Automated control of concurrency in multi-user hierarchical information systems

by ALAN F. SWEET and ARTHUR E. OLDEHOEFT
Iowa State University
Ames, Iowa

ABSTRACT

This paper presents a systematic approach to providing a high degree of concurrent access to information in hierarchically structured systems. An algorithm is presented which is designed to operate on the procedures and tree-structured information of two adjacent levels. The algorithm analyzes the procedure and structure refinements and generates the appropriate monitor calls to increase the degree of concurrent access. Assuming the initial level is correct, the refined system remains deadlock free, the integrity of the information is preserved, and individual procedures are checked for determinacy. Graph structured models are used to illustrate examples and definitions.

INTRODUCTION

In this paper, we present a systematic approach to providing a high degree of concurrent access to information structures in a hierarchical information system. The technique may be applied to existing systems which operate on tree-structured information and which appears as levels of functionally decomposed procedures. It may also be applied to the successive levels of procedures and information tree refinements during the top-down design process of information systems.

The basic approach is to analyze concurrent processes, at a given level, for interference and to automatically place requests for access capabilities to structures at those points where the structures are first referenced. Releases are automatically placed after those points where the structures are last referenced. As a new level of procedures are specified and the information trees are refined, the analysis is repeated resulting in the introduction of new controls and possible movement of old ones. Since the information is tree-structured, the access to a substructure may, in certain cases, result in the release of the access capabilities to the predecessor node in the tree, thereby clearing the way for concurrent access to substructures at the same level in different branches of the tree.

Numerous systems have been proposed and implemented using the concept of a hierarchical structure.\cite{1,5,8,9} Our work, however, is restricted to subsystems (file systems, database systems, etc.) dedicated to a single language which might typically run as a module in a host operating system. These systems manage their own resources and control the flow of information to the users.

The complicating factor, and the major focus of this paper, is the assumption of a multi-user environment in which each user interacts with the system sharing, and possibly modifying, information. It is now critically important that our design process satisfy two additional constraints:

1. the introduction of potential concurrency wherever possible, and
2. the guarantee that the system will be correct with respect to three problems of concurrency
   a. preservation of information integrity among interfering independent processes
   b. deadlock avoidance, and
   c. determinacy within a single process

These two requirements place significant analytic burdens on the designer of such a system. Later, we will present an algorithm which can relieve the designer of these burdens. We first, however, more rigorously define the properties of the information system.

Process decomposition is based on a "uses" concept similar to that of Parnas.\cite{8} We, however, define the level of a procedure in a top-down manner as follows:

1. Level 0 is the outer-most level of the system.
2. Level i is the set of all procedures which, if they do use any procedures, use only procedures at level i+1, and
3. Level k, the inner-most level, is a set of procedures which use no other procedures.

Structures organized by this definition have three desirable properties. They are easy to test, since the interfaces between adjacent levels are the only procedure interactions that need be tested. All communication paths to system resources must use common procedures and, as a consequence, scheduling criteria can be more easily enforced.
with respect to the resources. Finally, we can guarantee a property (in the overall system) by ensuring that the property is maintained in each newly defined level.

Since this definition restricts the communication paths and does not allow intralevel communication between procedures, the structure may contain identity procedures (procedures which only perform a reference to a single procedure at the next level). This definition, for example, would require that a system whose communication and access path are as shown in Figure 1 be restructured to appear like that in Figure 2 with I₁ and I₂ as identity procedures. Identity procedures can increase the amount of overhead time spent as a result of procedure communication. Bernstein and Siegel² have recently proposed a solution to this problem with some simple hardware mechanisms which can decrease this overhead.

The algorithm we will present for placing access controls is applied to hierarchical systems like those we have just described. The algorithm assumes the existence of a monitor which can be used to control access capabilities to the structures. The algorithm will analyze and modify the procedures of two adjacent levels, placing monitor uses in the procedures to introduce additional concurrency and still guarantee the stated properties of correctness. In order to perform the modifications, the algorithm will require a parser of the procedural language and additional information about the operands of the language constructs. The monitor will have the unique property in that a process which has access capabilities to a structure will be allowed to request access capabilities to any substructures. This will allow the procedures to release the major structure while maintaining access to substructures. The result is an additional increase in potential concurrency.

**CORRECTNESS CONSIDERATIONS**

Figure 3 shows an example of an information structure. Nodes of these structures are accessed by path names. For example, the node n₁ in Figure 3 is referenced by the path name A.B, and the substructure containing nodes n₂ and n₃ is referenced by the pathname A.C.

Let P=\(u₁ \ldots uₙ\) and \(P'=v₁ \ldots vₘ\) be path names. Then \(P\) is said to contain \(P'\), if \(n≤ₘ\) and \(uᵢ=vᵢ\) for all \(i=1, \ldots, n\). In Figure 3, the path name A.G contains A.G.H. For two sets of path names \(X\) and \(Y\), we define:

\[X \cup Y = \{x_i \mid x_i \in X \text{ and } x_i \text{ contains some } y_j \in Y\}\]

The *path intersection* of \(X\) and \(Y\) is represented by \(X \cap Y\) and defined by the set union of \(X\) and \(Y\) minus \(X\) and \(Y\), i.e. \((X \cap Y) \cup (Y \cap X)\). For example, if \(X=\{A,B,A.C.D,A.G\}\) and \(Y=\{A.C,A.C.H,A.G.K\}\), then \(X \cap Y=\{A.C,A.G\}\).
We use the concept of path intersection to extend Bernstein’s\(^3\) definition of noninterference. Let \(H_1\) and \(H_2\) be procedures with the domains of pathnames, \(D_1\) and \(D_2\), and the ranges of pathnames, \(R_1\) and \(R_2\), respectively. The \(H_1\) and \(H_2\) are defined to be mutually noninterfering, if either

1. \(H_1\) is a successor or predecessor of \(H_2\), or
2. \(R_1\cap R_2 = R_1\setminus R_2 = R_2\setminus R_1 = \emptyset\)

This generalized definition, as it applies to tree-structures, forms the basis for analyzing interference and maintaining the integrity of the information in our system as well as checking for determinacy within a single process. For a procedure \(H\), we use a notational convenience \(Y=((R),(D))\) to represent the domain \(D\) and range of \(H\). For \(H_1\) and \(H_2\) we define \(Y_1\cap Y_2 = \emptyset\) if \(R_1\cap R_2 = R_1\setminus R_2 = R_2\setminus R_1 = \emptyset\).

The procedures in the system will be represented by flow graphs in which nodes represent operations and the arcs between nodes represent the sequential flow of control. Figure 4 illustrates a procedure graph and Figure 5 illustrates the special nodes which represent the monitor calls to request and release access capabilities. Figure 6 illustrates the operations cobegin and coend for parallel execution within a procedure.

When independent procedures operate concurrently on shared structures and free access is allowed to the structures, the result may be unpredictable. The term critical region\(^4,6\) has been used to describe that portion of a procedure which operates on a shared item. If a procedure \(H\) operates on the shared item \(A\), then \(H'(A)\) is used to denote the critical region of \(H\) with respect to \(A\). In terms of the procedure graph, \(H'(A)\) is the subgraph of \(H\) which may access \(A\). This idea is illustrated in Figure 7.

Any execution sequence of the critical region \(H'(A)\) is represented by \(C(A)\). For the example in Figure 7, \(C(A)\) represents either of two sequences of operations, \(n_1n_3\) or \(n_2n_5\).

Let \(H_1\) and \(H_2\) be two procedures at the same level with critical regions \(H_1'(A)\) and \(H_2'(A)\), respectively. The procedures \(H_1\) and \(H_2\) are defined to be mutually exclusive with respect to \(A\), if for all concurrent realizations of \(H_1\) and \(H_2\) either 1) \(H_1'(A)\) or \(H_2'(A)\) is empty, or 2) \(C_1(A)\) executes prior to \(C_2(A)\) or 3) \(C_2(A)\) executes prior to \(C_1(A)\). In Figure 8, the monitor is used to ensure mutual exclusion between two procedures \(H_1\) and \(H_2\).

Let a procedure \(H\) operate on a set of shared structures \(A=\{A_1, \ldots, A_n\}\) with \(H'(A)=\{H'(A_1), \ldots, H'(A_n)\}\) as corresponding critical regions for \(A\). Let \(C(A)=\{C(A_1), \ldots, C(A_n)\}\) represent the execution of the critical regions in \(H\). For example in Figure 9, \(A=\{A_1, A_2\}\) and \(C(A)=\{C(A_1), C(A_2)\}\) where \(C(A_1)=[n_2n_3n_1]\) and \(C(A_2)=[n_5n_4n_2]\). Suppose we have two procedures \(H_1\) and \(H_2\) at the same level which share such a set \(A\) of structures and let \(S_0(A)\) denote the initial state of \(A\). Let \(S_0(A)\) denote the final state of \(A\), if \(H_1\) executes prior to \(H_2\) and let \(S_1(A)\) denote the final state of \(A\) if \(H_2\) executes prior \(H_1\). The procedures \(H_1\) and \(H_2\) are said to preserve the information integrity of \(A\), if every concurrent execution of \(H_1\) and \(H_2\) either produces \(S_0(A)\) or \(S_1(S)\) as the final state. In Figure 10, for example, procedures \(H_1\) and \(H_2\) may not preserve...
Figure 8—Monitor uses to ensure mutual exclusion

Figure 9—A process with two critical regions

Figure 10—Processes operating on shared structures A₁ and A₂

The algorithm presented will assume the existence of a monitor similar to the type described by Hoare and Brinch Hansen. The monitor will have two usages, request (X) and release (X) where X={x₁, ..., xₖ} is a list of path names. The request will be allowed only if logical access can be granted for all path names in the list. Otherwise, the procedure will wait until the entire request can be granted. Release returns the logical access capabilities to the path names in the list. The monitor operates on the following data structures:

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure with access capability to xᵢ</td>
<td>P(xᵢ)</td>
</tr>
<tr>
<td>Set of procedures waiting for access capability to xᵢ</td>
<td>W(xᵢ)</td>
</tr>
<tr>
<td>Set of path names to which access has been granted</td>
<td>A</td>
</tr>
</tbody>
</table>

The monitor operations, as invoked by a procedure H, can be described as follows.

**request (X):**

if for every xᵢ in X, xᵢ−A=∅ or   
  (xᵢ−A=zᵢ implies P(zᵢ)=H)  
then  
P(xᵢ)=H for all xᵢ in X  
A=A∪X  
else  
W(xᵢ)=W(xᵢ)∪H for all xᵢ in X  
place H in wait state

**release (X):**

P(xᵢ)=∅ for every xᵢ in X  
A=A−{A∩X}  
If there exists some procedure H' and some xᵢ in X such that H' is in W(xᵢ) and H' is waiting for the request (Y)  
then  
if for every yᵢ in Y, yᵢ−A=∅ or   
  (yᵢ−A=zᵢ implies P(zᵢ)=H)  

then
remove $H'$ from wait state
$P(y_j) = H'$ for all $y_j$ in $Y$
$W(y_j) = W(y_j) - H'$ for all $y_j$ in $Y$
$A = A \cup Y$

DESCRIPTION OF THE ALGORITHM

The algorithm is designed to operate on the procedures and the information structures of two adjacent levels. We illustrate the result of applying the algorithm in Figures 14 through 19. The critical region for the procedure in Figure 14 is shown prior to the refinement of the structure $A$ and the specification of the lower level procedures. Let $F(A) = (A.A_1, A.A_2)$ be the refinement of $A$ and suppose the lower level procedure $L_1$ operates an $A.A_1$ while $L_2$ and $L_3$ operate on $A.A_2$. $O_1$ is assumed to be an operation on the major structure $A$.

The critical regions for $A$ and $F(A)$ are expressed as in Figure 15. The information integrity of $A$ is preserved by the high level monitor uses in Figure 16. The potential concurrency may be increased, deadlock avoided, and information integrity preserved for $A$, if the procedures are modified with monitor uses to appropriately request sub-structures and release the major structure. Figure 17 shows how our algorithm will perform such placements and rearrangements of request and releases.

Suppose $L_1$ and $L_3$ of the previous example have the refinements as shown in Figure 18 with the critical regions $L_1(A.A_1)$ and $L_3(A.A_2)$ respectively. The potential concurrency is increased still further, if the releases for $A.A_1$ and $A.A_2$ are removed from the high level procedure $H$ and placed in the low level procedures $L_1$ and $L_3$. Figure 19 shows how our algorithm will relocate these releases into the lower level procedures.

If every use of $L$ is followed by a release of the structure $A_i$ of $A$, and there exists no operations on $A$, other than releases, between the terminations of $L$ and the corresponding releases, of $A_i$, then these releases of $A_i$ are said to be movable with respect to $L$. This concept is exploited by our algorithm.

The algorithm operates over procedures and requires information about the domain and range operands of the language constructs. This information may be determined through a combination of precompilation of the procedures and direct specification by the designer in the case of procedure and function statements. The operands of each language construct are determined by the union of the operands of its parts. This is demonstrated in Figure 20 which shows a procedure and the operand sets $Y$, $Y_1$, $Y_2$, and $Y_3$.

The algorithm also uses a set of transformations defined over the procedural language which perform the actual placement of the requests and releases. For purposes of discussion, the algorithm is applied to an example language. There are four sets of transformations needed by the algorithm to place the controls in the procedures. These transformations along with the formal definition of the example language are given in the Appendix. These example transformations are for illustrative purposes and may be non-optimal. The algorithm initiates the transformations through the following four starting procedures:

1. $B_1(H,X)$ places the initial requests for the set $X$ of structures in the outer-most procedure $H$.
2. $C_1(H,X)$ places the releases for the structure $X$ in $H$. 

From the collection of the Computer History Museum (www.computerhistory.org)
(3) \( D1(H,X,F(X)) \) determines which of the refinements \( F(X) \) of \( X \) have critical regions outside the critical region \( H'(X) \) for the path-name \( X \) (see Figure 15 for example).

(4) \( E1(H,X',X) \) places the requests for the refinement \( X' \) of \( X \) in \( H \).

As an example, suppose we wish to execute \( C1(H,X) \). This leads to an application of \( C4 \). Figure 21 shows the transformation \( C4 \) which operates over the “statement” construct to place a release for \( X \). If the “statement” will parse as an “assignment statement” or “procedure statement,” the release for \( X \) is placed immediately following the “statement.” Otherwise, the transformation \( C5 \) is applied to the “statement,” since it must parse as a “structured statement.”

The algorithm uses one other set of functions to analyze the procedures for the sufficient conditions for determinacy. This set is initiated by \( A1(H) \) and returns the value \( \text{true} \), if the procedure \( H \) satisfies these conditions. Otherwise, the value \( \text{false} \) is returned.

The entire algorithm is presented in Figure 22. Several procedures are assumed to support the algorithm.

(1) \textbf{input-outer-level-procedures} \((H,X0)\)

This is an initialization procedure which accepts as input the source code for the outer-level procedures \( H \) and the set of path names \( X0 \) of their operand structures. In Figure 14, \( X0 = \{A\} \).

(2) \textbf{input-next-level-procedure} \((L,X2)\)

This is the first step in the iterative process. The source code of the next level procedures \( L \) and their operand structures \( X2 \) are input. In Figure 15, we

\[
\begin{align*}
&\text{procedure } L_1(A,A_1); \\
&\quad \begin{cases} \\
&\quad \text{if has2}(A,A_1) \\
&\quad \text{then begin} \\
&\quad \quad \text{erase2}(A,A_1) \\
&\quad \quad \text{end} \\
&\quad \text{else begin} \\
&\quad \quad \text{create2}(A,A_1) \\
&\quad \quad \text{end} \\
&\quad \text{end} \\
&\quad \text{end} \\
&\text{begin} \\
&\quad \text{request}(A,A_1) \\
&\quad \text{release}(A) \\
&\quad \text{L_1(A,A_1)} \\
&\quad \text{L_2(A,A_2)} \\
&\quad \text{L_3(A,A_2)} \\
&\quad \text{end} \\

\text{end}
\end{align*}
\]
have \( X_2 = \{A, A_1, A_2\} \) for \( L_1 \), \( X_2 = \{A, A_3\} \) for \( L_2 \), and
\( X_2 = \{A, A_3\} \) for \( L_3 \).
(3) restate-high-level-operands-with-refinements \((H, X_1, X_0, L, X_2)\)
The operands for the constructs of the procedure \( H \) (as defined by the grammar) are restated in terms of
the original structures \( X_0 \) and the refined structure operands \( X_2 \) accessed by the low level \( L \). These are
represented by \( X_1 \). For example, in Figure 15, we have \( X_1 = \{A, A_1, A_2\} \).
(4) remove-all-releases \((H)\)
All the releases are removed from the high level procedure \( H \).

The algorithm will relocate and generate requests and
releases as new levels are defined. An example of a high
level procedure with requests and releases is given in
Figure 23. This is prior to the input of the next level
procedures. Figure 24 gives the next level procedures
which operate on the refined structures. Figures 25 and 26
give the placement of the requests and releases after both
levels have been processed by the Algorithm in Figure 22.

begin
input-outer-level-procedures \((H, X_0)\)
if for any \( H' \) in \( H \), \( AI(H') = \text{false} \) then stop
for every \( H' \) in \( H \) with operands \( X_0' \) do
\( BI(H', X_0') \)
end
while another level exists do
begin
input-next-level-procedures \((L, X_2)\)
if for any \( L' \) in \( L \), \( AI(L') = \text{false} \) then stop
restate-high-level-operands-with-refinements \((H, X_1, X_0, L, X_2)\)
for every \( H' \) in \( H \) with operands \( X_1' \) and \( X_0' \) do
remove-all-releases \((H')\)
for every \( X \) in \( X_0' \) do
\( U = DI(H, X, F(X)) \)
\( CI(H, X) \)
end
for every \( Y \) in \( U \) do
\( EI(H, Y, X) \)
\( CI(H, Y) \)
end
end
for every \( L' \) in \( L \) with operands \( X_2' \) do
for every \( X \) in \( X_2' \) do
if \( \text{releases for } X \text{ are movable with respect to } L' \) then do
"remove release(X) with respect to \( L' \) from every high level process"
\( CI(L, X) \)
end
end
\( H \leftarrow L \)
\( X_0 \leftarrow X_2 \)
end

Figure 22—The algorithm for analyzing levels and placing monitor uses
procedure update (Employee,input)
begin
request (Employee-records)
if hasl (Employee-records, Employee)
then
begin
modifyl(Employee-records.Employee,input)
release (Employee-records)
end
else
begin
release (Employee-records)
write ('Employee not in files')
end
end
Figure 23—The high level procedure 'update' with monitor uses

procedure hasl(records, selector): returns boolean;
begin
if selector in records
then
begin
hasl:=true
end
else
begin
hasl:=false
end
end

procedure modifyl(record,input)
begin
if input.selector in record
then
begin
erase2(record.'input.selector')
assign2(record.'input.selector',input.val)
release(record)
end
else
begin
create2(record,'input.selector')
assign2(record.'input.selector',input.val)
release(record)
end
end
Figure 24—The low level procedures 'modify!' and 'has I'

procedure update(Employee,input)
begin
request (Employee-records)
if hasl(Employee-records, Employee)
then
begin
request(Employee-records,Employee)
release(Employee-records)
modifyl(Employee-records.Employee,input)
end
else
begin
release(Employee-records)
write('Employee not in files')
end
end
Figure 25—The high level procedure 'update' with refined monitor uses

procedure hasl(records,selector): returns boolean;
begin
if selector in records
then
begin
hasl:=true
end
else
begin
hasl:=false
end
end

procedure modifyl(record,input)
begin
if input.selector in record
then
begin
erase2(record.'input.selector')
assign2(record.'input.selector',input.val)
release(record)
end
else
begin
create2(record,'input.selector')
assign2(record.'input.selector',input.val)
release(record)
end
end
Figure 26—The low level procedure with refined monitor uses

Note that in its refinement, the input type has two components, selector and val.

CONCLUSION

The method described in this paper has been extended to apply to systems whose procedures are expressed in a Pascal-like language, allowing for complex control structures. It has been applied to successive levels in the design of a personnel records information system. The algorithm and its associated transformations, when used in the context described in this paper, illustrate a technique for automating the analysis needed to increase concurrency and still maintain correctness in hierarchically structured information system. This means that each level maintains the originally implied information integrity, deadlock will not be introduced, and any indeterminacy introduced by the designer into any individual procedures will be detected. This paper assumed only mutually exclusive access to structures. It is possible, however, to extend this technique by allowing concurrent readers and exclusive writers. This requires somewhat more sophisticated transformations and monitor support, but it will further increase the degree of potential concurrency.

REFERENCES

Appendix

The BNF form of the Example Procedural Language

Rule No.

\( Gl \) \quad \langle \text{procedure} \rangle = \langle \text{procedure heading} \rangle \langle \text{block} \rangle

\( G2 \) \quad \langle \text{block} \rangle = \langle \text{begin} \rangle \langle \text{statement list} \rangle \langle \text{end} \rangle

\( G3 \) \quad \langle \text{statement list} \rangle = \langle \text{statement} \rangle \langle \text{statement} \rangle \langle \text{statement list} \rangle

\( G4 \) \quad \langle \text{statement} \rangle = \langle \text{statement} \rangle \langle \text{assignment} \rangle

\( G5 \) \quad \langle \text{statement} \rangle = \langle \text{procedure statement} \rangle

\( G6 \) \quad \langle \text{statement} \rangle = \langle \text{structured statement} \rangle

\( G7 \) \quad \langle \text{statement} \rangle = \langle \text{access control statement} \rangle

\( G8 \) \quad \langle \text{structured statement} \rangle = \langle \text{block} \rangle

\( G9 \) \quad \langle \text{structured statement} \rangle = \langle \text{conditional} \rangle

\( G10 \) \quad \langle \text{structured statement} \rangle = \langle \text{parallel block} \rangle

\( G11 \) \quad \langle \text{conditional} \rangle = \langle \text{if} \rangle \langle \text{expression} \rangle \langle \text{then} \rangle \langle \text{block} \rangle \langle \text{else} \rangle \langle \text{block} \rangle

\( G12 \) \quad \langle \text{parallel block} \rangle = \langle \text{cobegin} \rangle \langle \text{parallel statement list} \rangle \langle \text{coend} \rangle

\( G13 \) \quad \langle \text{parallel statement list} \rangle = \langle \text{block} \rangle \langle \text{parallel statement list} \rangle

\( G14 \) \quad \langle \text{access control statement} \rangle = \langle \text{release} \langle \text{id list} \rangle \rangle

\( G15 \) \quad \langle \text{access control statement} \rangle = \langle \text{request} \langle \text{id list} \rangle \rangle

\( G16 \) \quad \langle \text{access control statement} \rangle = \langle \text{request} \langle \text{id list} \rangle \rangle

In the following paragraphs, \( Y, Y', Y'' \), and \( Y''' \) specify the domain-range pair associated with the syntactic unit that immediately precedes their use.

Functions which Analyze a Procedure for Noninterference

The returned result is true or false.

\( A1(\langle \text{procedure} \rangle Y) = A2(\langle \text{block} \rangle Y) \) \quad \text{if rule } G1

\( A2(\langle \text{block} \rangle Y) = A3(\langle \text{statement list} \rangle Y) \) \quad \text{if } G2

\( A3(\langle \text{statement list} \rangle Y) = \begin{cases} A4(\langle \text{statement} \rangle Y) \quad \text{if } G3 \\ A4(\langle \text{statement} \rangle Y') \wedge A3(\langle \text{statement list} \rangle Y'') \quad \text{if } G4 \\ \end{cases} \)
Transformations for Placing the Initial Requests for the Set X of Paths

Names.

The result is a procedure with request(s) for X.

B1(<procedure> Y,X)
= <procedure heading> B2(<block> Y,X) if rule G1
B2(<block> Y,X) = <block> Y if X \( \cap \) Y=\( \emptyset \)
B2(<block> Y,X) = begin B3(<statement list> Y,X) end if \( X \cap Y \neq \emptyset \) and G2
B3(<statement list> Y,X)
= B4(<statement> Y,X) if G3
B4(<statement> Y,X)
= B4(<statement> Y,X) <statement list> Y" if X \( \cap \) Y" \( \neq \) \( \emptyset \) and G4
<statement> Y" B3(<statement list> Y",X) if X \( \cap \) Y"=\( \emptyset \) and G4
B4(<statement> Y,X)
= request(X) <assignment> Y if G5
= request(X) <procedure statement> Y if G6
B5(<structured statement> Y,X)
= if G7
B5(<structured statement> Y,X)
= B2(<block> Y,X) if G9
B6(<conditional> Y,X)
= B6(if <expression> Y then <block> 1 Y" else <block> 2 Y"" ,X) if G10
request(X) if <expression> Y then <block> 1 Y"
else request(X) if <expression> Y then <block> 1 Y"
else <block> 2 Y"" ,X) if G11
= if <expression> Y then B7(<block> 2 Y",X) else
B7(<block> 2 Y",X) if X \( \cap \) Y"=\( \emptyset \)
B7(<block> Y,X)
= B2(<block> Y,X) if X \( \cap \) Y=\( \emptyset \)
Transformations for Placing the Release(s) of a Path Name X.

The result is a procedure with release(s) for X.

\[ C_1(\text{procedure } Y, X) = <\text{procedure heading } > C_2(\text{block } Y, X) \]
\[ C_2(\text{block } Y, X) = <\text{block } > Y \]
\[ \begin{cases} \text{begin } C_3(\text{statement list } Y, X) \end{cases} \]
\[ C_3(\text{statement list } Y, X) = <\text{procedure statement } > Y, X \]
\[ \begin{cases} \text{begin } C_4(\text{statement } Y, X) \end{cases} \]
\[ C_4(\text{statement } Y, X) = \begin{cases} C_4(\text{statement } Y', X) \end{cases} \]
\[ C_5(\text{structured statement } > Y, X) = <\text{procedure statement } > Y, X \]
\[ \begin{cases} \text{begin } C_6(\text{conditional } Y, X) \end{cases} \]
\[ C_6(\text{conditional } Y, X) = \begin{cases} <\text{assignment } > Y \text{ release}(X) \end{cases} \]
\[ C_7(\text{block } Y, X) = \begin{cases} <\text{block } > Y, X \end{cases} \]
\[ \begin{cases} \text{begin } \text{release}(X) \end{cases} \]

Functions for Determining Refinements of X not Accessed in Critical Region H'(X).

The returned result is a subset of the total refinement F(X) of X.

\[ D_1(\text{procedure } Y, X, F(X)) = D_2(\text{block } Y, X, F(X)) \]
\[ D_2(\text{block } Y, X, F(X)) = <\text{block } > Y, X, F(X) \]
\[ \begin{cases} \text{begin } \text{statement list } Y, X, F(X) \end{cases} \]
\[ D_3(\text{statement list } Y, X, F(X)) = <\text{structured statement } > Y, X, F(X) \]
\[ \begin{cases} \text{begin } \text{statement list } Y, X, F(X) \end{cases} \]
\[ D_4(\text{statement } Y, X, F(X)) = \begin{cases} \text{begin } \text{statement list } Y, X, F(X) \end{cases} \]
\[ D_5(\text{statement list } > Y' <\text{statement list } > Y'', X, F(X)) = \begin{cases} \text{begin } \text{statement list } > Y'', X, F(X) \end{cases} \]
Transformations for Placing Request(s) for the Refinement X' of X.

The result is a procedure for request(s) X'.

E1(< procedure >Y,X',X)  
= < procedure heading > E2(< block > Y,X',X)  

   if G1

E2(< block > Y,X',X)  
= {begin E3(< statement list > Y,X',X ) end  

   if X \notin Y  

   if X \in Y and G2

E3(< statement list > Y,X',X)  
= {E4(< statement > Y,X',X)  

   if G3

   E4(< statement > Y,X',X) < statement list > Y''  

   if (X \notin Y'' or x \notin Y'')  

   and G4

   < statement > Y' E3(< statement list > Y'',X',X)  

   if X' \in Y'' and x \in Y''  

   and G4

E4(< statement > Y,X',X)  
= {request(X') < assignment > Y  

   if G5

   request(X') < procedure statement > Y  

   if G6

   request(X') < access control statement > Y  

   if G7

   E5(< structured statement > Y,X',X)  

   if G8

E5(< structured statement > Y,X',X)  
= {E2(< block > Y,X',X)  

   if G9

   E6(< conditional > Y,X',X)  

   if G10

   request(X') < parallel block > Y  

   if G11

E6(< conditional > Y,X',X)  
= {E6(if < expression > Y' then < block > Y'' else < block > Y'')  

   if G12

   request(X') if < expression > Y'  

   then < block > Y'' else < block > Y'')  

   if X \notin Y'' \cap Y''

   if < expression > Y' then E2(< block > Y'',X',X)  

   else E2(< block > Y'',X',X)  

   if X \in Y'' \cap Y''