Correctness in multi-user hierarchically structured information systems

Alan Francis Sweet
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeib Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR
SWEET, Alan Francis, 1942-
CORRECTNESS IN MULTI-USER HIERARCHICALLY
STRUCTURED INFORMATION SYSTEMS.

Iowa State University, Ph.D., 1977
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106
Correctness in multi-user hierarchically structured information systems

by

Alan Francis Sweet

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of The Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1977
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1. General</td>
<td>1</td>
</tr>
<tr>
<td>1.2. A Review of Previous Work</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1. Hierarchical structures</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2. Information handling in a concurrent environment</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3. Reliability and the design process</td>
<td>9</td>
</tr>
<tr>
<td>1.3. Statement of the Problem</td>
<td>10</td>
</tr>
<tr>
<td>1.4. Organization of the Dissertation</td>
<td>32</td>
</tr>
<tr>
<td>2. THE MODEL</td>
<td>34</td>
</tr>
<tr>
<td>2.1. Introduction</td>
<td>34</td>
</tr>
<tr>
<td>2.2. Processes and Computations</td>
<td>35</td>
</tr>
<tr>
<td>2.3. Process Representation</td>
<td>38</td>
</tr>
<tr>
<td>2.4. Information Structures</td>
<td>45</td>
</tr>
<tr>
<td>2.5. Process Definition</td>
<td>46</td>
</tr>
<tr>
<td>3. THE DESIGN METHODOLOGY</td>
<td>51</td>
</tr>
<tr>
<td>3.1. Introduction</td>
<td>51</td>
</tr>
<tr>
<td>3.2. Hierarchical Structures</td>
<td>53</td>
</tr>
<tr>
<td>3.3. Information Structures</td>
<td>61</td>
</tr>
<tr>
<td>3.4. The Design Process and an Example</td>
<td>63</td>
</tr>
<tr>
<td>3.5. Comments on the Design Process</td>
<td>77</td>
</tr>
<tr>
<td>3.5.1. Consistency</td>
<td>77</td>
</tr>
<tr>
<td>3.5.2. Language constructs</td>
<td>82</td>
</tr>
<tr>
<td>4. CORRECTNESS</td>
<td>84</td>
</tr>
<tr>
<td>4.1. Introduction</td>
<td>84</td>
</tr>
</tbody>
</table>
7.3. Monitor 2 229
7.4. Monitor 3 232
8. CONCLUSIONS 245
9. BIBLIOGRAPHY 248
1.

1. INTRODUCTION

1.1. General

We introduce our work with an explanation of the meaning that we associate with the term 'information system'. The term, information system, is used to refer to a subsystem, such as a file system or data base system, dedicated to a single language which runs as a part of a host operating system. The subsystem is designed to own and manage the resources and information structures within its local operating system. For example, when the subsystem is loaded, the host system allocates a block of resources to the subsystem to maintain its information structures; thereafter, the subsystem controls the flow of information between the structures and the users of the subsystem. The resources owned by the subsystem are private to that system and may only be accessed on user demand. Thus, no user can gain access rights to the structures, only to the information.

Information systems and operating systems have been proposed, and in most cases, implemented in a bottom-up manner. The activities of the system are divided into modules. These modules are placed at various hierarchical levels. The entire system is viewed as a hierarchical structure of levels with modules in the lower (earlier)
levels being closest to the machine supporting the modules in the higher (later) levels.

Each level attempts to define the support of an important abstraction of the management of some system resource or information structure. For example, one level might support complex information structures using multidimensioned arrays, while a lower level supports an array by a vector. These systems may be viewed as layers of virtual machines where the lower levels hide information (abstract out details) of the machine from the higher levels.

This systematic approach to systems design is widely acclaimed and appears suitable for general purpose systems which support diverse programming languages on a specified hardware system.

In this investigation, we will consider a design methodology that may contribute to the art of system design as applied to information systems. We approach the design of such systems in a top-down manner employing the technique of functional decomposition in much the same way as in the design of structured programs.

Our interest will be confined to information subsystems dedicated to a single language. In this sense, we view the system as a semantic base for executing instructions in a given language. By considering the commands in the user language as primitive operations on abstract information
objects, functional decomposition may be employed to define a sequence of virtual machines which converge to a target machine. The target machine may exist before the fact or be built to directly support any of the virtual machines.

A complicating factor, which is the prime interest in our work, is the assumption of a multi-user environment in which each user interacts with the system using the same command language. Sharing of information, including modification, is also assumed. It is now critically important that our design process satisfy two additional constraints.

1) We wish to introduce as much concurrent activity as possible, both between separate users and within a single user, working strictly from the semantic definition of the user language. Concurrent activities between users allow for higher resource utilization and may lower response time. Concurrent activities within user contribute to lower response time.

2) We wish to ensure that each virtual machine is correct in the sense that it contains three classic properties in concurrent execution:
   a. mutual noninterference between concurrent processes,
   b. determinacy within a single computation, and
c. deadlock free.

The first, or outer, level of the hierarchical structure defines the semantics of the commands producing a virtual machine which operates on abstract information structures. Subsequent levels of process decomposition and structure refinement will begin to make assumptions about the supporting information structures. By working in such a top-down fashion, we can specify a provably correct design for a certain class of lower level supporting architectures. This is highly desirable in that such designs are transportable and may be defined to support processing by the appropriate hardware.

1.2. A Review of Previous Work

Three characteristics of our work in the art of systems design are:

1) An information system is specified as a hierarchical structure of information handling routines.

2) The information system specified must support the concurrent activity of a single language whose semantics are precisely defined.

3) The information system is to be specified by the technique of functional decomposition with respect to the semantic definitions. This specification is
provably correct with respect to concurrency for each decomposition.

While there are a number of studies and systems having some of these characteristics, we are aware of only one attempt to incorporate all three (7). In the following sections, we review the previous work in systems and software specification having any of the above characteristics and discusses the relation to our work.

1.2.1. Hierarchical structures

Parnas (32) discusses various systems which have been designed as hierarchical structures. The differences between them is described and how these differences effect the overall design of the system. He defines a structure hierarchical, if a relation on pairs of the parts of the structure \( R(X,Y) \) allows levels to be defined by

1) Level 0 is the set of part \( X \) such that there does not exist a \( Y \) such that \( R(X,Y) \) and

2) Level \( i \) is the set of parts \( X \) such that

a. there exists a \( Y \) in level \( i-1 \) such that \( R(X,Y) \) and

b. if \( R(X,Z) \) then \( Z \) is in level \( i-1 \) or lower.

It is noted that such a relation \( R \) exists only if the directed graph representing \( R \) contains no loops.
The parts to which Parnas refers is the collection of programs and modules which form a level. The definition indicates a property of all hierarchical systems, but it does not indicate, as he points out, the motivation for any particular structure.

The T.H.E. - Multiprogramming System is one of the earliest significant examples of a computer system whose operating system was systematically designed in a manner to cope with the complexity of resource handling (17,15). The parts of the system are subprograms used as procedures. The subprograms at a low-level define some "abstract machine" which the programs at a higher level use. The hierarchical structure was developed by building more complex "machines" around each existing "machine". A hierarchical structure of abstraction was also used by Liskov (26) in the design of the VENUS operating system, a system which exploits architectural features of the machine.

Madnick and Alsop (27) and Ritchie and Thompson (37) describe operating systems whose file system design is based on a hierarchical structure. The hierarchical structure of Zurcher and Randell (46) was motivated by the desire to be able to simulate a system as it was designed. The RC4000 system (5) based its hierarchical structure on the ownership of memory. Varney (42) extends this concept so that the hierarchical structure controls the management of other
resources as well. MULTICS (30, 39) uses a hierarchical structure of "rings" to control access to information structures and enforce a protection scheme.

All the projects and studies represent the development of structures whose conceptual basis, the hierarchical structure, provided a clear and logical design.

1.2.2. Information handling in a concurrent environment

The design and construction of information systems that support concurrent activity has been chiefly motivated toward achieving a high processing rate by exploiting the machine organizations made feasible by technological advances. Such systems can increase total production and decrease the overall cost to the user. Unfortunately, the information handling in such systems has been complicated by the potential interfering use of information structures by concurrent users.

The problem of being able to guarantee noninterfering access to information structures has been studied and solutions found for particular cases. Dijkstra (17), Cerf (8), and others have presented solutions to the problem by introducing new operations to be used in conjunction with the accessing of information structures. Brinch Hansen (6) and Hoare (22) give solutions that introduce additional software
support, called critical regions and monitors, which have sole control of the access rights to the information structures and can be designed to guarantee mutual noninterference.

In any concurrent system, there will exist an acceptable amount of indeterminacy when executing independent systems of processes. Indeterminacy between concurrent processes is natural and is normally acceptable, while within a single process indeterminacy is normally undesirable.

Deadlock prevention, detection, and recovery have been studied and some solutions are given by Holt (23), Russell (38), Havender (20), Coffman et al. (10), and others. These solutions are for concurrent systems which support the dynamic sharing of information and resources. Criteria for granting access capabilities depending on the system state have been determined.

The problems have been analyzed and solutions offered with respect to resources and access capabilities to information structures. The solutions have normally been determined for general purpose systems and not toward the particular demands of a special purpose subsystem.
1.2.3. Reliability and the design process

Recently, there has been increased concern over the design and construction of programs and systems of programs. This has been motivated primarily by the fact that the traditional design mechanisms have not always produced reliable products. The additional complexity of concurrency has only accentuated the inefficiencies of the traditional methods and has increased the need to be able to generate reliable programs and systems.

The difficulties in software construction in finding techniques of determining correctness of a program are documented by Floyd (19), Elspas et al. (18), and others. Lauer (25), Newton (29), and Ashcroft and Manna (2) have only recently been able to formalize and prove properties about concurrent processes. These techniques have their applications in demonstrating correctness of processes, but they lend little insight into the design of an operating system which guarantees correct resource or information structure handling.

Due to the difficulties in proving properties about programs, and resulting partly from the research in that area, another approach toward reliable software has recently evolved. This approach has been to develop programming techniques and languages constructs which aid in avoiding the
potential problems. These methods result from the structured programming philosophies of Dijkstra (16), Wirth (45), Mills (28), and others. One such approach is to apply the method of functional decomposition in the generation of the code in a system of programs. Hoare (22), Brinch Hansen (6), Parnas (33), and others have considered the problems of reliability by studying language requirements and suggesting certain language constructs or specification techniques which lend themselves to analytic approaches. While these methods have aided in the generation of the code, they have not, however, been able to guarantee the correct specification of error free algorithms.

1.3. Statement of the Problem

In our work, we will consider the design process of an information system which is supported by a hierarchical structure to be specified in a top-down manner. The levels of the hierarchical structure will be composed of processes which can only communicate with those processes in the immediate lower level. We follow the lead of Parnas (35) and apply the term uses. He states that the primary advantages of having system parts 'use' each other can be retained and the disadvantages reduced if the uses are restricted so that the graph of uses relation is loop-free.
The information structures will belong to that class of structures that can be modeled by directed graphs called trees: hierarchical data bases and some file systems, for example. The processes will be restricted in that the structures upon which they operate must either be local to the process or be passed as an argument into the process. This eliminates global structures.

When a new level is being defined, the instructions of the new processes are considered as primitive, that is, the operations of the lowest 'machine' and, as such, uninterruptible operations. The semantics of these operations is given in a definitional, or base, language similar to that used by Dennis (12) and Hawryszkiewycz (21).

When the processes which define a level are accepted, the process definitions of the next lower level are determined by investigating the functional statements of the newly accepted level. The processes of that level are then specified to meet the definitions of those functional demands upon the structures, possibly refined, of that level.

The most important aspect in our work is the assumption that the system exists in a multi-user, information sharing environment. Now, the primary consideration of the design process becomes two-fold:

1) to increase the degree of concurrency when possible,
2) to ensure that the system is correct, and remains correct, with each new level with respect to three major problems of concurrency, namely,

a) mutual noninterference between concurrent processes,

b) determinacy within a single computation, and

c) maintaining a deadlock free system.

We wish to guarantee these two properties at each stage of the design process. Then as each new level is completed, there exists a structure which is correct with respect to the stated problems of concurrency.

We now illustrate the process by considering the development of some of the processes of two neighboring levels.

Let the processes of a level be written on the assumption that the information structures have the form as shown by example in Figure 1.1.

The total set of information structures are collected together to form a tree. Each information structure is selected by the arc name at the first level. Subsequent levels are selected by giving the path name, the concatenation of arc names separated by periods, directed into the structure. Thus, if we wish to indicate node n₁ we use the path name A.B. Let the arguments to the level processes take the form of path names. We illustrate two
processes at this level in Figure 1.2.

Informally we define 'destroy1' as destroying the structure to which x is the path name. Create1 creates a structure with the path name x. For the moment, we will describe the instructions as follows:
has2(s, x) = true if x is a path name in s
           = false otherwise

destroy2(s, x) remove x from s
add2(s, x) add x to s
write(error) inform user of error condition.

If destroy1 were invoked by destroy1(IS, A.B) with the structures having the state as given in Figure 1.3, we would expect the final form as given in Figure 1.4.

If, on the other hand, create1 were invoked by create1(IS, A.E) and the structure had the initial state as given in Figure 1.3, then we would expect the final state to be as given in Figure 1.5.

Figure 1.3. The initial state of the structure.
Figure 1.4. The final state of the structure after destroy1(IS,A,B).

Figure 1.5. The state of the structure IS after create1(IS,A,E).

If, however, both destroy1 and create1 were invoked concurrently by destroy1(IS,A) and create1(IS,A,E), then it would be possible for both processes to test and find that
both destroy2(IS, A) and add2(IS, A.E) could be performed, but to have the structure A destroyed while add2(IS, A.E) was performing the addition. In other words, the final state is unpredictable.

This is an example of mutual interference between two copies of processes at the same level of a hierarchical structure. The problem can be solved by introducing access control mechanisms similar to the monitors of Hoare (22). We call these mechanisms request and release. The mechanisms are considered as operations which must be used to acquire and return access capabilities to the structures. Both must be required to allow no two interfering processes access to the same structure. If we further require that these mechanisms be used by every process to acquire access capabilities and to return access capabilities, then we can eliminate any potential interference by rewriting the procedures as in Figure 1.6.

If, now, destroy1 and create1 were invoked, then only one of the processes can gain access to the structures at any time. When that process is complete and returns the access capability, then the other process can acquire that capability and perform its function.

Thus, we would have the effective result as if destroy1 executed first then create1, giving the final state as in Figure 1.7, or as if create1(IS, A.E) executed first then
destroy1(IS, A), giving the final state as in Figure 1.8. Either case is correct and any concurrent execution of the restated procedures will result in a correct result.

\[
\text{destroy1}(s, x) \quad \text{create1}(s, x)
\]

\[
\begin{align*}
\text{begin} & \quad \text{begin} \\
\text{request}(s, x) & \quad \text{request}(s, x) \\
\text{if has2}(s, x) & \quad \text{if has2}(s, x) \\
\text{then do} & \quad \text{then do} \\
\text{destroy2}(s, x) & \quad \text{write}(\text{error}) \\
\text{end} & \quad \text{end} \\
\text{else do} & \quad \text{else do} \\
\text{write}(\text{error}) & \quad \text{add2}(s, x) \\
\text{end} & \quad \text{end} \\
\text{release}(s, x) & \quad \text{release}(s, x) \\
\text{end} & \quad \text{end}
\end{align*}
\]

Figure 1.6. The level processes restated with access control operations

\[
\begin{array}{c}
\text{IS} \\
\text{A} \\
\text{E} \\
\text{D} \\
3
\end{array}
\]

Figure 1.7. The final state if the execution sequence were destroy1(IS, A) then create1(IS, A, E).

The degree of concurrency, however, in either case is small; only one process can be executing at any time. We can increase the degree of concurrency, in some cases, if we
again restate the procedures as in Figure 1.9.

Figure 1.9. The procedures restated to take advantage of any potential increase in concurrency.

In more complicated procedures, the analysis of code to increase the degree of concurrency may be tedious and very time consuming. The restated procedures are correct, however, in the sense that they guarantee no potential interference between these two processes.
Having guaranteed the correctness of the procedures of one level, we can then consider the next lower level structures and procedures.

Let the information structures as operated on by the next lower level be refined to reflect the general form as shown in Figure 1.10. The structures are composed of directory objects and value objects. The objects are now selected by positional selectors as in vectors. Each object in a directory will either be another directory or a value object.

Figure 1.11a gives a logical structure as used by the higher level and Figure 1.11b gives the supporting structure used by the next lower level. The argument structures which are passed to the next level procedures are given in Figure 1.12.

Figure 1.13 gives the argument structure for the argument A.B and Figure 1.14 gives an argument structure which points to IS as given in Figure 1.11b.
Figure 1.10. A general form of information structures as accessed by the supporting level.

Figure 1.11a. The logical information structure as used by the higher level.
Figure 1.11b. The supporting structure used by the lower level to support the logical structure of the higher level.
Figure 1.12. General form of the argument structures passed to the lower level procedures.

Figure 1.13. The argument structure for the argument whose value is A.B.

Figure 1.14. The argument structure for the argument whose value is IS.
The structures and the processes of the lower level must support the logical path name accessing used by the previous level. The procedures to be defined to support that accessing methodology are has2, destroy2, and add2. It is assumed that the defined procedures and structures will provide the necessary support. The lower level becomes the machine (real or virtual) upon which the higher level runs. The procedures of the next level are defined in Figures 1.15a through 1.15c.

The 'has2' procedure searches an information structure to see if the path name specified is a path name in the structure. It returns the values true or false depending on whether it finds the path name or not.

The 'destroy2' procedure will remove the node of a structure to which a path name refers.

The 'add2' procedure will add a new node which can be referenced by the new path name.
has2(s,x)

begin
pcount3(ptr,s)
i=1
pathfound=true
while i LE x.size and pathfound do
j=1
arcfound=false
while j LE ptr.size and not arcfound do
if ptr.j.name=x.i
then do
arcfound=true
point3(ptr,ptr.j.val)
end
else do
j=j+1
end
end
pathfound=arcfound
i=i+1
end
return (pathfound)
end

Figure 1.15a. The procedure 'has2' of lower level.
destroy2(s,x)

begin
    point3(ptr,s)
    i=1
    while i LE x.size-1 do
        j=1
        arcfound=false
        while j LE ptr.size and not arcfound do
            if ptr.j.name=x.i
                then do
                    arcfound=true
                    point3(ptr,ptr.j.val)
                end
            else do
                j=j+1
            end
        end
        i=i+1
    end
    done=false
    while j LE ptr.size and not done do
        if ptr.j.name = x.'x.size'
            then do
                destroy3(ptr,j)
                done=true
            end
        else do
            j=j+1
        end
    end
end

Figure 1.15b. The procedure 'destroy2' of the lower level.
add2(s,x)

begin
  points(ptr,s)
  i=1
  pathfound=true
  while i LT x.size and pathfound do
  j=1
  arcfound=false
  while j LE ptr.size and not arcfound do
  if ptr.j.name=x.i
  then do
    arcfound=true
    points(ptr,ptr.j.val)
  end
  else do
    j=j+1
  end
  end
  pathfound=arcfound
  i=i+1
end

while i LT x.size do
  add3(ptr, x.i)
  points(ptr,ptr.'ptr.size'.val)
  i=i+1
end

Figure 1.15c. The procedure 'add2' of lower level.

There are three operations in these procedures for which we give an informal definition. The 'points' operation points a pointer variable at a specified structure. The 'destroys' operation removes an arc and the associated structure from a given structure. The 'add3' operation adds an arc to a specified structure.

If we assume that the logic of the new processes is correct, as we will continue to do since that is not the
focus of this paper, then the processes will operate on the refined structures and also give correct results with respect to the problems of concurrency. This is true since the use of the access control mechanisms of the previous level guarantees noninterference and any refinement of the structures or operational support does not negate that guarantee. We can not, however, increase the degree of concurrency by leaving these controls at the higher level. We can only increase that degree by increasing the concurrency at that level.

We can, however, increase the degree of concurrency, and still maintain correctness, by moving some of the controls into the lower level procedures. This is demonstrated by giving the modified low level procedures in Figure 1.16a through 1.16c. The modified high level procedures are given in Figure 1.17.

Thus, by moving the access operations to the lower level we have increased potential concurrency. The designers of hierarchical structures can achieve the same increase by moving at each level those requests and releases to the next lower level which, if moved, will still maintain correctness.
has2(s, x)

begin
    point3(ptr, s)
    i = 1
    pathfound = true
    request(s, x)
    while i ≤ x.size and pathfound do
        j = 1
        arcfound = false
        while j ≤ ptr.size and not arcfound do
            if ptr.j.name = x.1 then do
                arcfound = true
                point3(ptr, ptr.j.val)
            end
            else do
                j = j + 1
            end
        end
        pathfound = arcfound
        i = i + 1
    end
    return(pathfound)
end

Figure 1.16a. The modified 'has2' procedure.
\texttt{destroy2}(s, x) \\
\textbf{begin} \\
\texttt{point3}(ptr, s) \\
i=1 \\
\textbf{while} i \textbf{LT} x.\texttt{size} \textbf{do} \\
j=1 \\
arcfound=false \\
\textbf{while} j \textbf{LE} ptr.\texttt{size} \textbf{and not} arcfound \textbf{do} \\
\textbf{if} ptr.j.\texttt{name} = x.i \\
\textbf{then} do \\
arcfound = true \\
point3(ptr, ptr.j.\texttt{val}) \\
end \\
\textbf{else} do \\
j=j+1 \\
end \\
end \\
i=i+1 \\
end \\
j=1 \\
done=false \\
\textbf{while} j \textbf{LE} ptr.\texttt{size} \textbf{and not} done \textbf{do} \\
\textbf{if} ptr.j.\texttt{name} = x.\texttt{x.size} \\
\textbf{then} do \\
destroy3(ptr, j) \\
release(s.x) \\
done=true \\
end \\
\textbf{else} do \\
j=j+1 \\
end \\
end \\
end \\

\textbf{Figure 1.16b. The modified 'destroy2' procedure.}
add2(s,x)

begin
  point3(ptr,s)
i=1
  pathfound=true
  while i LT x.size and pathfound do
    j=1
    arcfound=false
    while j LE ptr.size and not arcfound do
      if ptr.j.name=x.i
        then do
          arcfound=true
          point3(ptr,ptr.j.val)
        end
      else do
        j=j+1
      end
    end
    pathfound=arcfound
    i=i+1
  end
  while i LT x.size do
    add3(ptr,x.i)
    point3(ptr,ptr.'ptr.size'.val)
    i=i+1
  end
  release(s.x)
end

Figure 1.16c. The modified 'add2' procedure.
destroy_1(s,x)                          create_1(s,x)
begin                                     begin
  if has_2(s,x)                            if has_2(s,x)
    then do                                then do
      destroy_2(s,x)                        release_1(s,x)
      write(error)                          write(error)
    end                                      end
  else do                                  else do
    release_1(s,x)                         add_2(s,x)
    write(error)                           end
  end                                       end
end                                        end

Figure 1.17. The higher level procedures restated after some of the access controls have been moved to lower level procedures.

This, however, is another undue burden to place upon the designer. It requires that each level be analyzed for potential problems of concurrency and the determination be made for the locations where the releases and requests can be placed to increase concurrency while still maintaining correctness with respect to concurrency.

Thus, the problem has become as follows:

1) to specify the hierarchical structure, increasing concurrency whenever possible,

2) to ensure that the system is correct, and remains correct, with each new level with respect to three major problems of concurrency:
   a) mutual noninterference between concurrent processes,
   b) determinancy within a single computation, and
c) deadlock avoidance, and
3) determination of a method, which if automated, will
relieve the designer of the burden of 1) while
ensuring 2).

1.4. Organization of the Dissertation

Chapter 2 presents the graph models used to represent
process and information structures. The concept of
computation of a process is discussed and the use of the
model in the design process is illustrated.

Chapter 3 introduces and discusses the design process in
detail.

Chapter 4 discusses correctness with respect to
concurrency. Determinacy, mutual exclusion, information
integrity, and deadlock conditions are defined and
demonstrated with the aid of the models.

Chapter 5 presents the algorithm used in the design of a
hierarchical information system which will guarantee
correctness with respect to the properties discussed in
chapter 4.

Chapter 6 demonstrates that the algorithm will guarantee
the properties of correctness with respect to concurrency.

Chapter 7 discusses access control mechanisms and
demonstrates various types that can be used in conjunction
with the algorithm.

Chapter 8 discusses conclusions and recommendations for further work.
2. THE MODEL

2.1. Introduction

In this chapter, a model for processes and information structures is presented. The directed graph is used to represent the processes where the nodes of the graph represent computation steps and the edges represent transitions between computations. This is similar to the graph models used by Adams (1), Cerf (8), Dennis (12), and others. The information structures are represented as tree structured graphs and changes to the structures are demonstrated by appropriate changes to the graphs. A definitional, or base, language similar to VDL (44), and the definitional languages of Hawryszkiewycz (21) and Dennis (12), is used to define the semantics of the computations in terms of tree manipulation operations on the information structure representations.

We find this approach useful because it gives a concise representation of the logic of the processes, a logical representation of the information structures, the effect of a computation upon the structures, and the state of a collection of processes and structures at any point in a computation.
2.2. Processes and Computations

In our work, we adopt the concept of a process as used by Horning and Randell (24). Much of the following is either borrowed directly or paraphrased from their work.

Their definition is based on the concepts of state variable, state variable sets, and states. *State variables* are elementary quantities which assume certain well-defined values. A set of named state variables constitutes a state variable set. An assignment of values to all the variables in a state variable set defines a state of the set. ... The set of possible states for a given state variable set is the state space of the set."

An example of a state variable set is given by

\[ V = (x, y, z), \]

which contains the three state variables \( x, y, \) and \( z \). Let \( x \) be assigned the value of 1, \( y \) the value of 2, and \( z \) the value of 3. The state is defined by

\[ (x=1, y=2, z=3) \]

or just

\[ V = (1, 2, 3). \]

The state space of this set, if \( x, y, \) and \( z \) are limited to the positive reals, is the set given by

\[ s(V) = \{(x=i, y=j, z=k) | i>0, j>0, k>0\} \]
"A computation in a state space is a sequence of states from that space. The first element of the sequence is called the initial state, the last, if the computation is finite, is the final state."

A finite computation from the above state space is the sequence,
\[
c=\{(1,2,3), (2,4,6), (4,8,12)\},
\]
where (1,2,3) is the initial state and (4,8,12) is the final state.

"An action in a state space is a set of assignments to some of the variables of its state variable set."

Thus, in the above example, if the state (1,2,3) is followed by the action (x:=2, y:=4, z:=6) then the immediate successor is the state (2,4,6).

"An action function in a state space is a mapping from states into actions. We may use an action function to generate a sequence (computation)."

The previous computation could be generated by the action function,
\[
f(x,y,z)=(x:=2x, y:=2y, z:=2z), \text{ if } x\leq 4.
\]

"A process is a triple \((S,f,s_0)\) where \(S\) is a state space, \(f\) is an action function in that space, and \(s_0\) is the subset of \(S\) which defines the initial states of the process."

A process then is "an abstract, timeless entity." Only when a 'machine' or a processor realizes a process, (i.e.
generates the same set of computations) is the process related or interpreted in time. If the two sets of computations are identical then the processor is said to be an exact realization of the process and the process an exact specification of the processor.

In a concurrent environment, there may be more than one process active at the same time and we may need to be able to consider the concurrent sequences of action functions as they are applied. In a concurrent environment, it is possible for one action to initiate before another has been terminated. Using the notation presented in Coffman and Denning (9), it is possible to represent this overlap in actions. Let p be an action, then define \( \tilde{p} \) to be the initiation of the action p and \( p \) to be its termination. These initiations and terminations are called events. They will be considered timeless. We now can represent concurrent actions as a sequence of events. For example, if p and q are concurrent actions, then a possible event sequence could be \( \tilde{p} \quad \tilde{q} \quad q \quad p \).

We will use these concepts of Horning and Randell (24) and Coffman and Denning (9) when we discuss processes and their analysis.
2.3. Process Representation

We represent the operations and the flow of control of a process by directed graphs called process graphs. The operations are represented as nodes and the flow of control by arcs connecting the nodes. The execution of an operation is represented by placing a marker, or token, in a node. The execution of a process is represented by moving a token along the arcs executing the nodes encountered. Moving a token along an arc is called a transition. This is similar to the operation of petri-nets (36) or data-flow schemas (13).

Each of the nodes is labeled. Following Lauer (25), we interpret this label as the action to be associated with the execution of the node. When a token is placed in a node, we say the action has been initiated. When a token is removed from a node, we say the action has been terminated. An example of a procedure and the corresponding process graph is given in Figure 2.1.
destroy1(s,x)
begin
  if has2(s,x)
    then do
      destroy2(s,x)
    end
  else do
    write(error)
  end
end

Figure 2.1. The procedure 'destroy1' and the corresponding process graph.

The **start** node, Figure 2.2, corresponds to the entry point of a procedure. The action of a start node is interpreted as an event and is represented as an initiation in event sequences. If a token is placed in a start node, we interpret this state as the initiation of a process. We require that each process graph contain one, and only one, start node. This corresponds to a good programming practice of having only one entry point in a procedure.

![start node](image)

Figure 2.2. The start node.

The **stop** node, Figure 2.3, corresponds to the exit point of a procedure. The action of a stop is interpreted as an event and is represented as a termination in event sequences.
Figure 2.3. The stop node.

The assignment node, Figure 2.4, corresponds to an assignment statement of a procedure. The action of an assignment node is to make value assignments to some state variable. For example, if a token were placed in the node of Figure 2.4, the state variable i would be assigned the value 1.

\[ i := 1 \]

Figure 2.4. The assignment node.

The decider node, Figure 2.5, corresponds to the testing portion of the if-then-else construct used in most programming languages. The label associated with this node states the tests to be performed on the state variable. In Figure 2.5, the value of the state variable i is compared to the constant 1. The arcs emanating from the decision node are labelled with the condition that must be satisfied in order for the transition to be taken.
Figure 2.5. The decider node.

The cobegin and coend nodes, Figures 2.6 and 2.7, correspond to the cobegin and coend constructs used by Brinch Hansen (6) for introducing concurrency in a process. They are used to represent the initiation and termination of concurrent execution paths in a process.

![cobegin node](image)

Figure 2.6. The cobegin node.

![coend node](image)

Figure 2.7. The coend node.

In Figure 2.8 we give a partial procedure and the corresponding partial process graph. The cobegin and coend nodes are each composed of two parts. The cobegin has one entry node and n initiation nodes (one for each of n parallel execution paths). The coend node has n termination nodes (one for each of the parallel execution paths) and one exit node. These nodes do not have any actions or events.
associated with them; they just serve to graphically represent the initiation and termination of concurrent portions of a process. We demonstrate the use of these nodes by example.

```plaintext
cobegin
cbegin
  y:=1
end
begin
  x:=2
end
coend
```

Figure 2.8. An example of the `cobegin - coend` constructs and their graphic representations.

Assume a token is placed in the entry node of the cobegin construct, as shown in Figure 2.9. The operation of the cobegin node will remove this token and place a token in each of the initiation nodes, as shown in Figure 2.10. An execution sequence for each of these initialized parallel paths can be generated by moving these tokens through the paths. After all of the parallel paths have completed, there will be a token in each of the termination nodes of the coend node, as shown in Figure 2.11. At this time, the coend node can operate, removing the tokens from each of the termination nodes, and placing a token in the exit node, as shown in
Figure 2.12. If this example had been used in a procedure, the flow would have continued to the next node immediately following the coend node.

Figure 2.9. Initialized cobegin node.

Figure 2.10. Parallel paths initialized.
The request and release nodes, Figures 2.13 and 2.14, are used to represent request and release operations used in procedures to acquire and return access capabilities to shared information structures. The request is associated with two events, the request for the access capabilities and the granting of those capabilities. The request can be made for access capabilities to more than one structure, but the process is assumed to own none of the capabilities until they
are all granted simultaneously. The release is associated with one event, the returning of access capabilities to an information structure. We require that the operation of the request and release actions must not allow interfering processes to be granted simultaneous access capabilities.

\[ \text{request}(x) \]

Figure 2.13. The request node.

\[ \text{release}(x) \]

Figure 2.14. The release node.

2.4. Information Structures

We represent the logical structure of all the information structures by graphs called trees. These graphs are built from two node types and labeled arcs. The nodes are branching nodes and value nodes as shown in Figure 2.15. Arcs may enter and emanate from branching nodes but may only enter value nodes. An example is shown in Figure 2.16. The nodes of a tree may be referenced by giving the arc labels which point to the nodes. In the examples, the value node \( n_4 \) is specified by IS.AB and the branching node \( n_1 \) is specified by IS.
2.5. Process Definition

We define the semantics of the operations of a process by using a definitional, or base, language. The language is a tree-manipulation language and defines the effect of the operations upon the information structures. The instructions in the base language and their definitions are given in Figure 2.17.

A process is completely specified when all the operations have been defined. When a process is specified, it is assumed that the language is powerful enough to
interpret the path names and reference the specified nodes. When the operations are evident, such as addition or multiplication, we do not give the definition. The effect of other operations, which may become procedure calls, must be given, however, for this is the definition used to generate necessary procedures. An example of a completely defined procedure is given in Figure 2.18 along with an example information structure.

We can trace the execution sequences of the processes by specifying the node labels of the sequences. For example, a possible execution sequence of the process in Figure 2.18 is

\[ \text{start, has2}(Q, y), \text{has2}(P, x), \text{copy2}(P, x, Q, y), \text{delete2}(Q, y), \text{stop}. \]

When generating sequences, we often, for convenience, give the nodes of a process graph symbolic labels as in Figure 2.19 and specify the sequences using the symbolic labels rather than the operation name. For example, we can now represent an action sequence from the process in Figure 2.19 by

\[ \text{start n}_1, n_2, n_4, n_6, \text{stop}. \]

There are cases in which the action sequences are not a fine enough representation. In these cases we can expand the action sequences into event sequences giving the initiation and termination events associated with the actions. The event sequence associated with the above action sequence would be
Such sequences will aid us later in specifying and representing concurrent execution sequences.

Let $P$ and $Q$ be pointer variables, $x$ a path name, and $y$ an arc name.

- $\text{isp}ath(c,x) = \begin{cases} \text{true} & \text{if } x \text{ is a path name emanating from } c. \\ \text{false} & \text{otherwise.} \end{cases}$

- $\text{link}(P,Q)$ point the pointer variable $P$ at the node to which $Q$ refers.

- $\text{addarc}(P,y)$ add an emanating arc $y$ to the node to which $P$ refers.

- $\text{deletearc}(P,y)$ delete the arc $y$ emanating from the node to which $P$ refers.

- $\text{assign}(P,v)$ assign the value $v$ to the node to which $P$ refers.

- $\text{val}(P)$ the value of the node to which $P$ refers.

- $\text{apply}(x,y)$ apply the function selected by $y$ to the structure selected by $x$.

Figure 2.17. The base language used to specify operational definitions.
\begin{verbatim}
has2(P,x) := true if ispath(P,x) = true := false otherwise

copy2(P,x,Q,y) := assign(P*x, val(Q*y))
delete2(Q,y) := deletearc(Q,y)
add2(P,x) := addarc(P,x)
\end{verbatim}

**Figure 2.18.** A process move1(P,x,Q,y) and an example information structure.
Figure 2.19. A process graph with symbolic labels.
3. THE DESIGN METHODOLOGY

3.1. Introduction

In this chapter, we explain how we view the information system and how the design method supports that view. In general, we wish to specify an information system whose control programs form a hierarchical structure and whose information structures can be modeled by trees. The information system is to be specified in a top-down manner proceeding from the semantic definition of the user language and applying the method of functional decomposition. This is the assumed underlying model on which we base our investigation.

This method may be applied to design systems whose information structures have been defined over that model. The model and the definition of the language constructs can be used as a guide to the refinement of the information structures and the decomposition of the procedures written to realize the language. The model and definitions may also be used to determine whether or not the system realizes its logical definition.

This method may also be applied to the design of a system to be implemented on different machines each of which has different software or hardware support already available.
The design method can generate a system whose higher levels are transportable across the machines.

This method may also be applied to the design of a system which is to be implemented in phases where each phase increases the power of the user language constructs. Each level of the hierarchical structure can, as the system is designed, allow the use of more complicated and more powerful arguments. The resulting user language allows the user to operate at various levels in the hierarchy.

In this section, we will present an example of the first category as applied to an information system for personnel records. We will specify the system top-down, using as a guide to design decisions, the semantic definition of the user language.

In this work, we are not concerned with the correctness of the procedures apart from correctness with respect to the problems of concurrency. We are only concerned with relieving the designer of the burdens of analysis necessary to increase concurrency and still ensure correctness with respect to concurrency.

The next two sections of this chapter give our views on hierarchical structures and informations structures. Section 3.4 gives an example and shows how the design process progresses. Section 3.5 comments on the method and its application.
3.2. Hierarchical Structures

In recent years, considerable effort has been devoted to the development of large and complex systems. As a result of their size and complexity, there has been considerable concern about the design of reliable software systems (40,34,31). In order to overcome these difficulties, some systems have been constructed according to the principles of modularity (17,15,33,43). It is generally acknowledged that these principles present the right approach. The code to perform a function is placed within a single module and standards are established concerning the use of the module. This allows the designer to concentrate on particular aspects of the system whose entire dimensions can be mentally visualized. The designer can then implement and verify these parts independent of the other parts. Unfortunately, this may not eliminate problems, especially in a multi-user environment, because different modules may share information structures in conflicting ways or call each other in unpredictable ways (33).

Dijkstra (17,15,14) extended the concept of modularity by introducing the idea of a hierarchical control structure. In this structure, the hardware is regarded as the lowest level, level 0, in a multilevel system. The system is developed in stages, one level at a time. The designer
chooses some feature not supported by the hardware, say logical file accessing, and implements that feature in software; this becomes level 1. The next level, level 2, uses the level 1 machine to implement some other feature, say virtual memory. This design process has the advantages of being easily testable, since all calls are downward and each level is intellectually manageable by the designer.

Parnas (35) states the characteristics of such a structure in terms of how the programs in the structure use each other. He states that the disadvantages of interrelated processes can be reduced if the graph of the relation between the programs is loop-free. If this restriction is accepted, then it is possible to design a hierarchical structure whose levels are defined according to the following rules:

1) Level 0 is the set of all programs which use no other programs.

2) Level i is the set of all programs that use at least one program at level i-1 and no programs at a level higher that i-1.

Parnas indicates that if such a hierarchical ordering exists, then each level is a testable and usable subset of a system.

We find some disadvantages in using this definition. First, according to this definition, it is allowable for a program at level i to use programs at any level below level
i. We illustrate this in schematic form in Figure 3.1. This means the interface between a low level can be with any of the higher levels.

Thus, any system modifications which effects a low level will have to be tested against every higher level interface.

Figure 3.1. Example of communication paths from a level i program.

Second, the levels are only defined by what programs they use. In this case, it may be possible for a resource to be shared between different levels, thus the problem of
controlling access to the resource and the accessing methods becomes complicated. This sharing is demonstrated in Figure 3.2.

![Diagram](image)

Figure 3.2. Processes at different levels sharing a resource.

The third problem in Parnas' approach is the complexity in invoking features available to the user. If a feature is to be available at the outer level, then the system designer may have to include this requirement in the design decisions when implementing many lower levels, thus, complicating the designer's task. If the desired user feature is buried at some lower levels, then the user must interface with multiple levels and coordinate those different interfaces.
Last, and most important to us, if the designer must guarantee some property of the overall system, there is no intermediate method of checking for that property until the system is complete.

For the reasons stated above, we modify the definition of a hierarchical system to be as follows:

1) Level 0 is the outer-most and first defined level.
2) Level i is the set of programs, which if they do use any other programs, use only programs at level i+1 and no programs at any lower level.
3) Level k is the inner-most, and the last defined, and it is composed of a set of programs which use no other programs.

A representation of such a structure and the communication paths is shown in Figure 3.3. We have maintained, with such a structure, the advantages stated by Parnas, testability and downward communication, and we will also be able to gain advantages.

First, the only interfaces that exist are between adjacent levels. Programs in level i may only communicate with programs in level i+1 or level i-1. Thus, the only interfaces that need be tested are between immediate adjacent levels.
Second, the programs in a level are defined by the refinement or abstraction they represent. Thus, they may only access a resource whose access is restricted to a given level, not shared among several levels. This means that any higher level program can only access a resource through the programs of the level which owns the resource. This is shown in Figure 3.4.
Third, if any feature is to be made available to the users, the logical support of this feature can be incorporated into the highest level and the system designed to support that feature. Thus, the user can use the feature directly and the designer can still proceed using the functional demands of the outer level.

Last, we can guarantee a system property by ensuring the property is guaranteed by each new level and assuming the property will be guaranteed by its supporting level. This will ensure that the previously defined levels will guarantee the property.

As a result of this viewpoint, some hierarchical systems would have their structure defined differently. There would be no communication within a level, for example. Any system with such communication paths would have to be
restructured with a new level for the used programs. This fact, and the resource policy of communication, may be major drawbacks to this approach. Such an approach will result in the generation of identity programs (programs which perform no operations except a single program at the next lower level). For example, assume we had a system with the original structure and communication paths as shown in Figure 3.5. If this system were restructured according to our definition, the system would then appear as in Figure 3.6 with the identity programs $I_1$ and $I_2$.

Figure 3.5. Original level structure and communication paths.
Figure 3.6. Restructured system with identity programs.

This generation of identity programs will increase the time spent in level switching. Bernstein and Siegel (4) have recently addressed this problem, and the protection problem, and proposed some simple hardware mechanisms for level structured systems. They state that these mechanisms are available and can decrease the amount of switching costs.

3.3. Information Structures

We restrict the logical construction of the information structures in our system to structures whose graphic representation is given by trees. These types of structures are used in hierarchical data bases and in some file systems. As the levels in a hierarchical system are developed, portions of this logical definition may be supported by lower
levels through refinements of the information structures.

The decisions by the designer about which portions of the structures are to be refined are not within the scope of this work. We only assume the designer is guided by some design criteria, and our concern is in the possible increase in concurrency. If, at level $i$, a structure is defined as a single object and at level $i+1$ this object is refined into multiple objects, then it may be possible to increase the degree of concurrency by allowing concurrent access to the separate objects at level $i+1$.

We illustrate this process as follows. An information object is defined, as in Pascal, as a type but with no structure. For example,

```pascal
  type IS = ?
  var BD: IS
```

In the next level, this type is refined and its structure specified in terms of new types, as in the following example.

```pascal
  type A = ?
  type B = ?
  type IS : record
    F1 : A
    F2 : set of B
  end
```
The refinement and the type definition may take place at each level but must continue until all types are defined in terms of known types supported by some machine. In our work, we assume the existence of certain primitives and structure types which may be used at any level. The primitive types are boolean, integer, real, and character. The structure types are record, array, and set. We will also assume the existence of certain operations on the set structures. These operations are

\[
\begin{align*}
x &:= x + y & \text{put } y \text{ in the set } x, \\
x &:= x - y & \text{remove } y \text{ from the set } x, \\
\text{empty}(x) & & \text{boolean, true if the set } x \text{ is empty, and} \\
y & \text{in } x & \text{boolean, true if } y \text{ is in the set } x.
\end{align*}
\]

The value of \( y \) in the set \( x \) is accessed by using \( x.y \) as a reference.

3.4. The Design Process and an Example

In this section, we will demonstrate the design process with an example. We will not, however, address the problems of concurrency until the next chapter. We have chosen for this demonstration a modification of an existing model of an information system (41). For our example, we will assume that an organization wishes to automate the management of personnel records by building a hierarchical information system that may be accessed from a number of terminals. The
information structures will be collected into two logical sets: employee records and department records as shown in Figure 3.7. The semantic definitions of the user language constructs must be defined over these logical collections in the definitional language (see section 2.4). We illustrate these definitions by giving two example language constructs in Figure 3.8.

From an implementation standpoint, these examples may lack some of the security and validity checks normally found in information systems. These checks only complicate this presentation and do not add significantly to this study of the design process.

![Diagram of information items for employees and departments.]

Figure 3.7. The sets of information items for employees and departments.
update(emp,input1)
    if ispath(emp-recs,emp)
        then
            apply(emp-recs.'emp',input1)
        else
            undefined
    transfer(emp,dept,input2)
        if ispath(emp-recs.'dept',emp,input2)
            then
                apply(dept-recs.'dept',emp,input2)
                apply(emp-recs.'emp',dept,input2)
            else
                undefined

Figure 3.8. The semantic definition of the 'update' and 'transfer' language constructs.

The first step for the designer is to specify the information structures as collections, or sets, of unstructured items of some named type. The argument structures are also specified, and some may be unstructured. The type and variable declarations for our example are given in Figure 3.9. The question mark in the type declaration indicates that the type is yet to be defined.
\textbf{type} emp-rec=?
\textbf{type} dept-rec=?
\textbf{type} arg1=?
\textbf{type} arg2=?
\textbf{var} emp, dept: char
\textbf{var} input1: arg1
\textbf{var} input2: arg2
\textbf{var} emp-recs: set of emp-rec
\textbf{var} dept-recs: set of dept-rec

Figure 3.9. The original type and variable declarations.

The operations 'update' and 'transfer' will operate on these structures. Let the initial state of the information structures be as given in Figure 3.10. Then Figures 3.11 and 3.12 give the final state after update and transfer have been applied, respectively.

The graphic representation of these operations (also called processes) is shown in Figure 3.13. We adopt the notational convenience and naming convention of suffixing operation names with integers, update1 and transfer1, for example. This is used to distinguish the level in which an operation is used and differentiate it from possible identical operational names in other levels. The rule for naming is to suffix the operational name op at level \( i-1 \) with the integer \( i \); thus, op becomes opi.
Figure 3.10. The initial state of information and argument structures.

Figure 3.11. Information structures after 'update'.

Figure 3.12. Information structures after 'transfer'.
Figure 3.13. The level 0 process graphs for update and transfer.

Level 0 is interpreted as a machine whose operations include 'update1' and 'transfer1', and whose machine operands have the structure of the logical information structures. The machine, as defined at level 0, may be real or virtual. If virtual, the decision may be made to realize the machine in hardware or the decision may be made to refine the machine to increase concurrency (or more closely approach a known target machine). We will assume the refinement of the operations and the information structures is desired.

In order to specify the refinement, the designer selects some logical portion of the information structure to refine, makes the refinement, and then generates the level 1 processes to support the operational definitions of level 0. Thus, the next step for the designer, in the example, is to specify the types to be refined. In this case, the types to
be defined are dept-rec, emp-rec, arg1, and arg2. In the process of refining these structures, new unknown types are introduced. The structure refinements and new type declarations are given in Figure 3.14.

The next step for the designer is to define the level 1 processes upon the structure refinements to satisfy the semantic definitions given in the higher level, in this case, 'update1' and 'transfer1'. These processes are given in Figures 3.15 and 3.16. An example of the application of update1 is given in Figures 3.18 and 3.19.

```plaintext
type arg1=record
  spouse:char
  vital-input:vital-arg
  change-spouse:(yes,no)
end

type dept-rec=record
  dep-head:char
  dept-emps:set of emp
  admin-data:admin-rec
end

type emp-rec=record
  spouse:char
  child:children
  vital-data:vital-rec
end

type dept-arg=?
```

Figure 3.14. The type declarations for level 1.
update1(x:char,y:arg1)

Figure 3.15. The level 1 procedure 'update1'.

\[ n_1: \text{x in emp-recs} \]
\[ n_2: \text{write(error)} \]
\[ n_3: \text{update-child2(emp-recs.x.children,y.child-input)} \]
\[ n_4: \text{y.change-spouse=yes} \]
\[ n_5: \text{update-vital2(emp-recs.x.vital-data,y.vital-input)} \]
\[ n_6: \text{emp-recs.x.spouse:=y.spouse} \]
transfer1(x:char,y:char,z:arg2)

Figure 3.16. The level 1 procedure 'transfer1'.
update-child2(x,y) = apply(x,y)
update-vital2(x,y) = apply(x,y)
update-dept2(x,y) = apply(x,y)

Figure 3.17. The semantic definitions of the level 1 operations.

Figure 3.18. The state of the information structures prior to update1(x,y).
Level 1 is now interpreted as a machine whose operations include 'update-child2', 'update-vital2', and 'update-dept2'. The machine operand of this level have the types 'emp', 'admin-rec', 'children', 'vital-rec', 'child-arg', 'vital-arg', and 'dept-arg'. Again, this machine may be real or virtual. We again assume virtual and that the refinement of the operations and operands is desired.

Again, the designer selects some portion of the information structure to refine, makes the refinement, and generates the level 2 procedures to support the operational definitions of level 1. The types for this level are given in Figure 3.20. The level 2 procedures are given in Figures 3.21 through 3.23. The semantic definitions of the level 2 operations 'update-child3', 'update-skill3', 'update-ed3',
and 'update-dept-emp3' are given in Figure 3.24.

At this point, we stop the design process. The process steps have been illustrated for three levels of the machine, namely, operand structure specification, process specification, and operation definition. These three steps are applied repeatedly until the structures and processes are completely supported.

```plaintext
type children=set of child

type vital-rec=record

  skills: skill-rec
  education: ed-rec
  pay-rate: real

end


type child-arg=record

  name: char
  change: (yes, no)
  child-data: child-rec

end


type vital-arg=record

  skill-data: skill-arg
  ed-data: ed-arg
  pay-change: (yes, no)
  pay-rate: real

end


type dept-arg=emp-d-arg

type child=


type skill-rec=

type ed-rec=


type emp-d-rec=

type child-rec=


type skill-arg=

type ed-arg=

Figure 3.20. The type declarations for level 2.
```
update-child2(x:children, y:child-arg)

Figure 3.21. The level 1 procedure 'update-child2'.
update-vital2(x:vital-rec,y:vital-arg)

\[ \text{Figura 3.22. The level 2 procedure 'update-vital2'.} \]

\[ n_1: \text{update-skill3}(x\text{.skills},y\text{.skill-data}) \]
\[ n_2: \text{update-ed3}(x\text{.education},y\text{.ed-date}) \]
\[ n_3: \text{pay-change=\texttt{yes}} \]
\[ n_4: x\text{.pay-rate}=y\text{.pay-rate} \]

update-dept2(x:emp,y:dept-arg)

\[ \text{Figura 3.23. The level 2 procedure 'update-dept2'.} \]

\[ n_1: \text{update-dept-emp3}(x,y) \]
update-child3(x,y) = apply(x,y)
update-skill3(x,y) = apply(x,y)
update-ed3(x,y) = apply(x,y)
update-dept-emp3(x,y) = apply(x,y)

Figure 3.24. The level 2 operational definitions.

3.5. Comments on the Design Process

3.5.1. Consistency

In his work on the semantics of relational data base system, Rawryszkiewycz (21) studied the correspondence between the relational data base model and an abstract representation of that model. The operations on the data base are defined by semantic procedures whose instructions were defined over acyclic graph models of information structures. He was interested in being able to prove that the definitions were correct, i.e. they defined exactly the effect of the data base operations on the data base information structures. In order to show correctness, he introduced the concept of consistency. We will review his definition of consistency and comment on how the designer may relate consistency to the design process.

Let the state of the data base model be given by S and let R be a representation, called an abstract representation,
of $S$. Let $H$ be a data base operation over $S$ and let $Z$ be a sequence of abstract transformations over $B$ which result from the application of the semantic procedure which defines $H$. This can be represented as in Figure 3.25. The upper path represents operations in the data base. The application of a data base operation $H$ to the data base state $S$ results in the data base state $S'$ where

$$S' = H(S)$$

The lower path represents the transformation of the abstract state by a sequence of transformations from the semantic procedure corresponding to the data base operation. The application of $Z$ to the abstract state $R$ results in the abstract state $R'$.

$$R' = Z(R)$$

Let $\mathcal{H}$ be a relation between the data base state and the abstract state; that is, $\mathcal{H}$ is defined over the pair $(S, R)$, and let $\mathcal{H}^{-1}$ be the converse of $\mathcal{H}$. The transformation of $R$ by $Z$ is said to be consistent with the transformation of $S$ by $H$, if for all initial states of $S$

$$S' = H(S) = \mathcal{H}^{-1}(Z(\mathcal{H}(S)))$$

This condition is illustrated in Figure 3.25.
In our work, we view consistency somewhat differently. Let $H$ be a process at level $i$. $H$ is constructed of operations which operate on the information structures. The effect of these operations is well defined in the definitional language. As a result of refinement and the generation of the new level $i+1$, we divide the information structures into two parts, the abstract part and the real part. The abstract part of the structures is that portion of the structures which is refined and operated on by level $i+1$ or lower. The real part is that portion not refined. For example, assume that Figure 3.26 gives a logical structure used in level $i$ and Figure 3.27 gives a refined structure used to support level $i$, then Figure 3.28 gives the abstract and real parts of the level $i$ structures.
Figure 3.26. The logical structure accessed by level $i$.

Figure 3.27. The refined structure for the type $CT$. 
Figure 3.28. The real and abstract parts of the level $i$ structures.

Let $S_i$ be the abstract part of level $i$ and let $R_{i+1}$ be the information structures of level $i+1$. Let $H$ be an operation in level $i$ defined over $S_i$. As a result of the refinement and the specification of level $i+1$, this operation will become a procedure $Z$ in level $i+1$. Let the transformation over $S_i$ by $H$ as defined in the semantic definition be defined by

$$S_i^1 = H(S_i)$$

Let the transformation over $R_{i+1}$ by the procedure $Z$ be defined by

$$R_{i+1}^1 = Z(R_{i+1})$$

Let $\mathcal{F}$ be a relation between $S_i$ and $R_{i+1}$ and let $\mathcal{F}^{-1}$ be its converse. Then we say the transformation of $R_{i+1}$ by $Z$ is
consistent with transformation of $S_i$ by $H$, if for all initial states of $S_i$

$$S_i^* = H(S_i) = \mathcal{F}^{-1}(Z(H(S_i))).$$

The difference between these two definitions is that the first is applied to the semantic definition of an operation and the second is applied to the implementation of an operational definition. The first states when a semantic definition is consistent with an operation. The second states when a procedure is consistent with its definition.

In the top-down design of a system, the second definition appears to have more application than the first. The designer may need to determine whether the procedures at a new level are consistent with their definition. It appears more critical for the designer to be able to guarantee that the implementation of the prescribed user language is consistent with the original definition. These questions are not considered in this work, since they apply to definitions of correctness other than those addressed in this work.

3.5.2. Language constructs

At this point, we discuss the types of languages to which this process can be applied. We assume that all of the operations to be supported by level 0 are to include arguments that are involved in the same action. That is, the
operations will not support lists of argument. For example, some languages have commands like

\[ \text{copy}(A_1, B_1), (A_2, B_2), (A_3, B_3). \]

This command will have the effect of copying the value from \( A_1 \) into \( B_1 \), copying the value from \( A_2 \) into \( B_2 \), and copying the value from \( A_3 \) into \( B_3 \). We interpret such a command as a macro-type command which, if expanded, would result in the sequential application of three different operations, specifically

\[ \text{copy}(A_1, B_1) \]
\[ \text{copy}(A_2, B_2) \]
\[ \text{copy}(A_3, B_3). \]

We assume that the types of user language constructs to be supported will not be of the macro-type.

The procedural language used to express the procedures at each level will also restrict the application of this design process and the use of the algorithm presented in chapter 5. If the language is restricted to a few primitive data types (real, integer, array) as in some algebraic languages, then the amount of structure refinement is severely limited. Also, the use of program branches can reduce the level of potential concurrency and complicate the development of the level programs.
4. CORRECTNESS

4.1. Introduction

In the previous chapter, we presented a top-down design methodology which produced a concurrent environment. In this chapter, we consider issues of correctness within such an environment. Specifically, we are concerned with mutual exclusion and information integrity, deadlock, and determinacy (within an individual process). Our intent is to take some existing ideas about solutions to these problems and extend them to apply to the design methodology. Using these extensions, we will present an algorithm in chapter 5, which if used as the system is designed, will increase the degree of potential concurrency while maintaining the specified correctness. The algorithm, if automated, relieves the designer of the analysis associated with these two tasks. This algorithm may also be applied to existing hierarchical systems which have the properties described in chapter 3.

In our work, we will only consider information structures which are accessed as logical resources. In a multiuser environment which shares these types of resources, uncontrolled access can result in chaos. We will discuss some specific problems in sections 4.3, 4.4, and 4.5 and comment on some limitations to solutions in section 4.6.
As each problem is considered, with its associated conditions for correctness, we will relate it to hierarchical control structures, and information structures modeled as trees.

4.2. Notation

In this section, we introduce the notation and some definitions used to discuss the problems of concurrency. The notation will be similar to our representations of processes and information structures. It is motivated by what appears to be the most important applications to our problems.

4.2.1. Process graphs

When considering the properties of correctness, we will discuss the action and event sequences generated by the process graphs used to define the level processes.

We will also assume that each level may be analyzed as a set of independent processes and that a sufficient number of copies of each process is available to satisfy all users.

When considering the interface between two levels, we interpret the nodes of the higher level processes differently than we interpret the nodes of the lower level processes. The nodes of the higher level are interpreted as operations,
only when they do not use a low level process. If a token enters a node which represents the use of a low level process, it is interpreted as the initiation of the low level process. We represent this initiation by placing a token in the start node of a copy of that process. When a token enters a stop node of a low level process, it is interpreted as the termination of the process. We represent this interaction by a set of the event sequences for both processes. For example, if we had the high level and low level processes as in Figure 4.1, then we could have the event sequences

\[ E = \overline{\text{n}_1} \text{L}_1 \overline{\text{n}_2} \text{L}_2 \]

for H and

\[ G = \overline{\text{m}_1} \text{M}_1 \]

for L.

![Diagram of processes H and L](image)

**Figure 4.1.** An example high level process H which uses the low level process L.

Given the processes in Figure 4.1, we say that \( n_1 \) is a **predecessor** of \( n_2 \) and that \( n_2 \) is a **successor** of \( n_1 \). We say
that the nodes on the separate paths emanating from a decider are **alternatives**. In Figure 4.1, $n_2$ is an alternative to $n_3$.

### 4.2.2. Information structures

When discussing the effect of the processes on the state variables (the information structures), we may use an alternate representation to the graphs, the **list form**. We represent a value node and its list form in Figure 4.2. If a value node has not been assigned a value, then its state is represented as in Figure 4.3. The list form of a labelled arc and value node is given in Figure 4.4.

\[12\]

Figure 4.2. A value node and its list form representation.

\[()\]

Figure 4.3. A valueless value node and its list form.
Figure 4.4. A labelled arc and value node with list form.

We represent a branch node by giving the emanating labelled arcs and the list form of the component nodes as in Figure 4.5. A branch node with no arcs emanating from it is represented in Figure 4.6. Figure 4.7 gives an example of a tree structure and its list form. It is noted that the order in which the arcs are specified is not important. Another list form for Figure 4.7 is

\[ <A,C,E,G(2),F(),D\rightarrow,B(1)> . \]

Figure 4.5. A branch node and list form.
Figure 4.6. A labelled arc and empty branch node.

\[ \langle A<B(1), C<D<->, E<G(2), F(\rangle \rangle \rangle \]

Figure 4.7. An example structure and a list form.

As indicated in chapter 3, when one level is being specified some of the types may be unstructured. An example
of unstructured types and their list form is shown if Figure 4.8. The states of these types, $t_1$ and $t_2$, are called symbolic states.

These symbolic states may be manipulated as other states. For example, if $A.CS.C1$ in Figure 4.8 was assigned the state of $A.CS.C2$, then the new state would be given by

$$<A<B(1), CS<C1(t_1), C2(t_2)>>$$

Figure 4.8. An information structure with nodes of unstructured types.

The application of operations to unstructured nodes is indicated by superscripts. The superscript value of $t_i$ is assumed to be $t_i^0$ and with every application of an operation which could modify the unstructured node, we increment the superscript. For example, if $A.CS.C1$ in Figure 4.8 is modified by some operation, then the state would be given by
The state of a path name, if it terminates within the structure, will be the state of the node to which the path name refers. Thus, in Figure 4.8, the state of A.B is \((1)\) and the state of A.CS is \(<C1(t_1),C2(t_2)>\). The state of a path name which does not exist in the structure is \(*\); this would be the state of the path name A.D in Figure 4.8. The state of a path name which extends beyond the structure into an unstructured node is given a unique symbolic state and is manipulated as mentioned above. Thus, in Figure 4.8 A.CS.C1.E.F may be assigned the state \(t_3\).

4.2.3. Communicating processes

One of the complicating factors in our work is that the processes of any two adjacent levels can communicate. When a process uses a process at the next lower level, it relinquishes control while the lower level process executes. As a consequence, if a level has been shown to exhibit a property of correctness when its actions are analyzed as operations, this property may not be guaranteed when those actions are supported by lower level processes.

Another analytic problem is that if we analyze the lower level as a system independent from the higher, then we can only analyze the lower level in terms of the variables named
in those processes. A system of processes at a low level may possess a property with respect to its local names, but may not possess that property as a result of the states assigned to their formal parameters by the user processes.

For the above reasons, we introduce the following notation which can be used when discussing the interaction between two levels.

Let $HS$ be the information structures as defined at a high level. As noted in chapter 3, these structures are partitioned into two parts, the real part which is modified by the high level processes and the abstract part which is modified by using a low level process. Let $HS_1$ be the real part and $HS_2$ be the abstract part. For example, let the high level structures be represented by

$$HS = \langle A<FN(N),CTS<B(t_1),C(t_2)\rangle \rangle.$$ 

Then, the real and abstract parts would be represented as

$$HS_1 = \langle A<FN(N)\rangle$$

and

$$HS_2 = \langle A<CTS<B(t_1),C(t_2)\rangle \rangle.$$ 

When a low level process is used, portions of the abstract part are passed as arguments from the higher level. As a result of the typing at the low level, a structure may be imposed on some unstructured nodes, (this corresponds the relation $\mathcal{G}$ mentioned in section 3.5.1). This imposed structure may result in the introduction of state variables.
We let LS represent the information as structured in the lower level. For example, if A.CTS.B is passed as an argument to the low level, then it could be represented, depending on the structuring, as

\[ LS = <A<CTS<B<TY(1), E(t_3), F(4)>>. \]

As a result of renaming in a low level process, this structure may be operated on as if it had the structure

\[ <x<TY(1), E(t_3), F(4)>>. \]

In the above example, we would represent this mapping by

\[ \mathcal{F}(<A<CTS<B>) = (<x<TY,E,F>). \]

The state \( s(<x<TY,E,F>) \), in this case, is given by \((1), (t_3), (4))\). The converse of this relation is given by

\[ \mathcal{F}^{-1}(<x<TY,E,F>) = (<A<CTS<B>>). \]

We also define the relation over the states. That is, if

\[ \mathcal{F}(x) = <x_1, \ldots, x_k>, \]

then

\[ \mathcal{F}(s(x)) = <s(x_1), \ldots, s(x_k)>. \]

4.3. Determinacy

4.3.1. Formal definition of determinacy

In this section, we extend the definitions and the approach to determinacy presented by Bernstein (3) to our
design methodology. We will present the definitions as they apply to tree structured operands and hierarchical control structures.

Let the logical information system on which the processes of a level operate be represented by the set of path names

\[ P = \{P_1, \ldots, P_m\} \]

where each path name can have the states as described in the previous section. As the processes of a level execute, these states may change. Let

\[ E = e_1 e_2 \cdots e_{2n} \]

be an event sequence of a process at a level where

\[ S = s_0 s_1 \cdots s_{2n} \]

is the corresponding state sequence for the path names. If \( P_i(k) \) is the state of a path name \( P_i \) immediately following the event \( e_k \), then \( s_k \) is defined as

\[ s_k = (P_1(k), P_2(k), \ldots, P_m(k)). \]

Associated with each action are two subsets of \( P \) called the domain \( D \) and the range \( R \) of the action. The range of an action is that subset of \( P \) which the action modifies and the domain is that subset which it reads. These sets may intersect.

For example, assume we have the process graph as given in Figure 4.9. Then, the ranges and domains for \( L_1 \) and \( L_2 \) are
\[ D_1 = \{A, C\}, \ R_1 = \{A, B\}, \]
\[ D_2 = \{A, F\}, \ R_2 = \{A, E\}. \]

Figure 4.9. A level process with two actions.

1. Assign \((A, B, \text{val}(A, C))\)
2. Assign \((A, E, \text{val}(A, F))\)

Let the initial state of the structure be given by
\[ <A\langle B(1), C(2), E(3), F(4)\rangle>. \]

Then the initial state is given by
\[ S_0 = s_0(<A\langle B, C, E, F\rangle>) = ((1), (2), (3), (4)). \]

If these two actions in Figure 4.9 execute concurrently, then two possible event sequences are given by \(E_1\) and \(E_2\)
where
\[ E_1 = \overline{L_1}L_1\overline{L_2}L_2, \]
and
\[ E_2 = \overline{L_1}L_2\overline{L_2}L_1. \]
The corresponding state sequences are given by
\[ S_1 = (((1),(2),(3),(4)), ((2),(2),(3),(4)), ((2),(2),(3),(4))) \]
for \( E_1 \) and
\[ S_2 = (((1),(2),(3),(4)), ((1),(2),(3),(4)), ((1),(2),(4),(4))) \]
for \( E_2 \).

Given the event sequences, the domains, and the ranges of the actions of the system, we define the value sequence of a path name. Let

\[ V(P, e_0 e_1 \ldots e_k) \]

be the state of the value sequence for the path name \( P_i \) after the event \( e_k \), then we define the value sequence of the path name \( P_i \) by

\[
V(P, e_0 e_1 \ldots e_k) = \begin{cases} 
V(P, e_0 e_1 \ldots e_{k-1}) & \text{if } e_k = e_j \text{ and } P_i \text{ in } R_j \\
V(P, e_0 e_1 \ldots e_{k-1}) & \text{otherwise.} 
\end{cases}
\]

If the event sequence is given by
\[ E = e_0 e_1 \ldots e_k , \]
then
\[ V(P, e_0 e_1 \ldots e_k) \]
will have one entry for each action that has \( P_i \) in its range and those entries will be the state assigned to \( P_i \) by the
action. The value sequence for \( A.B \) from above is given by

\[
\begin{align*}
V(A.B, e_0) &= (1) \\
V(A.B, e_0e_1) &= V(A.B, e_0) = (1) \\
V(A.B, e_0e_1e_2) &= V(A.B, e_0e_1) = (1) \\
V(A.B, e_0e_1e_2e_3) &= V(A.B, e_0e_1e_2) = (1) \\
V(A.B, e_0e_1e_2e_3e_4) &= (V(A.B, e_0e_1e_2e_3), S_4(A.B)) = (1, 2).
\end{align*}
\]

We may also represent the value sequence by specifying it in terms of the event sequence; for example,

\[
V(A.B, E_1) = (1, 2).
\]

The value sequences for the path names in Figure 4.9 as a result of \( E \) are given by

\[
\begin{align*}
V(A.B, E_1) &= (1, 2), \\
V(A.C, E_1) &= (2), \\
V(A.E, E_1) &= (3, 4), \\
V(A.F, E_1) &= (4).
\end{align*}
\]

When a system of processes is executed concurrently, there exists a set \( E \) of all valid event sequences that can be generated. In this work, any member of the set \( E \) is also called a realization. The event sequences \( E_1 \) and \( E_2 \) are realizations for the process in Figure 4.9, while the sequence \( \overline{I_1L_2L_2I_1} \) is not. Determinacy is defined in terms of the realizations and value sequences.
**Determinate:**

A system of processes $P$ is **determinate**, if for any given initial state $s_0$, $V(P_i,E) = V(P_i,E')$ for all realizations $E$ and $E'$. 

Functionality of a system is defined in terms of the realizations and the final states. Let $F(P_i,E)$ be the final state of $P_i$ when $E$ completes.

**Functional:**

A system of processes $P$ is **functional**, if for any given initial state $s_0$, $F(P_i,E) = F(P_i,E')$ for all realizations $E$ and $E'$.

Let $P$ and $P'$ be path names of the form

$$P = u_1 \cdot u_2 \cdots u_n$$

and

$$P' = v_1 \cdot v_2 \cdots v_m$$

We say that $P$ **contains** $P'$, if $n \leq m$ and $v_i = u_i (i \leq n)$. For example, if $P = A$ and $P' = A.B$, then we say $P$ contains $P'$.

Let $X$ and $Y$ be two sets of path names, then define

$$X\|Y = \{ x_i \text{ in } X \mid x_i \text{ contains } y_j \text{ for some } y_j \text{ in } Y\}$$

and

$$Y\|X = \{ y_j \text{ in } Y \mid y_j \text{ contains } x_i \text{ for some } x_i \text{ in } X\}.$$
Path Intersection:

The path intersection of the sets of path names X and Y is represented by $X\sim Y$ and defined by the set union of $X\mid Y$ and $Y\mid X$, that is,

$$X\sim Y = (X\mid Y)\cup(Y\mid X)$$

For example, let X and Y be defined by

- $X = \{A, C, B, C\}$
- $Y = \{A, B, W, C\}$

then

- $X\mid Y = \{B, C\}$
- $Y\mid X = \{A, C\}$

and

- $X\sim Y = \{A, B, C\}$. Using these definitions, we now modify Bernstein's (3) definition of noninterference.

Noninterference:

Process $H_1$ and $H_2$ are noninterfering if either

1) $H_1$ is a successor, predecessor, or alternative of $H_2$, or

2) $H_1 \sim H_2 = H_1 \sim D_2 = H_2 \sim D_1 = \emptyset$.

That is, potentially concurrent processes may only share elements in their domains and be noninterfering.
With this definition of noninterference, we have Bernstein's theorem on determinacy.

**Theorem 4.1:**

A system of processes consisting of mutually noninterfering processes is determinate.

A level is determinate (functional), if all the processes of that level are determinate (functional).

**4.3.2. Determinacy and communicating levels**

In this section, we investigate determinacy as applied to communicating processes between two levels. We consider a high level which is analyzed to be determinate when its actions are interpreted as operations and then consider the possible results when the actions are interpreted as uses of low level processes.

Let a determinate high level have the set of processes

\[ H = \{H_1, H_2, \ldots, H_n\} \]

and the low level have the set of processes

\[ L = \{L_1, L_2, \ldots, L_m\} \]

Let \( H_1 \) be defined as in Figure 4.10, and let the structure have the state

\[ \langle A, B(t_1), C(t_2), D(t_3), E(t_4) \rangle \]

The ranges and domains of the actions of \( H_1 \) are given by
The value sequences, for any realization $E$, will be

$$V(A, B, E) = (t_1),$$
$$V(A, C, E) = ((t_2), (t_1)),$$
$$V(A, D, E) = ((t_3), (t_4)),$$
$$V(A, E, E) = ((t_4)).$$

Let $H_1$ use $L_1$ and let the mapping of the structures be given by

$$f(<A<B>>, <A<C>>) = (<x_1>, <x_2>)$$

and the state be given by

$$f(s_0(<A<B>>, <A<C>>)) = s_0(<x_1>, <x_2>)$$
$$= (<F(1)>, <G(2)>)$$

That is,
\[ \mathcal{H}(t_1, t_2) = \mathcal{H}(s_0, (A<B>, A<C>)) = s_0(x_1, x_2) = (\langle F(1) \rangle, \langle G(2) \rangle) \]

and

\[ V(\mathcal{H}(A.B), E) = V(x_1, E) = (\langle F(1) \rangle) \]
\[ V(\mathcal{H}(A.C), E) = V(x_2, E) = (\langle G(2) \rangle, \langle F(1) \rangle) \]

Given \( L_1 \) to be defined as in Figure 4.11, then it is possible for \( L_1 \) to generate the following event sequences \( E_1 \) and \( E_2 \) where

\[ E_1 = \overline{n}_0 \overline{n}_1 \overline{n}_2 \overline{n}_3 \overline{n}_4 \overline{n}_5 \overline{n}_6 \overline{n}_7 \overline{n}_8 \overline{n}_9 \overline{n}_10, \]
and

\[ E_2 = \overline{n}_0 \overline{n}_1 \overline{n}_2 \overline{n}_3 \overline{n}_4 \overline{n}_5 \overline{n}_6 \overline{n}_7 \overline{n}_8 \overline{n}_9 \overline{n}_10 \overline{n}_11 \overline{n}_12 \overline{n}_13 \overline{n}_14 \overline{n}_15 \overline{n}_16 \overline{n}_17 \overline{n}_18 \overline{n}_19 \overline{n}_20. \]

The value sequences for \( E_1 \) are

\[ V(\mathcal{H}(A.B), E_1) = V(x_1, E_1) = (\langle F(1) \rangle) \]
\[ V(\mathcal{H}(A.C), E_1) = V(x_2, E_1) = (\langle G(2) \rangle, \langle F(1) \rangle) \]

and for \( E_2 \) are

\[ V(\mathcal{H}(A.B), E_2) = V(x_1, E_2) = (\langle F(1) \rangle) \]
\[ V(\mathcal{F}(A,C), E_2) = V(x_2, E_2) = ((<G(2)> , <->, <F()> , (<P()> , <F()> ,
(<P(1)> , F()> , (<P(1)> , <F(1)> ))) . \]

\begin{itemize}
  \item \( n_0 \): erase (x1)
  \item \( n_1 \): x := 0
  \item \( n_2 \): getnext (x2, x, y)
  \item \( n_3 \): x := y
  \item \( n_4 \): y := 0?
  \item \( n_5 \): putlist (x1, 'y')
  \item \( n_6 \): assign (x1, 'y', val(x2, 'y'))
  \item \( n_7 \): getnext (x2, x, z)
  \item \( n_8 \): x := z
  \item \( n_9 \): z := 0?
  \item \( n_{10} \): putlist (x1, z)
  \item \( n_{11} \): assign (x1, 'z', val(x2, 'z'))
\end{itemize}

Figure 4.11. The low level process L1.
In the first case, the value of \( h(A, B) \) was copied into \( h(A, C) \). In the second a different value was written into \( h(A, C) \). This is an example of a low level process that terminates with indeterminate results.

An analysis of \( L_1 \) shows it to be constructed of processes which do not satisfy the noninterference condition. In order to maintain the determinacy of a high level process, we must guarantee that the value sequences of the high level process are equal to those generated when the low level processes are used. It follows directly that we must be able to guarantee consistent low level processes if the high level is to be determinate. Assume \( H \) is a high level process which is analyzed to be determinate when its nodes are interpreted as operations. Let \( n \) be any node in \( H \) with \( Y = (R, D) \) and \( Y' = (P', D') \) as the range and domain when \( n \) and \( n \) occur, respectively. Let \( n \) use the functional low level process \( L \) and let there exist at least one realization \( E \) of \( L \) such that for all \( x \) in \( R \cup D \)

\[ f(s'(x)) = F(s(x), E) \]

where \( s(x) \) and \( s'(x) \) are the states of \( x \) in \( Y \) and \( Y' \), respectively. If this is true for all nodes in \( H \), then \( H \) is determinate.

The importance of this fact to the designer is that if a high level process is analyzed to be determinate when its
nodes are interpreted as operations, then this property is guaranteed if the processes it uses are functional.

Since determinacy implies functionality, then the processes of a high level in a hierarchical structure are determinate if supported by lower levels which are determinate.

4.4. Mutual Exclusion and Integrity

4.4.1. Independent processes

In the last section, noninterference was introduced as means of defining the sufficient conditions for determinacy within a computation. In this section, we consider the application of the noninterference condition to the processes of a given level and the immediate lower level.

An example of interfering processes at the same level is given in Figure 4.12. Let the initial state of the structure is given by

\[ <A^B(1), C(2)>, <D^E(3), F(4)>, <G^H(5), I(6)> \]

The final state, with the action sequence \( n_1 n_2 n_3 n_4 \), would be

\[ <A^B(5), C(6)>, <D^E(3), F(4)>, <G^H(5), I(6)> \]

With the action sequence \( n_3 n_4 n_1 n_2 \), however, the final state would be

\[ <A^B(3), C(4)>, <D^E(3), F(4)>, <G^H(5), I(6)> \].
Either of these two final states, as we will see later, will be considered correct. If, however, the action sequence was $n_1 n_3 n_4 n_2$, then the final state would be given by
This final state will not be accepted. The motivating reason is that the operations of the two processes were intermixed and both had common variables in their ranges. In order for these processes to execute and guarantee one of the two acceptable states, there must be some method to guarantee the mutual exclusion of these processes. That is, when one is operating on a variable the other can not gain access to that variable.

\[
H_1 \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad H_2
\]

\[
\begin{array}{c}
n_1 \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad n_3 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad n_4 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad n_2
\end{array}
\]

$n_1$: assign(A.B,val(D.E)) \quad \quad n_3$: assign(A.B,val(G.H))

$n_2$: assign(A.C,val(D.F)) \quad \quad n_4$: assign(A.C,val(G.I))

Figure 4.12. A set of interfering processes.

Sections of processes to be executed mutually exclusively are called critical regions (29,9,14). In this
work, we shall assume that mutual exclusion must be implemented by the processes themselves, using globally available control mechanisms. We will discuss this topic in section 4.4.3.

There are times when processes may have overlapping ranges and mutual exclusion need not be guaranteed. For example, consider Figure 4.13. The domains and ranges of these processes are

\[
D_1 = \{A\}, \quad R_1 = \{A, C\}, \\
D_2 = \{A, B\}, \quad R_2 = \{A, B\}.
\]

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[draw] (n1) at (0,0) {}; \node[draw,above] at (n1) {$n_1$};
\node[draw,above right] at (n1) {$n_2$}; \node[draw,above left] at (n1) {$n_3$}; \node[draw,above right] at (n1) {$n_4$};
\node[draw,above] at (n1) {$H_1$}; \node[draw,above] at (n1) {$H_2$};
\node[draw,above] at (n1) {$n_1$: ispath(A)?}; \node[draw,above] at (n1) {$n_3$: ispath(A,B)?}; \node[draw,above] at (n1) {$n_2$: alldpath(A,C)}; \node[draw,above] at (n1) {$n_4$: assign(A,B,'1')}$
\end{figure}

Figure 4.13. An example of interfering processes.

From the definition of containment, we say that A contains A.B, thus we would say that these two processes do not satisfy the noninterference condition. We will see, however, these two processes do guarantee a property called information integrity. We will consider this property in section 4.4.4, but it is motivated by the desire to allow
processes to share a data item if that sharing will not cause unreliable results for either of the two processes. A classic example of this type of sharing is the problem of concurrent readers and exclusive writers (11). Informally, the problem is to control access to shared item in such a manner as to allow reading processes concurrent access. If, however, some process wishes to write the item, it must be given exclusive access.

We solve these, and other problems, by assuming the existence of a globally accessible collection of data and procedures known as a monitor (6,22). It presents a method of controlling access to shared information structures and preserving certain properties of correctness. We address the topic of monitors in the section 4.4.2 and chapter 7.

4.4.2. Monitors

The concept of a monitor was introduced by Brinch Hansen (5,6) and developed by Hoare (22) as a method of structuring operating systems. The monitors were designed primarily to control the allocation of a class of shared resources (main store, backing store, consoles, etc.). A monitor consists of local data structures and a set of processes used by other processes wishing to acquire and release access capabilities to resources.
The processes of a monitor are similar to other processes, in the sense that any process may attempt to use a monitor. It is, however, imperative that only one process at a time actually succeeds in entering a monitor, and any later uses must be held up until the previous use has been completed. Otherwise, if two processes were serviced simultaneously, the effects on the local information of the monitor may be erroneous. The processes local to a monitor should not access any information structures other than those local to the monitor and the structures local to the monitor should be inaccessible to any process outside of the monitor. With these restrictions, it is possible to guarantee certain properties about the operating system.

There will be times that it will be necessary for a monitor to delay a process requesting a resource until some other process returns the required resource. For this reason, operations are will be used which control the entry to the monitor by other processes. These operations are traditionally called the P and V operations. These operations are defined over unique variable types called semaphores and on which no other operations may be performed (17,8,6).

The effect of the P and V operations can be described in terms of the values of the special variables and a set of waiting processes. Let S be a semaphore and Q a set of
waiting processes associated with the semaphore, then the effect of the operations can be defined for the process \( x \) by

\[
P(S): \quad S \leftarrow S - 1 \\
\text{if } S < 0 \text{ then put } x \text{ in } Q
\]

\[
V(S): \quad S \leftarrow S + 1 \\
\text{if } S < 0 \text{ then get some } y \text{ from } Q \\
\text{restart } y.
\]

An example of a monitor and the use of the semaphores is given in Figure 4.14.

```plaintext
begin count: integer;
    sem1, sem2: semaphore;
    "initially sem1=1, sem2=0, and count=0"

procedure request
    begin
        P(sem1)
        count = count + 1
        if count > 1 then V(sem1)
            else V(sem1)
        P(sem2)
    end

procedure release
    begin
        P(sem1)
        count = count - 1
        if count > 0 then V(sem2)
            V(sem1)
        end

end
```

Figure 4.14. An example monitor.

We will depart from the definition of a monitor and allow multiple copies of the processes of a monitor but only one copy of the semaphores. This will allow many processes
to be using monitor at the same time which can increase the degree of potential concurrency.

We will also consider the processes of a monitor to be developed top-down in the same manner the system was developed. This will mean that the monitor processes may be distributed throughout the hierarchical structure.

We can represent the use of the monitor by including it in process graphs using the request and release nodes. For example, in Figure 4.15, a process is shown which uses a monitor. The action sequences this process can generate are

\[ n_1, a_1, n_2, f_1, n_3 \]

or

\[ n_1, a_1, n_2, f_2, n_4 \]

We will use \( A \) to represent the request for the access capabilities to an information structure and \( F \) for returning the capabilities. When we consider event sequences, we will associate two events with \( A \), symbolized by \( \bar{A} \) and \( A \), corresponding to the request and the granting of the access capability. We associated only one event with returning the capabilities symbolized by \( F \). Thus, an event sequence for the process is Figure 4.15 is given by

\[ \bar{n}_1 a_1 A_1 \bar{n}_2 n_2 f_1 \bar{n}_3 n_3. \]

As we will see later, we will need a monitor which can control the access capabilities to multiple information structures. It will be assumed until chapter 7 that such
monitors exist

4.4.3. Mutual exclusion

In Figure 4.16, there are two interfering processes. Both these processes access A.C and, hence, have a critical region with respect to A.C. The following algorithm determines the subgraphs which define the critical regions.

![Diagram](image)

Figure 4.15. An example of a process using a monitor.
Algorithm 1:

Let \( H = \{ H_1, \ldots, H_m \} \) be a set of process graphs which share \( x_i \) of \( X = \{x_1, \ldots, x_n\} \). Define the subgraphs \( H'(x_i) = \{H_1'(x_i), \ldots, H_m'(x_i)\} \) of \( H \) by the following algorithm.

1) \( j = 1 \).
2) Let \( H_j'(x_i) = H_j \).
3) If there exists any initial, or terminal, node \( n_k \) of \( H_j'(x_i) \) such that \( x_i \sim (R(n_k) \cup D(n_k)) = \phi \), then remove the node \( n_k \) and all emanating, or entering, arcs.
4) Continue 3 until no more nodes can be removed.
5) \( j = j + 1 \).
6) Continue 2 through 5 until \( j > m \).

Figure 4.16. Two interfering processes at the same level.
The final graph \( H_j(x_i) \) will be the smallest subgraph of \( H_j \) containing all the nodes accessing \( x_i \) and their connecting arcs.

**Critical Regions:**

The graph \( H'(x_i) \) is called the critical region of \( H \) with respect to \( x_i \). For example, given the process graph \( H \) in Figure 4.17, \( H'(x) \) and \( H'(y) \) are given in Figure 4.18 and 4.19.

\[
D_1 = (x), \quad R_1 = ( ) \quad D_2 = ( ), \quad R_2 = (y) \quad D_3 = (y), \quad R_3 = (x) \quad D_4 = (x), \quad R_4 = (z) \quad D_5 = (z), \quad R_5 = ( )
\]

Figure 4.17. A process graph.

\[
H'(x)
\]

\[
D_1 = (x), \quad R_1 = ( ) \quad D_2 = ( ), \quad R_2 = (y) \quad D_3 = (y), \quad R_3 = (x) \quad D_4 = (x), \quad R_4 = (z) \quad D_5 = (z), \quad R_5 = ( )
\]

Figure 4.18. The critical region \( H'(x) \).
We represent the action sequences generated in a critical region \( H'(x) \) by the set \( G'(x) \). For the critical region in Figure 4.18,

\[
G'(x) = \{ n_1 n_2 n_4, n_1 n_3 \}.
\]

Let the \( C(x) \) be a symbolic sequence which can represent any member of \( G'(x) \). This sequence \( C(x) \) is called the **characteristic critical sequence**. The nodes in a critical region are called **critical nodes**. The initial and terminal nodes of a critical region are referred to as the **initial critical nodes** and the **terminal critical nodes**, respectively. In Figure 4.18, \( n_1 \) is an initial critical node and \( n_3 \) and \( n_4 \) are terminal critical nodes. In Figure 4.19, \( n_2 \) and \( n_3 \) are initial and terminal critical nodes.

We will use the characteristic critical sequence to represent any action sequence generated by a critical region. Thus, given the action sequence \( n_1 n_3 n_5 \) for Figure 4.17, \( C(x) n_5 \) represents the action sequence with respect to the critical region of \( x \) and \( n_1 C(y) n_5 \) represents the action sequence with respect to the critical region of \( y \).
We will also represent all possible action sequences with respect to a given critical region by \( a(x)C(x)b(x) \). This is called the characteristic action sequence of the process with respect to the critical region \( H(x) \). Any sequence of actions of a process executed prior to the critical region of that process is represented by \( a(x) \), while \( b(x) \) represents any sequence of actions following the execution of the critical region. In this sequence, any of the three subsequences \( a(x), C(x), \) or \( b(x) \) may be the null sequence. The characteristic action sequence for \( x \) in Figure 4.17 is given by

\[
a(x)C(x)b(x) = \{C(x)_{n_5}, C(x)\}.
\]

We represent the realization of a process by representing the characteristic action sequences in the event form

\[
a(x)\bar{a}(x)\bar{C}(x)\bar{C}(x)\bar{b}(x)b(x).
\]

Let \( D(H'(x)) \) and \( R(H'(x)) \) be the union of the domains and the union of the ranges of the nodes in the critical region \( H'(x) \). In Figure 4.17, we have

\[
D(H'(x)) = \{x, y\}, \\
R(H'(x)) = \{x, y, z\}, \\
D(H'(y)) = \{y\}, \\
R(H'(y)) = \{x, y\}.
\]

Assume \( S_0(x) = (4) \) prior to the execution of the processes in Figure 4.20. If the processes were realized by the action sequence \( n_1n_2n_3 \), then the value 4 would be written and the final state of \( x \) would be (7). If the processes were realized
by the action sequence $n_3 n_1 n_2$, then the value 5 would be written and the final state of $x$ would be 0. If, however, the sequence $n_1 n_3 n_2$ were executed, then 4 would be written and the final value of $x$ would be 0. Some information was lost as a result of interference between the two processes. In order to avoid such interference, the concept of mutual exclusion was introduced (17,8,10).

Figure 4.20. A set of interfering processes.

**Mutual Exclusion:**

Let $H'(x_i) = \{(H_1'(x_i), \ldots , H_m'(x_i))\}$ be the critical regions of $H = \{H_1, \ldots , H_m\}$ with respect to $x_i$ of $X = \{x_1, \ldots , x_n\}$. Let the characteristic action sequence of $H_x$ for $x_i$ be given by $a_r(x_i)C_r(x_i)b_r(x_i)$. If for every concurrent realization of $H_x$ and $H_s$ and any initial state $s_0(x_i)$ of $x_i$, either
1) \( C_R(x_i) \) precedes \( C_S(x_i) \), or
2) \( C_S(x_i) \) precedes \( C_R(x_i) \).

then, we say that \( H_R \) and \( H_S \) are mutually exclusive with respect to \( x_i \). If \( H_R \) and \( H_S \) are mutually exclusive with respect to all \( x_i \) of \( X \), then we say that \( H_R \) and \( H_S \) mutually exclusive with respect to \( X \).

Consider the following monitor, designed to ensure mutual exclusion of any \( x_i \) of \( X \).

**Monitor 1:**

Let \( E=e_1 e_2 \ldots e_n \) be any concurrent realization of the processes in \( H=\{H_1, \ldots , H_m\} \) which share the set of path names \( X=\{x_1, \ldots , x_n\} \). Define the following sets and events.

- \( B(x_i) \): processes which have access capability to \( x_i \)
- \( W(x_i) \): processes waiting on access capability to \( x_i \)
- \( K(e_j) \): the \( x_i \) to which access capability is granted when \( e_j \) occurs
- \( A(H(x_i)) \): the request for access capability to \( x_i \) by \( H \)
- \( A(H(x_i)) \): the granting of access capability to \( x_i \) to \( H \)
- \( F(H(x_i)) \): the return of the access capability to \( x_i \) by \( H \)

It is assumed that if \( e_j=F(H(x_i)) \), then there exists some \( e_k=A(H(x_i)) \) where \( k<j \). We require the monitor to have the following properties. Initially, let \( K(e_1)=\emptyset \) and \( B(x_i)=W(x_i)=\emptyset \) for all \( x_i \) in \( X \).

1) If \( e_j=A(H(x_i)) \) and \( x_i \sim K(e_j)=\emptyset \)
then
\[ e_{j+1} = A(H(x_i)) \]
\[ K(e_{j+2}) \cap K(e_{j+1}) = K(e_j) \cup x_i \]
\[ B(x_i) = H \]

2) If \( e_j = A(H(x_i)) \) and \( x_i \sim K(e_j) \neq \emptyset \)
then
\[ W(x_i) = W(x_i) \cup H \]
\[ K(e_{j+1}) = K(e_j) \]

3) If \( e_j = F(H(x_i)) \)
then
\[ B(x_i) = \emptyset \]
if some \( H' \) in \( W(x_i) \)
then
\[ W(x_i) = W(x_i) - H' \]
\[ E(x_i) = H' \]
\[ e_{j+1} = A(H'(x_i)) \]
\[ K(e_{j+1}) = K(e_j) \]
else
\[ K(e_{j+1}) = K(e_j) - x_i \]

4) If \( e_j \) any other event
then
\[ K(e_{j+1}) = K(e_j) \]

An example of the use of this monitor is given in Figure 4.21. A possible concurrent realization for these two processes is given by
\[
\overline{A}(H_1(x)) \quad A(H_1(x)) \quad \overline{A}(H_2(x)) \quad \overline{n}_1 \quad n_1 \quad \overline{n}_2 \quad n_2 \quad F(H_1(x))
\]
\[
\overline{A}(H_2(x)) \quad \overline{n}_3 \quad n_3 \quad F(H_2(x))
\]

Figure 4.21. Two processes using Monitor 1.

**Proposition 4.2:**

Let \( H = \{H_1, \ldots, H_m\} \) be a set of processes such that the characteristic action sequence of \( H_r \) is such that \( A_r(x_i) \) precedes \( C_r(x_i) \) which precedes \( F_r(x_i) \). If every characteristic action sequence for \( x_i \) of \( X = \{x_1, \ldots, x_n\} \) and all \( H_r \) of \( H \) is of this form, then the processes \( H = \{H_1, \ldots, H_m\} \) are mutually exclusive with respect to \( x_i \).

**Proof:**
Assume there exists some concurrent realization $E$ of $H$ such that $H$ is not mutually exclusive with respect to $x_i$. Then for some $H_r$ and $H_s$, $C_r(x_i)$ precedes $C_s(x_i)$ which precedes $C_r(x_i)$. Thus, $e_k^r = A_r(x_i)$ and $e_j^s = A_s(x_i)$ precede $C_s(x_i)$. Then $e_{k+1}^r = A_r(x_i)$ and $K(e_{k+1}^r) = K(e_k^r) \cup x_i$. As a result, $A_s(x_i)$ can not occur until after $F_r(x_i)$ which precedes $C_s(x_i)$. Thus, $C_r(x_i)$ precedes $C_s(x_i)$ and no such sequence $E$ exists and $H$ is mutually exclusive with respect to $x_i$.

This proposition just says that if the critical regions $H'_i(x_i) = (H'_1(x_i), \ldots, H'_m(x_i))$ of $H = \{H'_1, \ldots, H'_m\}$ with respect to $x_i$ of $X = \{x_1, \ldots, x_n\}$ are entered after a request use of monitor 1 and a release use occurs after exiting the critical section, then mutual exclusion is guaranteed.

This approach works as long as we do not try to acquire access capabilities to more than one structure. Consider Figure 4.22. A possible partial concurrent realization of these two processes could be

$$A_1(H_1(x))A_1(H_1(x))A_3(H_2(y))A_3(H_2(y))A_4(H_2(x))A_2(H_1(y)) \ldots$$

In this case, the processes have reached a state known as deadlock. Neither can continue until the other has released it access capability, which neither is able to do. For this reason, we will need another monitor which will allow requests for multiple structures. We also need to consider the combined critical regions of a process.
Figure 4.22. Processes accessing two shared structures which use Monitor 1.

Algorithm 2:

Let $H=\{H_1, \ldots, H_m\}$ be a set of process graphs which represent the set of processes which share the information structures $X=\{x_1, \ldots, x_n\}$. Define the subgraphs $H'=\{H'_1, \ldots, H'_m\}$ of $H$ by the following algorithm.

1) $j=1$,
2) $H'_j=H_j$,
3) If there exists any initial, or terminal, node $n_k$ of $H'_j$ such that if $X=(R(n_k) \cup D(n_k)) = \emptyset$, then remove
node $n_k$ and all emanating, or entering, arcs,

4) Continue 3 until no more nodes can be removed from $H'_{j}$.

5) $j = j + 1$.

6) Continue 2 through 5 until $j > m$.

The $H' = \{H'_1, \ldots, H'_m\}$ are the critical regions of $H = \{H_1, \ldots, H_m\}$ with respect to $X = \{x_1, \ldots, x_n\}$.

The critical regions $H'_{\lambda} = \{H'_1, H'_2\}$ for $H = \{H_1, H_2\}$ with respect to $X = \{y, z\}$ in Figure 4.23 are given in Figure 4.24.

For $H_\lambda$, let $a_\lambda(X)C_\lambda(X)b_\lambda(X)$ be the characteristic action sequence with respect to $X = \{x_1, \ldots, x_n\}$. For $H_1$ and $X$ in Figure 4.23, $a_1(X) = n_1$, $C_1(X) = n_2 n_3$ or $n_2 n_4$, and $b_1(X) = n_6$. All the final graph $H_\lambda(X)$ will be the smallest subgraph of $H_\lambda$ containing the nodes accessing $X$ and their connecting arcs.

**Compatible:**

Let $H' = \{H'_1, \ldots, H'_m\}$ be the critical regions for $H = \{H_1, \ldots, H_m\}$ with respect to $X = \{x_1, \ldots, x_n\}$. Let $R(H'_\lambda)$ and $D(H'_\lambda)$ be the union of the ranges and the union of domains of the nodes from $H'_\lambda$ of $H_\lambda$. Let $Y'_\lambda = R(H'_\lambda) \cup D(H'_\lambda)$ and let $a_\lambda(Y'_\lambda)C_\lambda(Y'_\lambda)b_\lambda(Y'_\lambda)$ be the characteristic action sequence of $H_\lambda$ with respect to $Y'_\lambda$. We then define $H_\lambda$ and $H_S$ of $H$ to be **compatible** with respect to $X$, if for all concurrent realizations $E$ of $H_\lambda$ and $H_S$, either

1) $Y'_\lambda \sim Y'_S = \emptyset$, or

2) $C_\lambda(Y'_\lambda)$ precedes $C_S(Y'_S)$, or
3) \( C_s(Y^g) \) precedes \( C_r(Y^h) \).

Thus, in Figure 4.23, \( H_1 \) and \( H_2 \) would be compatible if in every realization \( n_3 \) or \( n_4 \) precedes \( n_5 \) or \( n_5 \) precedes \( n_2 \).

Let \( H'(X) = \{ H'_1(X), \ldots, H'_m(X) \} \) be subgraphs of \( H = \{ H_1, \ldots, H_m \} \). If for every concurrent realization of \( H'_r \) and \( H'_s \) of \( H \), the terminal nodes of \( H'_r \) precede the initial nodes of \( H'_s \) or the terminal nodes of \( H'_s \) precede the initial nodes of \( H'_r \), we say that \( H'_r \) and \( H'_s \) are disjunctly realized.

\[ D_1 = ( ), \quad R_1 = (a); \quad D_2 = (y), \quad R_2 = ( ); \quad D_3 = (z), \quad R_3 = (y); \]
\[ D_4 = (y), \quad R_4 = (b); \quad D_5 = (z), \quad R_5 = (y); \quad D_6 = (c), \quad R_6 = ( ) \]

Figure 4.23. Two processes accessing two shared structures.

Figure 4.24. The critical regions of the processes in Figure 4.23
Proposition 4.3:

Let \( H' = \{H'_1, \ldots, H'_m\} \) be the critical regions of the processes \( H = \{H_1, \ldots, H_m\} \) with respect to \( X = \{x_1, \ldots, x_n\} \). If for every realization of processes from \( H \), the critical regions are disjointly realized, then \( H \) is a set of compatible processes.

Proof:

Let \( E \) be any concurrent realization of \( H \) and \( H' \) from \( H \). Let \( n_r \) and \( n_s \) be in \( H'_r \) and \( H'_s \), respectively, and \( \overline{n}_r \) and \( \overline{n}_s \) be in \( E \). Let \( Y'_r = R(H'_r) \cup D(H'_r) \). Since \( H'_r \) and \( H'_s \) are disjointly realized, then for all such \( n_r \) and \( n_s \), \( n_r \) precedes \( \overline{n}_s \), or for all such \( n_r \) and \( n_s \), \( n_s \) precedes \( \overline{n}_r \). That is, \( C_r(Y'_r) \) precedes \( C_s(Y'_s) \) or \( C_s(Y'_s) \) precedes \( C_r(Y'_r) \). Thus, the processes are compatible.

Therefore, compatibility can be guaranteed for the processes \( H = \{H_1, \ldots, H_m\} \) if we can guarantee that the critical regions \( H' = \{H'_1, \ldots, H'_m\} \) do not interact when they share \( X = \{x_1, \ldots, x_n\} \). Consider Monitor 2 which is an extension of Monitor 1.

Monitor 2:

Let \( E = e_1 \ldots e_{2k} \) be any concurrent realization of the processes in \( H \) which share the set of path names \( X = \{x_1, \ldots, x_n\} \). Define the following sets and events. Let \( Y \subseteq X \).
B(x_i) processes with access capability to x_i
W(x_i) processes waiting for access capability to x_i
K(e_j) the x_i to which access capability is granted when e_j occurs
A'(H(Y)) the request for access capability to Y by H
A(H(Y)) the granting of access capability for Y to H
F(H(Y)) release of access capability to Y by H

It is assumed that if e_j = F(H(Y')) and there exists some e_i = A(H(Y')) where Y' \subseteq Y and i < j. We then require that the monitor have the following properties. Initially, K(e_1) = \emptyset, and B(x_i) = \emptyset for all x_i in X.

1) If e_j = A'(H(Y)) and Y \sim K(e_j) = \emptyset
   then
   e_{j+1} = A'(H(Y))
   K(e_{j+1}) = K(e_j) \cup Y
   B(x_i) = H for all x_i in Y

2) If e_j = A'(H(Y)) and Y \sim K(e_j) \neq \emptyset
   then
   K(e_{j+1}) = K(e_j)
   W(x_i) = W(x_i) \cup H for all x_i in Y

3) If e_j = F(H(Y))
   then
   B(x_i) = \emptyset for all x_i in Y
   if there exists H' in some W(x_i) such that
   Y' \sim (K(e_j) \cap Y) = \emptyset for A'(H'(Y'))
then
\[ e_{j+1} = A(H'(Y)) \]
\[ K(e_{j+2}) = K(e_{j+1}) = (K(e_j) - Y') \]
\[ B(x_i) = R' \text{ for all } x_i \text{ in } Y' \]
\[ W(x_i) = W(x_i) - R' \text{ for all } x_i \text{ in } Y' \]

else
\[ K(e_{j+1}) = (K(e_j) - Y) \]

4) If \( e_j \) any other event
then
\[ K(e_{j+1}) = K(e_j) \]

The event \( A(H(Y)) \) corresponds to the initiation of the monitor use request \((x_1, \ldots, x_k)\) by the process \( H \) where \( Y = (x_1, \ldots, x_k) \), \( A(H(Y)) \) corresponds to the termination of that use, and \( F(H(Y)) \) corresponds to the use release\((x_1, \ldots, x_k)\).

An example realization for the processes in Figure 4.25 using Monitor 2 is given by
\[
\text{[Formal expression]} \]
where \( Y_1 = (A, B, A, C) \) and \( Y_2 = (A, C, D, A, E) \). We will use the symbolic representation of the realizations when analyzing and discussing realizations in general.
Proposition 4.4:

Let $H'=[H'_1, \ldots, H'_m]$ be the critical regions of $H=[H_1, \ldots, H_m]$ with respect to $X=[x_1, \ldots, x_n]$. Let $Y' = R(H') \cap D(H')$ and let the characteristic action sequence of $H_r$ be such that $A_r(Y'_r)$ precedes $C_r(Y'_r)$ which precedes $F_r(Y'_r)$. If every characteristic action sequence for every $H_r$ of $H$ is of this form, then the processes $H=[H_1, \ldots, H_m]$ are compatible.

Proof:

Assume there exists some concurrent realization $E$ such that $H$ is not compatible with respect to $X=[x_1, \ldots, x_n]$. Then for some $H$ and $H$, $C_r(Y'_r)$ precedes $C_s(Y'_s)$ which precedes $C_r(Y'_r)$. If this is the case, then $e_i = \overline{A}(Y'_r)$ and
$e_j = \bar{A}(Y^i_s)$ precede $\bar{C}_s(Y^i_s)$. If $i < j$, then $e_{i+1} = A_r(Y^i_r)$ and $K(e_{i+2}) = K(e_{i+1}) = K(e_i) \cup Y^i_r$. Since $e_j = \bar{A}_s(Y^i_s)$ can not occur until $Y^i_s \sim K(e^r_k) = \emptyset$, for some $k > i$, it can not occur until after $K(e_{i+2}) = K(e_{i+1}) = K(e_i) \cup Y^i_r$. It can not occur until after $F_r(Y^i_r)$. Since $F_r(Y^i_r)$ precedes $\bar{C}_s(Y^i_s)$, then $C_r(Y^i_r)$ precedes $\bar{C}_s(Y^i_s)$. Thus, there exists no such sequence $E$ and $H$ is a compatible set of processes.

Consider the processes in Figure 4.26, the placement of the requests and releases to ensure compatibility. Note, however, the relocation of the access controls in Figure 4.27 will also ensure mutual exclusion. For this reason, we weaken the definition of compatibility.

![Diagram](image)

- $A_1$: request $(x,y)$
- $A_2$: request $(x,z)$
- $F_1$: release $(x,y)$
- $F_2$: release $(x,z)$

Figure 4.26. Compatibility ensured by monitor uses.
Compatibility:

Let \( H' = \{ H'_1, ..., H'_m \} \) be the critical regions for \( H = \{ H_1, ..., H_m \} \) with respect to \( X = \{ x_1, ..., x_n \} \). Let \( Y'_i = r(H'_i) \cup (D r(H'_i)) \). We define \( H_r \) and \( H_s \) to be compatible with respect to \( X \), if for every concurrent realization \( E \) of \( H_r \) and \( H_s \), either

1) \( Y'_r \sim Y'_s = \emptyset \), or
2) \( \overline{C}_r(Y'_r) \) and \( C_r(x_i) \) precede \( \overline{C}_s(Y'_s) \), for all \( x_i \sim (Y'_r \sim Y'_s) \neq \emptyset \), or
3) \( \overline{C}_s(Y'_s) \) and \( C_s(x_j) \) precede \( \overline{C}_r(Y'_r) \), for all \( x_j \sim (Y'_r \sim Y'_s) \neq \emptyset \).
Proposition 4.5:

Let $H' = \{H'_1, \ldots, H'_m\}$ be the critical regions of $H = \{H_1, \ldots, H_m\}$ with respect to $X = \{x_1, \ldots, x_n\}$. Let $Y'_i = R(H'_i) \cup D(H'_i)$ and let the characteristic action sequence for each $x_k$ in $Y'_i$ be such that $A_i(Y'_i)$ precedes $C_i(x_k)$ which precedes $F_i(x_k)$. If every characteristic action sequence for each $x_k$ in $Y'_i$ and for every $H_i$ in $H$ is of this form, then the processes $H = \{H_1, \ldots, H_m\}$ are compatible with respect to $X = \{x_1, \ldots, x_n\}$.

Proof:

Assume there exists some concurrent realization $E$ of processes from $H$ such that $H$ is not compatible with respect to $X = \{x_1, \ldots, x_n\}$. Then for some processes $H_r$ and $H_s$ of $H$ and some $x_k$ in $Y'_r \cap Y'_s$, $C_r(x_k)$ precedes $C_s(x_k)$ and $C_s(x_k)$ precedes $C_r(x_k)$. Then, both $e_i = \overline{A}_r(Y'_i)$ and $e_j = \overline{A}_s(Y'_j)$ precede $C_s(x_k)$. If $j < k$, then $A_r(Y'_r)$ precedes $C_s(Y'_j)$ and $x_k$ is in $K(e_{i+2}) = K(e_{i+1}) = K(e_i) \cup Y'_r$. In this case, $A_s(Y'_s)$ can not occur until from some $e_n$, $Y'_s \sim K(e_n) = \phi$. This can only occur after $F_r(x_k)$. Thus, since $C_r(x_k)$ precedes $F_r(x_k)$, then $C_r(x_k)$ precedes $A_s(Y'_s)$ which precedes $C_s(x_k)$. Thus, no such sequence $E$ exists.

This Proposition just states that by using Monitor 2, we can increase concurrency and guarantee the same results as the first definition of compatibility, if we return access capabilities as soon as possible.
Let the high level processes $H_1$ and $H_2$ use the low level processes $L_1$, $L_2$, and $L_3$ as shown in Figure 4.28. The critical sequences for $Y$ in the high level processes are $n_1 n_2$ for $H_1$ and $n_3$ for $H_2$. The critical sections for $Y$ in the low level processes are $n_4 n_5$ for $L_1$, $n_6$ for $L_2$, and $n_9$ for $L_3$.

We can guarantee mutual exclusion for the high level processes with respect to $Y$ by using Monitor 2 and restating $H_1$ and $H_2$ as in Figure 4.29. By using the monitor, we see that $L_3$ can not run concurrently with either $L_1$ or $L_2$, and $L_1$ always precedes $L_2$ in execution. Thus, we can ensure compatibility of $L_1$ and $L_3$ with respect to $Y$.

The uses of Monitor 2 are not restricted to any particular level. There may be situations in which the uses of the monitor are desired in the lower levels. In fact, as we will show later, it may be possible to increase concurrency by moving some of the uses into the lower levels. We can, however, restate the processes as in Figure 4.30, we still guarantee that the low level processes are compatible with respect to $Y$. Process $L_1$ always precedes $L_2$ and $n_9$ can never be executed concurrently with $n_4$, $n_5$, or $n_6$.

As a result of the placement of the monitor uses in the lower levels we have guaranteed the mutual exclusion of the higher level processes. That is, we have guaranteed mutual exclusion by ensuring that the lower level processes are compatible.
\[ \exists(x,y,z) = (w,x_1,x_2,<y_1,y_2,y_3>,z) \]

\[ D_1 = (x), \quad R_1 = (y); \quad D_2 = (y), \quad R_2 = (z); \quad D_3 = (y), \quad R_3 = (w) \]

\[ D_4 = (x_1), \quad R_4 = (y_2); \quad D_5 = (x_2), \quad R_5 = (y_2,y_3), D_6 = (y_3), \quad R_6 = (z); \]

\[ D_7 = (y_3), \quad R_7 = (\_); \quad D_8 = (\_), \quad R_8 = (w); \quad D_9 = (w), \quad R_9 = (y_2) \]

Figure 4.28. High and low level processes with domains and ranges.
Figure 4.29. Monitor uses in high level.

Figure 4.30. Monitor uses in low level.
4.4.4. Information integrity

In the last section, we looked at an example where we were able to move the uses of the monitor into the lower level and still maintain mutual exclusion. We did not, however, increase concurrency among the low level processes. Consider the processes in Figure 4.31. We can guarantee that $H_1$ and $H_2$ are compatible with respect to $X$ as shown in Figure 4.32. If the processes are executed concurrently, then a possible realization is

$$[1, 2, 1, 2, 2, 1, 2, 3, 3, 4, 4, 2].$$

Given any realization and the following refinement of the information structures

$$\mathcal{F}(x, y, z) = (x_1, x_2, y_1, y_2, z_1, z_2)$$

then with the initial state

$$\mathcal{F}(s_0 (x, y, z)) = (1, 2, 3, 4, 5, 6)$$

the only two final states that $\mathcal{F}(x, y, z)$ may have are

$$(1, 0, 5, 7)$$

and

$$(0, 0, 5, 7).$$

If, instead, we had restated the low level processes as in Figure 4.33, then with the same initial state we will have either of the same final states. We have, however, increased the potential concurrency in this case by releasing the structures in the lower level prior to the point that they
could have been released in the higher level.

Figure 4.31. High and low level processes.
Figure 4.32. Monitor uses in the high level.

Figure 4.33. Monitor uses in the low level.

**Information Integrity:**

Let $H^i=\{H^i_1, \ldots, H^i_m\}$ be the sets of critical regions for the processes $H=\{H_1, \ldots, H_m\}$ with respect to $X=\{x_1, \ldots, x_n\}$. Let $S_0(X)$ be the initial state of $X$ and let $E$ be any concurrent realization of processes from $H$, say
$H_1, \ldots, H_j$. Let the final state of $X$ be given by $F(X, E)$.

For the refinement, $\mathcal{F}(X)$ of the information structures $X$, let

$$\mathcal{F}(x_i) = \langle x_{i1}, \ldots, x_{ik_i} \rangle.$$  

Then

$$\mathcal{F}(X) = \langle \langle x_{11}, \ldots, x_{1k_1} \rangle, \langle x_{21}, \ldots, x_{2k_2} \rangle, \ldots, \langle x_{n1}, \ldots, x_{nk_n} \rangle \rangle.$$  

Let $\mathcal{F}_0(X)$ be the initial state of the refinement, and  

$\mathcal{F}(F(X, E))$ be final state of the refinement. We say that the information integrity for $X$ is preserved by $E$, if there exists some sequential ordering of the processes $H_1, \ldots, H_j$ such that if $E'$ is the execution sequence resulting from executing those processes in that order the final state

$$\mathcal{F}(F(X, E')) = \mathcal{F}(F(X, E)).$$

We will redefine the critical regions for the processes in terms of the operations of the lower level processes and the refinement on the information structure.

We now define the critical regions of $H$ with respect to the refinements of $X = \{x_1, \ldots, x_n\}$. We expand the process graphs of $H = \{H_1, \ldots, H_m\}$ in terms of the lower level process graphs by substituting the process graphs of the lower level for the nodes which use them. With this graph, we then define the critical region $H^n(x_{ij})$ of $H$ with respect to $x_{ij}$ of the refinement

$$\mathcal{F}(x_i) = \langle x_{i1}, \ldots, x_{ik_i} \rangle.$$  

We also define the critical region $H^n$ of $H$ with respect to
the refinement

\[ \mathcal{F}(x) = \langle x_{i_1}, \ldots, x_{k_1}, \ldots, x_{i_k}, \ldots, x_{k_2}, \ldots \rangle \]

of \( X = \{ x_1, \ldots, x_n \} \). Algorithm 3 will generate the critical regions \( H^r(x_{ij}) \) and Algorithm 4 will generate the critical regions \( H^r \).

Let \( H = \{ H_1, \ldots, H_m \} \) be a set of high level processes which share \( X = \{ x_1, \ldots, x_n \} \). Expand the graph \( H_r \) by replacing each node in \( H_r \) with a copy of the low level process the node uses. Let

\[ \mathcal{F}(x_i) = \langle x_{i_1}, \ldots, x_{i_k} \rangle \]

be the refinement of \( x_i \). Define the subgraphs

\[ H^r(x_{ij}) = \{ H_1^r(x_{ij}), \ldots, H_m^r(x_{ij}) \} \]

with respect to \( x_{ij} \) of \( (x_i) \) by the following algorithm.

**Algorithm 3:**

1) \( r = 1 \),
2) Let \( H_r^r(x_{ij}) \) be the expanded graph of \( H_r \),
3) If there exists any initial, or terminal, node \( n_p \) of \( H_r^r(x_{ij}) \) such that \( x_{ij} \sim (R(n_p) \cup D(n_p)) = \emptyset \), then remove the node \( n_p \) and all emanating, or entering, arcs,
4) Continue 3 until no more nodes can be removed,
5) \( r = r + 1 \),
6) Continue 2 through 5 until \( r > m \).

We will use the results of Algorithm 3 later in the chapter. If, however, Algorithm 3 were applied to the
processes in Figure 4.34, then $H''(A,B)$ and $H''(A,C)$ are given in Figure 4.35.

![Diagram of processes](image)

- $n_1$: $L_1(A,B)$
- $n_2$: $L_2(A,B)$
- $n_3$: $L_3(A,C)$
- $n_4$: ispath('x'.'y')
- $n_5$: return(true)
- $n_6$: return(false)
- $n_7$: add path('x'.'y')
- $n_8$: assign('x'.'y',1)
- $n_9$: assign('x'.'y',2)

**Figure 4.34.** Low level processes.

![Diagram of critical regions](image)

**Figure 4.35.** Critical regions $H''(A,B)$ and $H''(A,C)$.

Let $H=\{H_1, \ldots, H_m\}$ which share access to $x=\{x_1, \ldots, x_n\}$. Expand the processes of $H$ by replacing each node of every process $H_x$ with a copy of the low level process graph.
which the node uses. Let

\[ \mathcal{F}_1(x_i) = \{x_{i1}, \ldots, x_{ik_i}\} \]

and

\[ \mathcal{F}(x) = \{\{x_{11}, \ldots, x_{1k_1}\}, \{x_{21}, \ldots, x_{2k_2}\}, \ldots, \{x_{n1}, \ldots, x_{nk_n}\}\} \]

Define the subgraphs \( H^n = \{H^n_1, \ldots, H^n_m\} \) of \( H \) by the following algorithm.

**Algorithm 4:**

1) \( r = 1 \),
2) Let \( H^n_r \) be the expanded graph of \( H_r \),
3) If there exists any initial, or terminal, node \( n_p \) of \( H^n_r \), such that \( \mathcal{F}(X) \sim (R(n_p) \cup D(n_p)) = \emptyset \), then remove the node \( n_p \) and all emanating, or entering, arcs,
4) Continue 3 until no more nodes can be removed from \( H^n_r \),
5) \( r = r + 1 \)
6) Continue 2 through 6 until \( r > m \).

If we apply Algorithms 3 and 4 to the processes in Figure 4.36, we would have the process graphs as given in Figure 4.37. We let \( a_r(x_{ij}) c_r(x_{ij}) b_r(x_{ij}) \) be the characteristic action sequence for the process \( H_r \) with respect to \( x_{ij} \) of \( X \). For the process \( H_r \) in Figure 4.37, \( a_1(x_2) \) is \( n_5 \), \( c_1(x_2) \) is \( n_6 \), \( b_r(x_2) \) is \( n_7 \). We let \( a_r(\mathcal{F}(X)) c_r(\mathcal{F}(X)) b_r(\mathcal{F}(X)) \) be the characteristic critical
sequence for the process $R_x$ with respect to $\mathcal{W}(X)$. For process $R_x$ in Figure 4.37, $a_1(\mathcal{W}(x,y))$ is null, $C_1(\mathcal{W}(x,y))$ is $n_5, n_6, n_7$, and $b_1(\mathcal{W}(x,y))$ is $n_8$.

Let $D(R_x^m)$ and $R(R_x^m)$ be the union of the domains and the union of the ranges of the nodes in $R_x^m$. Let $Y^m_x = D(R_x^m) \cup R(R_x^m)$. We restate the definition of compatibility in terms of the $R_x^m$ and $Y^m_x$.

**Compatible:**

The processes $R_x$ and $R_s$ of $H = \{H_1, \ldots, H_m\}$ are compatible with respect to $(X)$ of $X = \{x_1, \ldots, x_n\}$, if for every concurrent realization of $R_x$ and $R_s$ then either

1) $Y^m_x \cap Y^m_s = \emptyset$, or

2) $C_x(Y^m_x)$ and $C_x(x_i)$ precede $C_s(Y^m_s)$, for all $x_i \sim (Y^m_x \cap Y^m_s) \neq \emptyset$, or

3) $C_s(Y^m_s)$ and $C_s(x_j)$ precede $C_x(Y^m_x)$, for all $x_j \sim (Y^m_x \cap Y^m_s) \neq \emptyset$.

Thus, in Figure 4.36, $H_1$ and $H_2$ would be compatible, if for every concurrent realization of $H_1$ and $H_2$, either $n_7$ precedes $n_9$ or $n_{10}$ precedes $n_5$. This is a more general definition for compatible. It can be applied to mutual exclusion as well as information integrity.
Before refinement:

\[ D_1 = (x) \quad R_1 = (x) \]
\[ D_2 = (x) \quad R_2 = (y) \]
\[ D_3 = (x) \quad R_3 = () \]

Uses:

\[ n_1 : L_1 \]
\[ n_2 : L_2 \]
\[ n_3 : L_3 \]

Refinement:

\[ \mathcal{H}(x, y) = (\langle x_1, x_2 \rangle, \langle y_1, y_2, y_3 \rangle) \]

After refinement:

\[ D_1 = (x_1, x_2) \quad R_1 = (x_1) \quad D_6 = (x_2) \quad R_6 = (y_1, y_2) \]
\[ D_2 = (x_2) \quad R_2 = (y_1, y_2) \quad D_7 = (a) \quad R_7 = () \]
\[ D_3 = (x_1, x_2) \quad R_3 = () \quad D_8 = (x_2) \quad R_8 = (x_2) \]
\[ D_4 = (x_1) \quad R_4 = (x_1) \quad D_9 = (x_1) \quad R_9 = () \]
\[ D_5 = (x_2) \quad R_5 = () \]

Figure 4.36. High and low level processes with structures and refinements.
Proposition 4.6:

Let $H^* = \{H^*_1, \ldots, H^*_m\}$ be the critical regions of the compatible functional processes $H = \{H_1, \ldots, H_m\}$ with respect to $\mathcal{F}(X)$ of $X = \{x_1, \ldots, x_n\}$. Then for every concurrent

![Diagram]

Figure 4.37. Critical regions for Figure 4.36.
realization of processes from $H$ and any initial state $s_0(X)$, the information integrity of $X$ is preserved.

This proposition gives the sufficient conditions to preserve information integrity in the concurrent realization of the processes and their decompositions. It states that we only need to ensure that the processes are compatible in order to guarantee information integrity. From the following proposition, the locations of the monitor uses can be identified. These locations will guarantee information integrity and may increase potential concurrency.

By using monitor 2, we can achieve these conditions.

**Proposition 4.7:**

Let $H^n(x_{ij}) = \{H^n(x_{ij}), \ldots, H^n(x_{ij})\}$ and $H^n = \{H^n, \ldots, H^n\}$ be the critical regions (defined by Algorithms 3 and 4) of the functional processes $H = \{H, \ldots, H\}$ with respect to $x_{ij}$ of $\mathcal{H}(x_i)$ and with respect to $\mathcal{H}(X)$. Let $Y^n = R(H^n) \cup D(H^n)$ and let the characteristic action sequence of $x_{ij}$ in $Y^n$ be such that $A^R(Y^n)$ precedes $C^R(x_{ij})$ which precedes $F^R(x_{ij})$. If every action sequence for each $x_{ij}$ in $Y^n$ and every $H_r$ in $H$ is of this form then the information integrity of $X$ is preserved.

**Proof:**

For every concurrent realization $R$ and every initial state $s_0(X)$ and for every $H_r$ and $H_s$ in $H$ such that $Y^n \cap Y^n \neq \emptyset$, we have $A^R(Y^n)$ precedes $A^S(Y^n)$ or $A^S(Y^n)$ precedes $A^R(Y^n)$. In
the first case, for all $x_{ij}$ in $\mathcal{Y}_T \cup \mathcal{Y}_S$, $C_r(x_{ij})$ precedes $C_s(x_{ij})$. That is $H_r$ and $H_s$ are compatible. In the second case, the same argument applies and they are still compatible. Thus, the information integrity of $X$ is preserved.

Thus, we can guarantee information integrity if every critical region for every process is preceded by a request for all access capabilities needed in the critical region. Concurrency is increased by releasing the individual capabilities as early as possible. An example of the use of Monitor 1 in this manner is given in Figure 4.38.

When the high level processes access a structure, they do so by local reference. When a node in a high level process uses, or calls, a low level process it passes arguments. These arguments may be structures or values. The low level processes then reference the structures by the formal parameter name. For example, in Figure 4.39, the action $n_1$ in the high level process passes the structure $A$ and the value $B$ to $L$. The low level process references this structure as '$x$' and the substructure $A.B$ of $A$ by '$x'.y'$. For the present, we will only consider the mapping of $\mathcal{F}(A)$, but we will return to the reference problem later in chapter 5.
Figure 4.38. Information integrity and Monitor 2 uses.
We have represented the refinement of a $x_i$ in $X$ by $\mathcal{N}(x_i) = \langle x_{i1}, \ldots, x_{ik_i} \rangle$. Consider, however, the structure in Figure 4.40. The nodes $n_1$ and $n_2$ are referenced by $A.B$ and $A.C$, respectively. Consider the high level and low level processes in Figure 4.40. We can guarantee mutual exclusion for $A$ by rewriting the high level processes as in Figure 4.41 using Monitor 2. This does not allow any concurrency when assessing the structure $A$. The same applies if we had rewritten the low level processes as in Figure 4.42. We could have, however, increased the concurrency by some degree if we had rewritten the level processes as in Figure 4.43.
Figure 4.40. High and low level processes with structure references and information structure.
$A_1$: request($A$)  \hspace{1cm} F_2$: release($A$)

$A_2$: request($A$)  \hspace{1cm} F_3$: release($A$)

$F_1$: release($A$)

Figure 4.41. Monitor uses in the high level.

Figure 4.42. Monitor 2 uses in the low level.
If, however, we had a monitor which allows a process to request access capability to a substructure of a structure to which the process has the access capability, then it is possible to increase concurrency as shown in Figure 4.44. The time for these remaining requests to be granted should not increase the time in granting since the process already owns the access capabilities.
Consider the monitor as specified below which allows a process to request access to substructures.

Figure 4.44. Monitor uses for substructures.
Monitor 3:

Let $E = e_1 e_2 \ldots e_n$ and define the following sets and events. Let $Y \subseteq X \cup \mathcal{M}(X)$ where $X = \{x_1, \ldots, x_n\}$ and

$\mathcal{M}(X) = \langle x_{1_1}, \ldots, x_{1_k_1}, x_{2_1}, \ldots, x_{2_k_2}, \ldots, x_{n_1}, \ldots, x_{n_k_n} \rangle$.

- $B(z)$: processes with access capability to $z$,
- $W(z)$: processes waiting for access capability to $z$,
- $K(e_j)$: the $x_i$ and $x_{ik}$ to which access capabilities have been granted when $e_j$ occurs,
- $\overline{A}(H(Y))$: the request for access capabilities to $Y$ by $H$,
- $A(H(Y))$: the granting of access capabilities to $Y$ to $H$,
- $F(H(Y))$: the return of access capabilities to $Y$ by $H$.

It is assumed that if $e_j=P(H(Y'))$, then there exists some $x_i$ such that $e_i=\overline{A}(H(Y))$ where $Y \subseteq Y'$ and $i < j$. Let $x_{ij}$ be in $\mathcal{M}(X)$. It is required that the monitor have the following properties. Initially, $K(e_1) = \emptyset$ and

$B(x_i) = W(x_i) = B(x_{ij}) = W(x_{ij}) = \emptyset$ for all $x_i$ in $X$ and $x_{ij}$ in $\mathcal{M}(X)$.

1) If $e_k = \overline{A}(H(Y))$ and

for all $x_i$ in $Y$, $x_i \sim K(e_k) = \emptyset$ and

for all $x_{ij}$ in $Y$ such that $x_{ij} \sim K(e_k) \neq \emptyset$, we have $B(x_i) = H$ then

$e_{k+1} = A(H(Y))$

$B(x_i), B(x_{ij}) = H$ for all $x_i$ and $x_{ij}$ in $Y$

$K(e_{k+2}) = K(e_{k+1}) = K(e_k) \cup Y$
2) If $e_k = A(H(Y))$ and
   
   for some $x_i$ in $Y$, $x_i \sim K(e_k) \neq \emptyset$ or
   
   for some $x_{ij}$ in $Y$ such that $x_{ij} \sim K(e_k) \neq \emptyset$,
   
   $B(x_i) \neq H$

   then

   $K(e_{k+1}) = K(e_k)$
   
   $W(x_i) = W(x_i) \cup H$ for all $x_i$ in $Y$
   
   $W(x_{ij}) = W(x_{ij}) \cup H$ for all $x_{ij}$ in $Y$

3) If $e_k = F(H(Y))$

   then

   if there exists some $H'$ in some $W(x_i)$ or $W(x_{ij})$

   where $\tilde{A}(H'(Y'))$ and

   for all $x_i$ in $Y'$, $x_i \sim (K(e_k) - Y) = \emptyset$, and

   for all $x_{ij}$ in $Y'$ such that

   $x_{ij} \sim (K(e_k) - Y) \neq \emptyset$, $B(x_i) = H'$

   then

   $e_{k+1} = A(H'(Y'))$
   
   $B(x_i), B(x_{ij}) = H'$ for all $x_i$ and $x_{ij}$ in $Y'$
   
   $W(x_i) = W(x_i) - H'$ for all $x_i$ in $Y'$
   
   $W(x_{ij}) = W(x_{ij}) - H'$ for all $x_{ij}$ in $Y'$
   
   $K(e_{k+2}) = K(e_{k+1}) = (K(e_k) - Y) \cup Y'$

   else

   $B(x_i), B(x_{ij}) = \emptyset$ for all $x_i$ and $x_{ij}$ in $Y$

   $K(e_{k+1}) = (K(e_k) - Y)$
Monitor 3 was demonstrated in Figure 4.44. We will now consider the use of that monitor. We have three sets of critical regions to consider for the processes \( H = \{ H_1, \ldots, H_m \} \) with respect to \( X = \{ x_1, \ldots, x_n \} \) and the refinement \( \mathcal{S}(X) \).

1) \( H' = \{ H'_1, \ldots, H'_m \} \), the critical regions of \( H \) with respect to \( X \),

2) \( H''(x_{ij}) = \{ H''_1(x_{ij}), \ldots, H''_m(x_{ij}) \} \), the critical regions of \( H \) with respect to \( x_{ij} \) of \( \mathcal{S}(X) \), and

3) \( H''' = \{ H'''_1, \ldots, H'''_m \} \), the critical regions of \( H \) with respect to \( \mathcal{S}(X) \) of \( X \).

We also have two sets of structures to consider.

1) \( Y' = D(H'_1) \cup R(H'_1) \) for \( H'_1 \) in \( H' \), and

2) \( Y'' = D(H''_1) \cup R(H''_1) \) for \( H''_1 \) in \( H'' \).

**Proposition 4.8:**

Let the characteristic action sequence for \( x_{ij} \) in \( Y''_1 \) have the following three properties

1) \( A_r(Y''_1) \) precedes \( A_r(x_{ij}) \),

2) \( A_r(x_{ij}) \) precedes \( C_r(x_{ij}) \) and \( F_r(Y''_1) \), and

3) \( C_r(x_{ij}) \) precedes \( F_r(x_{ij}) \).

If for every \( H'_1 \) and for all \( x_{ij} \) in \( Y''_1 \), the characteristic action sequences have these properties, then the information integrity of \( X \) is preserved.
Proof:

For every concurrent realization \( E \), every initial state \( s_0(X) \) and every \( r \) and \( s \) in \( H \) such that \( Y^r \neq Y^s \neq \emptyset \), we have \( A_r(Y^r) \) precedes \( A_s(Y^s) \) or \( A_s(Y^s) \) precedes \( A_r(Y^r) \). In the first case, for all \( x_{ij} \) in \( Y^r \neq Y^s \), \( C_r(Y^r) \) and \( C_r(x_{ij}) \) precede \( C_s(Y^s) \). In the second case, for all \( x_{ij} \) in \( Y^r \neq Y^s \), \( C_s(Y^s) \) and \( C_s(x_{ij}) \) precede \( C_r(Y^r) \). That is, they are compatible, and, therefore, by Proposition 4.6, the information integrity of \( X \) is preserved.

This proposition states that if we can build a monitor with the specifications of Monitor 3, and if we use that monitor by requesting all major structures and later all substructures of the major structures, then we can return access to the major structures and still ensure information integrity. Such a use of a monitor will increase the potential concurrency by allowing for concurrent access to subcomponents of major structures. Monitor 2 did not allow for such interaction which motivated the development of Monitor 3.

4.5. Deadlock:

In section 4.4.3., an example of deadlock was presented. Two processes were waiting for the other to return the access capabilities to a path name.
In our particular case with Monitor 3, the only way this can happen is for two or more characteristic action sequences to contain two or more requests with respect to the critical regions \( H' = \{ H'_1, \ldots, H'_m \} \) of \( H = \{ H_1, \ldots, H_m \} \). This holds because a process which has been granted access to any structures can request any substructures and not deadlock. Thus, in the use of the monitor, if we require that access be requested to all major structures together, and in turn, the monitor is built to support the specifications of Monitor 3, then we will have no deadlock. In this work, the placement of the monitor uses within the procedures is the responsibility of the algorithm to be presented in chapter 5. Detailed specifications for implementation of the three monitors are presented in chapter 7.

4.6. Comments

In this chapter, we have introduced three monitors and seen how the use of Monitor 3 can increase concurrency while ensuring mutual exclusion and information integrity where necessary. We have also seen that there are some assumptions on the use of this monitor in the processes of the control structure. They are

1) the monitor will be usable from any level

2) the monitor will be used to request access
capabilities to the structure prior to the use of the structures.

There is an additional assumption that may be necessary in the execution environment. When structures and arguments are passed from level to level, there may be information loss. For example, in some situations we may be required to request access, say in level $i$, but be allowed to release in level $i+1$. The release in the lower level may increase concurrency. When the arguments are passed from level $i$ to level $i+1$, it may be possible to lose the identity of the structure for which the original request was made. We will assume that the releases can be bound to the original structure for which the request was made. We will discuss a method of handling this situation in chapter 7.

We now have the following requirements for the monitor.

1) It must be able to handle requests and releases for lists of structures.

2) It must not grant any access capabilities to a process unless capabilities to all the structures in the request list can be granted.

3) It must guarantee that in its operation no processes deadlock as a result of processes competing for access capabilities.

4) It must be able to interface in the execution time environment in such a way as to be able to
bind all subsequent requests and releases to the first structures requested.

The algorithm presented in chapter 5 assumes these properties of the monitor.
5. DESIGN ASSISTANCE ALGORITHM

5.1. Introduction

In this chapter, we present an algorithm which if used by the designer of an information system has the properties discussed in chapter 3. It will

1) increase concurrency as the system is designed,
2) notify the designer of nondeterminate processes,
3) guarantee information integrity of the information structures, and
4) guarantee a deadlock free system.

Three different monitors were presented in chapter 4. The first monitor was primarily specified as an access control mechanism to individual components of the information structures. This monitor required a separate request and release for each component to be accessed. It was demonstrated that the use of this monitor could, in some cases, lead to deadlock. The second monitor could be used for requests and releases of access capability to multiple structures and would, if used according to the form presented in Proposition 4.5, increase the degree of concurrency. The third monitor could be used as the second monitor, but it could also be used to request and release subcomponents of structures to which a process already had access capability.
It was demonstrated in Proposition 4.7 that this monitor could preserve the information integrity of the structures and allow for an increase in concurrency over the second monitor.

The algorithm presented in this chapter will assume the existence of the third monitor and will base its decisions on that assumption. The algorithm can be modified to be used in an environment where the first or second monitor is sufficient. The algorithm will also assume that the procedures for the level processes are written in a Pascal-like language presented in the next section. It is not, however, restricted to this language, but may be modified to apply to other languages. The algorithm will also require information about the domains and ranges of the language constructs and the uses of lower level processes. This information can be generated by a precompilation of the procedures and explicit input by the designer in the case of procedure or function statements. This information will be used to place and remove request and release uses of the monitor. The algorithm will also need access to a parser for the procedural language so that it may make decisions based on the structure of procedures.

In general, the algorithm analyzes the procedures of a high level and the immediate lower level. It assumes that the monitor uses located in the high level processes will
guarantee correctness with respect to information integrity and deadlock. (In the outer level, this requires an initial effort by the designer of declaring the shared information structure, or structures.) The algorithm places request for subcomponents and releases for major components and subcomponents in the lower level processes based on the location of the requests and release in the higher level and on the structure of the lower level processes.

This portion of the algorithm requires the parser and the information about the domain and range of the language constructs. As the algorithm is applied, it also analyzes the procedures for the sufficient conditions for determinacy. If a procedure fails to satisfy these conditions, the designer is notified. The algorithm may be halted at this point, or continued, depending on the desirability of determinacy. As stated above, the final state of the procedures is assumed to be supported by a monitor having the properties described for monitor 3 in section 4.4.4.

5.2. The Procedural Language

In this section, we present the procedural language over which we define our algorithm. This language is Pascal-like, and can be useful in the top-down design of a software system. The syntax for this language is given in Figure 5.1
in Backus-Naur form and an example procedure is given in Figure 5.2. The data and type declarations are omitted in this language since they are not pertinent to the algorithm.

Figure 5.1. The BNF of the grammar of the procedural language.
procedure command interpreter(command, A1, A2, A3, A4)


type information structure = record
    boards: set of board;
    basic components: set of basic;
    tests: set of test;
end


type board = ?
type basic = ?
type test = ?
var database: information structure

begin
  if command = 'test'
  then
    begin
      test (database, A1, A2, A3, A4)
    end
  else
    begin
      if command = 'build'
      then
        begin
          build (database, A1, A2, A3, A4)
        end
      else
        begin
          write ('error')
        end
    end
end

Figure 5.2. Example procedure.

5.3. Automatic Procedure Analysis and Control Placement

When analyzing a procedure for determinacy, or attempting to make information integrity decisions, it is necessary to know the domain and range of each operation in the procedure. It will be assumed that this information will
be available.

For each construct $s$ of a procedure, let $R(s)$ and $D(s)$ be the range and the domain of the construct, and let $Y=(R(s), D(s))$. Then, we can represent the ranges and the domains of the parts of the procedure by forming the unions of the domains and ranges of the subparts. For example, in Figure 5.3 the ranges and the domains of the parts are given.

The analysis for noninterference and compatibility will compare these sets. In particular, we define $Y\sim Y' = \emptyset$ if $R \sim R' = D \sim D' = \emptyset$. For example, the procedure in Figure 5.3 has two parallel statements, and since $Y_7 \sim Y_8 = \emptyset$ and $Y_{11} \sim Y_{12} = \emptyset$, this procedure satisfies the sufficient conditions for determinacy.

The analysis for determinacy is part of the algorithm presented in this section. It applies a set of boolean functions and the parser to compare the ranges and the domains of each parallel statement. Figure 5.4 demonstrates one of these functions. The function is $A_3$ which is applied to the language construct '〈statement list〉'. The range and the domain of this construct is $Y$. If the construct will parse as a '〈statement〉', then $A_3$ has the boolean value of the function $A_4$. Otherwise, it is the logical product of $A_4$ applied to a '〈statement〉' with range and domain of $Y'$, and $A_3$ applied to the remaining '〈statement list〉' with the range and domain of $Y'$. 
procedure swap1(w,x,y,z)

begin
  cobegin
    begin
      copy2(v,t1,w,x)------Y1    Y7  Y9
      erase2(w,x)---------Y2
    end
    begin
      copy(v,t2,y,z)-------Y3    Y8  Y10
      erase2(y,z)---------Y4
    end
  coend
begin
  copy2(y,z,v,+1) Y5 Y11
  copy2(*,x,v,t2)------Y6    Y12
end
coend
end

Y1= ((t1),(x))       Y9 = ((t1,x),(x))
Y2= ((x),())         Y10= ((t2,z),(z))
Y3= ((t2),(z))       Y11= ((z),(t1))
Y4= ((z),())         Y12= ((x),(t2))
Y5= ((z),(+1))       Y13= ((t1,t2,x,z),(x,z))
Y6= ((x),(t2))       Y14= ((x,z),(t1,t2))
Y7= ((t1,x),(x))     Y15= ((t1,t2,x,z),(t1,t2,x,z))
Y8= ((t2,z),(z))

Figure 5.3. Procedure with range-domain pairs.
In general, the functions compare the ranges and domains returning the value *true*, if the procedure parts are mutually noninterfering, or the value *false*, if the procedure does not satisfy the noninterference condition. The complete set of functions is given in Figure 5.5. The sets $Y$, $Y'$, $Y''$, and $Y'''$ represent the operands of the associated language constructs. These sets are used to determine whether the procedures satisfy the extended noninterference condition.

In the remaining analytic and transformation procedures presented in this chapter, we will use these operand sets extensively.

A step-by-step application to 'swap1' of Figure 5.3 is given in Figure 5.6.

\[
A3(<\text{statement list}>Y) =
\begin{align*}
A4(<\text{statement}>Y) & \quad \text{if } G8 \\
A4(<\text{statement}>Y') & \land A3(<\text{statement list}>Y'') & \text{if } G9
\end{align*}
\]

Figure 5.4. The determinacy analysis function $A3$. 
A1(<procedure and function>Y)
  = A1(<procedure or function heading><block>Y)
  = A2(<block>Y)

A2(<block>Y)
  = A2(begin <statement list>Y end)

A3(<statement list>Y)
  = \begin{cases} 
  A4(<statement>Y) & \text{if } G8 \\
  A4(<statement>Y') \& A3(<statement list>Y'') & \text{if } G9 
\end{cases}

A4(<statement>Y)
  = \begin{cases} 
  \text{true} & \text{if } G10, G11, \text{ or } G12 \\
  A5(<structured statement>Y) & \text{if } G13 
\end{cases}

A5(<structured statement>Y)
  = \begin{cases} 
  A2(<block>Y) & \text{if } G16 \\
  A6(<conditional statement>Y) & \text{if } G17 \\
  A7(<repetitive statement>Y) & \text{if } G19 \\
  A9(<parallel statement list>Y) & \text{if } G18 
\end{cases}

A6(<conditional statement>Y)
  = \begin{cases} 
  A6(if <expression>Y' then <block>Y else <block>Y'') & \text{if } G16 \\
  A2(<block>Y') \& A2(<block>Y'') & \text{if } G17 
\end{cases}

A7(<repetitive statement>Y)
  = \begin{cases} 
  A7(while <expression>Y' do <block>Y'') & \text{if } G18 \\
  A2(<block>Y'') & \text{if } G19 
\end{cases}

A8(<parallel block>Y)
  = A8(cobegin <parallel statement list>Ycoend)
  = A9(<parallel statement list>Y)

A9(<parallel statement list>Y)
  = \begin{cases} 
  A2(<block>Y) & \text{if } G23 \\
  A10(<block>Y' <parallel statement list>Y'') & \text{if } G24 
\end{cases}

A10(<block>Y' <parallel statement list>Y'')
  = \begin{cases} 
  A2(<block>Y'') \& A9(<parallel statement list>Y'') & \text{if } Y' \sim Y'' \neq \emptyset \\
  \text{false} & \text{if } Y' \sim Y'' \neq \emptyset 
\end{cases}

Figure 5.5. Determinacy analysis functions A1 through A10.
A1('swap1'Y15)
  = A2(<block>Y15)
  = A2(beginstatement list>Y15end)
  = A3(<statement list>Y15)
  = A4(<statement list>Y13) & A3(<statement list>Y14)
  = A5(<structured statement>Y13) & A4(<statement list>Y14)
  = A8(<parallel block>Y13) & A5(<structured statement>Y14)
  = A9(<parallel statement list>Y13) & A8(<parallel block>Y14)
  = A10(<block>Y9<parallel statement list>Y10) & A9(<parallel statement list>Y14)
  = A2(<block>Y9) & A9(<parallel statement list>Y10) & A10(<block>Y11<parallel statement list>Y12)
  = A3(<statement list>Y9) & A2(<block>Y10) & A2(<block>Y11) & A9(<parallel statement list>Y12)
  = A4(<statement list>Y1) & A3(<statement list>Y2) & A3(<statement list>Y8) & A3(<statement list>Y5) & A2(<block>Y12)
  = true & A4(<statement list>Y2) & A4(<statement list>Y3) & A3(<statement list>Y4) & A4(<statement list>Y5) & A3(<statement list>Y6)
  = true & A4(<statement list>Y4) & A4(<statement list>Y6)
  = true

Figure 5.6. Determinacy analysis of 'swap1'.

Two other tasks the algorithm will perform are the placement of requests and releases for access capabilities. The algorithm uses two sets of transformations for this purpose. Their objective can be illustrated graphically as in Figure 5.7, which gives a process graph $H$ and the critical regions $H'(z)$ and $H'(y)$. In order to ensure compatibility, the lack of deadlock, and an increase in concurrency the requests and releases for $z$ and $y$ are arranged as in Figure 5.8.

The transformations which perform this location are given in Figures 5.9a through 5.10c. As in the functions for noninterference analysis, these transformations use a parser and the ranges and the domains of the constructs. They also take arguments in the form of a set of structures $X = \{X_1, \ldots, X_n\}$ for the first transformations and an individual structure $X = \{X_1\}$ for the second set. A step-by-step application to the process in Figure 5.11 is given in Figures 5.12 through 5.17. These transformations assume determinate procedures.
Figure 5.7. Process graph and critical regions.

Figure 5.8. Process graph with requests and releases.
BO(-anything-Y,X) = -anything-

B1(<procedure and function>Y,X) = BO(<procedure heading>)B2(<block>Y,X);if G1

B2(<block>Y,X)

\[
B0(<block>Y)
\]

if X\sim Y \neq \emptyset and G7

B0(begin)B3(<statement list>Y,X)B0(end)

if X\sim Y \neq \emptyset and G7

B3(<statement list>Y,X)

\[
B0(<statement list>Y,X)
\]

if X\sim Y \neq \emptyset

B4(<statement>Y,X)

\[
B0(<statement>Y,X)
\]

if X\sim Y \neq \emptyset and G8

B4(<statement>Y',X)B0(<statement list>Y",X)

if X\sim Y' \neq \emptyset and G9

B0(<statement>Y',X)B3(<statement list>Y",X)

if X\sim Y' \neq \emptyset and G9

B4(<statement>Y,X)

request(X)B0(<assignment statement>Y,X);if G10

\[
request(X)
\]

B0(<procedure or function statement>Y,X)

if G11 or G12

B5(<structured statement>Y,X);if G13

B5(<structured statement>Y,X)

B2(<block>Y,X);if G16

B6(<conditional statement>Y,X);if G17

\[
request(X)B0(<repetitive statement>Y,X);if G19
\]

B9(<parallel block>Y,X);if G18

B6(<conditional statement>Y,X)

\[
B7(if<expression>Y'
\]

then<block>Y"else<block>Y",X) by G20

Figure 5.9a. Transformations for locating requests.
Fig. 5.9b. Transformations for locating requests.
Figure 5.10a. Transformations for locating releases.
Figure 5.10b. Transformations for locating releases.
procedure union1(A,A1,B,B1)
  begin
do begin
  begin if has2(A,A1)---------Y1 then begin---------------Y2
  end else begin
    insert2(A,A1)---------Y3
  end end
  begin if has2(B,B1)---------Y4 then begin
    copy2 (t,B1,B,B1)-----Y5
  end else begin
    assign2 (t,B1,nil)--Y6
  end end
do end
  append2 (A,A1,t,B1)---------Y7
  end

Y1=((), (A)) Y7=((t, A), (t))
Y2=(((), ()))     Y8=Y1vY2vY3
Y3=((A), ())     Y9=Y4vY5vY6
Y4=(((), (3)))   Y10=Y8vY9
Y5=((t), (B))    Y11=Y7vY10
Y6=((t), ())

Figure 5.11. The procedure 'union1'.
B1('union1' X={A,B})
   = B0(procedure union1 (A,A1,B,B1))B2(block>Y11,X)
      where
      Y11=((A,t),A,B,t))

B2(block>Y11)
   = B0(begin)B3(statement>Y11,X)B0(end)

B3(statement list>Y11,X)
   = B4(statement+Y10,X)B0(statement list>Y7,X)
      where
      Y10=((A,t),(A,B))
      Y7=(((A),(t))

B4(statement>Y10,X)
   = B5(structured statement>Y10,X)

B5(structured statement>Y10,X)
   = B9(parallel block>Y10,X)

B9(parallel block>Y10,X)
   = if B10(parallel statement list>Y10,X)
      then B13(parallel block>Y10,X)
      else request(X)B0(parallel block>Y10,X)

B10(parallel statement list>Y10,X)
   = B12(block>Y8<parallel statement list>Y9,X)
      where
      Y8=(((A),(A))
      Y9=(((t),(B))

B12(block>Y8<parallel statement list>Y9,X)
   = false
      since
      XnY8={A} and XnY9={B}

B9(parallel block>Y10,X)
   = request(A,B)B0(parallel block>Y10,X)

Figure 5.12. Locating the requests for A and B in 'union1' of Figure 5.11.
procedure union(A,A1,B,B1)
begin
request(A,B)------------------------X12= ((),(A,B))
cobegin
begin
if has2(A,A1) then
begin
end
else
begin
insert2(A,A1)
end
end
begin
if has2(B,B1) then
begin
copy2(t,B1,B,B1)
end
else
begin
assign2(t,B1,nil)
end
end
coend
append2(A,A1,t,B1)
end

Figure 5.13. The procedure 'union1' after B1(union1, [A,B]).
Figure 5.14. Locating the releases in 'union1' of Figure 5.13.
procedure union1(A, A1, B, B1)
begin
  request(A, B)
cobegin
    begin
      if has2(A, A1) then
        begin
          end
        else
          begin
            insert2(A, A1)
            end
        end
    begin
      if has2(B, B1) then
        begin
          copy2(t, B1, B, B1)
          end
        else
          begin
            assign2(t, B1, nil)
          end
    end
coend
append2(A, A1, t, B1)
release(A)-----------------------------Y13=()?, (A))
end

Figure 5.15. The procedure 'union1' after C1('union1'[A]).
Figure 5.16a. Locating releases for B in 'union1' of Figure 5.15.
Figure 5.16b. Locating releases for B in 'union1' of Figure 5.15.
When a new level is introduced, and after it has been analyzed for determinacy, the ranges and domains of the new level will be interpreted in terms of the refinements (if any were made) of a structure $X$ by $F(X) = (X_1: X_2, \ldots, X_n)$. If some of the $X_i$ are structures (arrays, sets, lists, etc.), then the structure is given as $F(X) = (X: \text{set})$, for example. Given the high level in Figure 5.17, let Figures 5.18 through 5.22 represent a new set of low level procedures with structure refinement.

We assume that the low level processes have been analyzed for determinacy and found to satisfy the definition of noninterference. The ranges and domains for the high level procedures can now be given in terms of the domains and ranges of the low level procedures. Since we are only concerned with the shared structures, we will limit the ranges and domains to include only shared items. The refined domains and ranges for each procedure are given in Figure 5.23 along with the restated domains for the high level procedures, (see also Figure 5.11).

The operand sets in the associated procedures, both the analytic and transformational, will only contain the variable names whose values will be path names to shared structures or substructures.
procedure union1(A, A1, B, B1)
begin
    request(A, B)
    cobegin
        begin
            if has2(A, A1) then begin
                end
            else begin
                insert2(A, A1)
                end
            end
        begin
            if has2(B, B1) then begin
                copy2(t, B1, B, B1)
                release(B)
                end
            else begin
                release(B)
                assign2(t, B1, nil)
                end
            end
    coend
    append2(A, A1, t, B1)
    release(A)
end

Figure 5.17. The procedure 'union1' after C1('union1' {B}).
function has2(A,B): returns boolean:

begin
  if B.name in A then
    begin
      if B.part=nil then
        begin
          return (true)
        end
      else
        begin
          if has3(A,B.name, B.part) then
            begin
              return (true)
            end
          else
            begin
              return (false)
            end
        end
    end
  else
    begin
      return (false)
    end
end

Y1=(((),(A,B.name)))
Y2=(((),(A,B.name, B.part)))

F(E)=(B;name,part)
F(A)=(A;set)

Figure 5.18. The low level procedure 'has2'.
procedure insert2(A, B)
begin
if B.name in A then
    insert3(A, B.name, B.part)
else
    begin
        A:=A+B.name
        if B.part=nil then
            begin
            end
        else
            begin
                insert3(A, B.name, B.part)
            end
    end
end

Y1=(),(A,B.name)  \quad F(B)=(B:name,part)
Y2=((A,B.name),(B.part))  \quad F(A)=(A:set)
Y3=((A),(A,B.name))
Y4=((),(B.part))
Y5=((A,B.name),(B.part))

Figure 5.19. The low level procedure 'insert2'.

**procedure** `copy2(A, A1, B, B1)`

```
begin
  if A1.part=nil then
    begin
      erase3(A.A1.name)            \ Y2
      copyval3(A.A1.name, B.B1.name) \ Y3
    end
  else
    begin
      copy3(A.A1.name, A1.part,    \ Y4
            B.B1.name, B1.part)
    end

Y1 = (((), (A1.part))           F(A) = (A: set)
Y2 = (((A.A1.name), ()))        F(B) = (B: set)
Y3 = (((A.A1.name), (B.B1.name))) F(A1) = (A1: name, part)
Y4 = (((A.A1.name), (A1.part,    F(B1) = (B1: name, part)
            B.B1.name, B1.part)))
```

Figure 5.20. The low level procedure 'copy2'.

procedure assign2(A,B,C)
begin
if B.name in A then begin
  erase3(A,B.name) // Y2
  copyval3(A,B.name,C.val) // Y3
end
else begin
  A := A + B.name // Y4
  copyval3(A,B.name,C.val) // Y5
end
end

Y1 = ((), (A,B.name)) F(A) = (A:set)
Y2 = ((A,B.name), ()) F(B) = (B:name,part)
Y3 = ((A,B.name), (C.val)) F(C) = (C:val)
Y4 = ((A), (A,B.name))
Y5 = ((A,B.name), (C.val))

Figure 5.21. The low level procedure 'assign2'.

procedure append2(A, A1, B, B1)

begin
  if A1.part=nil then
    begin
      copyval3(A.A1.name, B.B1.name) begin
    end
  else
    begin
    end
  end

Y1=((() (A1.part))
Y2=((A.A1.name), (B.B1.name))
Y3=((A.A1.name), (A1.part, B.B1.name, B1.part))

F(A) = (A:set)
F(B) = (B:set)
F(A1) = (A1:name, part)
F(B1) = (B1:name, part)

Figure 5.22. The low level procedure 'append2'.

Y('has2') = (((), (A,A.B.name))
Y('insert2') = ((A:A.B.name), (A,A.B.name))
Y('copy2') = ((A.A1.name), (B.B1.name)))
Y('assign2') = ((A,A.B.name), (A.B.name))
Y('append2') = ((A.A1.name), (B.B.name))

For 'union1',

Y1 = (((), (A.A.A1.name))
Y2 = (((), ()))
Y3 = ((A,A.A1.name), (A,A.A1.name))
Y4 = (((), (B,B.B1.name))
Y5 = (((), (B.B1.name))
Y6 = (((), ()))
Y7 = ((A.A1.name), ()))

Figure 5.23. The restated ranges and domains of procedures.
The objective of the algorithm, after the restatement of ranges and domains, can be illustrated graphically, as in Figures 5.24 through 5.27. Let the enclosed nodes in Figure 5.24 represent the critical regions $H'(A)$ and $H'(B)$ of $H$ with respect to $A$ and $B$ prior to the refinement of $A$. As a result of the application of the $B1$ and $C1$ transforms, the requests and releases for $A$ and $B$ will be placed in the procedure as shown in Figure 5.25. This placement will ensure compatibility and the lack of deadlock. This, effectively, changes the critical region of $H$ with respect to $A$ as shown in Figure 5.25. Let the restated critical regions as a result of the refinement of $A$ and $B$ and the low level procedures be as given in Figure 5.26. The old effective critical regions are given in dashed lines. We can increase the degree of concurrency by releasing $A$ and $B$ as soon as possible and only retaining control of those subcomponents needed outside the critical regions of $A$ and $B$. We can realize some of the potential increase in concurrency while ensuring information integrity, and the lack of deadlock by placing requests and releases as shown in Figure 5.27. This results in the effective critical regions shown. If we remove the releases to the original structures with the refined domains and ranges, the new critical regions for those structures will be protected. Figure 5.28 gives the procedure in Figure 5.27 after removing all releases and
reapplying C1('union1' A) and C1('union1' B) using the refined ranges and domains as given in Figure 5.23.

Figure 5.24. The critical regions of R prior to refinement.
$A_1$: request \((A,B)\)

$F_1,F_3,F_5,F_7$: release \((B)\)

$F_2,F_4,F_6,F_8,F_9$: release \((A)\)

*Figure 5.25.* The process $H$ restated with requests and releases.
Figure 5.26. The restated critical regions of $H$. 
Figure 5.27. The restatement of \( H \) with requests and releases.
We do not need to control access to substructures, if their critical regions are wholly contained within the critical region for the group structure. Then we will only place requests and releases for substructures which lie partially, or wholly, outside of the major structure's critical region. The functions that determine which are these substructures are given in Figures 5.29a through 5.29b. For the procedure in Figure 5.28, these substructures are A.AI.name and B.Bl.name. In the transformation we use a notational convenience to indicate the relationship of a path name and a structure, we let $z(X,Y) = \{ x | x \text{ in } Y \text{ and } x \sim X \neq \emptyset \}$.

When the subcomponents of a structure which are accessed outside the critical region for the structure have been determined, then the requests and releases for the subcomponents can be located in the high level procedures. Figures 5.30a through 5.30b locate the requests for the subcomponents. We can use the transforms in Figures 5.10a through 5.10c to locate the releases for the structures and the subcomponents resulting from the refinements. Figure 5.31 gives the procedure from Figure 5.28 after $E1('\text{union1'A.AI.name}, A), C1('\text{union1'A.AI.name}), E1('\text{union1'B.Bl.name}, B), \text{ and } C1('\text{union1'B.Bl.name})$ have been applied. The locations of the new releases and requests for the structures and the substructures are shown.
procedure union1(A, A1, B, B1)

begin
  request(A, B)
  cobegin
    begin
      if has2(A, A1) then
        begin
          release(A);
        end
      else begin
        insert2(A, A1);
        release(A);
      end
    end
    begin
      if has2(B, B1) then
        begin
          release(B);
          copy2(t, B1, B, B1);
        end
    end
    begin
      release(B);
      assign2(t, B1, nil);
    end
  coend
  append2(A, A1, t, B1)
end

Figure 5.28. Relocation of releases for A and B.
DO(-anything-) =-anything

D1(<procedure and function>Y,X,U)
   = [U=0; DO(<procedure or function heading>),
      D2(<block>Y,X,U))
      if (G1 and G3) or (G2 and G5)

D2(<block>Y,X,U)
   = [DO(<block>); U=U+Z(X,Y)]
      if X not in Y
      = [DO(begin) D3(<statement list>Y,X,U) DO(end)]
      if X in Y

D3(<statement list>Y,X,U)
   = [DO(<statement list>Y,X,U); U=U+Z(X,Y)]
      if X not in Y
      D4(<statement>Y,X,U)
      if X in Y and G8
      = [U=U+Z(X,Y); D4(<statement>Y",X,U);
         DO(<statement list>Y")]
      if X not in Y" and G9
      D0(<statement>Y") D3(<statement list>Y",X,U)
      if X in Y" and G9

D4(<statement>Y,X,U)
   = [DO(<assignment statement>); if G10
      DO(<procedure or function statement>); if G11
      DO(<access control statement>); if G14
      D5(<structured statement>Y,X,U); if G13

Figure 5.29a. Analysis functions to determine those refinements of A accessed outside H'(A).
Figure 5.29b. Analysis functions to determine those refinements of $A$ accessed outside $H'(A)$.
\[ E0(-anything-Y)=-anything- \]

\[ E1(<procedure and function>Y,X,Z) \]
\[ \begin{cases} 
E0(<procedure heading>E2(<block>Y,X,Z) \\
\text{if } G1 \text{ and } G3 \\
E0(<function heading>E2(<block>Y,X,Z) \\
\text{if } G2 \text{ and } G5 
\end{cases} \]

\[ E2(<block>Y,X,Z) \]
\[ \begin{cases} 
E0(<block>Y) \\
\text{if } X \text{ not in } Y 
\end{cases} \]

\[ E3(<statement list>Y,X,Z) \]
\[ \begin{cases} 
E4(<statement>Y,X,Z) \\
\text{if } X \text{ in } Y \text{ and } G8 
\end{cases} \]

\[ E4(<statement>Y',X,Z)E0(<statement list>Y') \]
\[ \text{if } (X \text{ in } Y' \text{ or } Z \text{ not in } Y') \text{ and } G9 \]

\[ E0(<statement>Y'E3(statement list>Y'',X,Z) \\
\text{if } X \text{ not in } Y' \text{ and } G9 \]

\[ E4(<statement>Y,X,Z) \]
\[ \begin{cases} 
request(X)E0(<assignment statement>Y); \text{if } G10 \\
request(X)E0(<procedure or function statement>Y) \\
\text{if } G11 \text{ or } G12 \\
request(X)E0(release(Z)); \text{if } G14 \text{ and } G25 \\
E5(<structured statement>Y,X,Z) \text{if } G13 
\end{cases} \]

Figure 5.30a. Transformations for locating requests of substructures.
E5({structured statement}Y,X,Z) 
= E2({block}Y,X,Z);if G16 
  = E6({conditional statement}Y,X,Z);if G17 
      = E9({repetitive statement}Y,X,Z);if G19 
          E10({parallel block}Y,X,Z);if G18 

E6({conditional statement}Y,X,Z) 
= E7(if<expression>Y' 
     then<block>Y"else<block>Y"',X,Z) by G20 

E7(if<expression>Y'then<block>Y"else<block>Y"',X,Z) 
  Request (X) E0(if<expression>Y' 
                 then<block>Y"else<block>Y"'if X in Y' 
= E0(if<expression>Y' 
     then) E8({block}Y",X,Z) E0(else) 
     E8({block}Y"',X,Z) if X not in Y' 

E8({block}Y,X,Z) 
= E2({block}Y 
     if Z in Y and X in Y 
     E0(begin)request(X)E0({statement list}end) 
     = if X not in Y and Z not in Y 
        and by G7 
     E0(begin)E4({statement list}Y)E0(end) 
     if ((X in Y and Z not Y in Y) or 
         (X not in Y and Z in Y)) and by G7 

E9({repetitive statement}Y,X,Z) 
= request(X)E0({repetitive statement}) 

E10({parallel block}Y,X,Z) 
= E0(cobegin)E11({parallel statement list}Y,X,Z) 
    E0(coend) by G22 

E11({parallel statement list}Y,X,Z) 
= E2({block}Y,Z,Z);if G23 
    = E2({block}Y',X,Z)E11({parallel statement list}Y",X,Z) 
       if G24 

Figure 5.30b. Transformations for locating requests of substructures.
procedure union1(A, A1, B, B1)

begin
  request(A, B)
  cobegin
  begin
    request(A, A1, name)
    if has2(A, A1) then
      begin
        release(A)
      end
    else
      begin
        insert2(A, A1)
        release(A)
      end
  end
  begin
    request(B, B1, name)
    if has2(B, B1) then
      begin
        release(B)
        copy2(t, B1, B, B1)
        release(B, B1, name)
      end
    else
      begin
        release(B)
        release(B, B1, name)
        assign2(t, B1, nil)
      end
  end
  coend
append2(A, A1, t, B1)
release(A, A1, name)
end

Figure 5.31. Requests and releases for A.A1.name and B.B1.name.
After the requests and releases for the substructures have been placed in the high level procedures, those procedures may be analyzed to determine which of the requests and releases may be moved into the low level procedure. This may lead to an increase in concurrency.

If the use of a low level procedure by high level procedure is immediately followed in all execution sequences by a release to a structure, the use is said to exit protected with respect to the structure. If, in addition, there exists no operations, or only release operations, between the procedure use and the release, then the release is movable with respect to the low level procedure use. For the procedure in Figure 5.31, the release for B is movable with respect to copy2, we may increase concurrency by moving the releases into the low level procedures. Given the low level procedures in Figures 5.18 through 5.22 and the high level procedure in Figure 5.31, we can increase concurrency and still maintain correctness with respect to concurrency by restating the procedures as in Figures 5.32 through 5.37.

Such increases in concurrency can be, in some cases, actually designed into the system procedures by designing the procedures to take advantage of the possible release movements. We do not consider any such design procedures. We are only interested in using the existing procedures generated by the system designer.
procedure union1(A, A1, B, B1)

begin
request(A, B)
cobegin
begin
request(A, A1, name)
if has2(A, A1)
then
begin
release(A)
end
else
begin
insert2(A, A1)
end
end
begin
request(B, B1, name)
if has2(B, B1)
then
begin
release(B)
copy2(t, B1, B, B1)
end
else
begin
release(B)
assign2(t, B1, nil)
end
end
coend
append2(A, A1, t, B1)
end

Figure 5.32. The procedure 'union1' with monitor uses moved into the low level.
function has2(A, B): returns boolean:

begin
  if B.name in A
  then
    begin
      if B.part=nil
      then
        begin
          return(true)
        end
      else
        begin
          if has3(A, B.name, B.part)
            then
              begin
                return(true)
              end
            else
              begin
                return(false)
              end
        end
  else
    begin
      return(false)
    end
end

Figure 5.33. The low level procedure 'has2'.

**procedure** `insert2(A,B)`

```
begin
  if B.name in A then
    begin
      release(A)
      insert3(A,B.name, B.part)
    end
  else
    begin
      $A := A + B.name$
      release(A)
      if B.part=nil then
        begin
        end
      else
        begin
          insert3(A,B.name, B.part)
        end
    end
end
```

**Figure 5.34.** The low level procedure 'insert2' with releases.

**procedure** `copy2(A,A1,B,B1)`

```
begin
  if A1.part=nil then
    begin
      erase3(A,A1.name)
      copyval3(A,A1.name, B,B1.name)
      release(B,B1.name)
    end
  else
    begin
    end
end
```

**Figure 5.35.** The low level procedure 'copy2' with release.
procedure append2(A,A1,B,B1)
begin
  if A1.part=nil then
    begin
      copyval3(A,A1.name, B,B1.name)
      release(A,A1.name)
    end
  else
    begin
      release(A,A1.name)
    end
end

Figure 5.36. The low level procedure 'append2' with releases.

procedure assign2(A,B,C)
begin
  if B.name in A then
    begin
      erase3(A,B.name)
      copyval3(A,B.name,C,val)
    end
  else
    begin
      A:=A+B.name
      copyval3(A,B.name,C,val)
    end
end

Figure 5.37. The low level procedure 'assign2'.

The high level procedures are now complete. The low level procedures may now become the high level procedures and the process begins again. The algorithm which binds these transformations and concepts together is given in Figure 5.38.
begin
L1: input-first-level-procedures-and-structures(H,Y0)
if A1(H')=false for any H' in H
then notify designer and wait
if acceptable
then continue
else go to L1
place-original-controls(H,Y0)
done:=false
while done do
begin
L2: input-next-level-procedures-and-structures(L,Y2)
if A1(L')=false for any L' in L
then notify designer and wait
if acceptable
then continue
else go to L2
restate-operands-of-high-level(H,Y1,Y0,L,Y2)
set-temporary-controls(H,Y1,Y0)
analyze-and-move-controls(H,Y1,L,Y2)
Y0:=Y2
H:=L
end
input(done)

Figure 5.38. The general design algorithm.

The routines and variables are listed below.

1) input-first-level-procedures-and-structures(H,Y0)

This is an initialization step where the outer level procedures are input along with the domains and ranges of instructions.

2) place-original-controls(H,Y0)

This is another initialization step where the original requests and releases are placed. This can be done by the designer or by using the B1 and C1 transformations.
3) **input-next-level-procedures-and-structures**(L,Y2)

   The first step of the iterative process. The next level procedures are input along with the domains and ranges of the structure refinements.

4) **restate-operands-of-high-level**(H,Y1,Y0,L,Y2)

   The domains and ranges of the high level procedures are restated in terms of the low level structure refinements.

5) **set-temporary-controls**(H,Y1,Y0)

   The temporary requests and releases are placed in the high level procedures.

6) **analyze-and-move-controls**(H,Y1,L,Y2)

   The high level and low level procedures are analyzed to determine which releases may be moved. They are then moved into the low level procedures.

7) **H**: the high level procedures,

8) **L**: the low level procedures,

9) **Y2**: the unrefined high level ranges and domains,

10) **Y1**: the refined high level ranges and domains, and

11) **Y2**: the low level domains and ranges.

   The only procedures to be considered here will be 'set-temporary-controls' and 'analyze-and-move-controls'. These procedures are given in Figures 5.39 and 5.40.
set-temporary-controls(H,Y1,Y0,L,Y2)

begin 
for every high level procedure H' in H do 
  for every X in Y0 do 
    "remove all release to X from H'"
    U=D1(H',X,P(X))
    C1(H',X)
    for every X' of X in U do 
      E1(H',X',X)
      C1(H',X')
  end
end

Figure 5.39. The procedure 'set-temporary-controls'.

analyze-and-move-controls(H,Y1,L,Y2)

begin 
for every low level process L' of L do 
  for every X in Y2 of L' do 
    if for every use of L' in every H' of H 
      there is a movable release for X 
      then 
      remove all movable releases for X with respect to L' 
      C1(L',X)
    end 
  for every X in Y1 of H do 
    if for every use of L' in every H' of H 
      there is a movable release for X 
      then 
      remove all movable releases for X with respect to L' 
      C1(L',X)
  end 
end

Figure 5.40. The procedure 'analyze-and-move-controls'.
These procedures plus the transformations A1, B1, C1, D1, and E1 compose the entire algorithm which, if used as the system is designed, will increase concurrency and guarantee correctness with respect to the problems of concurrency.

We discuss the demonstrations of correctness in the next chapter.
6. CORRECTNESS OF PROCEDURES

6.1. Introduction

The algorithm presented in the last chapter operated on the procedures generated by the designer. Two of its objectives were to detect processes which did not satisfy the sufficient conditions for determinacy and to place monitor uses so as to increase concurrency. It was also required to place those uses to ensure information integrity and to avoid creating any possible deadlock situation. We will demonstrate in this chapter that those objectives were achieved.

6.2. Determinacy

The processes are analyzed for the sufficient conditions of determinacy by the functions resulting from the application of function A1. The value true is to be returned if these conditions are satisfied; otherwise, the value false is to be returned. In this section, we will demonstrate that these functions behave in this manner.

The only way a false value can be returned is by the application of the functions A9 and A10, as shown in Figure 6.1. If the function A9 is not applied to a procedure, then
the value true is returned. Since such procedures do not contain the parallel block construct, they are strictly sequential. Hence, they are determinate. Thus, it suffices to show that A9 and A10 will return the value true if a parallel block satisfies the sufficient conditions for determinacy and false otherwise.

\[
A9(\text{<parallel statement list}> Y) = \begin{cases} A2(\text{<block}> Y) \\ A10(\text{<block}> Y' \text{<parallel statement list}> Y'') \end{cases}
\]

\[
A10(\text{<block}> Y' \text{<parallel statement list}> Y'') = \begin{cases} A2(\text{<block}> Y') \& A9(\text{<parallel statement list}> Y'') \\ \text{false} \end{cases}
\]

if \( Y' \sim Y'' = \emptyset \)

if \( Y' \sim Y'' \neq \emptyset \)

Figure 6.1. The key transformations for determinacy analysis.

Consider the generalized parallel block construct as shown in Figure 6.2. We will assume that each block \((i)\) is determinate. Let \( Y(i) = (R(i), D(i)) \) be the range and domain of block \((i)\). Then define \( Y(i) \sim Y(j) = \emptyset \), if

\[ R(i) \sim R(j) = D(i) \sim D(j) = \emptyset. \]

In order for the sufficient conditions for determinacy to be satisfied, then for all \( Y(i) \) and \( Y(j) \), \( i \neq j \), \( Y(i) \sim Y(j) = \emptyset \).

Let the parallel block as represented in Figure 6.2 satisfy the sufficient conditions for determinacy. Then for
all \( Y(i) \) and \( Y(j) \), \( i \neq j \), \( Y(i) \sim Y(j) = \emptyset \). Consider the application the function \( A9 \) as applied to a parallel statement list where the parallel statement list is constructed of

\[
\text{block}(1), \text{block}(2), \ldots, \text{block}(n).
\]

Then, for \( i = 1, \ldots, n-1 \), we have

\[
A9(<\text{parallel statement list}>Y) = A10(<\text{block}>Y(i)<\text{parallel statement list}>Y) = A2(<\text{block}>Y(i)) \& A9(<\text{parallel statement list}>Y)
\]

where \( Y'' = Y(i+1) \cup Y(i+2) \cup \ldots \cup Y(n) \). Then \( Y(i) \sim Y'' = \emptyset \).

For block(\( n \)), we have

\[
A9(<\text{parallel statement list}>Y(n)) = A2(<\text{block}>Y(n)).
\]

Since each block is determinate, then for all \( i \),

\[
A2(<\text{block}>Y(i)) = \text{true}.
\]

Thus, the value true will be returned.

![Schematic form of a generalized parallel block.](image)

**Figure 6.2.** Schematic form of a generalized parallel block.

Assume the sufficient conditions are not satisfied, that is for some \( i \) and \( j \neq j \), \( Y(i) \sim Y(j) \neq \emptyset \). Consider the
application of the function $A_9$ as applied to the language construct parallel statement list constructed of the blocks, \[ \text{block}(1), \text{block}(2), \ldots, \text{block}(n) \].

Let $i < j$, then for some application of $A_9$, we have

\[ A_9(<\text{parallel statement list}>Y) = A_{10}(<\text{block}>Y(i)<\text{parallel statement list}>Y'') \]

where $Y'' = Y(i+1) \cup \ldots \cup Y(j) \cup \ldots \cup Y(n)$. Since $Y(i) \cap Y(j) \neq \emptyset$, then $Y(i) \cap Y'' \neq \emptyset$ and

\[ A_{10}(<\text{block}>Y(i)<\text{parallel statement list}>Y'') = \text{false}. \]

Thus, if the sufficient conditions for determinacy are satisfied the function will return the value true, otherwise the value false is returned.

6.3. Deadlock

In this section, we will consider the placement of the requests in the procedures and show that deadlock can not result from the concurrent execution of procedures in which the requests were placed. It is assumed that the operation of the monitor will not lead to a deadlock state as a result of concurrent uses of the monitor. That is, processes can only deadlock as a result of the order in which the individual requests are made, and not as a result of the internal operation of the monitor for each individual request. We will consider the properties of the monitor in chapter 7. It is also noted that the processes can only
deadlock as a result of the requests and not from releases.

There are only two sets of transformations which place the requests. These are initiated by the transformations B1 and E1, respectively. The first set is only applied in the initialization step to the outer-most procedures. The second set is applied to the procedures of the levels after the structures and the procedures have been refined.

The first set of transformations will only place requests in each outer-level procedure which, if executed, will request access capabilities to every structure accessed by the procedure. Also, as a result of the manner in which the requests are placed, at most one request use will occur for each execution of any outer-level procedure. By the monitor assumption, none the outer-level procedures can deadlock as a result of those requests.

The second set of transformations will place requests in the level procedures which access refinements of structures. The request for the access capabilities to the refinements \( A_1, \ldots, A_n \) will be placed in the procedures so that the request use for \( A_1 \) will execute prior to the release for \( A \). Since the procedure will have the access capability to the structure, it will be allocated access capability to the refinement immediately. That is, no request for a refinement will ever cause a procedure to wait. Thus, the procedure can not
deadlock. Therefore, both sets of transformations will result in placing request uses that will avoid deadlock.

6.4. Information Integrity

Let the algorithm, as described in chapter 5, be applied to all levels of a hierarchical structure with $H$ as a member of the set of outer-level procedures which share access to a set of information structures. Let $X_0 = \{X_{01}, \ldots, X_{0n}\}$ be the set of shared information structures that may be accessed by $H$ in the outer-level. Let $X_{ik}$ be an information structure accessed by the level $i$ processes as a result of the execution of $H$. Let $X_{i+1j}$ be a refinement of some structure accessed by the level $i$ processes which may be accessed by some level $i+1$ process as a result of the execution of $H$.

Then we can represent the structures accessed as a result of the execution of $H$ at any level by

$$X_0 = \{X_{01}, \ldots, X_{0n}\} \text{ for level } 0,$$

$$X_1 = \{X_{11}, \ldots, X_{1n}\} \text{ for level } 1,$$

$$\vdots$$

$$\vdots$$

$$X_k = \{X_{k1}, \ldots, X_{kn}\} \text{ for the inner-most level } k.$$

Consider the characteristic action sequence resulting from the execution of $H$ where $C(X_{ij})$ represents the execution
of the critical region with respect to $X_{ij}$. As a result of
the application of the transformations initiated by the
function $B1$, then the characteristic action sequence for $H$
will be such that request($X_0$) will precede any $C(X_{ij})$ for all
$X_{ij}$.

Let $X_{ij}$ be a structure accessed by level $i$ and let $X_{i+1m}$
be a refinement of $X_{ij}$ accessed outside the critical region
of $X_{ij}$ by level $i+1$. Then as a result of $B1$ or $E1$ either
request($X_0$) precedes request($X_{i+1m}$) or request($X_{ij}$) precedes
request($X_{i+1m}$). Also, as a result of $E1$ and $C1$,
request($X_{i+1m}$) will precede release($X_{ij}$) and $C(X_{i+1m})$ will
precede release($X_{i+1m}$).

Since the above is true for all outer level procedures,
then the sufficient conditions are satisfied for Proposition
4.6, and the request and release uses will be placed such
that the information integrity of all the shared information
structures will be preserved.

6.5. Conclusion

We have shown that the transformation will place the
monitor uses such that the procedures are analyzed for the
sufficient conditions of determinacy, deadlock free, and
information integrity. The joint application of the
transformations and the algorithm will also increase
potential concurrency by placing releases such that structures will be released as quickly as possible while still maintaining the above conditions of correctness.
7. MONITORS

7.1. Introduction

This chapter discusses the three monitors described in chapter 4. We address five specific problems: 1) the global use of the monitor, 2) the form of the data structures used by the monitor, 3) the binding of argument at low levels to the appropriate data structures, 4) the deadlock problem, and 5) the correct allocation of access capabilities with respect to mutual exclusion. For monitors 1 and 2, only points 4 and 5 are applicable. Monitor 3 will contain a larger and more detailed data structure, since it will be required to honor access requests to substructures. Monitor 3, also, takes advantages of the hierarchical structure of the system and some properties about the location of the requests and releases within the procedures.

All monitors will make use of a dynamic data structure which will be private to the monitor, but shared among copies of the monitor procedures. This will be a tree structure whose terminal nodes may contain control information or semaphore values. As access capabilities are requested and released, branches may be added to or deleted from the tree, control information modified, and processes placed into or removed from the waiting lists associated with the
semaphores.

An alternative to this method is to place the control information in the information structures operated on by hierarchical processes. This method presents some potential problems which we have avoided by our approach. First, a process may be requesting access capabilities to test for the existence of a structure. If the control information were included in the information structures, special control information would be needed to be added to indicate the existence, or nonexistence, of the requested structure. Since the proposed monitors and structures are only concerned with logical access capabilities, then the existence of a structure is not important. Second, in the proposed monitor the control information is isolated from direct access from the system processes while this information may be accessible to the processes if included in the system information structures. Finally, since only a portion of the information structures will be accessed at any time, much of the additional space and time required to handle the control information within the information structures is wasted.
The control structure used by monitor 1 is given in Figure 7.1. It is referenced by the pointer 'ctree' and contains two subcomponents: 'mutex' and 'parts'. The subcomponent 'parts' will contain the control information about the access capabilities requested and granted. The subcomponent 'mutex' is a binary semaphore used by the monitor procedures to ensure mutual exclusion to the 'parts' component when adding to, deleting from, or testing that component. It is initialized to the value 1 when created. All subcomponents of the 'parts' component will have branch names corresponding to the name of the structure requested. Each subcomponent of parts will in turn contain three subcomponents as shown in Figure 7.2. The 'count' subcomponent will give the number of processes presently accessing and waiting for access capabilities to the structure. The 'mutex' and 'waiters' subcomponents are the semaphores used to ensure mutual exclusion to the 'count' component and to place a process in the wait state, respectively. They are initialized to the values 1 and 0 when created, respectively.
The waiting lists associated with each semaphore are represented as in Figure 7.3. The waiting processes are represented by the process identifier which is a unique identifier assigned to each process when it is initiated. We are not concerned here with the method of placing process identifiers in these lists, or the method of selection and restarting any process from that list. We only assume that the methods are 'fair' by some definition.
The detailed specifications of the monitor procedures request and release are given in Figures 7.4 and 7.5. These specifications are similar to the procedure definitions. Let Figure 7.6 be the initial state of the control tree, then Figures 7.7 and 7.8 give the final states if either request (Jones) or release (Smith) are applied to that initial state.

Figure 7.3. The waiting lists associated with the semaphores Q.ctree.mutex, Q.ctree.partsA.mutex, and Q.ctree.partsA.waiters.
request (A)

begin
  P(ctree*mutex)
  if A not in ctree*parts
  then do
    ctree*parts=ctree*parts+A
    P(ctree*parts*A*mutex)
    V(ctree*mutex)
    ctree*parts*A*count=1
    V(ctree*parts*A*mutex)
  end
  else do
    P(ctree*parts*A*mutex)
    V(ctree*mutex)
    ctree*parts*A*count=ctree*parts*A*count+1
    V(ctree*parts*A*mutex)
    P(ctree*parts*A*waiters)
  end

end

Figure 7.4. The procedural definition of the request procedure of monitor 1.

release (A)

begin
  P(ctree*mutex)
  P(ctree*parts*A*mutex)
  ctree*parts*A*count=ctree*parts*A*count-1
  if ctree*parts*A*count>0
  then do
    V(ctree*parts*A*waiters)
    V(ctree*parts*A*mutex)
    V(ctree*mutex)
  end
  else do
    V(ctree*parts*A*mutex)
    ctree*parts=ctree*parts-A
    V(ctree*mutex)
  end

end

Figure 7.5. The procedural definition of the release procedure of monitor 1.
Figure 7.6. An initial state of the control tree of monitor 1.

Figure 7.7. The state of the control tree after request (Jones) has been applied to the state in Figure 7.6.
Figure 7.8. The state of the control tree after request (Smith) has been applied to the state in Figure 7.6.

Monitor 1 will only control the granting and releasing of access capabilities to single, individually requested structures. As mentioned in chapter 4, the improper placement of the request uses can result in a deadlock. We will show here that monitor will ensure mutual exclusion of requested structures and will not allow deadlock between single uses of the monitor.
First, the monitor uses will guarantee mutual exclusion to any structure A. We will consider two cases the interaction between concurrent requests for A and the possible interaction between concurrent requests and releases for A. In the first case, and as a result of the operation \( P(\text{ctree}\cdot\text{mutex}) \), only one process can test whether A is a component of ctree\cdot parts. (This corresponds to testing whether the access capabilities have been granted to A.) If A is in ctree\cdot parts, then the operations \( P(\text{ctree}\cdot\text{parts}\cdot A\cdot\text{mutex}) \) and \( V(\text{ctree}\cdot\text{mutex}) \) are performed. This will allow some competing process to test for A and gives the first process exclusive access to ctree\cdot parts\cdot A\cdot count. The first process then increments the count, gives up exclusive control of the count variable and places itself in the wait list. If A is not in ctree\cdot parts, then A is added to ctree\cdot parts, exclusive control acquired to the count variable, and \( V(\text{ctree}\cdot\text{mutex}) \) performed. The first process then increments the count, executes \( V(\text{ctree}\cdot\text{parts}\cdot A\cdot\text{mutex}) \) and is granted the access capabilities to A. In any case, until the access capabilities to A have been released any requesting process will find the access capabilities have been granted and will be placed in the wait state. Thus, at any point in time only one process will have access capabilities to A.
If the access capabilities to A have been granted, then the interaction between requests and releases will not result in more than one process with access capabilities to A. If the operation \( P(\text{ctree*mutex}) \) is executed by any release, no process can gain access to the tree until the release has performed \( V(\text{ctree*mutex}) \). In all cases, this is the last operation performed by the release. Thus, either no process will have access to A when the release terminates or only one process will be selected by the release procedure to have access. It was demonstrated in the previous paragraph that under these conditions, no more than one process can gain access to A. Thus, at no time can more than one process have access capabilities to any structure A.

We now show that the manner of use of the various semaphores in the monitor will not in itself lead to deadlock. To show this, assume competing requests for a resource A. First, because no request can perform \( P(\text{ctree*parts*A*mutex}) \) without first performing \( P(\text{ctree*mutex}) \), no request will be waiting on \( \text{ctree*mutex} \) and have access to \( \text{ctree*parts*A*count} \). If a process is in the wait list for \( \text{ctree*parts*A*waiters} \), then it has performed \( V(\text{ctree*parts*A*mutex}) \) and has allowed some competing request to continue. In any case, the request will not deadlock. Since the release use maintains exclusive control of the ctree when executing, the only way it can produce a deadlock
is for some request to maintain exclusive control of 
ctree parts account and be waiting for control of ctree. 
This can never happen; thus, no deadlock can occur between 
competing requests or releases.

Thus, monitor 1 will ensure mutual exclusion of 
requested structures and will not deadlock as a result of the 
interaction of concurrent uses of the monitor.

7.3. Monitor 2

The second monitor differs from the first in that it can
process a list of structures. This allows us to avoid the
potential deadlock resulting from the improper use of monitor
1. The control structure is the same, only the specification
of the monitor differs. In order to avoid potential
deadlock, the input list is sorted alphabetically and the
request for each element in the list is processed according
to the sorted order. The monitor procedures are given in
Figure 7.9 and 7.10. They have the equivalent effect as the
procedures in Figure 7.11, where request' and release' are
monitor procedures for monitor 1. Since monitor 1 guarantees
mutual exclusion, then the procedures of monitor 2 will also
guarantee mutual exclusion. The only potential of deadlock
is when two or more processes have request lists containing
structure names which are the same. For example, if the two
uses, request \((A, B)\) and request \((B, A)\), were realized concurrently as request \((A)\) request \((B)\), ..., then the processes would be deadlocked. As a result of the sort, the request and allocation sequences satisfy the requirements of the ordered resource usage policy (20); thus, they will not deadlock.

\[
\text{request}(A) \quad \{\text{where } A = (A_1, \ldots, A_n)\}
\]

\[
\text{begin} \\
\quad A' = \text{sort}(A) \quad \{\text{where } A' = (A'_1, \ldots, A'_n)\} \\
\quad i = 1 \\
\quad \text{while } i \leq n \text{ do} \\
\quad \quad P(\text{ctree} \cdot \text{mutex}) \\
\quad \quad \text{if } A'_i \text{ not in ctree} \cdot \text{parts} \\
\quad \quad \quad \text{then do} \\
\quad \quad \quad \quad ctree \cdot \text{parts} = ctree \cdot \text{parts} + A'_i \\
\quad \quad \quad \quad P(\text{ctree} \cdot \text{parts} \cdot A'_i \cdot \text{mutex}) \\
\quad \quad \quad \quad V(\text{ctree} \cdot \text{mutex}) \\
\quad \quad \quad \quad ctree \cdot \text{parts} \cdot A'_i \cdot \text{count} = 1 \\
\quad \quad \quad \quad V(\text{ctree} \cdot \text{parts} \cdot A'_i \cdot \text{mutex}) \\
\quad \quad \quad \text{end} \\
\quad \quad \text{else do} \\
\quad \quad \quad P(\text{ctree} \cdot \text{parts} \cdot A'_i \cdot \text{mutex}) \\
\quad \quad \quad V(\text{ctree} \cdot \text{parts}) \\
\quad \quad \quad ctree \cdot \text{parts} \cdot A'_i \cdot \text{count} = ctree \cdot \text{parts} \cdot A'_i \cdot \text{count} + 1 \\
\quad \quad \quad V(\text{ctree} \cdot \text{parts} \cdot A'_i \cdot \text{mutex}) \\
\quad \quad \quad P(\text{ctree} \cdot \text{parts} \cdot A'_i \cdot \text{waiters}) \\
\quad \quad \text{end} \\
\quad \quad i = i + 1 \\
\quad \text{end} \\
\text{end}
\]

Figure 7.9. The request procedure of monitor 2.
**release(A)**

\begin{align*}
\text{begin} \\
i = 1 \\
\text{while } i \leq n \text{ do} \\
\quad P(\text{ctree}\cdot \text{mutex}) \\
\quad P(\text{ctree}\cdot \text{parts}\cdot A_i \cdot \text{mutex}) \\
\quad \text{ctree}\cdot \text{parts}\cdot A_i \cdot \text{count} = \text{ctree}\cdot \text{parts}\cdot A_i \cdot \text{count} - 1 \\
\quad \text{if } \text{ctree}\cdot \text{parts}\cdot A_i \cdot \text{count} > 0 \\
\quad \quad \text{then do} \\
\quad \quad \quad \text{V(ctree}\cdot \text{parts}\cdot A_i \cdot \text{waiters}) \\
\quad \quad \quad \text{V(ctree}\cdot \text{parts}\cdot A_i \cdot \text{mutex}) \\
\quad \quad \quad \text{V(ctree}\cdot \text{mutex}) \\
\quad \quad \text{end} \\
\quad \text{also do} \\
\quad \quad \text{V(ctree}\cdot \text{parts}\cdot A_i \cdot \text{mutex}) \\
\quad \quad \text{ctree}\cdot \text{part} = \text{ctree}\cdot \text{parts}\cdot A_i \\
\quad \quad \text{V(ctree}\cdot \text{mutex}) \\
\quad \text{end} \\
\quad i = i + 1 \\
\text{end}
\end{align*}

*Figure 7.10. The release procedure of monitor 2.*

**request(A)**

\begin{align*}
\text{begin} \\
A' = \text{sort}(A) \\
i = 1 \\
\text{while } i \leq n \text{ do} \\
\quad \text{request'}(A_i) \\
\quad i = i + 1 \\
\text{end}
\end{align*}

*Figure 7.11. The equivalent monitor 2 procedures using the monitor 1 procedures.*
Monitor 3 is the assumed monitor when the request and release uses are placed in the procedures by the algorithm in chapter 5. The important difference between this monitor and the other two is that a process is allowed to request, and be granted, access capabilities to subcomponents of structures to which that process already possesses access capabilities. As a result of this difference, requests and releases will be located throughout the various levels of the hierarchical structure. This will allow some structures to be requested at one level and released at a lower level. Thus, the monitor uses must be serviceable from any level.

As a process is being realized and processes in the lower level are initiated, the identity of the original structures accessed at the higher levels is lost to the processes at the lower levels. The monitor needs some method of binding the requests and releases to the appropriate structure pathname. We assume the following method.

The control structure for monitor 3 is shown in Figures 7.12 and 7.13. The first level will be the root level of the control tree and any requests will result in possible additions to the parts component of this level. The request
for any structure, or substructure, not in the tree will result in the addition of a structure at some lower level in the tree of the form shown in Figure 7.13. Both forms contain the semaphore 'mutex', a 'count' variable, and a 'parts' component as in monitors 1 and 2. The lower level component also contains the 'waiters' semaphore and an additional counter 'wait_count' used to count the number of waiting processes. Both forms also contain the pointer variable 'pred' to point to the immediate predecessor structure. In the lower levels, this will be used by the releases to return back up the tree to remove unnecessary branching nodes. In the first level, the pointer variable has the value 'nil' and is used to ground the reference chain. If, for example, a process issued a request for the structure A and the subcomponents A.B and A.B.C, then the control structure would have the form as shown in Figure 7.14. When a new structure or substructure is added to the tree, the semaphores and counters are initialized. At this point, the 'parts' component will contain no branches, and the 'pred' component will not point to anything.

An argument passed to the monitor will be assumed to have the form as shown in Figure 7.15. The 'name' component will be the name of a structure, the 'ref' component will be used to point to a substructure in the control tree, and the 'succ' component will be the name of a logical subcomponent
of the named structure. When the initial request is made, say to the structure A, the argument structure will have the state as shown in Figure 7.16. This will result in the argument and the control tree having the state as in Figure 7.17. It is the responsibility of the monitor to bind the argument. This reference will be assumed present when the same process either releases A or requests or releases a subcomponent of A.

![Diagram of control structure for monitor 3](image)

**Figure 7.12.** The first level of the control structure for monitor 3.

![Diagram of lower level component of control structure](image)

**Figure 7.13.** The format of any lower level component of the control structure for monitor 3.
Figure 7.14. An example state of the control tree for monitor 3.
When a process requests a subcomponent of A, the argument structure will be assumed to have the form as in Figure 7.18 shown bound to the control tree. This will result in state of the control tree and arguments as shown in Figure 7.19. The references in the arguments are assumed to be carried from one level in the hierarchical structure to the next, so that, even though the identity of the original structure is lost, the control component for a subcomponent will be bound to all references to the subcomponent and allow the monitor to grant requests for further subcomponents and process all releases.
Figure 7.16. The state of an argument structure when an initial request is made for the structure A.

Figure 7.17. An example state of the control tree after the initial request for the structure A.
Figure 7.18. The state the argument structure at the request for A.B.
Figure 7.19. The state of the control tree after request A.B.

Each level of procedures will contain monitor procedures. The request and release procedures for the outer level will be in level 1 and are specified in Figures 7.20 and 7.21. The request and release procedures for level i (i≥1) will be in level i+1 and are shown in Figure 7.22 and
7.23. They also take into account the fact that if any request is issued by any level (outside level 0), then the requested structure must be a logical subcomponent of a structure in the control tree. They also make use of the fact that a level may release a structure or a subcomponent of a structure.

By extension of the arguments for monitors 1 and 2 with respect to the properties of the location of the requests and releases, monitor 3 can be shown to be deadlock free and to ensure mutual exclusion.
\[ \text{request}(A) \quad \{ \text{where } A = (A_1, \ldots, A_n) \} \]

\begin{align*}
\text{begin} \\
A' &= \text{sort}(A) \quad \{ \text{where } A' = (A'_1, \ldots, A'_n) \} \\
i &= 1 \\
\text{while } i \leq n \text{ do} \\
P(\text{ctree} \cdot \text{mutex}) \\
\text{ctree} \cdot \text{count} &= \text{ctree} \cdot \text{count} + 1 \\
X &= A'_i \cdot \text{name} \\
\text{if } X \text{ not in } \text{ctree} \cdot \text{parts} \\
\text{then do} \\
\text{ctree} \cdot \text{parts} &= \text{ctree} \cdot \text{parts} + X \\
P(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{mutex}) \\
V(\text{ctree} \cdot \text{mutex}) \\
\text{point}(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{pred}, \text{ctree}) \\
\text{point}(A'_i \cdot \text{ref}, \text{ctree} \cdot \text{parts} \cdot X) \\
V(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{mutex}) \\
\text{end} \\
\text{else do} \\
P(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{mutex}) \\
V(\text{ctree} \cdot \text{mutex}) \\
\text{point}(A'_i \cdot \text{ref}, \text{ctree} \cdot \text{parts} \cdot X) \\
\text{ctree} \cdot \text{parts} \cdot X \cdot \text{count} &= \text{ctree} \cdot \text{parts} \cdot X \cdot \text{count} + 1 \\
\text{if } \text{ctree} \cdot \text{parts} \cdot X \cdot \text{count} > 1 \\
\text{then do} \\
\text{ctree} \cdot \text{parts} \cdot X \cdot \text{waitcount} &= \text{ctree} \cdot \text{parts} \cdot X \cdot \text{waitcount} + 1 \\
V(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{mutex}) \\
P(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{waiters}) \\
P(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{mutex}) \\
\text{end} \\
V(\text{ctree} \cdot \text{parts} \cdot X \cdot \text{mutex}) \\
\text{end} \\
i &= i + 1 \\
\text{end} \\
\text{end}
\end{align*}

Figure 7.20. The request procedure at level 1.
release(A) \{where A=(A_1, \ldots, A_n)\}

begin
  \textcolor{red}{i=1}
  \textcolor{red}{\textbf{while} i \leq n do}
    \textcolor{red}{\text{point}(q, ctree\_\textcolor{red}{\textbullet}\_i)}
    \textcolor{red}{P(q\_\textcolor{red}{\textbullet}mutex)}
    \textcolor{red}{q\_\textcolor{red}{\textbullet}count=q\_\textcolor{red}{\textbullet}count-1}
    \textcolor{red}{\text{if} q\_\textcolor{red}{\textbullet}waitcount>0}
      \textcolor{red}{\textbf{then} do}
        \textcolor{red}{V(q\_\textcolor{red}{\textbullet}\text{waiters})}
        \textcolor{red}{q\_\textcolor{red}{\textbullet}waitcount=q\_\textcolor{red}{\textbullet}waitcount-1}
        \textcolor{red}{V(q\_\textcolor{red}{\textbullet}mutex)}
      \textcolor{red}{P(ctree\_\textcolor{red}{\textbullet}mutex)}
      \textcolor{red}{ctree\_\textcolor{red}{\textbullet}count=ctree\_\textcolor{red}{\textbullet}count-1}
      \textcolor{red}{V(ctree\_\textcolor{red}{\textbullet}mutex)}
    \textcolor{red}{\textbf{end} do}
  \textcolor{red}{\text{else} do}
    \textcolor{red}{V(q\_\textcolor{red}{\textbullet}mutex)}
    \textcolor{red}{P(ctree\_\textcolor{red}{\textbullet}mutex)}
    \textcolor{red}{ctree\_\textcolor{red}{\textbullet}count=ctree\_\textcolor{red}{\textbullet}count-1}
    \textcolor{red}{\text{if} ctree\_\textcolor{red}{\textbullet}count=0}
      \textcolor{red}{\textbf{then} ctree\_\textcolor{red}{\textbullet}parts=empty}
    \textcolor{red}{V(ctree\_\textcolor{red}{\textbullet}mutex)}
  \textcolor{red}{\textbf{end do}}
  \textcolor{red}{i=i+1}
\textcolor{red}{\textbf{end do}}
\textcolor{red}{\textbf{end}}

Figure 7.21. The release procedure at level 1.
request(A)

begin
  point(q,A•ref)
  point(Y,A•succ)
  X=Y•name
  P(q•mutex)
  if X not in q•parts then do
    q•parts=q•parts+X
    P(q•parts•X•mutex)
    V(q•mutex)
    q•parts•X•count=q•parts•X•count+1
    point(q•parts•X•ref,q)
    point(Y•ref,q•parts•X)
    V(q•parts•X•mutex)
  end
  else do
    P(q•parts•X•mutex)
    V(q•mutex)
    point(Y•ref,q•parts•X)
    q•parts•X•count=q•parts•X•count+1
    if q•parts•X•count>1 then do
      q•parts•X•waitcount=q•parts•X•waitcount+1
      V(q•parts•X•mutex)
      P(q•parts•X•waiters)
      P(q•parts•X•mutex)
    end
    V(q•parts•X•mutex)
  end
end

Figure 7.22. The request procedure at level i(i>1).
release(A)
begin
    if A\textcdot succ=nilk
    then
        point(q,A\textcdot succ\textcdot ref)
    else
        point(q,A\textcdot ref)
    while q\textcdot pred not nil do
        P(q\textcdot mutex)
        q\textcdot count=q\textcdot count-1
        if q\textcdot count=0
        then
            q\textcdot parts=empty
        else
            if q\textcdot waitcount>0
            then
                V(q\textcdot waiters)
                q\textcdot waitcount=q\textcdot waitcount-1
            end
        end
        point(t,q)
        point(q,q\textcdot pred)
        V(t\textcdot mutex)
    end
    P(ctree\textcdot mutex)
    ctree\textcdot count=ctree\textcdot count-1
    if ctree\textcdot count=0
    then
        ctree\textcdot parts=empty
    V(ctree\textcdot mutex)
end

Figure 7.23. The release procedure at level i (i>1).
8. CONCLUSIONS

This thesis has presented a methodology and supporting theory for the development of multiuser hierarchically structured information systems. The algorithm in chapter 5, and its associated transformations, relieve the designer of the burden of analysis needed to increase concurrency and still maintain correctness in such systems. This means that each level preserves the originally implied information integrity, deadlock will not be introduced, and any indeterminacy introduced by the designer into any individual procedures will be detected. The major contributions are a formal model suitable for representing hierarchical structures, a characterization of the potential concurrency problems within information systems, the definition of compatibility and information integrity, and a study of the relationship between the correctness properties and the monitors needed to ensure these properties.

This work combines the areas of systems design, programming techniques, language design, automated program analysis and verifiers, resource allocation mechanisms, and the theory of correctness for concurrent systems of programs.

The conditions for correctness, as discussed in chapter 4, have implications for systems design and resource allocation mechanisms. Programming languages and techniques
which facilitate the development of systems that satisfy these conditions could be developed to enhance such automatic analysis and verification.

The structures considered in this study were limited to those that can be modeled by tree graphs. There is a need to investigate the potential of extending this work to include more sophisticated information structures (acyclic graph models, for example).

Performance measures and protection were not considered in this work, but need to be investigated. With the development of applicable performance measures, it may be possible to develop both acceptable and correct systems. The dynamic control of access rights may extend the definition of information integrity and result in the development of systems with secure and protected information structures.

The transformations used by the algorithm in chapter 5 are directly related to the study of formal languages. Work still needs to be done in this area. Also, there appears to be certain classes of languages over which the transformations are more easily constructed and applied. It would be helpful when constructing systems to be able to have some definition of these classes. The algorithm has applications to many languages, but some languages may have properties which allow for a higher degree of concurrency. Languages should be developed which allow for higher
concurrency and which lend themselves in the development of systems.

This thesis considered only the logical statement of information systems and not the implementation with secondary storage devices or systems whose programs are often subject to change. These areas also need to be investigated to determine the applicability of the methodology presented in this work.


33. Parnas, D. L. On the criteria to be used in decomposing systems into modules. *CACM* 15, No. 12 (December 1972): 1053-1058.


42. Varney, R. C. Process selection in a hierarchical operating system. *SIGOPS Operating Review*, June 1972


