are permitted to have one or more entries each with its own service time distribution. It is assumed that callers are served in a first-come-first-served order with one queue shared by all entries[1]. In this way, the server provides non-fixed ordered service. In general, the means and variances of the service time distributions of devices are assumed to be known and given as input parameters. If they are not specified then service time distributions are assumed to have a squared coefficient of variation of one. The service time distributions of the software servers are computed using the presented technique. Multiple-entry servers have one task per serving group.

Finally, in section 3.3, the multi-server is considered. In the layered group models presented in chapter 2, each serving group contained one process. The technique that is presented permits many first-come-first-served servers to share a single queue of callers. The serving processes are assumed to have negative exponential service time distributions with the same mean service time for each serving process. Using the approach, the MOL can predict the performance of LGMs containing serving groups that represent one or more servers.

In each of the sections, the behaviour to be examined is described, the technique is provided, and then the technique is validated against results from simulation studies and, for the rendezvous and multiple-entry servers, from the literature.

3.1 The Rendezvous Server

In this section, a method is presented that permits a server to provide both rendezvous and post-rendezvous service. The server is referred to as a **Rendezvous Server**. A technique is presented to model its behaviour, and then the method is validated against

[1] In Ada, each task entry has its own first-come-first-served queue.

results from the literature and from simulation studies.

With the rendezvous, a caller can be released and resume its own processing while the server continues to provide service on behalf of the caller. When more than one processor is available, this can increase the level of parallelism in a system. The server is said to have two **phases** of service. The first phase is called **rendezvous processing**: the caller and server are synchronized, the server is performing a service on behalf of the caller. After the caller is released the server enters its second or **post-rendezvous** phase of service.

A task can incur a queueing delay at a rendezvous server that is caused by its own previous request for service. For example: a task can request a rendezvous with a server, be accepted by the server, and be released by the server. The server can then start its second phase of processing, but before it is able to finish, the requesting task can make another call to the server. Even if the queue at the server is empty, the calling task will be blocked until the server completes its second phase of service corresponding to the previous call. This behaviour is considered within the analysis.

3.1.1 Overview Of The Technique

In the model that is considered, each phase's processing is represented using a central server queueing network model. The central server is usually a processor. The task alternates between visits to the central server and, according to predefined routing probabilities, to devices and other tasks' entries. A sample phase is shown in figure 3.1. In the figure, the average number of visits to each of the servers is defined by the routing probabilities p_i [Triv 82]. When the server completes its first phase of service it immediately begins its second phase. When the second phase is finished the server is ready to serve another caller.

The following input parameters are required for the rendezvous server in an LGM.

- P_g The number of phases in group g, P_g is assumed to be one or two.
- $V_{p,g,h}$ The average number of visits from phase p of group g to group h per invocation of group g.

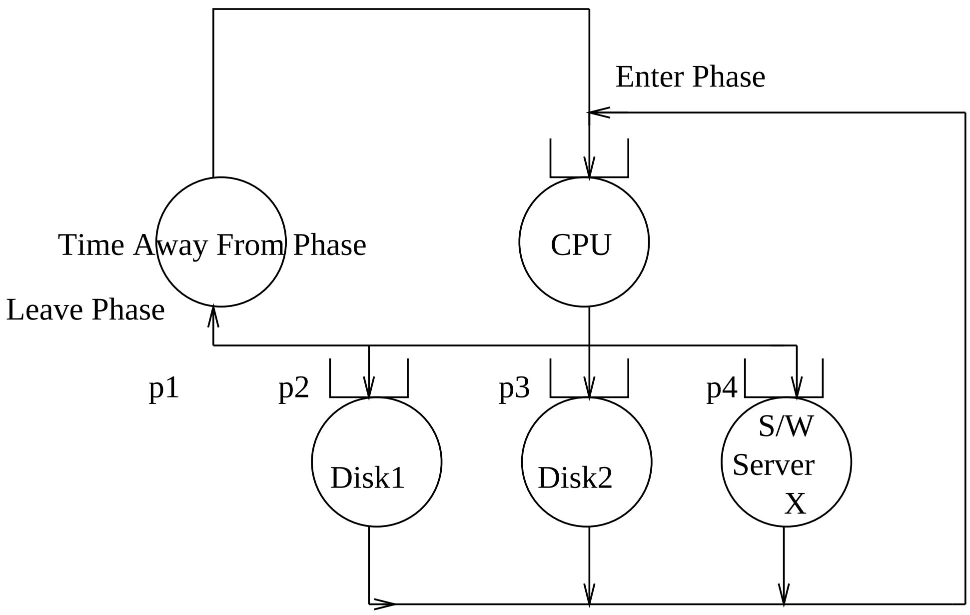


Figure 3.1: A phase of processing.

- $V_{g,h}$ The total average number of requests from group g to group h per invocation of group g. It equals $V_{1,g,h} + V_{2,g,h}$.
- $V_{p,g,k}$ The average number of visits from phase p of group g to device k per invocation of group g.
- $S_{p,g,k}$ The average service time of a visit from phase p of group g to device k.
- $V_{g,k}$ The total average number of visits from group g to device k per invocation of group g. It equals $V_{1,g,h} + V_{2,g,h}$.
- $S_{g,k}$ The average service time of a visit from group g to device k. It is defined using $S_{1,g,h}$ and $S_{2,g,h}$ weighted using the $V_{p,g,k}$.

In addition to the normal MOL output parameters, the rendezvous server provides the following values.

- $R_{p,h}(\vec{N})$ The average response time of phase p in group h.
- $R_h(\vec{N})$ The average response time of group h. It is the sum of $R_{1,h}$ and $R_{2,h}$.

MVA modelling makes use of the arrival instant theorem (see page 33) to predict the response times of customers and the queue lengths of servers. Intuitively, a customer's response time is the sum of its own resource demands and any work that is ahead of the customer at its arrival instant.

The rendezvous server's residence time expression can be expressed as follows:

$$\begin{aligned} R_{g,h}(\vec{N}) \;\leftarrow\; & \text{phase 1 service time} + \\ & \text{queueing delay due to other callers} + \\ & \text{delay for caller's own previous request for service} \end{aligned}$$

In the following detailed expression for $R_{g,h}(\vec{N})$, the effects of the server's phases upon the customer are introduced using the γ_h and $\Gamma_{g,h}(\vec{N}-1_g)$ terms. The γ_h term describes the percentage of the total processing per request for service at server h done by phase two, the post-rendezvous phase. The $\Gamma_{g,h}(\vec{N}-1_g)$ term is an estimate of the probability that a caller of class g will suffer a queueing delay due to its own previous request's phase 2, post-rendezvous, processing.

$$R_{g,h}(\vec{N}) \;\leftarrow\; D_{g,h}((1-\gamma_h) + Q_h(\vec{N}-1_g) + \Gamma_{g,h}(N \vec{-} 1_g)\,\gamma_h)$$

The following terms complete the residence time expression:

$$\begin{aligned} S_{p,h} \quad & \text{The service time of phase } p \text{ of server } h \\ D_{g,h} \;\leftarrow\; & V_{g,h}\,(S_{1,h}+S_{2,h}) \quad \text{The total demand of customer class } g \text{ at} \\ & \text{phase 1 and 2 of server } h \\ N_g \;\leftarrow\; & \text{The population of class } g \\ \gamma_h \;\leftarrow\; & S_{2,h}/(S_{1,h}+S_{2,h}) \\ \Gamma_{g,h}(\vec{N}-1_g) \;\leftarrow\; & \frac{D_{g,h}\,\gamma_h}{D_{g,h}\,\gamma_h + R_g(\vec{N}) + Z_g - R_{g,h}(\vec{N})} \end{aligned}$$

Z_g is defined as group g's average idle time.

To explain the $\Gamma_{g,h}(\vec{N} - 1_g)$ term, consider the time between completions of the customer as it is broken into four components.

a The time away from the server.

b The waiting delay at the server not including any service.

c The duration of the first phase of service. It is defined as $D_{g,h}\ (1 - \gamma_h)$.

d The duration of the second phase of service. It is defined as $D_{g,h}\ \gamma_h$.

Assume Z_g is zero. $R_{g,h}(\vec{N})$ will be the sum of b, c, and some portion of d, d'. $R_g(\vec{N})$ will be the sum of a, b, c, and the same portion of d, d'. Therefore the denominator of $\Gamma_{g,h}(\vec{N} - 1_g)$ is approximately,

$$d + (a + b + c + d') - (b + c + d')$$

which is $a + d$. Now the numerator of $\Gamma_{g,h}(\vec{N} - 1_g)$ is simply d, thus $\Gamma_{g,h}(\vec{N} - 1_g) = \frac{d}{a+d}$. As the time that the customer spends away from the server h approaches zero probability of being blocked by a previous request for service approaches one. This is similar to the approach used by Woodside [Woods 86].

The population $\vec{N}$ is used to determine $\Gamma_{g,h}(\vec{N} - 1_g)$ because the probability that a customer delays itself must be computed without the assumption that the customer has been removed from the network. In Linearizer, $\Gamma_{g,h}(\vec{N} - 1_g)$ should be assigned in the core support routine, and the value of γ_h should be assigned in the initialize routine.

3.1.2 Validating The Technique With Respect To Exact Results

Woodside provides the exact results for a suite of eighteen test cases that are used to investigate the accuracy of the SRVN method [Woods 86]. The models are all based upon the software process architecture shown in figure 3.2. The exact performance metrics for each of the models has been found using global balance techniques.

In the software process architecture, $T1$ visits $T5$. $T1$ is at level three of the architecture, and $T5$ is at level one. As was discussed in chapter two, the MOL requires that a task's requests for service only be made to tasks one level lower in the hierarchy.

For this architecture, a flow equivalent server must be introduced before the MOL can be applied. First, the models are described without the use of the flow equivalent server, then its FESC's parameters are given.

The basic model and the suite of test cases, based upon the model, are described in table 3.1. The basic model parameters are given in part a) of the table. The average number of visits from each task's phases to each other task and device is given. The service time at devices is also specified. For example, in task $T3$'s second phase of service, on average, task $T6$ is visited once and the CPU is visited twice with an average service time of one time unit per visit. The scheduling disciplines of the tasks and devices are described in part b) of the table. There is only one processor in the model and it is a delay server. It acts as the central server for each of the tasks. Effectively, each task has its own processor. The assignment of tasks to processors is given in part c) of the table. All of the tasks are assigned to processor one. The scheduling disciplines of the tasks are also given. In this example, all of the tasks are rendezvous servers. Note that task $T1$ is a non-serving rendezvous server. Finally, the different test cases are described in part d) of the table by listing the differences of a model from the basic model parameters given in part a). For example, test case twelve is described as the basic model except that task T5's phase one and two of service each have a service time at the CPU of three time units instead on one time unit, $S_{1,T5,CPU} = 3.0$ and $S_{2,T5,CPU} = 3.0$, respectively.

Now consider the flow equivalent server FESC1 that is shown in the modified model of the software process architecture in figure 3.3. It is introduced to act as a place holder that does not affect the performance behaviour of the system but enables the application of the MOL. It represents $T5$ as a server in the higher level submodel and $T1$ as a customer in the lower level submodel. FESC1 has $T1$ as its only customer and $T5$ as its only server. The new model no longer has the visit from T1 to T5 but has the following parameters $V^{new}_{T1,FESC1} = V^{original}_{T1,T5}$ and $V^{new}_{FESC1,T5} = 1$ instead. The scheduling discipline of FESC1 is DELAY.

The exact and predicted task T1 throughputs for the models are shown in table 3.2. The changes in model parameters affect the impact of post-rendezvous processing upon model throughput by changing the ratio of rendezvous to post-rendezvous processing and relative service rates of tasks. For the eighteen cases considered, the MOL has a lower average and worst case error than SRVN.

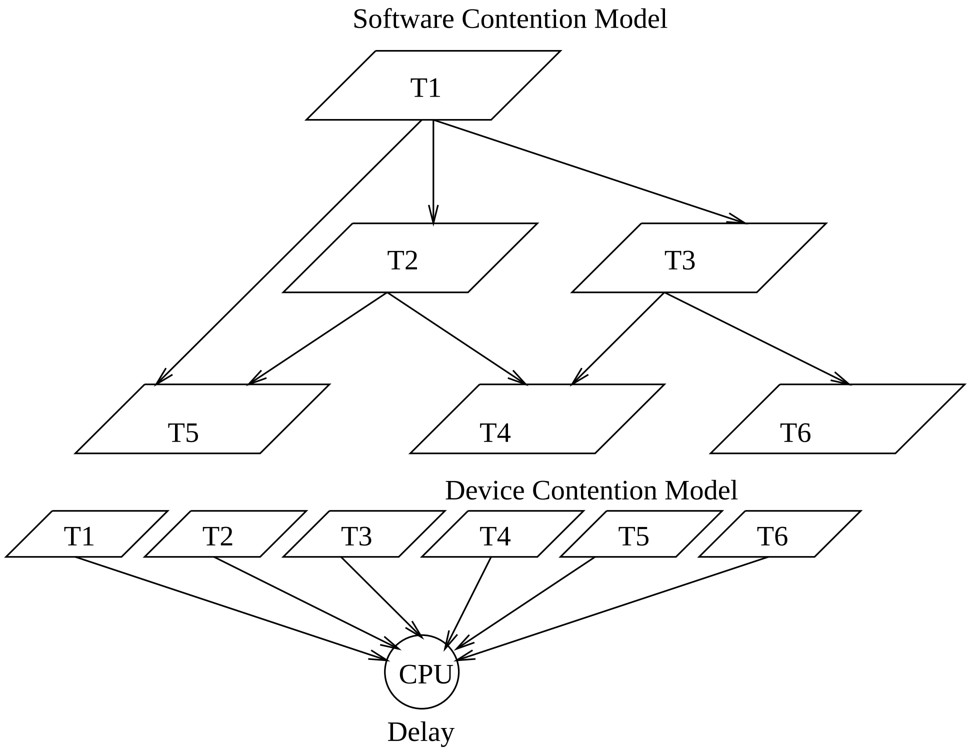


Figure 3.2: Woodside's SRVN Example.

Entity	V_{T2}	V_{T3}	V_{T4}	V_{T5}	V_{T6}	V_{CPU}	S_{CPU}
T1 Phase 1	2	5	-	1	-	9	1
T2 Phase 1	-	-	0	2	-	3	1
T2 Phase 2	-	-	3	1	-	5	1
T3 Phase 1	-	-	1	-	0	2	1
T3 Phase 2	-	-	0	-	1	2	1
T4 Phase 1	-	-	-	-	-	1	1
T4 Phase 2	-	-	-	-	-	1	1
T5 Phase 1	-	-	-	-	-	1	1
T5 Phase 2	-	-	-	-	-	1	1
T6 Phase 1	-	-	-	-	-	1	1
T6 Phase 2	-	-	-	-	-	1	1

Table 3.1: a) Task Service Demands per Invocation for Woodside's Models.

Processor Name	Scheduling Discipline
Processor 1	Delay Server (DELAY)

Table 3.1: b) Processor Descriptions for Woodside's Models.

Entity	Assigned to Processor Number	Task Scheduling Discipline
T1	1	Non-Serving
T2	1	Rendezvous Server
T3	1	Rendezvous Server
T4	1	Rendezvous Server
T5	1	Rendezvous Server
T6	1	Rendezvous Server

Table 3.1: c) Processor Allocation and Task Scheduling Disciplines for Woodside's Models.

Case	Changes To Model
1	No Changes
2	$V_{T1,T5} = 3$, $V_{T1,CPU} = 11$
3	$V_{T1,T5} = 10$, $V_{T1,CPU} = 18$
4	$V_{T1,T5} = 30$, $V_{T1,CPU} = 38$
5	$V_{2,T2,T4} = 1$, $V_{2,T1,CPU} = 3$
6	$V_{2,T2,T4} = 10$, $V_{2,T1,CPU} = 12$
7	$V_{2,T2,T4} = 30$, $V_{2,T1,CPU} = 32$
8	$V_{1,T3,T4} = 3$, $V_{1,T1,CPU} = 4$
9	$V_{1,T3,T4} = 10$, $V_{1,T1,CPU} = 11$
10	$V_{1,T3,T4} = 30$, $V_{1,T1,CPU} = 31$
11	$S_{1,T5,CPU} = 0.333$, $S_{2,T5,CPU} = 0.333$
12	$S_{1,T5,CPU} = 3.0$, $S_{2,T5,CPU} = 3.0$
13	$S_{1,T5,CPU} = 10.0$, $S_{2,T5,CPU} = 10.0$
14	$S_{1,T5,CPU} = 33.33$, $S_{2,T5,CPU} = 33.33$
15	$S_{1,T4,CPU} = 0.333$, $S_{2,T4,CPU} = 0.333$
16	$S_{1,T4,CPU} = 3.0$, $S_{2,T4,CPU} = 3.0$
17	$S_{1,T4,CPU} = 10.0$, $S_{2,T4,CPU} = 10.0$
18	$S_{1,T4,CPU} = 33.33$, $S_{2,T4,CPU} = 33.33$

Table 3.1: d) Woodside Rendezvous Model Test Cases

Parameters (Case)	No Rendezvous Processing X_{T1}	Exact X_{T1} Results	SRVN X_{T1}	%Error wrt. Exact Res.	MOL X_{T1}	%Error wrt. Exact Res.
1	0.0286	0.0175	0.0188	−7.4%	0.01846	−6.1%
2	0.0256	0.0166	0.0164	1.2%	0.01723	−5.1%
3	0.0189	0.0137	0.0136	0.7%	0.01367	−2.8%
4	0.01075	0.00851	0.00845	0.7%	0.00823	−0.2%
5	0.0286	0.0196	0.0211	7.7%	0.0203	−6.5%
6	0.0179	0.0117	0.0116	0.8%	0.01253	−1.4%
7	0.00735	0.00557	0.00567	−1.8%	0.00574	3.3%
8	0.0182	0.0122	0.0130	6.6%	0.01258	−7.5%
9	0.00769	0.00592	0.00555	6.3%	0.00587	−3.5%
10	0.00308	0.00239	0.00243	−1.7%	0.00230	1.0%
11	0.0316	0.0192	0.0204	−6.3%	0.02025	−4.8%
12	0.0241	0.0124	0.0134	−8.1%	0.01288	−6.7%
13	0.00714	0.00618	0.00626	−1.3%	0.00612	−0.4%
14	0.00214	0.00210	0.00204	2.9%	0.00206	1.7%
15	0.0316	0.0208	0.0223	−7.2%	0.0216	−4.2%
16	0.0136	0.0101	0.0101	0%	0.0106	−4.7%
17	0.00454	0.00425	0.00397	6.6%	0.00434	0.6%
18	0.00136	0.00135	0.00125	7.3%	0.00138	1.6%
Avg. Error %				4.14		3.4

Table 3.2: A comparison of predicted task T1 throughput among Exact results, the SRVN technique, and the MOL with the Rendezvous Server

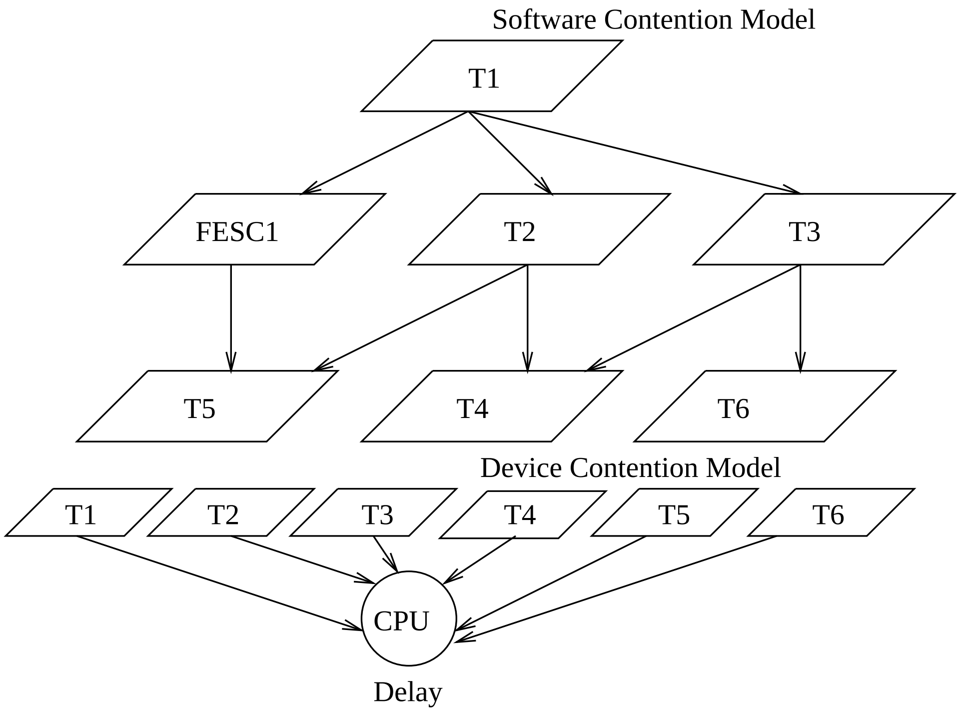


Figure 3.3: A MOL version of Woodside's SRVN Example.

3.1.3 Validating The Technique With Respect To Simulation

To further test the accuracy of the rendezvous server residence time expression, a model is considered with one class of tasks sharing a rendezvous server task. Both the requesting tasks and the server task share a delay server device that acts as their respective central servers. The following values are systematically altered so that a full set of parameter values that affect the accuracy of ther server are tested:

- N_g the number of requesting tasks sharing the rendezvous server,
- γ_h the percentage of post-rendezvous processing at the rendezvous server, $\gamma_h = R_2/(R_1 + R_2)$ where R_1 and R_2 are the average total times the rendezvous server requires for phase 1 and phase 2 processing, respectively,
- U_h the utilization of the rendezvous server.

Service time variation is investigated when considering the multiple-entry server.

The software process architecture that is used is shown in figure 3.4. A set of models that includes all combinations of the following model parameters has been chosen:

- Customer population $N_g \in \{1, 3, 5\}$,
- percentage of Rendezvous server post-rendezvous service $\gamma_h \in \{0.25, 0.5, 0.75, 1.0\}$,
- Rendezvous server utilization $U_h \in \{0.25, 0.5, 0.75, 0.98\}$.

Forty eight models are considered.

When choosing the models, for each N_g and γ_h combination, the time that customers spend at the CPU, $S_{cust,cpu}$, was systematically altered until instances of models were found with the desired rendezvous server utilization U_h.

The basic model parameters for the architecture are given in table 3.3.

The purpose of the test cases is to determine the effects that post-rendezvous processing and server utilization have upon the accuracy of the analytical technique. Because the $S_{cust,cpu}$ are chosen to achieve the desired rendezvous server utilizations, the response times of customers are high when the rendezvous server utilization is low. In the forty eight test cases, the average customer response times range from 100 time units down to 5 time units.

The error in predicted customer response time and server utilization for each of the test cases are presented in figures 3.5 and 3.6, respectively. The errors are relative errors with respect to simulation. Their values are rounded to the nearest integer and are shown at their respective points on the grid. In figure 3.5, a contour diagram of estimated average response times is imposed on top of the grid to show how changes in model parameters affect customer response times. The response time associated with each curve is printed next to the curve to one decimal place. In figure 3.5 for the case in which one customer uses the server, three contours are given. The contours show average customer response times and are, from the left, equal to 15.0, 7.0, and 4.0. The purpose of the contours is to show that the errors remain consistent as customer response times change.

From figure 3.5, errors in customer response time are most significant when there is a single customer using the rendezvous server. In this case, response times are overestimated which suggests that $\Gamma_{g,h}(\vec{N})$ is also being overestimated. The worst error is 8% and occurs at $\{N_g, \gamma_h, U_h\} = \{1, 1.0, 0.75\}$. As the number of customers increases or the fraction of post-rendezvous processing decreases, the effect of $\Gamma_{g,h}(\vec{N})$ diminishes and so do the errors in estimated response time.

From figure 3.6, the errors in predicted server utilization are greatest at high server utilizations. When $N_g > 1$, response time estimates tend to be underestimated which causes a high throughput estimate. As a result the rendezvous server utilizations are overestimated. The worst error is 9% and occurs at parameter vector $\{N_g, \gamma_h, U_h\} = \{1, 0.75, 0.98\}$.

To conclude, the rendezvous server has its greatest errors when there is a single customer and more than 50% post-rendezvous processing, and when there is more than one customer and the server is highly utilized.

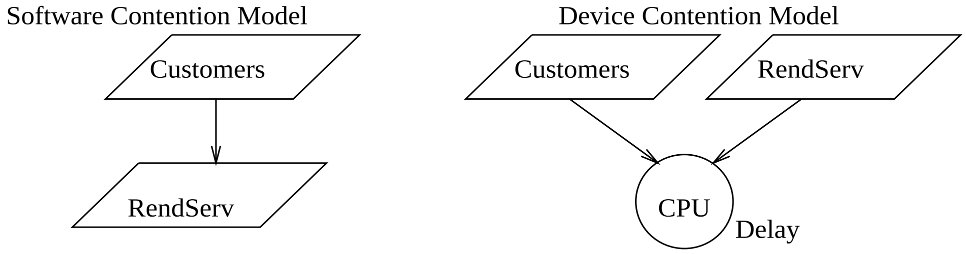


Figure 3.4: Rendezvous Server Test Case

Entity	Population	Visits to RendServ	V_{cpu}	S_{cpu}
Customers	N_g	1	5	$S_{cust,cpu}$ varies to satisfy U_h
RendServ Phase 1	1	-	$4\ \gamma_h$	1
RendServ Phase 2	.	-	$4\ (1-\gamma_h)$	1

Table 3.3: a) Rendezvous Server Basic Model For Test Cases

N_g	γ_h	U_h	$S_{cust,cpu}$
1	0.25	0.25	2.54
5	0.75	0.75	4.46
5	0.75	0.98	2.51
5	1.0	0.98	1.97

Table 3.3: b) Sample Model Parameters for the Rendezvous Server Test Case.

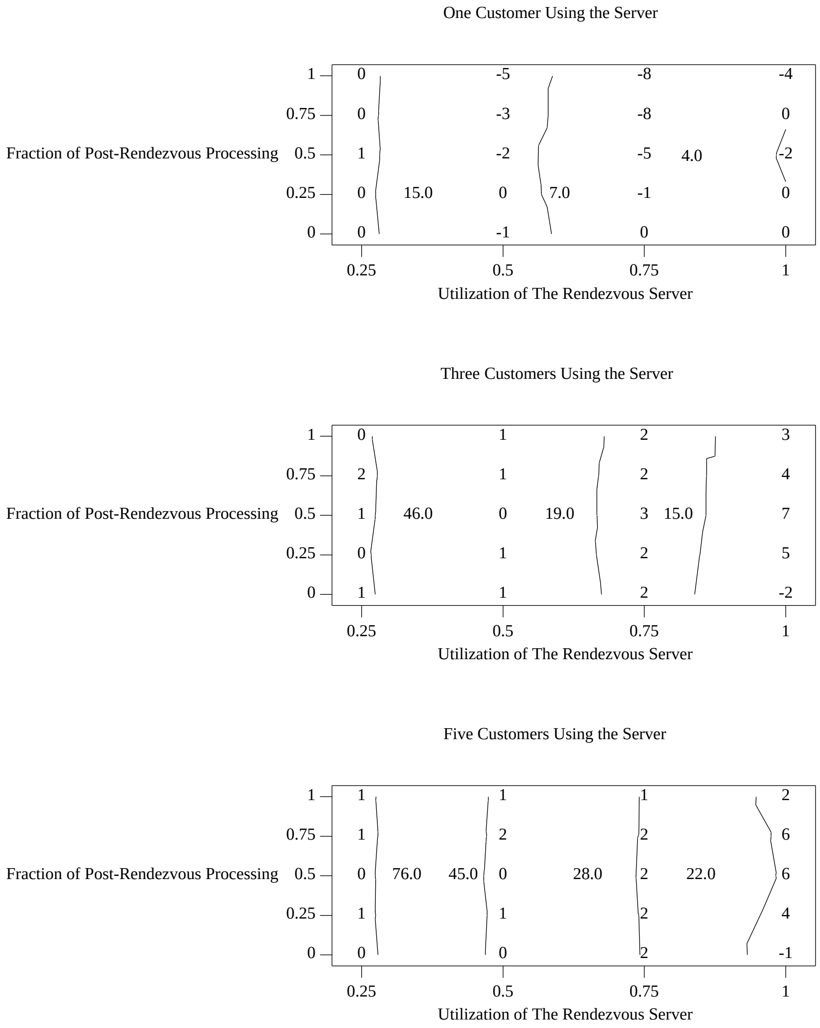


Figure 3.5: Rendezvous Server Test Case: Error in Estimated Customer Response Time, and Response Time Contours.

One Customer Using the Server

Fraction of Post-Rendezvous Processing \ Utilization of The Rendezvous Server	0.25	0.5	0.75	1
1	0	5	9	4
0.75	0	4	6	1
0.5	-3	2	4	1
0.25	0	1	2	1
0	1	-1	0	0

Three Customers Using the Server

Fraction of Post-Rendezvous Processing \ Utilization of The Rendezvous Server	0.25	0.5	0.75	1
1	0	0	0	-3
0.75	-2	-1	-3	-6
0.5	1	-1	-3	-7
0.25	-2	-2	-2	-5
0	-2	1	1	0

Five Customers Using the Server

Fraction of Post-Rendezvous Processing \ Utilization of The Rendezvous Server	0.25	0.5	0.75	1
1	-1	0	-2	-2
0.75	0	-1	-2	-6
0.5	0	-1	-2	-7
0.25	0	0	-1	-5
0	-1	0	1	0

Figure 3.6: Rendezvous Server Test Case: Error in Estimated Rendezvous Server Utilization.

3.2 Multiple-Entry Server

The multiple-entry server can be used to represent servers that provide two or more functions having significantly different resource requirements. Such servers have high service time variation and are not accurately modelled using the rendezvous-server. To overcome this, separate entries are used to characterize the different services provided by the server, and approximate MVA techniques are used to account for the high service time variation. Interactions with entries are via the rendezvous primitive. Each entry can have both rendezvous and post-rendezvous phases of service that are modelled as central server networks. The entries are assumed to share a single first-come-first-served queue of callers.

FCFS servers with high service time variation cannot be represented exactly in separable queueing models. However, Reiser has developed a technique to introduce the effects of high service time variation into such models through the use of a residence time expression [Reiser 79, Lazow 84]. The server type that corresponds to the residence time expression is referred to as HVFIFO. The technique is adapted here for use with the rendezvous-server to provide the multiple-entry server.

Multiple-entry servers have two significant characteristics to consider that are not usually represented for servers with single entries. First, the queueing delays customers incur at a server are influenced by the service time variance of each entry of the server and the relative frequency of use of the server's entries. Second, the server's behaviour is dependent upon which of its entries is in use. The way in which a multiple-entry server acts as a customer when using its servers depends upon the entry that is currently active. Both of these effects are considered.

The mean and variance of service times for devices are required as input parameters to the model. If a variance is not specified, the squared coefficient of variation is assumed to be one. Estimates for the mean response time and variance in response times of each server and its entries are computed.

Miernik et al. also consider a multiple-entry server [Mier 88]. They do not consider the server service time variance to the same level of detail as is presented here. The accuracy of the multiple-entry server technique is compared with their technique for the models presented in their paper.

The multiple-entry server can also be used to study servers that have single entries yet still have high service time variation. The models presented in section 3.1.2 are reconsidered with multiple-entry servers so that the coefficient of variation of the server's response times are used in the analysis.

3.2.1 Overview of the Technique

A multiple-entry server residence time expression is developed here that is based upon Reiser's residence time expression for servers with high service time variation and the Rendezvous server presented in the previous section. Reiser's technique requires the service time variance at a server. For the multiple-entry server in an LGM, this implies computing the squared coefficient of variation, C_v^2, and hence service time variance for each of the LGM's server's entries.

Reiser's residence time expression for the HVFIFO server introduces into the residence time expression the fact that at the arrival instant, an arriving customer is more likely to be blocked by a customer that is in service with a large service requirement rather than a small one. It can be expressed as follows [Lazow 84]:

$$R_{c,k}(\vec{N}) = V_{c,k}[S_{c,k} + \sum_{i=1}^{C} S_{i,k}[Q_{i,k}(\vec{N}-1_c) - U_{i,k}(\vec{N}-1_c)] + \sum_{j=1}^{C} L'_{j,k}\, U_{j,k}(\vec{N}-1_c)\,]$$

where,

$$L'_{j,k} = S_{j,k}/2 + \sigma_k^2/(2\, S_{j,k}).$$

C is the number of customer classes, and σ_k^2 is the service time variance at the server. At the arrival instant of a customer at server k, if a class j customer is in service, $L'_{j,k}$ is the expected remaining service time of that class j customer at server k [Klein 75]. Note that when the variance is equal to $S_{j,k}^2$ for all j then $C_v^2 = 1$ and $L'_{j,k} = S_{j,k}$; the equation reduces to that for a first-come-first-served separable queueing server with identical exponential service time requirements for each class. The service time variance, σ_k^2, at a server k is determined by the different average service time requests $S_{c,k}$ and variances in service time requests made by the arriving customers.

The residence time equation includes the known average service time requirements of each customer in the queue that is not receiving service and the unknown remaining service time of the customer in service. The identity of the customer in service is not known with certainty. The utilization term $U_{j,k}(\vec{N} - 1_c)$ represents an estimate of the probability that an arriving class c customer will find a class j customer in service at server k. The mean residual life [Klein 75] of the customer found in service (if any) by an arriving class c customer at server k is given as:

$$\sum_{j=1}^{C} L'_{j,k} \; U_{j,k}(\vec{N} - 1_c).$$

For the multiple-entry server, the queueing delay incurred by an arriving customer at a server is determined by the service times of entries requested by customers waiting for service and the mean residual life of the customer already in service at the serving task. Reiser's technique accounts for such behaviour but requires the service time variance for each of the server's entries to compute the mean residual life of a customer in service.

The method that is used to compute the service time variance for server entries is based upon the standard queueing theory **series-parallel** (SP) model (see figure 3.7) [Klein 75]. An SP model can be used to determine the mean and variance of service times when probabilistically choosing one event from a set of events that have different service time distributions.

An SP model is defined by its branching probabilities, the number of stages on each branch, and the service times of the stages. The service times of the stages are exponentially distributed with stages on the same branch having the same mean service time. Upon arrival of a customer at the SP model, one of the branches is selected, probabilistically, from the set of branches. Service is then obtained from each of the stages on the branch. Afterwards, the customer leaves the model. Only one customer receives service from the model at any time. Standard formulas that are used to compute the average service time, variance in service times, and squared coefficient of service times at the SP model are given later in this section.

The following is a procedure for the construction of an SP model. It is used to choose branching probabilities and stage service times used to model a set of events with known service time distributions. Depending on its squared coefficient of variation of service times, each event adds either one or two branches with one or more stages to the SP model. The branches are conditioned on the probability of choosing the event. Let S_e be the service time of the event e, $C_v^2(e)$ be its squared coefficient of variation, and π_e be the probability of selecting event e. The three cases that must be considered are:

- $0 < C_v^2(e) < 1$: The event has low variance in service times. The event can be represented using one branch b with a series of $r_b = \lceil 1/C_v^2(e) + 0.5 \rceil$ stages each with mean service time $S_b = S_e/r_b$. The representation is only approximate because an integral number of stages is introduced. The probability of selecting the branch is the probability of selecting the event $\pi_b = \pi_e$.

- $C_v^2(e) \cong 1$: The event can be represented with one branch b that has a single stage, $r_b = 1$, with service time S_b equal to S_e. The probability of selecting the branch is the probability of selecting the event $\pi_b = \pi_e$.

- $C_v^2(e) > 1$: The event has high variance in service times. The event can be represented as two branches, b_1 and b_2, with single stages $r_{b_1} = 1, \quad r_{b_2} = 1$ that have service times S_{b_1} and S_{b_2}, and branching probability $\pi_{b_1} = \pi_e \ a$ and $\pi_{b_2} = \pi_e \ (1-a)$, respectively [Sevcik 77a], where:

$$a = (1/2)(1 - \sqrt{(C_v^2(e) - 1)/(C_v^2(e) + 1)})$$
$$S_{b_1} = 2\ a\ S_e$$
$$S_{b_2} = 2\ (1-a)\ S_e\ .$$

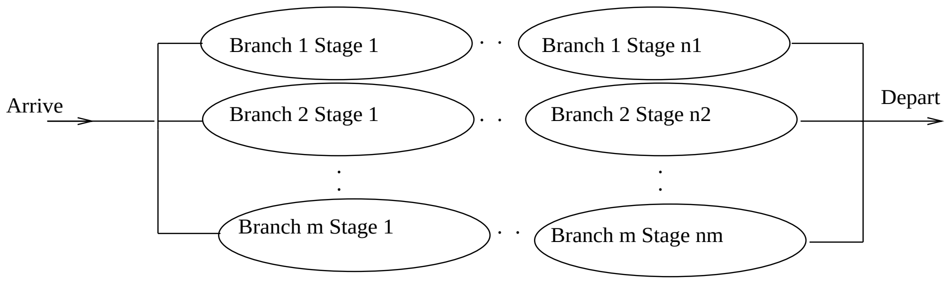


Figure 3.7: Series-Parallel (SP) Model. Service times associated with each stage satisfy the exponential distribution.

After associating each event with the appropriate branches, the expected service time S, squared coefficient of variation of service times C_v^2, and variance of service times σ^2 for a visit to the SP model can be found.

The mean service time S of a visit to the SP model is

$$S = \sum_{b=1}^{B} \pi_b \; S_b \; r_b,$$

the squared coefficient of variation C_v^2 of the service times for visits is[2]

$$C_v^2 = [(\sum_{b=1}^{B} \pi_b \; (S_b)^2 (r_b + 1)/r_b)/(\sum_{b=1}^{B} \pi_b \; (S_b))^2] - 1,$$

and the variance in service times for visits is

$$\sigma^2 = C_v^2 \;\; S^2.$$

The outputs of the formulas for the SP model are (S, C_v^2, σ^2).

In the models that are considered in the thesis, entries are represented as a series of two phases. An example of a phase structure is shown in figure 3.1 on page 98. The execution of a phase of an entry corresponds to visits to devices and software servers. Once phase 1 completes, phase 2 begins. When phase 2 completes, the entry has fulfilled its resource requirements and the server is ready to accept another caller.

The following additional LGM parameters are required for the multiple-entry server:

- E_g The number of entries in group g, E_g is assumed to be one or two.
- $P_{e,g}$ The number of phases in entry e of group g, $P_{e,g}$ is either one or two.
- $V^f_{p,e,g,h}$ The average number of requests for service in phase p of entry f of group g for entry e of group h per invocation of entry f.
- $V^f_{p,e,g,k}$ The average number of requests for service in phase p of entry f of group g for entry e of device k (Devices only have one entry).
- $S^f_{p,e,g,k}$ The average service time of a request for service in phase p of entry f of group g for entry e of device k.
- $\pi_{f,g}(\vec{N})$ the probability that a customer of group g is executing on behalf of its entry f.

The following parameters can be derived from the input parameters:

[2] See appendix B for derivations.

- $V^f_{e,g,h}$ The total average number of requests for service by entry f of a customer of group g to entry e of server h per invocation of entry f. It is defined as the sum of $V^f_{1,e,g,h}$ and $V^f_{2,e,g,h}$.
- $V_{e,g,h}$ The total average number of requests for service by a customer of group g to entry e of server h. It is defined as the sum of the $V^f_{e,g,h}$ weighted by the $\pi_{f,g}(\vec{N})$.
- $V^f_{e,g,k}$ The total average number of requests for service by entry f of a customer of group g to entry e of device k per invocation of entry f. It is defined as the sum of $V^f_{1,e,g,k}$ and $V^f_{2,e,g,k}$.
- $S^f_{e,g,k}$ The average service time of a single request for service by entry f of a customer of group g to entry e of device k. It is defined using $S^f_{1,e,g,k}$ and $S^f_{2,e,g,k}$ weighted by the $V^f_{p,e,g,h}$ values.
- $V_{e,g,k}$ The total average number of requests for service of a customer of group g to entry e of device k per invocation of group g. It is defined using the $V^f_{e,g,k}$ values weighted by the $\pi_{f,g}(\vec{N})$ values.
- $S_{e,g,k}$ The average service time of a single request for service by a customer of group g to entry e of device k. It is defined using the $S^f_{e,g,k}$ values weighted by the $\pi_{f,g}(\vec{N})$ and $V^f_{e,g,k}$ values.

The following output values become available from the MOL.

- $R_{e,g}$ The average response time of entry e of group g.
- $R_{p,e,g}$ The average response time of phase p of entry e of group g.
- $\pi_{e,h}$ The probability that a customer of group h will arrive at entry e of group h.

The method presented computes the average response time and variance of phases, entries, and servers. An SP model is required to compute the variance of each phase and each server in the model. The computation is done for groups at each level, by level, in order 1 to L. Groups at level 1 only visit devices and their service time distributions are input parameters for the model. The SP models for groups at higher levels may depend on the computed service time distributions of lower level groups that provide them with service.

Consider the following notation that describes the components that affect a phase's average response time and variance. Let phase p be a phase that uses one or more servers; the visited servers can include both devices and software servers. For the purpose of notational simplicity, the analysis of the phase is presented without indicating the server or entry to which it belongs. The analysis is the same for both rendezvous or post-rendezvous phases.

S_{cs} The expected service time of a visit to the central server cs of phase p;

σ^2_{cs} The service time variance of the central server cs of phase p;

q The probability the task executing phase p leaves the phase after a visit to the central server cs. (In figure 3.1, $q = p1$.)

v_{ncs} The probability the task executing phase p visits non-central server ncs after a visit to the central server. (In figure 3.1, the non-central servers are Disk1, Disk2, and S/W Server X. For Disk2, $v_{Disk2} = p3$.)

v The probability the task executing phase p visits some non-central server after a visit to the central server, $v = \sum_{ncs=1}^{T} v_{ncs}$, where T is the number of non-central servers. T includes both devices and software servers. (In figure 3.1, $T = 3$ and $v = p2 + p3 + p4 = 1 - q$.)

When a phase is executing, the task visits the central server then with probability q leaves the phase and with probability v visits a non-central server and returns to the central server.

The mean and variance for the phases can be computed as follows. To begin, the mean and variance of a visit to a non-central server must be found. A series-parallel model is created for this purpose. Each of the non-central servers visited by the phase is included in the SP model according to the rules provided for constructing an SP model. The event routing probabilities, v_{ncs}, from the phase are divided by v because one of the non-central server events will be executed. Thus branching probabilities in the SP model sum to unity.

Examples of SP models used to represent the non-central server event selection within a phase and entry selection are shown in figure 3.8. In part A) of the figure, a server's phase is shown. In part B), the phase's corresponding SP model is shown. It is used to estimate the average service time and squared coefficient of variation of a visit to a non-central server within the phase. Ultimately, the squared coefficient of variation is computed for entries. The boxes formed with dotted lines indicate that the procedure used to create an SP model has associated the set of branches and stages within the box with a single event. In part B), two branches with single stages were required to represent the visit to the Disk because it has a $C_v^2 > 1$. The ServTask has a $C_v^2 < 1$ so it is represented using one branch with several stages.

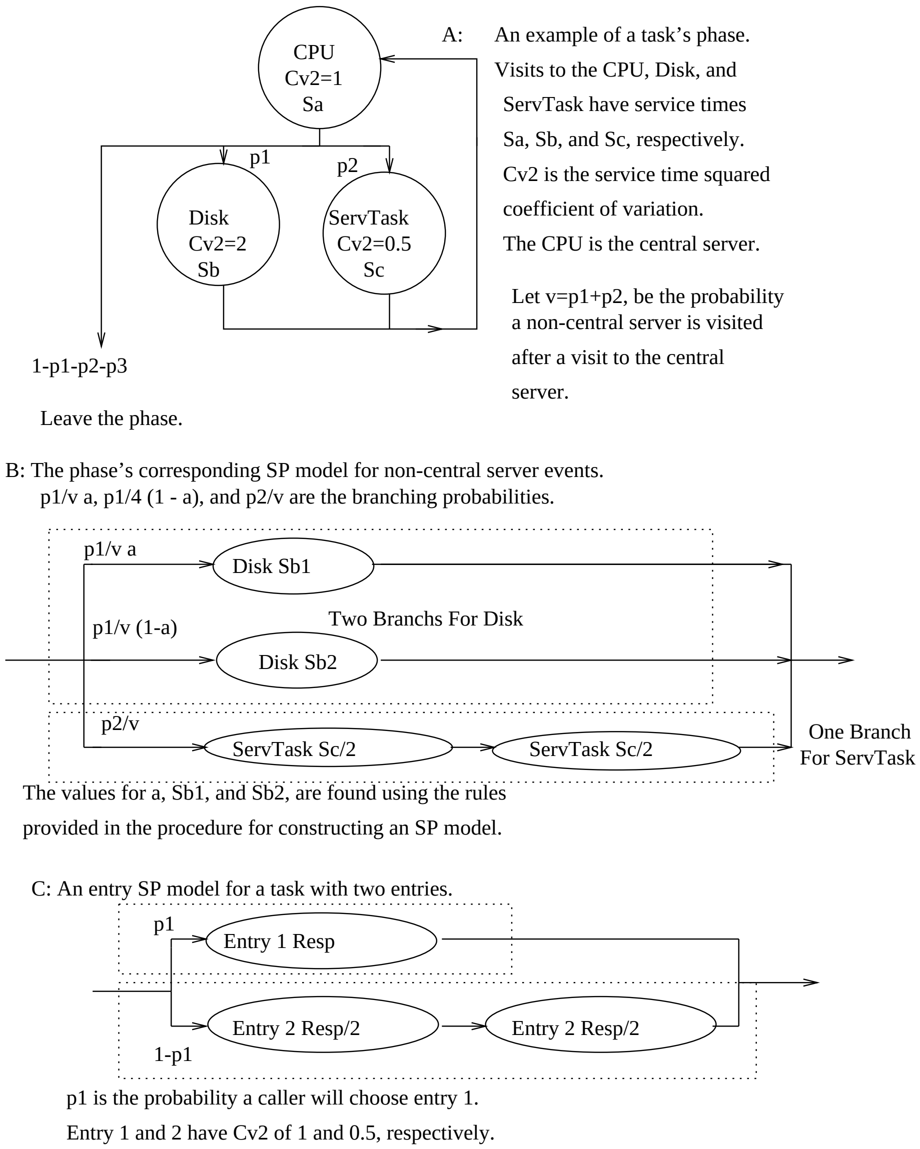


Figure 3.8: An Example Showing How To Find The C_v^2 Of A Task

For a phase p, the SP model outputs (S, σ^2) are used to compute the following values.

r_{ncs} The expected time required for a visit to one of the non-central servers.

$$r_{ncs} = S$$

σ^2_{ncs} The service time variance of a visit to one of the non-central servers.

$$\sigma^2_{ncs} = \sigma^2$$

Now, it is possible to compute the mean and variance of an iteration of a phase's service. An iteration encompasses a visit to the central server, and a visit to a non-central server or an exit from the phase.

r_{iter} The expected response time of one iteration of a phase.

σ^2_{iter} The variance in response times of one iteration of a phase.

The response time and variance for one iteration are:

$$r_{iter} = S_{cs} + v\, r_{ncs},$$

and[3]

$$\sigma^2_{iter} = \sigma^2_{cs} + \ v\, \sigma^2_{ncs} + v\,(1 - v)\,(r_{ncs})^2.$$

Consider the expected service time and variance of phases and entries.

$R_{p,e}$ The expected response time of phase p in entry e.

$\sigma^2_{p,e}$ The response time variance of phase p of entry e.

R_e The expected response time of entry e.

σ^2_e The response time variance of entry e.

[3]See appendix B for the derivation

Let r_{iter} and σ^2_{iter} correspond to phase p of entry e. The average response time and variance for a central server phase p of an entry e are based upon the time required for an iteration, r_{iter}, and the average number of iterations. The average number of iterations is determined by the probability, q, of leaving the phase:

$$R_{p,e} = r_{iter}/q,$$

and[4]

$$\sigma^2_{p,e} = \frac{\sigma^2_{iter}}{q} + \frac{1-q}{q^2}\,(r_{iter})^2.$$

The expected remaining life of a phase p is defined as

$$L^{'}_{p,e} = R_{p,e}/2 + \sigma^2_{p,e}/(2\ R_{p,e})$$

An entry e is described as a series of two phases. The average response time and variance for an entry are:

$$R_e = R_{1,e} + R_{2,e},$$

and since the phases are independent

$$\sigma^2_e = \sum_{p=1}^{2} \sigma^2_{p,e}.$$

From this the squared coefficient of variation of entries is found:

$$C^2_v(e) = \frac{\sigma^2_e}{(R_e)^2}.$$

It is used to construct its callers' SP models.

Now consider the average response time and squared coefficient of variation of response times for a server.

R The average response time of server h.

σ^2 The response time variance of server h.

C^2_v The squared coefficient of variation of the response times of server h.

[4] See appendix B for the derivation

First it is necessary to determine each entry's probability of being selected. The weighted sum of throughputs is used to estimate this value. Consider the following terms:

- $X_g(\vec{N})$ The throughput of class g;
- $X_{total,h}(\vec{N})$ The total throughput by all classes at server h;
- $\pi_{e,h}(\vec{N})$ The weighted frequency of a customer choosing entry e.

Assuming that successive acceptances of entries are statistically independent, the terms are related in the following way:

$$\begin{aligned} X_{total,h}(\vec{N}) &\leftarrow \sum_{g \,\epsilon\, G} \sum_{e=1}^{E_h} X_g(\vec{N})\, V_{e,g,h} \quad \forall\, h \\ \pi_{e,h}(\vec{N}) &\leftarrow \sum_{g \,\epsilon\, G} X_g(\vec{N})\, V_{e,g,h} / X_{total,h}(\vec{N}) \;\; \forall\, e, h. \end{aligned}$$

An SP model is constructed, according to the given procedure, that uses the average entry response time and variance, and selection frequency of each of the server's entries. An example of an SP model used to represent entry selection is shown in part C) of figure 3.8. It is used to estimate the average service time and squared coefficient of variation of service time for the server. This is used to determine the mean residual life of the server's entries. In figure 3.8C) Entry 1 has a $C_v^2 = 1$ so it is represented using one branch with one stage.

The SP model outputs (S, C_v^2, σ^2) provides the following values:

R The average response time of the server is the average service time as output from the SP model $R = S$.

C_v^2 The squared coefficient of variation of response times of the server is the C_v^2 output from the SP model.

σ^2 The variance of response times of the server is the variance, σ^2, as output from the SP model.

The task's variance σ^2 is used to define the expected remaining service time, $L'_{e,j,h}$, of group j customer in service at entry e of a server h.

$$L'_{e,j,h} = R_{e,h}/2 + \sigma^2/(2\ R_{e,h}).$$

Thus the variance in expected remaining service time due to the frequency with which processes use each entry is considered.

The residence time expression of the multiple-entry server combines the rendezvous server residence time expression from section 3.1.1 and Reiser's expression for a server with high service time variation. In the expressions, when a customer of class i visits server h, $\eta_{e,i,h}$ is the weight that describes how frequently it visits entry e of server h.

$$\eta_{e,i,h} \quad \leftarrow \quad V_{e,i,h} / \sum_{f=1}^{E_h} V_{f,i,h} \ \ \forall\ e, i, h$$

The term is included in the analysis because at a customer's arrival instant, it must predict which entry each of the customers in the queue is visiting.

$$\begin{aligned} R_{e,g,h}(\vec{N}) \quad \leftarrow \quad & V_{e,g,h}[S_{e,g,h}\ (1-\gamma_{e,h}) + \sum_{e=1}^{E_h}\sum_{i=1}^{G} \eta_{e,i,h}\ S_{e,i,h}[Q_{e,i,h}(\vec{N}-1_g) - U_{e,i,h}(\vec{N}-1_g)] + \\ & \sum_{e=1}^{E_h}\sum_{j=1}^{G} \eta_{e,j,h}\ L'_{e,j,h}\ U_{e,j,h}(\vec{N}-1_g)\ + \sum_{e=1}^{E_h} \eta_{e,g,h}\ \ L'_{2,e,h}\ \Gamma_{e,g,h}(\vec{N}-1_g)] \end{aligned}$$

In the residence time expression, $L'_{2,e,h}$ is the expected remaining service time of the second phase of service of entry e of group h. It was denoted earlier as $L'_{p,e}$.

The terms γ_k and $\Gamma_{g,h}(N \vec{-} 1_g)$ are defined in section 3.1.1. They must be determined for each entry of the server to provide the terms $\gamma_{e,h}$ and $\Gamma_{e,g,h}(N \vec{-} 1_g)$.

Now consider the characterization of a multiple-entry server g with E_g entries when it acts as a customer. It is described using one entry with one phase. The entry represents the sum, weighted by $\pi_{e,g}(\vec{N})$, of its group's entries' phases' requests for service. This affects the $V_{e,g,h}$ values used to compute $\eta_{e,g,h}$, and the $S_{e,g,k}$ values of group g when it is considered as a customer in the device contention model. Thus the group is represented by its average behaviour in the next submodel.

In Linearizer, all of the MVA variables need to be augmented to include entry information. For example, $D_{c,k}$ becomes $D_{e,c,k}$, and $F_{c,d,s}(\vec{N})$ becomes $F_{e,c,d,s}(\vec{N})$. The entries of a server are treated like separate $F_{c,d,s}(\vec{N})$ values in the original Linearizer. The variables $\Gamma_{e,g,h}(N \vec{-} 1_g)$, $\pi_{e,h}(\vec{N})$, $L^{'}_{p,e}$, and $L^{'}_{e,g,h}$ must all be assigned in the core support routine. The values of $\eta_{e,g,h}$ and $\gamma_{e,h}$ should be assigned in the initialize routine.

3.2.2 The Multiple-Entry Model by Miernik et al.

Miernik et al. investigate a multiple-entry server [Mier 88]. For each server, the variance of service times of each of its entries is estimated and used to predict average entry residence times. However, the variance of a serving task's entries is estimated with the assumption that each of the visits to processors and other software servers have exponentially distributed service times. This differs from the approach considered in this thesis. The effort of their work was to characterize task resource requirements based upon the identity of task's callers, the entries they use, and their throughputs.

Miernik et al. consider a four level software model that contains a task with two entries. The tasks in the model share three processors with squared coefficients of variation for their service times equal to one. The model is shown in figure 3.9 and model parameters are given in table 3.4. In the test cases, the changes in the service requirements at the processor 3 correspond to various speeds of processing at processor 3. The first test case corresponds to the base case.

In figure 3.10, the corresponding MOL version of the model is shown. It has the same model parameters except that T1 no longer visits T4, and $V^{new}_{T1,Fesc} = V^{original}_{T1,T4}$ and $V^{new}_{Fesc,T4} = 1$, instead.

The models have been solved using the MOL and the throughput estimates for task T1 are presented along with those of Miernik et al. in table 3.5. From table 3.5, it seems that the analysis introduced for the multiple-entry server is about as accurate as the approach of Miernik et al. for the models considered in their paper. However, it must be noted that the MOL assumes the devices are scheduled as PS. The SRVN algorithm distinguishes between each of a task's entries' phases separately when predicting device contention. The MOL does not support this feature. The models have been simulated with PS scheduled devices and the results are compared with the estimates of the MOL in table 3.6.

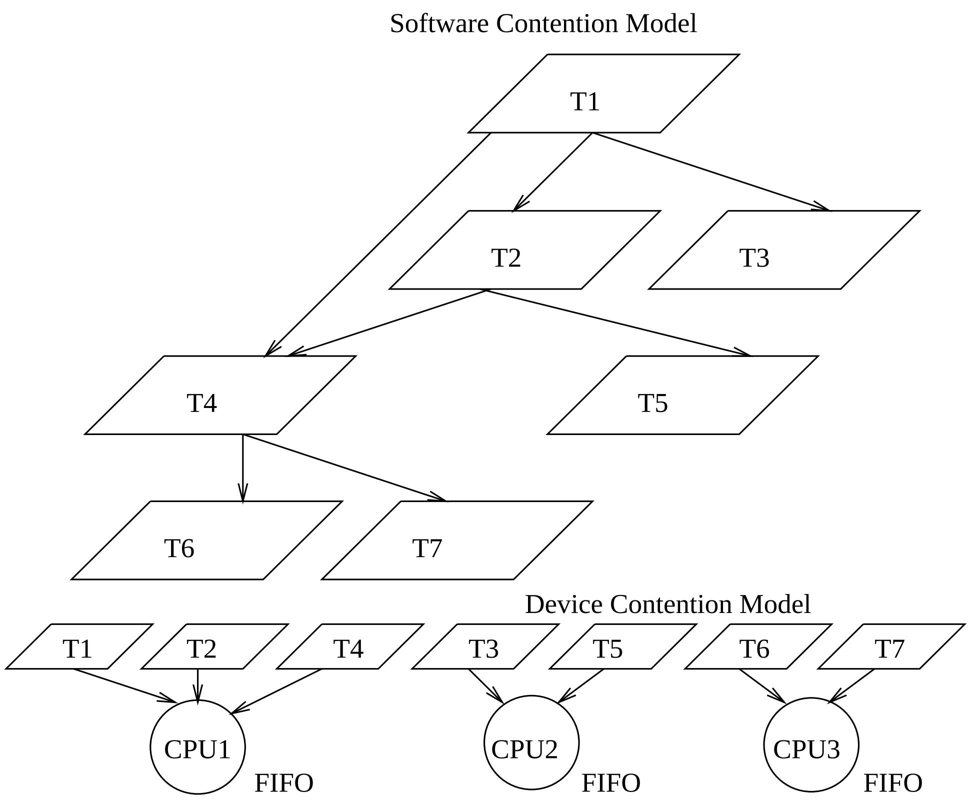


Figure 3.9: Miernik et al.'s Multiple-Entry Model

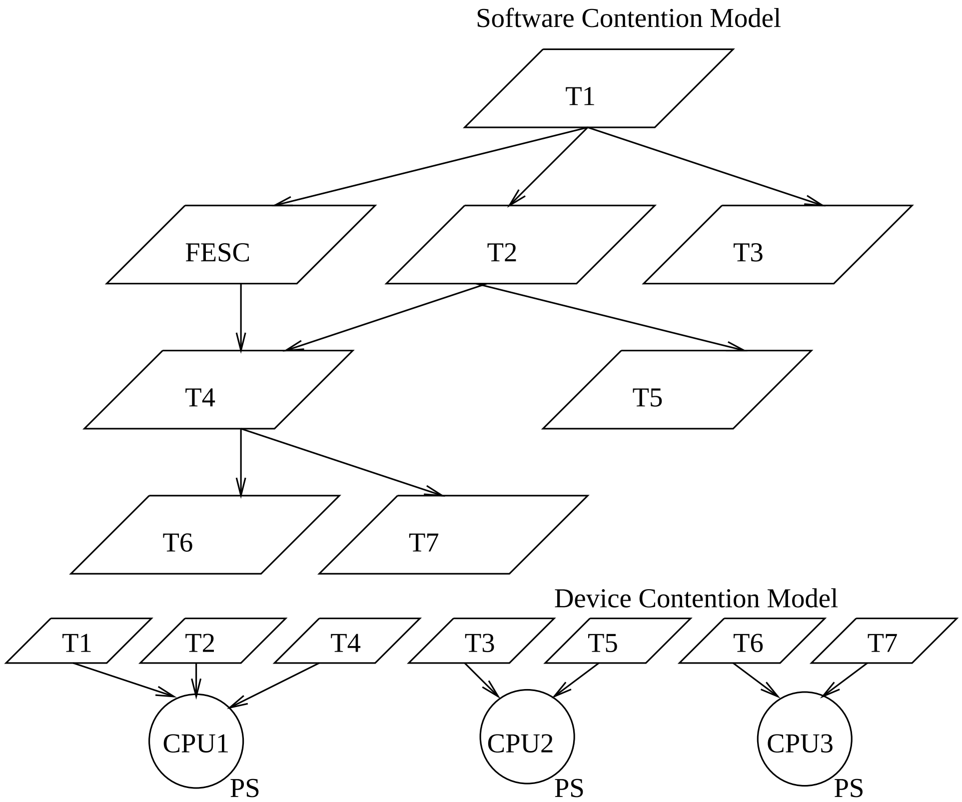


Figure 3.10: An LGM corresponding to Miernik's Multiple-Entry Model

Entity	T2	T3	T4 Entry 1	T4 Entry 2	T5	T6	T7
T1 Phase 1	0	1	2	-	-	-	-
T1 Phase 2	1	0	0	-	-	-	-
T2 Phase 1	-	-	-	2	0	-	-
T2 Phase 2	-	-	-	1	1	-	-
T4 E1 Phase 1	-	-	-	-	-	2	1
T4 E1 Phase 2	-	-	-	-	-	0	1
T4 E2 Phase 1	-	-	-	-	-	1	0
T4 E2 Phase 2	-	-	-	-	-	1	2

Table 3.4: a) Number of Rendezvous Requests per Invocation for Miernik's Model.

Entity	Phase 1 V_{cpu}	Phase 1 S_{cpu}	Phase 2 V_{cpu}	Phase 2 S_{cpu}
T1	4	0.025	2	0.1
T2	3	0.067	3	0.167
T3	1	0.1	1	1.0
T4 E1	4	0.125	2	0.05
T4 E2	2	0.1	4	0.20
T5	1	0.5	1	2.0
T6	1	2.0	1	2.0
T7	1	0.60	1	2.63

Table 3.4: b) Base Case Service Demands at the CPU for Miernik's Model.

Processor Name	Scheduling Discipline
1	First-Come-First-Served (FIFO)
2	First-Come-First-Served (FIFO)
3	First-Come-First-Served (FIFO)

Table 3.4: c) Processor Descriptions for Miernik's Model.

Entity	Processor Number	Task Scheduling Discipline
T1	1	Non-Serving
T2	1	Multiple-Entry
T3	2	Multiple-Entry
T4	1	Multiple-Entry
T5	2	Multiple-Entry
T6	3	Multiple-Entry
T7	3	Multiple-Entry

Table 3.4: d) Processor Allocation and Task Scheduling Descriptions for Miernik's Model.

Test Case	Entity	Phase 1 S_{cpu}	Phase 2 S_{cpu}
1	T6	2.0	2.0
	T7	0.6	2.63
2	T6	1.0	1.0
	T7	0.3	1.30
3	T6	0.5	0.5
	T7	0.15	0.65
4	T6	0.25	0.25
	T7	0.075	0.325
5	T6	0.125	0.125
	T7	0.0375	0.162

Table 3.4: e) Test Case Service Demands at CPU for Miernik's Model.

Case	Simulation X_{T1}	Miernik et al. X_{T1}	Error %	MOL X_{T1}	Error %
1	0.0119	0.0125	-5.0	0.0112	5.9
2	0.0238	0.0229	3.8	0.0222	6.7
3	0.0476	0.0435	8.6	0.0416	12.6
4	0.0785	0.0711	9.4	0.0720	8.3
5	0.1094	0.0990	9.5	0.1030	5.9
Avg % Error	-	-	7.3	-	7.9

Table 3.5: Throughput For Task T1 of Miernik's Model With Multiple Entries.

Case	Simulation X_{T1}	MOL X_{T1}	Error %
1	0.0116	0.0112	3.4
2	0.0230	0.0222	3.5
3	0.0438	0.0416	5.0
4	0.0730	0.0720	1.4
5	0.1051	0.1030	2.0
Avg % Error	-	-	3.1

Table 3.6: Throughput For Task T1 of Miernik's Model With Multiple Entries: Simulated with PS scheduled devices.

3.2.3 Using the Multiple-Entry Server For Single Entry Servers with High Service Time Variation

The multiple-entry server has been used to re-evaluate the models presented by Woodside [Woods 86] and considered in section 3.1.2. The new results are presented in table 3.7 and can be compared with the previous results in table 3.2. They are more accurate with respect to the exact results than when using the rendezvous server approach alone. The average error has decreased from 3.3% to 2.9% and the worst error has decreased from 7.4% to 6.5%. Neither of which are significant.

Parameters (Case)	No Rendezvous Processing X_{T1}	Exact X_{T1} Results	SRVN X_{T1}	%Error wrt. Exact Res.	MOL X_{T1}	%Error wrt. Exact Res.
1	0.0286	0.0175	0.0188	−7.4%	0.01811	−3.5%
2	0.0256	0.0166	0.0164	1.2%	0.01701	−2.5%
3	0.0189	0.0137	0.0136	0.7%	0.01376	−0.4%
4	0.01075	0.00851	0.00845	0.7%	0.00840	1.3%
5	0.0286	0.0196	0.0211	7.7%	0.02040	−4.1%
6	0.0179	0.0117	0.0116	0.8%	0.01164	0.5%
7	0.00735	0.00557	0.00567	−1.8%	0.00533	4.3%
8	0.0182	0.0122	0.0130	6.6%	0.01282	−5.1%
9	0.00769	0.00592	0.00555	6.3%	0.00604	−2.1%
10	0.00308	0.00239	0.00243	−1.7%	0.00235	1.63%
11	0.0316	0.0192	0.0204	−6.3%	0.01961	−2.1%
12	0.0241	0.0124	0.0134	−8.1%	0.01285	−3.6%
13	0.00714	0.00618	0.00626	−1.3%	0.00595	3.7%
14	0.00214	0.00210	0.00204	2.9%	0.00196	6.5%
15	0.0316	0.0208	0.0223	−7.2%	0.0213	−2.3%
16	0.0136	0.0101	0.0101	0%	0.01018	−0.8%
17	0.00454	0.00425	0.00397	6.6%	0.00408	3.9%
18	0.00136	0.00135	0.00125	7.3%	0.00130	4.0%
Avg. Error %				4.14		2.9

Table 3.7: A comparison of the estimated Task T1 throughputs amongst Exact results, the SRVN technique, and the MOL using the Multiple-Entry Server.

3.2.4 Validating The Technique With Respect To Simulation

To test the accuracy of the multiple-entry server residence time expression, a set of models that contain a multiple-entry server have been simulated. The results of the analysis are compared with the simulated results. The following model parameters are systematically altered to ensure a wide range of scenarios are examined:

- N_g the number of customers sharing the server,
- π_h the percentage of entry 1 usage (used to alter the service time variance of the multiple-entry server),
- U_h the utilization of the multiple-entry server.

The software process architecture that is used is shown in figure 3.11. A set of models that correspond to all combinations of the following model parameters has been chosen:

- Customer population $N_g \epsilon \{2, 3, 5\}$,
- percentage of entry 1 service $\pi_h \epsilon \{0.1, 0.3, 0.5, 0.7, 0.9\}$,
- Multiple entry server utilization $U_h \epsilon \{0.25, 0.5, 0.75, 0.98\}$.

Sixty models are considered.

When choosing the models, for each N_g and π_h combination, the time that customers spend at the CPU, $S_{cust,cpu}$, was systematically altered until instances of models were found with the desired multiple-entry server utilization U_h.

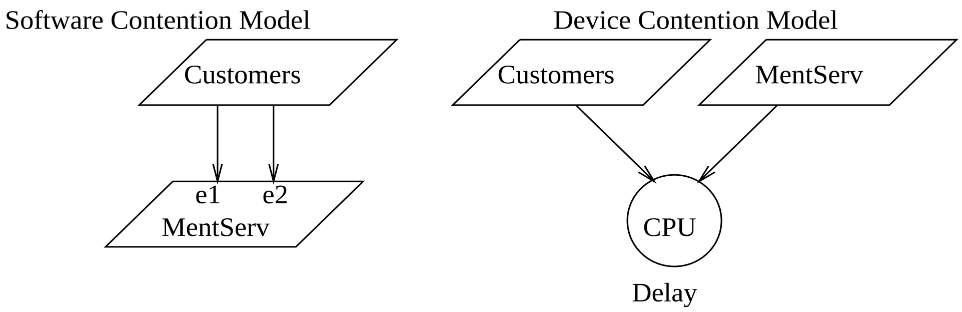


Figure 3.11: Multiple-Entry Server Test Case

Entity	Population	$V_{e1,MentServ}$	$V_{e_2,MentServ}$	V_{cpu}	S_{cpu}
Customers	N_g	$4\ \pi_h$	$4\,(1-\pi_h)$	5	$S_{cust,cpu}$ chosen
-	-	-	-	-	to satisfy U_h
MentServ Entry1	1	-	-	1	10
MentServ Entry2	-	-	-	1	1

Table 3.8: a) Model Parameters for the Multiple-Entry Server Test Case

N_g	π_h	U_h	Customers $S_{cust,cpu}$
2	0.1	0.25	50.0
2	0.1	0.5	20.3
2	0.7	0.98	0.391
2	0.9	0.25	10.4
5	0.9	0.98	3.42

Table 3.8: b) Sample of Model Parameters for the Multiple-Entry Server Test Case.

Since the customer service times at the CPU, $S_{cust,cpu}$, are chosen to achieve the desired multiple-entry server utilizations, high customer response times correspond to low server utilizations. The response time estimates for customers range from over 400 time units down to 2 time units.

The error in predicted customer response time and server utilization for each of the test cases are presented in figures 3.12 and 3.13, respectively. The errors are relative errors with respect to simulation. They are shown on a grid. In figure 3.12, a contour diagram is imposed on top of the grid to show how changes in model parameters affect customer response times.

The test cases are chosen to determine the effects of a multiple-entry server's coefficient of variation upon the accuracy of average response time estimates of customers that use the server. The multiple entry server has respective average service times of 10 and 1 time units at its two entries. The coefficient of variation of service times at each entry is 1. Changes in server C_v^2 are achieved by changing the ratio of entry usage. The maximum C_v^2 is 5 and occurs when $\pi_h = 0.9$. An alternate approach would have been to fix the ratio of entry usage and alter the entry service time variation. Either approach achieves the desired effect. The C_v^2 of the multiple-entry server is given for each of the test cases in table 3.9. The squared coefficient of variation at the server does not depend upon the number of customers using the server nor the server's utilization. The predicted C_v^2 for the server (found using the SP model) was always within 2.5% of the simulated value so tables of results have not been provided.

From figure 3.12, the greatest errors in predicting customer response times arise at high multiple-entry server utilizations. As can be seen from table 3.9, the worst errors are not with a high squared coefficient of variation at the server, but when $\pi_h = 0.5$ which is when the entry usage is balanced. This suggests that a better estimate for $\eta_{e,i,h}$ may improve the accuracy of the approach. Also, the residence time expression is very dependent upon the accuracy of the server's utilization estimates at various customer population levels. The Linearizer software must estimate these values. From figure 3.13, it is clear that the utilization estimates need to be improved at high utilization levels. This is a topic for future research.

To conclude, the ratio of entry usage has a more significant effect upon the accuracy of predicted average response times and server utilizations than the squared coefficient of variation of service times. As expected, errors are worst when the utilization of the server is high.

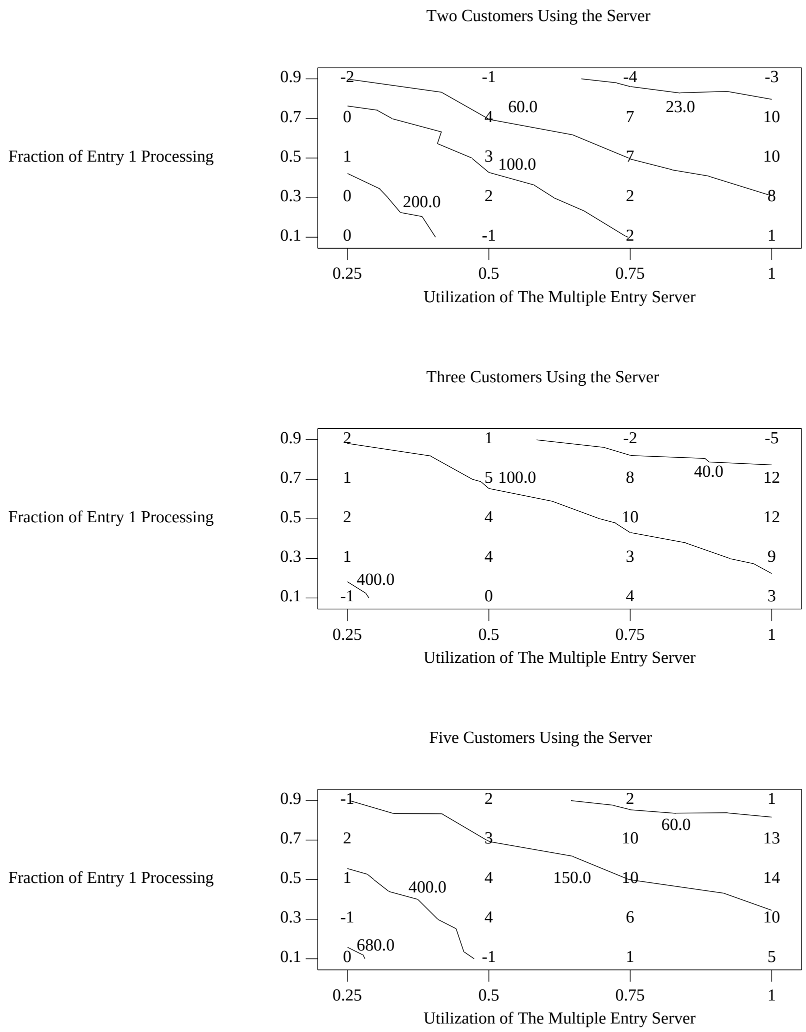


Figure 3.12: Multiple-Entry Server Test Case: Error in Estimated Customer Response Time, and Response Time Contours.

Two Customers Using the Server

Fraction of Entry 1 Processing \ Utilization of The Multiple Entry Server	0.25	0.5	0.75	1
0.9	-1	1	4	3
0.7	-2	-4	-6	-11
0.5	-2	-3	-7	-10
0.3	-1	-1	-5	-5
0.1	0	-1	-1	-2

Three Customers Using the Server

Fraction of Entry 1 Processing \ Utilization of The Multiple Entry Server	0.25	0.5	0.75	1
0.9	0	-1	2	5
0.7	-1	-5	-10	-13
0.5	-1	-4	-9	-14
0.3	-1	-2	-6	-10
0.1	-1	-1	-2	-3

Five Customers Using the Server

Fraction of Entry 1 Processing \ Utilization of The Multiple Entry Server	0.25	0.5	0.75	1
0.9	-2	-1	-1	-1
0.7	0	-7	-11	-16
0.5	-1	-4	-10	-17
0.3	0	-2	-6	-12
0.1	-1	0	-2	-5

Figure 3.13: Multiple-Entry Server Test Case: Error in Estimated Multiple-Entry Server Utilization.

Fraction of Entry 1 Processing	$U = 0.25$	$U = 0.50$	$U = 0.75$	$U = 1.00$
0.9	5.0521	4.9336	5.0394	5.1245
0.7	3.5003	3.4657	3.4936	3.4978
0.5	2.3779	2.3137	2.3467	2.3451
0.3	1.6015	1.6633	1.6486	1.6606
0.1	1.1801	1.1770	1.1670	1.1729

Table 3.9: Simulated Squared Coefficient of Variation Of Multiple-Entry Server

3.3 The Multi-Server

The Multi-Server is a service center at which a single queue of customers shares two or more servers. The modelling technique that is presented can be used when the servers are either software servers or hardware servers such as multiple processors.

In software, a multi-server can be used to provide multiple instances of a service simultaneously. This is particularly useful when a task is a software bottleneck. To correct the problem, the task can be replicated so that any member of the set of tasks can provide the service. However, areas within the serving tasks that require mutual exclusion must still be protected. Two ways to manage the problem are:

1. use semaphores to enclose the exclusive or "critical" operations,

2. place critical operations in another task that provides mutual exclusion.

An Ada version of a multi-server is given in figure 1.11 on page 22.

The multi-server technique will help system performance if there is a significant amount of code in the task that can be executed in parallel. The number of servers can be increased until the critical operations become the bottleneck.

There are multi-processor systems with operating systems that assign tasks to the next available processor as processors become available. IBM's MVS is an example of such a system. The multi-server can be used to study this behaviour as well.

In the following sections, a technique is presented that can be used to model the multi-server. Results from a simulation study are also presented to demonstrate the effectiveness of the technique.

3.3.1 Overview of the Technique

In the approach that has been developed, the multi-server is a server with two entries. The first entry is used by customers who request access to one of the multiple servers. Its second entry is used by servers that provide service to the customers. The servers have one phase of service and are permitted to visit both devices and other software servers. Each entry of the multi-server has its own residence time expression. The residence time expressions for the entries estimate the time required for a caller to receive D units of service, and for a free server to be matched with a customer in order to provide D units of service, respectively.

It is assumed that the servers have an exponential service time distribution and that a single class of customers uses the multi-server. These assumptions could be relaxed. To do so the analysis would have to be extended to consider server service time variation as was done in section 3.2. This is a topic for future research.

The servers have one entry with one phase, the multi-server has two entries but does not actually have any phases since service is provided by the servers. There are no new parameters required for LGMs that include multi-servers.

An example of a software process architecture that contains a multi-server is shown in figure 3.14 on page 149. In the analysis, it is assumed that there is no overhead at the multi-server itself. The operation that associates a customer with a server is assumed to require zero time. However if desired, overhead could be introduced by using an overhead task, as indicated in figure 3.14. The serving tasks would have a resource requirement at the first-come-first-served overhead task that matches the multi-server overhead. Each of the serving tasks makes a call to the overhead task, possibly queueing for access. This would reflect overhead incurred when matching customers with servers.

Let the number of servers, S, be greater than one, and the number of customers that share the server be N. Server utilization is determined by the utilization of the multi-server s and is defined as follows. Let $X_{g,s}(N)$ and $X_{g,s}(N-1)$ be the throughput of group g processes at multi-server s at populations N and $N-1$, respectively. The utilization of the multi-server s at population $N-1$ is:

$$U_s(N-1) = X_{g,s}(N-1) \;\; D.$$

The utilization of the multi-server s at population N is:

$$U_s(N) = X_{g,s}(N) \;\; D.$$

And finally, the servers in group h provide service on behalf of the multi-server s. The utilization of each of the servers in group h is:

$$U_h^1(N-1) = U_s(N-1)/S.$$

Define $P[Q \geq S]$ as the probability that there are S or more customers at the multi-server. An arriving customer at the multi-server will only suffer a queueing delay if all the servers are busy serving other customers. This must be taken into account in the multi-server's residence time expression.

The $M/M/m/M$ model is a closed multi-server model with a Poisson arrival process, exponentially distributed service times, m servers, and M customers [Klein 75]. It provides the exact value of $P[Q \geq S]$ for the Multi-server. Unfortunately, it requires too much computation to use with the multi-server residence time expression. Two estimates for $P[Q \geq S]$ have been considered. They are the Erlang C formula [Klein 75] for an open model of the multi-server, and another estimate (the thesis estimate) that is based on each of the server's utilization.

The Erlang C formula is:

$$P[Q \geq S] \quad = \quad \frac{\frac{(S\rho)^S}{S!}\frac{1}{1-\rho}}{\sum_{k=0}^{S-1}\frac{(S\rho)^k}{k!} + \frac{(S\rho)^S}{S!}\frac{1}{1-\rho}}$$

Where $\rho = U_h^1(N)$ is the utilization of each of the servers.

The thesis estimate is based on the assumption that the times at which the servers are busy are independent and uncorrelated. The thesis estimate is:

$$P[Q \geq S] \quad = \quad U_h^1(N)^S$$

$U_h^1(N)$ raised to the power S is the joint probability that all servers in group h are busy.

In table 3.10 the exact value of $P[Q \geq S]$ is compared with the Erlang C formula and the thesis estimate for a range of models. In the table, N is the number of customers sharing the multi-server, S is the number of servers at the multi-server, and $U_h^1(N)$ is the utilization of each of the S servers. Both estimates are good at very high utilizations and when the number of customers is at least twice as large as the number of servers. The Erlang C formula is always pessimistic, and the thesis estimate always optimistic. Since the thesis estimate is more accurate when the number of customers is close to the number of servers, of comparable accuracy elsewhere, and easier to compute than the Erlang C formula, it has been chosen for use in the analysis.

In the residence time expression, the probability that a customer arriving at server s will find that all servers in group h are busy is defined as:

$$B_s(N-1) \quad \leftarrow \quad U_h^1(N-1)^S \quad S > 1$$

In this closed model, the $N-1$ population vector is used because an arriving customer will only suffer a delay if all the servers are busy serving other customers (see the AIT on page 33). When the number of servers equals one, a FCFS server should be used.

The customer's residence time expression introduces into the analysis the fact that when a server is available it will not be necessary to suffer a queueing delay. Also, when all servers are busy the expected time to the next completion is assumed to be D/S. Thus the residence time equation for a customer of class g at entry 1 of multi-server s, $R_{1,g,s}(N)$, is:

$$R_{1,g,s}(N) \leftarrow D\,(1 + \frac{Q_{1,g,s}(N-1)}{S}\; B_s(N-1)).$$

The arrival instant queue length for a customer at a multi-server, $Q_{1,g,s}(N-1)$, includes customers receiving service and those queued for service. If there is a free server then the arriving customer will not be delayed.

Each of the class h servers has a residence time at entry 2 of the multi-server s, $R_{2,h,s}$, that is a function of its utilization. It is the time that a server in class h spends idle (that is, not serving a customer) which is the server's cycle time minus the service time.

$$R_{2,h,s}(S) \leftarrow D/U_h^1(N) - D.$$

The following equations summarize the multi-server:

$$
\begin{aligned}
B_s(N-1) &\leftarrow U_h^1(N-1)^S \quad S>1, \\
R_{1,g,s}(N) &\leftarrow D\,(1+\frac{Q_{1,g,s}(N-1)}{S}\,B_s(N-1)) \\
R_{2,h,s}(S) &\leftarrow D/U_h^1(N)-D
\end{aligned}
$$

In Linearizer, $B_s(N-1)$ is assigned in the core support routine. Also note that when evaluating $R_{1,g,s}(\vec{N})$, successive estimates for $R_{1,g,s}(N)$ are averaged to encourage convergence. This is done because small changes in $U_h^1(N-1)$ can cause large changes in $B_s(N-1)$ and hence $R_{1,g,s}(N)$.

$$
\begin{aligned}
prev &\leftarrow R_{1,g,s}(N) \\
R_{1,g,s}(N) &\leftarrow D\,(1+\frac{Q_{1,g,s}(N-1)}{S}\,B_s(N-1)) \\
R_{1,g,s}(N) &\leftarrow (R_{1,g,s}(N)+prev)/2
\end{aligned}
$$

3.3.2 Validating The Technique With Respect To Simulation

To test the accuracy of the multi-server residence time expression, the following parameters are considered:

- S the number of servers,
- N the number of customers sharing the servers,
- U_h the utilization of each server that provides service at the multi-server.

Note that there is one multi-server and several serving tasks that share a single queue of customers at the multi-server.

The software process architecture that is used is shown in figure 3.15. A set of models that correspond to the combinations of the following model parameters has been chosen:

- Number of servers $S = N_h \in \{1,2,3,4,5\}$,

- Number of customers $N = N_g \epsilon \{5, 7, 9, 12\}$,
- Server utilization of each of the servers $U_h \epsilon \{0.25, 0.5, 0.75, 0.98\}$,

Eighty models are considered.

When choosing the models, for each N_g and N_h combination, the time that customers spend at the CPU, $S_{cust,cpu}$, was systematically altered until instances of models were found with the desired multi-server server utilization U_h. The CPU is a DELAY device.

The purpose of the test cases is to study the error in estimated average customer response time based upon server utilization and the number of servers shared by the customers. Customer response times contain both demand at the DELAY CPU and time the multi-server. Low multi-server utilizations correspond to a large amount of time at the CPU, and hence high response times.

The errors in predicted average customer response time and server utilization for each of the test cases are presented in figures 3.16 and 3.17, respectively. The errors are relative errors with respect to simulation. They are shown on a grid. In figure 3.16, a contour diagram is imposed on top of the grid to show how changes in model parameters affect customer response times.

The multi-server is a first-come-first-served server. In the test cases, the coefficient of variation of service times is one. This makes the multi-server a separable queueing server [Chandy 77]. The results of the approximation are expected to be accurate.

From figure 3.16, there are only five models out of eighty with errors in predicted average customer response times greater than 5%, relative to simulation. The worst error is 7.1%. Errors are significant only at very high server utilization.

From figure 3.17, there are only two models with errors in predicted server utilization greater than 5%. The worst error is -7.0% and occurs when there are five customers sharing four fully utilized servers.

To conclude, the multi-server residence time expression does a good job at estimating customer response times and server utilizations.

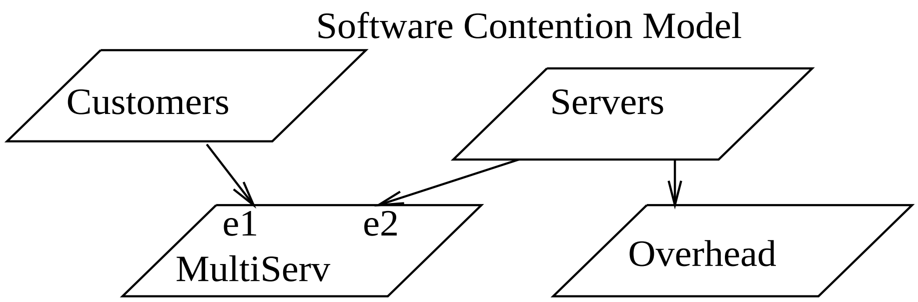


Figure 3.14: A Multi-Server with an Overhead Task.

N	S	$U_h^1(N)$	Exact $P[Q \geq S]$	Thesis $P[Q \geq S]$	Erlang C $P[Q \geq S]$
5	4	0.1	0.0003	0.0002	0.001
		0.3	0.012	0.007	0.033
		0.5	0.085	0.058	0.165
		0.7	0.29	0.24	0.42
		0.9	0.68	0.64	0.78
		0.98	0.89	0.88	0.93
10	4	0.1	0.0009	0.0002	0.0015
		0.3	0.03	0.01	0.05
		0.5	0.15	0.07	0.18
		0.7	0.39	0.26	0.45
		0.9	0.74	0.66	0.79
		0.98	0.93	0.91	0.95
10	7	0.1	0.0000	0.0000	0.0000
		0.3	0.001	0.0002	0.006
		0.5	0.03	0.008	0.08
		0.7	0.17	0.08	0.30
		0.9	0.59	0.48	0.72
		0.98	0.91	0.88	0.95
10	9	0.1	0.0000	0.0000	0.0000
		0.3	0.0001	0.0000	0.0023
		0.5	0.005	0.002	0.046
		0.7	0.07	0.04	0.24
		0.9	0.49	0.43	0.71
		0.98	0.83	0.81	0.92

Table 3.10: A comparison of $P[Q \geq S]$ approximations.

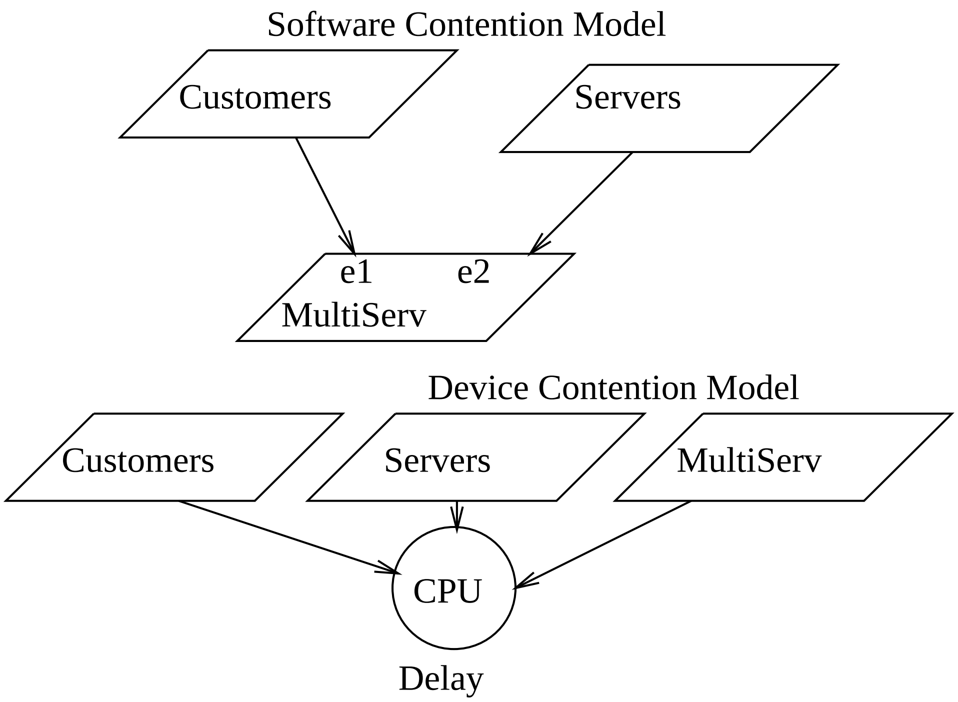


Figure 3.15: Multi-Server Test Case

Entity	Population	$V_{e_1,MultiServ}$	$V_{e_2,MultiServ}$	V_{cpu}	S_{cpu}
Customers	N_c	1	-	5	$S_{cust,cpu}$ chosen
-	-	-	-	-	to satisfy U_h
Servers	N_s	-	1	1	1
MultiServ	1	-	-	-	-

Table 3.11: a) Basic Model Parameters for the Multi-Server Test Case.

N_g	N_h	U_h	Customers $S_{cust,cpu}$
5	1	0.25	3.76
5	4	0.98	0.0
7	1	0.98	0.358
7	2	0.25	2.59
12	3	0.5	1.40
12	5	0.75	0.428

Table 3.11: b) Sample of Model Parameters for the Multi-Server Test Case.

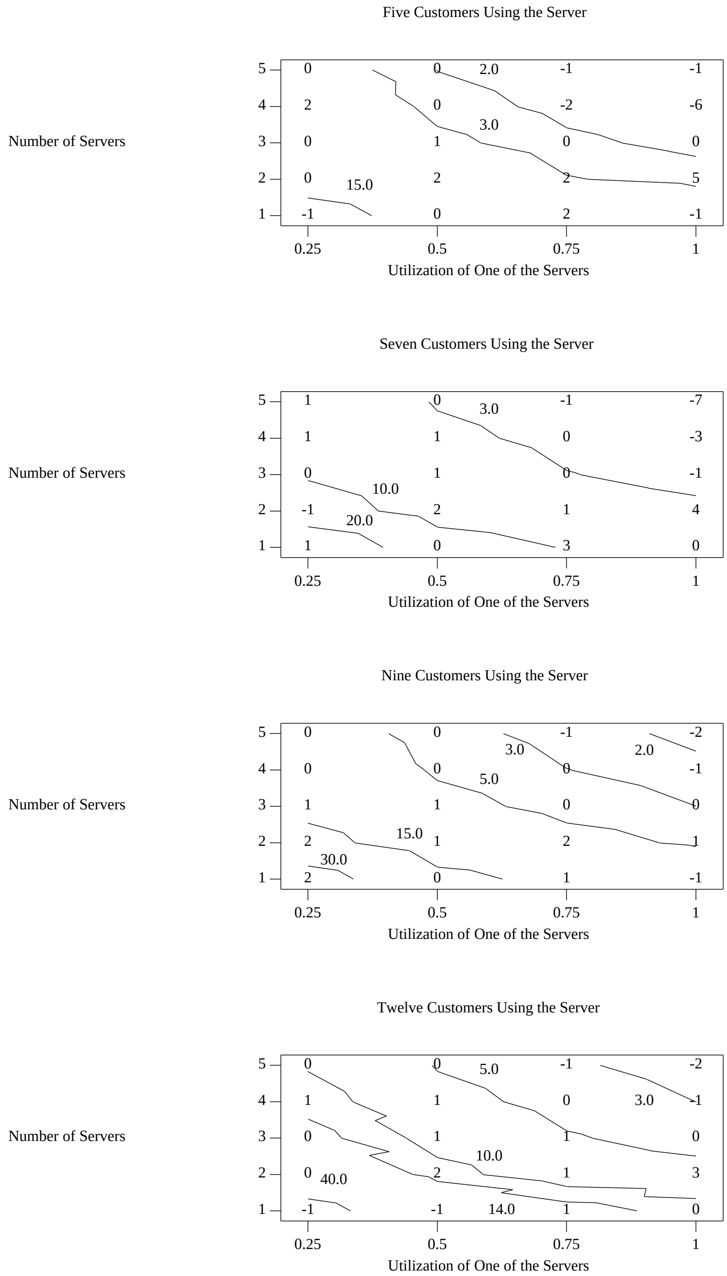


Figure 3.16: Multi-Server Test Case: Error in Estimated Customer Response Time, and Response Time Contours.

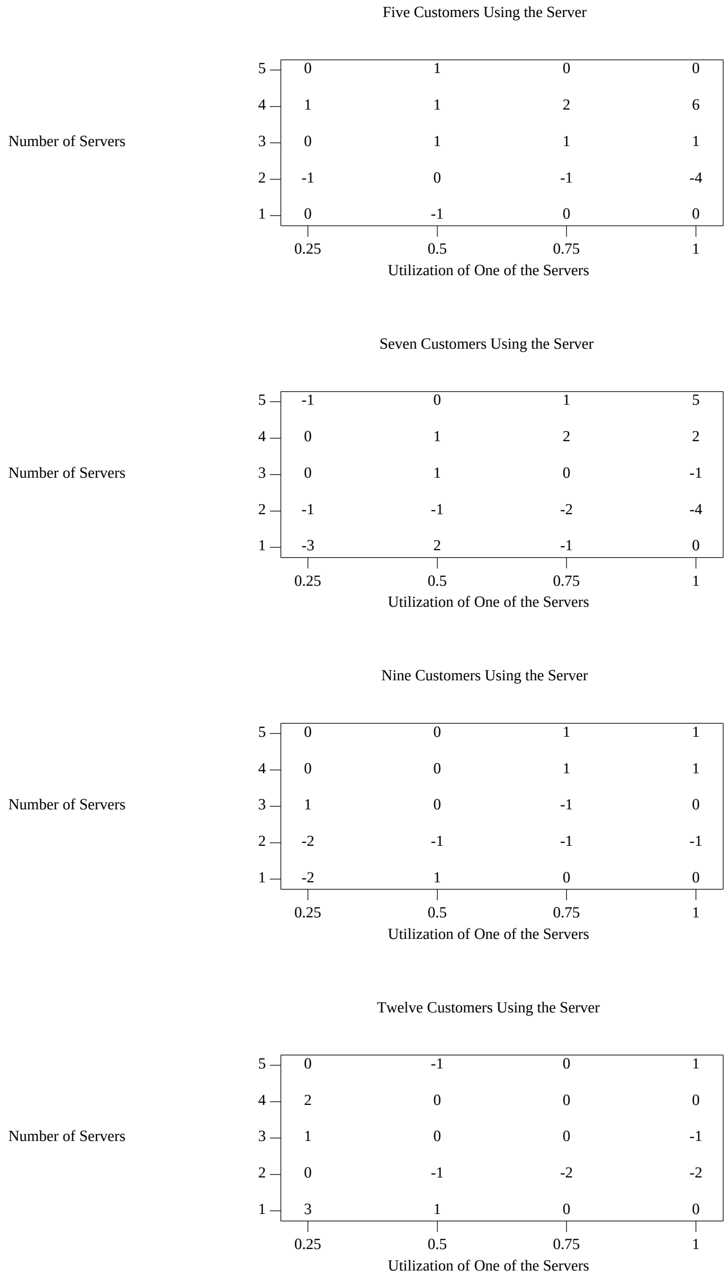


Figure 3.17: Multi-Server Test Case: Error in Estimated Multi-Server Server Utilization.

3.4 Conclusions

Mean value analysis techniques have been developed that can be used to model the rendezvous, a rendezvous with a multiple-entry server, and a call to a multi-server. Layered group models can be constructed that contain such behaviour and their performance can be predicted using the MOL.

The three servers can be used to create models of systems with simple client-server relationships. The multiple-entry server acts a software version of Reiser's HVFIFO server. The multi-server can be used to represent pools of agent tasks that provide service to callers in a first-come-first-served manner.

Chapter 4

Servers With Synchronization

In software systems, it may be necessary for processes to synchronize. For example in a factory, it may be the case that a pressure release valve should not be opened unless a safety system has been initiated and an operator has confirmed that the valve may be opened safely. The initiation of the safety system and the request for confirmation from an operator can proceed in parallel. However, the valve cannot be opened until both parallel activities are complete. The software processes that correspond to these activities must synchronize before the valve opening process can execute.

In this chapter, three new Mean Value Analysis modelling techniques are introduced. Two of the techniques are very similar and are presented in section 4.1. They are used to analyse synchronization between customer classes and are implemented in the MOL as new forms of service centers called SYNC and SYNCDEL. In section 4.2, a technique is introduced that uses two SYNC servers to predict the performance measures for Producer-Consumer systems that share a buffer with a finite number of elements.

In each of the sections, the behaviour that is being modelled is described, the method is presented, and the results of the analysis are validated with respect to simulation.

4.1 The SYNC and SYNCDEL Servers

The synchronization server is used to provide a synchronization point for processes in an LGM. It corresponds to nested accept statements in a software system. The server is implemented using multiple entries. When a customer process is available on each entry of the serving process, synchronization occurs. Let an instance of a set of customers that synchronize be called a **synchronization set**. The server description includes a single phase of service during which the customers remain synchronized. They receive service in unison. At the end of the phase, the customers are released from the server and are no longer synchronized. Post-rendezvous service at the synchronization server has not been considered. Also, a group that visits a synchronization server only visits one of the entries, and that entry identifies the calling group.

The synchronization servers have two factors that affect their performance: customers must synchronize, and then receive service at the server. The two types of servers discussed in this section differ in the way customers receive service. At **SYNC** servers, synchronized sets of customers queue for service in the same manner as with separable queueing centers. Only one set of customers can be in a service period at the server at a time. At **SYNCDEL** servers, any number of synchronized sets of customers can receive service at the same time. This is similar to the separable DELAY service center.

The residence time for a customer at an entry includes: queueing delays incurred because of customers ahead in the queue, synchronization time for customers to arrive at the other entries, and the service time at the server. In the analysis that is presented, it is assumed that the synchronization server has two entries. Generalizing the technique to permit more entries is a topic for future research.

SYNC servers have a population, N_g, of one. SYNCDEL servers have their populations determined by the maximum number of customer sets that can be synchronized at the server at any one time. This is defined as the minimum over all entries of the maximum number of customers that use the entry. A SYNC or SYNCDEL server can be referred to as a **server group** with population N_g when providing service.

SYNC and SYNCDEL servers can also act as customers in a software model hierarchy. A synchronization server g is **customer group** with population N_g when obtaining service. The server is permitted to use devices and other software servers. The processing done by a server is described using an entry with one phase of service. (See section 3.1.1 for a description of phases.) Thus, no additional parameters are required to inlcude synchronization servers in an LGM.

The analysis for the SYNC and SYNCDEL servers leads to residence time equations that differ in only one term so they are presented together.

4.1.1 Overview Of The Technique

In MVA modelling, a customer's residence time at a server is the sum of its own demand and any work that is ahead of the customer at its arrival instant at the server. The group server residence time expression must account for: delays due to service within the serving group for the arriving customer, synchronization time for the arriving customer, queueing delays for customers already in the queue that are waiting for synchronization, and queueing delays for customers in the queue that are receiving service.

$R_{e,g,h}(\vec{N}) =$ the arriving customer's synchronization delay +
the synchronization delays due to customers ahead in the queue +
the arriving customer's own service time and the service time
queueing delays due to customers ahead in the queue

By the construction of the residence time expression, customers arriving at each entry are served in a first-come-first-served manner at the entry.

The residence time equation for the SYNC and SYNCDEL servers is now given and is followed by an intuitive justification. Only one term, $\beta_{e,g,h}(\vec{N}-1_g)$, differs between residence time expressions for the two servers. The difference is due to the way that service time queueing is handled (queueing versus DELAY, respectively). $\Upsilon_{e,g,h}(\vec{N}-1_g)$ and $\gamma_{e,g,h}(\vec{N}-1_g)$ are terms used to predict synchronization delays. The residence time equation for a customer of group g visiting entry e of the synchronization server h is:

$$\begin{aligned} R_{e,g,h}(\vec{N}) = \; & \Upsilon_{e,g,h}(\vec{N}-1_g) + \\ & \gamma_{e,g,h}(\vec{N}-1_g) \;\; V_{e,g,h} \;\; Q_{e,g,h}(\vec{N}-1_g)\,(1-U_h(\vec{N})) \quad + \\ & D_{e,g,h} \;\; (1+Q_{e,g,h}(\vec{N}-1_g) \;\; \beta_{e,g,h}(\vec{N}-1_g)) \end{aligned}$$

where:

- $D_{e,g,h}$ is the average service demand of group g at entry e of server h.
- D_h the average service demand at synchronization server h. It is assumed that all groups that visit synchronization server h have the same average service demand, $D_{e,g,h}$ equals D_h for all g, e.
- $\Upsilon_{e,g,h}(\vec{N}-1_g)$ the average time required for a customer of group g arriving at entry e of server h to synchronize with a customer from each other entry.
- $\gamma_{e,g,h}(\vec{N}-1_g)$ for a customer of class g arriving at entry e of server h, this is the expected time it takes for a customer to arrive at each of server h's other entries.
- $V_{e,g,h}$ is the mean number of visits, per invocation, by a member of group g to entry e of server h.
- $Q_{e,g,h}(\vec{N}-1_g)$ is the arrival instant queue length of a customer of group g at entry e of server h.
- $U_h^1(\vec{N})$ is the utilization of server h. When h is a SYNCDEL server, it is the utilization of each of the servers of h.
- N_h is the population of serving group h. It is equal to one when the server is a SYNC server. For a SYNCDEL server it is the $min(N_{1,h}, N_{2,h})$, where $N_{1,h}$ and $N_{2,h}$ are the number of customers sharing entry 1 and entry 2 of server h, respectively.
- $\beta_{e,g,h}(\vec{N}-1_g)$ is the service rate speedup factor. It is necessary when N_h exceeds one. This is the only term that differs for the SYNC and SYNCDEL servers.

$D_{e,g,h}$, $V_{g,h}$, and N_h are input parameters to the model; $Q_{e,g,h}(\vec{N}-1_g)$ and $U_h(\vec{N})$ are intermediate results found using the residence time expression in conjunction with Linearizer. $\beta_{e,g,h}(\vec{N}-1_g)$ is a function of intermediate population vectors, and $\Upsilon_{e,g,h}(\vec{N}-1_g)$ and $\alpha_{e,g,h}(\vec{N}-1_g)$ are based upon intermediate response time and throughput estimates.

First, consider the term $\Upsilon_{e,g,h}(\vec{N}-1_g)$. When a customer arrives at entry e of synchronization server h, it must synchronize with a customer from each other entry. The term $\Upsilon_{e,g,h}(\vec{N}-1_g)$ is used to represent this in the residence time expression. It is defined as:

$$\Upsilon_{e,g,h}(\vec{N}-1_g) = \Gamma_{e,g,h}(\vec{N}-1_g)\ \ \alpha_{e,g,h}(\vec{N}-1_g).$$

At the customer's arrival instant, $\Gamma_{e,g,h}(\vec{N}-1_g)$ is the expected time before a customer arrives at each of the other entries. The technique used to predict $\Gamma_{e,g,h}(\vec{N}-1_g)$ assumes that there is at most one customer queued at each of the entries. However in this model, many customers can share an entry; it is possible for customers to queue behind other customers at an entry and for customers to belong to many synchronization sets. This affects the probability that it is actually necessary to wait for a customer to arrive at each of the other entries. $\alpha_{e,g,h}(\vec{N}-1_g)$ estimates the probability that it is necessary to wait for synchronization because customers are not yet available. When only one customer uses each of the other entries, $\alpha_{e,g,h}(\vec{N}-1_g)$ has value one. Thus $\alpha_{e,g,h}(\vec{N}-1_g)$ is used to condition $\Gamma_{e,g,h}(\vec{N}-1_g)$ to provide $\Upsilon_{e,g,h}(\vec{N}-1_g)$ for systems where more than one customer uses each entry.

Finally, when an arriving customer must queue behind other customers at the same entry, it must wait for each of the customers in its arrival instant queue to form a synchronization set. $\gamma_{e,g,h}(\vec{N}-1_g)$ represents the expected time before a customer in the entry e arrival instant queue synchronizes with customers from other entries. It is assumed that there are no customers present at the other entries at the arrival instant.

To determine the value of $\Gamma_{e,g,h}(\vec{N}-1_g)$, the intersynchronization time (the average time between successive synchronizations) of customers must be found. The synchronization times are predicted using the expression for the expected maximum of T exponentially distributed mutually independent variables [Heid 83]:

$$\begin{aligned} Max_T = \; & \textstyle\sum_{i=1}^{T} \frac{1}{\lambda_i} - \sum_{i<j} \frac{1}{\lambda_i+\lambda_j} + \sum_{i<j<k} \frac{1}{\lambda_i+\lambda_j+\lambda_k} \\ & \textstyle - \cdots + (-1)^{T-1} \sum_{i_1<i_2\cdots<i_T} \frac{1}{\lambda_{i_1}+\cdots+\lambda_{i_T}}, \end{aligned}$$

where $1/\lambda_i$ is the expected value of variable i. The case of $T=2$ is considered in this thesis, and is used for synchronization servers with two entries.

Let $\lambda_{1,g,h}(\vec{N})$ be the arrival rate of group g customers to entry 1 of server h, and $\lambda_{2,j,h}(\vec{N})$ be the arrival rate of group j customers to entry 2 of server h. Assuming that the interarrival times of customers using a synchronization server are exponentially distributed and mutually independent, the expected time between successive synchronizations at server h is [Triv 82]:

$$Max_2 = \; 1/\lambda_{1,g,h}(\vec{N}) + 1/\lambda_{2,j,h}(\vec{N}) - 1/(\lambda_{1,g,h}(\vec{N}) + \lambda_{2,j,h}(\vec{N})).$$

$\Gamma_{e,g,h}(\vec{N} - 1_g)$ is the arrival instant synchronization time of a customer arriving at entry e of server h. Consider the expressions for entry 1, $e = 1$. (The case of $e = 2$ is symmetrical.)

$$\begin{aligned} \Gamma_{1,g,h}(\vec{N}) = & \ 1/\lambda_{2,j,h}(\vec{N}) - 1/(\lambda_{1,g,h}(\vec{N}) + \lambda_{2,j,h}(\vec{N})) = Max_2 - 1/\lambda_{1,g,h}(\vec{N}) \\ \gamma_{1,g,h}(\vec{N}) = & \ 1/\lambda_{2,j,h}(\vec{N}). \end{aligned}$$

Note that if the interarrival time of a customer at entry 2 is small with respect to that of the customer at entry 1, the delay will be negligible.

The analysis for the Max_2 function assumes that its arrival time processes are not affected by the synchronization time. In this way it is an analysis for open systems. To compensate for this in the closed analysis of the MOL, the interarrival times at a synchronization server do not include the time spent receiving service and waiting for customers ahead in the queue to synchronize or obtain service.

The **arrival rate**, $\lambda_{e,g,h}(\vec{N})$, of a group g at entry e of server h is computed using response and residence time calculations found at intermediate population vectors within Linearizer. $R_g(\vec{N})$ is the average response time of a customer in group g, and Z_g is the average think time of a customer in group g.

$$\lambda_{e,g,h}(\vec{N}) = \frac{N_g V_{e,g,h}}{R_g(\vec{N}) + Z_g - R_{e,g,h}(\vec{N})},$$

where $R_g(\vec{N}) + Z_g$ is the **cycle time** of a group g customer. $R_{e,g,h}(\vec{N})$ is the residence time of the group g that uses entry e of the synchronization server h at the server. The $R_{e,g,h}(\vec{N})$ is removed because it includes queueing and service time delays that should not be included in the expected synchronization delay.

In general, the interarrival time distributions of customers will not have a squared coefficient of variation equal to one. Under such conditions, using the Max function will provide inaccurate results with respect to the exact expected maximum. Consider the following interarrival time information for synchronization server h:

$C_v^2(e1)$ The squared coefficient of variation of interarrival times of customers using entry one.

$C_v^2(e2)$ The squared coefficient of variation of interarrival times of customers using entry two.

Let C_v^2 refer to either $C_v^2(e1)$ and $C_v^2(e2)$. When $C_v^2 = 0$, the interarrival times are deterministic and the maximum will simply be the larger of the two interarrival times. In this case the computation of $\Gamma_{e,g,h}(\vec{N} - 1_g)$ will be pessimistic. When the $C_v^2 > 1$, $\Gamma_{e,g,h}(\vec{N} - 1_g)$ will be optimistic. This is the area of most concern since the errors are bounded when $C_v^2 < 1$.

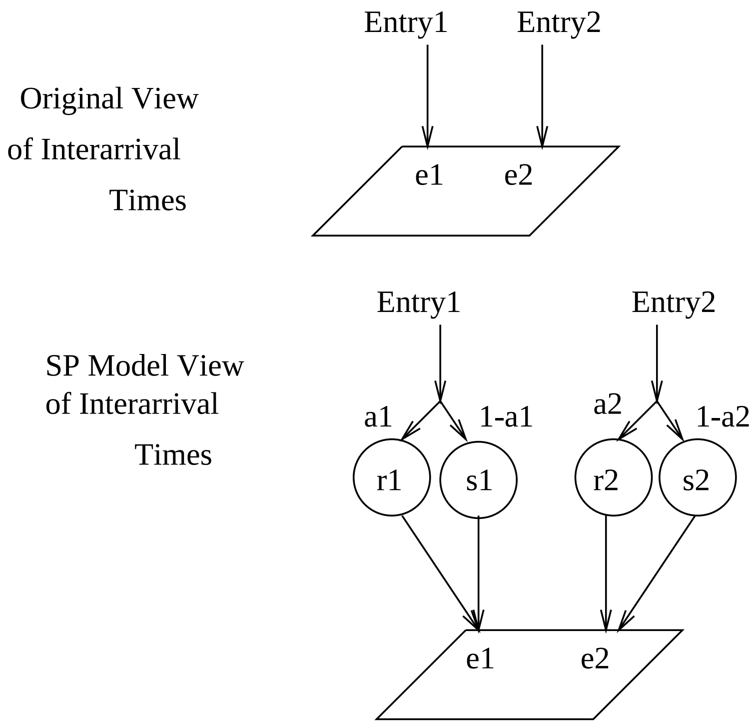


Figure 4.1: Mapping an Interarrival Time Distribution onto an Series Parallel Model.

A method is now given for finding the expected maximum value of random variables from two distributions with $C_v^2 \geq 1$. Each of the interarrival time distributions with $C_v^2 \geq 1$ can be replaced with a mixture of two interarrival time distributions, which are represented using two parallel stages, each with $C_v^2 = 1$, where the combination of the stage service times has the desired average interarrival time. The mean service times at the stages and the probability of selecting each stage can be found using the procedure for the construction of an SP model as described in section 3.2.1.

In figure 4.1, the two interarrival time distributions with $C_v^2 > 1$, that correspond to the synchronization server entries, are partitioned into four interarrival time distributions with $C_v^2 = 1$. The values, a_i, r_i, and s_i are based upon the means and variances of the interarrival times at the respective entries. The expected maximum for the original two distributions is simply the weighted sum of the expected maximums for the different stage combinations. The expected maximum for the original two distributions shown in figure 4.1 can be computed as follows:

$$\begin{aligned} Max_4 = \; & a_1 \; a_2 \; Max_2(r_1, r_2) + \\ & a_1 \; (1 - a_2) \; Max_2(r_1, s_2) + \\ & (1 - a_1) \; a_2 \; Max_2(s_1, r_2) + \\ & (1 - a_1) \; (1 - a_2) \; Max_2(s_1, s_2), \end{aligned}$$

Where

$$Max_2(x, y) = x + y - 1/(1/x + 1/y).$$

The function Max_4 is used to provide a better estimate for the value of $\Gamma_{1,g,h}(\vec{N})$,

$$\Gamma_{1,g,h}(\vec{N}) = Max_4 - 1/\lambda_{1,g,h}(\vec{N}).$$

Computing the interarrival time squared coefficient of variation of customers at an entry is similar to computing the response time squared coefficient of variation of the customers as is done in section 3.2.1. The only difference is that the value q, which corresponds to the probability that a phase's processing completes after a visit to the central server, should be replaced with the probability of visiting the synchronization server after visiting the central server. The probability of leaving the phase should be used to visit a delay server for the duration of the time that the customer spends not competing for the synchronization server. This delay time is an idle time for the group. Its squared coefficient of variation has be approximated as 1. The distribution of a group's idle times affects its arrival time squared coefficient of variation. Estimating the squared coefficient of variation for idle times is a topic for future research.

Now consider the value of the probability, $\alpha_{1,g,h}(\vec{N}-1_g)$, that a customer of entry 1 has to wait to synchronize with a customer of entry 2. When the number of customers using entry 2 is greater than one, the entry 2 customers can incur queueing delays at their entry of the synchronization server. The queueing delay is a waiting time suffered because there are other customers ahead of the arriving customer in its own entry's queue. To account for this behaviour, the $\Gamma_{e,g,h}(\vec{N}-1_g)$ value is conditioned with the probability that it is necessary to synchronize. $\alpha_{2,j,h}(\vec{N})$ is defined symmetrically.

The probability that a customer of class g using entry e of server h is waiting for synchronization, δ_e, is estimated as the fraction of time that a customer spends waiting for service at entry e:

$$\delta_e = (R_{e,g,h}(\vec{N}) - D_h)/(R_g(\vec{N}) + Z_g).$$

The probability that a customer of class g arriving at entry 1 will suffer a synchronization delay while waiting for an entry 2 customer is defined as $\alpha_{1,g,h}(\vec{N}-1_g)$. It is the percentage of time that no entry 2 callers are waiting for synchronization. With the assumption that entry 2 customers behave independently, it is estimated as follows:

$$\alpha_{1,g,h}(\vec{N}) \;=\; (1-\delta_2)^{N_{2,h}-1}.$$

$N_{2,h}-1$ is used because one of the entry 2 customers is already considered to be in the entry 1 customer's synchronization set and is considered in $\Gamma_{e,g,h}(\vec{N}-1_g)$. If there is only one customer using entry 2, $\alpha_{1,g,h}(\vec{N}-1_g)$ is defined as one.

Now consider the queueing delays for an arriving customer that are caused because the customers in the arrival instant queue, $Q_{e,g,h}(\vec{N}-1_g)$, must synchronize. The expected time between customer arrivals at the other entry is $\gamma_{e,g,h}(\vec{N}-1_g)$. However, only a portion of customers will not synchronize by the time they are ready to receive their D_h units of service from the server. The portion is estimated as $(1 - U_h^1(\vec{N}))$. When the server utilization is low, there is not likely to be much overlap in queueing delays for service and synchronization delays. When the utilization is high, there is likely to be a great deal of overlap.

For the SYNC server, only one customer can receive service at a time so $\beta_{e,g,h}(\vec{N} - 1_g) = 1$. For the SYNCDEL server, many sets of customers can receive service at the same time. $\beta_{e,g,h}(\vec{N} - 1_g)$ is a service rate speedup factor that represents an increase in the service rate seen by customers of entry e because more than one set of customers can receive service at the same time. Consider the case where there are N_1 customers using entry 1 and N_2 customers using entry 2 of server k. If $N_1 \leq N_2$ then $N_h = N_1$ and customers on entry 1 will never be delayed because the server h is busy. Hence $\beta_{1,g,h}(\vec{N} - 1_g) = 0$. On the other hand if $N_1 > N_2$ then $N_h = N_2$ and the average time between completions at the server is D_h/N_2 hence $\beta_{1,g,h}(\vec{N} - 1_g) = \frac{\alpha_{1,g,h}(\vec{N}-1_g)}{N_2}$. The $\alpha_{1,g,h}(\vec{N} - 1_g)$ term is included because the requesting customer can start service as soon as one of the N_2 customers from the other entry is available.

Finally, it is assumed that the service requirement distribution at each entry is the same. Since a customer from each entry is required before service can begin, the throughputs at each entry are identical. The utilization of the server is:

$$U_h^1(\vec{N}) = \frac{X_{e,g,h}(\vec{N})\, S_h}{N_h},$$

where $X_{e,g,h}(\vec{N})$ is the throughput at one of the entries, and S_h is the service requirement per visit at the server. Only one entry is considered because the customers from each entry receive service in unison.

Permitting more entries at the server is a topic for future research. The following aspects of the analysis are affected. The $\Gamma_{e,g,h}(\vec{N} - 1_g)$ term should be computed with a Max_n function where n is greater than four. The $\gamma_{e,g,h}(\vec{N} - 1_g)$ term should be based on a Max_n function when the number of entries is greater than one. The $\alpha_{e,g,h}(\vec{N} - 1_g)$ term must be based on the δ_f values for all other entries f.

In Linearizer, the terms $\Gamma_{e,g,h}(\vec{N}-1_g)$, $\alpha_{e,g,h}(\vec{N}-1_g)$, $\gamma_{e,g,h}(\vec{N}-1_g)$, $\beta_{e,g,h}(\vec{N}-1_g)$, and the interarrival time $C_v^2(e)$ are assigned in core support. The value of N_h is assigned in the initialize routine. For a synchronization server s, the values for $F_{1,c,d,s}(\vec{N})$ and $F_{2,c,d,s}(\vec{N})$ are assigned to zero in Linearizer because the assumption that $F_{e,c,d,s}(\vec{N})$ equals $F_{e,c,d,s}(\vec{N}-1_l)$ for all l is not valid. Consider the case where l is a customer from the other entry of the synchronization server. The removal of the class l customer can increase the fraction of class c customers at server s significantly because of higher average synchronization times. But if it is some other customer, the change in the fraction is not likely to be as significant.

4.1.2 Validating The Technique With Respect To Simulation

The accuracy of the synchronization server approximation is considered in this section. The SYNC server receives the majority of the attention in the study presented in this section. However, one third of the test cases that are presented in this section apply to the SYNCDEL server as well. The SYNCDEL server is considered in a further twenty eight models in chapter 5.

To test the accuracy of the synchronization server residence time expression, the following parameters are considered:

- the number of customers sharing each entry (N_{e1}, N_{e2}),
- the ratio of average interarrival times for the entries I_{e1}/I_{e2}, where $I_{e1} \geq I_{e2}$,
- the ratio of the service demand at the server to the largest average interarrival time for the entries D_h/I_{e1}, where D_h is the demand at the synchronization server h and I_{e1} is the average interarrival time of customers to entry $e1$,
- the coefficient of variation of interarrival times.

The software process architecture that is used is shown in figure 4.2. A set of models that correspond to all combinations of the following model parameters has been chosen:

- Customer populations $\{(N_{e1}, N_{e2})\} \,\epsilon\, \{(1,1),(3,3),(5,5)\}$,

- Interarrival time ratio for the entries $I_{e1}/I_{e2} \epsilon \{1, 2, 5, 10\}$,
- Ratio of service demand at the server to the larger average interarrival time between entries $D_h/I_{e1} \epsilon \{0.0, 0.1, 0.5, 1.0, 2.0\}$,
- The C_v^2 at the CPU1, $C_v^2(CPU1) \epsilon \{0.5, 1.0, 15.0\}$. The squared coefficient of variation at $CPU2$ is always 1. The $C_v^2(CPU1)$ cause interarrival time C_v^2 of approximately $\{0.5, 1, 5\}$, respectively.

One hundred an eighty models are considered. The test cases where $\{(N_{e1}, N_{e2}) = (1, 1)\}$ apply to the SYNCDEL server as well. The test cases in chapter 5 consider cases where N_{e1} and N_{e2} are greater than one.

In table 4.1a) the parameters for the synchronization server test cases are given. The populations, $S_{cust,cpu1}$, and $C_v^2(CPU1)$ have been chosen to obtain the set of 180 test cases. Some sample values for test cases are shown in table 4.1b).

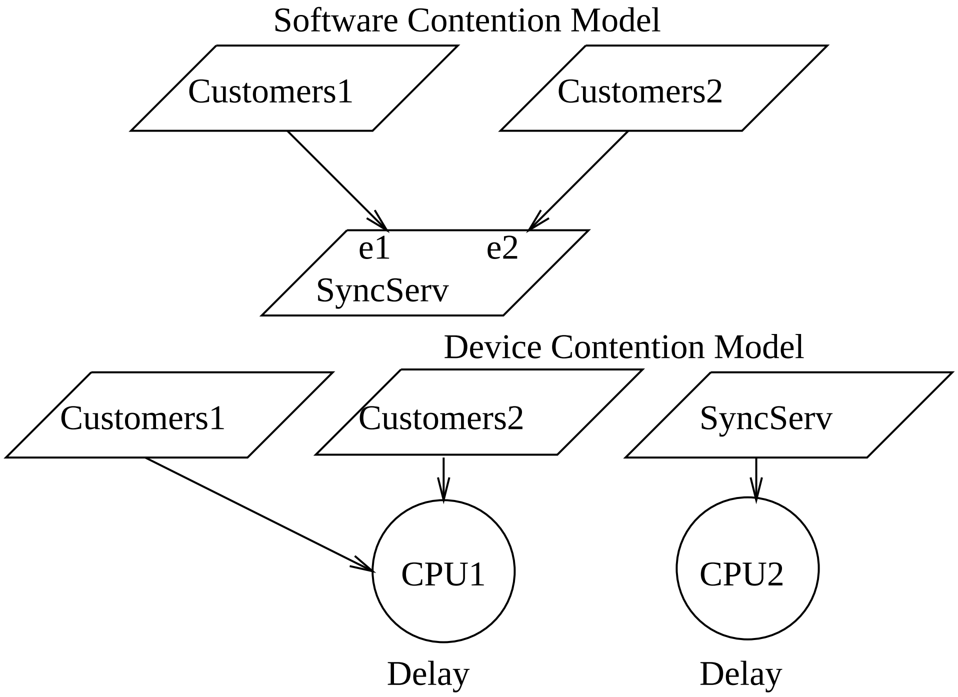


Figure 4.2: Synchronization Server Test Case

Entity	Population	Visits to SyncServ	V_{cpu1}	S_{cpu1}	V_{cpu2}	S_{cpu2}
Customers_{e1}	N_{e1}	1	5	$S_{cust,cpu1}$ varies to	-	-
-	-	-	-	satisfy I_{e1}/I_{e2}	-	-
Customers_{e2}	N_{e2}	1	5	1	-	-
SyncServ	1	-	-	-	1	$D_h/I_{e1}\ S_{cust,cpu1}$

Table 4.1: a) Basic Model Parameters for the Synchronization Server Test Cases.

N_{e1} and N_{e2}	I_{e1}/I_{e2}	D_h/I_{e1}	$S_{cust,cpu}$	$C_v^2(CPU1)$
1	1	0.0	1.0	0.5
1	1	2.0	1.0	1.0
1	2	0.0	2.0	15.0
1	10	0.0	10.0	0.5
1	10	0.0	10.0	15.0

Table 4.1: b) Sample of Model Parameters for the Synchronization Server Test Case.

The errors in predicted average customer response time and server utilization for each of the test cases are presented in figures 4.3 through 4.8 and figures 4.9 through 4.11, respectively. The errors are calculated relative to simulation. They are shown on a grid. In figures 4.3 through 4.8, a contour diagram of estimated average customer response times is imposed on top of the grid to show how changes in model parameters affect customer response times.

From figures 4.3 through 4.11, the squared coefficient of variation of interarrival times is not significantly affecting the accuracy of the technique. The grids of errors are almost exactly the same for each $C_v^2(CPU1)$ that is considered.

The cases where N_{e1} and N_{e2} equal one apply to the both the SYNC and SYNCDEL servers. The worst errors are when I_{e1}/I_{e2} equals one and D_h/I_{e1} equals two. The worst error is 6%. This is the case where the interarrival times are equal, and they are only half of the the service time at the server.

The ratio of server demand to the larger interarrival time (D_h/I_{e1}) affects the server utilization and has more of an affect on the accuracy of the residence time estimates than does the ratio of interarrival times at entries (I_{e1}/I_{e2}). In the set of models considered, the synchronization server has been examined with utilizations ranging from 0% to 100%. The largest errors arise when the utilization of the synchronization server is near 75%. For these models, the greater the difference in interarrival times, the lower the error in response time estimates.

For the remaining SYNC test cases, the error in predicted average customer response time is highest at $\{(5,5),10,0.5\}$ and $\{(5,5),1,0.5\}$ and is 9%. In these cases there are five customers sharing an entry and the $C_v^2(CPU1) = 0.5$. When there are fewer customers using an entry, the errors are lower. The major cause of the error is most likely an inaccuracy in the computation of the overlap of synchronization queueing delays and service queueing delays in the residence time expression. As the queueing delays due to service times at the synchronization server increase, the overlap between synchronization delays and service time queueing delays should also increase. The relationship need not be linear with the utilization of the server. However in the residence time expression, the $1 - U_h^1(\vec{N})$ term used to estimate the overlap is a linear function of the utilization. This is expected to introduce error into the estimated residence times.

To conclude, improved estimates for the overlap in synchronization and service queueing delays should improve the accuracy of the SYNC server technique. The technique appears to be robust enough to handle a wide range of interarrival time distributions for both the SYNC and SYCDEL servers.

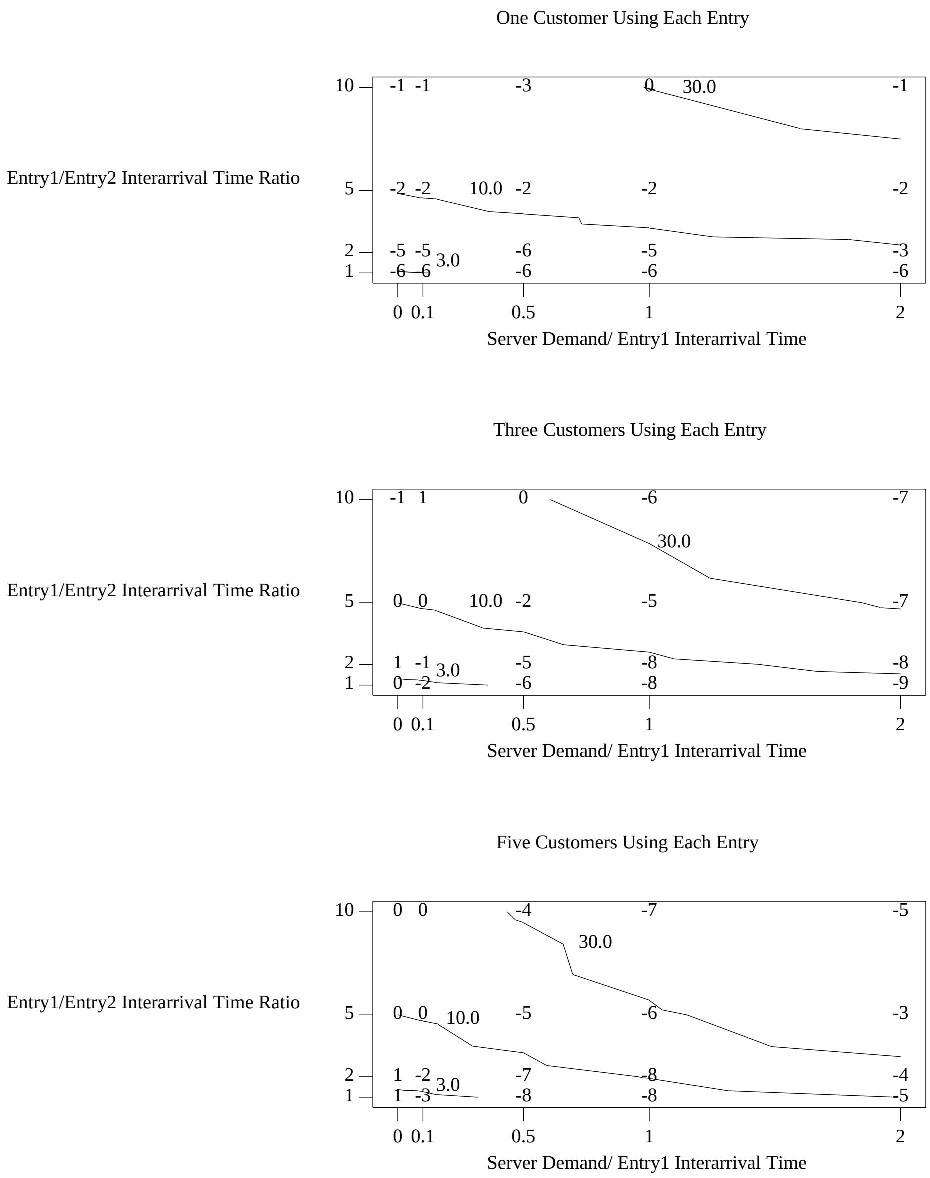


Figure 4.3: Synchronization Server Percentage Error in Predicted Entry 1 Customer Response Times and Customer Response Time Contours. Interarrival Time $C_v^2 = 0.5$.

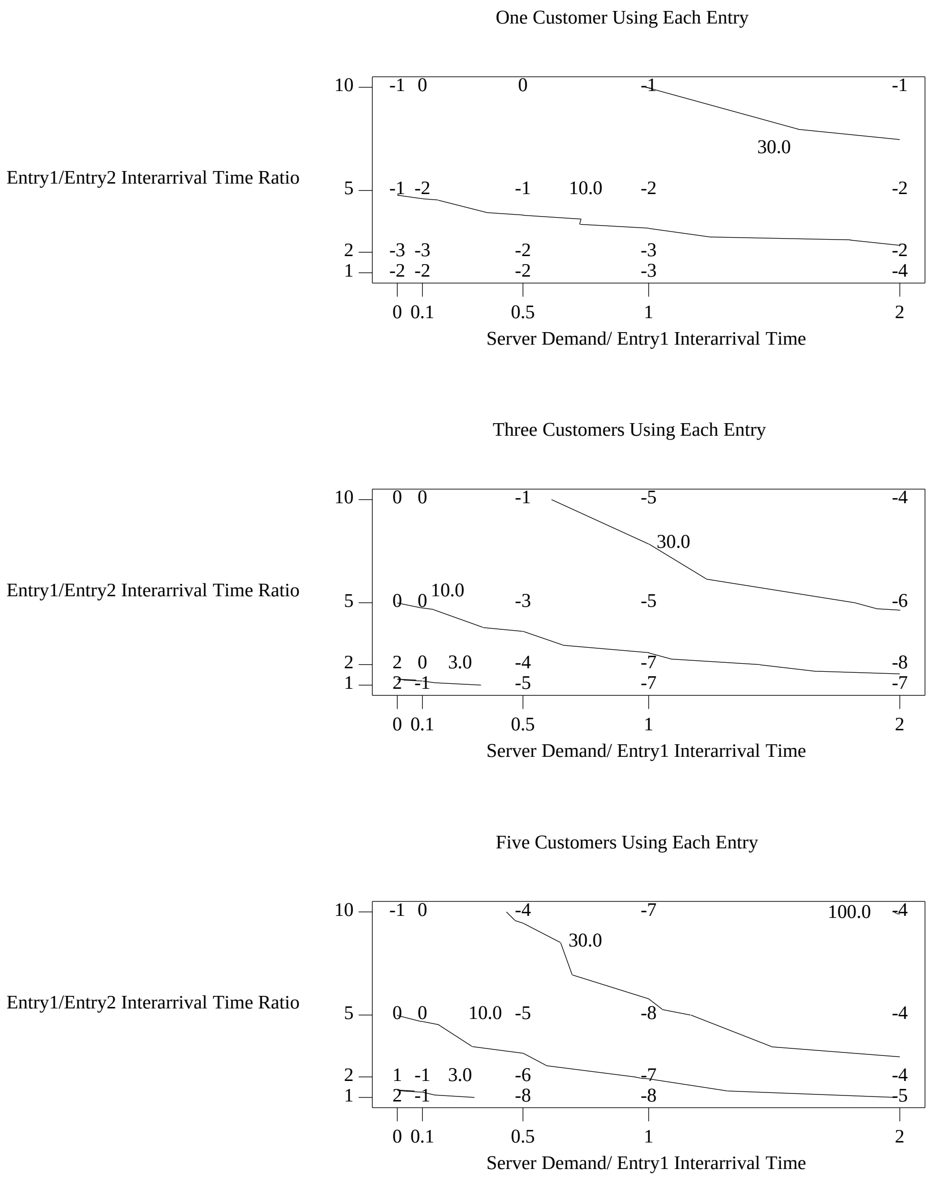


Figure 4.4: Synchronization Server Percentage Error in Predicted Entry 1 Customer Response Times and Customer Response Time Contours. Interarrival Time $C_v^2 = 1$.

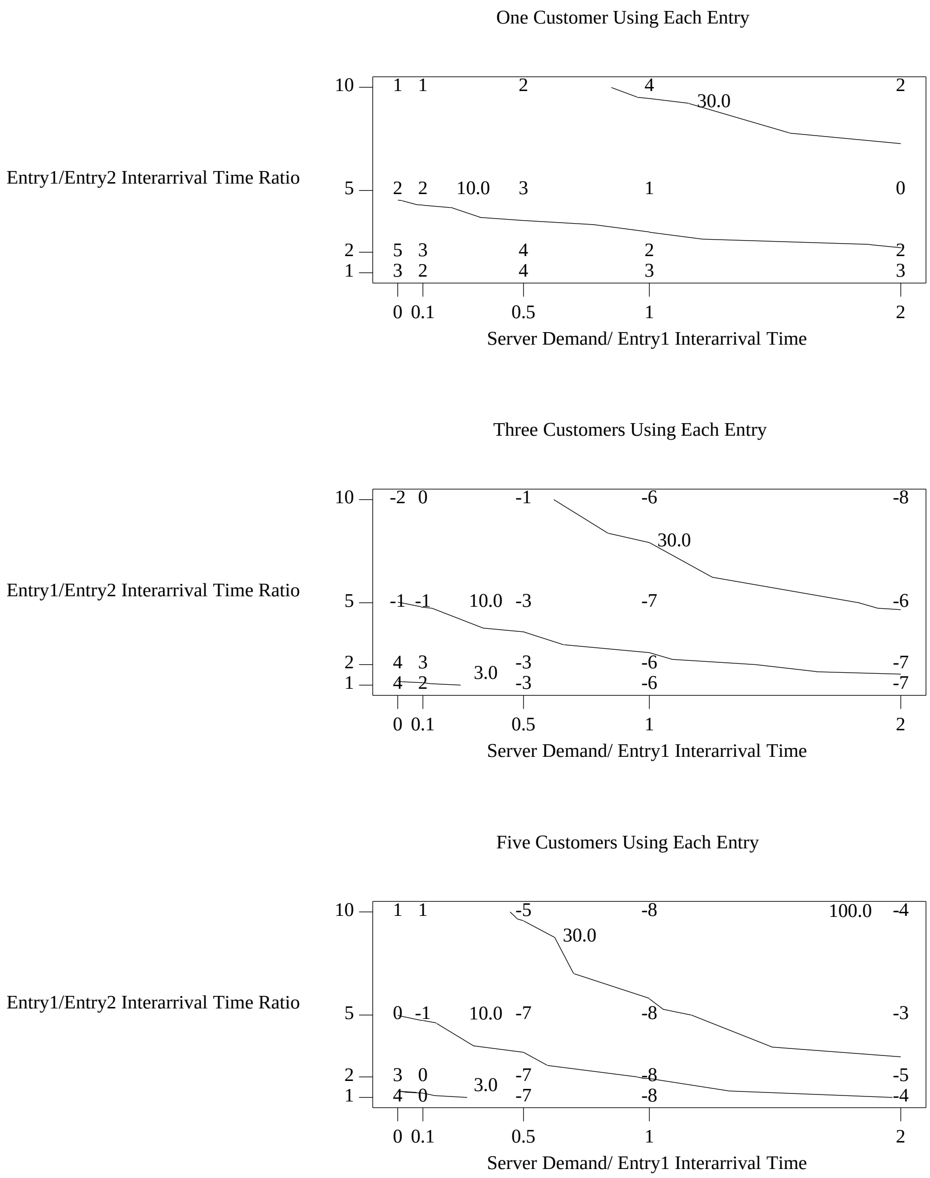


Figure 4.5: Synchronization Server Percentage Error in Predicted Entry 1 Customer Response Times and Customer Response Time Contours. Interarrival Time $C_v^2 = 5$.

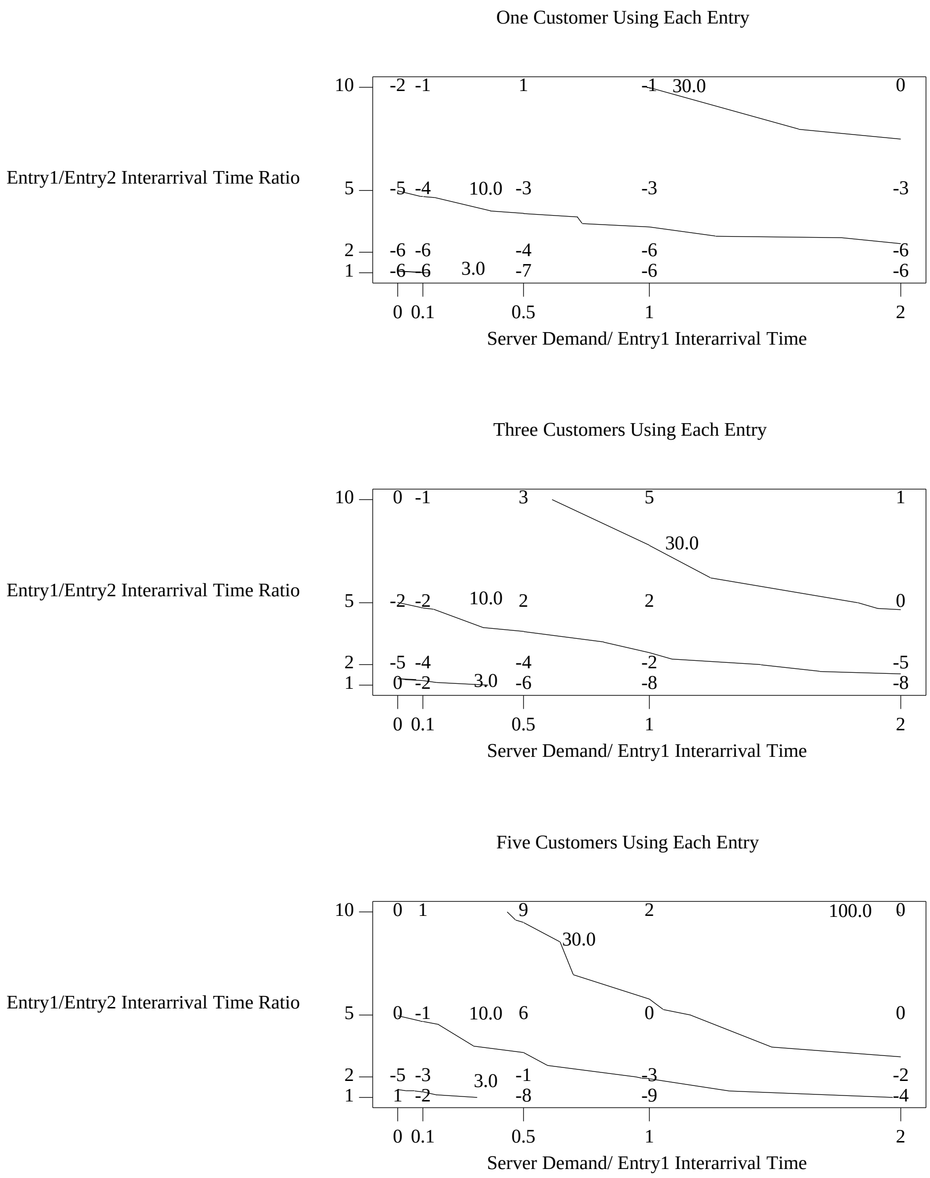


Figure 4.6: Synchronization Server Percentage Error in Predicted Entry 2 Customer Response Times and Customer Response Time Contours. Interarrival Time $C_v^2 = 0.5$.

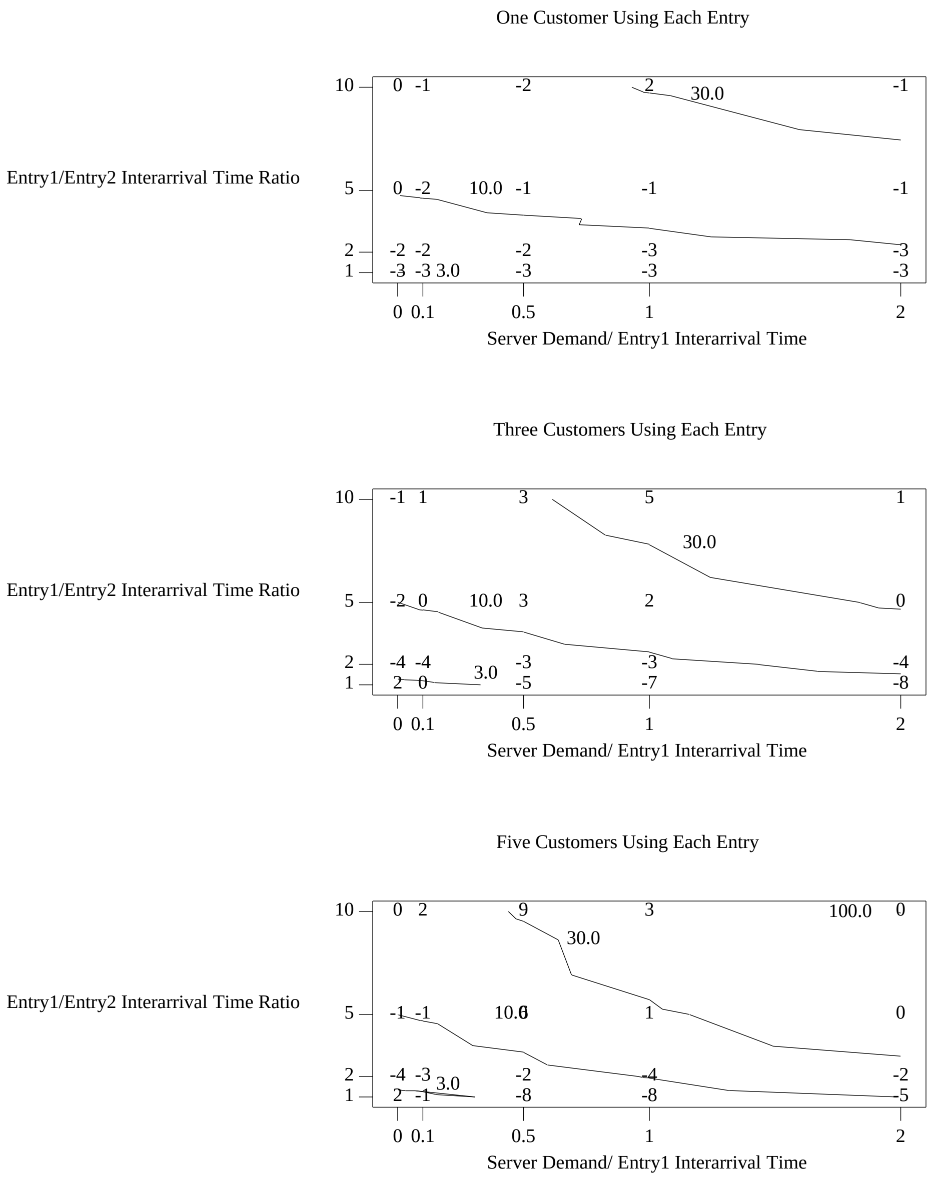
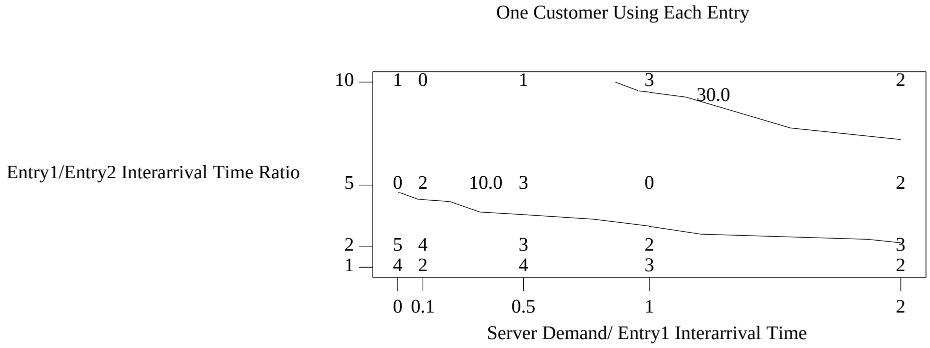
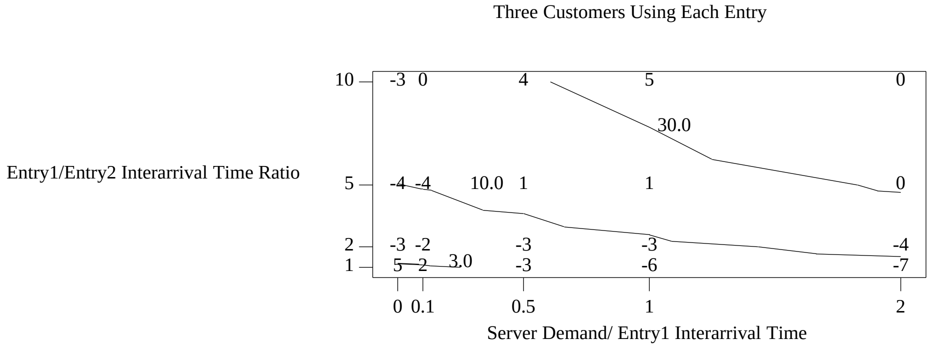


Figure 4.7: Synchronization Server Percentage Error in Predicted Entry 2 Customer Response Times and Customer Response Time Contours. Interarrival Time $C_v^2 = 1$.





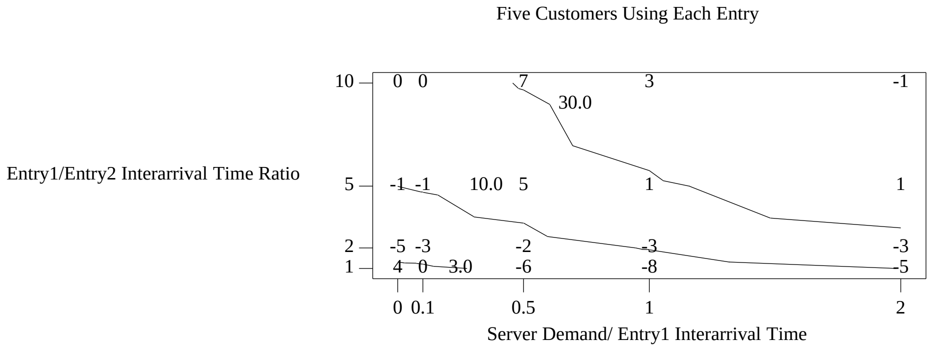


Figure 4.8: Synchronization Server Percentage Error in Predicted Entry 2 Customer Response Times and Customer Response Time Contours. Interarrival Time $C_v^2 = 5$.

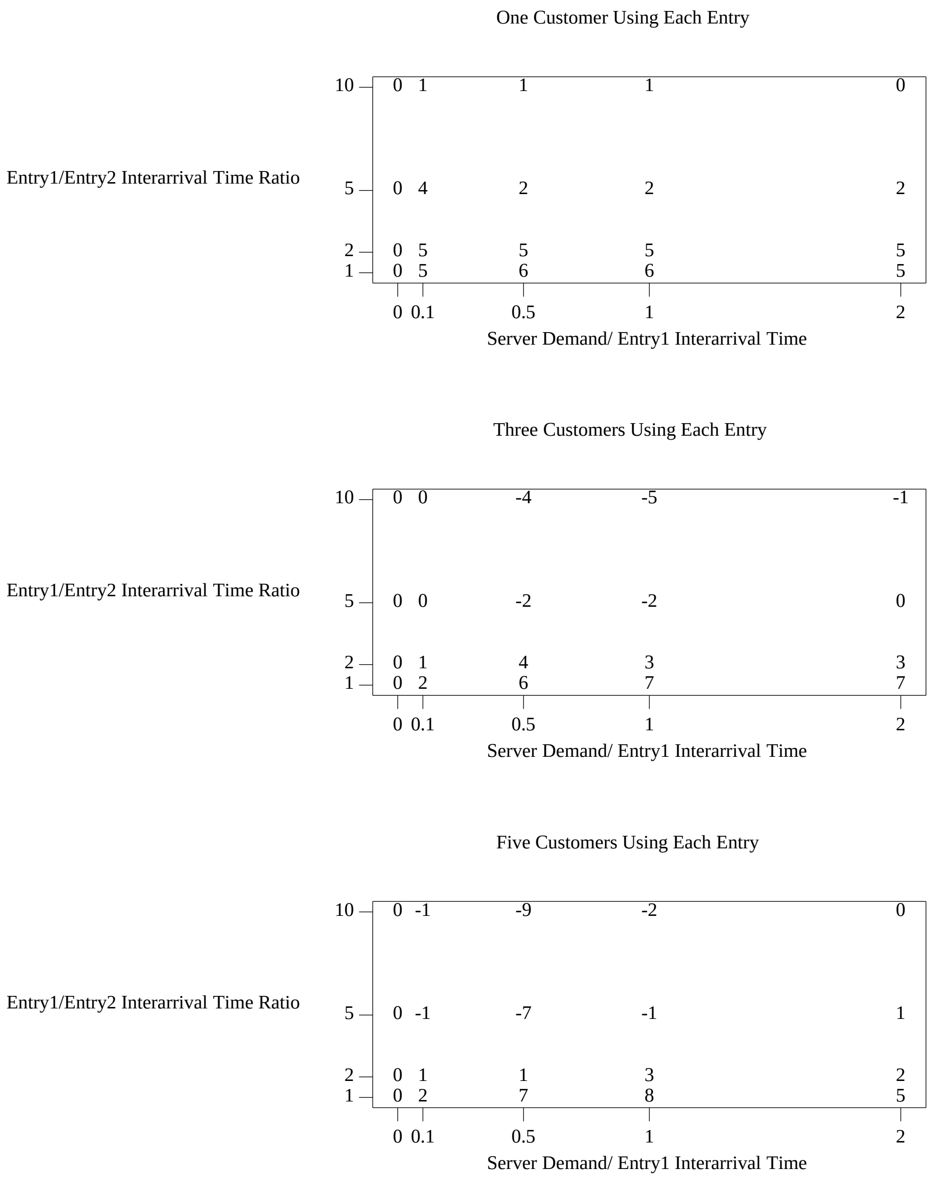


Figure 4.9: Synchronization Server Percentage Error in Predicted Server Utilization. Interarrival Time $C_v^2 = 0.5$.

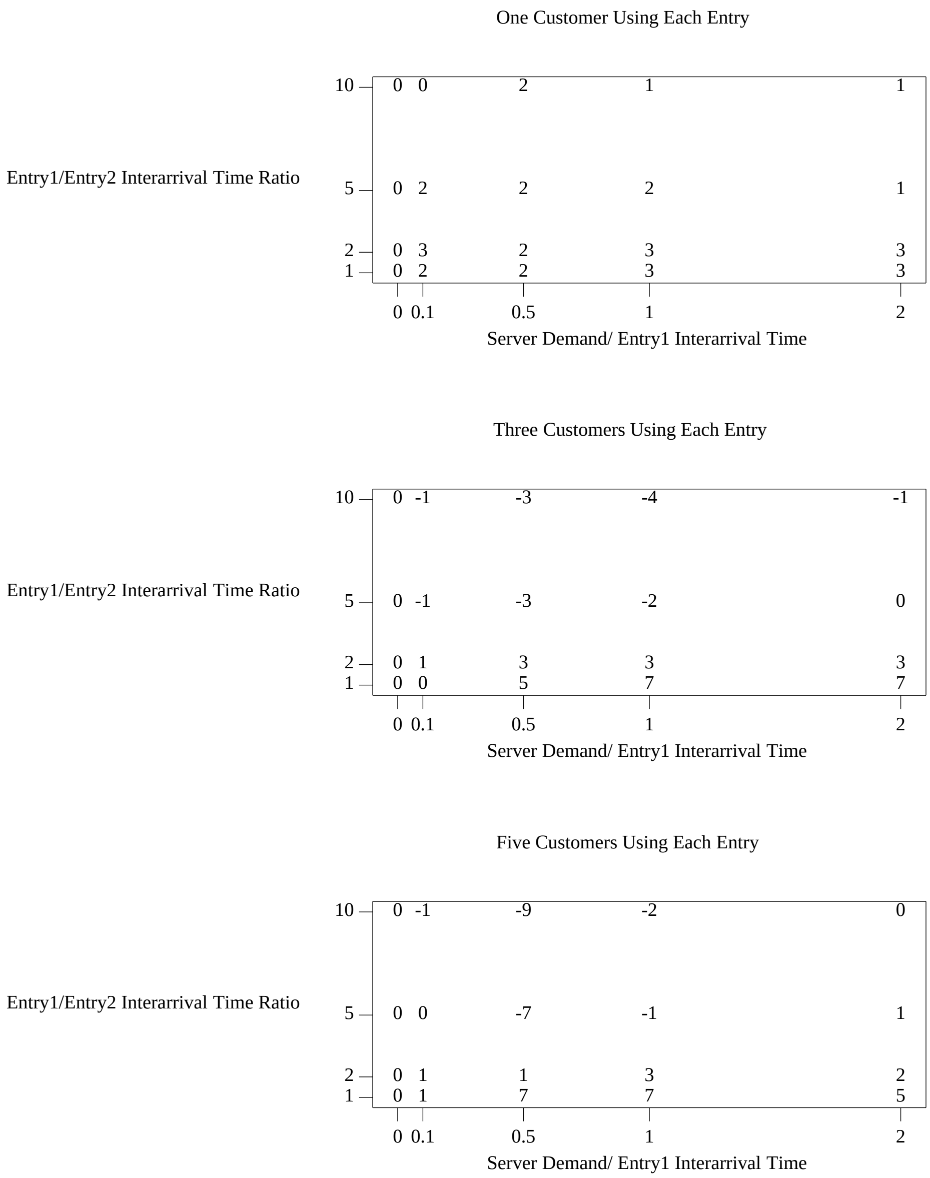


Figure 4.10: Synchronization Server Percentage Error in Predicted Server Utilization. Interarrival Time $C_v^2 = 1$.

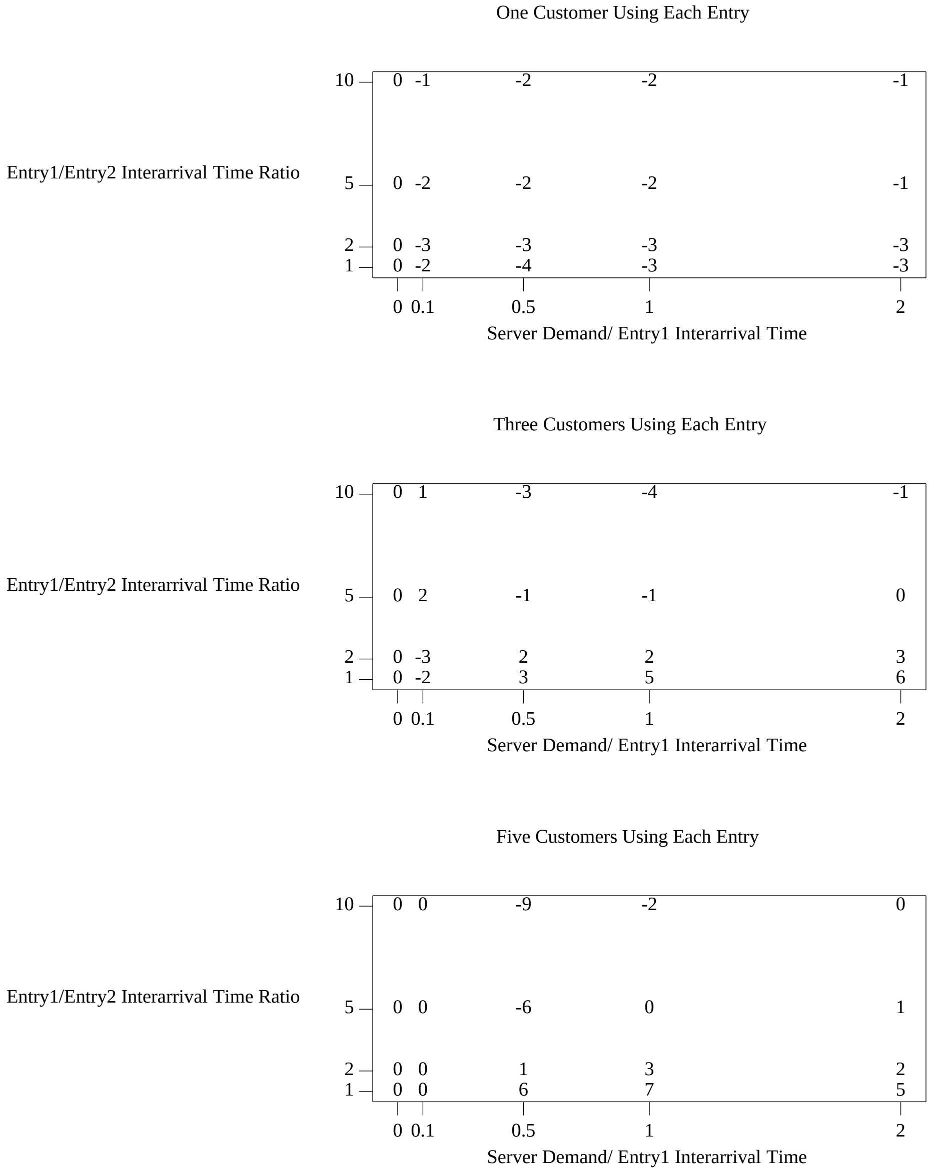


Figure 4.11: Synchronization Server Percentage Error in Predicted Server Utilization. Interarrival Time $C_v^2 = 5$.

4.2 The Producer-Consumer Server

Producer-consumer relationships exist in many software process architectures. The **producer-consumer problem** is also known as the **bounded buffer** problem. Two classes of customers share a common fixed-size buffer. One class, the producer class, puts information into the buffer, and the other one, the consumer class, takes it out. If a producer wants to put something in the buffer but it is already full the producer must wait. Similarly, a consumer must wait if no item of information is present in the buffer. Producer-consumer relationships are discussed in various operating system text books [Holt 78, Tanen 87].

The producer-consumer server can be used to predict the waiting times of producers and consumers at a buffer with a fixed number of elements. There can be several producer-consumer servers managing different buffer pools in the same model. Some of the parameters that affect performance are: the number of buffers, the buffer element holding time descriptions for the producer and consumer classes, and the populations of producer and consumer classes. The variance of interarrival times for customers has not been considered and is a topic for future research. Though producers and consumers can differ, customers of each class are assumed to be statistically identical.

Operating systems provide classical examples of producer-consumer relationships. Holt et al. [Holt 78] describe an Disk Input/Output system in which two processes, a paging process and a disk process, communicate with one another through a buffer manager. The buffer manager is a producer-consumer server. Producers, consumers, and buffer managers can be represented in LGMs as groups. Since the producer and consumer groups of processes require entries for requesting and releasing buffer elements, the buffer manager has four entries. The elements are of PAGE-SIZE words and are used to communicate information between virtual memory and a storage device.

Each of the processes can advance as long as an appropriate buffer element is available. The paging process moves pages of data to and from the buffer that are read and written to disk by the disk process. If the paging process operates quickly it can use up all of the elements before the disk process is able to consume them. In this case, the paging process must wait until an element is available. Similarly, the disk process is idle unless it is necessary for a disk operation to read or write into an available element.

In the following sections a technique is presented for modelling producer-consumer relationships. The technique is validated with respect to simulation to demonstrate its accuracy.

4.2.1 Overview Of The Technique

The producer-consumer server is based upon the SYNC server. Instead of using a new residence time equation, the producer-consumer server is implemented as a combination of two SYNC servers and a **buffer pool group** of customers in which the population of the group is set equal to the number of buffer elements. Each of the customers represents a buffer element that is to be shared between the producer and consumer groups (see figure 4.12).

A producer obtains an empty buffer element, performs a specified average number of visits to other servers and then returns the full element to the pool. The sum of the service times for the visits is defined as the time that the buffer element is held. After this time, the buffer element is then considered to be full. A consumer obtains a full buffer element, performs a specified average number of visits to other servers and then returns the empty element to the pool. Similarly, the sum represents the consumer's buffer element holding time. The buffer element is then considered to be empty. Again, producers and consumers become blocked when they require elements that are not available.

The sum of average holding times incurred by an element while being held by the producers and consumers is used to define a think time for the buffer pool group. Thus a customer of the buffer pool group thinks for a time that corresponds to the total time an element is held, it then waits to synchronize with a producer or alternatively a consumer. If customers only hold a buffer element for a short time, for instance to copy data, the think time will always be low.

The analytical technique does not force strict alternation of a buffer between producers and consumers. When considering the accuracy of the technique with respect to simulation, the simulator does enforce a strict alternation of buffers between producers and consumers. Hence the accuracy of the technique is validated with respect to the intended behaviour. As is shown in the next section, this loss of information does not cause significant errors in the estimates of average group response times for the wide range of model parameters that have been considered.

The producer-consumer server is two SYNC servers that manage the relationship between three groups:

- The producer group;
- The buffer pool group that represent elements in the buffer;
- The consumer group.

The following notation describes the producer-consumer server.

N_{buffer} The number of elements in the buffer pool. (N_{buffer} is an input parameter for the server.)

H_{prod} The average buffer element hold time of a producer using a buffer element from a producer-consumer server. There can be different buffer hold times for different producer-consumer servers.

$$H_{prod} = \sum_{k=1}^{K} V^{'}_{prod,k} \; R_{prod,k}/V_{prod,k},$$

where the K servers includes both devices and software servers, $V^{'}_{prod,k}$ is the average number of visits to resource k while holding the buffer, $R_{prod,k}$ is the total average residence time of the producer at resource k, $V_{prod,k}$ is the average number of visits by the producer to resource k, and $R_{prod,k}/V_{prod,k}$ is the average time required for a producer to obtain a single visit's worth of service from resource k. The $V^{'}_{prod,k}$ and $V_{prod,k}$ are input parameters for the model; the $R_{prod,k}$ are found using intermediate model results.

H_{cons} The buffer hold time of a consumer using a buffer element from a producer-consumer server. It is defined in the same way as H_{prod}. The average number of visits and residence time of each visit to each of the servers, $V'_{cons,k}$ and $R_{cons,k}/V_{cons,k}$, can differ from the corresponding producer class values, $V'_{prod,k}$ and $R_{prod,k}/V_{prod,k}$.

Z_{buffer} The think time of each buffer pool class customer is defined to be the sum of the buffer hold times:

$$Z_{buffer} = H_{prod} + H_{cons}.$$

The following parameters must be added to the MOL to support the producer-consumer server.

- $V'_{prod,k}$ and $V'_{cons,k}$, the descriptions for the requests for service that take place when a respective producer or consumer holds a buffer element. They are of the form $V_{e,g,s}$, where a producer or consumer group g requests service from entry e of software server or device s.

In Linearizer, the values of H_{prod}, H_{cons}, and Z_{buffer} are all assigned in the core support routine.

4.2.2 Validating The Technique With Respect To Simulation

To test the accuracy of the producer-consumer server implementation based on two SYNC servers, the following parameters are considered:

- N_p the number of producers,
- N_c the number of consumers,
- N_{buffer} the number of buffers elements available,
- U_b the utilization of each buffer element. It is the percentage of the time that an element is "held" with respect to a buffer element holding time.

The software process architecture that is used is shown in figure 4.12. A set of models with all combinations of the following model parameters has been chosen:

- Producer-consumer population vectors
  $\{(N_p, N_c)\} \, \epsilon \, \{(1,1),(4,4),(10,10),(3,6),(4,10)\}$ (the analysis is symmetrical so (6,3) and (3,6) will have the same results),
- Number of buffers $N_{buffer} \, \epsilon \, \{1,2,3,4\}$,
- Buffer utilization $U_b \, \epsilon \, \{0.0, 0.25, 0.5, 0.75, 0.98\}$,

One hundred models are considered.

When choosing the models, for each (N_p, N_c) and N_{buffer} combination, the buffer holding time for producers and consumers was altered until instances of models were found with the desired buffer element utilization U_b. A high buffer utilization value implies that each buffer element spends a high fraction of its time being produced into or consumed from.

The purpose of the test cases is to discover how the accuracy of the producer-consumer modelling technique is affected by the number of customers sharing buffer elements, the number of buffer elements, and buffer element utilization. From table 4.2, producer and consumer buffer hold times are altered to achieve the different buffer utilizations. High customer response times correspond to high buffer utilizations.

The error in predicted customer response time and buffer element utilization for each of the test cases are presented in figures 4.13 through 4.14 and 4.15 through 4.16, respectively. The errors are calculated relative to simulation results. They are shown on a grid. In figures 4.13 through 4.14, a contour diagram is imposed on top of the grid to show how changes in model parameters affect customer response times.

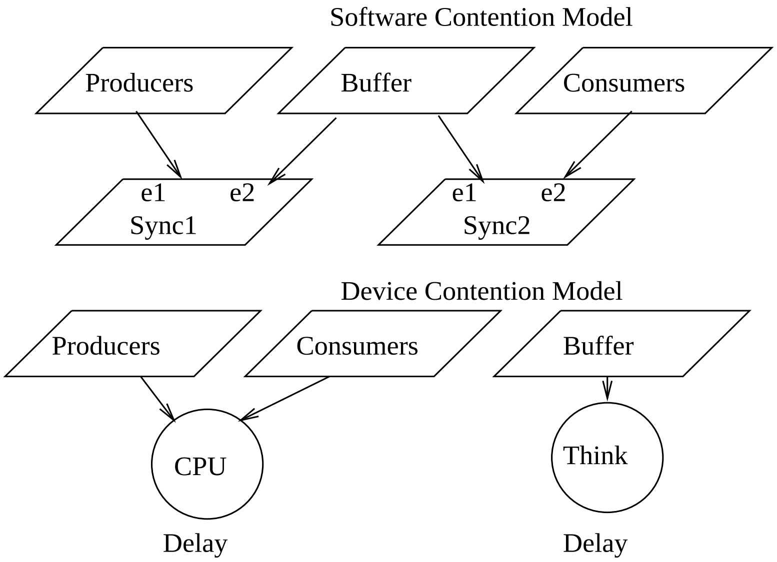


Figure 4.12: Producer-Consumer Server Test Case

Entity	Population	Visits to PCServ	Phase 1 V_{cpu}	S_{cpu}	Buffer hold Visits
Producers	N_p	1	5	1	$V'_{prod,cpu}$ varies to satisfy U_b
Consumers	N_c	1	5	1	$V'_{cons,cpu} = V'_{prod,cpu}$
PCServ	1	-	-	-	-

Table 4.2: a) Basic Model Parameters for the Producer Consumer Server Test Case.

N_p	N_c	N_{buffer}	U_b	$V'_{prod,cpu}$
1	1	1	0.25	0.903
1	1	4	0.75	12.1
4	4	2	1.0	2.34
4	4	4	1.0	4.69
10	10	3	0.75	0.609
10	10	4	0.25	0.275
3	6	4	1.0	6.25
4	10	1	0.25	0.159

Table 4.2: b) Sample of Model Parameters for the Producer Consumer Server Test Case.

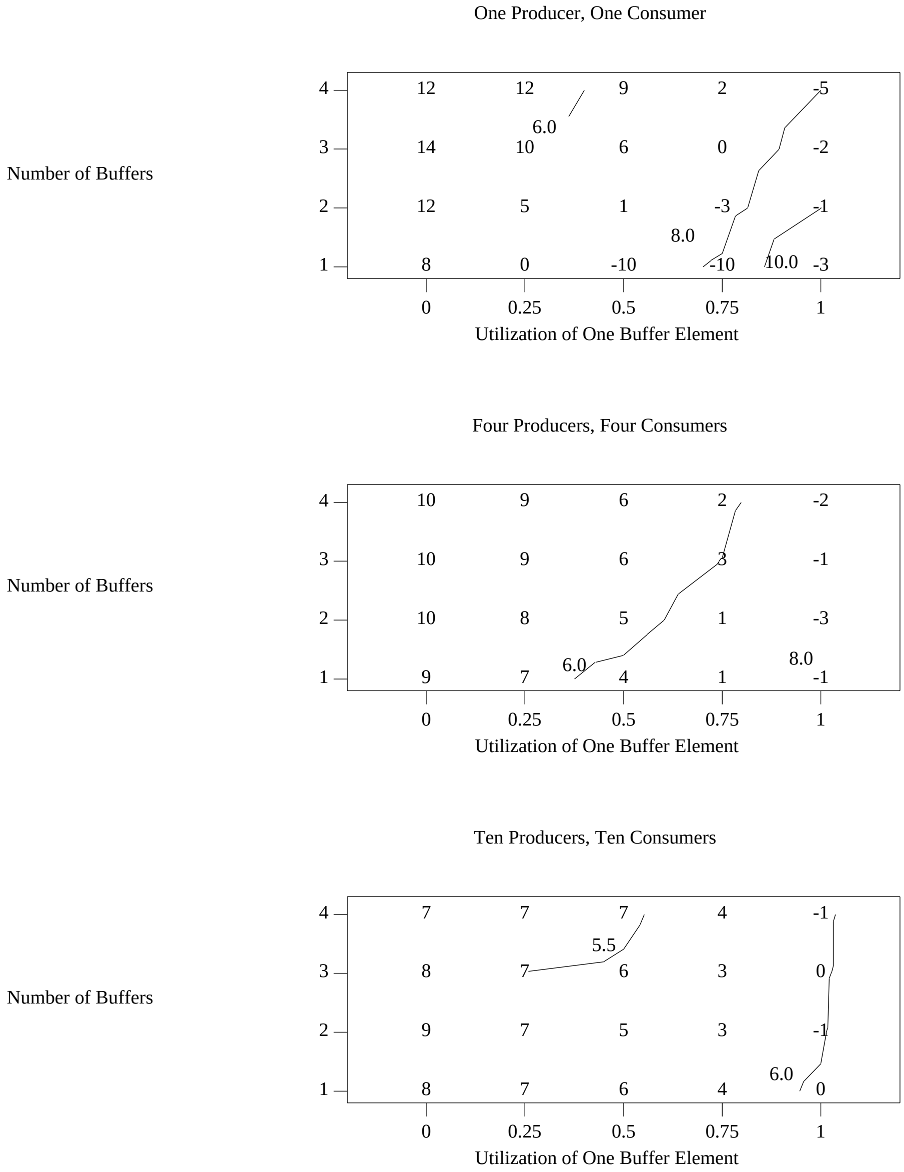


Figure 4.13: Producer-Consumer Server Test Case: Error in Estimated Customer Response Time, and Response Time Contours. Symmetric Test Cases.

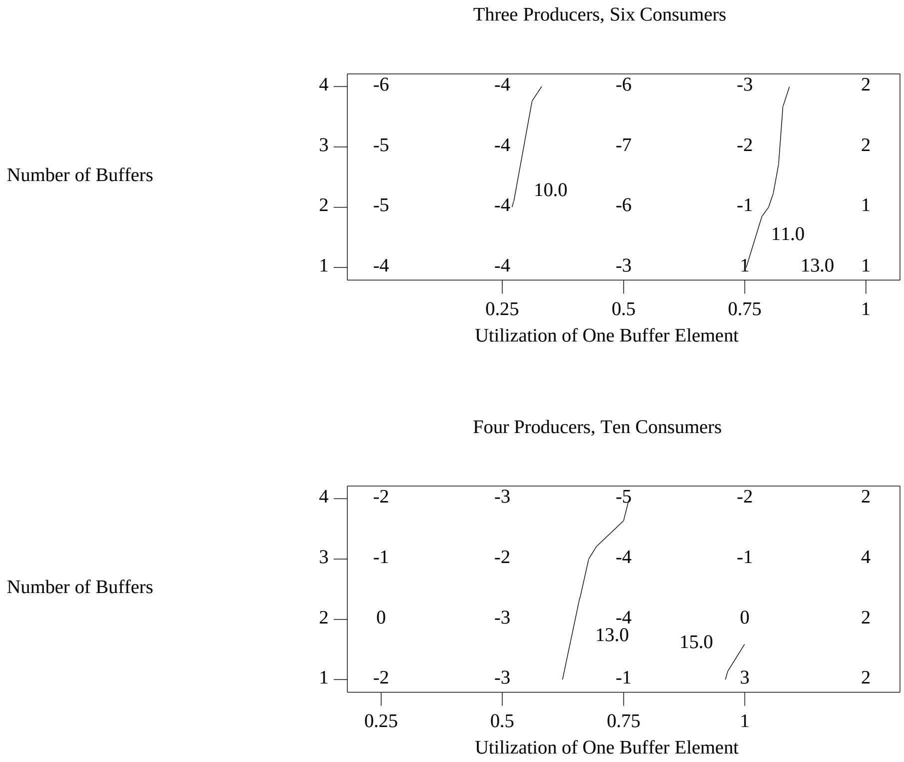


Figure 4.14: Producer-Consumer Server Test Case: Error in Estimated Customer Response Time, and Response Time Contours. Asymmetric Test Cases.

One Producer, One Consumer

Number of Buffers \ Utilization of One Buffer Element	0	0.25	0.5	0.75	1
4	0	1	1	0	-1
3	0	1	1	-1	-1
2	0	1	0	-3	-1
1	0	0	-4	-8	-2

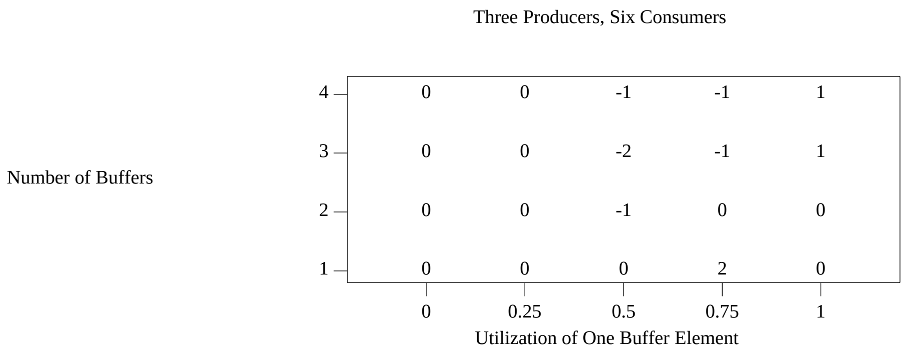
Four Producers, Four Consumers

Number of Buffers \ Utilization of One Buffer Element	0	0.25	0.5	0.75	1
4	0	1	1	3	2
3	0	1	2	3	2
2	0	1	3	3	2
1	0	2	4	4	1

Ten Producers, Ten Consumers

Number of Buffers \ Utilization of One Buffer Element	0	0.25	0.5	0.75	1
4	0	0	1	5	6
3	0	1	2	6	6
2	0	1	3	7	6
1	0	3	5	9	6

Figure 4.15: Producer-Consumer Server Test Case: Error in Estimated Buffer Element Utilization. Symmetric Test Cases.



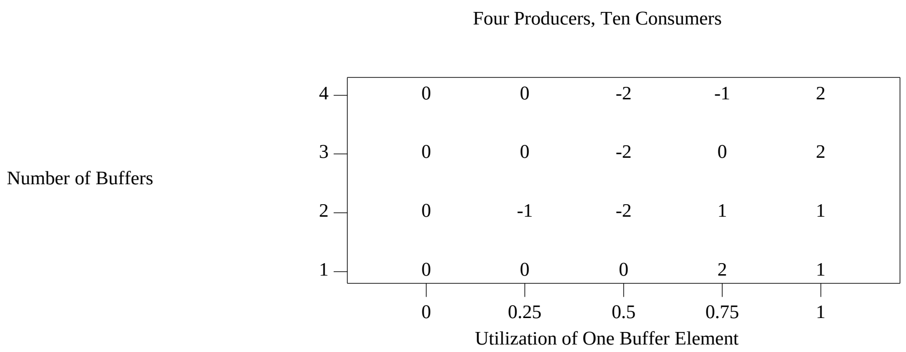


Figure 4.16: Producer-Consumer Server Test Case: Error in Estimated Buffer Element Utilization. Asymmetric Test Cases.

Essentially, two sets of test cases are considered. In the first set of test cases, the average interarrival times of the producer and consumer classes are equal. The classes behave symmetrically. The average buffer hold times are chosen to provide the desired buffer element utilization so, in general, they differ from the average interarrival time of producers and consumers. The percentage error in predicted response time for producers and consumers are nearly identical so only one set of results is presented. They appear in figure 4.13, the corresponding errors in estimated buffer element utilization appear in figure 4.15. In the second set of test cases, the numbers of producers and consumers differ, hence the average interarrival times of the classes also differ. The classes behave asymmetrically. The percentage error in predicted response times for producers and consumers are very similar so only one set of results is presented. They appear in figure 4.14, the corresponding errors in estimated buffer element utilization appear in figure 4.16.

The accuracy of the results for the producer and consumer classes in the symmetric test cases are very similar. The worst errors are when there is one producer, one consumer and three buffer elements. Under this scenario, the worst error in predicted customer response time is 14% and it occurs when the buffer elements have a 0% utilization. In this case elements take zero time to fill and empty, so they are always waiting to be produced into or consumed from. The cause of the error is most likely due to the stochastic routing assumption for buffer elements made in the analysis and is discussed later in this section. The other test cases have errors in predicted customer response times of less than 10% and usually less than 5%.

The asymmetric test cases have low errors. The worst errors are less than 7%. The SYNC server is expected to be accurate for these test cases because the interarrival rates at the entries differ.

Now consider the predicted performance measures for the buffers. The percentage **error in buffer usage** is defined as follows.

$B_{sim,a}$ simulated average number of buffers elements that are available (available for use because they are not currently being "held" with respect to a buffer hold time)

$B_{est,a}$ analytical prediction of number of buffers available;

$$\text{Percentage error in buffer usage} = \ (B_{sim,a} - B_{est,a})/N_{buffer}$$

The purpose of the measure is to describe how well the analysis is able to predict whether buffers are being actively used or are waiting to be used by a producer or consumer.

In figures 4.15 and 4.16 the percentage errors in buffer usage are presented. In general, the worst errors occur when there is a small number of buffer elements and the elements are three quarters utilized. The errors do not have a strong relationship with errors in predicted customer response times. The test cases with different numbers of producers and consumers, and hence different arrival rates, have much lower and much more consistent percentage errors in predicted buffer usage than the symmetric test cases.

The producer-consumer server's analysis makes a stochastic routing assumption for buffer elements. Once a buffer element has been used it has a 50% probability of being routed to producer customers, and similarly an equal probability of being routed to consumer customers. The simulator enforces a strict alternation of buffers between producers and consumers so that the reported errors show the effect of ignoring the true behaviour. In general, with more than one producer and consumer, the error decreases as the number of buffer elements increases, or the element utilization is extreme (low or high). As the number of buffer elements increases, the combined behaviour of the buffer elements appears as if they are alternating between producers and consumers.

To summarize, the estimated customer response times are optimistic with the greatest errors occurring at low buffer element utilizations. The symmetric cases had the largest errors in customer response times. The worst error in producer and consumer response times was 14%. The asymmetric cases had much lower errors with the worst error being 7%.

4.3 Conclusions

Synchronization between groups of processes can be represented using mean value analysis techniques. The server can be used to model nested acceptance in Ada and similar behaviour in other languages. The combination of different synchronizing groups can be used to represent the condition synchronization that is present for some producer-consumer relationships. It may also be possible to use a synchronization server to represent contention for memory in memory constrained systems. Such behaviour can be represented in layered group models. The performance of the models can be predicted using the MOL.

Chapter 5

Fast Performance Estimates For A Class Of Generalized Stochastic Petri Nets

A class of Generalized Stochastic Petri Nets (GSPN) is described that can be solved using the Method of Layers. The class of nets is appropriate for the modelling of concurrent software systems. The enumeration of the Petri Net state space is avoided and the complexity of the method is independent of the number of tokens in the net. The performance metrics that can be derived include the average number of tokens in a place and the enabling rates of transitions.

The result shows that the techniques considered in this thesis can be used to model an interesting class of Petri Net models. It also provides an example of how a complex environment, in this case a class of Petri Nets, relates to Layered Group Models (LGM).

5.1 Introduction

Stochastic Petri Nets (SPN)[1] and Generalized Stochastic Petri Nets (GSPN) have been used as tools to predict the average performance metrics for a variety of systems [Molloy 82, Marsan 84, Balbo 86]. Unfortunately, except for special classes of mod-

[1] Please refer to appendix C for definitions of the Petri-Net terminology used throughout the chapter.

els and the techniques used to model them [Ramam 80, Smith 85, Kant 87], the state space and time complexity of solution techniques for both SPN and GSPN precludes the exact analytical solution of all but the smallest models.

Finding the exact performance measures of a GSPN model requires the enumeration of its reachability set of states. Global balance techniques are then applied to this set to predict the performance of the corresponding GSPN. The space and time complexity of the solution procedure is combinatorial in the number of places and tokens. Thus only small models can be solved exactly.

Balbo et al. [Balbo 86] use a combination of separable Queueing Network (QN) [Lazow 84] and Petri Net models to predict the performance measures of a class of systems that have software serialization delays. Separable QNs can be solved using solution techniques such as Mean Value Analysis (MVA), which are much more efficient than global balance. Unfortunately, the separable QNs do not represent all the forms of behaviour that can be present in Petri Nets.

The technique of Balbo et al. uses MVA to predict contention for resources such as devices. Portions of the net are then replaced with flow-equivalent transitions that have marking-dependent firing rates that are estimated by the results of the application of MVA. The flow-equivalent transitions behave like the flow-equivalent-service-centers described on page 38. The remaining portions of the GSPN relates these flow-equivalent transitions in a manner that represents the features of the net that are not treated directly by means of separable queueing networks. The new GSPN is solved using global balance techniques.

The new GSPN model still requires the enumeration of a state space, though with the lower number of transitions it is smaller than if the reduction technique were not used. The performance measures for the state space are still found using global balance techniques so it is again possible to examine only small models using this approach.

Vernon et al. [Vernon 86] consider mapping separable QNs onto GSPNs. Each queue in the QN is represented using a place, each server as a timed transition, and routing choices (probabilities) are represented using immediate transitions. Customers in the QN are represented as tokens in the GSPN. Define an initial class of GSPN that can be generated from separable QNs as **QNPN**. The performance measures of QNPNs can be predicted using the same MVA techniques that are used to predict the performance of QNs. An example of a QN and its corresponding QNPN is shown in figure 5.1. In the QNPN, the timed and immediate transitions are represented using rectangles and line segments, respectively.

QNPNs are a subset of GSPNs that have the following restrictions:

- Timed transitions have either:
  - general service times, and act as infinite servers or processor sharing servers
  - exponentially distributed service times
- All transitions t have one input place and one output place $|IP(t)| = |OP(t)| = 1$ (see appendix C for a discussion of PN notation).

QNPNs are persistent, bounded, and conservative. In general they are live as well because separable queueing network models only contain service centers that are visited in the steady state.

An example of the restriction on the modelling power of QNs and hence QNPNs is the inability to describe simultaneous resource possession. Non-separable QNs that represent the feature must be associated with GSPNs that have transitions with more than one input and output place. A non-separable QN with simultaneous resource possession and its corresponding GSPN is shown in figure 5.2. Customers must compete for memory before accessing the CPU and disks. In the GSPN, a token that corresponds to a customer must synchronize with a token that corresponds to a unit of memory before accessing the CPU and disks.

In section 5.2, an extended class of QNPNs is described called MQNPN. MQNPNs permit the structured use of synchronization amongst QNPNs which relaxes the $|OP(t)| = |IP(t)| = 1$ constraint that exists for QNPNs. By construction, the parameters of Layered Group Models can be derived from those of MQNPNs. Thus the MOL can be used to predict the performance of MQNPNs as well as LGMs. In section 5.3, the relationship between MQNPNs and LGMs is given.

In section 5.4, two applications from Agrawal and Buzen [Agra 83], and Balbo et al. [Balbo 86] are used to evaluate the accuracy of the technique. The predicted performance measures of the LGM are compared with exact results found using global balance techniques, and the estimated values of Agrawal and Buzen, and Balbo et al. Finally, in section 5.5, the results presented in this chapter are discussed, and conclusions are presented.

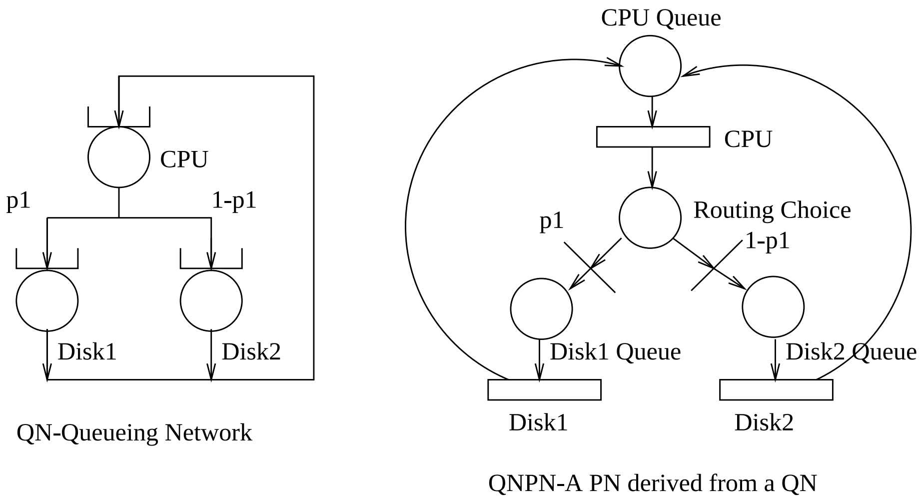


Figure 5.1: A QN and its corresponding QNPN

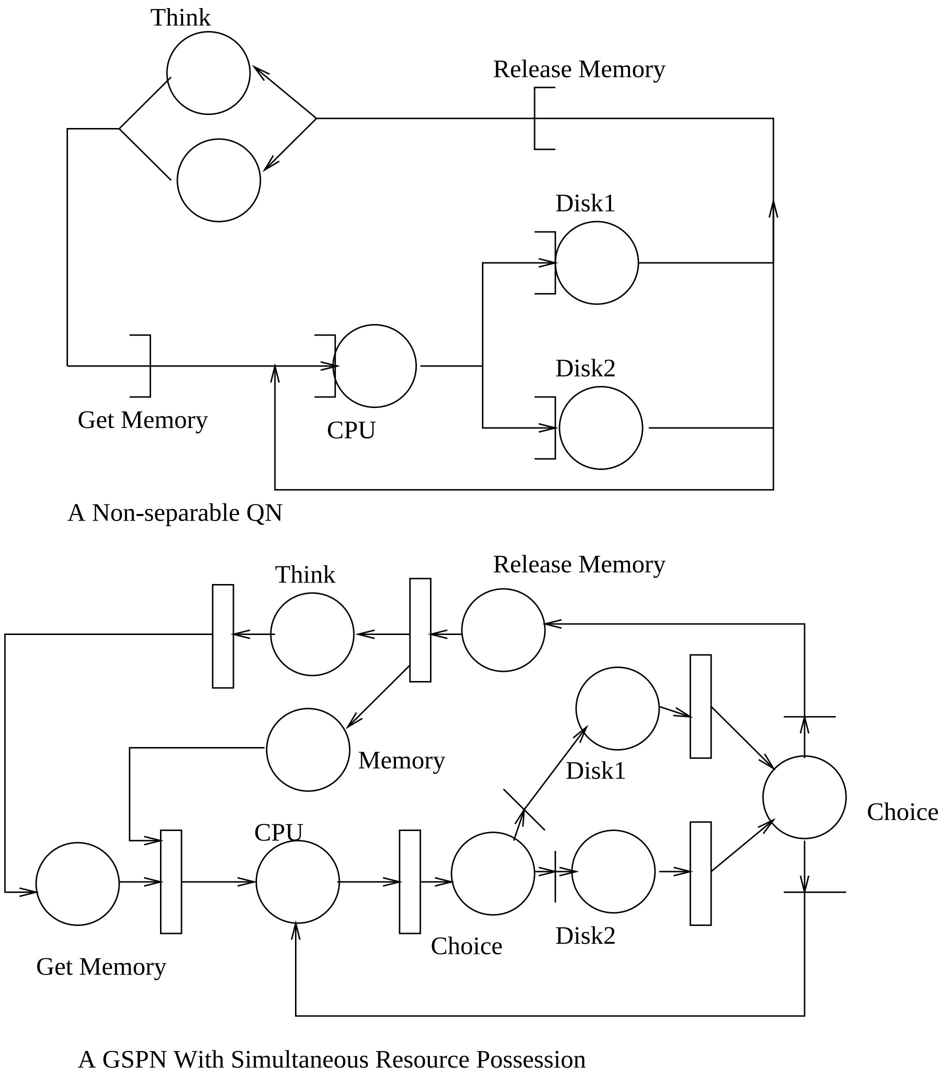


Figure 5.2: A non-separable QN with simultaneous resource possession and its corresponding GSPN

5.2 Matched QNPN (MQNPN)

Consider the following extension to QNPNs that defines a class of GSPN called MQNPN. The definition of MQNPNs ensure that they have the same structural properties as QNPNs, but in addition also permit the use of a form of synchronization.

The class of MQNPN is defined as a superset of the class of QNPN that permits the structured use of sets of **matching** transitions to synchronize what are otherwise QNPNs and/or other MQNPNs.

An example of an MQNPN is given in figure 5.3. Two QNPNs, Q_1 and Q_2, are synchronized using a pair of matching transitions T_1 and T_2. When transition T_1 fires, it consumes two tokens, one from Q_1 and one from Q_2. It produces one token into Q_3. When T_2 fires, it consumes one token from Q_3 and produces two tokens, one is placed in Q_1 and the other in Q_2. Arcs are used to connect the QNPNs to both of their matching transitions. The set of places and transitions between T_1 and T_2 can be more than just a single place. It can contain any number of places and transitions and may contain nested MQNPNs.

The structured use of sets of matching transitions has the following properties.

- The starting matching transition, T_1, of a set of matching transitions has two input arcs and one output arc; each completing transition, T_2, in the set has one input arc and two output arcs

  $|IP(T_1)| = 2$ and $|OP(T_1)| = 1$, and $|IP(T_2)| = 1$ and $|OP(T_2)| = 2$

- Arcs must go from Q_1 and Q_2 to T_1, and from each completing transition T_2 back to Q_1 and Q_2.

  $OT(p_1) = T_1$ and $IT(p_2) = T_2$ for some places $p_1, p_2 \in Q_1$ and $OT(p_1) = T_1$ and $IT(p_2) = T_2$ for some places $p_1, p_2 \in Q_2$

- The places and transitions between T_1 and the completing transitions must be a MQNPN. The MQNPN is called the **child** of the two MQNPN that synchronize at T_1.
- Two MQNPN may have many children.
- Other than via T_1 and the completing transitions, a child must not synchronize with any of its parents or siblings.
- Each place must be directly enclosed by at most one set of matching transitions.

Using these rules, MQNPNs are persistent, bounded, and conservative. If the arcs that attach the completing transitions to parent MQNPNs do not introduce any routing cycles that remove liveness, the entire MQNPN will be live as well.

An example of the unstructured use of matching transitions is shown in figure 5.4. The transition between $P2$ and $P3$, and between $P2$ and $P4$ are immediate transitions that have zero service times, but are used to introduce branching probabilities into the model. The sum of the probabilities is one. The problem behaviour arises because $Q3$, a child of $Q2$, synchronizes with $Q2$ at $Q4$. Since the behaviour of tokens in a QNPN are independent, such a net is not live for any marking. In the example, a token that flows from the parent's place $P3$ into the child's place $P7$, must synchronize with another token from the parent's place $P4$. Since all of the parent's tokens could flow into $P4$, deadlock may occur. Similarly if a child synchronizes with a sibling the net is not live for any initial marking.

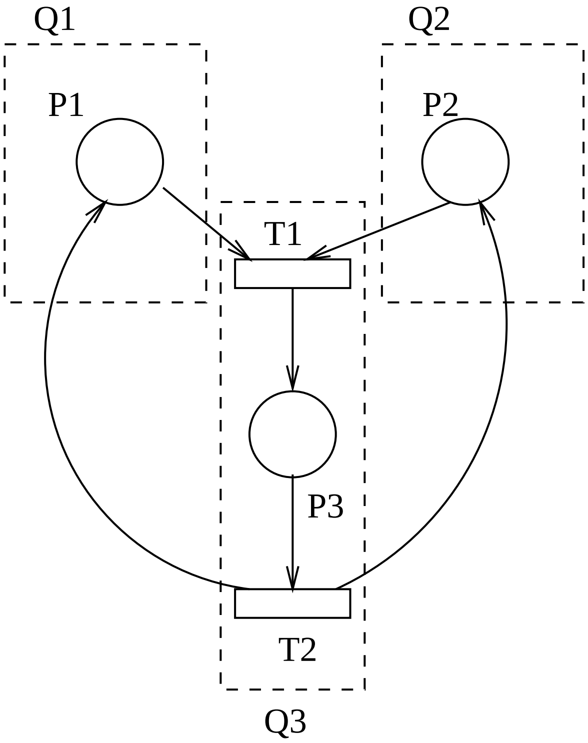


Figure 5.3: An MQNPN. Two QNPN, Q_1 and Q_2, are synchronized using matching transitions T_1 and T_2. Their child is Q_3.

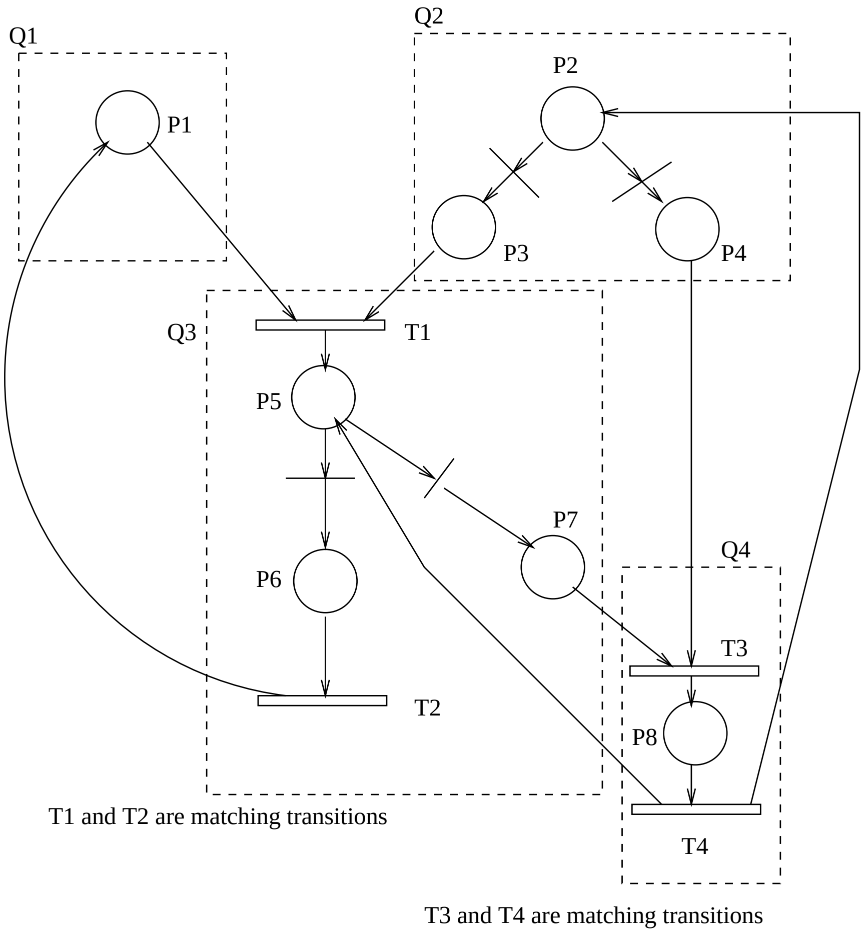


Figure 5.4: The unstructured use of matching transitions.

5.3 Layered Group Models

The models that are solved using the MOL are called Layered Group Models (LGM) and are constructed using **groups** that request service from other groups. LGMs are defined in chapter 2. For the purpose of this chapter, groups correspond to the parents and children of an MQNPN. Two parent groups synchronize via a serving or child group. The corresponding QN model contains synchronization and is therefore not separable. The serving group is a SYNCDEL server (as defined in Chapter 4). The server has two customer entries, one for each calling group. A customer must be queued for service at each of the SYNCDEL server's entries before service can begin. The customers are released in unison when the service period has completed. Any number of pairs of customers can by served at the same time. Thus the server acts as both a synchronization server and a delay server. The server may also interact with other servers.

The groups of an LGM that do not provide service to other groups, and hence are not represented as SYNCDEL servers, are defined as DELAY servers. DELAY servers do not cause synchronization delays. An example of an LGM is shown in figure 5.5. It corresponds to the MQNPN in figure 5.3. The group $Q1$ covers place $\{p1\}$, $Q2$ covers place $\{p2\}$, and $Q3$ covers places and transitions $\{T1, p3, T2\}$.

The relationship amongst tokens, places, and transitions of the MQNPNs to the customers, routing, and service centers of the groups is the inverse of the relationship defined by Vernon et al. between GSPNs and QNs [Vernon 86]. Namely, each timed transition with only one input and/or output arc is associated with a device. There can be many timed transitions associated with a device (with GSPNs, the state space generating technique takes this into account when computing transition rates between states). The place that has an arc leading to the timed transition is the queue for the device. Places that lead to immediate transitions disappear, and immediate transitions become routing choices for the group. Transitions with more than one input arc start a child MQNPN and its corresponding child group. The input arcs are distinguished in the child group through the use of entry numbers. Tokens in the MQNPN correspond to customers in the groups.

Given an MQNPN it is possible to define a corresponding LGM. The matching transitions are a part of the children in the MQNPN. The set of QNPNs that are not enclosed by matching transitions represent the groups at the highest level of the LGM and have queueing discipline DELAY. Their child MQNPNs recursively define lower levels of the LGM and have queueing discipline SYNCDEL. This defines the set of groups G in the LGM and their scheduling disciplines ψ_g. From this definition, the sets of places and transitions in a MQNPN are partitioned into the groups of the LGM. Let:

P_g be the set of places in a group g and,

T_g be the set of transitions in a group g.

The number of tokens that can reside in an MQNPN defines the population of its corresponding group N_g.

Timed transitions in the MQNPN define the devices K in the LGM, their scheduling disciplines ψ_k, and the set of devices visited by a group K_g. Device scheduling disciplines are those that are associated with the timed transitions. The average service demands by a group at a device, $S_{g,k}$, is the inverse of the service rate for its corresponding transition.

For a group g it is possible to find the average number of visits to each device k that is visited by its transitions, $V_{g,k}$, and each other group's entries $V_{e,g,h}$. To do this, treat each SYNCDEL group that is visited by group g as a single timed transition. Once all the SYNCDEL group's matching transitions are replaced by timed transitions, the group g has the structure of a QNPN. The visit ratios can be found using the approach described in section 1.2 [Triv 82]. In that section, each of a software process' statement blocks are associated with a row and a column in a matrix. The entries in the matrix correspond to the probability of moving from one statement block to another. Use of the matrix leads to the visit ratios for the statement blocks. For a QNPN, immediate

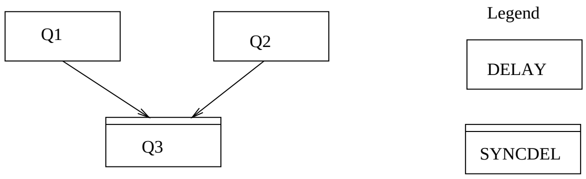


Figure 5.5: A Layered Group Model.

transitions and transitions with $|IP(t)| > 1$ or $|OP(t)| > 1$ in a group enclose its statement blocks. The visit ratios are found in the same way.

Performance estimates for a MQNPN's corresponding LGM can be found using the MOL. The results can be mapped back onto the MQNPN using the relationship between QNs and QNPNs. Namely, the average queue length of group g's customers at a device k defines the average number of tokens in the input place of group g's transition for device k. The enabling rate for a transition t is the product of the throughput of the corresponding group's customers and their average number of visits to t.

Another example of an MQNPN and its corresponding LGM is shown in figure 5.6.

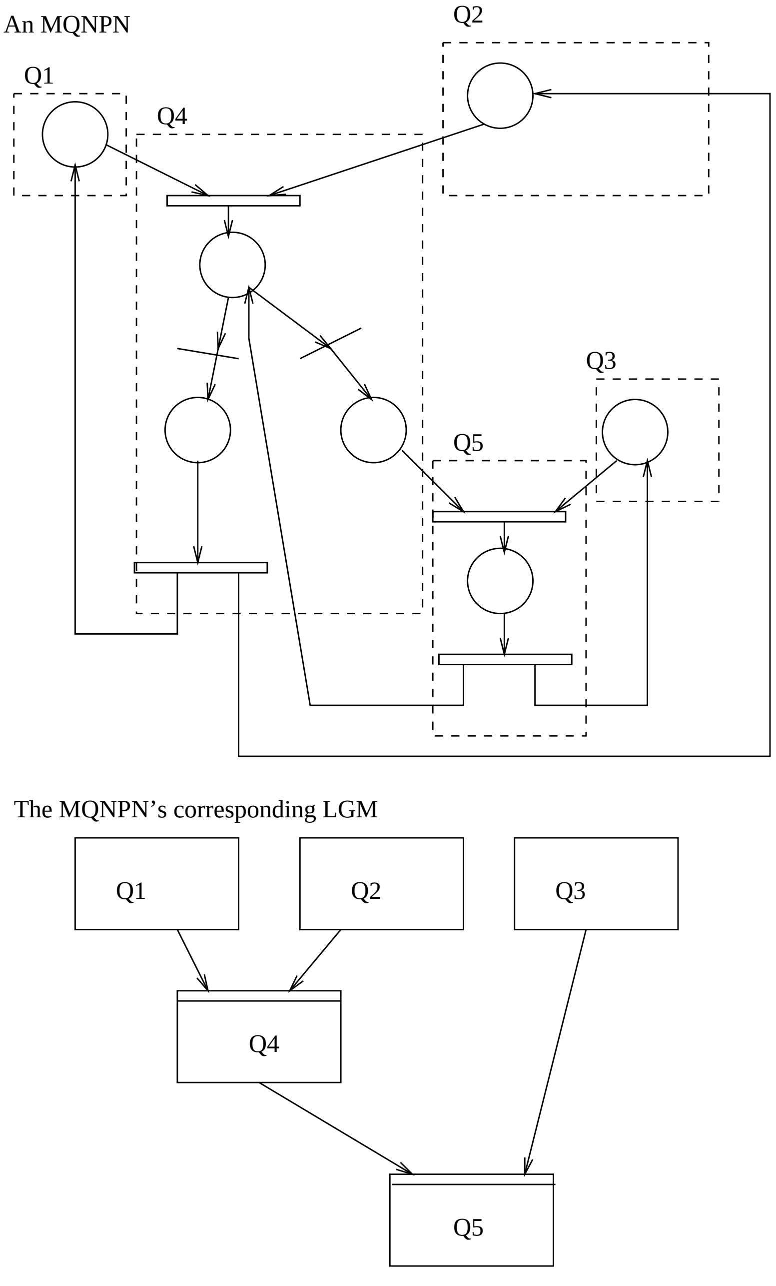


Figure 5.6: A MQNPN and its corresponding LGM

5.4 Applications

Two GSPN models are reduced to LGMs and then solved to demonstrate the accuracy of the technique. It must be noted that the solution of LGMs requires only seconds of CPU time, whereas the exact solution of GSPNs requires hours of CPU time and cannot be performed for models much larger than those considered within this paper. Similarly, the method of Balbo et al. requires minutes of CPU time for the models. As the number of tokens increases, the corresponding solution time also moves into hours.

The first model is a GSPN representing the software blocking problem described in a paper by Agrawal and Buzen [Agra 83]. The system has several customers that compete for a CPU, disk, and two software servers. The software servers also compete for the CPU and disks. The GSPN and its corresponding LGM are shown in figure 5.7. The GSPN is described in table 5.1. The mapping of places and transitions onto groups is shown in table 5.2, the scheduling disciplines are given in table 5.3, and the visit ratios are given in table 5.4. In table 5.4, N_1 corresponds to the number of customers in the model. N_1 varies from one to eight. The throughput of the class of customers is given in table 5.5.

The tables of results give the exact throughput of the primary customer class, X_{G1}, as computed by Agrawal and Buzen, and Balbo et al. It also gives the predicted throughputs of the LGMs found using the MOL, and the methods of Balbo et al. [Balbo 86], and Agrawal and Buzen [Agra 83]. The throughput corresponds to the rate at which tokens complete the specified number of visits to each transition in the QNPN that is associated with the customer class $G1$.

At $N_1 = 1$, the LGM has been solved with the assumption that there is no device contention. At $N_1 = 2$ the device contention model is pessimistic because it has four customers represented in the model when there are actually only two that can use the devices at a time. This causes a low throughput estimate. As the number of customers increases, this effect diminishes and the performance measures predicted using the MOL approach the exact results. This shows that the LGM and its corresponding GSPN have the same bottleneck.

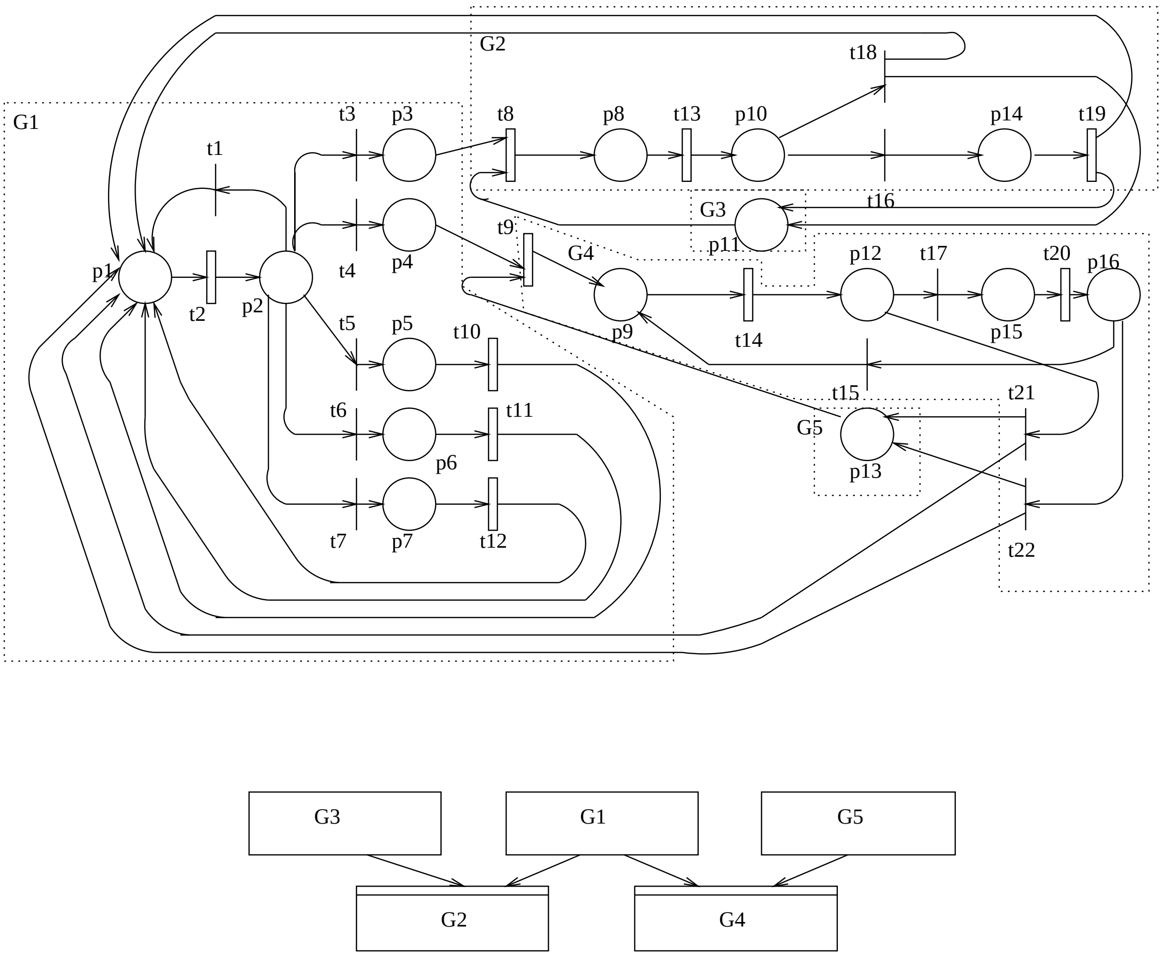


Figure 5.7: Application 1: Software Blocking GSPN and corresponding LGM.

Place	Description
p_1	CPU Queue for jobs outside Critical Section
p_2	Decision place for a job that obtained a time slice from CPU
p_3	Critical section 1 queue
p_4	Critical section 2 queue
p_5	Disk1 queue for jobs outside CSs
p_6	Disk2 queue for jobs outside CSs
p_7	Disk3 queue for jobs outside CSs
p_8	CPU queue for jobs inside CS 1
p_9	CPU queue for jobs inside CS 2
p_{10}	Decision place after obtaining some CPU service in CS 1
p_{11}	Nobody in CS 1
p_{12}	First decision place after obtaining CPU service in CS 2
p_{13}	Nobody in CS 2
p_{14}	Disk1 queue for jobs inside CS 1
p_{15}	Disk2 queue for jobs inside CS 2
p_{16}	Second decision place after obtaining some CPU service in CS 2

Device	Queueing Discipline
CPU	Processor Sharing
Disk1	Processor Sharing
Disk2	Processor Sharing
Disk3	Processor Sharing

Transition	Description	Service Time or Probability
t_1	One more timeslice, probability	0.1
t_2	CPU, service time	0.200
t_3	CS 1 entry, probability	0.3
t_4	CS 2 entry, probability	0.28
t_5	Selection of Disk1, probability	0.14
t_6	Selection of Disk2, probability	0.09
t_7	Selection of Disk3, probability	0.09
t_8	Permission to enter CS 1, service time	0.0
t_9	Permission to enter CS 2, service time	0.0
t_{10}	Disk1 service for jobs outside CSs, service time	0.056
t_{11}	Disk2 service for jobs outside CSs, service time	0.036
t_{12}	Disk3 service for jobs outside CSs, service time	0.036
t_{13}	CPU service for jobs in CS 1, service time	0.060
t_{14}	CPU service for jobs in CS 2, service time	0.081
t_{15}	Job inside CS 2 wants more service from CPU, probability	0.41
t_{16}	Job inside CS 1 wants some service from Disk1, probability	0.48
t_{17}	Job inside CS 2 wants some service from Disk2, probability	0.75
t_{18}	Job leaving CS 1 after some service from CPU, probability	0.52
t_{19}	Disk1 service for job inside CS 1, service time	0.058
t_{20}	Disk2 service for job inside CS 2, service time	0.121
t_{21}	Job leaving CS 2 after some service from CPU, probability	0.25
t_{22}	Job leaving CS 2 after some service from Disk2, probability	0.59

Table 5.1: Application 1: Software Blocking GSPN Model Description.

Group Name	Places and Transitions
G1	p1,t2,p2,t1,t3,p3,t4,p4,t5,p5,t10,t6,p6,t11,t7,p7,t12
G2	t8,p8,t13,p10,t18,t16,p14,t19
G3	p11
G4	t9,p9,t14,p12,t21,t17,p15,t20,p16,t15,t22
G5	p13

Table 5.2: Application 1: Software Blocking Model's corresponding Groups.

Device	Queueing Discipline	Transitions
CPU	Processor Sharing	t2,t18,t14
Disk1	Processor Sharing	t10,t19
Disk2	Processor Sharing	t11,t20
Disk3	Processor Sharing	t12

Group	Queueing Discipline
G1	DELAY
G2	SYNCDEL
G3	DELAY
G4	SYNCDEL
G5	DELAY

Table 5.3: Application 1: Software Blocking LGM's Servers.

Group	N_1	$V_{1,cpu}$	$V_{1,disk1}$	$V_{1,disk2}$	$V_{1,disk3}$	$V_{1,t8}$	$V_{1,t9}$
G1	Varying	10	0.9	0.9	1.4	3.0	2.8
Group	N_2	$V_{2,cpu}$	$V_{2,disk1}$	-	-	-	-
G2	1	1	0.48	-	-	-	-
Group	N_3	$V_{3,t8}$	-	-	-	-	-
G3	1	1	-	-	-	-	-
Group	N_4	$V_{4.1,cpu}$	$V_{4.1,disk2}$	-	-	-	-
G4	1	1.444	1.083	-	-	-	-
Group	N_5	$V_{5,t9}$	-	-	-	-	-
G5	1	1	-	-	-	-	-

Table 5.4: Application 1: Group Model Parameters For Software Blocking Model.

Pop. $G1$	Exact X_{G1}	LGM X_{G1}	% Err	Balbo X_{G1}	% Err	AG&BU X_{G1}	%Err
1	1.54	1.54	0.0	1.54	0.0	1.54	0.0
2	2.11	1.91	9.5	2.15	1.9	2.21	4.7
3	2.37	2.23	5.9	2.42	2.1	2.51	5.9
4	2.49	2.40	3.6	2.55	2.4	2.64	6.0
5	2.56	2.50	2.3	2.61	1.9	2.71	5.9
6	2.60	2.55	1.9	2.65	1.9	2.75	5.8
7	2.62	2.59	1.0	2.67	1.9	2.77	5.7
8	2.63	2.62	0.4	2.68	1.9	2.78	5.7

Table 5.5: Group $G1$ Throughput for Application 1 the Software Blocking Problem.

The second model appears in a paper by Balbo et al. [Balbo 86]. It represents the Class Migration Problem. Several customers compete for a CPU, disk, and n instances of a software server that share a single queue of requests for service. The software servers also compete for the CPU and disk. The GSPN and its corresponding LGM are shown in figure 5.8. The GSPN is described in table 5.6. The mapping of places and transitions onto groups is shown in table 5.7, the scheduling disciplines are given in table 5.8, and the visit ratios are given in table 5.9. In table 5.9, N_1 corresponds to the number of customers in the model. N_1 varies from one to ten. N_3 is the number of instances of the software server. The throughput of the class of customers is shown in tables 5.10 and 5.11 for $N_3 = 2$ and $N_2 = 4$, respectively. The value of N_2 is defined as the minimum of N_1 and N_3 for a given model. It represents the maximum number of customers that can receive service at the same time.

As in the first example, with population $N_1 = 1$ the models have been solved assuming no device contention. For each case $N_3 = 2$ and $N_2 = 4$, when $N_1 = 2$ the throughput estimate is pessimistic. Again this is because the device contention model is pessimistic. As the number of customers increases, the throughput estimates approach the exact results for the models which again shows that the LGM and its corresponding GSPN have the same bottleneck.

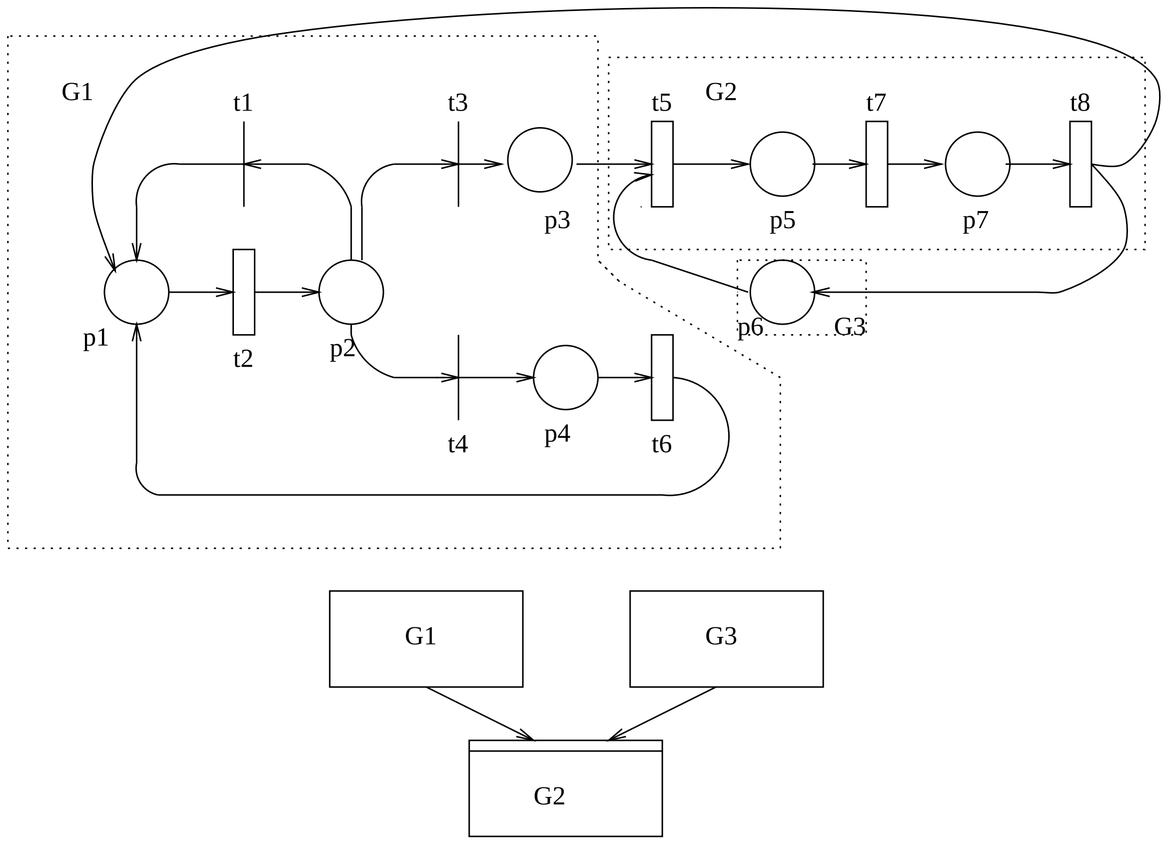


Figure 5.8: Application 2: Class Migration GSPN and corresponding LGM.

Place	Description
p_1	CPU for jobs inside main class
p_2	Decision place for a job that obtained time slice from CPU
p_3	Blocking queue for first migration class
p_4	Disk queue for jobs in main class
p_5	CPU queue for jobs of first migration class
p_6	How many more can enter the first migration class?
p_7	Disk queue for jobs inside the first migration class

Device	Queueing Discipline
CPU	Processor Sharing
Disk	Processor Sharing

Transition	Description	Service Time or Probability
t_1	One more time slice, probability	0.1
t_2	CPU service for jobs in main class, service time	0.02
t_3	First class migration, probability	0.4
t_4	Selection of Disk, probability	0.5
t_5	Permission to enter first migration class, service time	0.0
t_6	Disk service for jobs in main class, service time	0.025
t_7	CPU service for jobs in first migration class, service time	0.02
t_8	Disk service for jobs in first migration class, service time	0.025

Table 5.6: Application 2: Class Migration GSPN Model Description.

Group Name	Places and Transitions
G1	p1,t2,p2,t1,t3,p3,t4,p4,t6
G2	t5,p5,t7,p7,t8
G3	p6

Table 5.7: Application 2: Class Migration Model's corresponding Groups.

Device	Queueing Discipline	Transitions
CPU	Processor Sharing	t2,t7
Disk	Processor Sharing	t6,t8

Group	Queueing Discipline
G1	DELAY
G2	SYNCDEL
G3	DELAY

Table 5.8: Application 2: Class Migration LGM's Servers

Group	N_1	$V_{1,cpu}$	$V_{1,disk}$	$V_{1,t5}$
G1	Varying	10	5	4
Group	N_2	$V_{2,cpu}$	$V_{2,disk}$	-
G2	$\min(N_1, N_3)$	1	1	-
Group	N_3	$V_{3,t5}$	-	-
G3	Varying	1	-	-

Table 5.9: Application 2: Group Model Parameters For Class Migration Problem.

Pop. $G1$	Exact X_{G1}	LGM X_{G1}	% Err	Balbo X_{G1}	% Err
1	1.98	1.98	0.0	1.98	0.0
2	2.63	2.39	9.1	2.63	0.0
3	2.92	2.73	6.5	2.93	0.3
4	3.07	2.93	4.5	3.09	0.6
5	3.16	3.04	3.8	3.19	2.2
6	3.21	3.11	3.1	3.24	0.9
7	3.24	3.15	2.8	3.28	1.2
8	3.25	3.17	2.5	3.30	1.5
9	3.26	3.21	1.5	3.31	1.5
10	3.27	3.22	1.5	3.31	1.2

Table 5.10: Group $G1$ Throughput for Application 2 the Class Migration Problem, $Pop.G2 = 2$.

Pop. $G1$	Exact X_{G1}	LGM X_{G1}	% Err	Balbo X_{G1}	% Err
1	1.98	1.98	0.00	1.98	0.0
2	2.63	2.35	10.6	2.63	0.0
3	2.95	2.81	4.7	2.95	0.0
4	3.13	3.04	2.9	3.13	0.0
5	3.25	3.19	1.8	3.25	0.0
6	3.32	3.28	1.2	3.33	0.3
7	3.38	3.35	0.1	3.38	0.0
8	3.41	3.40	0.1	3.42	0.1
9	3.44	3.42	0.1	3.45	0.1
10	3.45	3.44	0.1	3.47	0.1

Table 5.11: Group $G1$ Throughput for Application 2 the Class Migration Problem, $Pop.G2 = 4$.

5.5 Remarks and Conclusions

The accuracy of the technique is lower than that of Balbo et al. [Balbo 86] but requires significantly less computation. The predicted performance estimates are closer to the exact results than those of Agrawal and Buzen [Agra 83] for the models considered in their paper. The new technique has a low performance prediction cost that grows polynomially with the number of places and transitions in the GSPN model.

The results appear to improve as the number of tokens in the nets increase. This is precisely when GSPN state generating techniques become infeasible. The MOL has been used to predict the performance of LGMs with more than twenty groups and many customers. LGMs with a eighty groups and several hundred customers (tokens) are not expected to pose any significant computational problems to the MVA software used to implement the MOL. Thus it is believed that large GSPNs can be studied using this approach. However, as the number of groups in each layer of the model increases the Bard-Schweitzer MVA would be more appropriate for use with the MOL algorithm because it is more efficient.

The technique could be improved by including more information in the model regarding the variance in group response times and visit ratios. Further work has already been done (but not presented in this thesis) that relaxes the need for the output arcs of matching transition T_2 to go directly back to the parent QNPNs. However, MQNPNs cannot be used to model systems that have arbitrary synchronization behaviour. Inhibitor arcs and coloured GSPN have not yet been considered.

The results produced by the proposed method can be used to narrow design spaces of the GSPNs under consideration. If necessary, the reduced set of GSPNs can be solved using more accurate (but computationally intensive) techniques.

Finally, the technique can be used to predict transition enabling rates and the average number of tokens in each place. Though the average number need not be an integer, it may be possible to use this information in techniques that generate state spaces. For example, a feasible marking near the *average* number of tokens in each place may be used to define an initial state in a Markov chain. The chain could be constructed so that it contains the states that are near to the initial state but not the full reachability set. Also, the size of state spaces that are generated could be reduced by eliminating low probability states or by initializing state probabilities in iterative solutions of the underlying Markov chains.

Chapter 6

Ada Applications

In this chapter the general problem of developing performance models for Ada applications is considered. Section 6.1 discusses the Ada language features that affect an application's performance, how information about Ada programs can be measured, and how this relates to software engineering and performance modelling. The discussion does not assume that the applications only contain the performance features that have been analysed in this thesis.

Secondly, two Ada applications are presented in section 6.2. They are prototypes that have software process architectures and resource consumption patterns that are of interest. The applications have been implemented and executed, and their performance has been measured. The results are compared with predictive performance models that have been solved using techniques developed for this thesis. The source code for the prototypes is given in appendix D.

Finally, section 6.3 provides a summary of the results.

6.1 The Performance of Ada Applications

The Ada programming language [DOD 83] permits a single application to make use of many tasks executing concurrently, to accomplish some overall goals. Tasks are dynamic software processes that share one or more processors are permitted to synchronize and communicate with one another. The synchronization caused by the interactions amongst processes can lead to counter-intuitive performance behaviour. This is the case in any language or operating system (O/S) environment that permits concurrent processing within an application. The software development methodology being used must be augmented to ensure that the performance requirements of an application that are affected by task synchronization will be met.

If the performance of an application is to be studied, a methodology is required for specifying responsiveness goals for the application as a whole and for its components. For the goals to be useful, it must be possible to verify that they are feasible, and, after the application is implemented, that they are being met.

The type of information that can be collected to describe an executing application should influence the way in which performance requirements are specified for its performance model, design, and corresponding source code. In this way, performance requirements, estimates, and measurements can be easily compared.

The following topics are considered in this section.

- An example of a software performance engineering methodology.
- Why software monitoring is necessary.
- Information that needs to be collected to reconstruct an Ada application's tasking performance behaviour.
- Examples of transitions in an Ada Run Time System at which monitoring should take place.
- The relationship between information that can be collected from monitoring and the performance requirements specification for the source code of an Ada application.
- Features that should be considered when constructing performance models of applications.

A brief example of a software performance engineering methodology is given in section 6.1.1. It suggests how performance can be considered at each stage of an application's development. Any software engineering methodology can be adapted to consider performance [Smith 90]. To make effective use of such methodologies, it is necessary to quantify the resource requirements of an application. In section 6.1.2, it is shown that software monitoring is necessary for Ada applications.

Section 6.1.3 discusses the language features that influence Ada program behaviour: the masters, which are scopes that govern task creation and termination, and the tasks, which are concurrent processes that communicate with one another. Information about a system can be collected and expressed in terms of masters and tasks. The monitoring data for masters is used to describe parallelism within an Ada program; that for tasks, describes resource usage and communication delays.

In section 6.1.4 a list of the type of events that affect an Ada program's behaviour is provided. The list is expected to be similar for other language environments as well. The problem of monitoring overhead is also discussed.

Section 6.1.5 shows how information obtained from monitoring relates to Ada source code. **Services** are defined as portions of an Ada program whose execution can be distinguished from monitoring information. When choosing a software process architecture, its structure and performance requirements should be defined in terms of services. This approach provides a bridge between requirements, performance modelling, monitoring, and source code.

In section 6.1.6, a description of the features needed to describe predictive performance models of Ada software is given.

6.1.1 Software Performance Engineering

Software performance engineering is a term that has been coined by Dr. C.U. Smith. It can be defined as a combination of a software engineering methodology, predictive performance modelling techniques, and a method for measuring the performance of a functioning software system. The following describes an example of a software performance engineering approach.

A software engineering technique can be enhanced to include performance information [Smith 90, Buhr 89]. When the software modules for the system are first chosen, performance goals and resource consumption estimates can be assigned. As the modules are refined, appropriate goals and estimates can be set for their components.

Software performance modelling techniques, such as mathematical analysis [Smith 90, Reiser 78, Rolia 87, Rolia 88, Liu 73, Woods 91], simulation [Smith 90], and performance oriented prototypes [Howes 89], can be used to predict the performance of the software system being modelled. Thus, given a set of modules with performance goals and resource consumption estimates, a model can be created and used to predict whether the intended system will meet its overall responsiveness requirements. If it does, the design appears to be feasible; otherwise, either some of the modules' resource consumption estimates or else the performance goals must be reconsidered.

As the system is being implemented, the resource consumption and performance measures of the modules can be measured using monitoring. Performance models can be updated to include more accurate resource consumption estimates as these become available, and the predicted performance of the models can be compared with the responsiveness goals of the system. Once the system is fully implemented, the measured values can be used to confirm that the responsiveness goals of the system have indeed been satisfied.

If a system is to be modified, the performance impact of proposed changes on it can be considered using software performance modelling techniques. The system's model can be altered to reflect the changes. If the predicted effects of the changes are satisfactory, they can be permitted to take place. Examples of such changes are: upgrading system hardware to use faster processors, or redesigning a software subsystem in an attempt to improve its quality. The effects of such changes on performance are rarely intuitive. Performance models can be used to predict the effects without actually altering the system.

6.1.2 Why Software Monitoring is Necessary

For any software performance engineering technique to be effective, tools are needed to measure the resource consumption of software modules and the interactions amongst modules. In Ada, these values include device usage and information about intertask communication and parallelism. Unfortunately, most operating system tools are not designed to provide the detailed information that is desired. Depending on the implementation, many Ada tasks can be associated with a single O/S process. Multitasking can be indistinguishable from single-threaded program execution; thus, it can be difficult to relate O/S performance measures to individual tasks.

Ada manages an application's concurrency through a runtime system (RTS) kernel. Without modifying a user application's source code, the RTS can be instrumented by making it call monitoring procedures as an application executes. The procedures can collect information that describes the application's dynamic behaviour and record it in a log file for later analysis.

The monitoring has as its functional requirement the measurement of resource usage, system function usage, and delays due to intertask communication and task creation. Essentially, information about events that affect an application's performance should be logged in permanent storage. The method must be efficient so that it does not distort the application's performance. A user should be able to control the subset of information that is to be logged, and to enable and disable monitoring while an application is executing.

Ada tasking is highly structured, so a log file can be processed to provide significant information about a program's behaviour. For example:

- The maximum time required by a specific task to execute.
- The percentage of time that a task misses a processing deadline.
- The scenarios that cause the missed deadlines.
- The average time required for a task to execute.

- The average resource requirements of a task.

Such values can be used to ensure that responsiveness goals are met and to help create performance models.

6.1.3 Masters and Tasks

The purpose of software monitoring is to provide information about a program's execution. Thus, the log file information for a program should be directly related to its software. In Ada, "tasks" and "masters" are the high-level objects that influence the behaviour of Ada programs. They are defined by the source code that implements the program. Tasks, masters, and intertask communication are all defined in chapter nine of the Ada Language Reference Manual (ALRM) [DOD 83].

Tasks are declared in Ada programs through the use of task variables, simple tasks, and accesses (or pointers) to task variables. The latter two can be considered as special cases of task variables; hence, they will all be described here as task variables. In the example of figure 6.1, IO-Task, IO-User-1, and IO-User-2 are all task variables. In Ada, arbitrarily complex data structures, such as records or arrays, can contain task variables.

In general, there can be many instances of a task variable in existence at any time. Each instance can be associated with a unique task identifier. To present monitor data in the context of Ada source code, it is necessary to associate each instance of a task with the unique variable name in the Ada source that is used to create it. In this way performance measures can be deduced for each task variable in the program. A compiler could make this information available or a user could add such information to the log file each time task instances are created.

Ada tasks can be created and terminated within what are called masters. The number of masters in an Ada program is fixed; the ALRM defines the scope of each master. In figure 6.1, the procedure Master-Scope is the master scope for the three task variables IO-Task, IO-User-1, and IO-User-2. An analogy can be drawn between the

```
procedure Master_Scope is

task IO_Task is
  entry Read_Information(Information : out Information_Type);
  entry Write_Information(Information : in Information_Type);
end IO_Task;

task type IO_User_Type;

IO_User_1, IO_User_2 : IO_User_Type;

task body IO_Task is
Local_Copy : Information_Type;
 begin
   loop
    select
      accept Read_Information(Information : out Information_Type) do
        Read_From_Terminal(Information); -- I/O routine to read
      end Read_Information;
    or
      accept Write_Information(Information : in Information_Type) do
        Local_Copy := Information;       -- copy data
      end Write_Information;             -- release the caller
      Write_To_Terminal(Local_Copy);     -- I/O routine to write
    or
      terminate;
    end select;
  end loop;
end IO_Task;

task body IO_User_Type is
Information : Information_Type;
begin
  Information := Null_Information_Type; -- initialize value
  when Have_Finished(Information) loop  -- quit based on value
    IO_Task.Read_Information(Information);
    -- do some work to modify information
    IO_Task.Write_Information(Information); -- do asynchronous write
    -- do more work
  end loop;
end IO_User;

begin -- Master_Scope
  Null; -- In this example, calling the Master_Scope procedure
        -- causes the IO_Task, IO_User_1, and IO_User_2 to be
        -- created. The procedure can return to its caller when
        -- the tasks have terminated.
end Master_Scope;
```

Figure 6.1: An Ada example with three tasks and a master scope.

scope of a reentrant subroutine and a master's scope. The subroutine's local variables have a lifetime equivalent to that of the subroutine. Similarly, a master cannot finish until all of its child tasks have terminated. Thus masters can have an impact on the performance of an Ada application.

Since masters can be reentrant, it is possible to have many instances of a master scope at any time. Each instance can be associated with a unique identifier. Master scope instances should be associated with their unique master scope name in the source code so that the master's performance measures can be deduced. As with task variables, a compiler could make this information available or a user could add such information to the log file as master scope instances are created.

In figure 6.1, when the procedure Master-Scope is called it automatically creates the IO-Task, IO-User-1, and IO-User-2 tasks. It is assumed that each of the tasks executes on its own virtual processor (which may or may not be separate physical processors). The procedure is the master for the tasks. The IO-User-Tasks initiate rendezvous with the entries of the IO-Task. The IO-Task guarantees that only one read or write operation occurs at a time. Control will be passed back from the procedure Master-Scope to its caller once the tasks have terminated.

To conclude, the following information is needed to identify the context of a master.

- Master scope identifier (distinguishes instances of masters).

- Master scope name (to associate a master with source code).

The following information is needed to identify the context of a task.

- Task identifier (distinguishes instances of tasks).

- Task variable name (to associate a task with source code) .

- Master scope identifier (distinguishes instances of masters).

6.1.4 Events in a Runtime System

To collect performance information about masters and tasks, calls must be made to monitoring routines when events that affect multitasking performance occur. In Ada, these include:

- A master scope instance is created, or terminated.
- Task instance creation, or termination.
- A caller initiates a rendezvous, or resumes after the rendezvous is completed.
- A task is waiting to accept a caller, accepts a caller from an entry, or releases a caller.
- An exception is raised.
- Memory is requested from a heap, or released to a heap.
- A lock or semaphore is used.
- The request for service from a device (which may require the instrumentation of I/O routines).
- User defined events may also be included (which requires user instrumentation).

A complete description of valid states and state transitions for Ada tasks is provided in [Burns 87]. Two examples of task states are *suspended for rendezvous*, and *suspended for accept*. Some corresponding transitions for these states include *suspension by start of rendezvous*, and *continuation after the start of an accept*.

An RTS can recognize these transitions, or subsets of the transitions as is appropriate, and make calls to monitor routines that would pass information to a log file. The information should be sufficient to describe the context of the transition. For example, a task creation record may be as follows:

- Log record type is "task creation".

- The master scope identifier.
- The new task identifier.
- The task name.
- A time stamp.

A rendezvous log entry should contain the following information:

- Log record type which is type "rendezvous".
- The task identifier of the caller.
- The task identifier of the callee.
- The entry being called.
- A time stamp.

The acceptance of a call log entry should contain the following information:

- Log record type is "acceptance" of a caller by a callee.
- The task identifier of the caller.
- The task identifier of the callee.
- The entry that corresponds to the accept statement.
- An accept statement identifier.
- A time stamp.

Based on the semantics of the Ada language, the log file can be processed to provide significant information about its corresponding program's behaviour. For example, when a task initiates a rendezvous, barring program failure, the rendezvous will complete. The maximum time required for a rendezvous between tasks can easily be deduced by log file post-processing tools using task context information and timestamps.

Similarly, the average and variance of the time required for functions to be performed by a task, and the percentage of time that a task misses real-time deadlines can also be found.

For example, monitor data for the Ada code in figure 6.1 could be used to show that for a particular session, the maximum time for an IO-User to read information via the IO-Task was less than the number of time units indicated in a performance goal.

One issue that cannot be ignored is the performance degradation that can be incurred by such monitoring. If the overhead is too high, the behaviour of the program will be altered to such an extent that the information provided is no longer useful. Therefore to minimize degradation, an analyst should only request information that is necessary for the problem at hand. For example, a user may only wish to monitor one subset of masters or tasks at a time.

Consider the following bound on the amount of log data that can be collected per unit of time. Assume that a system that can move one megabyte of information from memory to permanent storage per unit of time without substantially altering the behaviour of the system under study. If, for example, a monitor record requires 32 bytes of information, approximately 30,000 monitor events can be recorded per unit time.

Another way to look at degradation is to quantify the effect of monitoring on the RTS. If monitoring increases RTS processing by 30% and the RTS is known to be responsible for 10% of an application's processing time, this suggests approximately a 3% degradation in application performance. Similarly, if the RTS is responsible for 50% of an application's processing, this suggests approximately a 15% degradation.

Hybrid monitors have been used to collect information about application performance on multi-processors [Hab90a, Hab90b]. A hardware probe can be used to examine information on a bus or memory port. When software emits messages that contain monitoring information, the probe is able to collect it and log it with minimal impact on the system's performance. Haban et al. have shown that this can be done with less than 0.1% overhead to a multi-processor system.

6.1.5 Task Bodies and Performance Requirements

The software performance engineering process described in section 6.1.1 presumes that it is possible to specify performance requirements for applications, to show that they are feasible using modelling, and to verify that they have been met using monitoring. It is essential that performance requirements be expressed in a way that makes this process both possible and straightforward.

The following method is suggested for describing requirements at the task interaction level. When a task executes, certain points in its code are recognizable by the RTS. These are the events where monitoring should occur. A task's body can be divided into portions of code whose execution can be distinguished in the monitoring log. These portions of code correspond to abstract functional services that are performed by the task on behalf of the system.

Services are defined as follows.

- The body of a master scope that is not a task is a service with one phase.
- The sequence of statements that can be executed from a task's first begin statement up to its first accept statement(s) or end of body is a service with one phase.
- The sequence of statements that can be executed from a unnested accept statement up to the next unnested accept statement(s) or end of body is a service with a rendezvous phase and a post-rendezvous phase.
- The body of a nested accept statement is a service with one phase.
- The sequence of statements that can be executed from an Ada delay (In Ada, *Delay N* causes a delay of approximately N seconds) statement in a select alternative statement up to the next accept statement(s) or end of body is a service with one phase.

From the definition, the number of services in a task is fixed, and is bounded by the sum of a single service for the start-up processing, the number of unnested accept statements, and the number of delay choices in select statements. An invocation of a service is defined as the execution of the statements within the service. For services that begin with an accept statement, the number of invocations of the service is defined as the number of calls accepted by the accept statement.

First, consider the services offered by the tasks in figure 6.1. The Master-Scope procedure has one service. It contains the statements between line 37 and 39. The processing attributed to the Master-Scope procedure includes the elaboration costs associated with the tasks it creates. The task body of IO-User-1 and IO-User-2 have one service. It is a processing service that begins at line 29 when the task is activated and ends at statement 36. The task body of IO-Task has three services: a start up service that begins when the task is activated and ends when the task reaches statement 12, a read service that begins at statement 13 and ends at statement 12, and a write service that begins at statement 17 and ends at statement 12, the termination overhead associated with statement 22 can be associated with the start up service.

The use of services provides the bridge between requirements, performance models, source code, and monitoring.

- Since services are operational it is straightforward to specify performance requirements. For example, the time to write information to the terminal must be no greater than 2 time units.

- Performance models can be created that describe the relationships amongst services. The models are described in section 6.1.6.

- Once an application is implemented and executed it can be monitored. From the monitoring log it can be inferred which of a task's services is in use. The timestamps in monitoring data can be used to verify that requirements have been met for each service.

Services based on accept statements are a convenient way of considering a task's behaviour in performance models. Relationships amongst entries may not be sufficient, because there can be many instances of accept statements within a task that correspond to the entry, and the behaviour of the task may differ dramatically with each instance.

Using master scope and task identifying information along with accept statement identifiers, the monitor data in the log file can be used to infer which of a task's services is in use. Performance measures can be deduced for each of the application's services by log file post-processing tools. In this way performance requirements that are specified for services can be verified.

If a task's service does not meet its performance goal, then either its own internal processing is at fault, or the services it depends on take too long. Monitoring can identify the intertask or interservice problems. Other tools such as subprogram level profilers can be used to study the internals of task or service code.

6.1.6 Performance Models

A software process architecture can be described as a set of interacting services. A service can initiate a rendezvous with a service of a task. Performance goals and resource consumption should be quantified for each service.

The following is a description of several of the issues that must be considered when creating predictive performance models of Ada applications.

A performance model should have the structure and parameters to describe:

- The flow of control between masters and tasks (to describe parallelism).
- The flow of control amongst services within a task (to describe a task's internal structure, including guards).
- For each phase of each service (to describe intertask synchronization and communication):

- The average number of visits to each device. For example, visits to the central processor, or requests for memory from a heap.
- The average amount of work to be performed by the device per visit. For example, three time units of service at a processor, or a 10 Kilobyte block from a disk.
- The average number of visits from a service to each other tasks' services. For example, the execution of a service requires an average of four rendezvous with another task's first service that corresponds to entry 1.

The analytical techniques presented in this thesis can be used to study fixed sets of Ada tasks that accept callers in a first-come-first-served manner and do not use guards. It is also assumed that a task that provides service to other tasks never requests service from those tasks, directly or indirectly.

6.2 Predicting the Performance of Two Ada Applications

Two multi-tasking Ada programs are considered in this section. The programs have been instrumented to collect response time information and then report the results. Analytical models for the programs are created and then solved using the MOL. It is shown that the predicted performance behaviour of the programs is in agreement with the measured values. Furthermore, a second version of each program is considered called the **baseline versions**. In these baseline versions, the same amount of work is done by the program but the software servers are removed. In this way, task synchronization is ignored. The measured performance of the baseline programs is significantly different from the original programs. This shows that task synchronization can have a significant effect upon a program's performance and, therefore, that techniques are needed to help understand the performance of multi-tasking software systems. The techniques developed within this thesis can be used for this purpose.

The Ada programs were developed and executed using version 1 release 2 of IBM's Ada/6000 compiler and runtime system on an RS/6000 running the AIX operating system. Processing time was allocated to Ada tasks on a Processor Sharing (PS) basis using the time slicing option given by the compiler. The Ada tasks communicated using the rendezvous primitive.

In Ada, each task entry has a first-come-first-served queue, but there is no guarantee that callers will be served in a first-come-first-served manner across entries of the same task. To overcome this, tasks that would normally be encoded with multiple entries are encoded with a single entry that has an operation code as a parameter. The service that is provided to the caller is based on this operation code. In this way each task has a first-come-first-served queue but still provides the services the caller requires. The Ada programs are given in appendix D.

6.2.1 Example 1: A Transaction Processing Ada Application

In this first example, an application is presented that permits a fixed number of users to initiate transactions from their keyboards. Transactions are defined here as sequences of operations that require both local and remote processing. Each user is associated with a Customer task; a pool of Agent tasks is used to manage the remote aspects of the processing. A user initiates a transaction which causes a Customer task to synchronize with an Agent task. While synchronized, the transaction's data is initialized by retrieving information from a disk, and then some information is written to a logging disk. Afterwards, the Customer and Agent tasks proceed independently. The Customer task performs some local work on behalf of the transaction and updates the user's display. When the remote portion of the transaction completes, the Agent updates the user's display, and more information is written to a logging disk. It is not defined whether the local or remote processing will be the first to complete and/or update the display. When the Customer task completes, the user is permitted to initiate another transaction.

The software model of the application is shown in figure 6.2. Users think and request transactions. Once a transaction is requested, the user's corresponding Customer task attempts to synchronize with an Agent task at the InitRemote task.

Once the Customer and Agent tasks synchronize, transaction specific information is loaded by the InitRemote task from a transaction description disk (DECODISK) into the Customer and Agent tasks' local data areas. The Log task is called by the InitRemote task so that some transaction information can be written to disk. The Customer and Agent tasks are then released by the InitRemote task. The Customer task performs some local processing on behalf of the transaction and updates the customer's display via the Display task. Afterwards, the user is free to initiate another transaction.

Once an Agent task is released from the InitRemote task, it has acquired a transaction. The transaction is forwarded to a remote site via the COMPORT and the Agent waits for information to be sent back. The amount of information is small so the communication time is very low. Once this is done the Agent calls the Display task. The Display task updates the appropriate user's display and then rendezvous with the Log task. The Log task writes the information to disk. Once the rendezvous with the Display task completes, the Agent's portion of the transaction is complete. The Agent visits the InitRemote task looking for another transaction.

The Display task uses both the processor and TERMPORT. Only one customer display can be updated at a time. The Log task uses the processor and provides mutual exclusion over the LOGDISK.

The application's model parameters are given in table 6.1. There are three users and hence three Customer tasks in the system. The devices COMPORT, TERMPORT, DECODISK, and LOGDISK are each accessed by one task so there are no queueing delays. The service demands to these devices are represented as a think time in the model. The system and model are altered by increasing the number of Agent tasks from 1 to 5.

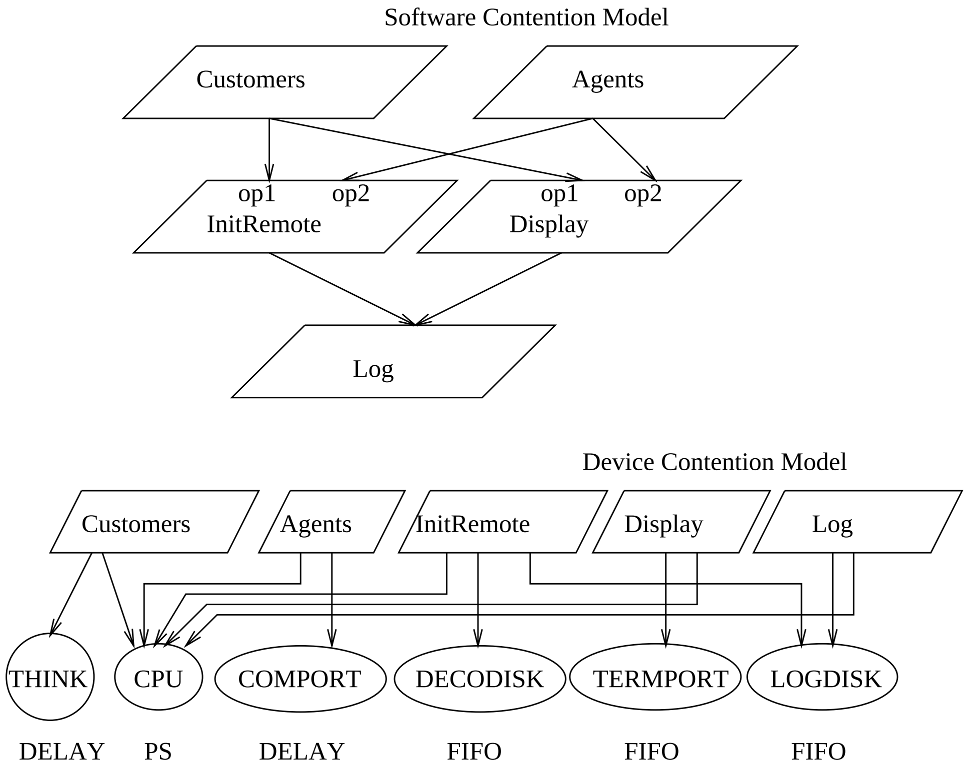


Figure 6.2: Transaction Processing Application.

Entity	Population	V_{Dop1}	V_{Dop2}	V_{IR}	V_{Log}	D_{CPU}	D_{Think}
Customer	3	1	-	1	-	20	4
Agent	Varies	-	1	1	-	16	-
Display E1 (Dop1)	1	-	-	-	-	6	-
Display E2 (Dop2)	-	-	-	-	1	4	-
InitRemote (IR)	1	-	-	-	1	4	-
Log	1	-	-	-	-	2	-

Entity	COMPORT	TERMPORT	DECODISK	LOGDISK
Customer	-	-	-	-
Agent	8	-	-	-
Display E1 (Dop1)	-	3	-	-
Display E2 (Dop2)	-	3	-	-
InitRemote (IR)	-	-	3	-
Log	-	-	-	3

Table 6.1: a) Service Demands Per Invocation for Transaction Example.

Enitiy Name	Scheduling Discipline
CPU	Processor Sharing (PS)
Think	Delay (DELAY)
COMPORT	Communications Port (DELAY)
TERMPORT	Terminal Display Port (FIFO)
DECODISK	Decoding Codes Disk (FIFO)
LOGDISK	Log Disk (FIFO)
Customer	Non-Serving
Agent	Non-Serving
Display	Multiple-Entry Server
InitRemote	SYNC Server
Log	Rendezvous Server

Table 6.1: b) Entity Descriptions for Transaction Example.

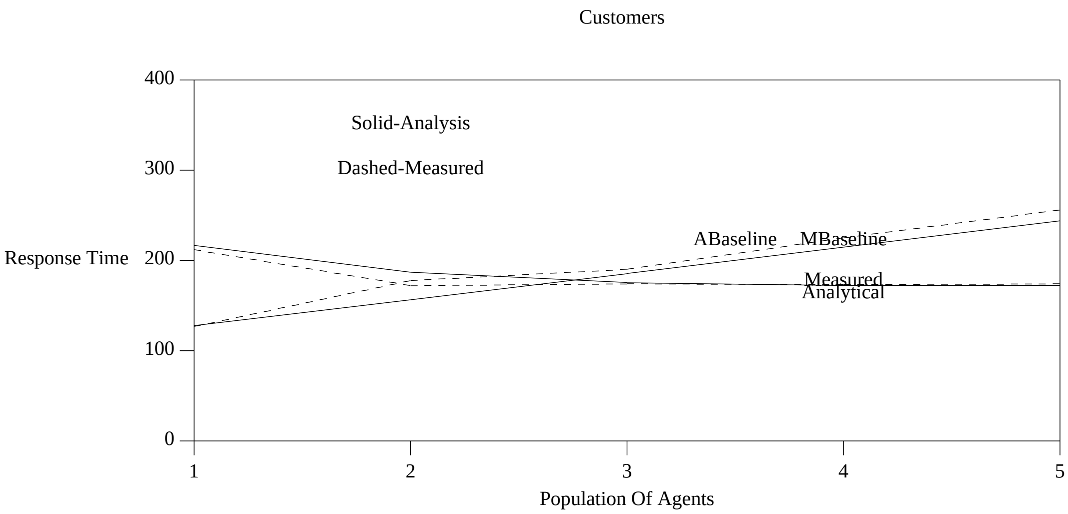


Figure 6.3: Example 1: Customer Response Time Versus Agent Population for Transaction Example.

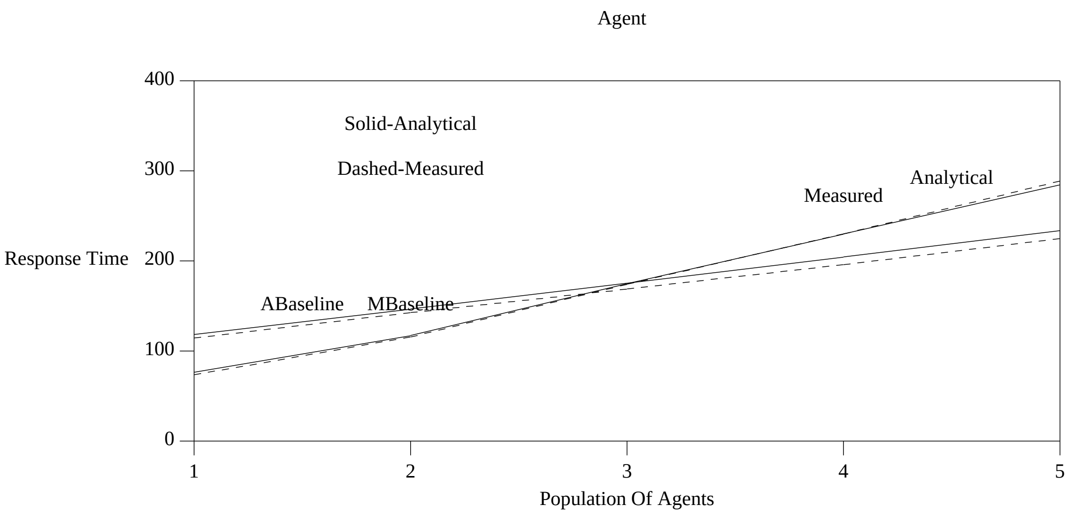


Figure 6.4: Example 1: Agent Response Time Versus Agent Population for Transaction Example.

The software model of the baseline version of the application is shown in figure 6.5. The baseline version of the application's model parameters are shown in table 6.5. The devices TERMPORT, DECODISK, and LOGDISK, remain as FIFO servers. COMPORT continues to be represented as a DELAY server. In this example, there is no operating system or hardware enforced mutual exclusion available for these devices. Ada tasks are used to provide mutual exclusion.

In figure 6.3 and 6.4, the predicted average response time estimates in time units for the prototypes are shown along with the measured values. A straight application of mean value analysis predicts the performance of the baseline system very well.

When compared with the measured values for non-baseline tests, it is clear that the task synchronization has a dramatic affect upon Agent and Customer responsiveness. Increasing the number of Agent tasks decreases the average time a Customer task has to wait to synchronize at the InitRemote task. The analytical techniques predict the effects.

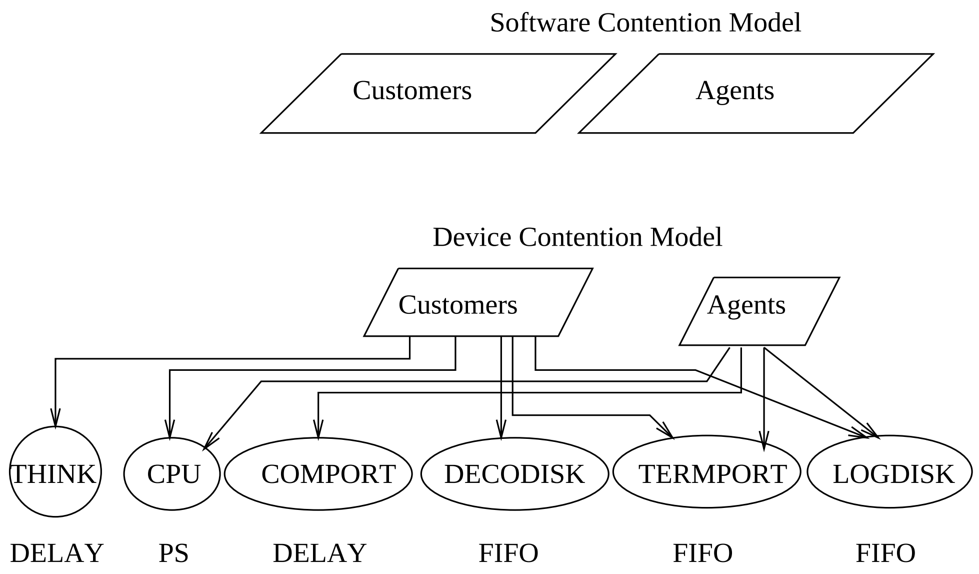


Figure 6.5: Transaction Processing Baseline Application.

Entity	Population	D_{CPU}	D_{Think}	COMPORT	TERMPORT	DECODISK	LOGDISK
Customer	3	30	4	-	3	3	3
Agent	Varies	30	-	8	3	3	6

Table 6.2: a) Service Demands Per Invocation for Transaction Baseline Example.

Enitiy Name	Scheduling Discipline
CPU	Processor Sharing (PS)
Think	Delay (DELAY)
COMPORT	Communications Port (DELAY)
TERMPORT	Terminal Display Port (FIFO)
DECODISK	Decoding Codes Disk (FIFO)
LOGDISK	Log Disk (FIFO)
Customer	Non-Serving
Agent	Non-Serving

Table 6.2: b) Entity Descriptions for Transaction Baseline Example.

6.2.2 Example 2: An Avionics Application

A software process architecture has been taken from a paper in Ada Letters [Locke 88] that describes a standard avionics program. The application is an example of a real-time problem called "sensor fusion." Information from functionally different sensors is combined using a correlator algorithm to create an integrated view of an external environment. The external view can be stored in a table, and then referenced by those tasks that need to "observe" the external environment.

For example, information from a variety of sensors may be used to monitor the movement of aircraft or "tracks." A correlator might associate the information from several sensors with a specific aircraft and store it in a track table. Observer tasks might then view the tracks, display data, and warn of unsafe conditions.

An Ada program has been written that has the software process architecture described in the paper. The software process architecture is shown in figure 6.6. Processing times have been chosen and assigned to the different Ada tasks in the program, and the program has been executed. The application's model parameters are shown in table 6.3. The number of sensors in the model is changed from 2 to 20 to determine the effects upon the application's client tasks.

The baseline model's parameters are shown in table 6.4. The number of sensors in the model is changed from 2 to 20 to determine the effects upon the application's client tasks.

In figures 6.7 through 6.10, the analytical performance estimates for average process response times in units for non-serving processes are compared with the measured values for the Ada application.

The analytical performance estimates match the measured values well. The worst error in an average response time estimate is 20% and the largest difference in measured average response time between original and baseline cases is 130%. The original and baseline cases have significantly different performance behaviour. This is most clear when observing the average response time curves for the Data and Display tasks. The analytical techniques are successful in predicting the difference in behaviour.

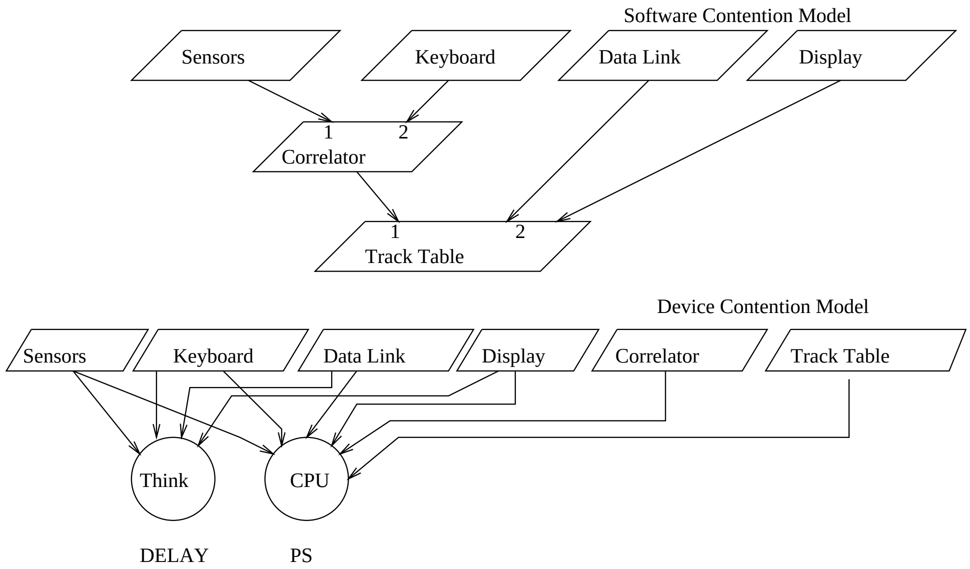


Figure 6.6: An Avionics Application's Software Architecture.

Entity	Population	V_{CorE1}	V_{CorE2}	V_{TTaE1}	V_{TTaE2}	D_{CPU}	D_{Think}
Sensor	Varies	1	-	-	-	6	100
Keyboard	1	-	1	-	-	6	100
DataLink	1	-	-	-	1	3	80
Display	1	-	-	-	1	15	70
Correlator E1 (CorE1)	1	-	-	1	-	4	-
Correlator E2 (CorE2)	-	-	-	1	-	16	-
Track Table E1 (TTaE1)	1	-	-	-	-	2	-
Track Table E2 (TTaE2)	-	-	-	-	-	3	-

Table 6.3: a) Service Demands Per Invocation for Avionics Example.

Enitiy Name	Scheduling Discipline
CPU	Processor Sharing (PS)
Think	Delay (DELAY)
Sensor	Non-Serving
Keyboard	Non-Serving
DataLink	Non-Serving
Display	Non-Serving
Correlator (CorE1 and CorE2)	Multiple-Entry Server
Track Table (TTaE1 and TTaE2)	Multiple-Entry Server

Table 6.3: b) Entity Descriptions of Avionics Example.

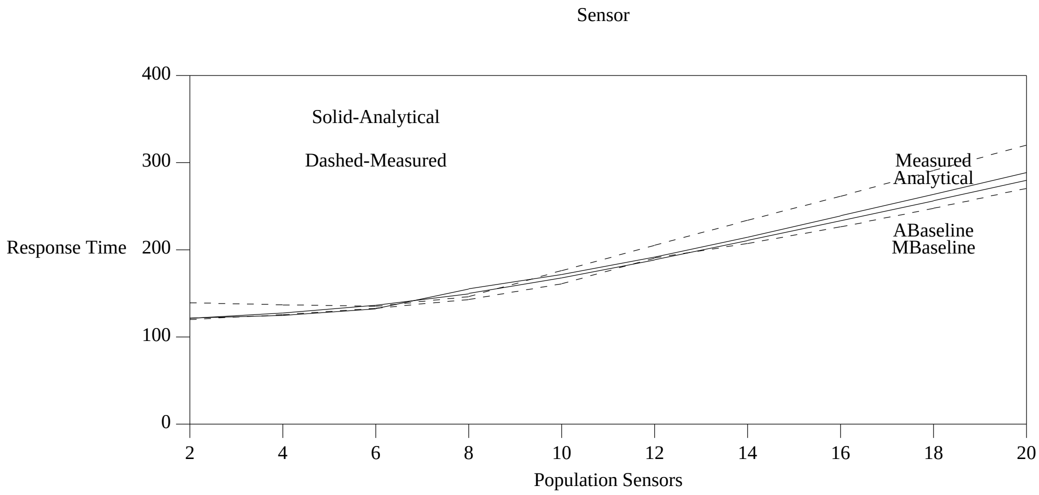


Figure 6.7: Example 2: Sensor Response Time Versus Sensor Population for the Avionics Example.

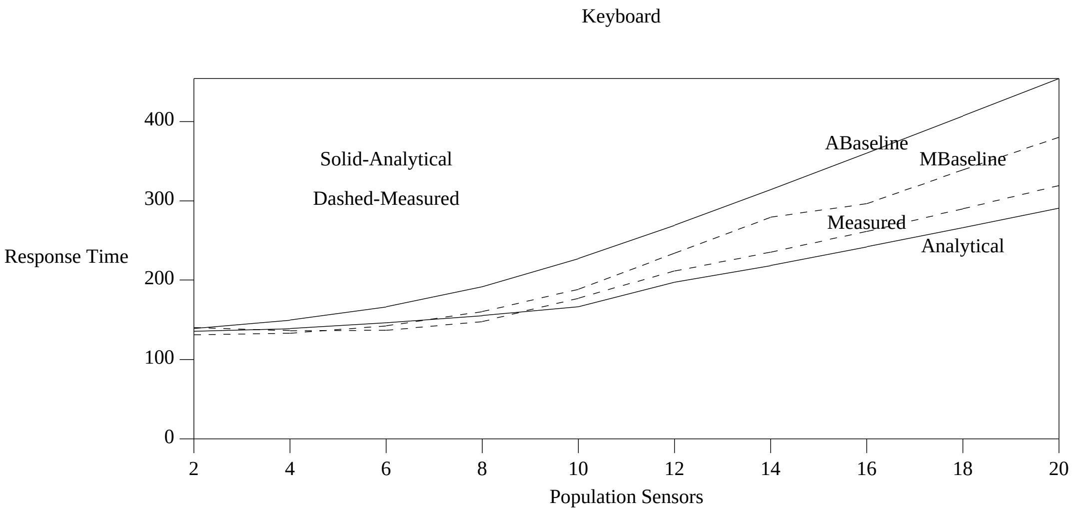


Figure 6.8: Example 2: Keyboard Response Time Versus Sensor Population for the Avionics Example.

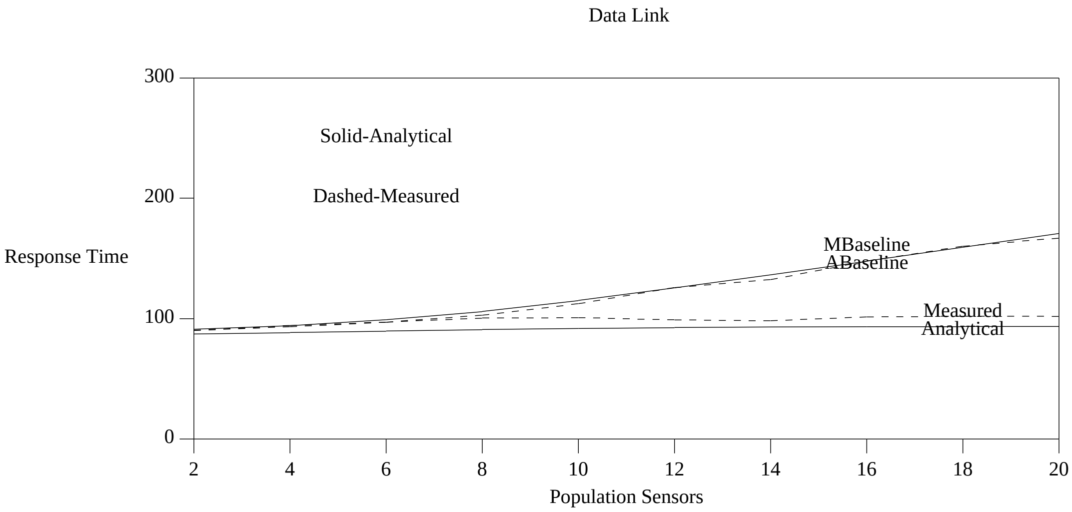


Figure 6.9: Example 2: Data Link Response Time Versus Sensor Population for the Avionics Example.

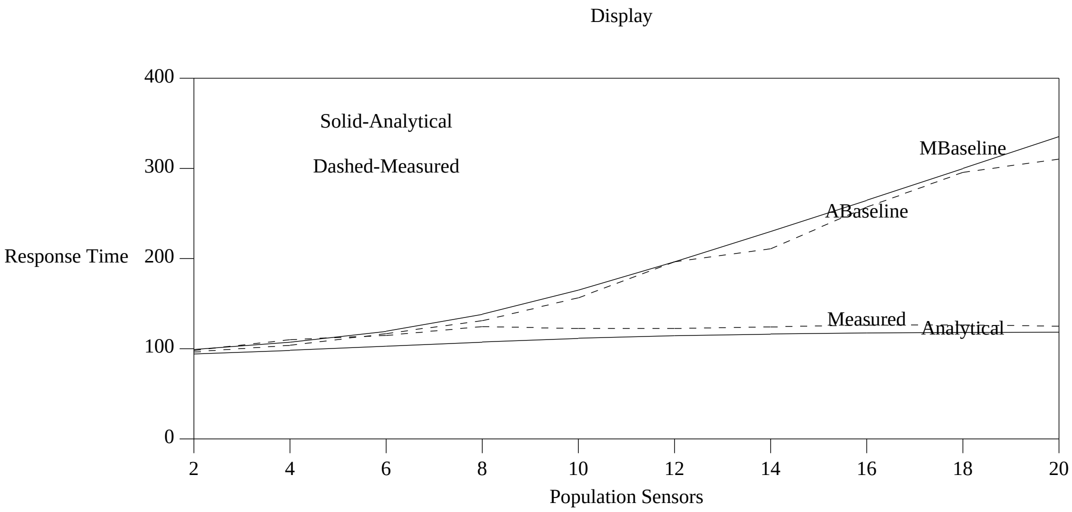


Figure 6.10: Example 2: Display Response Time Versus Sensor Population for the Avionics Example.

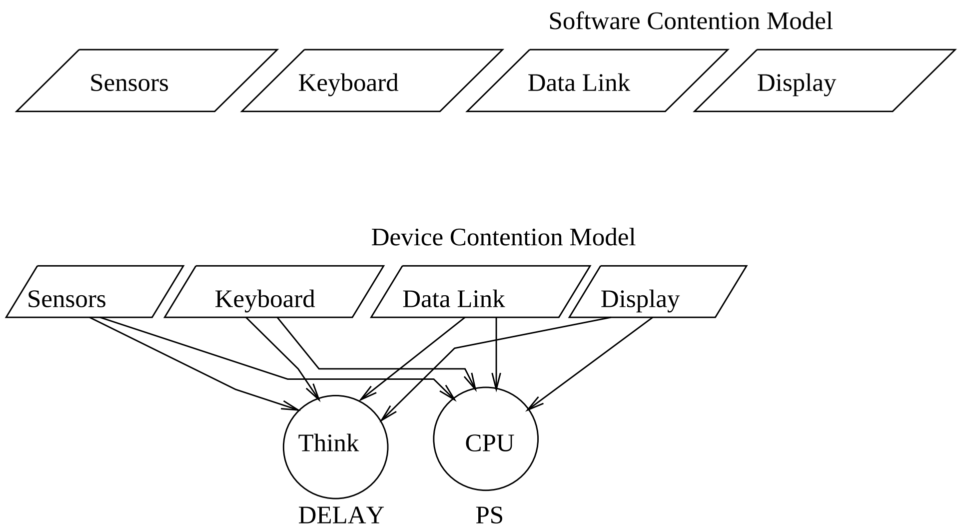


Figure 6.11: The Avionics Example's Baseline Software Architecture.

Entity	Population	D_{CPU}	D_{Think}
Sensor	Varies	12	100
Keyboard	1	24	100
DataLink	1	6	80
Display	1	18	70

Table 6.4: a) Service Demands Per Invocation of Avionics Baseline.

Enitiy Name	Scheduling Discipline
CPU	Processor Sharing (PS)
Think	Delay (DELAY)
Sensor	Non-Serving
Keyboard	Non-Serving
DataLink	Non-Serving
Display	Non-Serving

Table 6.4: b) Entity Descriptions of Avionics Baseline.

6.3 Conclusions

Software performance engineering techniques can be used to help develop and manage the performance of complex software systems. However, a set of tools are required to facilitate the engineering process. Performance modelling and monitoring tools are fundamental to this process.

The techniques presented in this thesis can be used to predict the performance of some multi-tasking systems. In section 6.2 the performance of two Ada applications were modelled successfully using the MOL. Such modelling techniques can play a part in a software performance engineering process.

A method has been proposed that describes how performance requirements can be specified for multi-tasking Ada applications. The feasibility of these requirements can be verified at design time using predictive modelling techniques. Once the system is built, monitoring can take place and be used to ensure that the requirements are indeed satisfied. The type of monitoring data that needs to be collected is also described.

Software performance engineering [Smith 90] can be used to help find problems early in an application's development. For example:

- Infeasible requirements can be found at design time.
- The performance of alternative designs can be investigated.
- Unsatisfied performance requirements can be recognized as the services are implemented.
- The execution state scenarios that result in poor performance can be recognized at design and execution time.

The first step towards solving performance problems is recognizing that they exist. Without a software performance engineering methodology, ensuring that response time goals are met is difficult, and determining the effects caused by successive system changes is even more difficult.

Chapter 7

Conclusions

The purpose of this thesis has been to develop analytical performance modelling techniques that can be used to study software systems. A summary of the results, observations, and a description of possible future research are provided in the following sections.

7.1 Summary

A technique called the Method of Layers has been developed that can be used to study software systems in which a fixed size set of processes can act as both customers and servers while sharing devices. The method is based on the Linearizer Mean Value Analysis algorithm. The residence time expressions commonly available for use with MVA are not adequate for describing the relationships between processes that exist in software process architectures. Several new residence time expressions have been developed that can be used in conjunction with the MOL to describe some of the possible interactions. Each of the interactions is represented in a model as a process requesting service from a type of serving process.

The types of servers considered are:

- a rendezvous server that provides rendezvous and post-rendezvous service;
- a multiple-entry server that provides rendezvous and post-rendezvous service to

callers at several entries (customers at the server are served in a first-come-first-served manner);

- a multi-server at which serving processes share a single queue of calling processes;
- a synchronization server at which processes can synchronize;
  - a producer-consumer server is also shown that is composed of two synchronization servers and represents condition synchronization (a producer-consumer relationship).

Each of the analytical techniques for the servers has been validated in isolation. The residence time expressions have been studied using 468 models. An additional 157 models have been examined that contain combinations of the residence time expressions presented in this thesis. Thirty of these models were validated with respect to the measured performance of software application prototypes written in Ada. In total, 605 models were considered.

The relationship between the MOL and Generalized Stochastic Petri Nets has also been considered. A class of GSPNs has been described that can be solved using the MOL.

Finally, the performance engineering of Ada applications has been discussed. An approach was suggested for describing the performance requirements of multi-tasking Ada applications, performance models for Ada applications, and a strategy for collecting information about the performance of applications as they execute. The three are linked together through the notion of abstract services that an Ada application provides, and the relationships amongst the services. The use of services permits information specified at one stage of a system's development to be easily used at subsequent stages. This is necessary for a software engineering methodology to be effective.

7.2 Observations

Analytical performance modelling tools based on mean value analysis can provide fast performance estimates for software models. The accuracy of the predictions appear to be quite good despite the fact that the models do not, in general, satisfy separability constraints.

The techniques developed in this thesis can be used for a rapid study of the performance behaviour of software process architectures under many different circumstances. Without such tools, design decisions must be based on personal intuition. It is difficult to predict the performance behaviour of applications without the assistance of such tools. The tools can be used to narrow design spaces and to perform *what if* type analysis.

Once a software process architecture is chosen and its parameters have been properly estimated, more detailed and accurate simulation models or performance prototypes can then be developed. The resources required to obtain performance estimates using these approaches is much higher than for the analytical techniques. Though it is still possible to do *what if* type analysis, the likelihood of doing so and benefiting from the exercise decreases with the time and effort required to get results.

The compiler industry could further its role in supporting the efforts of software engineers to construct efficient software process architectures. The way it can do this is to assist in the collection of meaningful performance information about applications while they execute. It is often difficult to relate an application's system resource consumption with specific parts of its code. Yet this is necessary to help construct performance models, show that performance requirements are being statisfied, and to isolate performance problems. Operating system tools do not usually have visibility into a program's internal state information. Support for such monitoring should be part of what compilers offer, especially for multi-tasking languages with runtime systems. Without such support it is difficult to show that applications are functioning as expected.

7.3 Future Research

Currently, the MOL can be applied to a fixed set of processes with relationships that can be described using the types of servers listed in the summary. Residence time expressions should be developed to consider the following.

- Two-phase multi-servers and synchronization servers with high service time variation and many calling groups per entry. Such work requires the computation of each process's idle time distribution and its interarrival time distribution at each of its servers.
- Conditional rendezvous, in which customers cancel their calls if they arrive at a busy server.
- An Ada style multiple entry server would also be useful. Unfortunately the order in which an Ada program accepts callers within a selective accept statement is undefined. Alternatives in choosing which accept statement will get to accept a caller include round-robin (move from entry to entry so there is no starvation of any caller), and first entry declared to last entry declared (gives high priority to the first entry). Each of these should be studied.
- Extending the producer-consumer server to consider interarrival time variance at each of the SYNC servers.

The effects of guarded conditions on an application's performance behaviour must also be considered in detail. It is not clear how such guards can be expressed in a model. Producer-consumer guarded conditions have a fairly straightforward relationship with synchronization servers and have been considered in this thesis, but more complex conditions are likely to exist. A method is needed to describe the conditions within an analytical modelling framework. Similarly, the fixed ordered acceptance of callers at servers must be studied.

Not all interactions can be well represented using residence time expressions. In particular, software interactions can constrain the set of processes that compete for servers at any particular time. These effects should be considered. A method based on the decomposition approximation [Heid 83] should be combined with the MOL to consider such behaviour. This would also permit the removal of the limitation that a fixed size set of processes compete for resources. In Ada this would correspond to a program in which the number of master scopes and the contention generated by their tasks varies and affects the program's performance behaviour.

Bibliography

[Agra 83] S.C. Agrawal and J.P. Buzen, The Aggregate Server Method for Analyzing Serialization Delays, *ACM Transactions on Computer Systems,* Volume 1, Number 2, May 1983, 116-143.

[Aho 83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. Data Structures and Algorithms. Addison-Wesley 1983.

[AndSh 83] G. R. Andrews and F. B. Schneider. Concepts and Notations for Concurrent Programming. *Computing Surveys*, Volume 15, Number 1, March 1983, 3-43.

[Balbo 86] G. Balbo, S Bruell and S. Ghanta. Combining Queueing Network and Generalized Petri Net Models for the Analysis of Some Software Blocking Phenomena. *IEEE Tranactions on Software Eng.,* April 1986, 561-576.

[Bard 79] Y. Bard. Some Extensions to Multiclass Queueing Network Analysis. In M.Arato, A. Butrimenko, and E. Gelenbe (eds), *Performance of Computer Systems.* North-Holland, 1979.

[Bask 75] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios, Open, Closed, and Mixed Networks of Queues with Different Classes of Customers, *JACM*, Volume 22, Number 2, April 1975, pages 248-260.

[Beizer 78] B. Beizer, *Micro-Analysis of Computer System Performance*, Von Nostrand Reinhold, New York, 1978.

[Beizer 84] B. Beizer. Software Performance. In C.R. Vicks and C.V. Ramamoorthy (eds), *Handbook of Software Engineering.* Von Nostrand Reinhold, New York, 1984, pages 413-436.

[BGS 82] Best/1 *User's Guide.* BGS Systems, Inc., Waltham, MA, 1982.

[Booth 86] Taylor L. Booth, Richard O. Hart, and Bin Qin. High Performance Software Design. *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences,* 1986, 41-52.

[Brinch73] P. Brinch Hansen. Concurrent Programming Concepts. *ACM Computer Surveys*, Volume 5, Number 4, December 1973, pages 223-245.

[Brinch75] P. Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, Volume SE-1, Number 2, June 1975, pages 199-206.

[Bryant 84] R.M. Bryant, A.E. Krzesinski, M.S. Lakshmi, and K.M. Chandy. The MVA Priority Approximation, *ACM Trans. On Comp. Systems,* Volume 2., Number 4, November 1984, 335-359.

[Buhr 83] R. Buhr. *Systems Design Using ADA.* Prentice-Hall 1983.

[Buhr 89] R.J. Buhr, G.M. Karam, C.J. Hayes, and C.M. Woodside. Software CAD: A Revolutionary Approach, *IEEE Transactions on Software Engineering,* Volume 15, Number 3, March 89, 235-249.

[Burns 87] A. Burns, A. M. Lister, and A. J. Wellings. A Review of Ada Tasking. *Lecture Notes in Computer Science,* Ed. G. Goos and J. Hartmanis, 262, Springer-Verlag.

[Buzen 73] J.P. Buzen. Computation Algorithms for Closed Queueing Networks with Exponential Servers. *CACM,* Vol. 16, September 1973, 527-531.

[Chandy 75] K.M Chandy, U. Herzog, and L. Woo. Approximate Analysis of General Queueing Networks. *IBM J. Res. and Develop., 19,* 1 (Jan. 1975), 43-49.

[Chandy 77] K.M Chandy, J.H. Howard Jr., and D.F. Towsley. Product Form and Local Balance in Queueing Networks. *JACM,* April 1977 250-263.

[Chandy 82] K.M. Chandy and D. Neuse. A Heuristic Algorithm for Queueing Network Models of Computing Systems. *CACM,* February 1982, 126-133.

[Ciardo 87] G. Ciardo. Toward a Definition of Modeling Power for Stochastic Petri Net Models. *International Workshop On Petri Nets and Performance Models,* Madison, Wisconson, USA, August 1987, 54-62.

[Cox 55] D.R Cox. A Use of Complex Probabilities in Theory of Stochastic Processes. *Proc. Cambridge Phil. Soc.,* 51, 313-319.

[DOD 83] U.S. Department of Defence, *Reference Manual for the Ada Programming Language,* MIL-STD-1815a, 1983.

[Eager 88] D.L. Eager and J.N. Lipscomb. The AMVA Priority Approximation. *Performance Evaluation,* Volume 8, Number 3, June 1988, 177-193.

[Geh 88] N. H. Gehani and W. D. Roome. Rendezvous Facilities: Concurrent C and the Ada Language. *IEEE Transactions on Software Engineering,* Volume 14, Number 11, November 1988, 1546-1553.

[Hab90a] D. Haban and D. Wybranietz. A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems, *IEEE Transactions on Software Engineering,* Volume 16, Number 2, February 1990, 197-211.

[Hab90b] D. Haban and K.G. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times, *IEEE Transactions on Software Engineering,* Volume 16, Number 12, December 1990, 1374-1389.

[Heid 82] P. Heidelberger and K.S. Trivedi. Queueing Network Models for Parallel Processing with Asynchronous Tasks. *IEEE Transactions on Computers,* November 1982, 1099-1109.

[Heid 83] P. Heidelberger and K.S. Trivedi. Analytic Queueing Models for Programs with Internal Concurrency. *IEEE Transactions on Computers,* January 1983, 73-82.

[Hoare 78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, Volume 2, Number 8, August 1978, pages 666-677.

[Holt 78] R.C. Holt, E.D. Lazowska, G.S. Graham, and M.A. Scott, *Structured Concurrent Programming with Operating Systems Applications.* Addison-Wesley 1978.

[Howes 89] N.R. Howes and A. C. Weaver, Measurements of Ada Overhead in OSI-Style Communication Systems, *IEEE Transactions on Software Engineering*, Volume 15, Number 12, December, 1989 1507-1517.

[Jacks 57] J.R. Jackson. Networks of Waiting Lines. *Operations Research,* 5, 1957, 518-521.

[Jacob 82] P.A. Jacobson and E.D. Lazowska. Analysing Queueing Networks with Simultaneous Resource Possession. *CACM,* February 1982, 141-152.

[Kant 87] K. Kant. Modeling Interprocess Communication In Distributed Programs. *International Workshop on Petri Nets and Performance Models,* Madison WI, USA, August 1987, 75-83.

[Klein 75] L. Kleinrock. *Queueing Systems, Volume 1: Theory.* John Wiley & Sons, 1975.

[Klein 76] L. Kleinrock. *Queueing Systems, Volume 2: Computer Aplications.* John Wiley & Sons, 1976.

[Laven 89] S.S. Lavenberg. A Perspective on Queueing Models of Computer Performance, *Performance Evaluation,* Volume 10, Number 1, 1989, 53-76.

[Lazow 84] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, 1984.

[Liu 73] Liu, C.L. and Layland, J. W. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment *Journal for the Association of Computing Machinery,* Volume 20, Number 1 (1973), 46-61.

[Locke 88] Locke, C. D. and Goodenough, J. B. A Practical Application of the Ceiling Protocol in a Real-Time System, *Ada Letters, 1988 Special Edition*, Volume 8, Number 7, 35-38.

[QSP 82] MAP *User's Guide.* Quantitative System Performance, Inc., Seattle, WA, 1982.

[Marsan 84] M. Ajmone-Marsan, G. Balbo, and G. Conte, A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multi-Processor Systems, *ACM Trans. Comput. Systems*, Volume 2, Number 2, May 1984, 93-122.

[Mier 88] J.W. Miernik, C.M. Woodside, J.E. Neilson, and D.C. Petriu. Throughput of Stochastic Rendezvous Networks with Caller-Specific Service and Processor Contention. *Proceedings of IEEE InfoCom,* 1988, 1040-1049.

[Mier 89] J.W. Miernik, C.M. Woodside, J.E. Neilson, and D.C. Petriu. *Performance of Stochastic Rendezvous Networks with Priority Tasks.* Technical Report. Carleton University, SCE-89-02. Presented at the International Seminar on Performance of Distributed and Parallel Systems, December 1988, Kyoto, Japan.

[Molloy 81] M.K. Molloy. *On the Integration of Delay and Throughput Measures in Distributed Processing Models.* Phd. Thesis, University of California, Los Angeles, September 1981.

[Molloy 82] M.K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE Transactions On Computers,* Volume C-31, Number 9, September 1982, 913-917.

[Molloy 87] M.K. Molloy. Structurally Bounded Stochastic Petri Nets. *International Workshop on Petri Nets and Performance Models,* Madison WI, USA, August 1987, 156-163.

[Murata 84] Tadao Murata. Modeling and Analysis of Concurrent Systems, *Handbook of Software Engineering,* ed. C.R. Vick and C.V. Ramamoorthy, Van Nostrand Reinhold 1984, 39-63.

[Petri 66] C.A. Petri. *Communication with Automata.* PhD Thesis, Traslated by C.F. Green, Information System Theory Project, Applied Data Research Inc., Princeton N.J., USA, 1966.

[Qin 86] Bin Qin. A Model to Predict the Average Response Time of User Programs. *Performance Evaluation,* 10, 1989, 93-101.

[Ramam 80] C.V. Ramamoorthy and Gary S. Ho. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transactions on Software Engineering,* Volume SE-6, Number 5, September 1980, 440-449.

[Reiser 78] M. Reiser and S.S. Lavenburg. *Mean Value Analysis of Closed Multichain Queueing Networks.* IBM Research Report RC 70 23, Yorktown Heights, N.Y., 1978.

[Reiser 79] M. Reiser. A Queueing Network Analysis of Computer Communication Networks with Window Flow Control. *IEEE Transactions On Communications,* August 1979, 1201-1209.

[Rolia 87] J.A. Rolia. *Performance Estimates For Multi-Tasking Software Systems,* Master's Thesis, University of Toronto, Canada, January 1987.

[Rolia 88] J.A. Rolia. Performance Estimates for Systems with Software Servers: The Lazy Boss Method. *VIII SCCC International Conference On Computer Science,* Santiago, Chile, July 1988, 25-43.

[Sevcik 77a] K.C. Sevcik, A.I. Levy, S.K. Tripathi, and J. Zahorjan. Improving Approximations of Aggregated Queueing Network Subsystems. *Computer Performance.* North-Holland, 1977.

[Sevcik 77b] K.C. Sevcik. Priority Scheduling disciplines in queueing network models of computer systems. In *Proceedings of the IFIP Congress 77*, Toronto, Canada (1977), 565-570.

[Sevcik 81] K.C. Sevcik and I. Mitrani. The Distribution of Queueing Network States at Input and Output Instants. *JACM,* April 1981, 358-371.

[Shatz 88] S.M. Shatz and W.K. Cheng. A Petri-Net Framework for Automated Static Analysis of Ada Tasking Behaviour. *Journal of Systems and Software,* Volume 8, December 1988, 343-359.

[Schwe 79] P. Schweitzer. Approximate Analysis of Multiclass Closed Networks of Queues. *Proc. International Conference on Stochastic Control and Optimization*, Amsterdam, 1979.

[Silva 90] E. de Souza e Silva, R.R. Muntz. A Note on the Computational Cost of the Linearizer Algorithm for Queueing Networks, *IEEE Transactions on Computers,* Volume 39, Number 6, June 1990, 840-842.

[Smith 85] C.U. Smith. *Performance Models for Software/Hardware Codesign,* Technical Report CS-1-1985, Duke University, Durham NC, USA, January 1985.

[Smith 90] C. U. Smith, *Performance Engineering of Software Systems.* Addison Wesley, 1990.

[Tanen 87] A.S. Tanenbaum, *Operating Systems Design and Implementation.* Prentice-Hall 1987.

[Triv 82] K.S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications.* Prentice-Hall 1982.

[Vernon 86] M. Vernon, J. Zahorjan and E.D. Lazowska. *A Comparison of Performance Petri Nets and Queueing Network Models,* Technical Report, University of Washington, Seattle, USA, September 1986.

[Wegner 83] P. Wegner and S.A. Smolka. Processes, tasks, and monitors: a comparative study of concurrent programming primitives. *IEEE Transactions on Software Engineering*, Volume SE-9, Number 4, July 1983, 446-462.

[Woods 86] C.M. Woodside. *Throughput Calculation For Basic Stochastic Rendezvous Networks,* Technical Report, Carleton University, Ottawa, Canada, April 1986.

[Woods 89] C.M. Woodside. Throughput Calculation For Basic Stochastic Rendezvous Networks. *Performance Evaluation,* Volume 9., 1989.

[Woods 91] C.M. Woodside, E.M. Hagos, E. Neron, R.J.A. Buhr. The CAEDE Performance Analysis Tool, *Ada Letters,* Volume XI, Number 3, Spring 1991.

[Zahor 88] J. Zahorjan, D.L. Eager, and H. Sweillam. Accuracy, Speed, and Convergence of Approximate Mean Value Analysis. *Performance Evaluation,* Volume 8, Number 4, August 1988, 255-270.

[Zahor 86] J. Zahorjan, E.D. Lazowska and K.C. Sevcik. The Use of Approximations in Production Performance Evaluation Software. *Proc. Performance Modelling Tools and Techniques.* 1986.

Appendix A

MOL Example

In this appendix a model is presented and its performance is predicted using the MOL. The model that is used has three layers of groups and two devices and is shown in figure A.1. One of the devices is a DELAY device and the other has the PS scheduling discipline. Model parameters are given in table A.1. The group at layer three has a population of three.

In table A.2, a summary of estimated average response times is given for the affected groups for each submodel that is solved. Column INIT contains the initialized group response times at the start of the MOL, after a device contention model has been solved, or when reporting the final results. In columns $l = 3$, the second submodel is being solved. In column DEV, the device contention submodel is being solved. After the last solution of the software contention model with $l = 2$, the response times estimates have converged for R_{TOP} so the algorithm terminates. The final estimates for average group response times are given in the last row of results in the INIT column.

Finally, the intermediate results for the MOL are given for an iteration of the device contention loop. The reported data lists the values of parameters and results for each application of MVA in the iteration.

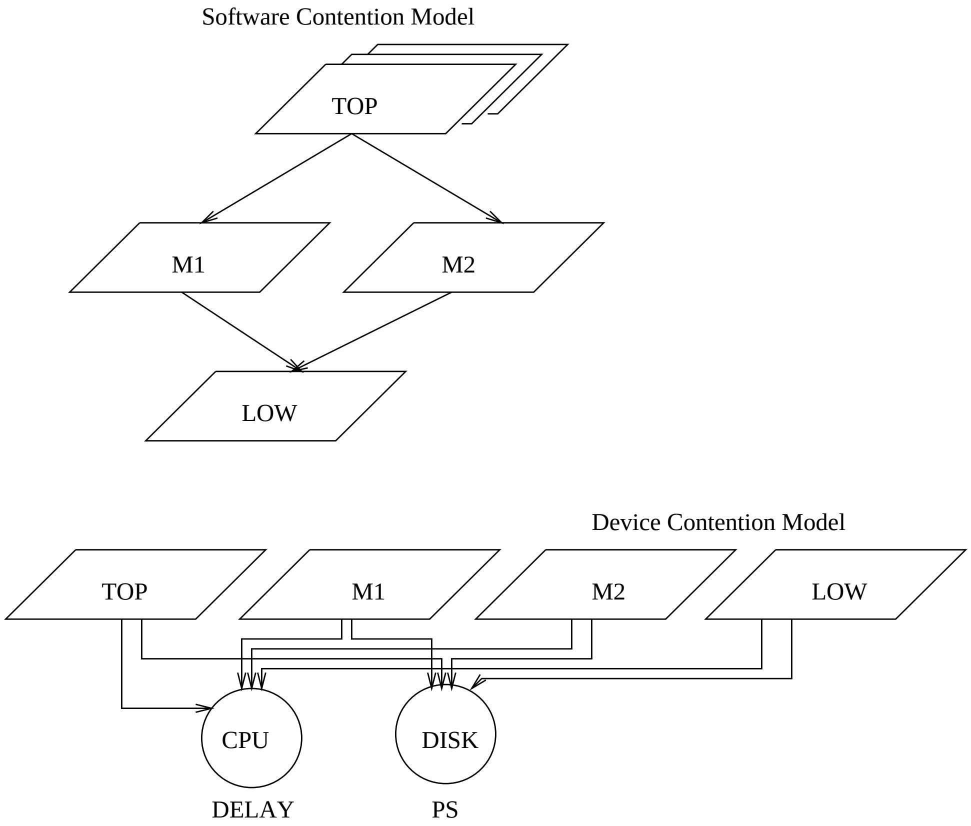


Figure A.1: MOL Example

Entity	Pop	V_{M1}	V_{M2}	V_{LOW}	V_{DCPU}	V_{DISK}	S_{DCPU}	S_{DISK}
TOP	3	2	2	-	6	1	1	0.5
M1	1	-	-	1	3	1	1	0.5
M2	1	-	-	1	1	1	1	0.5
LOW	1	-	-	-	2	1	1	0.5

Table A.1: a) Task Service Demands per Invocation.

Processor Name	Scheduling Discipline
DCPU	Delay Server
DISK	PS

Table A.1: b) Processor Descriptions.

Entity	Task Scheduling Discipline
TOP	Non-Serving
M1	FCFS Server
M2	FCFS Server
BOT	FCFS Server

Table A.1: c) Task Scheduling Discipline.

Dev Iter	Group	INIT	$l = 3$	$l = 2$	$l = 3$	$l = 2$	$l = 3$	$l = 2$	$l = 3$	$l = 2$	DEV
1	R_{BOT}	2.5									2.57
1	R_{M1}	6.0		6.47		6.42		6.46		6.46	6.17
1	R_{M2}	6.0		6.93		6.92		6.92		6.92	6.19
1	R_{TOP}	24.5	40.39		43.9		43.88		43.88		25.17
2	R_{BOT}	2.57									2.57
2	R_{M1}	6.17		6.65		6.64		6.64		6.64	6.18
2	R_{M2}	6.19		7.15		7.14		7.14		7.14	6.16
2	R_{TOP}	25.2	41.52		45.20		45.13		45.13		25.15
3	R_{BOT}	2.57									-
3	R_{M1}	6.18		6.65		6.64		6.64		6.64	-
3	R_{M2}	6.16		7.14		7.13		7.13		7.14	-
3	R_{TOP}	25.2	41.51		45.17		45.10		45.10		-
final	R_{BOT}	2.57	-	-	-	-	-	-	-	-	-
final	R_{M1}	6.64	-	-	-	-	-	-	-	-	-
final	R_{M2}	7.14	-	-	-	-	-	-	-	-	-
final	R_{TOP}	45.10	-	-	-	-	-	-	-	-	-

Table A.2: R_g Intermediate Results

BEGIN Method of Layers
Step 2.
Find R^{DEV}:

- $R^{DEV}_{BOT} = 2.5$
- $R^{DEV}_{M1} = 3.5$
- $R^{DEV}_{M2} = 3.5$
- $R^{DEV}_{TOP} = 6.5$

Find R^{GRP}:

- $R^{GRP}_{BOT} = 0$
- $R^{GRP}_{M1} = 2.5$
- $R^{GRP}_{M2} = 2.5$
- $R^{GRP}_{TOP} = 18$

Find R_g (Initial response time estimates assuming no device and software contention):

- $R_{BOT} = 2.5$
- $R_{M1} = 6$
- $R_{M2} = 6$
- $R_{TOP} = 24.5$

Find R^{IDL}:

- $R^{IDL}_{BOT} = 0$
- $R^{IDL}_{M1} = 0$
- $R^{IDL}_{M2} = 0$
- $R^{IDL}_{TOP} = I_g = 0$

LOOP Device Contention
LOOP Software Contention
$iter \leftarrow 0$
Step 3.
$iter \leftarrow 1$
Step 3. loop with $l = 3$:
MVA Inputs:

- TOP The set of customer classes.
- $N_{TOP} = 3$ The population of each customer class.
- $I_{TOP} = 6.5$ The think time of each customer class.
- $M1, M2$ The servers.
- $V_{TOP,M1} = 2, V_{TOP,M2} = 1$ Visits of customers to servers.
- $S_{TOP,M1} = 6, S_{TOP,M2} = 6$ Service time of customers at servers.
- $\phi_{M1} = FCFS, \phi_{M2} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{TOP,M1} = 12.41$, $R_{TOP,M2} = 9.07$ Customer residence times(for single visit).

- $U_{M1} = 0.8913$, $U_{M2} = 0.4456$ Server utilizations.

Find new R^{GRP} for customers:

- $R^{GRP}_{TOP} = 2(12.41) + 9.07 = 33.89$

Find new R_g for customers: contention):

- $R_{TOP} = 33.89 + 6.5 = 40.39$

Find new h_I for servers:

- $R^{IDL}_{M1} = R_{M1}/U_{M1} - R_{M1} = 6/0.89 - 6 = 0.73$

- $R^{IDL}_{M2} = 6/0.45 - 6 = 7.46$

Step 3. loop with $l = 2$:
MVA Inputs:

- $M1, M2$ The set of customer classes.

- $N_{M1} = 1$, $N_{M2} = 1$ The population of each customer class.

- $I_{M1} = 4.23$, $I_{M2} = 10.96$ The think time of each customer class.

- BOT The servers.

- $V_{M1,BOT} = 1$, $V_{M2,BOT} = 1$ Visits of customers to servers.

- $S_{M1,BOT} = 2.5$, $S_{M2,BOT} = 2.5$ Service time of customers at servers.

- $\phi_{BOT} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{M1,BOT} = 2.97$, $R_{M2,BOT} = 3.43$ Customer residence times(for single visit).

- $U_{BOT} = 0.5208$ Server utilization.

Find new R^{GRP} for customers:

- $R^{GRP}_{M1} = 2.97$

- $R^{GRP}_{M2} = 3.43$

Find new R_g for customers: contention):

- $R_{M1} = 2.97 + 3.5 = 6.47$

- $R_{M2} = 3.43 + 3.5 = 6.93$

Find new h_I for servers:

- $R^{IDL}_{BOT} = 2.5/0.5208 - 2.5 = 2.3$

Step 4. Convergence test fails. Go to LOOP Software Contention.
Step 3.
$iter \leftarrow 2$
Step 3. loop with $l = 3$:
MVA Inputs:

- TOP The set of customer classes.

- $N_{TOP} = 3$ The population of each customer class.

- $I_{TOP} = 6.5$ The think time of each customer class.

- $M1, M2$ The servers.

- $V_{TOP,M1} = 2$, $V_{TOP,M2} = 1$ Visits of customers to servers.

- $S_{TOP,M1} = 6.47$, $S_{TOP,M2} = 6.93$ Service time of customers at servers.
- $\phi_{M1} = FCFS$, $\phi_{M2} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{TOP,M1} = 13.35$, $R_{TOP,M2} = 10.75$ Customer residence times(for single visit).
- $U_{M1} = 0.8833$, $U_{M2} = 0.4733$ Server utilizations.

Find new R^{GRP} for customers:

- $R_{TOP}^{GRP} = 2(13.35) + 10.75 = 37.44$

Find new R_g for customers: contention):

- $R_{TOP} = 37.44 + 6.5 = 43.94$

Find new h_I for servers:

- $R_{M1}^{IDL} = R_{M1}/U_{M1} - R_{M1} = 6.47/0.8833 - 6.47 = 0.86$
- $R_{M2}^{IDL} = 6.93/0.4733 - 6.93 = 7.72$

Step 3. loop with $l = 2$:
MVA Inputs:

- $M1, M2$ The set of customer classes.
- $N_{M1} = 1$, $N_{M2} = 1$ The population of each customer class.
- $I_{M1} = 4.36$, $I_{M2} = 11.22$ The think time of each customer class.
- BOT The servers.
- $V_{M1,BOT} = 1$, $V_{M2,BOT} = 1$ Visits of customers to servers.
- $S_{M1,BOT} = 2.5$, $S_{M2,BOT} = 2.5$ Service time of customers at servers.
- $\phi_{BOT} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{M1,BOT} = 2.96$, $R_{M2,BOT} = 3.42$ Customer residence times(for single visit).
- $U_{BOT} = 0.5126$ Server utilization.

Find new R^{GRP} for customers:

- $R_{M1}^{GRP} = 2.96$
- $R_{M2}^{GRP} = 3.42$

Find new R_g for customers: contention):

- $R_{M1} = 2.96 + 3.5 = 6.46$
- $R_{M2} = 3.42 + 3.5 = 6.92$

Find new h_I for servers:

- $R_{BOT}^{IDL} = 2.5/0.5126 - 2.5 = 2.38$

Step 4. Convergence test fails. Go to LOOP Software Contention.
Step 3.
$iter \leftarrow 3$
Step 3. loop with $l = 3$:
MVA Inputs:

- TOP The set of customer classes.

- $N_{TOP} = 3$ The population of each customer class.
- $I_{TOP} = 6.5$ The think time of each customer class.
- $M1, M2$ The servers.
- $V_{TOP,M1} = 2$, $V_{TOP,M2} = 1$ Visits of customers to servers.
- $S_{TOP,M1} = 6.46$, $S_{TOP,M2} = 6.92$ Service time of customers at servers.
- $\phi_{M1} = FCFS$, $\phi_{M2} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{TOP,M1} = 13.33$, $R_{TOP,M2} = 10.72$ Customer residence times(for single visit).
- $U_{M1} = 0.8834$, $U_{M2} = 0.4729$ Server utilizations.

Find new R^{GRP} for customers:

- $R^{GRP}_{TOP} = 2(13.33) + 10.72 = 37.38$

Find new R_g for customers: contention):

- $R_{TOP} = 37.38 + 6.5 = 43.88$

Find new h_I for servers:

- $R^{IDL}_{M1} = R_{M1}/U_{M1} - R_{M1} = 6.46/0.8834 - 6.46 = 0.85$
- $R^{IDL}_{M2} = 6.92/0.4729 - 6.92 = 7.71$

Step 3. loop with $l = 2$:
MVA Inputs:

- $M1, M2$ The set of customer classes.
- $N_{M1} = 1$, $N_{M2} = 1$ The population of each customer class.
- $I_{M1} = 4.35$, $I_{M2} = 11.21$ The think time of each customer class.
- BOT The servers.
- $V_{M1,BOT} = 1$, $V_{M2,BOT} = 1$ Visits of customers to servers.
- $S_{M1,BOT} = 2.5$, $S_{M2,BOT} = 2.5$ Service time of customers at servers.
- $\phi_{BOT} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{M1,BOT} = 2.96$, $R_{M2,BOT} = 3.42$ Customer residence times(for single visit).
- $U_{BOT} = 0.5127$ Server utilization.

Find new R^{GRP} for customers:

- $R^{GRP}_{M1} = 2.96$
- $R^{GRP}_{M2} = 3.42$

Find new R_g for customers: contention):

- $R_{M1} = 2.96 + 3.5 = 6.46$
- $R_{M2} = 3.42 + 3.5 = 6.92$

Find new h_I for servers:

- $R_{BOT}^{IDL} = 2.5/0.5127 - 2.5 = 2.38$

Step 4. Convergence test fails. Go to LOOP Software Contention.
Step 3.
$iter \leftarrow 4$
Step 3. loop with $l = 3$:
MVA Inputs:

- TOP The set of customer classes.
- $N_{TOP} = 3$ The population of each customer class.
- $I_{TOP} = 6.5$ The think time of each customer class.
- $M1, M2$ The servers.
- $V_{TOP,M1} = 2, V_{TOP,M2} = 1$ Visits of customers to servers.
- $S_{TOP,M1} = 6.46$, $S_{TOP,M2} = 6.92$ Service time of customers at servers.
- $\phi_{M1} = FCFS, \phi_{M2} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{TOP,M1} = 13.33$, $R_{TOP,M2} = 10.72$ Customer residence times(for single visit).
- $U_{M1} = 0.8834$, $U_{M2} = 0.4729$ Server utilizations.

Find new R^{GRP} for customers:

- $R_{TOP}^{GRP} = 2(13.33) + 10.72 = 37.38$

Find new R_g for customers: contention):

- $R_{TOP} = 37.38 + 6.5 = 43.88$

Find new h_I for servers:

- $R_{M1}^{IDL} = R_{M1}/U_{M1} - R_{M1} = 6.46/0.8834 - 6.46 = 0.85$
- $R_{M2}^{IDL} = 6.92/0.4729 - 6.92 = 7.71$

Step 3. loop with $l = 2$:
MVA Inputs:

- $M1, M2$ The set of customer classes.
- $N_{M1} = 1, N_{M2} = 1$ The population of each customer class.
- $I_{M1} = 4.35, I_{M2} = 11.21$ The think time of each customer class.
- BOT The servers.
- $V_{M1,BOT} = 1, V_{M2,BOT} = 1$ Visits of customers to servers.
- $S_{M1,BOT} = 2.5$, $S_{M2,BOT} = 2.5$ Service time of customers at servers.
- $\phi_{BOT} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{M1,BOT} = 2.96$, $R_{M2,BOT} = 3.42$ Customer residence times(for single visit).
- $U_{BOT} = 0.5127$ Server utilization.

Find new R^{GRP} for customers:

- $R_{M1}^{GRP} = 2.96$
- $R_{M2}^{GRP} = 3.42$

Find new R_g for customers: contention):

- $R_{M1} = 2.96 + 3.5 = 6.46$
- $R_{M2} = 3.42 + 3.5 = 6.92$

Find new h_I for servers:

- $R_{BOT}^{IDL} = 2.5/0.5127 - 2.5 = 2.38$

Step 4. Convergence occurs for software model and $iter > 2$.
Step 5. $\delta(R_{TOP} = 43.88, 0) > \tau$. Convergence test fails; continue to step 6.
Step 6. Solve Device Contention Model
MVA Inputs:

- $TOP, M1, M2, BOT$ The set of customer classes.
- $N_{TOP} = 3$, $N_{M1} = 1$, $N_{M2} = 1$, $N_{BOT} = 1$ The population of each customer class.
- $I_{TOP} = 37.38$, $I_{M1} = 11.13$, $I_{M2} = 3.81$, $I_{BOT} = 2.38$ The think time of each customer class.
- $DCPU, COM$ The servers.
- $V_{TOP,DCPU} = 6$, $V_{TOP,COM} = 1$ Visits of customers to servers.
- $S_{TOP,DCPU} = 1$, $S_{TOP,COM} = 0.5$ Service time of customers at servers.
- $V_{M1,DCPU} = 3$, $V_{M1,COM} = 3$ Visits of customers to servers.
- $S_{M1,DCPU} = 1$, $S_{M1,COM} = 0.5$ Service time of customers at servers.
- $V_{M2,DCPU} = 3$, $V_{M2,COM} = 3$ Visits of customers to servers.
- $S_{M2,DCPU} = 1$, $S_{M2,COM} = 0.5$ Service time of customers at servers.
- $V_{BOT,DCPU} = 2$, $V_{BOT,COM} = 1$ Visits of customers to servers.
- $S_{BOT,DCPU} = 1$, $S_{BOT,COM} = 0.5$ Service time of customers at servers.
- $\phi_{DCPU} = DELAY, \phi_{COM} = FCFS$ Scheduling disciplines.

MVA Outputs:

- $R_{TOP,DCPU} = 1$, $R_{TOP,COM} = 0.63$ Customer residence times(for single visit).
- $R_{M1,DCPU} = 1$, $R_{M1,COM} = 0.62$ Customer residence times(for single visit).
- $R_{M2,DCPU} = 1$, $R_{M2,COM} = 0.59$ Customer residence times(for single visit).
- $R_{BOT,DCPU} = 1$, $R_{BOT,COM} = 0.57$ Customer residence times(for single visit).
- $U_{DCPU} = 1.42$, $U_{COM} = 0.24$ Server utilizations.

Find new R^{DEV} for customers:

- $R_{BOT}^{DEV} = 2(1) + 0.57 = 2.57$
- $R_{M1}^{DEV} = 3(1) + 0.62 = 3.62$
- $R_{M2}^{DEV} = 3(1) + 0.59 = 3.59$
- $R_{TOP}^{DEV} = 6(1) + 0.63 = 6.63$

Find new R^{GRP} for customers:

- $R_{BOT}^{GRP} = 0$

- $R_{M1}^{GRP} = 2.57$
- $R_{M2}^{GRP} = 2.57$
- $R_{TOP}^{GRP} = 2(3.62 + 2.57) + (3.59 + 2.57) = 18.54$

Find new R_g for customers: contention):

- $R_{BOT} = 2.57$
- $R_{M1} = 3.62 + 2.57 = 6.17$
- $R_{M2} = 3.59 + 2.57 = 6.19$
- $R_{TOP} = 18.54 + 6.63 = 25.17$

End Loop Device Contention, Go to top of loop Device Contention.

Appendix B

Analysis of the Multiple-Entry Server

The Laplace transform for the service time probability density function (pdf) of a series of r stages each with exponential service rate $r\ \mu$ is [Klein 75]:

$$B(s) = (\frac{r\ \mu}{s + r\ \mu})^r$$

The squared coefficient of variation of the series of stages with average service time $\bar{x}$ and second moment of service time $\bar{x^2}$ is:

$$C_v^2 = \frac{\sigma^2}{\bar{x}^2} = \frac{\bar{x^2}}{\bar{x}^2} - 1.$$

$\bar{x}$ and $\bar{x^2}$ can be found using the first and second derivatives of $B(s)$,

$$\bar{x} = -B'(s)|_{s=0}$$

and

$$\bar{x^2} = B''(s)|_{s=0}$$

$$B'(s) = -r\frac{(r\ \mu)^r}{(s + r\ \mu)^{r+1}} = -\frac{1}{\mu}\ \frac{(r\ \mu)^{r+1}}{(s + r\ \mu)^{r+1}}$$

So that

$$\bar{x} = -B'(s)|_{s=0} = \frac{1}{\mu}$$

$$B''(s) = \frac{r+1}{r\,\mu^2}\,\frac{(r\,\mu)^{r+2}}{(s+r\,\mu)^{r+2}}$$

So that

$$\bar{x^2} = B''(s)|_{s=0} = \frac{r+1}{r\,\mu^2}$$

Therefore:

$$C_v^2 = \frac{\frac{1}{\mu^2}\,\frac{r+1}{r}}{\frac{1}{\mu}^2} - 1 = \frac{r+1}{r} - 1 = \frac{1}{r}$$

For a series-parallel server with B independent branches, r_b stages on branch b, and probability π_b of choosing branch b this generalizes to:

$$C_v^2 = \frac{\sum_{i=1}^{B} \frac{\pi_i}{\mu_i^2}\,\frac{r_i+1}{r_i}}{(\sum_{i=1}^{B} \frac{\pi_i}{\mu_i})^2} - 1$$

Now consider the terms

$$r_{iter} = S_{cs} + v\; r_{ncs}$$

and

$$\sigma_{iter}^2 = \sigma_{cs}^2 + v\;\sigma_{ncs}^2 + v\;(1-v)(r_{ncs})^2$$

σ_{iter}^2 is simply the sum of the variances of the terms S_{cs} and $v\; r_{ncs}$. The variance of S_{cs} is σ_{cs}^2. The variance of $v\; r_{ncs}$ is computed as follows:

$$\begin{aligned}
\sigma^2 &= \bar{x^2} - \bar{x}^2 \\
\bar{x} &= v r_{ncs} \\
\bar{x^2} &= v \bar{r'^2_{ncs}} \\
\sigma^2 &= v \bar{r'^2_{ncs}} \;-\; v^2 (r_{ncs})^2 \\
&= v(\bar{r'^2_{ncs}} - v\;(r_{ncs})^2) \\
&= v(\bar{r'^2_{ncs}} - (r_{ncs})^2 + (1-v)(r_{ncs})^2) \\
&= v(\sigma_{ncs}^2 + (1-v)(r_{ncs})^2) \\
&= v\sigma_{ncs}^2 + v(1-v)(r_{ncs})^2
\end{aligned}$$

where $\overline{r'^2_{ncs}}$ is defined as the second moment of service times at the non-central servers.

Now consider the variance in phase response times $\sigma^2_{p,e}$. The number of iterations of a phase is geometrically distributed with the probability q of leaving the phase. Let $\bar{n}$ be the average number of iterations, and σ^2_n be the variance of the average number of iterations. From standard probability theory:

$$\begin{aligned}
\bar{n} &= 1/q \\
\sigma^2_n &= (1-q)/q^2
\end{aligned}$$

Let $\bar{x}$ be the average phase response time, $\overline{x^2}$ be the average square of phase response times, σ^2_x be the variance in average phase response times, and n be the number of iterations for some visit to the phase.

$$\begin{aligned}
\bar{x} &= r_{iter}\ \bar{n} \\
\overline{x^2} &= \sigma^2_x + \bar{x}^2 \\
\overline{x^2} &= \sum_{n=1}^{\infty} [n\ \sigma^2_{iter} + n^2\ r^2_{iter}]\ q\ (1-q)^{n-1} \\
\overline{x^2} &= \bar{n}\ \sigma^2_{iter} + \overline{n^2}\ r^2_{iter} \\
\sigma^2_{p,e} &= \overline{x^2} - \bar{x}^2 \\
&= \bar{n}\ \sigma^2_{iter} + \overline{n^2}\ r^2_{iter} - r^2_{iter}\ \bar{n}^2 \\
&= \bar{n}\ \sigma^2_{iter} + (\overline{n^2} - \bar{n}^2)\ r^2_{iter} \\
&= \bar{n}\ \sigma^2_{iter} + \sigma^2_n\ r^2_{iter} \\
&= \frac{\sigma^2_{iter}}{q} + \frac{1-q}{q^2}\ r^2_{iter}
\end{aligned}$$

Therefore:

$$\sigma^2_{p,e} = \frac{\sigma^2_{iter}}{q} + \frac{1-q}{q^2}(r_{iter})^2$$

Appendix C

GSPN Terminology

The definition of a GSPN and related terminology [Molloy 87, Murata 84] are provided here for reference.

Definition 1 A Petri Net is a bipartite graph with nodes P (Places), T (Transitions), input and output arcs A, and an initial marking M_0.

$$PN \equiv (P, T, A, M_0)$$
$$P = \{p_1, p_2, \cdots, p_n\}$$
$$T = \{t_1, t_2, \cdots, t_m\}$$
$$A \subset \{P\text{x}T\} \cup \{T\text{x}P\}$$
$$M_0 = \left\{m_1^0, m_2^0, \cdots, m_n^0\right\}$$

The marking function M describes the number of tokens in each place.

Definition 2 Define for a Petri net PN the four set functions:

- IP the set of input places for a transition t as $IP(t) = \{p|(p,t)\epsilon A\}$.
- OP the set of output places for a transition t as $OP(t) = \{p|(t,p)\epsilon A\}$.
- IT the set of input transitions for a place p as $IT(p) = \{t|(t,p)\epsilon A\}$.
- OT the set of output transitions for a place p as $OT(p) = \{t|(p,t)\epsilon A\}$.

Definition 3 A transition t is considered **enabled** when each and every one of its input places has one or more tokens (i.e. $\forall p \; \epsilon \; IP(t), M(p) > 0$).

Definition 4 A transition is said to have **fired** when it removes one token from each of its input places and places one additional token in each of its output places, thus changing the marking function.

Definition 5 A continuous time **Generalized Stochastic Petri Net** is extented from the Petri Net PN by associating either exponentially distributed firing times or firing probabilities with transitions. **Timed transitions** TT are associated with non-zero transition rates $\lambda = \{\lambda_1, \lambda_2, \cdots, \lambda_{q_1}\}$ and **immediate transitions** TI are associated with probabilities $\rho = \{\rho_1, \rho_2, \cdots, \rho_{q_2}\}$, where $q_1 + q_2 = m$ the number of transitions in the net. All output arcs of a place p go exclusively to timed transitions or immediate transitions. If the output arcs of p go to immediate transitions then the sum of the probabilities associated with the transitions is 1. The probabilities describe the ratio in which the immediate transitions become enabled. Immediate transitions require deterministically zero time to fire. The average firing or service times associated with TT are simply the inverse of the firing rates.

Definition 6 A marking M_2 is called **reachable** from a marking M_1 if there exists a sequence of transitions t^* such that $M_1 \rightarrow^{t^*} M_2$.

Definition 7 The **reachability set** R is the set of all markings M_i which are reachable from some initial marking M_0. $R(M_0) = \left\{M_i | \exists t^*, M_0 \rightarrow^{t^*} M_i\right\}$

Definition 8 A place p_i is said to be **bounded** if there exists a finite number $k \epsilon N$ such that $M(p_i) < k$ for all $M \ \epsilon \ R(M_0)$.

Definition 9 A GSPN is said to be **structurally bounded** if each place p_i is bounded for all $p_i \ \epsilon \ P$ for any initial marking M_0 with a finite number of tokens, that is $|M_0| < \infty$.

Definition 10 A transition t_i is said to be **live** if for for all $M_k \ \epsilon \ R(M_0)$ there exists a marking sequence M_j^* which enables t_i.

Definition 11 A GSPN is said to be **live** if each transition $t_i \epsilon T$ is live.

Definition 12 A set of transitions $T_c \subset T$ are termed in conflict if the firing of any subset of transitions $\{t_i\} \subset T_c$ results in a marking in which some other transition $t_j \ \epsilon \ T_c$ and $t_j \ not \ \epsilon \ \{t_j\}$ is disabled.

Definition 13 A GSPN is **persistent** if there is no subset of transitions which are in conflict for any marking M of the reachability set.

Definition 14 A Petri net is said to be **conservative** or an **S-invariant net** if there exists a positive integer $y(p)$ associated with each place p such that the weighted sum of tokens, $\sum_{p=1}^{n} M(p) \ y(p) = M^T y = M_0^T y$ is a constant for any fixed initial marking $M \ \epsilon \ R(M_0)$. M_T is the transpose of the marking matrix. Note that if $y(p) = 1$ for all places p, then the sum is the total number of tokens in the net.

Definition 15 An n-vector y is an **S-invariant** iff $M^T y \ = M_0^T y$ for any fixed initial marking M_0 and any $M \ \epsilon R(M_0)$.

Definition 16 A subset of places corresponding to nonzero entries of an n-vector y is called the **support** of y. A support is said to be **minimal** if no proper subset of the support is another support. Finally the minimal support is said to be a **minimal characteristic support** if $y(p) = 1$ for all places within the support.

For the purpose of analysing computer systems, timed transitions in GSPN often represent accesses to resources such as processors and disks. Thus several transitions may represent an visits to the same device. In such cases it is necessary to supplement the GSPN description with device queueing disciplines.

Definition 17 K is a set of resources or devices k that are associated with the TT of the GSPN. Though each timed transition is associated with exactly one device, many transitions may be associated with a single device. The scheduling discipline at each device may be first-come-first-served, processor-sharing, or infinite-server. The scheduling discipline of device k is denoted by ψ_k.

Appendix D

Ada Application Code

In this appendix the Ada programs are provided that were used in Chapter 6. A unit of service time in model parameters corresponds to 0.01 seconds in these programs. For example, a think time of 80 time units is 0.8 seconds. In Ada, a *delay n* statement delays a task for at least *n* seconds. A procedure named Work0x01 contains a *for loop* that had been timed as requiring 0.01 seconds of processing time. The procedure is called with a parameter that indicates how many time units of processor time are required.

In the applications, once all the tasks have started up the non-serving tasks begin to sum their response times. Once a non-serving task has completed its designated number of iterations it completes and all tasks stop summing their response times. When a task completes it calls a task that collects the response time information. Once all tasks have completed, the statistics task reports the average response times for the non-serving tasks.

In the Transaction processing example, the behaviour of the devices COMPORT, DECODISK, TERMPORT, and LOGDISK, are all emulated using Ada tasks that have *delay* statements.

D.1 Transaction Processing Original

```
with Text_IO, Instrument_Ada, Cpu_Time;
```

```
procedure Shell is
  package TIO  renames Text_IO;
  package IA renames Instrument_Ada;
  package FIO is new TIO.Float_IO(IA.Real);
  package IIO is new TIO.Integer_IO(Integer);

Num_Agent, Num_Customer, Agent_Cycle, Customer_Cycle : Integer := 0;

procedure Case_Study(Num_Agent, Num_Customer, Agent_Cycle,
                     Customer_Cycle : in Integer) is

Done_Stats : Boolean := False;

-- task types

task type Agent_Type;

task type Customer_Type;

task type Display_Type is
  entry Do_Op(Op : in Integer);
end Display_Type;

task type Sync_Type is
  entry Entry1;
  entry Entry2;
end Sync_Type;

task type Log_Type is
  entry Entry1;
end Log_Type;

task Register_Stats is
  entry Agent_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
  entry Customer_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
end Register_Stats;

-- task variables

Agents : array(1..Num_Agent) of Agent_Type;
Customers : array(1..Num_Customer) of Customer_Type;
Display : Display_Type;
InitRemote : Sync_Type;
Log : Log_Type;

-- stats task

task body Register_Stats is
Num_Accept : Integer := 0;
```

```
Agent_Done : Integer := 0;
Agent_Total_Time : IA.Real := 0.0;
Customer_Done : Integer := 0;
Customer_Total_Time : IA.Real := 0.0;
begin
  loop
    select
      accept Agent_Stats(Num_Done : in Integer;
                         Total_Time : in IA.Real) do
         Agent_Done := Agent_Done + Num_Done;
         Agent_Total_Time := Agent_Total_Time + Total_Time;
      end Agent_Stats;
    or
      accept Customer_Stats(Num_Done : in Integer;
                         Total_Time : in IA.Real) do
         Customer_Done := Customer_Done + Num_Done;
         Customer_Total_Time := Customer_Total_Time + Total_Time;
      end Customer_Stats;
    end select;
    Num_Accept := Num_Accept + 1;
    exit when Num_Accept = Num_Agent + Num_Customer;
  end loop;
  -- print out the stats and then terminate
  TIO.Put("The following Stats are for Agents, Customers = ");
  TIO.New_Line;
  IIO.Put(Num_Agent);
  TIO.Put(", ");
  IIO.Put(Num_Customer);
  TIO.New_Line;
  TIO.Put("Average Agent response time is ");
  if Agent_Done > 0
  then
    FIO.Put(Agent_Total_Time/IA.Real(Agent_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
  TIO.Put("Average Customer response time is ");
  if Customer_Done > 0
  then
    FIO.Put(Customer_Total_Time/IA.Real(Customer_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
end Register_Stats;

-- support procedures

procedure Work_0x01(Num_Times : in Integer) is
-- do 0.01 time units of work Num_Times
```

```
dummy : Integer := 0;
begin
  for i in 1..Num_Times loop
    for j in 1..20300 loop -- 20300 iterations has been timed
                           -- as 0.01 seconds on this machine
      dummy := j;
    end loop;
  end loop;
end Work_0x01;

-- task bodies

task body Agent_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
Dummy : Integer := 0;
begin
  loop
    exit when Num_Cycle = Agent_Cycle;
    Time_1 := Cpu_Time.Get_Clock;
    Work_0x01(4);
    delay 0.08; -- Mimic COMPORT
    Work_0x01(4);
    Display.Do_Op(2);
    Work_0x01(4);
    InitRemote.Entry1;
    Work_0x01(4);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := Cpu_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := True;
  Register_Stats.Agent_Stats(Num_Done,Total_Time);
end Agent_Type;

task body Customer_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Num_Cycle = Customer_Cycle;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(5);
```

```
    delay 0.04; -- THINK
    Work_0x01(5);
    Display.Do_Op(1);
    Work_0x01(5);
    InitRemote.Entry2;
    Work_0x01(5);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Customer_Stats(Num_Done,Total_Time);
end Customer_Type;

task body Display_Type is
begin
  loop
    select
      accept Do_Op(Op : in Integer) do
        if Op = 1
        then
          Work_0x01(2);
          delay 0.03; -- Mimic TERMPORT
          Work_0x01(2);
        else
          Work_0x01(2);
          delay 0.03; -- Mimic TERMPORT
          Work_0x01(2);
          Log.Entry1;
          Work_0x01(2);
        end if;
      end Do_Op;
    or
      terminate;
    end select;
  end loop;
end Display_Type;

task body Sync_Type is
begin
  loop
    select
      accept Entry1 do
        accept Entry2 do
          Work_0x01(1);
          delay 0.03; -- Mimic DECODISK
          Work_0x01(2);
```

```
          Log.Entry1;
          Work_0x01(1);
        end Entry2;
      end Entry1;
    or
     terminate;
    end select;
  end loop;
end Sync_Type;

task body Log_Type is
begin
  loop
    select
      accept Entry1 do
        Work_0x01(1);
        Delay 0.03; -- Mimic LOGDISK
        Work_0x01(1);
      end Entry1;
    or
      terminate;
    end select;
  end loop;
end Log_Type;

begin -- Case_Study
  NULL; -- do nothing but wait for tasks to terminate
end Case_Study;

begin -- shell
  TIO.Put("Enter the number of Agents ");
  IIO.Get(Num_Agent);
  TIO.New_Line;
  TIO.Put("Enter the number of Customers ");
  IIO.Get(Num_Customer);
  TIO.New_Line;
  TIO.Put("Enter the number of Agent cycles ");
  IIO.Get(Agent_Cycle);
  TIO.New_Line;
  TIO.Put("Enter the number of Customer cycles ");
  IIO.Get(Customer_Cycle);
  TIO.New_Line;
  Case_Study(Num_Agent,Num_Customer,Agent_Cycle,Customer_Cycle);
  TIO.New_Line;
  TIO.Put_Line("Th-Th-Th-That's all folks!");
end Shell;
```

D.2 Transaction Processing Baseline

```
with Text_IO, Instrument_Ada, Cpu_Time;
procedure Shell is
  package TIO  renames Text_IO;
  package IA renames Instrument_Ada;
  package FIO is new TIO.Float_IO(IA.Real);
  package IIO is new TIO.Integer_IO(Integer);

Num_Agent, Num_Customer, Agent_Cycle, Customer_Cycle : Integer := 0;

procedure Case_Study(Num_Agent, Num_Customer, Agent_Cycle,
                     Customer_Cycle : in Integer) is

Done_Stats : Boolean := False;

-- task types

task type Agent_Type;

task type Customer_Type;

task type Display_Type is
  entry Do_Op(Op : in Integer);
end Display_Type;

task type Sync_Type is
  entry Entry1;
  entry Entry2;
end Sync_Type;

task type Log_Type is
  entry Entry1;
end Log_Type;

task Register_Stats is
  entry Agent_Stats(Num_Done : in Integer;
                      Total_Time : in IA.Real);
  entry Customer_Stats(Num_Done : in Integer;
                      Total_Time : in IA.Real);
end Register_Stats;

-- task variables

Agents : array(1..Num_Agent) of Agent_Type;
Customers : array(1..Num_Customer) of Customer_Type;
Display : Display_Type;
InitRemote : Sync_Type;
Log : Log_Type;

-- stats task
```

```
task body Register_Stats is
Num_Accept : Integer := 0;
Agent_Done : Integer := 0;
Agent_Total_Time : IA.Real := 0.0;
Customer_Done : Integer := 0;
Customer_Total_Time : IA.Real := 0.0;
begin
  loop
    select
      accept Agent_Stats(Num_Done : in Integer;
                         Total_Time : in IA.Real) do
         Agent_Done := Agent_Done + Num_Done;
         Agent_Total_Time := Agent_Total_Time + Total_Time;
      end Agent_Stats;
    or
      accept Customer_Stats(Num_Done : in Integer;
                         Total_Time : in IA.Real) do
         Customer_Done := Customer_Done + Num_Done;
         Customer_Total_Time := Customer_Total_Time + Total_Time;
      end Customer_Stats;
    end select;
    Num_Accept := Num_Accept + 1;
    exit when Num_Accept = Num_Agent + Num_Customer;
  end loop;
  -- print out the stats and then terminate
  TIO.Put("The following Stats are for Agents, Customers = ");
  TIO.New_Line;
  IIO.Put(Num_Agent);
  TIO.Put(", ");
  IIO.Put(Num_Customer);
  TIO.New_Line;
  TIO.Put("Average Agent response time is ");
  if Agent_Done > 0
  then
    FIO.Put(Agent_Total_Time/IA.Real(Agent_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
  TIO.Put("Average Customer response time is ");
  if Customer_Done > 0
  then
    FIO.Put(Customer_Total_Time/IA.Real(Customer_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
--  terminate;
end Register_Stats;

-- support procedures
```

```
procedure Work_0x01(Num_Times : in Integer) is
-- do 0.01 time units of work Num_Times
dummy : Integer := 0;
begin
  for i in 1..Num_Times loop
    for j in 1..20300 loop -- 20300 iterations has been timed
                           -- as 0.01 seconds on this machine
      dummy := j;
    end loop;
  end loop;
end Work_0x01;

-- task bodies

task body Agent_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
Dummy : Integer := 0;
begin
  loop
    exit when Num_Cycle = Agent_Cycle;
    Time_1 := Cpu_Time.Get_Clock;
    Work_0x01(4);
    delay 0.08; -- Mimic COMPORT
    Work_0x01(4);
    Display.Do_Op(2); -- Still have to have contention for COMPORT
      Work_0x01(2); -- Display CPU
      Work_0x01(2); -- Display CPU
      Log.Entry1;  -- Display Visits Log. Still have to have contention for LOGDISK
        Work_0x01(1); -- Log CPU
        Work_0x01(1); -- Log CPU


    Work_0x01(4);
    InitRemote.Entry1; -- Still have to have contention for DECODISK
      Work_0x01(2); -- InitRemote CPU
      Work_0x01(2); -- InitRemote CPU
      Log.Entry1;  -- InitRemote Visits Log. Contention for LOGDISK
        Work_0x01(1); -- Log CPU
        Work_0x01(1); -- Log CPU


    Work_0x01(4);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := Cpu_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
```

```
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := True;
  Register_Stats.Agent_Stats(Num_Done,Total_Time);
end Agent_Type;

task body Customer_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Num_Cycle = Customer_Cycle;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(5);
    delay 0.04; -- THINK
    Work_0x01(5);
    Display.Do_Op(1); -- Does the TERMPORT
      Work_0x01(2); -- Display CPU TIME
      Work_0x01(2); -- Display CPU TIME
      Work_0x01(2); -- Display CPU TIME

    Work_0x01(5);

    --InitRemote.Entry2;
    InitRemote.Entry1; -- No Synchronization at InitRemote
      Work_0x01(2); -- InitRemote CPU Time
      Work_0x01(2); -- InitRemote CPU Time
      Log.Entry1;   -- InitRemote Visit To Log Disk
        Work_0x01(1); -- Log CPU Time
        Work_0x01(1); -- Log CPU Time

    Work_0x01(5);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Customer_Stats(Num_Done,Total_Time);
end Customer_Type;

task body Display_Type is
begin
  loop
    select
```

```
      accept Do_Op(Op : in Integer) do
        if Op = 1
        then
          -- Work_0x01(2);
          delay 0.03; -- TERMPORT
          -- Work_0x01(2);
        else
          -- Work_0x01(2);
          delay 0.03; -- TERMPORT
          -- Work_0x01(2);
          -- Log.Entry1;
          -- Work_0x01(2);
        end if;
      end Do_Op;
    or
      terminate;
    end select;
  end loop;
end Display_Type;

task body Sync_Type is
begin
  loop
    select
      accept Entry1 do
        -- accept Entry2 do
          -- Work_0x01(1);
          delay 0.03; -- DECODISK
          -- Work_0x01(2);
          -- Log.Entry1;
          -- Work_0x01(1);
        -- end Entry2;
      end Entry1;
    or
     terminate;
    end select;
  end loop;
end Sync_Type;

task body Log_Type is
begin
  loop
    select
      accept Entry1 do
        -- Work_0x01(1);
        Delay 0.03; -- LOGDISK
        -- Work_0x01(1);
      end Entry1;
    or
      terminate;
    end select;
```

```
  end loop;
end Log_Type;

begin -- Case_Study
  NULL; -- do nothing but wait for tasks to terminate
end Case_Study;

begin -- shell
  TIO.Put("Enter the number of Agents ");
  IIO.Get(Num_Agent);
  TIO.New_Line;
  TIO.Put("Enter the number of Customers ");
  IIO.Get(Num_Customer);
  TIO.New_Line;
  TIO.Put("Enter the number of Agent cycles ");
  IIO.Get(Agent_Cycle);
  TIO.New_Line;
  TIO.Put("Enter the number of Customer cycles ");
  IIO.Get(Customer_Cycle);
  TIO.New_Line;
  Case_Study(Num_Agent,Num_Customer,Agent_Cycle,Customer_Cycle);
  TIO.New_Line;
  TIO.Put_Line("Th-Th-Th-That's all folks!");
end Shell;
```

D.3 Avionics Original

```
with Text_IO, Instrument_Ada, Cpu_Time;
procedure Shell is
  package TIO  renames Text_IO;
  package IA renames Instrument_Ada;
  package FIO is new TIO.Float_IO(IA.Real);
  package IIO is new TIO.Integer_IO(Integer);

Num_Sensor, Num_Sensor_Cycle : Integer := 0;
Num_Keyboard : Integer := 1;
Num_Data_Link : Integer := 1;
Num_Display : Integer := 1;

procedure Tracker(Num_Sensor, Num_Sensor_Cycle : in Integer) is

Done_Stats : Boolean := False;

-- task types

task type Sensor_Type;

task type Keyboard_Type;
```

```
task type Correlator_Type is
  entry Track_Operation(Op : in Integer);
  entry Quit;
end Correlator_Type;

task type Track_Table_Type is
  entry Track_Operation(Op : in Integer);
  entry Quit;
end Track_Table_Type;

task type Data_Link_Type;

task type Display_Type;

task Register_Stats is
  entry Sensor_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
  entry Keyboard_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
  entry Data_Link_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
  entry Display_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
end Register_Stats;

-- task variables

Sensors : array(1..Num_Sensor) of Sensor_Type;
Correlator : Correlator_Type;
Keyboard : Keyboard_Type;
Track_Table : Track_Table_Type;
Data_Link : Data_Link_Type;
Display : Display_Type;
Num_Accept : Integer := 0;

-- stats task

task body Register_Stats is
-- Num_Accept : Integer := 0;
Sensor_Done : Integer := 0;
Sensor_Total_Time : IA.Real := 0.0;
Keyboard_Done : Integer := 0;
Keyboard_Total_Time : IA.Real := 0.0;
Data_Link_Done : Integer := 0;
Data_Link_Total_Time : IA.Real := 0.0;
Display_Done : Integer := 0;
Display_Total_Time : IA.Real := 0.0;
begin
  loop
    select
```

```
    accept Sensor_Stats(Num_Done : in Integer;
                        Total_Time : in IA.Real) do
       Sensor_Done := Sensor_Done + Num_Done;
       Sensor_Total_Time := Sensor_Total_Time + Total_Time;
    end Sensor_Stats;
  or
    accept Keyboard_Stats(Num_Done : in Integer;
                        Total_Time : in IA.Real) do
       Keyboard_Done := Keyboard_Done + Num_Done;
       Keyboard_Total_Time := Keyboard_Total_Time + Total_Time;
    end Keyboard_Stats;
  or
    accept Data_Link_Stats(Num_Done : in Integer;
                        Total_Time : in IA.Real) do
       Data_Link_Done := Data_Link_Done + Num_Done;
       Data_Link_Total_Time := Data_Link_Total_Time + Total_Time;
    end Data_Link_Stats;
  or
    accept Display_Stats(Num_Done : in Integer;
                        Total_Time : in IA.Real) do
       Display_Done := Display_Done + Num_Done;
       Display_Total_Time := Display_Total_Time + Total_Time;
    end Display_Stats;
  end select;
  Num_Accept := Num_Accept + 1;
  exit when Num_Accept = Num_Sensor + Num_Keyboard +
            Num_Data_Link + Num_Display;
end loop;
Track_Table.Quit;
Correlator.Quit;
-- print out the stats and then terminate
TIO.Put("The following Stats are for Num_Sensors = ");
IIO.Put(Num_Sensor);
TIO.New_Line;
TIO.Put("Average Sensor response time is ");
if Sensor_Done > 0
then
  FIO.Put(Sensor_Total_Time/IA.Real(Sensor_Done));
else
  TIO.Put(" NO COMPLETIONS ");
end if;
TIO.New_Line;
TIO.Put("Average Keyboard response time is ");
if Keyboard_Done > 0
then
  FIO.Put(Keyboard_Total_Time/IA.Real(Keyboard_Done));
else
  TIO.Put(" NO COMPLETIONS ");
end if;
TIO.New_Line;
TIO.Put("Average Data_Link response time is ");
```

```
  if Data_Link_Done > 0
  then
    FIO.Put(Data_Link_Total_Time/IA.Real(Data_Link_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
  TIO.Put("Average Display response time is ");
  if Display_Done > 0
  then
    FIO.Put(Display_Total_Time/IA.Real(Display_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
end Register_Stats;

-- support procedures

procedure Work_0x01(Num_Times : in Integer) is
-- do 0.01 time units of work Num_Times
dummy : Integer := 0;
begin
  for i in 1..Num_Times loop
    for j in 1..20300 loop -- 20300 iterations has been timed as 0.01 seconds
      dummy := j;
    end loop;
  end loop;
end Work_0x01;

-- task bodies

task body Sensor_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Num_Cycle = Num_Sensor_Cycle or Done_Stats;
    Time_1 := Cpu_Time.Get_Clock;
    Work_0x01(2);
    delay 1.0;
    Work_0x01(2);
    Correlator.Track_Operation(1);
    Work_0x01(2);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := Cpu_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
```

```
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := True;
  Register_Stats.Sensor_Stats(Num_Done,Total_Time);
end Sensor_Type;

task body Keyboard_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Done_Stats;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(2);
    delay 1.0;
    Work_0x01(2);
    Correlator.Track_Operation(2);
    Work_0x01(2);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Keyboard_Stats(Num_Done,Total_Time);
end Keyboard_Type;

task body Data_Link_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Done_Stats;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(1);
    delay 0.8;
    Work_0x01(1);
    Track_Table.Track_Operation(2);
    Work_0x01(1);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
```

```
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Data_Link_Stats(Num_Done,Total_Time);
end Data_Link_Type;

task body Display_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Done_Stats;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(5);
    delay 0.7;
    Work_0x01(5);
    Track_Table.Track_Operation(2);
    Work_0x01(5);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Display_Stats(Num_Done,Total_Time);
end Display_Type;


task body Correlator_Type is
Done : Boolean := False;
begin
  loop
    select
      accept Track_Operation(Op : in Integer) do
        if Op = 1
        then
          Work_0x01(2);
          Track_Table.Track_Operation(1);
          Work_0x01(2);
        else
          Work_0x01(8);
          Track_Table.Track_Operation(1);
          Work_0x01(8);
        end if;
```

```
      end Track_Operation;
    or
      accept Quit do
        Done := True;
      end Quit;
    end select;
    exit when Done;
  end loop;
end Correlator_Type;

task body Track_Table_Type is
Done : Boolean := False;
begin
  loop
    select
      accept Track_Operation(Op : in Integer) do
        if Op = 1
        then
          Work_0x01(2);
        else
          Work_0x01(3);
        end if;
      end Track_Operation;
    or
      accept Quit do
        Done := True;
      end Quit;
    end select;
    exit when Done;
  end loop;
end Track_Table_Type;

begin -- Tracker
  NULL; -- do nothing but wait for tasks to terminate
end Tracker;

begin -- shell
  TIO.Put("Enter the number of Sensors ");
  IIO.Get(Num_Sensor);
  TIO.New_Line;
  TIO.Put("Enter the number of Sensor Cycles ");
  IIO.Get(Num_Sensor_Cycle);
  TIO.New_Line;
  Tracker(Num_Sensor,Num_Sensor_Cycle);
  TIO.New_Line;
  TIO.Put_Line("Th-Th-Th-That's all folks!");
end Shell;
```

D.4 Avionics Baseline

```
with Text_IO, Instrument_Ada, Cpu_Time;
procedure Shell is
  package TIO  renames Text_IO;
  package IA renames Instrument_Ada;
  package FIO is new TIO.Float_IO(IA.Real);
  package IIO is new TIO.Integer_IO(Integer);

Num_Sensor, Num_Sensor_Cycle : Integer := 0;
Num_Keyboard : Integer := 1;
Num_Data_Link : Integer := 1;
Num_Display : Integer := 1;

procedure Tracker(Num_Sensor, Num_Sensor_Cycle : in Integer) is

Done_Stats : Boolean := False;

-- task types

task type Sensor_Type;

task type Keyboard_Type;

task type Correlator_Type is
  entry Track_Operation(Op : in Integer);
  entry Quit;
end Correlator_Type;

task type Track_Table_Type is
  entry Track_Operation(Op : in Integer);
  entry Quit;
end Track_Table_Type;

task type Data_Link_Type;

task type Display_Type;

task Register_Stats is
  entry Sensor_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
  entry Keyboard_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
  entry Data_Link_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
  entry Display_Stats(Num_Done : in Integer;
                     Total_Time : in IA.Real);
end Register_Stats;

-- task variables
```

```
Sensors : array(1..Num_Sensor) of Sensor_Type;
Correlator : Correlator_Type;
Keyboard : Keyboard_Type;
Track_Table : Track_Table_Type;
Data_Link : Data_Link_Type;
Display : Display_Type;
Num_Accept : Integer := 0;

-- stats task

task body Register_Stats is
-- Num_Accept : Integer := 0;
Sensor_Done : Integer := 0;
Sensor_Total_Time : IA.Real := 0.0;
Keyboard_Done : Integer := 0;
Keyboard_Total_Time : IA.Real := 0.0;
Data_Link_Done : Integer := 0;
Data_Link_Total_Time : IA.Real := 0.0;
Display_Done : Integer := 0;
Display_Total_Time : IA.Real := 0.0;
begin
  loop
    select
      accept Sensor_Stats(Num_Done : in Integer;
                       Total_Time : in IA.Real) do
         Sensor_Done := Sensor_Done + Num_Done;
         Sensor_Total_Time := Sensor_Total_Time + Total_Time;
      end Sensor_Stats;
    or
      accept Keyboard_Stats(Num_Done : in Integer;
                       Total_Time : in IA.Real) do
         Keyboard_Done := Keyboard_Done + Num_Done;
         Keyboard_Total_Time := Keyboard_Total_Time + Total_Time;
      end Keyboard_Stats;
    or
      accept Data_Link_Stats(Num_Done : in Integer;
                       Total_Time : in IA.Real) do
         Data_Link_Done := Data_Link_Done + Num_Done;
         Data_Link_Total_Time := Data_Link_Total_Time + Total_Time;
      end Data_Link_Stats;
    or
      accept Display_Stats(Num_Done : in Integer;
                       Total_Time : in IA.Real) do
         Display_Done := Display_Done + Num_Done;
         Display_Total_Time := Display_Total_Time + Total_Time;
      end Display_Stats;
    end select;
    Num_Accept := Num_Accept + 1;
    exit when Num_Accept = Num_Sensor + Num_Keyboard +
              Num_Data_Link + Num_Display;
```

```
  end loop;
  Track_Table.Quit;
  Correlator.Quit;
  -- print out the stats and then terminate
  TIO.Put("The following Stats are for Num_Sensors = ");
  IIO.Put(Num_Sensor);
  TIO.New_Line;
  TIO.Put("Average Sensor response time is ");
  if Sensor_Done > 0
  then
    FIO.Put(Sensor_Total_Time/IA.Real(Sensor_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
  TIO.Put("Average Keyboard response time is ");
  if Keyboard_Done > 0
  then
    FIO.Put(Keyboard_Total_Time/IA.Real(Keyboard_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
  TIO.Put("Average Data_Link response time is ");
  if Data_Link_Done > 0
  then
    FIO.Put(Data_Link_Total_Time/IA.Real(Data_Link_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
  TIO.Put("Average Display response time is ");
  if Display_Done > 0
  then
    FIO.Put(Display_Total_Time/IA.Real(Display_Done));
  else
    TIO.Put(" NO COMPLETIONS ");
  end if;
  TIO.New_Line;
end Register_Stats;

-- support procedures

procedure Work_0x01(Num_Times : in Integer) is
-- do 0.01 time units of work Num_Times
dummy : Integer := 0;
begin
  for i in 1..Num_Times loop
    for j in 1..20300 loop -- 20300 iterations has been timed as 0.01 seconds
      dummy := j;
    end loop;
```

```
  end loop;
end Work_0x01;

-- task bodies

task body Sensor_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Num_Cycle = Num_Sensor_Cycle or Done_Stats;
    Time_1 := Cpu_Time.Get_Clock;
    Work_0x01(2);
    delay 1.0;
    Work_0x01(2);
    -- Correlator.Track_Operation(1);
      Work_0x01(2); -- Correlator
        Work_0x01(2); -- Track Table
      Work_0x01(2); -- Correlator

    Work_0x01(2);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := Cpu_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := True;
  Register_Stats.Sensor_Stats(Num_Done,Total_Time);
end Sensor_Type;

task body Keyboard_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Done_Stats;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(2);
    delay 1.0;
    Work_0x01(2);
    -- Correlator.Track_Operation(2);
      Work_0x01(8); -- Correlator
        Work_0x01(3); -- Track Table
      Work_0x01(3); -- Correlator
```

```
    Work_0x01(2);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Keyboard_Stats(Num_Done,Total_Time);
end Keyboard_Type;

task body Data_Link_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Done_Stats;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(1);
    delay 0.8;
    Work_0x01(1);
--    Track_Table.Track_Operation(2);
        Work_0x01(3); -- Track Table

    Work_0x01(1);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Data_Link_Stats(Num_Done,Total_Time);
end Data_Link_Type;

task body Display_Type is
Num_Cycle : Integer := 0;
Num_Done : Integer := 0;
Total_Time : IA.Real := 0.0;
Time_1, Time_2 : Cpu_Time.Clock_Type;
begin
  loop
    exit when Done_Stats;
    Time_1 := CPU_Time.Get_Clock;
    Work_0x01(5);
```

```
    delay 0.7;
    Work_0x01(5);
    -- Track_Table.Track_Operation(2);
        Work_0x01(3); -- Track Table
    Work_0x01(5);
    Num_Cycle := Num_Cycle + 1;
    Time_2 := CPU_Time.Get_Clock;
    if (IA.Real(Time_2) > IA.Real(Time_1)) and not Done_Stats
    then
      Num_Done := Num_Done + 1;
      Total_Time := Total_Time + IA.Real(Time_2) - IA.Real(Time_1);
    end if;
  end loop;
  Done_Stats := true;
  Register_Stats.Display_Stats(Num_Done,Total_Time);
end Display_Type;

begin -- Tracker
  NULL; -- do nothing but wait for tasks to terminate
end Tracker;

begin -- shell
  TIO.Put("Enter the number of Sensors ");
  IIO.Get(Num_Sensor);
  TIO.New_Line;
  TIO.Put("Enter the number of Sensor Cycles ");
  IIO.Get(Num_Sensor_Cycle);
  TIO.New_Line;
  Tracker(Num_Sensor,Num_Sensor_Cycle);
  TIO.New_Line;
  TIO.Put_Line("Th-Th-Th-That's all folks!");
end Shell;
```