Open predicate path expressions for distributed environments: notation, implementation, and extensions

Mark Roger Headington
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations 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 through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

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

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Headington, Mark Roger

OPEN PREDICATE PATH EXPRESSIONS FOR DISTRIBUTED ENVIRONMENTS: NOTATION, IMPLEMENTATION, AND EXTENSIONS

Iowa State University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

PH.D. 1984
Open predicate path expressions for distributed environments:

Notation, implementation, and extensions

by

Mark Roger Headington

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

1984
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Table of Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I. INTRODUCTION</strong></td>
<td></td>
</tr>
<tr>
<td>A. Overview of the Material in the Thesis</td>
<td>3</td>
</tr>
<tr>
<td>B. Description of the Target Environment and Resource Modules</td>
<td>4</td>
</tr>
<tr>
<td>C. Path Expressions</td>
<td>10</td>
</tr>
<tr>
<td>1. Regular path expressions (RPEs) and their variants</td>
<td>13</td>
</tr>
<tr>
<td>2. Open path expressions (OPEs)</td>
<td>15</td>
</tr>
<tr>
<td>3. Predicate path expressions (PPEs)</td>
<td>17</td>
</tr>
<tr>
<td>4. Other variants</td>
<td>19</td>
</tr>
<tr>
<td><strong>II. PROPOSAL FOR A NEW PATH NOTATION</strong></td>
<td>22</td>
</tr>
<tr>
<td>A. Desirable Characteristics of a Path Expression in the Target Environment</td>
<td>22</td>
</tr>
<tr>
<td>1. Environmental considerations</td>
<td>22</td>
</tr>
<tr>
<td>2. Human factors</td>
<td>24</td>
</tr>
<tr>
<td>B. A Comparison of OPEs and PPEs</td>
<td>26</td>
</tr>
<tr>
<td>1. A common semantics: P-V implementation</td>
<td>26</td>
</tr>
<tr>
<td>2. A subjective comparison of OPEs and PPEs</td>
<td>43</td>
</tr>
<tr>
<td>3. Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>C. A New Notation—The Open Predicate Path Expression</td>
<td>49</td>
</tr>
<tr>
<td>1. Syntax of OPPEs</td>
<td>50</td>
</tr>
<tr>
<td>2. Advantages of OPPEs</td>
<td>51</td>
</tr>
<tr>
<td><strong>III. AN IMPLEMENTATION SEMANTICS FOR OPEN PREDICATE PATH EXPRESSIONS—THE DATAFLOW GRAPH</strong></td>
<td>54</td>
</tr>
<tr>
<td>A. P-V Implementation Semantics for OPPEs</td>
<td>54</td>
</tr>
<tr>
<td>B. Synthesis of OPEs into Dataflow Graphs</td>
<td>56</td>
</tr>
<tr>
<td>1. Background</td>
<td>56</td>
</tr>
<tr>
<td>2. A synthesis scheme for OPEs</td>
<td>58</td>
</tr>
<tr>
<td>3. Examples</td>
<td>69</td>
</tr>
<tr>
<td>C. Synthesis of OPPEs into Dataflow Graphs</td>
<td>83</td>
</tr>
<tr>
<td>1. The pred-controller</td>
<td>83</td>
</tr>
<tr>
<td>2. Distributed event counters</td>
<td>84</td>
</tr>
<tr>
<td>3. A preliminary algorithm for generating and connecting PCs</td>
<td>86</td>
</tr>
<tr>
<td>4. Predicate interference</td>
<td>94</td>
</tr>
<tr>
<td>5. Final translation algorithm</td>
<td>108</td>
</tr>
<tr>
<td>6. Examples</td>
<td>126</td>
</tr>
<tr>
<td>7. Deadlock and OPPEs</td>
<td>148</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

The research presented in this thesis is a contribution to the investigation of highly structured, object-oriented approaches to the design of distributed computing applications for network computer systems. Although the definition of the term "object-oriented" is currently in a state of flux [Rentsch 1982, Cohen 1984], most views converge to at least the following characteristics:

- An "object" is a self-contained yet passive entity which has processing ability. An object may simply be a dataless system resource such as a compiler or it may be an instance of an abstract data type which encapsulates data and a set of operations on that data (in which case a process may access or manipulate this data only by requesting the execution of these specific operations).

- Message communication is the only means by which a process can access an object. Additionally, objects communicate with each other only by passing messages.

When such objects are implemented in distributed environments, it is natural to assume that they will execute concurrently and, whenever possible, independently of each other. However, when concurrent requests for access to shared resources arise, the arbitration of such requests may temporarily cause such concurrency to be restricted. In most object-oriented approaches, synchronization of requests for operations on an object's data is performed within the object itself rather than by a centralized arbiter. Much work has been and is being

In most proposed implementations, the specification of synchronization is embedded within the program code for the actual access of the object's data. In contrast, this thesis research has as a primary goal the separation of the two functions—synchronization and access—so that each can be specified as independently of the other as possible. It is envisioned that each of these two functions will be specified in its own language notation and encapsulated as a separate submodule internal to an object (hereafter called a "resource module"). Although this idea of a clear separation of synchronization from access is not new in centralized systems, it has received only recent attention for distributed systems and also dataflow systems [Allchin and McKendry 1983, Durrieu 1979, Jayaraman and Keller 1982, Kolstad 1983, Lauer et al. 1980, Oldehoeft and Jennings 1984].
A. Overview of the Material in the Thesis

The research presented in this thesis focuses on the synchronization submodule of a general resource module. In particular, a very-high-level notation for this submodule will be introduced, an implementation semantics for the notation will be described, and then extensions to the notation will be explored.

The remainder of Chapter I presents background material consisting of a more detailed description of resource modules in the target environment and a discussion of the "path expression" notation as a non-procedural language for specifying synchronization constraints. In Chapter II, a comparison is made of two existing forms of path expressions, and then a new path notation is introduced as a desirable candidate for a specification language for the synchronization submodule. Chapter III contains the bulk of the thesis research: an implementation semantics for the new specification language with an algorithm for the automatic synthesis of synchronization specifications into dataflow graphs. It is then shown how the dataflow graphs can be implemented by networks of communicating submodules to effect the desired synchronization constraints. Chapter IV contains a proposal for extending the new notation to provide additional language features judged to be useful for more complex synchronization problems such as the disk scheduler, the alarm clock, and distributed database update algorithms. Finally, Chapter V presents a summary of the work and offers suggestions for further research.
B. Description of the Target Environment and Resource Modules

In our target environment, a "resource module" is the basic building block for constructing a network system. A resource module at this stage of our investigation represents an instance of an abstract data type. As such, the resource module can logically be considered to contain data and a set of allowable operations on that data such that the data can only be accessed or modified by the execution (within the resource module) of these allowable operations. A resource module will totally encapsulate the data (hereafter called the "resource object") and will be in complete charge of its synchronization and access (reading or modifying the resource object). These two functions are performed by two distinct submodules—the synchronization submodule and the access-mechanism submodule—which are separately specified and encapsulated within the resource module. Figure I-1 depicts the general organizational structure of a resource module. As shown in the diagram, our current view is that the access-mechanism encloses the resource object itself along with the program code for the operations on the resource object.

Resource modules communicate with the environment only through receipt and transmission of messages. Hence, a process can only access or modify a resource object by sending an operation request message to the enclosing resource module; additionally, resource modules can communicate with each other only by passing messages. In order to depict this message flow and to reflect the view that resource modules might be implemented on highly parallel machines as well as on
conventional machines, the structure of a resource module in the target environment can be illustrated more specifically by means of a parallel program graph which is prevalent in dataflow computer research: the dataflow graph [Dennis 1974].

In dataflow machines, an instruction is enabled for execution as soon as all of its operands are available, and the value produced by the instruction becomes an operand for any further waiting instructions. Furthermore, there is no concept of updatable memory cells, so the execution of an instruction has no side effects. A dataflow program can thus be represented by a directed graph in which the nodes represent operators, and a node has one input arc for each operand and one output arc for each result. Because each operation is free of side effects, independent operations will naturally execute in parallel. Figure I-2 shows a dataflow graph for computing the value of the arithmetic
expression \( a \times b + c/d \). Note that the multiplication and division operators may execute in parallel because there is no data dependency between them and each operator waits only for the presence of its two operands.

![Dataflow graph for evaluation of \( a \times b + c/d \)](image)

**FIGURE I-2.** Dataflow graph for evaluation of \( a \times b + c/d \)

![Dataflow graph illustrating "merge" operator](image)

**FIGURE I-3.** Dataflow graph illustrating "merge" operator

Figure I-3 depicts a dataflow program which evaluates either \( a+c \) or \( b+c \) and contains a special operator denoted by a circled X: the "merge"
operator. This operator is special in that it is enabled for execution as soon as any one of its operands is available; the effect of its execution is simply that the input value is copied onto its output arc. If more than one input operand arrives simultaneously, the operator non-deterministically selects one of them to be output and then returns to its waiting state. Hence, in Figure I-3 if a and b arrive simultaneously, then either a followed by b or b followed by a will proceed to the addition operator.

Figure I-4 shows the structure of a resource module as a dataflow graph in which the nodes represent entities at a higher level than primitive operators—namely, program submodules. The resource module is driven by the arrival of messages requesting operations to be performed on its resource object. Each of the two submodules is activated by the presence of a token (i.e., message) on its primary input arc and the availability of its internal state (represented as a token on its other input arc). Each submodule consumes an incoming message and the value of its internal state and produces an outgoing message and the value of its new state. It should be noted that the internal state of the access-mechanism submodule consists of the actual resource object as well as any other local data which may be required for the submodule's execution.

A request message will progress through both of the synchronization and access phases. However, if the synchronization submodule determines that a request must be blocked so as not to violate the specified synchronization constraints, the request will be enqueued until such
Regarding opportunities for parallelism internal to the resource module, one can distinguish between inter-submodule and intra-submodule parallelism. In our model, inter-submodule parallelism is provided by the two stage pipelining of request messages. Intra-submodule parallelism depends upon how a given submodule is implemented, both in
terms of the support architecture and in terms of the degree to which a submodule is further divided into sub-submodules. With respect to the latter, it will be seen in Chapter III that the synchronization submodule can be implemented by distributing the synchronization state among communicating sub-submodules which naturally execute concurrently (except when the desired synchronization constraints dictate otherwise). With respect to the underlying support architecture, the implementation of a submodule on, say, a dataflow machine would allow for overlapped processing of successive request messages, depending on how soon the internal state of the submodule becomes available. For example, the access-mechanism submodule will typically respond to requests for read-type and write-type operations on its resource object. Write-type operations produce a changed resource state, so the dataflow model naturally enforces exclusive access to the resource object because a subsequent request must wait for the new (updated) resource state to become available. On the other hand, with a read-type operation the new resource state can become available immediately since it is the same as the old resource state; thus a subsequent read request may proceed concurrently with the first one.

It should be noted that the use of dataflow graphs in this thesis does not require dataflow support architectures; a local cluster or a network of conventional computers might serve equally well. The advantage of using dataflow graphs—particularly as an implementation semantics for our synchronization submodule specification language as seen in Chapter III—is that they serve as a highly parallel

The scope of this thesis research is primarily the synchronization submodule and its communication with the access-mechanism. The next section will provide a motivation for our choice of a language notation for this submodule, and then in Chapter II the presentation of the new work begins.

C. Path Expressions

Early approaches to the synchronization of concurrent accesses to shared data required the programmer to incorporate explicit instructions into the program to effect this synchronization: P and V operations on semaphores [Dijkstra 1968], wait and signal operations [Habermann 1972], and conditional critical regions scattered throughout user programs [Dijkstra 1968, Hoare 1972]. With the appearance of the notion of the abstract data type, in which the specification of a data structure and the operations permitted on it are encapsulated, came the concept of removing the programming of synchronization from user processes and placing it with the definition of the abstract data type. This led to textual grouping of conditional critical regions into the type definitions, yielding secretaries [Dijkstra 1971] and monitors [Brinch Hansen 1973, Hoare 1974]. The latter are procedure-oriented concepts in that the creator of the abstract data type programs the synchronization...
explicitly by using the low-level synchronization primitives "signal" and "wait". In contrast, a specification-oriented approach to synchronization arose in the form of "path expressions" [Campbell and Habermann 1974]. Here, the specification of synchronization is also located entirely within the type definition, but the creator specifies the synchronization constraints without having to write program code which manipulates synchronization primitives. That is, one specifies non-procedurally what the synchronization should be, not how to enforce it. This non-procedural approach is one reason why path expressions are attractive in the context of our target environment.¹

A path expression is a declarative statement of the allowable order of execution of operations on a data object. The following is a simple path expression for specifying constraints on the execution of operations A, B, C, and D:

```
path A;(B,C);D end
```

In this example, the semicolon denotes sequencing and the comma denotes selection; also, the executions of all operations named in the path expression are mutually exclusive. Thus, this example specifies that operation A must execute before any of B, C, and D can execute; that once an A has terminated, either a B or C—but not both—can execute; and that after a B or C has terminated, a D may begin. After a D has terminated, the path expression is considered to be complete and the

¹ The interested reader will find in [Andrews and Schneider 1983] a description of the evolution of synchronization mechanisms as well as a survey of notations for concurrent programming.
entire sequence may then be repeated. Thus, if an A is executing and another request for A arrives, the latter will be blocked until the first A completes, then a B or C completes, and then a D completes.

What distinguishes path expressions from other synchronization primitives mentioned earlier is that their use does not require details of how the synchronization is to be implemented; thus, the burden of implementing any given synchronization is removed from the programmer. In our thesis, the use of path expressions as a specification language was explored for several reasons:

- Path expressions represent a non-procedural, very high level approach to specifying synchronization and concurrency. This is clearly compatible with an environment which may use highly parallel support architectures which conform to non-procedural, functional languages.
- Path expressions statically provide a clear separation of synchronization from operations on a data object. This corresponds closely with our target environment which presupposes the physical separation of the synchronization and access-mechanism submodules.
- Path expressions were designed from the outset to be an integral part of the definition of an abstract data type and are thus appropriate to our view of encapsulation and local control of resources.
- Path expressions in a particular form—"open path expressions"—will be seen to have attractive semantics for
highly parallel systems and have been shown to lend themselves
to automated translation into networks of communicating
submodules in a dataflow environment [Oldehoeft and Jennings
1984].

1. Regular path expressions (RPEs) and their variants

In Campbell's thesis [Campbell 1977] is presented a path notation
consisting of (limited) regular expressions composed of operator names,
parentheses, sequencing (";") and selection ("","") operators, and the
Kleene star ("*"). Campbell uses regular expressions to facilitate
discussion of theoretical properties of path expressions and their
implementation because of the well-known equivalence of regular
expressions and deterministic finite automata. The most widely known
version of path expressions, however, is the "original" notation of
Campbell and Habermann [Campbell and Habermann 1974]: regular
expressions without the internal "*" and with the inclusion of the
operator "{;}" for simultaneous execution. The latter is used, for
example, in the path expression

\[ \text{path } A,\{B\} \text{ end} \]

which means that either one A may execute or a burst of Bs may execute.
By the latter is meant that, once a B has been granted permission to
begin executing, any number of subsequent Bs may execute concurrently
until the last one has finished—at that point, the burst (and hence
-------------

\[ \text{path } A;B \text{ end} \]

denotes the same execution sequence as the regular expression \((A;B)^*\).
this portion of the path expression) is considered terminated. Although path expressions using this brace operator are no longer regular expressions, we will use the term "regular path expression" (RPE) to refer to such variants to distinguish them as a class from other major classes of path expressions.

With this original notation of Campbell and Habermann, each reader and writer of a data object will have exclusive access if the synchronization is specified as \text{path READ,WRITE end}. For the weak readers' priority solution of the readers-writers problem [Courtois et al. 1971] in which concurrent reading is allowed (yet waiting readers do not necessarily inhibit writing), Campbell and Habermann's solution is given as \text{path \{READ\},WRITE end}. However, their solution to the writers' priority version—in which any waiting writers inhibit reading—is far more complex. Their solution requires three separate path expressions and the definition of additional pseudo-operations. In the following, a user process requests the operation READ or WRITE, and the procedures \text{DO_READ} and \text{DO_WRITE} perform the actual reading and writing:

\begin{verbatim}
path READATTEMPT end
\text{path REQUESTREAD,\{REQUESTWRITE\} end }
\text{path \{OPENREAD;DO_READ\},DO_WRITE end }
\end{verbatim}

\text{where}
\begin{verbatim}
READATTEMPT = begin REQUESTREAD end
REQUESTREAD = begin OPENREAD end
REQUESTWRITE = begin DO_WRITE end
READ = begin READATTEMPT;DO_READ end
WRITE = begin REQUESTWRITE end
\end{verbatim}

Because their notation disallows multiple occurrences of an operation name within a path expression and because it is not powerful enough to
express certain synchronization requirements, "dummy" synchronization procedures [Bloom 1979] such as \textsc{readattempt}, \textsc{requestread}, etc. and auxiliary path expressions are required in the above example.

In order to make RPEs more powerful, numerous additions and alterations to the original path notation have appeared in the literature: multiple paths, parallel operators, connected paths, conditional path elements, priority operators, and numeric path elements [Campbell 1977, Flon and Habermann 1976, Habermann 1975, Lauer and Campbell 1975, Lauer et al. 1980]. We continue to classify these as variants of RPEs, however, to distinguish them from two other fundamentally different versions: open path expressions and predicate path expressions. These will be discussed separately in the next two sections.

2. Open path expressions (OPEs)

This path notation—originally proposed in Campbell's thesis [Campbell 1977] and incorporated into a Path Pascal language [Campbell and Miller 1978]—continues to use ";" as a sequencing mechanism and "]" (instead of "{") for specifying (unbounded) simultaneous execution. However, OPEs have semantics fundamentally opposite from that of RPEs with respect to concurrency. With RPEs, mutual exclusion of the operations named in the path expression is implicit; to specify concurrency of operations, one must explicitly prescribe this. With OPEs, on the other hand, unrestricted concurrency of the operations is implicit, and one must use explicit notations to enforce serialization and mutual exclusion. For example, the RPE path A end specifies that
only one execution of A is permitted at a time, whereas the OPE path A end does not impose mutual exclusion; instead, any number of executions of A may proceed concurrently. Also, the comma in OPEs does not denote selection—it is only a distributive mechanism and imposes no restriction on either the order or the number of concurrent invocations of the operations. Thus, while the RPE path A,B end signifies that only one of A or B may execute at a time, the OPE path A,B end provides no synchronization at all—any number of processes can invoke A and B concurrently. Furthermore, the semantics of the sequencing operator (";") now become subtly different. For example, the RPE path A;B end specifies that one A is to be followed by a B, then one A is to be followed by a B, and so on. But the OPE path A;B end specifies that any number of As and Bs may execute concurrently, provided that each B must be preceded by the termination of an A. Hence, for example, 50 As could execute concurrently and terminate and then 50 (but not 51) Bs could execute concurrently.

For limiting concurrency, OPEs provide a finite bounded parallelism construct n:(subexpr) which restricts to n the number of simultaneous executions of the operations named in the subexpression. For example, the OPE path 1:(A,B) end specifies the mutual exclusion of A and B and means exactly the same as the RPE path A,B end. As another example, the OPE path 2:(1:(A),1:(B)) end indicates that only one A may execute at a time, that only one B may execute at a time, and that at most two processes can cause As and Bs to execute (and thus one A and one B may execute concurrently). As a last example, the OPE path 5:(A;B) end
specifies that each $B$ may only start when at least one $A$ has completed and that up to five $A$s may execute before a $B$ is started.

OPEs have been shown to be useful enough to specify synchronization in a complete operating system: PathOS [McKendry et al. 1980]. Additionally, the notation is particularly attractive for our target environment because of the parallel semantics. On the negative side, however, is the disadvantage—shared with most of the RPE notations—that some problems require the introduction of contrived synchronization procedures whose only purpose is to allow further synchronization to be imposed. For example, for the writers' priority problem we have derived the following OPE solution from Campbell and Habermann's RPE solution presented in the previous section:

\[
\text{path 1: (READATTEMPT), 1: (READATTEMPT, WRITE),}
\text{1: (READATTEMPT; READ, WRITE) end}
\]

Here, we have had to use the synchronizing procedure READATTEMPT and to use a multi-level path expression to express the solution.$^3$

3. **Predicate path expressions (PPEs)**

This path notation [Andler 1979] turns out to be more powerful than RPEs or OPEs for expressing certain synchronization constraints such as the example mentioned in the preceding paragraph. The notation shares with RPEs the implicit mutual exclusion of operations named in the path expression and uses the operators "+" (instead of ",") for selection,

$^3$ In an OPE, an operation name may be repeated. In this case, the synchronization constraints for each occurrence of the name are applied from left to right as they occur in the expression.
";" for sequencing, "*" for repetition, and "{" for unbounded simultaneous execution. However, the most distinctive feature of this notation is the use of predicates which may be attached to any path element. In Andler's paper, the predicate is a boolean expression using implicit counters req(X), auth(X), and term(X), where X is an operation name. These predefined counters are non-negative, monotonically increasing integers whose values are assumed to be maintained by the underlying support system and represent the total number of requests for, authorizations of, and terminations of operation X, respectively.

Some interesting work has appeared which uses such implicit counters for synchronization [Andler et al. 1978, Ford 1978, Gerber 1977, Gerber 1978, Robert and Verjus 1977, Schmid 1976], and the specific form of predicate that Andler explores is that of a boolean function of linear relational expressions in which the variables are the counters req(X), auth(X), and term(X), and the coefficients are integer constants.

For example, Andler's solution to the writers' priority variant of the readers-writers problem is as follows:

```
def ww = req(WRITE) - auth(WRITE)
path ( {READ[ww=0]} + WRITE)*
```

What makes this example particularly striking is the brevity and clarity.

--------

4 The PPE notation also uses a collateral execution operator (","), which has cobegin-coend semantics. Its use will be discussed in Chapter II.

5 Andler uses act(X)—for "activations of X"—rather than auth(X). We use auth(X) because our implementation of predicates, to be discussed in Chapter III, distinguishes between authorizations and activations.
of the solution in contrast with the original RPE solution we displayed in Section I.C.1. In several other variants of the readers-writers problem presented by Andler, the expressive power of the PPE notation emerges. However, PPEs are still based on the semantics of mutual exclusion rather than the OPE notation's assumption of concurrency—a disadvantage in our target environment.

4. Other variants

In [van den Bos et al. 1981] appears a description of a model for communicating parallel processes called "input tools". Originally designed in the context of high-level graphics device handlers, these input tools use a path-like notation known as an "input expression" to match incoming input tokens. If one of the patterns specified by the input expression matches the sequence of input tokens, the tool body will be executed. Thus, whereas path expressions specify valid orderings of operations on a data object, input expressions indicate valid input sequences which trigger the execution of program modules. The input expressions are essentially regular expressions plus a parallel operator and conditional elements. However, the latter allow the testing of arbitrary variables local to the input tool and thus do not provide as clear a separation of synchronization from access as our research desires.

In [Kieburtz and Silberschatz 1983], "access-right expressions" are proposed. These are based on regular expressions and use sequencing, selection, and repetition operators. Although their appearance is similar to path expressions, their intended use is in the domain of
protection rather than synchronization. Access-right expressions specify valid sequences of operations that a given process can execute, and, as such, the focus is on the external rather than the internal consistency of the resource.

Another application of path expressions in a distinctly different domain—that of program debugging—is described in [Bruegge and Hibbard 1983]. The authors extend PPEs to allow arbitrary program variables in predicates and to allow the expression of execution sequences that are finer grained than entire operations—namely, single statements or groups of statements in a computation. By using these "generalized path expressions", any deviation of a program's observed execution sequence from the specified execution sequence can be detected, and the user may specify what actions are to be taken in such a case.

Finally, we mention an expression-based synchronization notation appearing in [Jayaraman and Keller 1982] which is used in an environment similar to ours. The authors' "resource expressions" are intended to constrain the parallel evaluation of functions used in resource operations in a distributed, applicative framework. The notation includes constructs for sequencing, selection, repetition, and simultaneous execution with each of the latter two mechanisms having two distinct forms. Formal semantics of their notation are provided which are fundamentally different from other path notations, and an implementation scheme is demonstrated for translating resource expressions into a set of queuing primitives for a demand-driven dataflow model. Although their notation is primarily based on regular
expressions, it appears that their semantics make the notation more powerful than RPEs. However, the paper indicates the possibility of making it even more powerful by the inclusion of predicates (as in PPEs).

The reader interested in further descriptions and comparisons of path expressions (and other expression-based specification languages) will find a useful synopsis in [Shaw 1980]. We now proceed, in Chapter II, to examine those features of existing path notations which are most attractive for our target environment and to propose a new path notation based on this examination.
II. PROPOSAL FOR A NEW PATH NOTATION

A. Desirable Characteristics of a Path Expression in the Target Environment

Given the proposed computing environment presented in Chapter I and the decision to use some variant of path expression to express non-procedurally the synchronization constraints upon accesses to resource objects, we now focus on which variants are most suited to (a) the environment and (b) the programmer (in a philosophical and/or stylistic sense).

1. Environmental considerations

Although not an exhaustive list of desirable characteristics, the following three factors are important in the choice of notation for use in a distributed, potentially highly parallel environment: parallel semantics, separability of the resource object from synchronization specifications, and ease of translation into asynchronous submodules using a message passing paradigm.

With respect to parallel semantics, all of the path notations surveyed earlier—except OPEs—are based on sequential semantics for von Neumann systems: exclusive access to a given path expression is the rule, whereas any desired parallelism must be explicitly specified. OPEs, on the other hand, have highly appropriate semantics for a parallel environment: unrestricted concurrency of operations is implicit, whereas mutual exclusion and serialization must be explicitly specified.
With respect to separability of the unsynchronized resource object from specifications of synchronization constraints, most of the notations purport to meet this goal intrinsically. That is, this separation is a primary reason that path expressions were developed in the first place. However, since several RPE-based notations were found not to be powerful enough to express certain complex synchronization and scheduling problems, either "dummy" synchronizing procedures (those whose only function is to allow further synchronization to be imposed) have been required or additional language features have been proposed (which have the effect of weakening this separation). An example of the latter is a proposal in Campbell's thesis [Campbell 1977] for adding to each element of a selection a boolean guard, the variables of which could be changed programmatically by the operations named in the path expression. As Campbell points out, this feature would tend to blur the distinction between resource implementation and synchronization specification (as does the use of synchronizing procedures).

With respect to the translation of path expressions, there exist at least four schemes for such translation. Campbell and Habermann's original paper [Campbell and Habermann 1974] gives transformation rules for their path expressions which generate the insertion of semaphores into the prologues and epilogues of the named operations, each of which is assumed to be a sequential procedure or process. Campbell's thesis [Campbell 1977] also presents a reduction algorithm for implementing OPEs by embedding semaphores in the prologues and epilogues of the sequential procedures named in the path expressions. Andler [Andler
1979] describes an implementation of PPEs and data abstraction using Algol 68, again with sequential semantics assumed. Lastly, [Oldehoeft and Jennings 1984] presents a scheme for the syntax-directed translation of OPEs into networks of communicating submodules in a dataflow environment. Because of the parallel nature of these submodules (as well as the parallel semantics of the path notation itself), the OPE notation is an attractive candidate for use in the target environment.

2. Human factors

Two important characteristics of a notation with respect to the programmer are ease of use and (in our target environment) a high degree of expressive power. In [Bloom 1979], Bloom relates these two factors to the notions of exclusion constraints (i.e., the enforcement of mutual exclusion) and priority constraints (i.e., the scheduling of access based on priorities) in essentially the following way: a synchronization mechanism has a high degree of expressive power if it provides straightforward methods for expressing priority and exclusion constraints, and is easy to use if it supports constraint independence (so that complex synchronization problems can be decomposed into individual constraints which can be implemented independently).

Campbell, in his thesis, also addresses the "power" of a path notation by introducing two measures: declarative power and scheduling power. A notation is relatively higher in declarative power the more it meets the goal of removing synchronization from the program text and expressing it statically; conversely, the lower it is in declarative power, the more frequent will be the need for synchronizing procedures.
A notation which is high in scheduling power will include an implicit scheduler to resolve potential conflicts between alternate sets of operations permitted by a specification; conversely, the lower it is in scheduling power, the more necessary it will be to include explicit programming to effect the scheduling of conflicting operations. Near the end of Campbell's thesis, he proposes several alternative path notations which yield more declarative power (although he states that little attempt has been made to actually measure the increase). One of these notations is the OPE, and others involve the use of predicates on selection and repetition. Andler [Andler 1979] takes the notion of predicates much further (and from a fundamentally different point of view) and proposes, in PPEs, a path notation which, in Bloom's opinion, comes closest to satisfying Bloom's requirements.

While both Bloom's and Campbell's proposals for quantitative and qualitative measures of the power and ease of use of synchronization mechanisms are useful, it is not in the scope of this research to further explore such objectification. Rather, their work has helped to confirm more subjective reasons for selecting OPEs and PPEs as best suited for the target environment—OPEs for their parallel semantics, and PPEs for their simplicity and expressive/declarative/scheduling power. These two notations will be compared in more detail in Section II.B with respect to the characteristics discussed in this section.
B. A Comparison of OPEs and PPEs

1. A common semantics: P-V implementation

In order to describe the meanings of path expressions using OPEs and PPEs and, more importantly, to show equivalent and non-equivalent expressions in the two notations, it is necessary to use a common semantics for both.

Shaw delineates three classes of methods used for defining the semantics of various path notations [Shaw 1980]: informal implementation-based semantics, Petri net transformations, and other, generally formal, methods. The first class includes the reduction algorithms for OPEs [Campbell 1977, Campbell and Miller 1978] and the original restricted RPEs [Campbell and Habermann 1974], leading to the insertion of P and V operations into the prologues and epilogues of the operations named in the path expressions. The second and third classes generally deal with variations of RPEs, although the third class includes Andler's definition of PPE semantics in terms of partial orderings induced by transformations of PPEs into non-deterministic programs.

Because the semantics of OPEs and PPEs do not have similar definitions in the literature, a common definition will now be presented—namely, P-V semantics (i.e., transformation rules for embedding P and V operations in the prologues and epilogues of named operations).

a. Syntax of OPEs The following BNF syntax of OPEs is a blend of those found in [Campbell and Miller 1978] and [Campbell and Kolstad
1980b]. In addition, we have substituted "{}" for their "[]" as the unbounded simultaneous execution operator.

\[
\text{<OPE> ::= path <list> end}
\]

\[
\text{<list> ::= <sequence> | <sequence> , <list>}
\]

\[
\text{<sequence> ::= <item> | <item> ; <sequence>}
\]

\[
\text{<item> ::= <unsigned integer> : ( <list> )}
\]

\[
\text{| { <list> }}
\]

\[
\text{| ( <list> )}
\]

\[
\text{| <operation_id>}
\]

The next-to-last production is meant to allow the normal use of parentheses for clarity and/or alteration of the default precedence. Furthermore, if an operation name is repeated within a path, the synchronization constraints for each occurrence of the name are applied from left to right as they occur in the expression.

b. Transformation rules for OPEs The following recursive algorithm [Campbell and Miller 1978] will translate OPEs into the P-V implementation. In the algorithm, each of L and R represents a previously generated synchronization primitive (or sequence of primitives) on the left and right of the subexpression. Each of L and R may be null.

The following transformation rules are applied in the order given by a left-to-right parse of the OPE according to the above production rules:

1. Replace path list end by null list null.
2. (The comma—a distributive mechanism)

Replace L sequence,list R by L sequence R
and L list R

3. (The semicolon—a sequencing mechanism)

Replace L item;sequence R by L item V(s1)
and P(s1) sequence R

with semaphore s1 initialized to 0.

4. (Resource restriction)

Replace L n:(list) R by P(s2) L list R V(s2)

with semaphore s2 initialized to n.

5. (Resource derestiction)

Replace L {list} R by P(c,s,L) list VV(c,s,R)

where PP and VV are defined as:

procedure PP (counter c; semaphore s; procedure synch);
begin
  P(s);
  c := c + 1;
  if c=1 then synch;
  V(s)
end;

procedure VV (counter c; semaphore s; procedure synch);
begin
  P(s);
  c := c - 1;
  if c=0 then synch;
  V(s)
end

and initialize counter c to 0 and semaphore s to 1.

6. For embedding synchronization primitives in the prologue and epilogue of a named operation, the original algorithm assumed that an operation name could not appear more than once in a given OPE.
The transformation rule was given as:

6(a). Replace $L\text{ operation_id } R$ by $L\text{ operation body } R$

The restriction against multiple appearances of an operation name was subsequently removed in favor of the previously mentioned left-to-right constraint rule [Campbell and Kolstad 1980a]. Hence, we include:

6(b). If operation_id has appeared and been transformed into $L'\text{ operation body } R'$ prior to the current appearance of operation_id, then replace the current instance of $L\text{ operation_id } R$ by $L'\text{ L operation body } R R'$

Translation trees corresponding to the reduction algorithm above are described in [Campbell and Habermann 1974] and [Campbell 1977]. Figure II-1 shows the translation tree for the OPE path 2:(A;B), 1:(A,{C}) end and Figure II-2 shows, in sequential language form, the resulting set of procedure prologues and epilogues created by Step 6 of the algorithm. Note in Figure II-1 how the rightmost occurrence of operation A inherits the enclosing primitives $P(s1)$ and $V(s2)$ from the leftmost occurrence of A.

c. Syntax of (restricted) PPEs PPEs [Andler 1979] are essentially RPEs (using regular operators "+" for selection, ";" for
FIGURE II-1. P-V translation tree for OPE
path 2: (A; B), 1: (A, [C]) end

semaphore s1, s2, s3, s4; counter c;
s1 := 2; s2 := 0; s3 := 1; s4 := 1; c := 0;
procedure A: begin P(s1); P(s3); (body of A); V(s3); V(s2) end;
procedure B: begin P(s2); (body of B); V(s1) end;
procedure C: begin PP(c, s4, P(s3)); (body of C); VV(c, s4, V(s3)) end

FIGURE II-2. Sequential program P-V implementation of OPE
path 2: (A; B), 1: (A, [C]) end
sequencing, and "*" for repetition) augmented by parallel operators ("{}" for simultaneous execution and binary operator ",," for collateral execution) and predicates on path elements. These augmentations yield non-regular expressions. The unrestricted use of "*" and ",," is problematical with respect to P-V implementation [Campbell 1977], and such use will be discussed later.

The following BNF syntax is thus for restricted PPEs—those without the ",," operator and having "*" only for repetition of the entire path expression.

\[
\text{<restricted_PPE> ::= path ( <list> )*}
\]

\[
\text{<list> ::= <sequence> | <sequence> + <list>}
\]

\[
\text{<sequence> ::= <item> | <item> ; <sequence>}
\]

\[
\text{<item> ::= \{ <list> \}}
\]

\[
| \text{<item> [ <predicate> ]}
\]

\[
| ( <list> )
\]

| <operation_id>

For the time being, the form of <predicate> will be unspecified beyond that it is a boolean function of linear relational expressions in integer constants and implicit counters req(X), auth(X), and term(X) denoting the number of requests for, authorizations of, and terminations of operation X.

d. Transformation rules for (restricted) PPEs As an extension to Campbell's reduction algorithm for RPEs, we have derived the following recursive scheme to translate restricted PPEs into the P-V
implementation. Again, each of L and R may be null or a sequence of previously generated synchronization primitives.

The following transformation rules are applied in the order given by a parse of the PPE according to the above production rules:

1. Replace \texttt{path (list)*} by \texttt{P(s1) list V(s1)}
   with semaphore \texttt{s1} initialized to 1.

2. (The plus sign—a selection mechanism)
   Replace \texttt{L sequence+list R} by \texttt{L sequence R}
   and \texttt{L list R}

3. (The semicolon—a sequencing mechanism)
   Replace \texttt{L item;sequence R} by \texttt{L item V(s2)}
   and \texttt{P(s2) sequence R}
   with semaphore \texttt{s2} initialized to 0.

4. (Braces—a simultaneous execution mechanism)
   Replace \texttt{L \{list\} R} by \texttt{PP(c,s,L) list VV(c,s,R)}
   with counter \texttt{c} initialized to 0 and semaphore \texttt{s}
   initialized to 1.

   Primitives \texttt{PP} and \texttt{VV} are as defined in the transformation rules for OPEs.

5. (The predicate)
   Replace
   \texttt{L item[predicate] R} by \texttt{W(predicate) L item R}
   where we introduce a new primitive \texttt{W ("when")} which may be thought of as an operation on a conditional semaphore: if the specified predicate is true, then the request may proceed
through the synchronization constraints specified by \( L \) item \( R \); if false, the request is blocked until the predicate becomes true. As with a \( P \) operation on a semaphore, an implicit queuing of blocked requests takes place, and the awakening of one of the blocked requests is done in an unspecified but "fair" manner. A discussion of the realization of the \( W \) operator is presented in Chapter III.

6. Replace \( L \) operation id \( R \) by \( L \) operation body \( R \)

Figure II-3 shows the translation tree for the PPE

\[
def \text{wb} = \text{req}(B) - \text{auth}(B)
\]

\[
\text{path} ([A[\text{wb}=0]] + B)^*
\]

Here, "\( \text{wb} \)" (i.e., the number of waiting requests for operation \( B \)) is defined in terms of the implicit counters representing the number of requests for and authorizations of \( B \). Figure II-4 then shows, in sequential language form, the resulting procedure prologues and epilogues created by Step 6 of the algorithm.

e. Equivalent OPEs and PPEs Given the above semantics and some further observations, it is possible to give examples of OPEs and PPEs which yield equivalent semantics. In the discussion which follows, the definitions of equivalence and the arguments supporting specific examples are of an informal nature. It is presumed that the arguments could be formalized, but we have chosen not to proceed further in this direction—rather, a new semantics will be introduced and built upon later.
FIGURE II-3. P-V translation tree for PPE
\[
\text{def } \text{wb} = \text{req}(B) - \text{auth}(B) \\
\text{path } ([\text{wb}=0] \cdot \text{VV}(c,s2,V(s1))) \\
\text{W(wb=0) PP}(c,s2,P(s1)) \ A \ \text{VV}(c,s2,V(s1)) \\
\]

semaphore s1, s2; counter c;
\[
\text{s1} := 1; \quad s2 := 1; \quad c := 0; \\
\text{procedure A: begin } \text{W}(\text{wb}=0) \ \text{PP}(c,s2,P(s1)); \ (\text{body of A}); \\
\text{VV}(c,s2,V(s1)) \ \text{end}; \\
\text{procedure B: begin } P(s1); \ (\text{body of B}); \ V(s1) \ \text{end} \\
\]

FIGURE II-4. Sequential program P-V implementation of PPE
\[
\text{def } \text{wb} = \text{req}(B) - \text{auth}(B) \\
\text{path } ([\text{wb}=0] \ + \ B)* \\
\]
Definition: A one-level path expression is a path expression in which each operation name appears only once.

All PPEs are one-level path expressions, but OPEs may be multilevel expressions in that an operation name may be repeated in several subexpressions.

Definition: Two path expressions are S-equivalent (written $S==$) if and only if they allow the same sequences of operations.

Definition: Two one-level path expressions are PV-equivalent if and only if the leaves on their P-V translation trees are identical (apart from a renaming of semaphores and without regard to their left-to-right ordering on the frontiers of the trees).

It seems intuitively reasonable to make the following claim: that PV-equivalence implies S-equivalence. That is, since two PV-equivalent one-level path expressions have identical implementation semantics, they must allow the same sequences of operations. However, the converse is not true as can be seen in several examples which follow.

The following examples are presented so that the reader may see how various operators or combinations of operators in one path notation may be "simulated" in the other notation. Furthermore, Examples 1 through 3 show OPEs and PPEs which are PV-equivalent and hence S-equivalent; on the other hand, Examples 4 through 9 demonstrate that an OPE and a PPE can be S-equivalent yet not be PV-equivalent.

Example 1. OPE path 1:(A) end $S==$ PPE path (A)*.

The single leaf in the translation tree for the PPE is $P(s_1) \ A \ V(s_1)$ with $s_1$ initially 1. This leaf is the same as for the given OPE. Thus,
the two path expressions are PV-equivalent, hence S-equivalent.

Example 2. OPE path 1:(A,B) end S== PPE path (A+B)*.
The two leaves in the translation tree for the PPE are P(s1) A V(s1)
and P(s1) B V(s1) with s1 initially 1. These leaves are the same as
for the given OPE. Thus, the two path expressions are PV-equivalent,

Example 3. OPE path 1:(A;B) end S== PPE path (A;B)*.
The two leaves in the translation tree for the PPE are P(s1) A V(s2)
and P(s2) B V(s1) with s1 and s2 initially 1 and 0, respectively.
These leaves are the same as for the given OPE. Thus, the two path
expressions are PV-equivalent, hence S-equivalent.

Example 4. OPE path A end S== PPE path ({A})*.
For the OPE, the single leaf in the translation tree is (trivially) A.
The single leaf in the translation tree for the PPE is
PP(c,s2,P(s1)) A VV(c,s2,V(s1)) with each of s1 and s2 initially 1.
This leaf is the same as for the OPE path 1:({A}) end; hence, the PPE
is PV-equivalent (and thus S-equivalent) to the OPE path 1:({A}) end.
However, we argue that this latter OPE allows the same sequences of
operations as the OPE path A end, the only difference between the two
being that of a time delay due to implementation overhead. The OPE
path A end imposes no synchronization whatsoever on requests for
operation A. The OPE path 1:({A}) end requires the first of a series
of requests for A to do a PP on s2 and then a P on s1 before being
authorized; thereafter, further requests for A may proceed in parallel.
Hence, although requests for A will presumably be slowed down by performing PP operations on s2 and P operations on s1 (whose value, in this case, will always be 1 when such a P is performed), there are absolutely no constraints on the concurrency of As.

Thus, \textbf{PPE path} \((\{A\})^* S == \textbf{OPE path} 1:(\{A\}) end S == \textbf{OPE path} A end} \ (even though the latter is not PV-equivalent to the PPE).

\textbf{Example 5. OPE path} A,B end S == \textbf{PPE path} (\{A+B\})^*.

The two leaves in the translation tree for the OPE are (trivially) A and B. The two leaves in the translation tree for the PPE are \(PP(c,s2,P(s1)) A VV(c,s2,V(s1))\) and \(PP(c,s2,P(s1)) B VV(c,s2,V(s1))\) with each of s1 and s2 initially 1. These leaves are the same as for the OPE \(1:(\{A,B\}) end\), yielding PV-equivalence. But, by the same argument as in Example 4, the OPE \(1:(\{A,B\}) end\) is S-equivalent to the OPE \(A,B end\) (even though the latter is not PV-equivalent to the PPE).

\textbf{Example 6. OPE path} n:(A) end S == \textbf{PPE path} (\{A[auth(A)-term(A)<n]\})^*.

The translation tree for the PPE yields a single leaf \(W(auth(A)-term(A)<n) PP(c,s2,P(s1)) A VV(c,s2,V(s1))\) with each of s1 and s2 initially 1. We argue that this leaf can be replaced by \(P(s3) PP(c,s2,P(s1)) A VV(c,s2,V(s1)) V(s3)\) with counting semaphore s3 initialized to n—the implicit counter auth(a) reflects the number of requests that have successfully passed the P(s3), and the counter term(a) reflects the number of times that V(s3) has been
executed. Hence, if s3 is initialized to n, then a request for A will be allowed past the P(s3) operation when and only when it would be allowed past the W(auth(A)-term(A)<n) operation. Finally, the leaf P(s3)PP(c,s2,P(s1)) A VV(c,s2,V(s1))V(s3) corresponds to the single leaf in the translation tree for OPE path 1:({ n:(A) }) end, yielding PV-equivalence. But by the argument in Example 4, this OPE is S-equivalent to the OPE path n:(A) end (even though the latter is not PV-equivalent to the PPE).

Example 7. OPE path n:(A,B) end S==

PPE path (({(A+B)[auth(A)-term(A)+auth(B)-term(B)<n]})*.

The translation tree for the PPE yields the two leaves

W(auth(A)-term(A)+auth(B)-term(B)<n) PP(c,s2,P(s1)) A VV(c,s2,V(s1))

and

W(auth(A)-term(A)+auth(B)-term(B)<n) PP(c,s2,P(s1)) B VV(c,s2,V(s1))

with each of s1 and s2 initially 1. As we did in Example 6, we argue that the positive terms in the W predicate correspond to P operations on a counting semaphore initialized to n, and the negative terms correspond to V operations on the same semaphore. So the two leaves can be replaced by

P(s3) PP(c,s2,P(s1)) A VV(c,s2,V(s1)) V(s3)

and

P(s3) PP(c,s2,P(s1)) B VV(c,s2,V(s1)) V(s3)

with s3 initialized to n. These two leaves correspond to those in the translation tree for the OPE path 1:({ n:(A,B) }) end, which is then S-equivalent, by the argument in Example 4, to the OPE path n:(A,B).end (even though the latter is not PV-equivalent to the PPE).
**Example 8.** OPE path n:(A;B) end S==

PPE path ([A[auth(A)-term(B)<n]; b])*.

The translation tree for the PPE yields the two leaves

\[ W(auth(A)-term(B)<n) \quad PP(c,s2,P(s1)) \quad A \quad V(s3) \]

and

\[ P(s3) \quad B \quad VV(c,s2,V(s1)) \]

with s1, s2, and s3 initially 1, 1, and 0, respectively. By the same argument as in Examples 6 and 7, the W operation can be replaced by suitable operations on a counting semaphore s4, initialized to n. The two leaves above are then replaced by

\[ P(s4) \quad PP(c,s2,P(s1)) \quad A \quad V(s3) \]

and

\[ P(s3) \quad B \quad VV(c,s2,V(s1)) \quad V(s4) \]

These two leaves correspond to those in the translation tree for the OPE path 1:({ n:(A;B) }) end, which is then S-equivalent, by the argument in Example 4, to the OPE path n:(A;B) end (even though the latter is not PV-equivalent to the PPE).

**Example 9.** OPE path n:(1:(A);1:(B)) end S==

PPE path ([A[auth(A)-term(A)<l]; auth(A)-term(B)<n];

B[auth(B)-term(B)<l]])*.

The translation tree for the PPE yields the two leaves

\[ W(auth(A)-term(A)<1) \quad W(auth(A)-term(B)<n) \quad PP(c,s2,P(s1)) \quad A \quad V(s3) \]

and

\[ W(auth(B)-term(B)<1) \quad P(s3) \quad B \quad VV(c,s2,V(s1)) \]

with s1, s2, and s3 initially 1, 1, and 0, respectively. By the same argument as in Examples 6, 7, and 8, the first W operation for A can be replaced by enclosing P and V operations on counting semaphore s4,
initialized to 1; the second W operation for A can be replaced by P and V operations (split between A and B) on counting semaphore \(s_5\), initialized to \(n\); and the W operation for B can be replaced by enclosing P and V operations on counting semaphore \(s_6\), initialized to 1. The two leaves above are then replaced by:

\[
P(s_4) P(s_5) PP(c,s_2,P(s_1)) A V(s_3) V(s_4)
\]

and

\[
P(s_6) P(s_3) B VV(c,s_2,V(s_1)) V(s_5) V(s_6).
\]

These two leaves correspond to those in the translation tree for the OPE \(\text{path } 1:(\{ n:(1:(A);1:(B)) \})\) end, which is then S-equivalent, by the argument in Example 4, to the OPE \(\text{path } n:(1:(A);1:(B))\) end (even though the latter is not PV-equivalent to the PPE).

f. Difficulties in determining S-equivalence

The preceding examples of S-equivalent OPEs and PPEs were quite straightforward in that: (a) the PPEs shown were either unpredicated or used predicates which corresponded directly to P and V operations on semaphores, (b) the OPEs shown did not use synchronizing procedures and were one-level path expressions (i.e., did not use repeated operation names with left-to-right synchronization precedence), and (c) the PPEs were restricted so that the collateral execution operator ""," was absent and the repetition operator "*" could appear only in a limited context. In more general cases, determination of S-equivalence is more difficult.

Regarding items (a) and (b) above, consider the following question: Is there an OPE which is S-equivalent to the PPE

\[
\text{path } ( \{\text{READ}[\text{req}(\text{WRITE})=\text{auth}(\text{WRITE})=0]\} + \text{WRITE})^* ?
\]

This PPE is Andler's solution to the writers' priority readers-writers
problem. The main difficulty in finding a PV-equivalent OPE is that the implicit counter req(WRITE) has no analogue in OPEs. That is, the expression \( \text{req}(\text{WRITE}) - \text{auth}(\text{WRITE}) \) represents the queue length for waiting writers, but OPEs have no explicit access to lengths of semaphore queues. Finding an OPE which is PV-equivalent to a PPE with such a predicate is difficult—in fact, in the above example we believe it is impossible. Thus, in the absence of any more formal techniques, one must attempt to use trial and error to find an S-equivalent OPE.

For the writers' priority problem, an OPE can be written which is based on Campbell and Habermann's RPE solution (shown in Section I.C.1):

\[
\text{path} 1: (\text{READATTEMPT}), 1: (\text{READATTEMPT}, \{\text{WRITE}\}), 1: (\{\text{READATTEMPT}; \text{READ}\}, \text{WRITE})\quad \text{end}
\]

This multilevel OPE is S-equivalent to the PPE (disregarding the synchronization procedure READATTEMPT), yet is clearly not PV-equivalent to the PPE.

Regarding item (c) above, the use of the PPE collateral execution operator "\,\" is problematical in that its use must be restricted if a simple P-V implementation is desired. In his thesis, Campbell gives the following example (here, using the PPE notation): \text{path} (\{(A,B)+C\};D)^*\). He argues that the prologues of A and B will contain P operations on different semaphores, but that the prologue of C cannot do a P on both of these semaphores simultaneously. Thus, if one of A or B passes its semaphore while C is doing a P on the other, then only one of A and B may be executed and a deadlock will result.

In addition, finding an S-equivalent OPE for a PPE with the ","
operator can present difficulties. It can be shown that there are PV-equivalent (hence S-equivalent) OPEs for PPEs which use the "," operator. For example:

\[
\begin{align*}
\text{PPE path } & (A;(B,C))^* \quad S== \quad \text{OPE path } 1:(A;B),1:(A;C) \quad \text{end} \\
\text{PPE path } & ((A,B);C)^* \quad S== \quad \text{OPE path } 1:(A;C),1:(B;C) \quad \text{end} \\
\text{PPE path } & (((A,B)+C);D)^* \quad S== \quad \text{OPE path } 1:(A,C);D),1:(B,C);D) \quad \text{end}
\end{align*}
\]

However, there seems to be no OPE which is PV-equivalent to the PPE path \((A,B)^*\). A P-V implementation for the latter would be

\[
P(s_1)P(s_2)A V(s_1)V(s_3) \text{ and } P(s_3)P(s_4)B V(s_2)V(s_4) \text{ with } s_1,s_2,s_3, \text{ and } s_4 \text{ initially 1. But this pattern of } P \text{ and } V \text{ operations does not match any known pattern resulting from the translation of an OPE. The effect of the cobegin-coend semantics in this case could be achieved by using } OPE \quad \text{path } 1:(A;DUMMY),1:(B;DUMMY) \quad \text{end} \quad \text{and requiring a process to execute the sequence } A;DUMMY \text{ or } B;DUMMY. \text{ However, this introduction of the synchronizing procedure DUMMY changes the nature of access to the resource module in an undesirable way.}
\]

Regarding the "*" operator, Campbell also found problems with its use (in the interior of a path expression) in terms of simplicity of implementation and analysis in a P-V environment. Its use was either absent [Campbell and Habermann 1974] or restricted [Campbell 1977] in various versions of RPEs. We have not attempted to proceed further with investigating its use in the P-V implementation context, although it will be mentioned later in this work after a new implementation semantics is introduced.
2. A subjective comparison of OPEs and PPEs

As indicated in Section II.A, Bloom and, to a lesser extent, Campbell have defined criteria by which to evaluate the notions of power and ease of use with respect to synchronization mechanisms. Although we have not applied formal measures in the following comparison of the two path notations, many of these criteria are certainly inherent in the judgments we have made.

a. Examples showing no clear advantage of one notation over the other

For some synchronization problems, there seems to be no major advantage in choosing an OPE over a PPE or vice versa. Examples 1 through 3 of Section II.B.1 showed S-equivalent OPEs and PPEs which we feel are comparable in terms of ease of use and understandability.

Another example in which we believe there is no clear advantage of one notation over the other is the following.

Example 10. (An n-element stack with operations PUSH and POP)

OPE: path n:(PUSH;POP), 1:(PUSH,POP) end

PPE: def ptr = auth(PUSH) - term(POP)

    path (PUSH[ptr<n] + POP[ptr>0])*

Here, extra consideration is required in the OPE to enforce the mutual exclusion of PUSH from POP, since concurrency is the default mode in OPEs—namely, the addition of the second-level subexpression 1:(PUSH,POP). On the other hand, extra consideration is required in the PPE by using the predicates to specify the correct sequencing of PUSH and POP. The simple use of a semicolon will not suffice in this case.
because that would require PUSHes and POPs to alternate.\footnote{1}

Another problem which yields approximately comparable solutions in the two notations is the n-slot bounded buffer problem. A PPE solution presented by Andler is the following:

```
def msgs = term(PUT) - auth(GET)
slots = n + term(GET) - auth(PUT)
path (PUT[slots>0] + GET[msgs>0])*  
```

Since it is presumed that there are two pointers within the code for the operations PUT and GET (namely, one for the next available slot and another for the last item removed), PUTs must exclude each other and GETs must exclude each other. However, because there are two distinct pointers, there is no reason why one PUT and one GET cannot proceed concurrently—an event disallowed by the above PPE. Therefore, we have modified the PPE to allow a concurrent PUT and GET as the comparable OPE from [Campbell and Kolstad 1980b] does:

**Example 11.** (An n-slot bounded buffer)

**OPE:** \[\text{path } n:(1:(PUT);1:(GET)) \text{ end}\]

**PPE:**

```
def slots = n + term(GET) - auth(PUT)
  busyget = auth(GET) - term(GET)
  busyput = auth(PUT) - term(PUT)
path (\{PUT[slots>0 and busyput=0]; GET[busyget=0]\})*
```

\footnote{1}{It should be noted here that the above PPE, based on one presented in [Andler 1979], is correct only if a centralized implementation of the synchronization mechanism is assumed. In Chapter III it will be seen that this PPE will not suffice in a particular environment involving a distributed, stepwise implementation of synchronization.}
In this example, we favor the OPE for its simplicity of notation; however, a case can be made for preferring the PPE in that it perhaps displays the logic more clearly.

Two more examples in which we feel that the OPE and PPE solutions are comparable are the following readers-writers problems.

**Example 12.** (Readers-writers with mutual exclusion)

OPE: \[ \text{path 1:}(\text{READ},\text{WRITE}) \text{ end} \]

PPE: \[ \text{path } (\text{READ}+\text{WRITE})^* \]

**Example 13.** (Readers-writers with weak readers' priority—waiting readers do not necessarily inhibit writing)

OPE: \[ \text{path 1:} \{\text{READ}\},\text{WRITE} \text{ end} \]

PPE: \[ \text{path } \{\text{READ}\}+\text{WRITE}^* \]

b. **Examples in which OPEs appear to be "better" than PPEs**

In Examples 4 through 9 of Section II.B.1, it is our opinion that the OPEs are preferable to the equivalent PPEs in terms of ease of use and understandability. The major reason for this is the power of the open path notation to express concurrency (since this notation assumes parallelism by default). As the examples show, it requires extra effort with PPEs to express other than simple forms of concurrency, particularly finite parallel execution as specified by the OPE's \[ \text{n:}(\text{list}) \text{ notation.} \]

c. **Examples in which PPEs appear to be "better" than OPEs**

In the following variations of the readers-writers problem, the PPEs are essentially those of Andler, and the OPEs are our constructions based on
the RPEs for these variations presented in [Campbell and Habermann 1974].

Example 14. (Readers-writers with writers' priority—any waiting writers inhibit reading)

PPE: \texttt{def} \texttt{ww} = \texttt{req(WRITE)} - \texttt{auth(WRITE)}
    \texttt{path} \{{READ[ww=0]} \} + \texttt{WRITE}\}

OPE: \texttt{path 1:(READATTEMPT), 1:(READATTEMPT,\{WRITE\}),
      1:(\{READATTEMPT;READ\},WRITE) end}

Example 15. (Readers-writers with strong readers' priority—any waiting readers inhibit writing)

PPE: \texttt{def} \texttt{wr} = \texttt{req(READ)} - \texttt{auth(READ)}
    \texttt{path} \{{READ\} + \texttt{WRITE[wr=0]}\}

OPE: \texttt{path 1:(WRITEATTEMPT), 1:{READ},WRITEATTEMPT),
      1:{READ},(WRITEATTEMPT;WRITE) end}

In Example 15 we have presented the OPE equivalent of Campbell and Habermann's original solution even though their solution may produce a very subtle imprecision in behavior.²

² Bloom [Bloom 1979] makes the following argument (referencing Campbell and Habermann's solution, but here stated in terms of our OPE): if a WRITE is in progress and a second writer sends a WRITEATTEMPT, this WRITEATTEMPT will successfully pass the first two levels of the OPE and then be blocked at the third. A READ request which arrives before the first WRITE has completed will be blocked at the second subexpression by the waiting WRITEATTEMPT; thus the second writer will gain access to the resource object before the reader, even though readers are to have priority. Upon exploring this anomaly, we constructed the following OPE which we believe eliminates this problem:

\texttt{path 1:(WRITE), 1:{(READ),WRITE) end.}
Example 16. (Readers-writers with alternating bursts of readers and writers—waiting writers inhibit only the start of a burst of readers)

\[ \text{PPE: } \textbf{def} \ \texttt{ww} = \text{req(WRITE)} - \text{auth(WRITE)} \]
\[ \text{path } ([\text{READ}[\texttt{ww}=0] + \text{WRITE}]^* \]

\[ \text{OPE: } \text{path } 1;([\text{READ}],[\text{WRITE}]), 1:(\text{WRITE})\text{ end} \]

We feel that the advantage goes to the PPE in each of the three examples above due to the straightforward solution and clarity of logic that each displays. Because of the OPE's inability to access what Bloom calls the synchronization state of the resource (i.e., state information which is not part of the resource object per se but is needed only for synchronization purposes), multilevel expressions using synchronization procedures are required. This need for synchronization procedures detracts from the goal in path expressions of separating the resource object from the synchronization specification—that is, it is difficult to distinguish between operations which access the resource object and operations which are only for synchronizing the accesses. On the other hand, the PPE notation fares well in these examples precisely because the synchronization state of the resource is explicitly accessible through the event counters associated with the operation names.

One further example which bears mentioning is the following, presented in [Andler 1979].
Example 17. (An n-element stack with operations PUSH, POP, and TOP)

PPE:  \begin{align*}
&\text{def} \; \text{ptr} = \text{auth(PUSH)} - \text{term(POP)} \\
&\text{path} \; (\text{PUSH}[\text{ptr<n}] + (\text{POP+TOP})[\text{ptr>0}])^*
\end{align*}

We have been unable to express this with an OPE. The predicate on POP and TOP is designed to prevent removing an item from or looking at an empty stack. It might be argued that this is not necessarily a synchronization issue—that attempting to do either a POP or a TOP on an empty stack should be allowed but would then lead to the report of an error condition. Nevertheless, the solution can be written directly with a PPE, but we have not been able to do so with any OPE. The problem is that the predicate on TOP involves event counters only for PUSH and POP—not for itself. Hence, attempting to place it in a sequencing subexpression of an OPE—for example, \( n:(\text{PUSH};(\text{POP},\text{TOP})) \)—to guarantee that there is something on the stack will not work because we do not want the completion of a TOP to do a V operation on the semaphore for PUSH. On the other hand, placing it outside the sequencing specification presents problems because it then becomes difficult to guarantee that a TOP will be allowed only if more PUSHes than POPs have been authorized. In short, it is again clear to us that the inability of an OPE to directly access the synchronization state can necessitate a great amount of ingenuity and lead to convoluted, difficult to understand solutions (if any can be found at all in a reasonable amount of time).
3. Conclusion

It is our judgment that PPEs are "better" for some problems whereas OPEs are "better" for others. PPEs are more powerful when it comes to problems for which it is natural to express a solution in terms of synchronization constraints external to the unsynchronized resource object, while OPEs are more powerful for expressing other than limited forms of parallelism. It is due to these observations that we propose combining the two notations into open predicate path expressions (OPPEs), to be described in Section II.C.

C. A New Notation—The Open Predicate Path Expression

In this section, we propose a new path notation—the open predicate path expression (OPPE)—which combines the features of the OPE and the PPE. We chose to combine these because we want from the OPE notation:

* the parallel semantics
* the n:(list) notation for finite parallel execution
* the ease of translation into networks of communicating submodules [Oldehoeft and Jennings 1984]

and from the PPE notation the straightforward access to the synchronization state of the resource provided by the implicit event counters and the predicates. Specifically, we adopt the OPE notation in its entirety and add the capability of attaching a predicate according to the following grammar.
1. Syntax of OPPEs

\[\text{<OPPE>} ::= \text{path} \text{<list>} \text{end}\]

\[\text{<list>} ::= \text{<sequence>} | \text{<sequence>}, \text{<list>}\]

\[\text{<sequence>} ::= \text{<item>} | \text{<item>}; \text{<sequence>}\]

\[\text{<item>} ::= \text{<unsigned integer>} : (\text{<list>} )\]

\[\{ \text{<list>} \}\]

\[\text{<item>} [\text{<predicate>} ]\]

\[\{ \text{<list>} \}\]

\[\text{<operation_id>}\]

For the time being, we will restrict the predicate to one or more linear relational expressions (in integer constants and implicit event counters) joined by not, and, and or as follows:

\[\text{<predicate>} ::= \text{<bool_expr>}\]

\[\text{<bool_expr>} ::= \text{<pred_term>} | \text{<bool_expr>} \text{or} \text{<pred_term>}\]

\[\text{<pred_term>} ::= \text{<factor>} | \text{<pred_term>} \text{and} \text{<factor>}\]

\[\text{<factor>} ::= \text{<rel_expr> } | \{ \text{<bool_expr> } \} | \text{not} \text{<factor>}\]

\[\text{<rel_expr>} ::= \text{<arith_expr> <rel_op> <arith_expr>}\]

\[\text{<arith_expr>} ::= \text{<term_symb> } | \text{<term_symb> <sum_op> <arith_expr>}\]

\[\text{<term_symb>} ::= \text{<event_ctr> } | \text{<integer> } | \text{<integer>*<event_ctr>}\]

\[\text{<event_ctr>} ::= \text{req(<operation_id>)}\]

\[\| \text{auth(<operation_id>)}\]

\[\| \text{term(<operation_id>)}\]

\[\text{<sum_op>} ::= + | -\]

\[\text{<rel_op>} ::= > | < | = | # | \geq | \leq\]
2. **Advantages of OPPEs**

For those synchronization problems for which an OPE provides a "better" solution than does a PPE, we can just use the OPE since every OPE is also an OPPE. For example, each of the following allows (at most) one A and (at most) one B to be executed concurrently. The OPE may be used directly since it is also an OPPE.

**Example 18.**

OPPE (and OPE): \[ \text{path 2:}(1:(A), 1:(B)) \text{ end} \]

PPE: \[ \text{path} \left( \{ (A[\text{auth}(A)\text{-term}(A)<1] + B[\text{auth}(B)\text{-term}(B)<1] \} \[\text{auth}(A)\text{-term}(A)+\text{auth}(B)\text{-term}(B)<2] \} \right) \left( \text{auth}(A)\text{-term}(A)+\text{auth}(B)\text{-term}(B)<2 \right) \]

On the other hand, when a PPE yields a "better" solution than does an OPE, we can write the PPE in the equivalent OPPE form. In most cases, this amounts to forcing mutual exclusion by writing \(X+Y\) as \(1:(X,Y)\). The following example shows the OPPE solution of the writers' priority readers-writers problem along with the OPE version for comparison.

**Example 19.** (Readers-writers with writers' priority)

OPPE: \[ \text{def} \text{ww = req(WRITE) - auth(WRITE)} \]

\[ \text{path 1:}([\text{READ[ww=0]}], \text{WRITE}) \text{ end} \]

OPE: \[ \text{path 1:}([\text{READATTEMPT}], 1:(\text{READATTEMPT}, \text{WRITE})), 1:([\text{READATTEMPT}; \text{READ}], \text{WRITE}) \text{ end} \]

The reader is encouraged to compare this OPPE with the PPE shown in Example 14 of Section II.B.1.

Another advantage afforded by the OPPE notation is that of
flexibility in expressing a problem solution. For example, if we define

\[
\text{msgs} = \text{term(PUT)} - \text{auth(GET)}
\]

\[
\text{slots} = n + \text{term(GET)} - \text{auth(PUT)}
\]

\[
\text{busyget} = \text{auth(GET)} - \text{term(GET)}
\]

\[
\text{busyput} = \text{auth(PUT)} - \text{term(PUT)}
\]

then the following are all valid OPPEs for the n-slot bounded buffer problem which allows a PUT and a GET to execute concurrently:

**Example 20.** (OPPEs for the n-slot bounded buffer)

a) path n:(l:(PUT);l:(GET)) end

b) path PUT[slots>0 and busyput=0], GET[msgs>0 and busyget=0] end

c) path 1:(PUT)[slots>0], 1:(GET)[msgs>0] end

d) path 1:(PUT)[slots>0]; 1:(GET) end

e) path PUT[slots>0 and busyput=0]; GET[busyget=0] end

Another advantage which is potentially even more important than those mentioned already is that for some problems the OPPE notation seems to be more powerful than either the OPE or PPE notation alone. The following example shows a solution in each of the three notations for allowing a single A and a single B to execute concurrently with the added restriction that an A may proceed only if there are no waiting requests for B.

**Example 21.**

OPPE: def wb = req(B) - auth(B)

\[
\text{path 2}:(1:(A)[wb=0], 1:(B)) \text{ end}
\]
PPE: \[\text{def } wb = \text{req}(B) - \text{auth}(B)\]

\[\text{busya} = \text{auth}(A) - \text{term}(A)\]

\[\text{busyb} = \text{auth}(B) - \text{term}(B)\]

\[\text{path } \{\text{(A[busya=0 and wb=0] + B[busyb=0])}
\text{[busya+busyb<2]}}\}^*\]

OPE: \[\text{path 1:(ATTEMP_A), 1:(ATTEM_A, \{B\}),}
2:(1:(ATTEM_A;A), 1:(B)) \text{ end}\]

In this example, extra effort is required in composing the PPE to specify the concurrency of an A and a B—a hindrance seen in previous examples. The writing of the OPE also requires extra effort in employing multiple path levels and the synchronizing procedure ATTEM_A to deduce whether any Bs are waiting—again, a drawback seen in previous examples. The OPPE, however, expresses both the desired parallelism and access to the synchronization state (via the predicate) directly.
III. AN IMPLEMENTATION SEMANTICS FOR OPEN PREDICATE PATH
EXPRESSIONS—THE DATAFLOW GRAPH

In this chapter we will begin by considering P-V implementation
semantics for OPPEs as was done in Chapter II for OPEs and PPEs. We
will then present and discuss a scheme for representing semaphores and
predicates (and the associated flow of information and control among
them) in graphical form. This graphical representation will lend itself
directly to implementation by applicative, stream-oriented program
submodules in highly parallel, data-driven environments. It will be
emphasized, however, that the graphical representation is useful even
for traditional von Neumann systems which support true concurrent
processing (as in multiprocessor systems) as well as an aid to
understanding synchronization problem solutions, regardless of the form
of the target architecture.

A. P-V Implementation Semantics for OPPEs

P-V implementation semantics for OPPEs can be specified by using
the transformation rules as presented in Section II.B.1 for OPEs and
adding the PPE transformation rule for predicates. Hence, the following
transformation rules are applied in the order given by a left-to-right
parse of the OPPE according to the production rules in the OPPE grammar
presented in Section II.C.1:

1. Replace path list end by null list null.
2. Replace L sequence,list R by L sequence R
and L list R.

3. Replace L item:sequence R by L item V(s1)
   and P(s1) sequence R
   with semaphore s1 initialized to 0.

4. Replace L n:(list) R by P(s2) L list R V(s2)
   with semaphore s2 initialized to n.

5. Replace L {list} R by PP(c,s,L) list VV(c,s,R)
   where PP and VV are defined as in Section II.B.1
   and initialize counter c to 0 and semaphore s to 1.

6. Replace L item[predicate] R by W(predicate) L item R
   where primitive W is as defined in Section II.B.1.

7(a). If operation_id has not appeared in the OPPE prior to this
     instance, replace L operation_id R by
     L
     operation body
     R

7(b). If operation_id has appeared and been transformed into

     L'
     operation body
     R'

     prior to the current appearance of operation_id, then
     replace the current instance of L operation_id R by

     L'
     L
     operation body
     R
     R'

Up to this point, the W operation has been introduced and used as a
primitive like the P and V operations. Although P and V operations are
widely known and understood independently of their implementations, a
realization of the W operation needs to be presented in order to establish its credibility as a synchronization primitive. Such a realization will be discussed in Section C of this chapter.

Regarding the usefulness of P-V implementation semantics in our proposed target environment, a major problem arises at Step 7 in the above transformation rules. Even though the specification of synchronization by path expressions is separate from the code for operations on the resource object (thus relieving the programmer of the task of explicitly programming the synchronization), the P-V implementation causes P and V operations to be embedded within the operation code. Thus, a total separation of synchronization from access to the resource—a design goal in our target environment—is not effected. The remaining sections of this chapter discuss an alternative semantics (or alternative implementation of the P-V semantics, depending upon whether one views various levels in an implementation hierarchy as new semantics).

B. Synthesis of OPEs into Dataflow Graphs

1. Background

The general design philosophy of our target environment—namely, total encapsulation of resource modules in distributed systems with potentially highly-parallel nodes—specifies that a resource module will include two internal submodules: one for synchronization/scheduling and one for accessing the resource object itself. These two functions are totally separated by encapsulating each as a separate submodule written
in its own high-level specification language, and they communicate by passing messages. In [Oldehoeft and Jennings 1984], Oldehoeft proposes the OPE notation as a possible choice for a specification language for the synchronization submodule. He presents a scheme for the automated synthesis of OPEs into dataflow graphs which are then implemented by networks of communicating submodules written as applicative language programs. In particular, the dataflow graphs and their corresponding program implementations were employed with the goal of direct application to data-driven dataflow architectures.

Upon investigating this scheme further and formulating its extension to translating OPPEs, we have concluded that the synthesis of path expressions into dataflow graphs yields an extremely useful definition of the semantics of a path expression, regardless of the form of the underlying support architecture. That is, the semaphores and predicates represented graphically in the dataflow schemata could be implemented in von Neumann systems which support true concurrency; however, these graph semantics naturally expose inter-submodule parallelism so that implementation on highly parallel, functional architectures is not precluded. Were sequential language semantics to be used, such parallelism might be limited [Oldehoeft et al. 1979], and massive transformations of such programs might be required to unfold the parallelism [Jennings and Oldehoeft 1983].
2. A synthesis scheme for OPEs

As discussed in Section I.B, the synchronization submodule is logically activated by the arrival of an operation request message (or a termination message from the access-mechanism submodule, to be discussed shortly) and the availability of its internal state. This submodule then produces a new value of its internal state and, potentially, an output message—the operation request—to be delivered to the access-mechanism submodule. The dataflow graph [Dennis 1974] will be used to depict the flow of execution based on the availability of information as shown in Figures I-2, I-3, and I-4.

a. Oldehoeft's controllers Oldehoeft's scheme involves synthesizing an OPE into a network of communicating controllers (submodules internal to the synchronization submodule). There are three types of controllers: the PV-controller, the burst-controller, and the distributor. These controllers may be represented by dataflow graphs with the following interpretation: each controller is activated by the presence of a message on its primary input arc and the availability of its internal state (i.e., the presence of a token on the other input arc); the controller consumes the input message and the value of its internal state and then produces the new value of its state and zero or more output messages.

1) The PV-controller (see Figure III-1) This is a message-driven implementation of the semaphore [Dijkstra 1968]. Its internal state consists of an integer counter and a queue of waiting messages. When an operation-request message arrives (logically, a P
operation on the semaphore), the message is sent along on the operation-request output arc if the counter is positive (else, it is enqueued), and the counter is decremented. If the arriving message is an operation-termination message sent by the access-mechanism (logically, a V operation on the semaphore), a waiting operation-request message is dequeued if the count is negative; then the counter is incremented, and the operation-termination message is passed along on the signal output arc. (The use of this output arc will be described when the synthesis algorithm is discussed.)

Oldehoeft represents the internal details of the PV-controller as a high-level applicative language program. The language is in the style of existing or proposed dataflow languages (Bryant and Dennis 1982, Dennis and Weng 1979, McGraw 1982), and the use of recursion in
conjunction with the concept of an input stream\(^1\) of data values forces a (temporary) serialization of responses to the incoming messages. In this language, the controller will only begin executing when all its input parameters are available—hence, a subsequent invocation of the controller will wait until the new internal state (i.e., the counter and the queue) has been produced.

Because internal parallelism in each controller submodule is thus not a major issue, we present here the details of the controllers in a more conventional sequential language notation.\(^2\) This Pascal-like notation includes message-passing primitives send and receive, where a process executing receive waits until a message is received, and send has "no-wait" semantics (i.e., the sending process does not wait for the destination process to receive the message). Also, these controllers are shown as parametrized generic modules (in the style of Ada [DoD 1980]), whereby each serves as a template and may be instantiated with different actual parameters to allow for multiple instances of the module. Figure III-2 displays the high-level code for a PV-controller. For this module, a typical instantiation would be:

\[
\text{module pv1: new pv_controller(2, pv2, bursts)}
\]

where "pv2" and "bursts" are existing instances of a PV- and burst-

\[^1\] A stream is a sequence of values, all of the same type, which are passed from one module to another in sequential order.

\[^2\] The reader is referred to Appendix A for the applicative language versions of these basic controllers. The applicative approach does provide for more parallelism within submodules and is, in fact, necessary for implementation on dataflow machines.
controller.

It should be noted that the procedure "insert" is assumed to be written by the programmer so that operation requests can be enqueued in any priority order desired. Thus, although FIFO enqueuing of requests is implicit in Campbell and Kolstad's description of OPEs [Campbell and Kolstad 1980a], our implementation does not make this assumption.

generic (pv_count: integer; op_dest, signal_dest: module)
module pv_controller;
var pv_queue: queue type;
head: message;
begin
  pv_queue := createq();
  while true do
    begin receive(msg: message);
      case msg.type of
      'req': begin
        if pv_count > 0
          then send (msg) to (op_dest)
        else insert(msg, pv_queue);
        pv_count := pv_count - 1
      end;
      'term': begin
        if pv_count < 0
          then begin
            head := dequeue(pv_queue);
            send (head) to (op_dest)
          end;
        send (msg) to (signal_dest);
        pv_count := pv_count + 1
      end
      end {case}
    end {while}
end {pv_controller}

FIGURE III-2. High-level program for a PV-controller

2) The burst-controller (see Figure III-3) This is a message-driven implementation of the compound semaphore which uses PP
FIGURE III-3. Burst-controller as a dataflow module

and VV operations [Campbell and Habermann 1974]. Its purpose is to synchronize a burst (i.e., parallel execution) of one or more operations with respect to other operations. Its internal state consists of a counter, a queue of waiting messages, and a phase designation ("idle", "initiate", or "active"). When an operation-request message arrives (logically, a PP operation) during the "idle" phase, the message is enqueued, a "start_burst" message is placed on the second output arc to effect any presynchronization required to start the burst, and the phase is changed to "initiate". During this phase, any incoming operation- request messages are simply enqueued. When the "start_burst" message eventually returns to the burst-controller, all waiting operation- request messages are dequeued and placed on the third output arc. The phase is then changed to "active", and all incoming operation requests are placed directly on the third output arc. Throughout all phases, the receipt of an operation request causes the counter to be incremented,
whereas the receipt of an operation-termination message causes the counter to be decremented. When the count reaches zero, a burst-termination message is placed on the fourth output arc to effect any desired postsynchronization and the phase is reset to "idle", ending the burst. Figure III-4 displays the high-level code for a burst-controller. For this module, a typical instantiation would be:

```module burst5: new burst_ctrlr(pv3, burst4, pv2, pv7)```

where "pv2", "pv3", and "pv7" are instances of a PV-controller and "burst4" is an instance of a burst-controller.

3) The **distributor** (see Figure III-5) This is a message-driven m-way output switch or router. It has no internal state. An incoming message is simply placed on the output arc which is labeled with the name of that message.

Figure III-6 shows a sequential language program representation of a 1-by-4 distributor which routes a stream of messages, each of which is a request for operation "opl" or "op2" or a termination message for "opl" or "op2". For this module, a typical instantiation would be:

```module dist2: new distributor4('opl','op2','term_opl','term_op2', burst1, pv3, pv5, pv6)```

where "pv3", "pv5", and "pv6" are instances of a PV-controller and "burst1" is an instance of a burst-controller.

b. The **translation algorithm** Oldehoeft's synthesis of an OPE into a network of PV-controllers, burst-controllers, and distributors is based on a left-to-right, bottom-up parse of the path expression. He defines attributes called "left sets" and "right sets" such that each
generic (start dest, op_dest, term dest, sig dest: module)
module burst_ctrlr;
var phase: string init ('idle');    head: message;
    count: integer init (0);     queue: ↑queuetype;
begin
    queue := createq();
    while true do
        begin
            receive(msg: message);
            case phase of
                'idle': begin
                    send ('start_burst') to (start dest);
                    count := 1;
                    insert(msg, queue);
                    phase := 'initiate'
                end;
                'initiate': case msg.type of
                    'start_burst': begin
                        while not empty(queue) do
                            begin
                                head := dequeue(queue);
                                send (head) to (op_dest)
                            end;
                            phase := 'active'
                        end;
                        'req': begin
                            count := count+1;
                            insert(msg, queue)
                        end
                    end {case};
                'active': case msg.type of
                    'req': begin
                        send (msg) to (op_dest);
                        count := count+1
                    end;
                    'term': if count = 1
                        then begin
                            send ('term_burst')
                            to (term_dest);
                            count := 0;
                            phase := 'idle'
                        end
                    else count := count-1
                end {case}
        end {while}
    end {burst_ctrlr}

FIGURE III-4. High-level program for a burst-controller
FIGURE III-5. M-way distributor as a dataflow module

\[
\text{messages} \downarrow \\
\text{dist} \\
\downarrow \\
\ldots \\
\leftarrow m \text{ arcs} \rightarrow
\]

nonterminal in the syntax tree for an OPE has a left set and a right set. These sets essentially identify which operation names and termination signal names are to be associated with each of the controllers generated during the parse.

Rather than present his algorithm for the generation and interconnection of controllers, we present here our adaptation which uses a top-down, rather than bottom-up, parse of an OPE. We have chosen
to do so because our previous discussion of P-V implementation semantics was based on top-down parsing of path expressions, and we would like to be able to relate this algorithm directly to that used with the P-V semantics.

Rules 1 through 5 of Step 1 below are direct augmentations of the transformation rules for the P-V implementation of OPEs as presented in Section II.B.1.b. However, Rule 6 defines a total departure in its specification of the interconnection of the controllers when a leaf in the translation tree is reached.

Step 1. The following six transformation rules are repeatedly applied in the order determined by a left-to-right, top-down parse of the OPE according to the production rules in the OPE grammar. (Again, each of L and R represents a previously generated synchronization primitive—or sequence of primitives—on the left and right of the subexpression.)

1. Replace path list end by null list null.
2. Replace L sequence,list R by L sequence R and L list R.
3. Replace L item;sequence R by L item V(sl) and P(sl) sequence R with semaphore sl initialized to 0.

Additionally, generate a PV-controller with an initial state consisting of a zero pv-counter and an empty pv-queue. Leave the primary input arc and both output arcs unlabeled for the
time being.

4. Replace $L n:(\text{list}) R$ by $P(s2) L \text{ list } R V(s2)$ with semaphore $s2$ initialized to $n$.
   Additionally, generate a PV-controller with an initial state consisting of $\text{pv-counter}$ initialized to $n$ and an empty $\text{pv-queue}$. Leave the primary input arc and both output arcs unlabeled for the time being.

5. Replace $L \{\text{list}\} R$ by $PP(c,s,L) \text{ list } VV(c,s,R)$ where $PP$ and $VV$ are defined as in Section II.B.1 and initialize counter $c$ to 0 and semaphore $s$ to 1. Additionally, generate a burst-controller with an initial state consisting of a zero burst-counter, an empty burst-queue, and an "idle" phase designation. Leave the signal and operation-request output arcs unlabeled for the time being.

6. When $L \text{ operation_id } R$ is encountered, do the following:
   a. For each PV- or burst-controller (in left-to-right order) corresponding to the semaphore(s) named in $L$, do:
      (i) Merge into this controller's primary input arc an arc labeled "operation_id", and connect to this arc any identically labeled unconnected output arc.
      (ii) Label this controller's operation-request output arc with "operation_id".
      (iii) (Burst-controller only) For the primitive $PP(c,s,L')$ which this controller implements, recursively apply Rule 6a (again, in left-to-right order) with $L'$
replacing L and "start_burst" replacing "operation_id".

b. For each controller (in right-to-left order) corresponding to the semaphore(s) named in R, do:

(i) Merge into this controller's primary input arc an arc labeled "term_operation_id".

(ii) Label this controller's signal output arc with "term_operation_id", and connect this output arc to any identically labeled unconnected input arc (which is input to a controller other than this one).

(iii) (Burst-controller only) For the primitive VV(c,s,R') which this controller implements, recursively apply Rule 6b (again, in right-to-left order) with R' replacing R and "term_burst" replacing "term_operation_id".

It should be noted that in substeps 6a(i) and 6b(ii), the connection of the output (input) of an existing controller to the input (output) of a newly labeled controller may make it necessary to insert a distributor if the output line is labeled with more than one message name.

With completion of the parse of the path expression, Step 1 is completed. All the controllers have now been generated, and a number of their interconnections will have been made. To complete the interconnections, Steps 2 through 4 are now applied (in that order).
Step 2. Connect all unconnected operation-request output arcs labeled "start_burst" to identically labeled unconnected input arcs, and connect all unconnected burst-controller "term_burst" output arcs to identically labeled unconnected input arcs.

Step 3. Connect each operation-request output arc having no terminal node to the access-mechanism submodule (via a merge if more than one).

Step 4. Generate a distributor which accepts a merge of two message streams: (a) operation-request messages from the entry port to the resource module and (b) operation-termination messages from the access-mechanism submodule. Connect the distributor's output arcs either to identically labeled unconnected controller input arcs or directly to the access-mechanism (for those operation-request messages for which there are no corresponding unconnected input arcs).

3. Examples

Before examples of the synthesis scheme are presented, an abbreviated dataflow graph notation for PV- and burst-controllers will be described. Consider the representation of a PV-controller (with unlabeled output arcs and primary input arc) shown in Figure III-7. Note that the state of a PV-controller must be initialized since, conceptually, the controller will only execute when there is a token on each of its input arcs. Hence, for the controller in Figure III-7, a token representing the initial state value (pv-counter of 2 and empty pv-queue) would be initially produced and input, via the merge operator,
FIGURE III-7. Sample PV-controller graph

to the controller. Once a token arrives on the primary input arc, the controller will then execute, producing as one of its outputs a token representing the new state value. This token will then get routed back through the merge operator as input for the next invocation of the controller.

From this point on, the graph for each PV- and burst-controller will be abbreviated as follows: since each PV- and burst-controller always has a recirculating state token, this arc will be omitted with the understanding that it is always implicitly present. As for the initial state values, each PV-controller has an initially empty pv-queue but its initial pv-counter value (n≥0) varies. Thus, our abbreviated graph will show only the initial pv-counter value. This will be done as shown in Figure III-8.

On the other hand, all burst-controllers have identical initial values: burst-counter of zero, empty burst-queue, and "idle" phase designation. Thus, the entire arc representing the state and initial values will henceforth be omitted.

Example 1 Consider the following OPE: path 2:(A;B) end. For this OPE, the complete translation tree is shown in Figure III-9.
FIGURE III-8. Abbreviated form of Figure III-7

FIGURE III-9. P-V translation tree for OPE path 2:(A;B) end

By the time the leftmost leaf is reached in the translation, application of Rules 1, 4, and 3 (in that order) of Step 1 will have caused two PV-controllers to be generated (with output arcs and primary input arc as yet unlabeled)—see Figure III-10. After this leftmost leaf in the tree is processed using Rules 6a and 6b of Step 1, the PV-controllers will have arcs labeled as in Figure III-11. Then, after the rightmost leaf in the tree is processed using Rules 6a and 6b, the parse (hence, Step 1 of the synthesis) will have been completed, yielding the configuration shown in Figure III-12. Step 2 of the synthesis is irrelevant here.
because there are no burst-controllers, so Step 3 is carried out and yields the graph shown in Figure III-13.

Note in Figure III-13 that we have simply "grounded" the signal output arc of each controller since application of Rule 6b did not cause either output to be connected to a previously unconnected, identically labeled input arc. Lastly, Step 4 of the synthesis causes a distributor to be generated and the remaining connections to be made. The resulting synchronization submodule is shown in Figure III-14 in the context of the entire resource module.
At this point, two more abbreviations will be made in all following graphs: the omission of "grounded" signal output arcs and of the front-end distributor (produced by Step 4 of the synthesis algorithm). It is an arbitrary design choice to use a distributor in conjunction with a single input port to the synchronization submodule. But one could as well use multiple input ports—one for each operation-request and
FIGURE III-14. Graph of resource module with OPE

path 2:(A;B) end

operation-termination name—so from now on, this initial distributor
will be omitted from the graphs (with the interpretation that the
synchronization submodule's incoming messages are either multi-ported or
are singly-ported and then distributed.) Our final abbreviated graph for Example 1 is thus the graph shown in Figure III-15.

Example 2 Consider the following OPE: \texttt{path 5:} (4:(A),3:(B)) \texttt{end.}

This OPE states that as many as four As may execute concurrently and up to three Bs may execute concurrently, subject to the constraint that no more than a total of five As and Bs may be executing at one time. This example is presented to clarify the actions of Rules 6a and 6b of Step 1 of the synthesis algorithm when each of L and R (in \texttt{L operation}_{id} R) consists of more than one P or V operation.

In this case, the parse of the path expression will cause the leaf \texttt{P(s2)P(s1) A V(s1)V(s2)} in the translation tree to be encountered first (with \texttt{s1} initially 5 and \texttt{s2} initially 4). Rule 6a will then be applied to the lefthand sequence \texttt{P(s2)P(s1)} in left-to-right order—that is,
first to s2, then to s1—yielding the graph in Figure III-16. Then, Rule 6b will be applied to the righthand sequence \( V(s1)V(s2) \) in right-to-left order—that is, first to s2 (see Figure III-17) and then to s1 (see dashed line in Figure III-18).

The second (and final) leaf encountered in the translation tree will be \( P(s3)P(s1) \) with s3 initially 3. Rule 6a will then be applied to the lefthand sequence \( P(s3)P(s1) \) in left-to-right order, yielding the graph shown in Figure III-19. (Note in Figure III-19 how the "grounded" signal output arc from s2 has disappeared.) Then, Rule 6b will be applied to the righthand sequence \( V(s1)V(s3) \) in right-to-left order; Step 2 will not be applied because there are no burst-controllers; Step 3 will be applied; and Step 4 will no longer be
FIGURE III-17. Partial graph 2 for OPE path 5:(4:(A),3:(B)) end

FIGURE III-18. Partial graph 3 for OPE path 5:(4:(A),3:(B)) end
applied in our abbreviated graphs. The resulting abbreviated dataflow graph is shown in Figure III-20.

Note in Figure III-20 that the signal output arc of s1 carries two signals, each having a different destination—hence, a distributor had to be inserted. (Note also another abbreviation we have made in Figure III-20 by labeling the central input arc to s2 with "term_A" and "term_B" instead of drawing two input arcs. This type of abbreviation will be made frequently in ensuing graphs.)

At this point, the reader is encouraged to try the synthesis scheme on the OPE path 4:(A), 3:(B), 5:(A,B) end. In doing so, he/she should discover that the derived dataflow graph is identical to that in Figure III-20 (and, hence, be convinced that the two OPEs have the same meaning.
Example 3  Consider the following OPE: path 1:([A]);B end. This OPE expresses the requirement that each request for B must wait for a burst of As to complete. This example is presented to clarify the role of Rule 6b(iii) of Step 1. Here, the first leaf encountered in the translation tree is PP(c,s3,P(s2)) A VV(c,s3,(V(s1);V(s2))) with s1 initially 0 and s2 initially 1. When Rule 6b is applied, Rule 6b(iii) will pertain because the VV is implemented by a burst-controller. Thus, Rule 6b will be recursively applied to the sequence V(s1)V(s2) in right-
to-left order, the result being that s1 receives the term_burst signal first and then passes it on to s2. The other leaf in the translation tree is \( P(s1) \) B null with s1 initially 0, and the graph will appear as in Figure III-21 when the parse (hence, Step 1) has been completed.

After this, application of Step 2 causes the burst-controller's burst-termination output arc to be connected to the corresponding input arc of s1. Then after application of Step 3, the final abbreviated
The dataflow graph is as shown in Figure III-22. Note that there is no arc labeled "term B" being ported in at the top of the graph (as feedback from the access-mechanism). It has simply been omitted because it would be "grounded" since it is not needed as input to any controller.

**Example 4** Consider the following OPE: path A;1;([B]) end. This example is symmetric to the previous example and is presented to clarify
the role of Rule 6a(iii) of Step 1. Here, the two leaves in the translation tree are null A V(s1) and PP(c,s3,(P(s2);P(s1))) B VV(c,s3,V(s2)) with s1 initially 0 and s2 initially 1. For the latter leaf, Rule 6a(iii) pertains to the left-hand operation PP(c,s3,(P(s2);P(s1))). Thus, Rule 6a is recursively applied to the sequence P(s2)P(s1) in left-to-right order. The resulting graph is as shown in Figure III-23.

FIGURE III-23. Dataflow graph for OPE path A;1:{B} end
Other examples The reader is referred to Appendix B for examples of dataflow graphs synthesized from OPEs representing a number of classical synchronization problems: the bounded buffer, the stack, and several versions of the readers-writers problem.

C. Synthesis of OPPEs into Dataflow Graphs

In this section, we define dataflow graph semantics for OPPEs. This material will be presented in a narrative style which reflects a heuristic approach to finding a general synthesis scheme. The reader will be led through several attempts which result in failure with respect to the generality of the scheme; however, we will ultimately arrive at what we believe to be a correct general synthesis algorithm. Formal proofs of correctness of the implementation semantics will be needed in order to confirm this, and this is discussed as an area for further research in Chapter V.

1. The pred-controller (see Figure III-24)

In addition to the basic controllers described in Section III.B for the synthesis of OPEs we introduce another controller: the pred-controller (abbreviated PC). This is a message-driven implementation of a predicate as used in an OPPE. Its primary input arc consists of a merge of operation-request messages and signal messages. The signal messages (usually operation-termination messages) are used by the PC and then passed on to the signal output arc in the same manner as with PV- and burst-controllers, and the operation-request output arc passes along an operation-request message when the predicate for that operation is
true. The internal state consists of one or more event counters (to be discussed shortly) and a queuing mechanism which functions to block requests when a predicate is false and to provide for unblocking one or more requests when an event occurs which makes a predicate true. The precise nature of this queuing mechanism will be unfolded in the discussion which follows.

2. **Distributed event counters**

For the implicit event counters—req(X), auth(X), and term(X)—associated with an operation X, a design choice has to be made with respect to their physical location. One could collect all the event counters into one "guardian" controller which would serially receive requests to increment and decrement the counters and send their values to appropriate PCs. Unfortunately, such a guardian controller could tend to become a bottleneck within the synchronization network and inhibit the parallelism provided by the synthesis scheme presented in
the previous section.

The alternative we have taken is to localize the event counters only in those PCs which actually reference them. An inspiration for doing so was provided by [Robert and Verjus 1977]. In that paper, the authors introduce an "effective control variable" as a variable whose value at any time is equal to the left-hand side of a relational expression which is in the following form: the left-hand side is a linear combination of event counters with integral coefficients and the right-hand side is an integer or an integer modulo. For example, for the predicate \( \text{term}(A) - \text{auth}(B) > 0 \) one would take \( t = \text{term}(A) - \text{auth}(B) \) as the effective control variable. They then make the point that, since operations on such effective control variables consist of additions and subtractions of 1 and tests on their values, they act as generalized semaphores. For the above predicate, we reasoned that the receipt of a signal "term\textsubscript{A}" from the access-mechanism would amount to performing a V-like operation on the variable \( t \) and the receipt of a signal "auth\textsubscript{B}" (from somewhere) would be a P-like operation. This approach seemed attractive because of the P-V implementation semantics already built on to date. Although we have not employed a pure "effective control variable" mechanism, we have chosen to distribute the event counters, and this distribution will be seen in the examples which follow.\(^3\)

\(^3\) "Effective control variable" is similar to "numerical value of a condition" as defined by Schmid [Schmid 1976] in another context—that of conditional critical regions.
3. A preliminary algorithm for generating and connecting PCs

We will begin by presenting a preliminary translation algorithm which deals with predicates in an OPPE and then go through several stages of modification of the algorithm, motivated by the work of others as well as our own reasoning. At each stage of refinement, we will find that some synchronization problems can be solved but that the scheme fails to be general enough to deal with others. It will only be at the final stage presented that we feel we have arrived at a general translation scheme.

a. Changes to the existing algorithm

We begin by considering the translation algorithm for OPEs as presented in Section III.B.2.b. Between Rules 5 and 6 of Step 1, insert:

5.5. Replace L item[predicate] R by W(predicate) L item R.

Additionally, generate a pred-controller with primary input arc and both output arcs unlabeled for the time being. The initial state will consist of integer event counters (all initially zero) corresponding to those stated in the predicate and an initially empty queue for waiting messages.

Then, replace Rule 6 of Step 1 by the following:

6. When L operation_id R is encountered, do the following:

a. For each PV-, burst-, or pred-controller (in left-to-right order) corresponding to the semaphore(s) or W operator(s) named in L, do:

(i) -same-
(ii) -same-

(iii) -same-

(iv) (Pred-controller only) For each event counter req(X), auth(X), or term(X) named in the predicate, merge into this controller's primary input arc an arc labeled "req_X", "auth_X", or "term_X". Also, label the signal output arc with "req_X", "auth_X", or "term_X", and connect this output arc to any identically labeled unconnected input arc to another controller.

b. An example: the stack (PUSH and POP only) In Example 10 of Section II.B was given a PPE for an n-element stack with operations PUSH and POP. The corresponding OPPE is as follows:

```plaintext
  def ptr = auth(PUSH) - term(POP)
  path 1:(PUSH[ptr<n], POP[ptr>0])
```

Using the translation algorithm as modified above to incorporate the PCs, we obtain the dataflow graph in Figure III-25 (which will be shown to be incorrect). As with the dataflow graphs presented in Section III.B, the unconnected input arcs at the top of the graph are assumed to emanate either from multiple ports to the synchronization submodule or from a distributor if the synchronization submodule has only a single input port. In particular, the signal messages term_PUSH and term_POP have been sent by the access-mechanism submodule upon completion of the corresponding operation. Furthermore, we assume at this point that the access-mechanism has been augmented to provide an authorization signal as feedback to the synchronization submodule as soon as it begins.
executing an operation for which an operation-request message was received. Hence, in Figure III-25 auth_PUSH messages also originate at the access-mechanism.

In examining Figure III-25 and picturing various combinations of tokens arriving at the controllers, it quickly becomes apparent that the graph does not implement the desired synchronization constraints. There are two fatal problems. First, suppose that all event counters are initially zero and let n, the size of the stack, be 1. If a request for
PUSH arrives at PCI it will be released immediately since the predicate is true, and it will then arrive at the PV-controller. The PV-controller will then release the request to be executed by the access-mechanism and will also decrement its pv-counter to zero. Now assume that 400 consecutive requests for PUSH arrive at PCI before it receives the auth_PUSH message from the access-mechanism. Each of the 400 requests will be released because the predicate remains true—it will only become false when the arrival of auth_PUSH causes the left-hand side of the relational expression to equal 1. Hence, even though any further requests for PUSH that arrive at the PC will correctly be blocked, there are still 400 requests enqueued at the PV-controller and they will proceed to execute one at a time on a (supposedly) one-element stack—it is now too late to rescind permission.

The second fatal problem involves the predicate for POP. Suppose, as before, that all event counters are initially zero and that n=1. Let a request for POP arrive at PC2. The request will correctly be blocked since it is false that 0>0. Suppose further that 1000 consecutive requests for POP then arrive at this PC. Since the predicate is still false, they will all be enqueued. Now let a request for PUSH arrive at PCI. It will pass through this PC and the PV-controller and arrive at the access-mechanism. The access-mechanism will then send auth_PUSH back to the synchronization submodule and proceed to execute the PUSH. Upon receipt of auth_PUSH, PC2 will then sequentially release all 1001 requests for POP since its predicate is now true (1>0). Hence, even if no more requests for PUSH arrive, there are 1001 requests for POP
enqueued at the PV-controller and they will begin executing consecutively as soon as the PUSH has completed (thus allowing a multitude of POP operations to execute on an empty stack).

The first problem is dealt with by comparing this graph with that of Figure VII-2 in Appendix B which implements the (unpredicated) OPE for this stack problem. There, a PV-controller with pv-counter initially n is used to prevent more than n consecutive requests for PUSH from proceeding without intervening terminations of POP. The distinction between that PV-controller and our pred-controller PCI is clearly that the PV-controller decrements its counter immediately upon authorizing a PUSH to proceed, whereas the PC must wait for an auth_PUSH message to arrive (too late) from the access-mechanism. The crux of the issue (and, in fact, a fundamental semantic principle in our synthesis scheme) is this: in our distributed implementation, "authorization" means authorization to proceed with further synchronization, not authorization to begin executing in the access-mechanism. This is why we have chosen to use the term "authorization" rather than Andler's "activation". Thus, we solve the first problem by making authorization of an operation a local matter. That is, the access-mechanism will not provide any authorization signals; rather, in releasing an operation request for further processing in the synchronization network, a PC will increment the corresponding authorization event counter immediately.

The second problem—that of PC2's releasing arbitrarily many POPs as soon as an authorization of PUSH is signalled—is also dealt with by examining the dataflow graph of Figure VII-2. There, it is the (local)
authorization of a POP that decrements the pv-counter and a termination of a PUSH which increments it. So it would seem that the correct predicate for POP in our implementation environment should be term(PUSH)-auth(POP)>0, rather than the predicate auth(PUSH)-term(POP)>0 given by Andler. Our observation is this: in Andler's (and others') implementation environment, the mutual exclusion of PUSH from POP in the context of a centralized granting of access to the resource object allows one to write term(PUSH) interchangeably with auth(PUSH)—likewise for auth(POP) and term(POP). Thus, in that context, either of the two predicates auth(PUSH)-term(POP)>0 or term(PUSH)-auth(POP)>0 would be valid. But in our environment, only the latter will suffice since it is crucial to know how many POP requests have been authorized to proceed down to the final PV-controller.

c. A revised preliminary algorithm

We now revise the preliminary translation algorithm given in III.C.3.a so that in Rule 6a(iv), we do not merge into the PC's primary input arc an arc labeled "auth X" if X is the named operation_id. Instead, it will be assumed that the PC will automatically decrement internally its event counter auth(X) when it releases the request for X for further synchronization.

Using this alteration to the synthesis scheme and changing the predicate on POP as discussed previously, we obtain the dataflow graph shown in Figure III-26. (Note that we have rewritten the predicate on PUSH so that it is clearer that term POP does a V-like operation on the PC and a request for PUSH does a P-like operation. The transformation of all predicates into such a normal form will be discussed later in
In comparing this graph to that of Figure VII-2, it is apparent that the two PCs behave like the two corresponding PV-controllers and that the graphs are nearly identical, the exception being the order in which the term_PUSH signal is propagated.

It can also be observed that if one writes the solution as:

FIGURE III-26. Dataflow graph for OPPE

path 1: (PUSH[n+term(POP)-auth(PUSH)>0], POP[term(PUSH)-auth(POP)>0]) end
then the dataflow graph synthesized by the preliminary algorithm is behaviorally identical to that of Figure VII-2 and appears in Figure III-27.
4. Predicate interference

The predicates in the OPPEs shown in Figs. III-25 and III-27 are "well behaved" in the sense that, when written in the form \( \text{arith}_\text{exp} > 0 \), each predicate has the following characteristics: (a) the subtractive term in the arithmetic expression is the number of authorizations of the predicated operation itself (which event occurs locally and is thus signalled immediately) and (b) the (non-constant) additive term is the number of terminations of an operation, signalled by a termination message from the access-mechanism. In fact, as pointed out in the comparisons with Figure VII-2, the PCs in these two examples behave as if they were PV-controllers. (This latter observation will be discussed later as a possible optimization in the synthesis of such "nice" predicates.)

However, the introduction of less "well-behaved" predicates into OPPEs soon presents some major, fundamental problems in our implementation environment with respect to communication of event information and the asynchrony of activity of the PCs. We begin by looking at the bounded stack problem with operations PUSH, POP, and TOP.

a. Example: the n-element stack (PUSH, POP, and TOP) Example 17 of Chapter II displayed Andler's PPE for this problem, and the discussion there focused on our inability to write an OPE to express the same synchronization constraints. The problem was that the predicate on TOP involved event counters for PUSH and POP but not for itself. This fact will be seen to be a stumbling block in our OPPE synthesis scheme as well.
Consider the following OPPE:

\[
\text{path 1: (PUSH} \text{[n+term(POP)-auth(PUSH)>0]}, \text{POP[term(PUSH)-auth(POP)>0]}, \text{TOP[term(PUSH)-auth(POP)>0]}) \text{ end}
\]

Using the preliminary algorithm of Section III.C.3 with this path expression, we would generate the dataflow graph shown in Figure III-28 (which will be shown to be incorrect).

In this graph, the input arc to PC3 labeled "auth_POP" has no source node. In Section III.C.3, we concluded that authorization signals cannot originate in the access-mechanism, so the originator must either be PC2 or the PV-controller at the bottom of the graph. We reject the latter from the standpoint of time delay: since the receipt of an auth_POP signal at PC3 could change the predicate from true to false, we would want it to be received as soon as possible in order to block any subsequent request for TOP at the earliest possible moment.

Unfortunately, even if we choose PC2 as the source of auth_POP, we end up with a timing problem. Suppose there is just one item on the stack, that there is no activity in the synchronization or access-mechanism submodules, and that there are no operations enqueued at the PV-controller. Now let requests for POP and TOP arrive simultaneously at their respective PCs. The POP will find its predicate true, so PC2

\[
\text{--------------------------}
\]

4 The reader will notice that an OPPE which more closely corresponds to Andler's PPE is:

\[
\text{path 1: (PUSH} \text{[n+term(POP)-auth(PUSH)>0]}, \text{(POP,TOP)[term(PUSH)-auth(POP)>0]}) \text{ end}
\]

However, we have chosen to first address the one given above and will then discuss this one.
FIGURE III-28. (Incorrect) Dataflow graph for OPPE path 1: (PUSH[n+term(POP)-auth(PUSH)>0], POP[term(PUSH)-auth(POP)>0], TOP[term(PUSH)-auth(POP)>0]) end
will allow the request to proceed, will increment the auth(POP) counter (thus changing the predicate from true to false), and will send an auth_POP message to PC3. However, if TOP tests its predicate at PC3 before the auth_POP arrives, it will also find its predicate to be true even though it has become false (in the eyes of a global observer). Thus, both requests will proceed down to the PV-controller. If the POP is selected by the PV-controller to proceed first, then upon its completion the TOP will proceed toward execution even though the stack is now empty.

The crux of this problem is that TOP tests its predicate at the same time that POP is, in a global sense, changing the value of that predicate. The obvious solution here is for requests for POP and TOP to be merged into the same PC so that the testing of the predicate is performed in mutual exclusion.

Instead of updating the translation algorithm at this point, we look at another OPPE for the stack problem which intrinsically eliminates the above problem—that mentioned in Footnote 4 of this chapter:

\[
\text{path 1:} \langle \text{PUSH}[n+\text{term(POP)}-\text{auth(PUSH)}>0], \text{POP, TOP}[\text{term(PUSH)}-\text{auth(POP)}>0] \rangle \text{ end}
\]

Here, the existing preliminary algorithm will cause requests for POP and TOP to be merged into the same PC due to the normal parsing of the subexpression \( \langle \text{POP, TOP}[\text{term(PUSH)}-\text{auth(POP)}>0] \rangle \). The dataflow graph resulting from the translation is shown in Figure III-29.

With this graph, the previously-mentioned objection disappears. If
a request for POP arrives at PC2 first, the serialization of requests by the PC will cause the predicate to become false before a subsequent request for TOP is serviced—hence, the TOP request will correctly be blocked. Noting this, we could now update the translation algorithm to force both POP and TOP requests into the same PC when performing the synthesis on the OPPE of Figure III-28.

However, further examination of Figure III-29 reveals another major
timing problem. Suppose, again, that there is just one item on the
stack, that there is no activity in the synchronization or access-
mechanism submodules, and that no operations are enqueued at the PV-
controller. Now let a request for TOP arrive, followed by a request for
POP. The TOP will pass the PC with no change to the value of the
predicate, then POP will be authorized by the PC to continue, and the
predicate then becomes false. We now have two tokens proceeding down
the arc to the PV-controller, presumably in the same sequence as
authorized: first the TOP and then the POP.

However, there are at least two events which could result in the
POP actually being executed in the access-mechanism before the TOP.
First, the POP could overtake the TOP on the way to the PV-controller if
the communication subnetwork were not able to guarantee the usual
assumption that messages are received in the order sent. Second, we
make no assumption in our environment about the queuing mechanism at a
controller other than it be "fair". In particular, we do not assume a
FIFO order of enqueuing at the PV-controller; hence, even if TOP arrived
at the PV-controller before POP, they could be enqueued in reverse
order, leading to TOP seeing an empty stack on arrival at the access-
mechanism.

The root of this problem lies in the time delay between
authorization of an operation (by a controller in the synchronization
submodule) and its activation (by the access-mechanism submodule). This
problem does not arise in those implementation environments which use a
centralized granting of access to the resource object. There,
authorization is synonymous with activation. But in our implementation, this time delay must be accounted for. With our stack problem, once a TOP has been authorized by the PC to proceed, a subsequent request for POP must be blocked until the TOP request has reached the access-mechanism so that TOP's predicate does not change from true to false while the request is in transit to the access-mechanism.

In this example, POP will be said to interfere with TOP. We now proceed to define the concept of predicate interference, to prescribe a normal form for predicates as an aid in detecting such interference, and then to present a mechanism in the synthesis algorithm for dealing effectively with predicate interference.

b. Predicate interference and secondary predicates

**Definition:** Let operation X be predicated by P. Predicate interference exists if Y is an operation (Y≠X) such that an authorization of or a request for Y can change the value of P from true to false. Operation Y is then said to interfere with operation X. Furthermore, in this case X is called the dependent operation and Y is called the independent operation.

The OPPE of Figure III-29, then, contains predicate interference. In that example, POP interferes with TOP; also, TOP is the dependent operation and POP is the independent operation.

At the end of Section III.C.4.a, the point was made that, once a request for TOP has been authorized by its PC to proceed, it must be allowed to reach the access-mechanism without having its predicate changed from true to false (by the subsequent authorization of a POP
request while the TOP is in transit). The principle which arises is thus the following: when the dependent operation is authorized to proceed from the PC, a subsequent request for the independent operation must be blocked until the dependent operation has begun executing in the access-mechanism. To enforce this requires (a) a signal from the access-mechanism that it has accepted the operation request and (b) a secondary predicate on the independent operation which causes a request for the operation to be delayed until the PC receives such a signal from the access-mechanism.

For the former, we require the access-mechanism to be augmented so that it sends a feedback message—call it "act\_X" (for "activation of X")—to the synchronization submodule whenever it accepts and begins executing an operation-request message for operation X. The name "act" thus is strongly suggestive of the distinction in our environment between authorization and activation of an operation. Hence, for every operation X executed by the access-mechanism, two feedback messages are sent to the synchronization submodule: act\_X (sent at the beginning of execution) and term\_X (sent at the completion of execution). The secondary predicate on the independent operation will then be

\[ 1 + \text{act}(X) - \text{auth}(X) > 0, \text{ where } X \text{ is the dependent operation}. \]

Without describing yet how to automate the detection of predicate

\[ \text{------------------------} \]

\[ ^5 \text{It must be emphasized very strongly at this point that activation signals are intended to be hidden from the programmer who is writing OPPEs. Although activation signals are required for the implementation of OPPEs in our environment, the only event counters that are visible and available to the programmer are req}(X), \text{auth}(X), \text{and term}(X). \]
interference and the generation of secondary predicates, let us reconsider what we would want the synthesized graph of the OPPE of Figure III-29 to look like. The desired graph is shown in Figure III-30. In this graph appears a secondary predicate on POP, labeled with "g" (used henceforth to visually set off the secondary from the primary predicate). In the program implementation of such a PC, the predicate on a request for POP would simply be a logical and of its primary and secondary predicates. It should also be noted that in augmenting the access-mechanism to provide activation signals, it is intended that these signals are sent for all operations executed by the access-mechanism. The fact that "act_POP" and "act_PUSH" do not label any arcs in this graph is simply due to the fact that these messages are not needed as input to any controllers; hence, they are considered "grounded" upon receipt from the access-mechanism.

In comparing this graph to that of Figure III-29, one can see how the race condition arising in Figure III-29 is now prevented. Here, if there is just one item on the stack, no activity in the synchronization or access-mechanism submodules, and an empty queue at the PV-controller, the authorization of a TOP request will then cause POP's secondary predicate to become false immediately. Hence, a subsequent request for POP will be enqueued until the PC receives act_TOP from the access-mechanism. The POP would then be authorized to proceed (but would, of course, be blocked at the PV-controller until TOP has completed executing because of the separate specification of mutual exclusion by 1:() in the OPPE).
 FIGURE III-30. Dataflow graph for OPPE

path 1: (PUSH[n+term(POP)-auth(PUSH)>0],
(POP,TOP)[term(PUSH)-auth(POP)>0]) end

At this point, it might seem that the use of secondary predicates will inhibit concurrency in the system. The synthesis scheme so far has emphasized an implementation of synchronization in which tokens are piped through a network of independent, concurrently executing controllers with the goal of avoiding any unnecessary serialization of
operation requests. In many examples, this goal has been achievable, but in others it will not be. In the graph of Figure III-30, the concurrency of movement of POP and TOP requests must be limited (by means of the secondary predicate) in order to achieve a correct solution.

Thus, while it is true that the use of secondary predicates does reduce concurrency, this inhibition of concurrency is only with respect to the synchronization network—not the access-mechanism. That is, concurrent access to a resource object (meaning—for us—concurrent execution of operations within the access-mechanism submodule) is not infringed upon by the use of secondary predicates. The stack problem under discussion does not serve to exemplify this because all three operations are required to execute in mutual exclusion, so let us consider the following OPPE:

\[ \text{path } (A,B)[1+\text{term}(A)-\text{auth}(A)>0] \text{ end} \]

This path expression specifies that only one A may execute at a time, that any number of Bs may execute at a time but a B may not start if an A is currently executing, and that an A may overlap with any Bs currently executing. Here, there is predicate interference since an authorization of A can change the predicate on B from true to false while B is in transit. Thus, we should generate a secondary predicate on the independent operation A and produce the graph shown in Figure III-31.

Here, if a request for B is authorized by the PC to proceed and then a request for A arrives, the A will be blocked until B "reaches"
the access-mechanism. Thus, B's predicate cannot change from true to false while it is in transit. However, once act_B is accepted by the PC, the secondary predicate on A becomes true and the request for A will be released to the access-mechanism, where it can execute concurrently with B. Hence, although the requests for A and B had to be serialized to some extent in the synchronization network, no such serialization was imposed in the access-mechanism. (Although A did have to undergo some delay before being admitted to the access-mechanism to execute concurrently with B, this delay would be negligible if both were lengthy operations and executed for a long time in the access-mechanism.)

c. A normal form for predicates In the examples of Figs. III-26 through III-31, the predicates were written in the form arith_exp>0, from which form it was relatively easy to determine which operations performed P-like and V-like operations on the pred-controllers and to visually identify predicate interference. Automating
the detection of interference would also be simplified if all predicates (in our restricted form) could be transformed into such a normal form if the programmer had not written them so.

A relevant technique for obtaining a normal form for predicates is discussed in [Ford 1978]. In this paper, Ford uses predicates in the implementation of generalized critical regions, and these predicates are based on two implicit event counters for an operation X: the number of "activities" of X (i.e., the number of processes currently executing X) and the number of processes waiting to execute X. His predicates are, as are ours, restricted to boolean functions of linear relational expressions in integer constants and implicit event counters. The ensuing description of Ford's transformation technique has been adapted to our environment and notation.

First, transform each predicate \( P \) into an expression which contains only the operators and, or, and >. This is done by:

- reversing relational operators to eliminate not
- creating pairs of relational terms connected by and or or to eliminate = and \( \neq \)
- negating arithmetic expressions on both sides to eliminate <

The predicate is now in the form \( P = B(R_1, R_2, \ldots, R_m) \) where \( B \) is a boolean function without not operators and each \( R_i \) is a relational expression containing a single > operator. Next, transpose terms algebraically to rewrite each \( R_i \) in the form:

\[ R_i = \text{auth}(X) - \text{term}(X) \]

\[ R_i = \text{req}(X) - \text{auth}(X) \]

6 In our scheme, the first of these would be equivalent to auth(X)-term(X), and the second would be equivalent to req(X)-auth(X).
\[ R_i = c_i + \sum_{j=1}^{n} a_{ij} e_{ij} > 0 \]

where the constant term \( c_i \) and the coefficients \( a_{ij} \) are known constants and the \( e_{ij} \) are event counters \( \text{req}(X), \text{auth}(X), \text{and term}(X) \).

This straightforward transformation process can be performed at semantic analysis time as a prelude to detection of predicate interference. In fact, this detection now becomes almost trivially simple.

**Proposition:** Assume that operation \( X \) is predicated by \( P \), that \( Y \) is an operation such that \( Y \neq X \), and that \( a^*\text{auth}(Y) \) (or \( a^*\text{req}(Y) \)) is an arithmetic term in \( N(P) \), the normal form of \( P \). Then, \( Y \) interferes with \( X \) iff \( a < 0 \).

**Proof:** Let \( a^*\text{auth}(Y) \) (or \( a^*\text{req}(Y) \)) be a term in \( N(P) \). Then, this must be a term in at least one of the component relational expressions—say \( R_j \)—of \( N(P) \). Since there are no complement operators in \( N(P) \), \( R_j \) is then of the form \( \text{arith}_{\exp} > 0 \).

Now suppose the coefficient \( a < 0 \). Since the counters \( \text{auth}(Y) \) and \( \text{req}(Y) \) are monotonically increasing integers, the product \( a^*\text{auth}(Y) \) (or \( a^*\text{req}(Y) \)) is monotonically decreasing. Thus, an authorization of (or request for) \( Y \) can cause the value of the left-hand side of \( R_j \) to change from positive to non-positive—hence, \( R_j \) from true to false. Then, \( P \) may also change from true to false, so \( Y \) interferes with \( X \).

\---

\[ 7 \text{ This assumes, of course, that the programmer would not write such nonsense as } 3^*\text{auth}(Y) - \text{auth}(Y) > 0. \]
Conversely, suppose Y interferes with X. Then, an authorization of (or request for) Y must be able to cause the arithmetic expression of $R_j$ to decrease in value from positive to non-positive. But auth(Y) and req(Y) are monotonically increasing integers, so $a$ must be negative. This completes the proof.

Thus, the detection of predicate interference simply amounts to transforming all predicates into the specified normal form and then checking each subtractive term in a predicate on, say, X. If this subtractive term contains auth(Y) or req(Y) where $Y \neq X$, then there is predicate interference. When this interference is detected, a secondary predicate on the independent operation must then be inserted as in the examples in Figures III-30 and III-31.

The dilemma presented in Figure III-28 has still not been resolved, however. There, two distinct pred-controllers—one for POP and one for TOP—were generated during the translation process. When generating the PC for TOP, predicate interference will be detected, but the independent operation POP is not an input to this PC—hence, an incorrect solution will result unless the POP requests can somehow be merged into this PC. A solution to this problem is intrinsic in the final translation algorithm which will now be presented.

5. Final translation algorithm

We believe the following translation algorithm is general in the sense that it covers the generation and interconnection of PV-, burst-, and pred-controllers in all situations we have been able to anticipate. In particular, it deals with predicate interference in which requests
for the independent operation would not normally be input to the PC for the dependent operation. This is resolved by attaching special labels to appropriate arcs when predicate interference is detected during the translation process and then removing these labels at a later stage. The rationale for this scheme will be discussed after the algorithm is first presented in its entirety.

a. The algorithm The portions of this algorithm which specifically relate to pred-controllers are Rule 5.5 of Step 1, Rule 6a(iv) of Step 1, and part b of Step 2.

Step 1. The following six transformation rules are repeatedly applied in the order determined by a left-to-right, top-down parse of the OPPE according to the production rules in the OPPE grammar (Section II.C.1). Each of L and R represents a previously generated synchronization primitive—or sequence of primitives—on the left and right of the subexpression.

1. Replace path list end by null list null.
2. Replace L sequence,list R by L sequence R and L list R.
3. Replace L item;sequence R by L item V(sl)
   and P(sl) sequence R
   with semaphore sl initialized to 0.

Additionally, generate a PV-controller with an initial state consisting of a zero pv-counter and an empty pv-queue. Leave the primary input arc and both output arcs unlabeled for the
time being.

4. Replace \( L \langle n \rangle \langle \text{list} \rangle R \) by \( P(s_2) L \text{ list} R V(s_2) \)
   with semaphore \( s_2 \) initialized to \( n \).
   Additionally, generate a PV-controller with an initial state consisting of \( pv \)-counter initialized to \( n \) and an empty \( pv \)-queue. Leave the primary input arc and both output arcs unlabeled for the time being.

5. Replace \( L \{ \text{list} \} R \) by \( PP(c,s,L) \text{ list} VV(c,s,R) \)
   where \( PP \) and \( VV \) are defined as in Section II.B.1
   and initialize counter \( c \) to 0 and semaphore \( s \) to 1.
   Additionally, generate a burst-controller with an initial state consisting of a zero burst-counter, an empty burst-queue, and an "idle" phase designation. Leave the signal and operation-request output arcs unlabeled for the time being.

5.5. Replace \( L \text{ item}[\text{predicate}] R \) by \( W(\text{predicate}) L \text{ item} R \).
   Additionally, transform the predicate into normal form and generate a pred-controller with primary input arc and both output arcs unlabeled for the time being. The initial state will consist of integer event counters (all initially zero) corresponding to those stated in the predicate and an initially empty queue for waiting messages.

6. When \( L \text{ operation\_id} R \) is encountered, do the following:
   a. For each PV-, burst-, or pred-controller (in left-to-right order) corresponding to the semaphore(s) or \( W \) operator(s) named in \( L \), do:
(i) (PV- or burst-controller only) Merge into this controller's primary input arc an arc labeled "operation_id", and connect to this arc any identically labeled unconnected output arc.

(ii) (PV- or burst-controller only) Label this controller's operation-request output arc with "operation_id".

(iii) (Burst-controller only) For the primitive PP(c,s,L') which this controller implements, recursively apply Rule 6a (again, in left-to-right order) with L' replacing L and "start_burst" replacing "operation_id".

(iv) (Pred-controller only) Apply subalgorithms PCTRANS.1 (see Figure III-32), PCTRANS.2 (see Figure III-33), and PCTRANS.3 (see Figure III-34) in that order.

b. For each controller® (in right-to-left order) corresponding to the semaphore(s) named in R, do:

(i) Merge into this controller's primary input arc an arc labeled "term_operation_id".

(ii) Label this controller's signal output arc with "term_operation_id", and connect this output arc to any identically labeled unconnected input arc (which is input to a controller other than this one).

---------------------

® These will only be PV- and burst-controllers.
(iii) (Burst-controller only) For the primitive \( VV(c,s,R') \) which this controller implements, recursively apply Rule 6b (again, in right-to-left order) with \( R' \) replacing \( R \) and "term_burst" replacing "term_operation_id".

In substeps 6a(i), 6a(iv), and 6b(ii) the connection of the output (input) of an existing controller to the input (output) of a newly labeled controller may make it necessary to insert a distributor if the output arc is labeled with more than one message name.

With completion of the parse of the path expression, Step 1 is completed. All the controllers have now been generated, and a number of their interconnections will have been made. To complete the interconnections, Steps 2 through 4 are now applied (in that order).

**Step 2.**

a. Connect all unconnected operation-request output arcs labeled "start_burst" to identically labeled unconnected input arcs, and connect all unconnected burst-controller "term_burst" output arcs to identically labeled unconnected input arcs.

b. Connect any unconnected output arc labeled \( X' \) (labeled as such by application of subalgorithm PTRANS.2) to an unconnected input arc labeled \( X \). Then, starting with any unconnected input arc labeled \( X' \), "deprime" all arcs labeled \( X' \).
Step 3. Connect each operation-request output arc having no terminal node to the access-mechanism submodule (via a merge if more than one).

Step 4. Generate a distributor which accepts a merge of two message streams: (a) operation-request messages from the entry port to the resource module and (b) operation-termination messages from the access-mechanism submodule. Connect the distributor's output arcs either to identically labeled unconnected controller input arcs or directly to the access-mechanism (for those operation-request messages for which there are no corresponding unconnected input arcs).

Suppose that X is the operation_id in $L \text{operation_id} R$ and that $W(P)$ is the operator under consideration in $L$. Then proceed as follows:

if there is no input arc to this PC labeled $X'$ (due to a prior detection of interference)
then (1) merge in an input arc labeled $X$ and connect to it any unconnected output arc labeled $X$;
(2) tag the predicate with "X";
(3) label the operation-request output arc with $X$;
else (1) "deprime" the label $X'$ and connect to it any unconnected output arc labeled $X$;
(2) tag the predicate with "X";
(3) "deprime" all arcs labeled $X'$ on the path from this PC to that which has an unconnected output arc labeled $X'$ ("depriming" this last unconnected output arc also);

To "tag" the predicate with "X" means (a) at the dataflow graph level, to label the predicate with the operation name "X" and (b) at the program level, to bind the predicate to incoming request messages for operation X.

FIGURE III-32. Subalgorithm PCTRANS.1 of the final translation algorithm
Suppose that $X$ is the operation id in $L$ operation id $R$ and that $W(P)$ is the operator under consideration in $L$. Then proceed as follows:

if there is at least one subtractive term in $P$ containing $\text{auth}(Y)$ or $\text{req}(Y)$, where $Y \neq X$

then for each such $Y$
do

(1) add a secondary predicate $1 + \text{act}(X) - \text{auth}(X) > 0$ (unless it already exists) and tag it with "$Y"; 

(2) merge in an input arc labeled $\text{act}(X)$, unless it already exists (and if $X$ is a start burst request, then connect to this input arc the $\text{act}_{\text{start burst}}$ output arc from the corresponding burst-controller);$^1$

(3) label the signal output arc with $\text{act}(X)$ (unless it is already labeled as such) and connect it to any identically labeled unconnected input arc to another controller;

(4) if there is no input arc to this PC already labeled $Y$ or $Y'$

then (a) merge in an input arc labeled $Y'$ (but do NOT connect to this arc any unconnected output arc labeled $Y'$ from another controller);

(b) label the operation-request output arc with $Y'$ and connect it to any unconnected $Y'$ input arc to another controller;

else {do nothing—no interference exists};

for each subtractive term containing $\text{term}(Z)$, where $Z$ is any operation including $X$
do

(1) merge in an input arc labeled $\text{term}(Z)$, unless it already exists;

(2) label the signal output arc with $\text{term}(Z)$ (unless it is already labeled as such) and connect it to any identically labeled unconnected input arc to another controller;

$^1$ It will be seen in Section III.C.6 that the activation signal for a "start burst" must originate at a burst-controller, not at the access-mechanism.

FIGURE III-33. Subalgorithm PCTRANS.2 of the final translation algorithm
Suppose that $X$ is the operation_id in $L$ operation_id $R$ and that $W(P)$ is the operator under consideration in $L$. Then proceed as follows:

```plaintext
for each additive term $a\times_{\text{event}} \text{ctr}$ in $P$ do
    case $\text{event}_\text{ctr}$ of
        $\text{req}(X)$ or $\text{auth}(X)$: {do nothing};
        $\text{term}(Z)$ (where $Z$ is any operation, including $X$):
            (1) merge in an input arc labeled $\text{term}_Z$, unless it already exists;
            (2) label the signal output arc with $\text{term}_Z$ (unless it is already labeled as such) and connect it to any identically labeled unconnected input arc to another controller;
        $\text{auth}(Y)$ (where $Y \neq X$):
            (1) change the term to $a\times_{\text{act}}(Y)$;
            (2) merge in an input arc labeled $\text{act}_Y$, unless it already exists;
            (3) label the signal output arc with $\text{act}_Y$ (unless it is already labeled as such) and connect it to any identically labeled unconnected input arc to another controller;
        $\text{req}(Y)$ (where $Y \neq X$):
            (1) merge in an input arc labeled $Y$ (unless it already exists) and connect to it any unconnected output arc labeled $Y$ from another controller;
            (2) label the operation-request output arc with $Y$;
```

FIGURE III-34. Subalgorithm PCTRANS.3 of the final translation algorithm
b. **Rationale for the X' labeling**

To describe the reasoning behind "priming" and "depriming" certain arcs in the final translation algorithm, we will present two OPPEs and show how the difficulties in synthesizing them led to the incorporation of this mechanism.

**Example 1**

Consider the following OPPE:

\[ \text{path } 1: (A, B, C[\text{term}(A) - \text{auth}(B) > 0]) \text{ end.} \]

The translation tree for this path expression has three leaves: \( P(s1) A V(s1) \), \( P(s1) B V(s1) \), and \( W(\text{term}(A) - \text{auth}(B) > 0) P(s1) C V(s1) \). Let us assume that the first two leaves have been processed in the normal fashion to yield a PV-controller with appropriate input and output arcs. Let us also assume that the final translation algorithm is not yet available and that we are attempting to process the W operator. Since the subtractive term in the predicate on C is auth(B), we have predicate interference and must introduce a secondary predicate (on B) into the pred-controller:

\[ 1 + \text{act}(C) - \text{auth}(C) > 0. \]

But B is not an input into this PC, so somehow requests for B must be forced into the PC just to cope with the interference—see dashed line in Figure III-35. In bringing the B input arc into the PC, suppose we were to connect the corresponding output arc from the PV-controller to it as we do with consecutively encountered PV-controllers. The resulting graph would be that shown in Figure III-36.

However, this solution will induce deadlock in the following case: if B passes the PV-controller and is enroute to the PC at the same time C passes the PC, then B and C will be permanently blocked at the PC and PV-controllers, respectively. The problem here is that B should be routed through the PC as early as possible so that it does not undergo
FIGURE III-35. Partial dataflow graph for OPPE
path 1:(A,B,C[term(A)-auth(B)>0]) end

any presynchronization first. That is, nothing in the path expression
suggests that B is subject to the predicate—it is to be forced into the
PC only to deal with the interference. Thus, for this problem we want
to route B from the entry port into the PC first and then send it to the
PV-controller—see Figure III-37.

Thus, on the basis of this example (and others similar to it) we
might (temporarily) conclude the following: when the independent
operation is forced into a PC due to interference, we should connect its
output from the PC to a corresponding unconnected input arc to another
To access-mechanism

FIGURE III-36. (Incorrect) Dataflow graph for OPPE
path 1: (A,B,C[term(A)-auth(B)>0]) end
FIGURE III-37. Dataflow graph for OPPE

path 1: (A, B, C[term(A)-auth(B)>0]) end
Example 2 Consider the following OPPE:

\[ \text{path } n:(A;B), (A,B)[1+\text{term}(B) - \text{auth}(B)>0] \text{ end.} \]

The translation tree for this path expression has four leaves: \( P(s1) A V(s2), P(s2) B V(s1), \)
\( W(1+\text{term}(B) - \text{auth}(B)>0) A, \) and \( W(1+\text{term}(B) - \text{auth}(B)>0) B. \) When the third leaf is processed and the predicate interference is detected, the graph constructed so far will appear as in Figure III-38. If we then apply the conclusion from Example 1 above—namely, to bring in a new arc for \( B \) and send the output back to the \( B \) input of the right-hand PV-controller—the graph will appear as in Figure III-39.

Then, in processing the fourth leaf of the translation tree, we have an explicit input of \( B \) to the PC. So the unconnected \( B \) output from the right-hand PV-controller would be merged into the PC as shown in Figure III-40. However, this produces a nonsensical solution because of the cycle in the graph. Instead, the appropriate action (rejected in Example 1) would be the more familiar one of connecting the existing output arc from the right-hand PV-controller to the newly-drawn input arc of the PC in Figure III-38. Doing so would yield the correct graph shown in Figure III-41.

Conclusion When interference is detected and a new input arc for the independent operation is drawn, there are some situations in which a corresponding unconnected output arc should be connected to it and other situations in which it should not. In the latter case, the PC's output arc for the independent operation should eventually be routed back to a corresponding unconnected input arc.
The use of the X' label in the algorithm deals with these cases in the following way. When interference is detected with X being the independent operation, an input arc labeled X' is merged into the PC (but is not connected to any corresponding unconnected output arc). If a leaf is later encountered which causes X to explicitly enter this PC, then an unconnected output arc labeled X will be connected to this X' input arc and the prime symbol will be removed from this label (and from all further X' labels). This covers Example 2 above.

On the other hand, if X is never explicitly brought into this PC during the processing of all further leaves, then the unconnected X' output arc is to be connected to the unconnected X input arc at some controller, and all X' labels are then to be "deprimed". This ensures that a request for X will be sent from the synchronization submodule's input port directly to the PC to deal with the interference before it undergoes any further (explicit) synchronization. This covers Example 1 above.

The reader is encouraged to apply the final translation algorithm in detail to both of the above examples to verify that the dataflow graphs generated will be the same as those shown in Figures III-37 and III-41.
FIGURE III-38. Partial graph 1 for OPPE path n:(A;B), (A,B)[1+term(B)-auth(B)>0] end
FIGURE III-39. Partial graph 2 for OPPE

path n: (A; B), (A; B) [1+term(B) - auth(B) > 0] end
FIGURE III-40. (Incorrect) Dataflow graph for OPPE
path n: (A; B), (A, B)[1 + term(B) - auth(B) > 0] end
FIGURE III-41. Dataflow graph for OPPE
path n: (A;B), (A,B)[1+term(B)-auth(B)>0] end
6. Examples

Example 1. This first example will show the result of applying the final translation algorithm to the following OPPE:

\[
\text{path 1: (PUSH}[n+\text{term}(\text{POP})-\text{auth}(\text{PUSH})>0], \\
\text{POP}[\text{term}(\text{PUSH})-\text{auth}(\text{POP})>0], \\
\text{TOP}[\text{term}(\text{PUSH})-\text{auth}(\text{POP})>0]) \text{ end}
\]

This is the path expression whose incorrect dataflow graph was shown in Figure III-28. In Section III.C.4, it was concluded that requests for POP and TOP would have to be merged into the same pred-controller to avoid the situation in which a TOP request was testing its predicate at the same time that a POP request was, in a global sense, changing the value of the same predicate. It will now be shown how the translation algorithm causes this merging to take place.

The three leaves in the translation tree for the above OPPE are \(W(P_1)P(s_1)\text{ PUSH }V(s_1), W(P_2)P(s_1)\text{ POP }V(s_1),\) and \(W(P_3)P(s_1)\text{ TOP }V(s_1),\) where \(P_1, P_2,\) and \(P_3\) denote the three predicates in the order in which they appear in the OPPE. In applying the algorithm to the first two leaves, subalgorithms PCTRANS.1 and PCTRANS.3 will be applied in a straightforward way, and the then-clause of PCTRANS.2 will not be executed since no predicate interference will be detected. After processing these two leaves, the graph will then appear as in Figure III-42.

In processing the third leaf, subalgorithm PCTRANS.1 will be applied with nothing unusual happening, but the application of PCTRANS.2 will result in the detection of predicate interference. In particular,
FIGURE III-42. Partial graph 1 for OPPE

path 1:(PUSH[n+term(POP)-auth(PUSH)>0],
POP[term(PUSH)-auth(POP)>0],
TOP[term(PUSH)-auth(POP)>0]) end
step (4) of the then-clause of PCTRANS.2 will cause (among other actions) an input arc labeled POP' to be merged into TOP's pred- controller PC3 and a secondary predicate on POP to be generated. PCTRANS.3 will then be applied with nothing unusual happening since term_PUSH is the only additive term in the primary predicate. Upon completion of the parse (hence, Step 1 of the translation algorithm), the graph will appear as in Figure III-43.

Finally, Step 2 of the algorithm will cause the unconnected output arc labeled POP' to be connected to the unconnected input arc of PC2, and all arcs labeled POP' will then be "deprimed". The resulting final graph is shown in Figure III-44.

Note that because PC3 will sequence requests for POP and TOP, the authorization of a POP request will cause a subsequent request for TOP to be blocked—that is, TOP cannot find its predicate to be true at the same time POP is changing it to false as in Figure III-28.

As a final comment on this example, it should be noted that, although the predicates on POP and TOP are identical, a simple parse of the path expression cannot determine this. A simpler graph would be synthesized if the programmer were to write the OPPE with a single predicate on the group (POP, TOP). The dataflow graph for such an OPPE was shown in Figure III-30, and, although that graph was presented before the final algorithm was specified, the reader is encouraged to work through the algorithm to verify that it produces exactly that graph.
FIGURE III-43. Partial graph 2 for OPFE
path 1: (PUSH[n+term(POP)-auth(PUSH)>0],
POP[term(PUSH)-auth(POP)>0],
TOP[term(PUSH)-auth(POP)>0]) end
FIGURE III-44. Dataflow graph for OPPE

path 1: \( (\text{PUSH}[n+\text{term}(\text{POP})]-\text{auth}(\text{PUSH})>0], \)
\( \text{POP}[\text{term}(\text{PUSH})-\text{auth}(\text{POP})>0], \)
\( \text{TOP}[\text{term}(\text{PUSH})-\text{auth}(\text{POP})>0]) \) end
Example 2     This example is presented to demonstrate the effect of applying the else-clause of subalgorithm PCTRANS.1. Consider the following OPPE for a simple stack with operations PUSH and POP only:

\[
\text{path } \text{PUSH}[n+\text{term(POP)}-\text{auth(PUSH)}>0], \\
\text{POP}[\text{term(PUSH)}-\text{auth(POP)}>0], \\
(\text{PUSH,POP})[1+\text{term(PUSH)}+\text{term(POP)}-\text{auth(PUSH)}-\text{auth(POP)}>0] \text{ end}
\]

With P1, P2, and P3 denoting the three predicates in the order written, the translation tree for this path expression contains four leaves: W(P1) PUSH, W(P2) POP, W(P3) PUSH, and W(P3) POP. The translation algorithm will process the first two leaves in a straightforward manner, but in processing the third leaf, predicate interference will be detected. Hence, among other actions, subalgorithm PCTRANS.2 will cause an input arc labeled POP' to be merged into the PC and a secondary predicate on POP to be generated. Figure III-45 shows the graph as it will appear after the third leaf has been processed.

In processing the fourth leaf, the else-clause of PCTRANS.1 will be invoked. Thus, the unconnected output arc labeled POP will be connected to the unconnected input arc to PC3 labeled POP', and all prime symbols will be removed from the POP' labels. Furthermore, subalgorithm PCTRANS.2 will determine that PUSH interferes with POP and will thus cause another secondary predicate (this time on PUSH) to be generated. However, at step (4) of this subalgorithm, the then-clause will not be executed because PC3 already has an input arc labeled PUSH. Thus, no input arc labeled PUSH' will be created. The final dataflow graph is shown in Figure III-46.
FIGURE III-45. Partial dataflow graph for OPPE path \[ \text{PUSH}[n+\text{term}(\text{POP})-\text{auth}(\text{PUSH})>0], \] \[ \text{POP}[\text{term}(\text{PUSH})-\text{auth}(\text{POP})>0], \] \[ (\text{PUSH},\text{POP})[1+\text{term}(\text{PUSH})+\text{term}(\text{POP})-\text{auth}(\text{PUSH})-\text{auth}(\text{POP})>0] \] end
FIGURE III-46. Dataflow graph for OPPE
path PUSH[n+term(POP)-auth(PUSH)>0],
POP[term(PUSH)-auth(POP)>0],
(PUSH,POP)[1+term(PUSH)+term(POP)]
-auth(PUSH)-auth(POP)>0] end
It is useful to compare this graph with that of Figure III-27 (in which a PV-controller replaces the lower pred-controller PC3) and that of Figure VII-2 of Appendix B (in which the graph contains only PV_controllers). The topologies of all three graphs are essentially identical because pred-controllers PCI and PC2 in Figure III-46 happen to behave exactly like PV-controllers, and pred-controller PC3 behaves like a PV-controller except for the additional activation signals and secondary predicates.9

Example 3 This example is presented to demonstrate the effect of subalgorithm FTRANS.3 when an additive term in a predicate contains auth(Y), where Y is not the operation name currently under consideration in the parse. The following OPPE is for the writers' priority variant of the readers-writers problem:

\[
\text{path 1: ( } \{ \text{READ}[^{req(WRITE)-auth(WRITE)=0}] \text{, WRITE) end}\]
\]

The predicate on READ (i.e., that the number of waiting writers is zero) is not in normal form as written. However, during the parse of this path expression, Rule 5.5 of Step 1 of the synthesis algorithm will have caused the predicate to be transformed such that the OPPE will logically become:

\[
\text{path 1: ( } \{ \text{READ}[^{l+auth(WRITE)-req(WRITE)>0}] \text{, WRITE) end}\]
\]

With P1 denoting the above transformed predicate, the translation tree for this path expression has two leaves:

-----------------------

9 These secondary predicates turn out to be essentially superfluous in this case because the primary predicate enforces mutual exclusion anyway!
W(Pl)PP(c,s2,P(sl)) READ VV(c,s2,V(sl)) and P(sl) WRITE V(sl). In processing the W operator of the first leaf, subalgorithm PCTRANS.1 will first be applied in a straightforward manner. Then PCTRANS.2 will detect predicate interference since the subtractive term in the predicate is req(WRITE). Hence, among other actions, an input arc labeled WRITE' will be merged into the PC and a secondary predicate on WRITE will be generated. When subalgorithm PCTRANS.3 is applied, the additive term of the primary predicate will be changed from auth(WRITE) to act(WRITE), and an input arc labeled act_WRITE will be merged into the PC. The rationale for this action will be presented shortly.

After the W operator has been processed, the treatment of the remaining PP and VV operators in the first leaf will be in the usual manner. Furthermore, processing of the second leaf is straightforward since it involves only P and V operators. Figure III-47 shows the partial graph as it will appear at the end of Step 1 of the synthesis algorithm.

Step 2 of the synthesis algorithm then causes (a) the unconnected start_burst output arc from the PV-controller to be connected to the corresponding input arc of the burst-controller, (b) the unconnected WRITE' output arc from the PC to be connected to the unconnected WRITE input arc of the PV-controller, and (c) all prime symbols to be removed from the WRITE' labels. Then, after Step 3 is applied, the resultant graph will be as shown in Figure III-48.10

------------------

10 The reader is encouraged to compare this graph—in terms of simplicity—with that of Figure VII-6 in Appendix B.
WRITE: 1 + act(WRITE) - req(WRITE) > 0
WRITE: 1 + act(READ) - auth(READ) > 0

start burst

burst (s2)

term burst

start burst

term burst

FIGURE III-47. Partial dataflow graph for OPPE
path 1: (READ[1 + auth(WRITE) - req(WRITE) > 0], WRITE) end
FIGURE III-48. Dataflow graph for OPPE

path 1: {{ READ[1+auth(WRITE)-req(WRITE)>0] }, WRITE} end
The reason for having PCTRANS.3 change the additive term of the primary predicate from auth(WRITE) to act(WRITE) is as follows. Suppose a burst of READ operations is in progress and a WRITE request arrives at the PC. In recognizing the WRITE request, the PC will cause the predicate on READ to become false. If no READ requests are in transit to the access-mechanism, the WRITE request will find its predicate to be true and will be authorized to proceed to the PV-controller (where it will be blocked until the readers all terminate). If the original additive term in READ's predicate—auth(WRITE)—had been retained, READ's predicate would then become true with the authorization of the WRITE request. Consequently, further READ requests would be allowed to proceed through the PC and the WRITE request could become starved at the PV-controller, contrary to the writers' priority requirement that a WRITE request block any further READ requests. Hence, in this case the reasonable interpretation is that READ requests should be blocked until the WRITE request actually begins executing—an event signalled by an act(WRITE) message from the access-mechanism.

The principle is thus the following: when auth(X) is in a subtractive term of a predicate (and hence performs a P-like operation on the PC), it represents a local authorization to proceed from the controller (thus potentially blocking any subsequent requests). On the other hand, when auth(X) is in an additive term of a predicate (and hence performs a V-like operation on the PC), it must be interpreted as the ultimate authorization to execute (thus potentially unblocking any waiting requests). Hence, for implementation purposes, auth(X) must be
internally changed to \textit{act(X)} if it is in an additive term.\textsuperscript{11}

\textbf{Example 4} This example is presented to clarify the reference to "act\textunderscore start\textunderscore burst" in step (2) of the then-clause of subalgorithm PTRANS.2. The following OPPE is for the alternating-burst variant of the readers\textendash writers problem (see Example 16 of Section II.B):

\begin{verbatim}
path 1: ( \{READ\}[req(WRITE)-auth(WRITE)=0], WRITE) end
\end{verbatim}

During the parse of this path expression, the predicate will be transformed (as in the last example) into normal form so that the OPPE logically becomes:

\begin{verbatim}
path 1: ( \{READ\}[1+auth(WRITE)-req(WRITE)>0], WRITE) end
\end{verbatim}

With P1 denoting the above transformed predicate, the translation tree for this path expression has two leaves:

\texttt{PP(c,s2,(W(P1);P(sl))) READ VV(c,s2,V(sl)) and P(sl) WRITE V(sl)}. In applying Step 1 of the translation algorithm to the first leaf, Rule 6a(iii) pertains because of the PP operator. So it will be the "start\textunderscore burst" pseudo-operation—not READ—which is subject to the synchronization sequence W(P1)P(sl). For the operation W(P1), subalgorithm PTRANS.2 will detect predicate interference since the subtractive term in the primary predicate is req(WRITE). Thus, an input arc labeled WRITE\textsuperscript{'} will be merged into the PC, and a secondary predicate 1+act\textunderscore (start\textunderscore burst)-auth\textunderscore (start\textunderscore burst)>0 will be generated for WRITE.

But where does the act\textunderscore start\textunderscore burst signal originate? The pseudo-

\textsuperscript{11} It must be stressed here that this alteration is internal only and is not visible to the programmer—he/she is unaware of the event counter \textit{act(X)} just as when secondary predicates are generated when predicate interference is detected.
operation "start_burst" is unknown to the access-mechanism—it is local to the synchronization submodule. One could choose to augment the access-mechanism to accept a start_burst request and just return it along with an act_start_burst signal to the synchronization submodule. But this approach would require the access-mechanism to respond to a request for an operation which does not access the resource object, contrary to our goal of logical separation of synchronization and access.

Instead we have chosen to augment the burst-controller in the following way. Upon return of the start_burst message (after it has made its way through any required presynchronization for the burst), the burst-controller will drop out an act_start_burst message on its signal output arc. Hence, it is the burst-controller, not the access-mechanism, which signals the actual commencement of the burst. With this augmentation, Figure III-49 shows the graph as it will appear after the first leaf has been processed. Notice that the additive term of the primary predicate has been changed from auth(WRITE) to act(WRITE) for the same reason as described in Example 3.

Then, the second leaf in the translation tree will be processed as usual, and Steps 2 and 3 of the synthesis algorithm will be applied, yielding the final graph shown in Figure III-50.\textsuperscript{12}

Finally, Figure III-51 displays the revised sequential language source code for a burst-controller. The revision consists of the

\textsuperscript{12} The reader is encouraged to compare this graph—in terms of simplicity—with that of Figure VII-7 in Appendix B.
FIGURE III-49. Partial dataflow graph for OPPE
path 1: ( {READ} [l+auth(WRITE)-req(WRITE)>0], WRITE) end
FIGURE III-50. Dataflow graph for OPPE
path 1: ( {READ} [{1+auth(WRITE)}-{req(WRITE)}>0] , WRITE) end
insertion of a single statement after line 18 of the original program (presented in Figure III-4)—namely, send (\texttt{act_start_burst}) to (\texttt{sig_dest}). Also, Appendix C shows the revised applicative language form for the burst-controller.

**Example 5** This last example is included to demonstrate the interconnection of \(Y'\) arcs in step (4) of the then-clause of subalgorithm PTRANS.2. Specifically, it exemplifies why part (a) of the then-clause specifies "do NOT connect to this arc any unconnected output arc labeled \(Y'\) from another controller" and why part (b) specifies that the \(Y'\) output should be connected to "any unconnected \(Y'\) input arc to another controller." Consider the following OPPE:

\begin{verbatim}
path A[1+term(C)-auth(C)>0], (B,C)[1+term(C)-auth(C)>0] end
\end{verbatim}

This OPPE specifies that only one instance of C may execute at a time and that any number of As and Bs may execute concurrently with each other (and with one C), but that no new request for A or B will be granted if a C is currently executing.\footnote{Since the predicate on each of A, B, and C is identical, one would clearly choose to write the path expression as: \begin{verbatim}
path (A,B,C)[1+term(C)-auth(C)>0] end
\end{verbatim} However, we have written it as shown above to demonstrate a point.}

With \(P_1\) and \(P_2\) denoting the two predicates in the above OPPE, the translation tree has three leaves: \(W(P_1)\) A, \(W(P_2)\) B, and \(W(P_2)\) C. When the first leaf is processed, C will be found to interfere with A and the graph produced will appear as in Figure III-52. Then, when the second leaf is processed, C will be found to interfere with B. Thus, at step (4) of PTRANS.2 an input arc labeled C' will be merged into the PC.
FIGURE III-51. Revised high-level program for a burst-controller
for predicate P2; however, as specified in the subalgorithm at this point, this input arc will not be connected to the unconnected C' output arc of PC1 in Figure III-52—instead the C' output arc from the current PC will be sent back and connected to the corresponding input arc of PC1. After processing the second leaf, then, the graph will appear as in Figure III-53.

![Diagram](image)

**FIGURE III-52.** Partial graph 1 for OPPE path A[1+term(C)-auth(C)>0], (B,C)[1+term(C)-auth(C)>0] end

In processing the third leaf, subalgorithm PTRANS.1 will discover that there is currently a C' input arc to pred-controller PC2. Thus, the else-clause of PTRANS.1 will be applied: the label C' on the input arc will be "deprimed", the primary predicate will be tagged with "C", and all further C' labels on arcs will be "deprimed". The resulting graph is shown in Figure III-54.

Note that the else-clause of PTRANS.1—applied as described in the
FIGURE III-53. Partial graph 2 for OPPE

path A[1+term(C)-\text{auth}(C)>0],
(B,C)[1+term(C)-\text{auth}(C)>0] end

previous paragraph—expects to find an unconnected $C'$ input arc to the
PC. It is for this reason that step (4) of the then-clause of PCTRANS.2
specifies that a newly created $Y'$ input arc is not to be connected to an
existing $Y'$ output arc—rather, the newly labeled $Y'$ output is sent back
to be connected to an unconnected $Y'$ input arc.
FIGURE III-54. Dataflow graph for OPPE

path A[1+term(C)-auth(C)>0],
(B,C)[1+term(C)-auth(C)>0] end
7. Deadlock and OPPEs

With OPPEs it is possible for the programmer to write expressions which will yield deadlock, regardless of a particular implementation. For example, consider rewriting the following OPPE (which is also an OPE) for a simple n-element stack:

```
path n:(PUSH;POP), 1:(PUSH,POP) end
```

by reversing the two main subexpressions:

```
path 1:(PUSH,POP), n:(PUSH;POP) end
```

Here, the semantics have changed both in an intuitive sense and in terms of the implementation. Because of the left-to-right rule with respect to repeated operation names, the latter OPPE will yield a deadlock state in the following scenario: if the stack is initially empty and the first request to arrive is one for POP, this request will be allowed through the first subexpression but will be blocked at the second because no PUSH has preceded it. However, a PUSH request cannot be granted by the first subexpression because the POP has not yet been executed. Thus, each request will be blocked until the other has executed—a classic impasse.

In a recent work [Campbell 1982], Campbell gives examples of other OPEs which can result in deadlock and employs a set of definitions, axioms, and invariants to support his claims. Examples are

```
path A;A end and path (A;B), (B;A) end.
```

It is always possible, then, for a programmer to write a "bad" path expression just as it is possible to write a "bad" program. With our proposed implementation of OPPEs, one would want to have confidence,
however, that the implementation was not responsible for producing
deadlock when the OPPE "intuitively" did not.\textsuperscript{14} To address this issue,
let us consider OPPEs which are written such that there is mutual
predicate interference—a highly visible potential source of deadlock.
With mutual interference, two questions should be asked: (a) Can our
synthesis scheme yield a deadlock-producing graph when the OPPE does not
suggest the possibility of deadlock? and (b) If the answer to (a) is
affirmative, will our synthesis scheme \textit{always} generate deadlock states
when mutual interference exists?

To answer the first question, consider the following OPPE:

\begin{verbatim}
path A[1+term(A)+term(B)-auth(A)-auth(B)>0],
B[1+term(A)+term(B)-auth(A)-auth(B)>0] end
\end{verbatim}

This OPPE is an alternative way of expressing the simpler OPPE

\begin{verbatim}
path 1:(A,B) end
\end{verbatim}

which specifies that As and Bs are to execute in mutual exclusion. The
OPPE contains mutual predicate interference: B interferes with A in the
first subexpression and A interferes with B in the second. However, in
looking casually at the path expression, one would not see an obvious
potential for deadlock. Our synthesis scheme will generate the graph
shown in Figure III-55 for this OPPE. Since the OPPE was written using
two distinct (though identical) predicates, two pred-controllers are
generated and deadlock will result in the following case: if requests

\textsuperscript{14} This is clearly a fuzzy area since the truth of the statement
"This OPPE cannot produce deadlock" is dependent on the programmer's
belief in the correctness of his/her path expression and on his/her
perception of the semantics of the path expression.
for A and B arrive at the two PCs and are authorized to proceed simultaneously, then each request will proceed to the other PC and find the value of its predicate to be false. Again, this is the classical standoff: neither can proceed until the other has executed.

Here, then, is an example in which our implementation has led to potential deadlock even though the path expression does not appear to imply this. Had the programmer written the OPPE with a single predicate:

\[
\text{path (}A,B\text{)[1+term(A)+term(B)-auth(A)-auth(B)>0] end}
\]

the potential for deadlock would not arise—see Figure III-56. Thus, it is an apparent shortcoming of our synthesis scheme that the implementation is highly sensitive to the form in which the path expression is written. However, it will be discussed in the next section that the detection of identical predicates will be a recommended optimization in the translation, in which case the graph in Figure III-56 would be generated even for the original OPPE.

Since the answer to the first question posed above is affirmative, we now address the second question—will the presence of mutual interference always lead to potential deadlock with our implementation? Consider the OPPE

\[
\text{path (}R,W\text{)[1+term(R)-auth(R)>0]}[1+term(W)-auth(W)>0] \text{ end}
\]

This path expression is an alternative specification of the weak readers' priority solution of the readers-writers problem—namely, a read request will be allowed provided no writer is writing, and a write request will be allowed provided no readers or writers are executing.
FIGURE III-55. Dataflow graph for OPPE
Path A[1+term(A)+term(B)-auth(A)-auth(B)>0],
B[1+term(A)+term(B)-auth(A)-auth(B)>0] end
FIGURE III-56. Dataflow graph for OPPE
path (A,B)[1+term(A)+term(B)-auth(A)-auth(B)>0] end

(The simpler OPPE here is, of course, path 1:({R},W) end.) In this OPPE, there is mutual interference as follows: R interferes with W with respect to the inner predicate, and W interferes with R with respect to the outer predicate. The dataflow graph produced by the synthesis algorithm is shown in Figure III-57.

We now give an informal argument that deadlock cannot result with this graph. For deadlock to occur, there are four possible cases.

Case 1: R and W are both permanently blocked at PC2.

For this to be true, it must be that both R and W are blocked by the primary predicate of this PC. (Note that it is impossible for W to be blocked by the secondary predicate of this PC—if an R had been authorized by PCI to proceed, a subsequent W would have been blocked at that PC until the R completed execution.) That is, both R and W are
waiting for the termination of an executing W. Barring a permanent crash of the access-mechanism, of course, the currently executing W will eventually complete and thus release one of R or W waiting at PC2. Thus, case 1 is impossible.

Case 2: R and W are both permanently blocked at PC1.

For this to be true, it must be that R is blocked because a previous W has not yet reached the access-mechanism and that W is blocked because a previous R has either not yet reached the access-mechanism or has reached it but has not yet finished executing. This
last event cannot permanently block the W (assuming the access-mechanism does not fail permanently), so it must be that both R and W are blocked because a previous R and W cannot reach the access-mechanism. Assuming no permanent failures in the communication links, this is only possible if the previous R and W are permanently blocked at PC2. But this was shown to be impossible in case 1. Hence, case 2 is impossible.

Case 3: R is permanently blocked at PC1 and W is permanently blocked at PC2.

Here, W must be blocked at PC2 because it is either waiting for a previous W to finish executing or because an R is in transit from PC2 to the access-mechanism. As in case 1, the latter is impossible; thus, W must be waiting for a previous W to complete execution. When that previous W does complete, then either an R or a W will be selected from PC2's waiting queue to proceed. For a W to be permanently passed by in this selection in favor of Rs, the waiting queue at PC2 must continually be replenished with new requests for R. But it was assumed that Rs are permanently blocked at PC1. Hence, there can be no replenishment of Rs in the queue for PC2, and the W must then eventually be selected to proceed. Hence, case 3 is impossible.

Case 4: W is permanently blocked at PC1 and R is permanently blocked at PC2.

The argument here is similar to that in case 3. R can only be blocked at PC2 because it is waiting for a W to finish executing in the access-mechanism. Once this termination occurs, PC2 will then select an R or a W from its waiting queue to proceed. But since it is assumed
that Ws are permanently blocked at PC1, only a finite number of Ws can be selected from the queue at PC2. Hence, a waiting R cannot be permanently blocked, and case 4 is impossible.

What we have shown, then, is that with our implementation, mutual predicate interference may lead to deadlock but does not imply deadlock. In doing so, it has also been shown (with the OPPE of Figure III-55) that our implementation can yield a potential for deadlock even though this potential is not obvious in the source path expression. A remedy for this class of path expressions will be discussed in the next section.

8. Suggested optimizations

In this section, we recommend two optimizations in the translation of OPPEs into dataflow graphs. These optimizations are intended to reduce the number of controllers and/or arcs in the dataflow graphs (hence, the number of nodes and/or the quantity of message traffic in the implementations of the graphs). Additionally, an important by-product of one of the optimizations is the reduction or elimination of the potential for deadlock in certain cases.

a. **Optimization 1** Wherever possible, replace multiple occurrences of a predicate in the OPPE by a single predicate on a grouped subexpression. For example, if A and B are operation names and P1 denotes a predicate, replace the subexpressions A[P1] and B[P1] by (A,B)[P1]. Two advantages will result from performing this optimization during the translation. First, the dataflow graph will be simplified by reducing the number of controllers and arcs in the graph. For example,
compare the graphs in Figures III-55 and III-56. Also, compare the graphs in Figures III-44 and III-30. The latter represents the graph which would be obtained with this optimization of detecting identical predicates. Second, this optimization can reduce or eliminate the likelihood of deadlock. This point was made clearly in the previous section with respect to the OPPEs of Figures III-55 and III-56.

This optimization could be performed by a preprocessor which would (a) transform all predicates into normal form and then (b) use a pattern-matching scheme to detect identical predicates. Thus, our synthesis algorithm would receive as input an OPPE containing only unique predicates.

b. **Optimization 2** Wherever possible, replace "PV-like" pred-controllers with PV-controllers. By a "PV-like" PC we mean one which has a primary predicate on operation X of the form \( n + \text{term}(Z) - \text{auth}(X) > 0 \), where Z is any operation (including X). Examples of such PCs are the two upper pred-controllers PCI and PC2 in Figures III-27 and III-46. Those two PCs could be replaced by PV-controllers initialized to n and 0, respectively. Regarding the lower pred-controller PC3 of Figure III-46, it is clear that it could be replaced by a PV-controller initialized to 1 because the secondary predicates are superfluous due to the mutual exclusion specified by the primary predicate. It is our conjecture that a PV-controller may substitute for any PC which contains a multiply-tagged predicate of the form
\[
X_1, X_2, \ldots, X_k : n + \text{term}(Z_1) + \ldots + \text{term}(Z_m) - \text{auth}(X_1) - \text{auth}(X_2) - \ldots - \text{auth}(X_k) > 0,
\]
even if there are secondary predicates present. However, further
research would be required to determine the generality of this conjecture.

This optimization would have to be performed at the end of the synthesis algorithm so that all predicate interference is assured of having been detected and treated and that all input and output arcs in the graph will have been generated and labeled. Hence, a postprocessor could be used to take as input a representation of all the controllers in the dataflow graph, determine which PCs could be replaced by PV-controllers, and then perform the substitutions. This would yield a more efficient synchronization submodule because, as will be seen in the next section, the code for a PC is more complex than for a PV-controller. Additionally, if our above conjecture about multiply-tagged predicates should turn out to be general, the elimination of certain secondary predicates would reduce the message traffic by eliminating the corresponding "act" messages.

9. High-level language representations of pred-controllers

Figures III-2, III-5, and III-51 contain Pascal-like program representations of the PV-controller, 4-way distributor, and burst-controller modules, respectively. These modules are general in the sense that specific operation names are not referenced in the program code. For example, the code for a PV-controller tests only the type field of an input message to determine whether it is an operation request or a termination signal. The name of the specific operation involved is immaterial; instead, it is the interconnection of the modules during the synthesis that determines the identities of the
various input operations. (At the program level, these interconnections are bound by instantiating each module with the names of appropriate destination modules.)

On the other hand, it is clearly insufficient for a program representation of a pred-controller to reference only the type of an input message. Since each predicate is bound to one or more specific operation names, each pred-controller must be customized so that it recognizes these names.

Figures III-58 and III-59 show programs in a Pascal-like sequential language notation for pred-controllers PC2 and PC1 of Figure III-30. The data type "wakers" enumerates event names corresponding to messages whose arrival can awaken operation requests which have been suspended on the PC's waiting queue. These event names will have been determined by application of subalgorithm PCTRANS.3 (Figure III-34) of the synthesis algorithm—namely by inspecting the additive terms in the normal form of the predicate. Then, for each of these event names there is a corresponding set of names of operations which it can awaken. These are represented by the array "can_wake_set".

Additional comments: the function "select_one", which selects the name of one of the operations to be potentially awakened, is to be written by the programmer so that an operation request can be dequeued from the waiting queue in random order or in some desired priority order. Hence, it is analogous to the procedure "insert" by which the programmer can specify a policy for enqueuing operation requests in the waiting queue. Also, the function "find" is assumed to return a pointer
to the first message in the waiting queue whose name field contains the string specified as the first argument. If none is found, the null pointer will be returned.

Although the programs in Figures III-58 and III-59 are presented as generic in that the destination modules would be specified at instantiation time, we have written them specifically for the two pred-controllers in Figure III-30. In an actual production implementation, one would want a single template for a pred-controller module at a higher level of genericism. Specifically, the data types "sleepers" and "wakers" and the array "can_wake_set" would be parametrized. Furthermore, the event counters and predicates would be specified as arrays, indexed by parametrized operation names. Hence, a specific instance of a pred-controller would be instantiated by supplying actual parameters for these data types and values. The details of this are essentially a programming exercise and are not included here.
generic (op_dest, signal_dest: module)
module pc_poptop;
type sleepers = (pop, top);
    wakers = (w_act_top, w_term_push);
var
    term_push, auth_pop, auth_top, act_top: integer init (0);
    queue, ptr: ↑queue; type;
    wakelist: set of sleepers;
    can_wake_set: array [wakers] of set of sleepers;
    wakeup: boolean;
    sleepy: sleepers;
    msg: message;
begin
    queue := createqO;
    can_wake_set[\text{w}_\text{act}_\text{top}] := \{\text{pop}\};
    can_wake_set[\text{w}_\text{term}_\text{push}] := \{\text{pop, top}\};
    while true do
        begin
            wakeup := false;
            receive(msg);
            case msg.name of
            'pop': if term_push-auth_pop>0 and 1+act_top-auth_top>0
                then begin
                    auth_pop := auth_pop+1;
                    send (msg) ^ (op_dest)
                end
                else insert(msg, queue);
            'top': if term_push-auth_pop>0
                then begin
                    auth_top := auth_top+1;
                    send (msg) to (op_dest)
                end
                else insert(msg, queue);
            'term_push': begin
                term_push := term_push+1;
                send (msg) to (signal_dest);
                wakeup := true;
                wakelist := can_wake_set[\text{w}_\text{term}_\text{push}]
            end;
            'act_top': begin
                act_top := act_top+1;
                send (msg) to (signal_dest);
                wakeup := true;
                wakelist := can_wake_set[\text{w}_\text{act}_\text{top}]
            end
        end {case};
    \{Continued next page\}

FIGURE III-58. High-level code for pred-controller PC2 of Figure III-30
if wakeup
  then while not emptyset(wakelist) do
    begin
      sleepy := select_one(wakelist);
      wakelist := wakelist - sleepy;
      case sleepy of
      pop; begin
        ptr := find('pop', queue);
        while ptr ≠ nil
          and term_push-auth_pop>0
          and 1+act_top-auth_top>0 do
          begin
            auth_pop := auth_pop+1;
            msg := delete(ptr, queue);
            send (msg) W (op dest);
            ptr := find('pop', queue)
          end {while}
        end;
      top; begin
        ptr := find('top', queue);
        while ptr ≠ nil
          and term_push-auth_pop>0 do
          begin
            auth_top := auth_top+1;
            msg := delete(ptr, queue);
            send (msg) to (op_dest);
            ptr := find('top', queue)
          end {while}
        end {case}
    end {while}
  end {while true}
end {pc_popoptop}

FIGURE III-58. (continued)
generic (op_dest, signal_dest: module)
module pc_push;
type sleepers = (push);
    wakers = (w_term_pop);
var
    term_pop, auth_push: integer init (0);
    queue, ptr: queue_type;
    wakelist: set of sleepers;
    can_wake_set: array [wakers] of set of sleepers;
    wakeup: boolean;
    sleepy: sleepers;
    msg: message;
begin
    queue := createq();
    can_wake_set[w_term_pop] := [push];
    while true do
        begin
            wakeup := false;
            receive(msg);
            case msg.name of
                'push': if n+term_pop-auth_push>0
                    then begin
                        auth_push := auth_push+1;
                        send (msg) to (op_dest)
                    end
                else insert(msg, queue);
                'term_pop': begin
                    term_pop := term_pop+1;
                    send (msg) to (signal_dest);
                    wakeup := true;
                    wakelist := can_wake_set[w_term_pop]
                end
        end {case};
    {Continued next page}
if wakeup
  then while not emptyset(wakelist) do
  begin
    sleepy := select_one(wakelist);
    wakelist := wakelist - sleepy;
    case sleepy of
      push: begin
        ptr := find('push', queue);
        while ptr ≠ nil
          and n+term_pop-auth_push>0 do
          begin
            auth_push := auth_push+1;
            msg := delete(ptr, queue);
            send (msg) to (op_dest);
            ptr := find('push', queue)
          end {while}
        end {case}
      end {while}
  end {while true}
end {pc_push}

FIGURE III-59. (continued)
IV. AN EXTENDED NOTATION FOR THE SYNCHRONIZATION SUBMODULE

Although path expressions in the major forms discussed in this work (RPEs, OPEs, PPEs, and OPPEs) are useful for expressing synchronization constraints for many types of problems, none of them is the ideal synchronization mechanism. Among the criteria proposed by Bloom [Bloom 1979] for evaluating synchronization mechanisms are (a) the ability to access the synchronization state of the resource and (b) the ability to specify dynamic priorities for operation requests according to arguments supplied with the requests.

The first of these can be dealt with in many cases by the use of predicates in path expressions. Specifically, the use of implicit counters req(X), auth(X), and term(X) in predicates provides the capability of expressing problem solutions in terms of constraints external to the unsynchronized resource object. However, there are problems for which even the use of such predicates in a path expression is inadequate and/or difficult to use. For example, problems involving certain distributed database management techniques require something more than a single path expression and are discussed in this chapter.

Regarding the second criterion, none of the major forms of path expressions has the ability to deal with request arguments. It is our opinion that this is a significant shortcoming of path expressions, and one of the purposes of the extended notation proposed in this chapter is to remedy this shortcoming.

It should be mentioned that the main thrust of our research has been presented in Chapter III. The material in Chapter IV represents
more of a proposal than a thoroughly worked out concept. High-level language notations are introduced, examples are presented, and implementation considerations are discussed; however, further research will be required to determine the appropriateness and generality of these language features and their implementation.

A. Description of the Notation

The primary goal here is to propose an extended notation for the synchronization submodule of a resource module. However, we also introduce a possible notation for the entire resource module (including the access-mechanism submodule) so that the synchronization submodule can be seen in its encompassing context. The notation represents a blend of constructs borrowed from synchronizers [Ramamritham and Keller 1980], Synchronizing Resources [Andrews 1981], and Ada [DoD 1980] along with OPPEs.

The overall structure of a resource module might be specified as:

```
resource sample;
operations
  A, B(pl,p2), C;
synchronizer
  {Specification of the synchronization submodule}
access-mechanism
  {Specification of the access-mechanism submodule}
end resource
```

The operations named in the operations section are those which are known
to the outside world and represent operations on the resource object (such as a database) enclosed by the access-mechanism. Any operation request bearing non-implicit parameters is indicated syntactically by the presence of a parameter list. It is envisioned, however, that the operation requests are received in the form of messages, and the explicit parameters are thus contained in message fields. Hence, in the program code for the synchronizer and/or access-mechanism, parameter \( p_1 \) of operation \( B \) above would be referenced as \( B.p_1 \).

Since it is intended that the synchronization and access-mechanism submodules are distinct and will communicate only through message passing when implemented, the scopes of any variables declared in those two sections are limited to the section in which they are declared.

The proposed structure of the synchronization submodule is:

```plaintext
synchronizer

state variables

{Declaration of variables local to the synchronizer}

state changes
A \to \text{statements;}
B \to \text{statements;}
C \to \text{statements;}

priorities

enqueue A by expression;
enqueue B by expression;

synchronization

path 1:(A, B, C[predicate]) end
```
The state variables are those representing the synchronization state of
the resource (for those problems needing more than the implicit counters
req(X), auth(X), and term(X)). The state changes are those changes to
any or all of the state variables which result from authorizations of
the various operations to proceed to the access-mechanism. The queuing
priorities are those which allow dynamic queuing of operation requests
or the selection among blocked operations of an operation to be
released. The expressions will generally involve synchronization state
variables and/or parameters supplied with the operation requests.
Lastly, the synchronization section is composed of an OPPE as described
in the earlier part of this work. It is intended that state variables
(and state changes) are needed only if (a) the OPPE uses them in
predicates and/or (b) any queuing priorities are specified.

The access-mechanism submodule might appear as:

```plaintext
access-mechanism

var

{Declaration of variables local to the access-mechanism}

entries

A: statements;
B: statements;
C: statements;
```

The `entries` section specifies what actions are to be performed for each
operation request which is received by the access-mechanism.
B. Examples of Usage

1. Disk scheduler

The first example is that of a disk scheduler which uses an elevator algorithm [Hoare 1974]. With this scheme, disk I/O requests are serviced according to the current direction of movement of the disk head. The request which has a destination cylinder nearest the current position in that direction will be serviced first. If there are no further requests in the current direction, the head changes direction and requests are serviced as it sweeps in the new direction. It is clear that if several requests for disk I/O are pending, they must be dequeued in priority order according to the requested cylinder and the current position and direction of the head. In Figure IV-1, which shows the disk scheduler resource module in our extended notation, the function "sign" is assumed to be an intrinsic function where sign(x)=1 if x>0 and sign(x)=-1 if x<0. Also, procedure "perform_IO" is assumed to perform the actual reading or writing of the disk and the details are not relevant here.

2. Alarm clock

The alarm clock resource [Hoare 1974] allows a process to suspend itself for a desired number of ticks of a "clock", after which it will be awakened by the alarm. The "clock" is just an integer counter contained in the resource which is updated at regular intervals by receipt of a TICK message, presumably as the result of a hardware interrupt somewhere. A process requests a wakeup by sending to the
resource disk_scheduler;

operations
    DISKIO(cyl, info);

synchronizer
    state variables
        dir: (up, down) init (up);
        posn: (1..maxcyl) init (1);
    state changes
        DISKIO \rightarrow \text{case dir of}
            up: if DISKIO.cyl < posn then dir := down;
            down: if DISKIO.cyl > posn then dir := up
        end {case};
        posn := DISKIO.cyl;
    priorities
        def diff = DISKIO.cyl - posn;
        dequeue DISKIO by \text{case dir of}
            up: abs(diff) - maxcyl*sign(diff);
            down: abs(diff) + maxcyl*sign(diff)
        end {case};
    synchronization
        path 1:(DISKIO) end;

access-mechanism
    entries
        DISKIO: perform_IO(DISKIO.info)
    end resource

FIGURE IV-1. Disk scheduler in extended notation
alarm clock resource a WAKE_ME message, supplying an argument n. It
will then receive a WAKEUP message from the alarm clock after n ticks of
the clock. The alarm clock resource in our extended notation is shown
in Figure IV-2. The resource object enclosed by the access-mechanism
essentially consists of two items: the clock (represented by the
variable "time") and a queue of wakeup times and requesting node
identifiers, ordered by wakeup time. The latter is an explicit queue as
opposed to implicit queues used at various points in the network (and by
synthesized modules such as PV- and pred-controllers) for waiting
messages.

3. Ellis's centralized solution for updating distributed databases

This example [Ellis 1977] involves the updating of fully replicated
databases in a network in which each node has a database copy guarded by
a database manager process (DBMP). One of the nodes contains a
supervisor process which dispenses updating privileges to all nodes. A
DBMP is the only process which can update the database at that node, and
it can communicate with all other DBMPs and the supervisor node.

When a user wishes to update a copy of the database, he/she sends
an internal update request (INT REQ) to his/her DBMP. Upon receipt of
this request, the DBMP sends an external request (EXT REQ) to the
central supervisor to obtain permission to perform the update. If
another DBMP currently has authorization to perform an update, the
supervisor will respond with a negative acknowledgement (ACK-); the DBMP
then sends REJECT to the user, requesting him/her to try again later.
However, if no other node is updating, the supervisor will grant
resource alarm_clock;

operations
  TICK, WAKE_ME(n);

synchronizer
  state variables {none};
  state changes {none};
  priorities {none};
  synchronization
  path 1:(TICK, WAKE_ME) end;

access-mechanism
  type clockq = record
    waketime: integer;
    originator: node_id
  end;
  var time: integer init (0);
  waketime: integer;
  queue: clockq init (nil);
entries
  WAKE_ME: waketime := time + WAKE_ME.n;
    priority_insert(waketime, WAKE_ME.source, queue);
    {Enqueue by wakeup time}

  TICK: time := time + 1;
    while not empty(queue) and time = head(queue).waketime do
      begin
        send (WAKEUP) to (head(queue).originator);
        dequeue(queue)
      end
end resource

FIGURE IV-2. Alarm clock in extended notation
permission to this DBMP by sending a positive acknowledgement (ACK+), and the update information is then broadcast to all other nodes. After each node has completed the update, it sends a done acknowledgement (ACKd) to the originating DBMP. When all n-1 ACKds (in a network of n DBMPs) have been received by the originating DBMP, it sends a DONE message to the user and to the supervisor so that the supervisor can give another node permission to update.

Figure IV-3 shows an evaluation net which is essentially that presented in [Ellis 1977] for this centralized control solution. An evaluation net is a variant of a Petri net in which circles denote states of the process, squares denote incoming message queues, and horizontal lines denote transitions. A transition "fires" if tokens are present on all its input arcs, at which point the tokens are removed from the input arcs and a token is delivered to each of its output locations. A hollow arrowhead on a transition denotes the side effect of sending a message. In Figure IV-3, there is initially a token in the circle marked "passive", which implies a token is present on each of its output arcs. Thus, the arrival of an INT REQ would cause transition T1 to fire (leading to the state "active") or the arrival of an UPD message would cause T5 to fire and the state would return to "passive".

Ellis's paper does not display an evaluation net for the supervisor process, but from the written description we have created the net shown in Figure IV-4. The representation of this supervisor in our extended notation is quite straightforward and is shown in Figure IV-5. The synchronizer simply enforces the mutual exclusion of operations EXTREQ.
FIGURE IV-3. Evaluation net—DBMP in Ellis's centralized control solution
FIGURE IV-4. Evaluation net—supervisor in Ellis's centralized control solution

```
resource supervisor;

operations
  EXTREQ, DONE;

synchronizer
  state variables {none};
  state changes {none};
  priorities {none};
  synchronization
    path 1: (EXTREQ, DONE) end;

access-mechanism
  var updating: boolean init (false);
  entries
    EXTREQ: if updating
          then send (ACK-) to (EXTREQ.source)
          else begin
                send (ACK+) to (EXTREQ.source);
                updating := true
          end;
    DONE: if updating
          then updating := false

end resource
```

FIGURE IV-5. Ellis's centralized supervisor in extended notation
and DONE, and the resource object enclosed by the access-mechanism is a single item: the boolean variable "updating".

In attempting to express the DBMP in our extended notation, numerous problems were encountered. A major problem was the allocation of state information between the synchronizer and the access-mechanism. In this problem, the synchronizer needs to know the state (passive, active, or updating) so that it can implicitly enqueue messages received at the "wrong" time, and the access-mechanism needs to know the state so that it knows what actions to take. (The latter is vitally important in Ellis's decentralized solution presented as the next example because, for instance, the receipt of an UPD message is valid in three distinct states of the algorithm.) Furthermore, writing a path expression to enforce the correct synchronization of message processing became confounded by an initial view of the DBMP and the database as being contained in a single resource module. That is, the problem became one of determining what the path expression was guarding: the primary database or the state information or both.

After trying alternative approaches, we came to the following conclusions. First, the DBMP and the database should be viewed as two distinct resource modules. With this view, synchronization of the various messages involved in the update protocol is separated entirely from synchronization of operations on the database proper. Secondly, with regard to the DBMP (or to any resource module in which both the synchronizer and the access-mechanism need access to common or similar state information), it is preferable to make the access-mechanism as
"dumb" as possible. This is only a conjecture on our part and may depend on the particular problem at hand, but the reason we conclude this is that there is a one-way flow of explicit information: from the synchronization submodule to the access-mechanism (disregarding the implicit termination and activation feedback signals from the access-mechanism to the synchronizer in our synthesized implementation). The choice then is between employing redundant state information in both submodules or keeping the state information only in the synchronizer, in which case the synchronizer must signal the access-mechanism when a change of state has occurred which requires some action by the access-mechanism. We have opted for the latter in writing the DBMP resource module by introducing the following notational feature: an operation name may be qualified with a suffix $<\text{identifier}>$ when it is sent from the synchronizer to the access-mechanism. The access-mechanism will then treat this as an operation distinct from the unqualified name but will return an unqualified termination signal (based only on the prefix) so that the synchronization enforced by the synthesized path expression will not be affected. This feature appears in the DBMP resource module of Figure IV-6 where the operation name ACKd is changed to ACKd$\text{ALL}$ by the synchronizer when all $n-1$ ACKds have been received. The access-mechanism will then, upon processing this message, send DONE messages to the user and the supervisor and then (implicitly) return the feedback signal term_ACKd to the synchronizer.

Figures IV-6 and IV-7 show the DBMP and database resource modules as written in our extended notation. The latter is straightforward as
resource DBMP_centralized;

operations
  UPD(upd_info), INTREQ(upd_info), ACK+, ACK-, ACKd;

synchronizer
  state variables
    state: (passive, active, updating) init (passive);
    c: (0..n-1) init (0);
  state changes
    UPD -> {none};
    INTREQ -> state := active;
    ACK+ -> state := updating;
    ACK- -> state := passive;
    ACKd if c < n-1
          then c := c+1
          else begin
            msg.name := ACKd$ALL;
            c := 0;
            state := passive
          end;

  priorities {none};

synchronization
  path 1: ((UPD,INTREQ)[state=passive],
           (ACK+,ACK-)[state=active],
           ACKd[state=updating] ) end;

access-mechanism
  var save_info: db update info;

entries
  UPD: DO UPDATE.upd_info := UPD.upd_info;
       send (DO UPDATE) to (Ellis_database);
       send (ACKd) to (UPD.source);

  INTREQ: save_info := INTREQ.upd_info;
          send (EXTREQ) to (supervisor);

  ACK-: send (REJECT) to (user);

  ACK+: DO_UPDATE.upd_info := save_info;
       send (DO UPDATE) to (Ellis_database);
       UPD.upd_info := save_info;
       broadcast (UPD);

  ACKd: {nothing};

  ACKd$ALL: send (DONE) to (user);
            send (DONE) to (supervisor);

end resource

FIGURE IV-6. DBMP resource module—Ellis's centralized control solution
resource Ellis_database;

operations
  DO_UPDATE(upd_info);

synchronizer
  state variables  {none};
  state changes   {none};
  priorities      {none};
  synchronization
    path 1:(DO_UPDATE) end;

access-mechanism
  var
    {Database variables}
  entries
    DO_UPDATE: perform_update(DO_UPDATE.upd_info);
end resource

FIGURE IV-7. Database resource module—Ellis's centralized control solution

its resource object is simply the pure database and the synchronizer thus needs only to exclude simultaneous updates to it. With respect to the DBMP resource module, we have used predicates in the OPPE which are not of the restricted form described earlier in this work. Interference analysis with general boolean predicates has not been discussed up to this point and is an area for further investigation. In terms of implementation, an interference analysis on this example would presumably cause at least some of the operation requests to be merged into a common pred-controller. Even in the worst case (i.e., one which permits the least amount of concurrent movement of requests through the synchronization subnetwork), all of the requests could be funneled into a single "state-controller" to serialize the changes to state variables. However, this serialization would have no impact on the concurrency of
operations (if any) permitted in the database resource itself, which is where concurrency is most desirable.

4. **Ellis's decentralized solution for distributed databases**

With this solution [Ellis 1977], Ellis assumes the same environment as in the previous solution except that there is no centralized supervisor node. Instead, a DBMP broadcasts EXT REQ messages to all other DBMPs and will proceed to an updating state if no consistency conflicts arise. (In this solution, simultaneous updates are allowed if, for example, the write sets of two updates do not intersect.) The DBMP waits, hoping to receive n-1 positive acknowledgements (ACK+) from the n-1 other nodes. If any node responds with an ACK-, the attempt is aborted and REJECT is sent to the user. However, if all n-1 ACK+ messages are received, the DBMP performs the update and broadcasts UPD to all other nodes. As in the centralized algorithm, the DBMP then waits until a done acknowledgement (ACKd) is received from all n-1 nodes and then sends DONE to the user and returns to the passive state.

Figure IV-8 shows an evaluation net for this decentralized solution which is somewhat simplified from that presented by Ellis. Transitions T1 and T2 assume the presence of an attempt number which is contained in each EXT REQ and acknowledgement message. When the user originates an update request, this attempt number is initially one. If the request is REJECTed at some point and another try is attempted, the attempt number is incremented by one. Hence, transitions T1 and T2 use the attempt number to determine if the acknowledgements are current or out of date (so that an acknowledgement arriving late from a previous attempt will
FIGURE IV-8. Evaluation net—DBMP in Ellis's decentralized control solution
not cause the current request to be accepted or rejected erroneously).

The test at transition T3 is made to determine whether to accept or reject an EXT REQ from another node. If the write sets of the EXT REQ and this DBMP's pending update request do not intersect, then the BOTH arc is appropriate—i.e., both updates can proceed simultaneously. However, if there is a consistency conflict, then it must be determined whether the EXT REQ has a higher priority than this DBMP's request or vice versa. This determination of priority is dependent upon the particular database structure and is left unspecified here.

Figure IV-9 contains the DBMP resource module written in our extended notation. Since UPD and EXT REQ can be processed in different states and the actions to be performed by the access-mechanism are dependent on the state, the synchronizer (which contains all the state information) uses the technique introduced in the last example of qualifying the operation name before sending it to the access-mechanism. For example, if the access-mechanism receives a request for ACK-, it does nothing; but if it receives ACK-$CURRENT, it sends REJECT to the user. In either case, our implementation would have the access-mechanism implicitly return a term_ACK- signal to the synchronizer so that the counter in the PV-controller in the synthesized path expression would be incremented correctly. Finally, the database resource module for this problem would be unchanged from that shown in Figure IV-7.
resource DBMP_decentralized;

operations
  INTREQ(upd_info, attempt_no, priority, writeset),
  EXTREQ(attempt_no, priority, writeset),
  UPD(upd_info), ACK+(attempt_no), ACK-(attempt_no), ACKd;

synchronizer
  state variables
    state: (passive, active, updating) init (passive);
    c: (0..n-1) init (0);
    current_attempt: integer;
    local_priority: prioritytype;
    local_writeset: writesettype;
  state changes
    UPD → {none};
    INTREQ → state := active;
      current_attempt := INTREQ.attempt_no;
      local_priority := INTREQ.priority;
      local_writeset := INTREQ.writeset;
    ACK+ → if ACK+.attempt_no = current_attempt
      then if c < n-1
        then c := c+1
        else begin
          state := updating;
          c := 0;
          msg.name := ACK+$ALL
        end;
    ACK- → if ACK-.attempt_no = current_attempt
      then begin
          state := passive;
          msg.name := ACK-$CURRENT
      end;
    ACKd → if c < n-1
      then c := c+1
      else begin
        msg.name := ACKd$ALL;
        c := 0;
        state := passive
      end;

{Continued next page}
EXTREQ → if state = passive
    then msg.name := EXTREQ$PASSIVE
    else if empty_intersection(EXTREQ.writeset, local_writeset)
        then msg.name := EXTREQ$BOTH
        else if EXTREQ.priority > local_priority
            then begin
                msg.name := EXTREQ$YOU_FIRST;
                state := passive
            end
        else msg.name := EXTREQ$ME_FIRST;

priorities {none};
synchronization
path 1: (UPD,
   (ACK+, ACK-)[state=active],
   ACKd[state=updating],
   INTREQ[state=passive],
   EXTREQ[state=updating] ) end;

access-mechanism
var save_info: db_update_info;
entries
   INTREQ: save_info := INTREQ.upd_info;
   EXTREQ.writeset := INTREQ.writeset;
   EXTREQ.attempt_no := INTREQ.attempt_no;
   EXTREQ.priority := INTREQ.priority;
   broadcast (EXTREQ);

   UPD: DO UPDATE.upd_info := UPD.upd_info;
   send (DO UPDATE) to (Ellis database);
   send (ACKd) to (UPD.source);

   ACK-: {nothing};

   ACK-$CURRENT: send (REJECT) to (user);

   ACK+: {nothing};

   ACK+$ALL: DO UPDATE.upd_info := save_info;
   send (DO UPDATE) to (Ellis database);
   UPD.upd_info := save_info;
   broadcast (UPD);

{Continued next page}

FIGURE IV-9. (continued)
ACKd: {nothing};

ACKd$ALL: send (DONE) to (user);

EXTREQ$BOTH,
EXTREQ$PASSIVE: ACK+.attempt_no := EXTREQ.attempt_no;
   send (ACK+) to (EXTREQ.source);

EXTREQ$ME_FIRST: ACK-.attempt_no := EXTREQ.attempt_no;
   send (ACK-) to (EXTREQ.source);

EXTREQ$YOU_FIRST: send (REJECT) to (user);
   ACK+.attempt_no := EXTREQ.attempt_no;
   send (ACK+) to (EXTREQ.source);

end resource

FIGURE IV-9. (continued)

5. Thomas's majority consensus algorithm for distributed databases

As in the previous example, Thomas's approach [Thomas 1979] to updating fully replicated databases uses decentralized control. The assumed environment is essentially the same as in the previous example: each database copy is associated with a DBMP, and the user interacts with a DBMP to update a database copy. However, in Thomas's scheme the DBMPs vote on the acceptability of update requests. For a request to be accepted, only a majority of the DBMPs need approve it. The steps in performing an update are as follows:

(1) The user queries the database. The DBMP responds by supplying the base variables for the transaction and their timestamps. (Each database element has a value and a timestamp—the time that the element received its current value.)

(2) The user computes new values for the update variables (some subset of the base variables).
(3) The user submits an update request to the DBMP. This update request contains the update variables with their newly computed values and the base variables with their timestamps.

(4) The DBMPs vote to accept or reject the update request.

(5) If the request is accepted, each DBMP applies the update to its copy of the database.

(6) The user is informed by a DBMP about how the request was resolved.

Thomas's majority consensus algorithm is composed of five rules: the DBMP/DBMP communication rule, the voting rule, the request resolution rule, the update application rule, and the timestamp generation rule. The details of all these rules will not be presented here; it is suggested that the interested reader examine Thomas's paper for this information. However, the voting rule and the request resolution rule will be stated here in order to clarify our representation of the algorithm in our extended notation. The DBMP voting rule is as follows:

(1) Compare the timestamps for the request base variables with the corresponding timestamps in the local database copy.

(2) Vote REJ if any base variable is obsolete.

(3) Vote OK and mark the request as pending if each base variable is current and the request does not conflict with any pending requests.

(4) Vote PASS if each base variable is current but the request
conflicts with a pending request of higher priority.\(^1\) (This PASS vote signals other DBMPs that a potential deadlock situation exists.)

(5) Otherwise, defer voting and remember the request for later reconsideration.

After voting, a DBMP uses the request resolution rule to see if its vote resolved the request. The request resolution rule which follows assumes daisy chained communication between DBMPs rather than broadcasting. The rule has two parts:

(1) After voting on a request R:
   (a) if the vote was OK and a majority consensus exists, accept R and notify all DBMPs and the user that R was accepted.
   (b) if the vote was REJ, reject R and notify all DBMPs and the user that R was rejected.
   (c) if the vote was PASS and a majority consensus is no longer possible, reject R and notify all DBMPs and the user that R was rejected.
   (d) otherwise, forward R and the votes accumulated so far to a DBMP that has not voted on it.

(2) After learning that a request R has been resolved:
   (a) if R was accepted,
      (i) apply R to the local copy of the database.
      (ii) reject conflicting requests that were deferred because

\(^1\) Each request is assigned a timestamp when it is originated, and the priority of a request is determined by this timestamp.
of R.

(b) if R was rejected,

(i) use the voting rule to reconsider conflicting requests that were deferred because of R.

In expressing Thomas's algorithm in our extended notation, we have again adopted the view that two resource modules are involved: one for the DBMP and another for the database. With this view, the DBMP module's resource object is the voting data, whereas the database module's resource object is the raw database. This separation thus allows one to specify the synchronization of access to each of these objects independently of the other. In Figures IV-10 and IV-11, which show the DBMP and database modules in the extended notation, we have made considerable use of pseudocode statements so that the programs do not become overburdened with detail.

In concluding this chapter, it should again be emphasized that what has been presented is only a proposal for an extended notation for the synchronization submodule. As such, it is only a starting point. Many questions are unresolved, particularly with respect to the aptness of the notational features and constructs and how they would be synthesized. For example, if dequeuing priorities are specified for an operation and the path expression translates into a simple PV-controller with that operation as an input, then the synthesis of the priority specification might simply amount to the replacement of the PV-controller's standard "dequeue" function by another which causes a different method of dequeuing. However, if the path expression is very
resource Thomas_DBMP;

operations
{From user:}
   QUERY(query_list),
   REQ_UPD(base_var, upd_var),
{From DBMPs:}
   REQ_VOTE(originator, timestamp, base_var, upd_var, okvotes, passvotes),
   DO_UPD(timestamp, upd_var),
   REJ(timestamp),
{From database resource:}
   READ_REPLY(inquisitor, base_var),
   DO_VOTE(originator, timestamp, base_var, upd_var, okvotes, passvotes, current_base);

synchronizer
state variables {none};
state changes {none};
priorities {none};
synchronization
   path 1:(QUERY, REQ_UPD, REQ_VOTE, DO_UPD, REJ,
   READ_REPLY, DO_VOTE) end;

access-mechanism
  var local_clock, newTS: timestamptype;
  pending_list, deferred_list: list of request;
  req: request;
  base_var, upd_var, current_base, query_list: list of dbelement;
  vote: string;
entries
  QUERY: READ.originator := QUERY.source;
       READ.query_list := QUERY.query_list;
       send (READ) to (Thomas_database);
  
  READ_REPLY: QUERY_REPLY.base_var := READ_REPLY.base_var;
       send (QUERY_REPLY) to (READ_REPLY.inquisitor);
  
  REQ_UPD: newTS := 1 + max(local_clock,
       maxtimestamp(REQ_UPD.base_var));
       local_clock := newTS;
       REQ_VOTE.timestamp := newTS;
       REQ_VOTE.originator := REQ_UPD.source;
       REQ_VOTE.base_var := REQ_UPD.base_var;

{Continued next page}

FIGURE IV-10. DBMP resource module—Thomas's majority consensus algorithm
REQ_VOTE.upd_var := REQ_UPD.upd_var;
REQ_VOTE.okvotes := 0;
REQ_VOTE.passvotes := 0;
send (REQ_VOTE) to (Thomas_database);

{To obtain current values of base variables}

REQ_VOTE: send (REQ_VOTE) to (Thomas_database);

DO_VOTE:
vote := voting_rule(DO_VOTE.timestamp, DO_VOTE.base_var,
DO_VOTE.current_base, pending_list);
case vote of
'O.K.': if DO_VOTE.okvotes+1 > nnodes/2
then begin
    send (UPD_ACCEPT) to (DO_VOTE.originator);
    DO_UPD.upd_var := DO_VOTE.upd_var;
    DO_UPD.timestamp := DO_VOTE.timestamp;
    send (DO_UPD) to all DBMPs {including self}
end
else begin
    copy all fields (except current_base) of
    DO_VOTE to REQ_VOTE;
    REQ_VOTE.okvotes := REQ_VOTE.okvotes+1;
    send (REQ_VOTE) to next DBMP
end;
'PASS': if DO_VOTE.passvotes+1 ≥ nnodes/2
then begin
    send (UPD_REJECT) to (DO_VOTE.originator);
    REJ.timestamp := DO_VOTE.timestamp;
    send (REJ) to all DBMPs {including self}
end
else begin
    copy all fields (except current_base) of
    DO_VOTE to REQ_VOTE;
    REQ_VOTE.passvotes := REQ_VOTE.passvotes+1;
    send (REQ_VOTE) to next DBMP
end;
'REJ': send (UPD_REJECT) to (DO_VOTE.originator);
REJ.timestamp := DO_VOTE.timestamp;
send (REJ) to all DBMPs {including self}
'DEFER': copy all fields (except current_base) of
DO_VOTE to record "req";
append(req, deferred_list);

{Continued next page}
DO_UPD: WRITE.upd_var := DO_UPD.upd_var;
    send (WRITE) to (Thomas database);
    delete record identified by DO_UPD.timestamp
        from pending list;
    for all req in deferred list do
        if req was deferred due to this DO_UPD request
            then begin
                delete(req, deferred_list);
                send (UPD_REJECT) to (req.originator);
                REJ.timestamp := req.timestamp;
                send (REJ) to all DBMPs {including self}
            end;

REJ: delete record identified by REJ.timestamp
    from pending list;
    for all req in deferred list do
        if req was deferred due to this REJected request
            then begin
                delete(req, deferred_list);
                copy all fields of req to REQ_VOTE;
                send (REQ_VOTE) to (self)
            end;

end resource

FIGURE IV-10. (continued)
resource Thomas_database;

operations
{All from DBMP:}
READ(originator, query_list),
WRITE(upd_var),
REQ_VOTE(originator, timestamp, base_var, upd_var, okvotes, passvotes);

synchronizer
state variables {none};
state changes {none};
priorities {none};
synchronization
{Writers' priority as an example:}
def wW = req(WRITE) - auth(WRITE);
path 1: {{READ, REQ_VOTE}[wW = 0]}, WRITE) end;

access-mechanism
var v: db_element;
base_var, upd_var, current_base, query_list: list of db_element;
entries
READ: READ_REPLY.base_var := emptylist();
for all v in READ.query_list do
append read(v) to READ_REPLY.base_var;
READ_REPLY.inquisitor := READ.originator;
send (READ_REPLY) ^ (READ.source);

REQ_VOTE: copy all fields of REQ_VOTE to DO_VOTE;
DO_VOTE.current_base := emptylist();
for all v in REQ_VOTE.base_var do
append read(v) to DO_VOTE.current_base;
send (DO_VOTE) to (REQ_VOTE.source);

WRITE: for all v in WRITE.upd_var do
if database_timestamp(v) < timestamp(v) in upd_var list
then write new value of v into database
{Update application rule}
end resource

FIGURE IV-11. Database resource module—Thomas's majority consensus algorithm
complex and the desired operation name appears more than once, would the specialized dequeuing function be applied at each relevant controller or only at the ultimate controller?

Another major problem is that of relating changes in the values of state variables to path expression predicates which reference those variables. One question in this regard is: should the state variables be incorporated into the pred-controllers which reference them, or should they be centralized and communicate their values to the appropriate pred-controllers? Another question to be asked is: exactly when does an operation cause a specified state change—on initial entry to the synchronization submodule or only when the message has made its way through the subnetwork of controllers and arrives at the access-mechanism? These questions and others will require further exploration before the extended notation can be claimed to be general and implementable.
V. CONCLUSION

A. Summary of Results and Proposals

In the target environment of which this research is a part, the goal is to employ totally encapsulated resource modules in a distributed system of potentially highly-parallel nodes. Within this framework, a resource module is composed of two internal submodules: a synchronization submodule which coordinates requests for access to the resource object and an access-mechanism submodule which localizes the operations on the resource object.

With path expressions viewed as a desirable means for expressing synchronization constraints non-procedurally, various forms of the path notation were examined. Two forms were seen to stand out as candidates for use in the target environment: OPEs and PPEs. A detailed investigation of the semantics of these two notations was presented, based on P-V implementation semantics for OPEs [Campbell and Miller 1978]. Having adapted their translation algorithm to a restricted form of PPEs, we then used this common semantics and introduced two definitions of equivalence—S-equivalence and PV-equivalence—to examine the equivalence and non-equivalence of various OPEs and PPEs. A subjective comparison of the two notations was also made in which it was concluded that OPEs are more desirable for some problems but PPEs are better for others. Specifically, OPEs were seen to be attractive because of their parallel semantics and the directness of their translation into dataflow graphs (and then into networks of
communicating submodules). However, the inability of OPEs to directly access the synchronization state of a resource leads in many cases to convoluted, multilevel expressions requiring "dummy" operations. PPEs, on the other hand, provide concise, clear problem specifications in many cases because the synchronization state is explicitly accessible through predicates involving the implicit event counters req(X), auth(X), and term(X). However, PPEs were judged to be limited in their ability to express concurrency in other than simple forms. As a result, a new notation—the OPPE—was introduced which combines the two notations. In particular, we adopted the OPE notation in its entirety and added the capability of attaching predicates to subexpressions.

An implementation semantics for OPPEs was then derived, based on initial work by Oldehoeft [Oldehoeft and Jennings 1984]. His approach was to synthesize OPEs into dataflow graphs using a left-to-right, bottom-up parse of the path expression; subsequently, the dataflow graphs would be implemented by networks of communicating submodules (PV-controllers, burst-controllers, and distributors) written as applicative language programs with the goal of applying them to data-driven dataflow architectures. In adapting and extending their work for the synthesis of OPPEs, we concluded that the intermediate form in the translation—the dataflow graph—was perhaps more important even than the final form—the program representations of the controllers—for two reasons: (a) the graphs prove to be important in their own right as a two-dimensional representational aid to the programmer in understanding synchronization problems, and (b) even though the controllers could be
represented as applicative language programs for dataflow architectures, we found that in writing the controllers in a more conventional sequential language notation for von Neumann systems, no substantial amount of concurrency of access to the resource object was sacrificed. Using the dataflow graph as an intermediate form leaves open the choice of an architecture of a node in a distributed network—it might be a cluster of communicating microprocessors, a dataflow machine, or some variant proposed for fifth generation computers [Treleaven and Lima 1982].

To synthesize OPPEs into dataflow graphs a fourth type of controller—the pred-controller—was introduced for the implementation of predicates; then the concept of predicate interference was defined, and a normal form for predicates was specified. It was postulated that if an operation interfered with another's predicate (i.e., the authorization of or request for the operation could change the value of the other's predicate from true to false), then requests for both operations must be input to the same pred-controller to avoid simultaneous testing and modification of the same predicate in different locations. In doing so, it was shown by example that a secondary predicate on the independent (i.e., interfering) operation is required within the pred-controller to avoid a race condition in which the dependent operation's predicate could change from true to false while it is transit through the synchronization subnetwork. With respect to automating the detection of predicate interference, it was proven that a very simple test for interference could be made by examining the
subtractive terms of the normalized predicate. A complete statement of
the synthesis algorithm was then presented and numerous examples were
given for clarification of specific steps in the algorithm.
Optimizations to the translation process were also proposed to make the
derived dataflow graph more efficient and to reduce the possibility of
deadlock. Lastly, the high-level language representation of pred-
controllers was discussed.

The final topic presented was a proposal for an extended notation
for the synchronization submodule. It is felt that such an extension is
necessary because of the inherent difficulty of using path expressions
to express solutions to problems involving priorities based on arguments
supplied with operation requests and to problems which depend on
synchronization states other than those expressible with the implicit
event counters req(X), auth(X), and term(X). Although the proposed
notation is strictly preliminary with respect to its generality and
implementability, solutions to two well-known synchronization
problems—the elevator-type disk scheduler and the alarm clock—and to
several multiple copy database concurrency control algorithms were
presented using the extended notation. A drawback to the extended
notation is that it represents somewhat of a regression from the
simplicity and non-procedurality of path expressions alone. The
tradeoff is between a very-high-level notation with its lower scheduling
power and a high-level notation with its greater scheduling power.
Whether the extra power gained by extending the OPPE notation justifies
the extra programming and translation effort required remains to be
B. Directions for Further Research

One area for further investigation is that of the usefulness of additional operators in OPPEs. The repetition operator "*" of RPEs and PPEs and the collateral execution operator "," of PPEs might be deemed desirable to add to the OPPE notation for some synchronization problems. However, as discussed in Section II.B.1, the employment of these operators is likely to be troublesome using our implementation semantics, and it is currently our feeling that these operators do not add substantially to the power of the OPPE notation. Another operator that might be examined further is one we will denote by "!"—an "exact count" operator. When used, for example, in the form n!(A);B this operator would specify that exactly n executions of A must take place and terminate before a B can be authorized to proceed. (This can be contrasted with n:(A);B which declares that at most n executions of A can take place before each B is executed, but that a B may begin as soon as at least one A has terminated.) In effect, this represents a bounded burst operation, and we envision the dataflow module representation of an exact-count-controller as it appears in Figure V-1.

Its internal state would consist of two counters (start_count and finish_count), a queue of waiting messages, and a phase designation ("idle", "initiate", or "active"). During the "idle" phase, the arrival of an operation-request message would cause the message to be enqueued, a "start-exact-count" message to be placed on the second output arc, and
FIGURE V-1. Proposed exact-count-controller as a dataflow module

the phase to be changed to "initiate". With the return of the "start-exact-count" message from any required presynchronization, the two counters would be initialized to n, and up to n operation-request messages which were enqueued while waiting for the return of the "start-exact-count" message would be dequeued and placed on the third output arc. Each such release would cause start_count to be decremented by one. The phase would then be changed to "active", and as long as start_count were greater than zero, any incoming operation requests would be placed directly on the third output arc and the counter would be decremented. If start_count were to reach zero, any further requests would be enqueued. Throughout the "active" phase, the receipt of an operation-termination message would cause the counter finish_count to be decremented. When finish_count reaches zero, an "exact-count-
termination" message would be placed on the fourth output arc to effect any desired postsynchronization, and the phase would be reset to "idle". Although further experience may be required to determine how often this operator is needed, we did find that the synchronizer for the DBMP in Ellis's centralized control algorithm (Figure IV-6) could be written without any state variables or predicates by using the "!" operator in the following manner:

\[
\text{path 1: (UPD, (INTREQ;1:(ACK-, (ACK+;(n-1)! (ACKd)))))) end}
\]

Another area for further research is the use of arbitrary boolean expressions as predicates in OPPEs. The major part of our work, appearing in Chapters II and III, assumes the restriction of predicates to boolean functions of linear relational expressions in integer constants and implicit event counters. In our synthesis scheme, the use of such predicates allows easily identified event messages to perform P-like and V-like operations on pred-controllers according to whether the events are named in additive or subtractive terms of the normalized predicate. Specifically, events named in additive terms can perform V-like operations that may result in unblocking suspended operation requests. Similarly, events named in subtractive terms can perform P-like operations that may block subsequent operation requests, and it was shown that the detection of predicate interference relies on the inspection of such subtractive terms. If arbitrary boolean predicates were allowed, this determination of potential blocking and unblocking of requests would not be as straightforward. In a relevant work [Schmid 1976], Schmid discusses the determination of such "enabling" and
"disabling" relations when the predicates are linear relational expressions in a form such as ours. He then addresses the "general case"—that in which the event variables are real valued and the relational expressions are arbitrarily complex—and claims that the determination of enabling and disabling relations is still possible using mathematical analysis methods. Although Schmid's work is within a different context—that of conditional critical regions which are collected into a centralized monitor—it could be a useful starting point for the consideration of arbitrary predicates in our environment.

Another area which merits additional investigation is the formal verification of synchronization problem solutions. Some work has been done on the formal specification of semantics of RPEs and their variants [Lauer and Campbell 1975, Berzins 1977] and on formal proofs of correctness of abstract data types which use them [Flon and Habermann 1976]. For PPEs, Andler defines their semantics in terms of partial orderings of events generated by corresponding nondeterministic programs [Andler 1979]. Then, to prove the correctness of solutions which use PPEs, he hopes to apply existing proof techniques for sequential programs to execution sequences of these nondeterministic programs. Regarding OPEs, a recent paper by Campbell [Campbell 1982] presents a definition of OPEs using a set of definitions and axioms which are used to derive invariants for the verification of synchronization constraints for abstract data types. Among the topics for further research mentioned in the paper is that of verifying that the P-V implementation of OPEs (which forms a substantial basis for our research) correctly
synchronizes operations according to those definitions and axioms. With respect to correctness proofs involving OPPEs, we envision that two directions might be explored initially: (a) applying a methodology similar to Andler's by transforming OPPEs into corresponding nondeterministic programs and then using existing program proof techniques or (b) working in Campbell's direction by providing a definition of OPPEs by incorporating predicates into his set of definitions and axioms.

Finally, another area in which further investigation would be useful is the possible revision of the OPPE notation when used with purely functional support architectures. Using data-driven dataflow architectures as an example, the access-mechanism submodule can be viewed as entity which:

- has an initial resource state (which includes the state or value of the enclosed resource object itself),
- has as inputs the current state of the resource and a stream of operation request messages,
- produces as output the new value of the resource state and a stream of response messages, and
- is recursively invoked by the arrival of an operation request message and the availability of the new resource state.

Thus, the resource state is treated as a value parameter which is input to the next invocation of the access-mechanism. With a read operation, the new resource state will be the same as the old resource state, so it is available immediately for reinvocation of the access-mechanism. This
allows a natural concurrency of read operations (and in our implementation the termination signal for a read operation would then be sent back to the synchronization submodule immediately instead of after the actual termination of the read operation). On the other hand, when the access-mechanism accepts a request for a write operation, a new resource state is produced by execution of the operation; hence, the next invocation of the access-mechanism is naturally delayed until the write operation terminates. The result of this design is that concurrent access to the resource object is implicitly synchronized by the access-mechanism: read operations naturally overlap and write operations are naturally permitted exclusive access.\(^1\) Thus, the programmer would not be required to specify simple read/write and write/write mutual exclusion when writing path expressions. For example, the solution to the weak readers' priority version of the readers-writers problem, originally expressed as
\[
\text{path 1:}([\text{READ}], \text{WRITE}) \text{ end},
\]
could be written as
\[
\text{path READ, WRITE end}
\]
using dataflow semantics. Also, the writers' priority solution, originally written as
\[
\text{path 1:}([\text{READ}[\text{req(WRITE)-auth(WRITE)=0}]], \text{WRITE}) \text{ end},
\]
could be written as
\[
\text{path 1:(READ, {WRITE}) end}
\]
using dataflow semantics. In spite of its appearance, the latter path expression would not enforce mutual

\(^1\) Given the above design, a write operation could overlap any initiated but unterminated read operations in progress. However, a write operation is viewed as creating a new logical copy of the resource object which is not available for access until its value is sent for the next invocation of the access-mechanism.
exclusion of READs from each other, nor would it allow a burst of concurrent WRITEs. Using dataflow semantics, it would simply prevent a READ from proceeding as long as there were WRITE requests waiting to join an in-progress burst of (implicitly mutually exclusive) WRITEs. Although OPPEs as currently written and synthesized would continue to "behave" correctly when imposed on such a dataflow architecture, they would be inefficient both in terms of programming time (for unnecessary considerations and specifications of mutual exclusion) and in terms of implementation (with more nodes and communication links in the synthesized synchronization network than are necessary).
VI. APPENDIX A: APPLICATIVE LANGUAGE FORMS OF THE BASIC CONTROLLERS

In this appendix are presented a PV-controller, a burst-controller, and a 1-by-4 distributor as represented in a stream-oriented dataflow language [Oldehoeft and Jennings 1984]. The language is in the style of existing or proposed dataflow languages [Bryant and Dennis 1982, Dennis and Weng 1979, McGraw 1982], and the use of recursion in conjunction with an input stream^ of data values forces a (temporary) serialization of responses to the incoming messages. In this language, the controller will only begin executing when all its input parameters are available—hence, a subsequent invocation of the controller will wait until the new internal state has been produced.

A. The PV-controller

The high-level language code for a PV-controller is shown in Figure VI-1.

---

1 A stream is a sequence of values, all of the same type, which are passed from one module to another in sequential order. The empty stream is denoted by []. Among the operations on a stream s are:

- **cons(c, s)** which yields a stream s' which has individual element c as its first element and stream s as the remaining elements.
- **first(s)** which yields a single element—the first element of s.
- **rest(s)** which yields the stream that remains after deleting the first element of s.
\begin{verbatim}
pv_controller = module (pv_count: integer;
    input_stm, pv_queue: stream of message)
    yields stream of message, stream of message;
let msg: message;
tail, op_stm, signal_stm: stream of message;
msg = first(input_stm);
tail = rest(input_stm);
in case msg.type of
    'req':
        if pv_count > 0
            then let signal_stm, op_stm =
                pv_controller(pv_count-1, tail, pv_queue);
                in signal_stm, cons(msg, op_stm)
            else pv_controller(pv_count-1, tail,
                insert(msg, pv_queue));
    'term':
        if pv_count < 0
            then let signal_stm, op_stm =
                pv_controller(pv_count+1, tail, rest(pv_queue));
                in cons(msg, signal_stm), cons(first(pv_queue),
                    op_stm)
            else let signal_stm, op_stm =
                pv_controller(pv_count+1, tail, pv_queue);
                in cons(msg, signal_stm), op_stm;
end {case};
end {pv_controller};
\end{verbatim}

FIGURE VI-1. Dataflow program for a PV-controller

B. The Burst-controller

Figure VI-2 shows a dataflow module representing a burst-controller. In this module, a fourth phase designation—"flush"—is used (in addition to "idle", "initiate", and "active"). Its purpose is to allow for flushing out the queue of waiting operation requests when the "start_burst" message returns to the burst-controller while it is in the "initiate" phase. Then in the "flush" phase, the controller recursively invokes itself to dequeue a message at a time until the queue is empty, at which time the phase is changed to "active."
burst_ctrlr = module (count: integer; phase: string;
queue, input_stm: stream of message);
yields stream of message, stream of message,
stream of message, stream of message;

let msg: message;
start_stm, op_stm, term_stm, sig_stm, tail: stream of message;
msg = first(input_stm);
tail = rest(input_stm);
in case phase of
  'idle':
    let sig_stm, start_stm, op_stm, term_stm =
      burst_ctrlr(1, 'initiate', insert(msg, queue), tail);
    in sig_stm, cons('start burst', start_stm), op_stm, term_stm;

  'initiate':
    case msg.type of
      'start_burst': burst_ctrlr(count, 'flush', queue, op_stm);
      'req': burst_ctrlr(count+1, 'initiate',
        insert(msg, queue), tail);
    end {case};

  'flush':
    if queue ≠ []
    then let sig_stm, start_stm, op_stm, term_stm =
      burst_ctrlr(count, 'flush', rest(queue), op_stm);
    in sig_stm, start_stm,
        cons(first(queue), op_stm), term_stm
    else burst_ctrlr(count, 'active', [], tail);

  'active':
    case msg.type of
      'req':
        let sig_stm, start_stm, op_stm, term_stm =
          burst_ctrlr(count+1, 'active', [], tail);
        in sig_stm, start_stm, cons(msg, op_stm), term_stm;
      'term':
        if count = 1
          then let sig_stm, start_stm, op_stm, term_stm =
            burst_ctrlr(0, 'idle', [], tail);
          in cons(msg, sig_stm), start_stm,
            op_stm, cons('term burst', term_stm)
        else let sig_stm, start_stm, op_stm, term_stm =
          burst_ctrlr(count-1, 'active', [], tail);
        in cons(msg, sig_stm), start_stm, op_stm, term_stm;
    end {case};
end {case};
end {burst_ctrlr};

FIGURE VI-2. Dataflow program for a burst-controller
C. A 1-by-4 Distributor

Figure VI-3 shows the representation of a 1-by-4 distributor which routes a stream of messages, each of which is a request for operation "opl" or "op2" or a termination message for "opl" or "op2".

distributor = module (input_stm: stream of message) 
yields stream of message, stream of message, 
stream of message, stream of message; 
let msg: message; 
opl_stm, op2_stm, term_opl_stm, term_op2_stm: stream of message; 
msg = first(input_stm); 
opl_stm, op2_stm, term_opl_stm, term_op2_stm = distributor(rest(input_stm)); 
in case msg.name of 
'opl': cons(msg, opl_stm), op2_stm, term_opl_stm, term_op2_stm; 
'op2': opl_stm, cons(msg, op2_stm), term_op1_stm, term_op2_stm; 
'term_opl': opl_stm, op2_stm, 
cons(msg, term_opl_stm), term_op2_stm; 
'term_op2': opl_stm, op2_stm, 
term_opl_stm, cons(msg, term_op2_stm); 
end {case}; 
end {distributor};

FIGURE VI-3. Dataflow program for a 1-by-4 distributor
VII. APPENDIX B: EXAMPLES OF DATAFLOW GRAPHS SYNTHESIZED FROM OPES

Contained in this appendix are examples of (abbreviated) dataflow graphs corresponding to several well-known synchronization problems: the bounded buffer, the stack, and variants of the readers-writers problem. Each graph is the final result of applying the synthesis algorithm introduced in Section III.B to the given OPE. The reader is encouraged to start with the OPE and apply the steps of the algorithm to see how the graph was synthesized.

A. The N-slot Bounded Buffer

An OPE from [Campbell and Kolstad 1980b] which allows a concurrent PUT and GET is:

\[
\text{path } n:(1:(PUT);1:(GET)) \text{ end}
\]

and the dataflow graph for this OPE is shown in Figure VII-1.

B. The N-element Stack

An OPE specifying the synchronization constraints for an n-element stack with operations PUSH and POP was presented in Section II.B as

\[
\text{path } n:(\text{PUSH;POP}), 1:(\text{PUSH;POP}) \text{ end.}
\]

The dataflow graph synthesized from this OPE is shown in Figure VII-2.
FIGURE VII-1. Dataflow graph for OPE
path n:(l:(PUT);l:(GET)) end
FIGURE VII-2. Dataflow graph for OPE
path n:(PUSH;POP), 1:(PUSH,POP) end
C. Readers-writers Problems

In the following variations of the readers-writers problem, the symbols R, W, RA, and WA will be used in place of READ, WRITE, READATTEMPT, and WRITEATTEMPT for the sake of brevity.

1. Weak readers' priority

Here, waiting readers do not necessarily inhibit writing. An OPE for this version is Example 13 of Section II.B:

\[ \text{path 1:} (\{R\}, W) \text{ end} \]

and the dataflow graph appears in Figure VII-3.

2. Strong readers' priority

In this variation, any waiting readers inhibit writing. Two OPEs and graphs will be presented: one (Example 15 of Section II.B) we derived from Campbell and Habermann's original solution [Campbell and Habermann 1974], and the other is our solution as presented in Footnote 2 of Section II.B.

a. Solution 1 of strong readers' priority:

\[ \text{path 1:} (WA), 1: (\{R\}, WA), 1: (\{R\}, (WA; W)) \text{ end} \]

The graph for this OPE is shown in Figure VII-4.

b. Solution 2 of strong readers' priority:

\[ \text{path 1:} (W), 1: (\{R\}, W) \text{ end} \]

This OPE is synthesized into the graph shown in Figure VII-5.
FIGURE VII-3. Dataflow graph for OPE

path 1:({R},W) end
FIGURE VII-4. Dataflow graph for OPE
path 1: (WA), 1: ({R}, WA), 1: ({R}, (WA; W)) end
FIGURE VII-5. Dataflow graph for OPE
path 1:(W), 1:([R],W) end
3. **Writers' priority**

In this variation, any waiting writers inhibit reading. An OPE for this version is Example 14 of Section II.B:

```
path 1:(RA), 1:(RA,{W}), 1:({RA;R},W) end
```

The graph for this OPE appears in Figure VII-6.

4. **Alternating burst solution**

In this variation, waiting writers inhibit only the start of a burst of readers. An OPE for this problem is Example 16 of Section II.B:

```
path 1:({R},{W}), 1:(W) end
```

and the synthesized graph is shown in Figure VII-7.

5. **Fair chance solution**

In this version, individual READ and WRITE invocations have an equal chance of being selected first for execution. We have derived the following OPE from Campbell and Habermann's original solution [Campbell and Habermann 1974]:

```
path 1:(RA,W), 1:({RA;R},W) end
```

The graph synthesized from this OPE appears in Figure VII-8.
FIGURE VII-6. Dataflow graph for OPE
path 1:(RA), 1:(RA, {W}), 1:({RA; R}, W) end
FIGURE VII-7. Dataflow graph for OPE
path 1;({R},{W}), 1;(W) end
FIGURE VII-8. Dataflow graph for OPE
path 1:(RA,W), 1:({RA;R},W) end
VIII. APPENDIX C: REVISED APPLICATIVE CODE FOR THE BURST-CONTROLLER

This appendix contains, in Figure VIII-1, a revised program for the burst-controller in the dataflow language notation presented in Appendix A. This revision is made so that the burst-controller will send an act_start_burst message on its signal output arc after it receives the returning start_burst message while in the "initiate" phase. The rationale for this revision was discussed in Example 4 of Section III.C.6.
burst_ctrlr = module (count: integer; phase: string;
queue, input_stm: stream of message)
yields stream of message, stream of message,
stream of message, stream of message;
let msg: message;
start_stm, op_stm, term_stm, sig_stm, tail: stream of message;
msg = first(input_stm);
tail = rest(input_stm);
in case phase of
'idle':
    let sig_stm, start_stm, op_stm, term_stm =
        burst_ctrlr(1, 'initiate', insert(msg, queue), tail);
    in sig_stm, cons('start burst', start_stm), op_stm, term_stm;
'initiate':
    case msg.type of
    'start burst':
        let sig_stm, start_stm, op_stm, term_stm =
            burst_ctrlr(count, 'flush', queue, op_stm);
        in cons(msg, sig_stm), start_stm, op_stm, term_stm;
'req': burst_ctrlr(count+1, 'initiate',
    insert(msg, queue), tail);
    end {case};
'flush':
    if queue # []
    then let sig_stm, start_stm, op_stm, term_stm =
        burst_ctrlr(count, 'flush', rest(queue), op_stm);
    in sig_stm, start_stm,
        cons(first(queue), op_stm), term_stm
    else burst_ctrlr(count, 'active', [], tail);
'active':
    case msg.type of
    'req':
        let sig_stm, start_stm, op_stm, term_stm =
            burst_ctrlr(count+1, 'active', [], tail);
        in sig_stm, start_stm, cons(msg, op_stm), term_stm;
'term':
        if count = 1
        then let sig_stm, start_stm, op_stm, term_stm =
            burst_ctrlr(0, 'idle', [], tail);
        in cons(msg, sig_stm), start_stm,
            op_stm, cons('term burst', term_stm)
        else let sig_stm, start_stm, op_stm, term_stm =
            burst_ctrlr(count-1, 'active', [], tail);
        in cons(msg, sig_stm), start_stm, op_stm, term_stm;
    end {case};
end {case};
end {burst_ctrlr};

FIGURE VIII-1. Revised dataflow program for a burst-controller
IX. REFERENCES


BYTE. 1981. Special issue on Smalltalk. BYTE 6(8).


