FOR FURTHER TRAN

AD A055375

ISI/SR-78-9
April 1978

ARPA ORDER NO. 2223

PROTECTION ERRORS IN OPERATING SYSTEMS:

Serialization

DDC
JUN 13 1978
F

Jim Carlstedt

AD No.
DDC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

78 06 06 038

JUN 13 1978

ISI/SR-78-9
April 1978

ARPA ORDER NO. 2223

PROTECTION ERRORS IN OPERATING SYSTEMS:

Serialization

Jim Carlstedt

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223.

VIEWS AND CONCLUSIONS CONTAINED IN THIS STUDY ARE THE AUTHOR'S AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF THE UNIVERSITY OF SOUTHERN CALIFORNIA OR ANY OTHER PERSON OR AGENCY CONNECTED WITH IT.

THIS DOCUMENT APPROVED FOR PUBLIC RELEASE AND SALE: DISTRIBUTION IS UNLIMITED.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ISI/SR-78-9	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): (6) Protection Errors in Operating Systems: Serialization		5. TYPE OF REPORT & PERIOD COVERED: (9) Research Report,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): (10) Jim Carlstedt		8. CONTRACT OR GRANT NUMBER(s): (15) DAHC 15-72-C-0308,
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order No. 2223
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209		(11) 12. REPORT DATE: April 1978 13. NUMBER OF PAGES: 37
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): --------- (12) 39p.		15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DDC
DRAFTED
JUN 13 1978
F

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer security, operating systems security, protection in operating systems, parallel processes, serialization errors

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

407952

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4B

20. ABSTRACT

This document describes a class of protection errors found in current computer operating systems. It is intended primarily for persons responsible for improving the security aspects of existing operating system software (here called protection evaluators) and secondarily for designers and students of operating systems. The purpose is to help protection evaluators find such errors in current systems, and to help designers and implementers avoid them in future systems, by analyzing them and suggesting methodical approaches for finding them.

The term 'protection evaluation' here denotes a search for errors based only on static information about a target operating system, primarily program listings but possibly other system documentation as well. These static methods discussed in this report are intended to complement dynamic methods such as testing, auditing, and penetration attempts.

This report deals with a class of errors initially identified empirically. The class formed itself around a group of protection errors (within a larger collection) having the common characteristic of involving operations or accesses occurring in the wrong order or at the wrong times relative to others; hence the name serialization for this class. In its broadest sense, it includes a large proportion of all programming errors-- those having to do with improper ordering or scheduling of "operations"; in a narrower sense it includes only those errors resulting from improper ordering of accesses to objects accessible by potentially concurrent operations.

A general treatment of the subject is far beyond the scope of this study. However, it was felt that certain types of errors fitting the broader definition but not the narrower one should be included, since an understanding of the former is helpful to an understanding of the latter. The result is neither a full analysis of the subject of the ordering of operations in programming systems nor only a discussion of process synchronization. Rather, it is an attempt to give perspective to several closely-related subclasses of problems in this area.

1473 B

ABSTRACT

This document describes a class of protection errors found in current computer operating systems. It is intended (1) for persons responsible for improving the security aspects of existing operating system software and (2) for designers and students of operating systems. The purpose is to help protection evaluators find such errors in current systems and to help designers and implementers avoid them in future systems, by analysis and methodical approach.

This report deals with a class of errors, initially identified empirically, that formed itself around a group of protection errors (within a larger collection) having the common characteristic of involving operations or accesses occurring in the wrong order or at the wrong times; hence the name "serialization". In its broadest sense, it includes a large proportion of all programming errors which may have improper order or scheduling, and, in a narrower sense includes only those errors resulting from improper ordering of accesses to objects accessible by potentially concurrent operations.

This study is neither a full analysis of the subject of the ordering of operations nor only a discussion of process synchronization, but rather an attempt to give perspective to several closely-related subclasses of problems in this area.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. SPECIAL
A

78 06 06 038

1. INTRODUCTION

This document is one in a series of related reports, each describing a class of protection errors found in current computer operating systems. These reports are intended primarily for persons responsible for improving the security aspects of existing operating system software (here called *protection evaluators*) and secondarily for designers and students of operating systems. The purpose is to help protection evaluators find such errors in current systems, and to help designers and implementers avoid them in future systems, by analyzing them and suggesting methodical approaches for finding them.

These studies were suggested by the "pattern-directed" methodology for protection evaluation proposed in [Carlstedt 75], intended to assist individuals such as system programmers (who may be nonexperts in the field of operating system protection but possess a good working knowledge of particular target systems) to effectively carry out enhancement tasks, i.e., to find protection errors. This is important because of the well-known vulnerability of current operating systems to operating and security losses due to protection errors, and the scarcity or unavailability of methods and expertise to reduce such losses.

Experience has shown that searches for errors are conducted most effectively by focusing on distinct well-defined types, one at a time, rather than by attempting to find errors of many different types all at once. Following this approach, the Protection Analysis Project at ISI has been engaged in studies of a series of error types. Earlier reports dealt with the integrity of parameters passed by reference [Bisbey 75], residual data in allocation and deallocation activities [Hollingworth 76], and enforcement of validity conditions [Carlstedt 76]. A fourth report described an experimental technique for determining data dependencies within and across procedure boundaries [Bisbey 76].

The term *protection evaluation* here denotes a search for errors based only on static information about a target operating system, primarily program listings but possibly other system documentation as well. The static methods discussed in these reports are intended to complement dynamic methods such as testing, auditing, and penetration attempts.

Protection evaluation is still primarily a manual and informal task. A few tools and techniques for formalizing or automating the task have been proposed, but as yet none of these can be applied with a high degree of power or generality. This is especially true for the class of errors discussed in this report.

Like others in the series, this report deals with a class of errors initially identified empirically. The class formed itself around a group of protection errors (within a larger collection) having the common characteristic of involving operations or accesses occurring

in the wrong order or at the wrong times relative to others; hence the name *serialization* for this class. In its broadest sense, it includes a large proportion of all programming errors--those having to do with improper ordering or scheduling of "operations". In a narrower sense it includes only those errors resulting from improper ordering of accesses to objects accessible by potentially concurrent operations.

A general treatment of the subject is far beyond the scope of this study. However, it was felt that certain types of errors fitting the broader definition but not the narrower one should be included, since an understanding of the former is helpful to an understanding of the latter (and since we wished to cover several interesting and apparently closely-related instances included in our collection). The result is neither a full analysis of the subject of the ordering of operations in programming systems, which would encompass the whole field of control specifications and mechanisms, nor only a discussion of process synchronization. Rather, it is an attempt to give perspective to several closely-related subclasses of problems in this area.

The report is organized as follows. Section 2 introduces the subject of serialization informally via several examples of errors taken from current operating systems, and indicates the general nature and importance of serialization errors. Section 3 introduces concepts and terminology common to the descriptions of the serialization problems in the following sections.

The next four sections describe four major classes of serialization concerns. These are closely related in that many serialization errors can be regarded as falling into more than one class. Also, the concern described in Section 4 is stated at a level of abstraction above those of the following three. Section 4 suggests the form of specification errors in general and ordering specifications in particular, and describes the most basic serialization concern of all, namely that operations occur during the occasions for which they are specified. Section 5 discusses interoperation communication, and the associated serialization concern of insuring the proper use of communication channels. Section 6 describes a class of serialization concerns commonly known as "mutual exclusion," from the point of view of preserving object integrity. Through most of this section, operations can be regarded as "atomic"; the primary concern is that of insuring that operations on objects occur only when such operations are "proper". Section 7 describes a somewhat different aspect of mutual exclusion concerns, that of the noninterference of nonatomic operations on each other. It identifies major types of critical access sequences and their mutual compatibilities.

Finally, Section 8 discusses some of the difficulties of detecting serialization errors in current large-scale operating systems.

2. SERIALIZATION--AN INFORMAL CHARACTERIZATION

This section presents serialization errors informally via a set of examples and general characterizations. The examples that follow have been abstracted from "raw" error descriptions to avoid explaining the particular operating system contexts in which they occurred.

1. A user process was able to apply a system procedure S to a file of a particular type at times when S was potentially incompatible with other operations possibly occurring concurrently on that file. S was overlooked as one of the types of operations requiring serialization on such files, due to the fact that it allowed its operand files to be designated by target address rather than by symbolic name.

2. A page management routine would initiate input of a page p for a given process U into a block containing a page belonging to another process V, which was waiting for some other type of event, before setting the "page missing" flag in V's page table. This allowed the possibility of an interval during which V might regain control and access part of U's new page p.

3. A reentrant storage management routine inhibited interrupts only while it removed or inserted elements in a doubly-linked list of free storage blocks, but not while updating the total remaining number of blocks, the accuracy of which was critical to its own correct functioning whenever the number of remaining blocks was zero.

4. Every request to a service program S resulted in the request handler setting flag f, enqueuing the request, and placing S on the scheduler's list of ready tasks. When S received control, it would process each request on its queue, calling an appropriate subprogram for each, before finally resetting f and terminating. If interrupted, S would not regain control from the scheduler if f were not set, since in that case it would have merely reset f and terminated. Unfortunately, one of its infrequently-used subroutines erroneously reset the flag itself, resulting in the possibility of the remaining enqueued requests being ignored until another request was received.

5. A flag for a rare event of type g was set and a corresponding descriptor provided in a prespecified location whenever an event of that type was detected. This required attention by an ansynchronous program which took appropriate action and reset the flag. If another event of type g happened to occur during this interval, the previous one could have been ignored, or its description modified while in use, or the second event ignored, depending on the timing of the second event relative to the actions being taken on the first.

6. A large and complex service routine was scheduled to receive control whenever a request was placed in its queue. Responsibility for deleting requests from the queue

was distributed throughout several sections of code. Under a certain unlikely combination of conditions, removal of the request just serviced could be inadvertently skipped, causing that same request to be processed again.

7. Two operating system tasks were regarded as mutually independent: A, concerned with address space and virtual storage allocation, and B, concerned with (physical) primary storage allocation. Both could be called into play and allowed to proceed concurrently when a process referenced for the first time any procedure not currently within that process's address space nor residing in primary storage. Under certain circumstances, storage space for the newly-referenced procedure could be allocated from space currently allocated either to the given process or to other processes, depending only on the relative real speeds of A and B, i.e., depending on which was dispatched first.

8. Parameters for a user-callable supervisor procedure S were passed by reference. The supervisor procedure first validated the parameters and then used them to determine the specific action to be taken. Immediately before calling this procedure, a user process could issue a request to an input process for a designated file to be read into a location that included one or more of the parameters for S. Thus the user could arrange to have the values of the parameters changed after they were validated but before they were used.

9. A reentrant system directory update procedure failed to lock a directory while modifying a file descriptor in that directory. When called by two or more processes concurrently, one modification could interfere with another, resulting in an invalid descriptor.

As is often true of examples purporting to illustrate errors of a given class, some of the errors illustrated by the above examples could also be used as examples for other types, and could have been avoided by means other than proper serialization. For example, (1) could have been prevented by a more consistent naming/addressing scheme, (4) and (6) by more centralized management of flags and queues, respectively, and (7) by a different storage allocation policy. Example (8) represents the type of error discussed in [Bisbey 75], which could also be prevented by requiring the passing of parameters by value only.

How do such errors qualify as protection errors? Not all of them appear sufficiently "critical". However, any error in an operating system that can cause invalid data and thereby invalid operations that access that data is potentially a protection error. The concern for protection has been the primary reason for the increased concern for correctness in current operating system design, and for defining data, procedure, and process hierarchies such that elements at any given level cannot be adversely affected by processes at higher or outer levels [Graham 68, Neumann 77, Popek 75, Schroeder 75]. A

malicious user will look for errors of any type whatsoever, knowing that their usefulness depends on the criticality of the data and procedures affected [Carlstedt 76].

Serialization can be viewed from various perspectives or at various levels of abstraction. The following aspects are distinguished:

** *Policy*, consisting of specifications about purposes or effects.

** *Mechanism*, consisting of procedures to enforce policy.

** *Invocation*, consisting of specifications for instances for the application of mechanisms.

Correspondingly, serialization errors can manifest themselves in the following ways:

** Inadequate or inconsistent policy.

** Inadequate or incorrect mechanisms.

** Missing, improper, or inappropriate invocation specifications.

As a part of both programming and protection, serialization is concerned with insuring that operations occur when and only when they are appropriate, meaningful, intended, helpful, legal, valid, proper, effective, etc. As an area of protection, it is concerned primarily with constraintive aspects of programming and programming systems, namely with avoiding or preserving (rather than achieving) certain conditions at certain points or over certain intervals of time, and with disallowing operations that would violate such conditions, or when the conditions required for those operations do not hold. (The term "condition" is used loosely here.)

Unlike the *prohibitive* mechanisms of access control, which might preserve conditions by disallowing access to the variables involved (e.g., write-protecting them), or which might terminate processes or force them into different control paths when condition violations would otherwise occur, serialization is *inhibitive*, causing processes to be delayed until the conditions for their continuation are satisfied, under the assumption that the conditions will be changed by operations of other processes. An access control constraint is of the form "<operation> only if <condition>"; in a serialization constraint the "only if" becomes "only when" or "not until," the assumption being that conditions will eventually change to allow the operation to occur.

As stated in the introduction, the serialization concern involves the ordering of operations relative to one another. Serialization errors are errors in ordering

specifications, either at the policy or mechanism level. The only reason for ordering any pair of operations relative to each other is that conditions affected by one might affect the appropriateness or result of the other, i.e., that the two operations are data-related. Thus, serialization is concerned with two kinds of relationships among operations: *data flow* and *control flow*. Roughly speaking, the purpose of serialization is to maintain the appropriateness of the latter with respect to the former.

The importance of serialization as an area of protection in operating systems is easily underestimated, because even where the potential for serious serialization errors exists, they may actually manifest themselves only rarely during system operation, or not be recognized as such when they do occur. Thus (from the standpoint of maintaining system integrity) serialization errors may seem to be less of a problem than some other types.

The reason that few actually occur (at least accidentally!) is that such occurrences usually require an improbable combination of conditions, for example the accessing of the same data element by two processes within a very small time interval. The reason they are often not recognized as such when they do occur is that their effects are often attributable to other causes, such as hardware unreliability or obscure functional errors in operating system code. Their effects may not even be noticed until much later, for example not until system shutdown when unprocessed messages may be noticed.

Serialization may be difficult to understand and serialization errors hard to find, but for these very reasons this is an attractive area for purposes of intentional exploitation by malicious users, a primary concern of protection evaluators.

3. CONCEPTS AND TERMINOLOGY

This section introduces some of the basic concepts of programming systems, in terms of which the serialization concerns of the next four sections can be more easily described.

Serialization concerns are best described and understood in terms of the dynamics of *operations* and *events*. Errors, however, are expected to be detected by analyzing static specifications for *operators* and *objects*. In what follows, these static and dynamic entities are carefully distinguished.

The starting observation is that an *operation*, the primary object of serialization, is the result of a three-way attachment: *processing system* to *operator* to *objects*.

An underlying *processing system* interprets and executes specifications expressed in its language S, or *S-specifications*, in basic units called *instructions*, to produce corresponding units of activity called *actions*. The processing system consists of two

major types of resources: processors and units of storage; nothing is assumed here about the individual unit capacity or total capacity of either (in particular, about the absolute or relative speeds of processors). Actions are not necessarily as primitive as those of the physical processors of most current systems; an underlying processing system might include high level virtual processors such as those provided in the kernel of an operating system. However, certain actions mentioned below and in Section 5 are assumed to be effectively instantaneous.

An *operator* is a unit of action specification consisting of a set of instructions related by control-type S-specifications that specify the potential orders in which the instructions are to be interpreted.

An *operation* is the interpretation of an operator by the processing system, and the carrying out of the specified actions, which occur as a single sequence in time. The operation is said to be *based* on its operator. The terms "operator" and "operation" are intended to suggest nothing about unit size; operators can range from very long specifications to single instructions, and operations can range from time-consuming processes to brief actions.

One of the most important actions specified by an operator is the *invocation* of another operator for execution. The instruction for this is an *invoker*. An invocation results in the initiation of another operation to be performed, based on the invoked operator. An obvious example of an invoker is the "call" statement in an actual programming system. However, only the operation-initiation semantics of invokers are of interest here; nothing is assumed about the immediacy with which initiated operations are actually performed, or about initiating operations being delayed while initiated ones are performed. Invocations are assumed to be instantaneous.

An *object* is a value-containing entity describable and denotable by name in S. It may be simple or composed of other objects. Examples in actual programming systems are simple and structured program variables, files, records, and data bases. In general, objects are related to other objects via the directed relations "contains as component or member" and "possesses as value". In an extended sense the concept also includes primitive data objects denoted as literal constants (e.g., characters, numerals, "true"/"false"); however, we restrict objects here to "container-like" entities whose identities are distinct from their values.

A collection of objects related in some given way is an *environment*. In the following sections the terms "environment" and "object" are usually interchangeable. Collectively, all the values and components of an environment (and the components' values, etc.) are its *state*. A *condition* is a class of states of a given environment E described by a predicate C[E]. In what follows, "the condition C[E]" means "the condition described by C[E]"; "the state S of E (or E itself) satisfies C[E]" means "S is in C[E]"; "condition (event) C[E] occurs" means "the state of E enters C[E]".

Two objects x and y are *covariant* if they occur together in a relational expressing an intended invariant condition, where a relational is a primitive term such as x<y or x=2*y. A *related group* is an equivalence class of objects under the transitive extension of the relation "x and y are covariant".

Every operation has an environment of objects that it *accesses*, as described in Section 5 below. This is determined by the processing system as follows: The instructions (including invokers) in an operator contain *identifiers* that denote objects in the *formal environment* of that operator. Whenever the operator is executed its identifiers are bound to actual objects, which they then denote, and which then constitute the environment of the operation being performed. Thus the same operator may be *applied* to the same or different environments for different operations.

The binding of identifiers is achieved via *contexts*. A context is an S-specification for a one-to-one mapping from identifiers of an operator into actual objects. A variety of contexts may exist, some of which may be permanently associated with the operators whose identifiers they bind and others of which may be associated with particular invocations. An invocation context is explicit or implicit in every invoker. Thus an invoker has the form

DO A[E,m],

where A designates an operator, E an environment, and m a context. (The context of an invoker existing in operator A and designating operator B maps all the to-be-invocation-bound identifiers of B into identifiers of A. Since the actual invocation occurs only during an operation based on A, its identifiers are necessarily already bound to actual objects; thus the given identifiers of B are also ultimately bound to actual objects.)

The expression A[E,m] defines an *applicator*, which instantiates to a specific operation based on the operator A for a particular state of the environment E. In what follows, the possibility of binding the same operator to the same environment in different ways is ignored, and applicators are denoted in the form A[E]. The word "application" is used in place of "operation" when a particular applicator is assumed.

In view of the definition of "object" above, to state that the identifiers of an operator are bound to actual objects when it is executed is actually an oversimplification, since some of them may be bound to literal constants. In the following sections, particularly in Sections 6 and 7, applicators are distinguished only by operator and actual environment. To account for the possibility of (input) identifiers in an operator being bound to literal constants, two applicators are regarded as identical only if the identifiers of their operator are bound to the same literal constants as well as to the same actual objects. Under this view, ADD[1,x] is a different applicator from ADD[2,x].

4. ORDERING SPECIFICATIONS AND OPERATION OCCASIONS

Programming errors can be thought of as incompatibilities of S-specifications with more authoritative, abstract, and possibly less formal *policy* or *reference specifications*. It follows that to describe serialization errors in a given class S of systems, it is necessary to describe the relevant classes of both S- and reference specifications. Since serialization is concerned with the ordering of events of certain types, these will be event-ordering specifications.

An *event ordering* is a partial time-ordering on a set of events. Two kinds of orderings are of interest:

- *Potential* orderings of events that may occur in the future.
- *Actual* orderings of events that have occurred in the past.

The effect of the activities of a processing system is to manifest a particular actual ordering from the class of potential orderings described by the set of S-specifications being interpreted (together with an initial state). The following are examples of the types of events involved in ordering specifications:

- Primitive read or write accesses.
- Beginnings and ends of nonprimitive use and modification accesses.
- Beginnings and ends of use and modification intervals (sequences of primitive and nonprimitive use and modification accesses).
- Beginnings and ends of operations (more generally).
- State changes that result in given conditions being entered or left.
- Interpretations of S-specifications themselves, i.e., events implicitly designated as "now".

The various types of accesses mentioned above are defined in Section 5 following.

The variety of potentially meaningful S-interpretable ordering specifications is virtually unlimited. In general, an ordering specification must specify two things:

1. An *effect*, which is a class or set of events to be achieved (in a progressive specification) and/or prevented (in a constraintive one).

2. An *occasion*, which is an interval during which the effect is to be achieved or prevented.

An *occasion predicate* is a predicate *Occ*(I) on a set I of intervals, each of which is bounded by one (for an infinite interval) or two (for a finite interval) events of one of the types listed above. As with conditions, one speaks most properly only of occasions "holding"; to say an occasion "occurs" means it begins to hold. The roles of effect and occasion may sometimes be indistinguishable, as in a simple specification of a form such as "*e1 before e2*".

In a progressive ordering specification or *p-spec*, the effect component specifies an application (or a set of applications from which a choice is to be made on the basis of lower-level considerations); hence such a specification has the general form

DURING *Occ* DO *Appl*

where *Occ* is an occasion predicate and *Appl* is an expression involving conditions and applicators. This form includes simple invokers of the form DO A[E], where a particular applicator is specified and where the occasion is implicitly specified as "(after) now". However, it is helpful to think of all p-specs as though they included explicit occasion predicates.

An *occasion* for a given application is a set of conditions under which some goal arises toward which progress can be achieved by performing that application while those conditions hold. As suggested by the above form, the same occasion might indicate a disjunction or conjunction of applications. For example, the completion of a data base update might indicate a number of different summarizing operations.

One of the most general expressions of serialization policy is the following:

Every application must be performed only during an occasion for it.

This can be qualified in two ways. First, the problem of insuring that the occasion for an operation does not end until the operation has been performed is discussed in Section 7 under "Noninterference," and can be ignored here. Second, only one application of the type indicated by an occurring occasion should normally be performed during that occasion. These qualifications yield the following policy:

R1. Every application must be performed only when the occasion for it has occurred, and then only once.

The primary problem in enforcing this policy is recognition of the occasions designated in p-specs. Not all p-specs occur in the forms most familiar to programmers, such as sequentially-interpreted unconditional and conditional function and procedure invocations. In particular, they may occur in "nonsequential" programming, as independent p-specs whose occasion predicates require virtually continuous evaluation by the processing system or by special operations created for such purposes. A variety of mechanisms are used to facilitate and signal such recognition, including special recognizers for suboccasions or events of various types, and the representation of occurrences of suboccasions via "event flags," logical-valued objects whose values are set to "true" when such suboccasions are recognized.

The recognition of an occasion may result in the direct invocation of the indicated applicator, or it may be represented by an "occasion signal," indicating the requirement for an invocation at some later time (before the occasion ends). A sequence of occasions of a given type (i.e., described by the same predicate) constitutes a conceptual "occasion channel". Such a channel may be physically embodied as an environment (usually a queue) of objects representing pending invocations. When an occasion of a given type can occur at a frequency greater than the rate at which indicated applications can be performed by the processing system, the elements of such an occasion channel must be buffered. Concerns specific to the management of occasion channels are similar to those of communication channels, discussed in Section 5 following.

5. *INTEROPERATION COMMUNICATION*

A large class of serialization concerns is associated with intentional communication among operations. These include those traditionally referred to as "producer-consumer" problems [Dijkstra 71] and "reader-writer" problems [Courtois 71].

Roughly speaking, communication by one operation with another occurs in the form of one or more *accesses* to objects in the intersection of the environments of the two operations. Accesses are of two types:

- A *use* access U[w,x] by an operation w to an object x obtains part or all of the state of x without changing it.
- A *modification* access M[w,x] assigns one or more other objects as values or components of x, thus in general changing its state.

In what follows, use and modification accesses to a given x by various operations such as v and w are denoted by U[v] or M[w]; use and modification accesses by a given operation to a given object are denoted simply by U and M.

Accesses by a given operation occur both *directly*, via *read* (use) and *write* (modification) instructions in its operator; and *indirectly*, via subsidiary (invoked) operations. Read and write actions involve only primitive objects; they also occur instantaneously, i.e., they are events. A nonprimitive direct U[w,x] (M[w,x]) may also occur, in the form of a sequence of read (write) accesses to various components of x. An indirect U[w,x] (M[w,x]) is carried out via a subsidiary w' of w, where w' in turn either reads (writes) x directly or uses (modifies) x indirectly.

In the broadest sense, operations v and w communicate with each other if there exists to some object x, the *communication object*, a sequence of accesses M[v]..U[w] or M[w]..U[v], with no intervening modifications to x in either case. However, such accesses may be unintended, in which case they represent cases of *interference*, discussed in Section 7 below. Actual *communication* means the intended variety.

U[w,x] (M[w,x]) is a *receive* (*send*) access if either of the following apply:

** It is the initial U[w,x] (final M[w,x]) by w.

** There exists at least one intervening M[v,x] (U[v,x]) by some operation other than w, i.e., relative to the last previous U[w,x] (next subsequent M[w,x]).

Serialization is concerned only with communication accesses, not with uses and modifications of objects in private environments of operations. The same object x may be employed as a communication object during parts of an operation w and as an object private to w during other parts.

An object x is an *input* to an operation w if it is used by w before being modified by it (if modified at all); it is an *output* of w if it is modified by w. The input and output environments of an operation may be disjoint, may partially overlap, or may coincide.

Communication concerns are often identified with occasion concerns, where the occasion for an operation w is mistakenly thought of as represented entirely by its inputs. However, an occasion for w need not be a condition on the environment of w, i.e., Z can be disjoint from E in the p-spec DURING C[Z] DO A[E]. An example is an operation intended to issue a notice on each occasion of a certain data base condition, where the notice does not include the values of the data base objects themselves. In principle, not all of the inputs to an operation need even have the same states when the occasion for it is recognized or when it is created, as they do later when first used by it.

In other words, the fact that its occasion occurs does not imply that all of its initial inputs need be "ready". The only policy that must be enforced in this regard is that while an operation is using its "occasion inputs" (i.e., those actually involved in the occasion predicate of the p-spec that indicated that operation) they must not be allowed to be modified so as to nullify its occasion. This is more suitably regarded as a policy of noninterference, which is discussed in Section 7 below.

The same object may be utilized as a communication object by different sets of operations for totally unrelated purposes (although this practice occurs less and less as computer storage becomes less expensive). A *channel* is a communication object associated with a specific purpose or role. The essential serialization concern of interoperation communication is that of coordinating accesses to communication objects as channels or elements of channels.

In general, a channel alternates between two *access phases*:

** A *modification phase*, during which its state is subject to change and cannot be guaranteed to be stable, consistent, or otherwise usable.

** A *use phase*, during which the opposite is true.

The following policy must therefore be enforced:

R2. The modification and use phases of a channel must be recognized or defined, and all modifications and uses of it must be confined to its modification and use phases, respectively.

Three conceptually distinct types of channels are employed in interoperation communication:

1. A *simple channel* is a communication object through which are communicated state changes primarily regarded as changes in value rather than in membership, and which is accessed by senders and receivers "in place".

2. A *message* is similar to a simple channel except that instead of remaining stationary in the joint environments of its sender and receiver, it is moved from the environment of its sender and into the environment of its receiver. This is done either directly or via a message channel ((3) below), and is carried out as a result of *put* and *get* instructions (or instructions with the same effect)by its sender and receiver, respectively. These indicate explicitly the end of its modification phase and the beginning of its use phase, the two phases possibly being separated by an in-channel phase.

3. A *message channel* is an expanding and contracting environment into which messages are "put" and from which they may be "gotten". As a channel, the state changes communicated are regarded as changes in membership rather than in value. (It is actually a metachannel.)

Both simple channels and message channels exist in actual systems in many forms, such as actual parameters and parameter lists, shared files, request queues, interrupt stacks, and input/output buffers.

Similar to a message channel is a *resource pool*, a set of available resources of the same type, allocated to and deallocated from operations. A resource pool is similar to a message channel in that allocation and deallocation actions explicitly distinguish *in use* and *available* phases for individual resource units. Objects serving as messages are placed in resource pools to indicate the ends of their use phases, and are obtained from resource pools to indicate the beginnings of their modification phases, via instructions similar to "put" and "get". Policy analogous to R2 applies to resource pools as well.

Aspects of the management of simple channels, message channels, and resource pools by an underlying system, other than distinguishing their phases and synchronizing their participants, are beyond the scope of this report. These include the problem of insuring that operations access communication objects only when they are included as participants in the current channel roles of those objects; enforcing implicit or explicit sender-receiver designations; and scheduling sending/receiving, production/consumption, or allocation actions both fairly and in such a way as to avoid deadlock, when those of more than one operation have been delayed due to lack of processing or storage resources. A general mechanism for such management, called a "monitor," is described in [Hoare 74].

6. *OBJECT INTEGRITY*

This section describes a class of serialization concerns associated with protecting objects from "untimely" operations, where the criterion used to determine whether an operation should be performed is its *propriety* with respect to its effect on objects it accesses, as well as its *compatibility* with operations currently accessing those objects. In Section 7 the discussion of compatibility is continued, but from the point of view of the integrity of the accessing operations themselves. The concerns described in these two sections are commonly known as "mutual exclusion" [Dijkstra 71, Horning 73]. The necessity for mutual exclusion under certain conditions is widely recognized, but the basic reasons have rarely been analyzed.

Associated with any object, related group, or environment x is its *integrity condition* C[x], the class of states regarded as ever possibly valid for it, implied by the

real-world objects or phenomena it is intended to represent, or by the types of operations expected to access it. (Integrity conditions are also called "consistency conditions," usually in connection with data bases. The former term applies better when single objects as well as related groups are of concern.) If C[x] does not hold, x is in an *invalid* state (meaning inherently invalid, rather than simply failing to meet the validity conditions of particular operators). An operation w that modifies x is *proper* with respect to the integrity of x if and only if the state resulting from w is contained in C[x], assuming no other operations participate in modifying x during w.

Since the propriety of an operation in general depends on its inputs, it is more meaningful to speak of the propriety of an application of an applicator A[x], where A is an operator and x is the total input and output environment to which it is applied. More precisely, propriety is a boolean function *Propr*[S[x],A[x]], where S[x] is the state of x when A[x] is applied. Thus one can speak of the propriety of an environment x for an application of A[x] as well as the converse. For a given applicator A[x] there is a class of states CA[x] (which need not be a subset of C[x]) for which *Propr*[S,A] is true. In terms of the dynamics of x, i.e., the sequence of states resulting from operations on it, for given applicator A[x], the state of x may from time to time enter CA[x] and then be proper for A[x] (Figure 1). A[x] is "totally improper", "totally proper", or "partially proper" depending on whether CA[x] is null, a superset of C[x], or neither. Prevention of totally improper applications is an access control concern; A[x] is here assumed to be at least partially proper for x.

Note the difference between propriety and other properties of an application and/or the state of its environment. First, A[x] can be proper even when the state of x is either inherently invalid, or invalid for A[x] in particular. In either case it could still perform correctly, or else produce spurious output that happened to satisfy a given integrity condition. For the description of object integrity concerns to make sense, however, we must assume the integrity of operations and therefore the validity of inputs. (A[x] can also be proper for states of x for which it is not appropriate, i.e., even when there is no occasion for such an application. The converse is of course false, since to achieve an invalid state is never a programming goal so there can be no occasion for it.)

Propriety policy is summarized by the following:

R3. An application must be performed only when its environment is proper for it.

In many cases only a given set of applicators {A[x]} needs to be considered that can be applied to a given environment x. In such cases the state space of x is partitioned into *phases* by the members of the powerset of {CA[x]}, such that for each phase only the operators of a unique subset of {A[x]} are proper. R3 is more easily enforced by keeping track of phases than states, particularly if the phase resulting from each application of some member of {A[x]} is only phase-dependent rather than

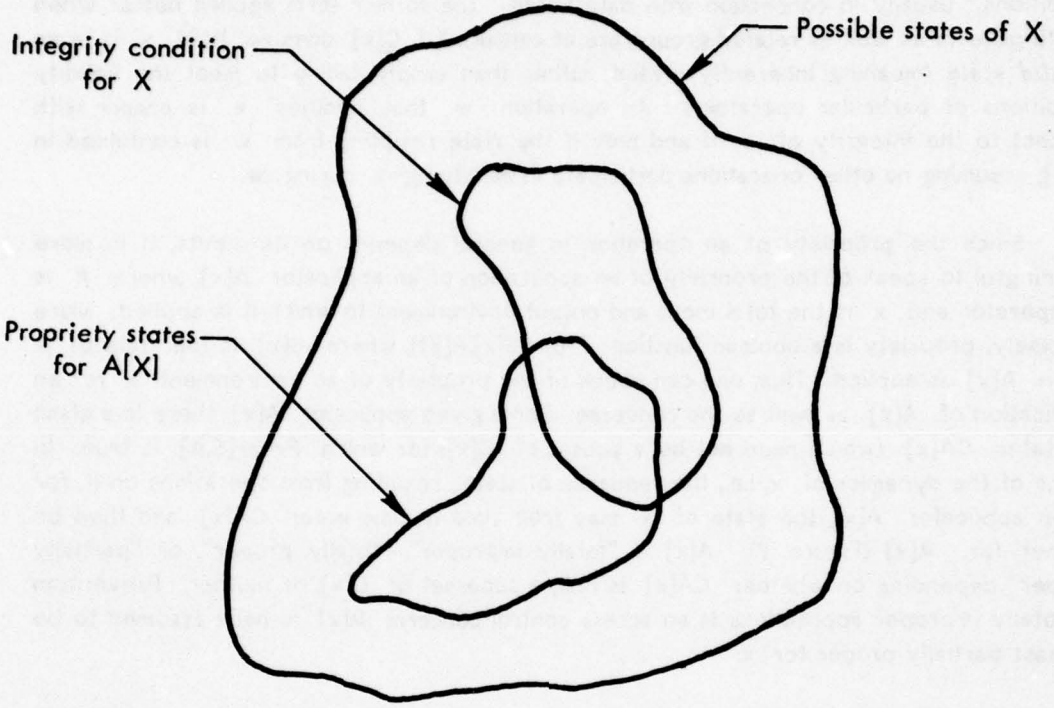

Figure 1. Integrity and propriety states

state-dependent. In this case, "state" is replaced by "phase" in R3, and the dynamics of x is modeled by a phase transition table, as illustrated in Table 1. (C[x] need not be covered by UNION{CA[x]}, i.e., a proper operation can in principle result in a state in which no applicators in a given set {A[x]} are proper. Avoiding such states is a scheduling or programming problem.)

Enforcement of R3 can be carried out by the underlying processing system on the basis of "path expressions" supplied to it [Campbell 74, Habermann 75, Chow 76]. Sequences of applications intended to be performed as a unit, called *transactors*, are also covered by R3. In Table 1, for example, the only proper phase in which to start the transaction A2-A3-A4 is phase 6. Any operation, sequence of operations, or sequence of accesses within an operation, which must be completed as a unit for reasons of either object or operation integrity, is called a *critical* operation or sequence. Policy to insure that critical operations complete without interference is discussed below and in Section 7 following.

TABLE 1
Sample Propriety Phase Transitions

Propriety Phase	APPLICATOR A1	A2	A3	A4	A5
P1	2	Improper	3	2	4
P2	1	5	6	2	8
P3	4	7	Improper	1	3
P4	Improper	5	4	Improper	2
P5	6	Improper	8	Improper	Improper
P6	Improper	2	1	5	Improper
P7	Improper	3	Improper	Improper	3
P8	Improper	Improper	7	Improper	Improper

Consider now all applications involving x, not just those that modify it. An application that does not modify x is of course totally proper for it. In general, not all of the applications that may be proper for x in a given phase may be performed concurrently, for reasons of either object or operation integrity to be identified in the remainder of this section and in the next. Two applications based on the same or different applicators are *totally compatible*, *totally incompatible*, or *partially compatible* with respect to their access to x, depending on whether they may always be performed concurrently, may never be performed concurrently, or may be performed concurrently except for certain critical accesses and sequences of accesses, respectively. The following policy must be enforced:

R4. Of the applications that may be proper for an environment in a given state or phase, only those that are totally or partially compatible may be allowed to proceed.

Because concurrent incompatible applications are likely to (though may not necessarily) result in an invalid state, selection of one or more proper initiated compatible applications that access x effectively constitutes a transition into a compatibility subphase of x. This is an *allocation* of x to that set of compatible applications.

As an example of transitions among compatibility subphases, consider (applications consisting of) single use and modification accesses to an object x. Assume a single propriety phase in which either access is proper, i.e., assume that a valid state will be assigned by any modification access. As noted in Section 5, use and modification accesses are inherently incompatible. The object x must undergo a transition to a compatibility subphase by being allocated to either use or modification accesses exclusively (Figure 2). Further, the result of concurrent modifications is (roughly speaking) unpredictable, so that modification accesses are incompatible with themselves. Thus when x is in the

modify-only subphase it can be allocated only to a single modification access, putting it in a subphase for which no other accesses are proper.

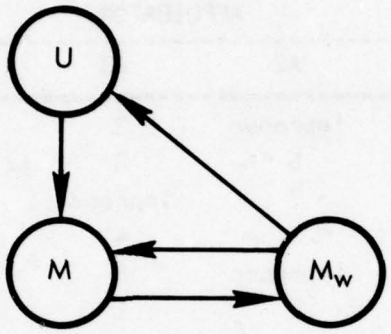

Figure 2. Use and modification phase transitions

The previous paragraph would apply in the same way if use and modification accesses were replaced by read and write accesses, where the latter were assumed to be noninstantaneous. Virtual instantaneity and nonsimultaneity of read and write accesses is provided by the underlying processing system by enforcing phase transitions similar to those just described. This is sometimes called preventing "race conditions"; the principle of no concurrent or simultaneous writes is called "certainty" [Gilbert 72, Dijkstra 68]. Under these assumptions, serialization policy for prevention of concurrent single use and modification accesses is covered by the communication policies stated in Section 5 above, and policy for preventing concurrent modification accesses is covered by policy stated in Section 7 below.

In summary, the primary problem of object integrity is to delay operations within given environments until the occurrences of proper states or phases for them, and to select compatible operations for performance from among those proper for a given phase. The unit of serialization is regarded here as the complete operation. Not all object integrity concerns can be so characterized, however. In some cases the unit of serialization is a critical access or sequence of accesses to a particular object within an operation. The noninterference concerns described in Section 7 involve such critical access sequences. Accesses and access sequences of the types described below are also critical, but more from the standpoint of object rather than operation integrity. For this reason they are discussed here rather than in Section 7.

A single nonprimitive modification access M[w,x] is critical if x is a related group. Without knowledge of the states to be assigned to the various components of x, once such a modification has begun, i.e., once the first write access has occurred to one of the

components, it must be assumed that x is in a temporarily invalid state and will remain so until and only until (assuming operation propriety and integrity) M[w,x] has ended. If in the interim some other operation attempts to modify any portion of x, such a modification must be assumed inconsistent with M[w,x] and must therefore be delayed at least until M[w,x] has completed.

Actually, the primitive write accesses of nonprimitive modification accesses to x by different operations can be interleaved so as to preserve the validity of x, but this requires advance knowledge of the order in which each of the operations will access the components of x [Eswaran 76]. For example, each of the write accesses of w can be immediately followed, component for component, by those of some other composite modification access that completely masks w but leaves x valid. Assuming no such advance knowledge, the following policy is implied:

R5. While a nonprimitive modification of a related group is in progress, all other modifications of any of its objects must be prevented.

A *use-modification* sequence consists of a U[w,x] followed some time later by a M[w,x] with the condition S'[x] = f(S[x]), where S is the state obtained by U, S' is the state assigned by M, and f denotes a function computed by w. Such a U-M sequence is a critical *sequential update* if x is a *sequential update object*, meaning that no two such sequences may be based on the same "state-interval" of x, i.e., their U's may not occur during the same interval in which the state of x remains unchanged. The modification phase of x can be regarded as beginning the moment the U access has completed (putting x in a "virtually" unstable, unreadable state for the remainder of the U-M sequence). Examples of sequential update objects in actual systems are those containing counts, for example, of available units of resources of various types.

Another way of characterizing a sequential update object is to say that it is one for which the total effect of any two updates must be the same as though they were performed entirely sequentially. Except for updates U[w]-M[w] and U[v]-M[v] in which f is identical, or in which f[v] is the inverse of f[w] with respect to x, the effect of U[w]-M[w] followed by U[v]-M[v] is different from the effect of U[v]-M[v] followed by U[w]-M[w]. The sequential update concern assumes that one of the updates has been selected or is already in progress, and is based on the recognition that sequential updates are incompatible in the sense defined above. It is summarized as follows:

R6. When a sequential update is in progress, all other sequential updates of that object must be prevented.

As a matter of fact, all modifications to the object being updated must be prevented during a sequential update, but for the sake of their own effectiveness, not for the sake of

the effectiveness of the updating operation nor for the sake of the integrity of the object being updated. This problem is discussed in Section 7.

7. *OPERATION INTEGRITY*

In the descriptions of the serialization concerns in Sections 4 and 5, and in the first half of Section 6, operations are regarded as *atomic*. An atomic operation is one whose environment is not or cannot be accessed by any other operation while the given operation is being performed, and which, once started, is not or need not be delayed in favor of any other operation. Actions of the underlying processing system are examples of (primitive) operations which are atomic. A system in which all operations are atomic is modeled by the "U-process-M" paradigm, where all use accesses occur together at the beginning and all modification accesses occur together at the end of an operation. In such a system or model, serialization is concerned only with the ordering of complete operations relative to one another; whether or not operations can exist concurrently is irrelevant.

When used properly, atomization is a powerful and efficient method of enforcing noninterference policies [Lomet 77]. A common but overly-coarse atomization technique is to inhibit all interrupts during the performance of operations regarded as critical.

In the last half of Section 6, and in this entire section, serialization concerns are described that arise from possibilities for effects of nonatomic concurrent operations on one another. Such possibilities exist in all systems capable of performing, in shared environments, more than one operation at a time, either because of the availability of more than one physical processor or because a processor can be interrupted during one operation and switched to another, temporarily suspending the first (effecting virtual multiprocessing).

To describe these concerns, operations must be regarded in more detail, in terms of the sequences of accesses they make to individual objects or related groups during their performances. Serialization then becomes an ordering of accesses and sequences of access, rather than of operations themselves.

Noninterference concerns relating more directly to object integrity were described in the latter part of the previous section. In this section they are described from the point of view of operation integrity. Obviously, the two types of concerns are closely related: roughly speaking, the integrity of an operation, in the sense that the states of its outputs bear intended relations with the states of its inputs, depends on the initial and maintained integrity of those input objects. Conversely, the integrity of an object depends on the integrity of operations modifying it. However, the notion of operation integrity can be generalized and sharpened by defining it in terms of various types of access sequences an operation might contain, rather than only in terms of its use accesses to its inputs.

Like communication, interference by one operation with another occurs in the form of accesses by the former to objects in the environment of the latter. One kind of interference can be read directly from the definition of communication in Section 5: *operation v interferes with operation w if there exists to some object x a sequence of accesses M[v]...U[w], with no intervening modifications to x, where such a sequence is unintended.* The characteristic common to all instances of interference is that v performs a modification access to the environment of w, so as to affect the states of outputs of w. (Just as every modification access is assumed to have an effect on the object accessed, the state of every input to an operation is assumed to have an effect on the subsequent state of some output of that operation.) However, not all cases of interference are characterized by the M[v]...U[w] sequence. Furthermore, not every instance in which one operation modifies the environment of another is a case of either communication or interference.

Interference occurs when and only when some object or related group x, accessed as either input or output by an operation w, is modified by another operation during a *critical interval* when such modifications, for one of a number of possible reasons, are not intended. Such an interval includes a *critical sequence* of direct or indirect accesses by w to x. The usual concern for operation integrity, implying protection of its input environment only, is extended here to require possible protection of its output environment as well. However, other components of operation integrity, namely operator correctness and input validity, are assumed as before.

In the following paragraphs, five types of critical access sequences to an object or related group x by an operation w are identified.

1. *Composite use* U = u...u. This is a sequence of use accesses to components of x, intended as a single access in the sense that its purpose is to obtain the state of x as it existed at some moment just prior to the beginning of the sequence. Clearly, all modifications of x must be prevented during some interval covering this sequence, at least all composite modifications of x (see below). (Primitive write accesses to components of x already read as a part of U can be permitted; however, policy covering such interleaving of composite accesses is beyond the scope of this report. For a discussion of possibilities for such interleaving, see [Eswaran 76].)

2. *Steady state use* U...U. This is a sequence of primitive or composite use accesses to x as a whole, where the intention is that throughout the sequence x satisfies a "critical condition," which is either an absolute condition C[x], or a relative condition C[x,S'] describing a class of states of x relative to the state S' at the beginning of the sequence. During such a sequence all modifications to x must be prevented (at least those that would cause C[x] not to hold). In practice, the requirement is almost always that the state S[x] remain unchanged throughout the interval, i.e., that S = S'. Under this assumption, policy governing steady state use sequences is the same as that governing composite use accesses (1), and the two can be treated as one concern.

3. *Composite modification* M = m...m. This is a sequence of modification accesses to components of x, intended as a single access in the sense that its purpose is to assign a particular state to x. Obviously, for this to be effective, all modifications to x by other operations must be prevented during some interval covering this sequence--at least to components that have already been written as a part of this composite access. Since modifications to components not yet written as a part of this composite access would themselves not be effective, the summary policy is that all other modifications must be prevented.

The difference between this concern and that expressed by R6 in Section 6 is that there the concern is merely for preserving object integrity, whether the modification itself is effective or is masked by a concurrent one, while here the concern is that the modification itself be effective. Under the assumption that possible interleaving is not to be considered (and only under this assumption), identical policy governs both; they are not distinguished in the rest of this report.

4. *Sole-source modification* M...M. This is a sequence of primitive or composite modification accesses to x as a whole, where the intent is simply that w is the only operation to modify x during some interval covering this sequence. This concern for separating modification accesses from one another complements the communication concerns described in Section 5, those of separating use from modification accesses. Since the requirement for the integrity of a sole-source modification is the same as for the integrity of a composite modification, the two can be treated as the same concern.

5. *Coherent modification-use* M-U. This is a sequence consisting of a M[w,x] followed at some point by a U[w,x], with the intent that the state of x does not change in the interim. Every instance in which an operation uses a shared object temporarily for communication with itself is an example of such a sequence. Although in most cases the private environment should be utilized for such purposes, coherent M-U sequences to shared objects are sometimes employed, such as in update-verify sequences on shared data bases.

The apparent converse of (5) is the *sequential update* of the form U-M discussed in Section 6 previous. The sequential update is viewed as critical from the point of view of object integrity rather than operation integrity, since its intention per se is always fulfilled regardless of intermediate modifications by other operations.

The critical interval for each of the types of critical accesses or access sequences identified above, i.e., the interval during which extraneous modifications are intended to be prevented, does not always coincide with the interval bounded by the first and last read or write actions of the sequence. For example, in the case of a sole-source modification M...M, the critical interval might begin some time before the first M and end some time after the last one, depending on how x is intended to be employed. A critical interval is a period of time during which x is allocated to a critical sequence or sequences of a

given type. In other words, a critical interval corresponds to (but does not necessarily coincide with) a phase of the object accessed. The beginning and end of a critical interval cannot in general be determined by the underlying processing system, but must be indicated by the accessing operations. This information is then used by the processing system to determine phase transitions.

The mutual compatibilities of the four major types of critical access sequences identified in Sections 5-7, with respect to a common object x, are summarized in Table 2. *U sequence* denotes either a composite use or a steady-state use sequence; *M sequence* denotes either a composite modification or a sole-source modification sequence; *U-M sequence* denotes a sequential update; and *M-U sequence* denotes a coherent modification-use sequence. Each entry in Table 2 has one of two values:

Y: the access sequence of that row (by an operation v) may proceed totally or partially (see note) concurrently with the access sequence of that column (by an operation w), to which x has already been allocated.

N: the access sequence of that row may not begin until the access sequence of that column, to which x has already been allocated, has ended.

TABLE 2
Access Sequence Compatibilities

	U seq.	M seq.	U-M seq.	M-U seq.
U seq.	Y	N	Y(1)	Y(2)
M seq.	N	N	N	N
U-M seq.	Y(3)	N	N	Y(4)
M-U seq.	N	N	N	N

(1) The U sequence must end before the M subsequence of the U-M sequence begins.

(2) The U sequence must not begin until the M subsequence of the M-U sequence has ended.

(3) The M subsequence of the U-M sequence must not begin until the U sequence has ended.

(4) The U subsequence of the U-M sequence must not begin until the M subsequence of the M-U sequence has ended; the M subsequence of the U-M sequence must not begin until the U subsequence of the M-U sequence has ended.

Serialization policy for avoiding concurrent mutually incompatible access sequences or subsequences is expressed in the following statement (which includes the policy expressed in R5 and R6 of the previous section):

R7. When a critical access sequence or subsequence is in progress, only other sequences or subsequences of compatible types may be allowed to proceed concurrently with it.

The access sequence compatibility concern may be rephrased in terms of compatibility phases and subphases of a shared object. Seven such phases and subphases are defined by the relations represented by the above matrix:

1. U phase (not allocated to any particular operation).

2. M phase allocated to a particular operation but not as a subphase of a U-M or M-U phase.

3. M subphase as the first subphase of a M-U phase allocated to a particular operation.

4. U subphase only as the second subphase of a M-U phase allocated to a particular operation.

5. U subphase both as the second subphase of a M-U phase allocated to one operation w and as the first subphase of a U-M phase allocated to another operation v.

6. U subphase only as the first subphase of a U-M phase allocated to a particular operation.

7. M subphase as the second subphase of a U-M phase allocated to a particular operation.

The transitions among these phases, corresponding to allocations and deallocations, are shown in Figure 3. R7 is restated in terms of object phases as follows:

R8. Critical access phase and subphase transitions for a shared object must be limited to those shown in Figure 3.

Note that phases and subphases 2,3,4,6, and 7 must be allocated to a single operation, while subphase 5 may be shared by two operations. Phase 1 is the only one that can be shared among more than two operations.

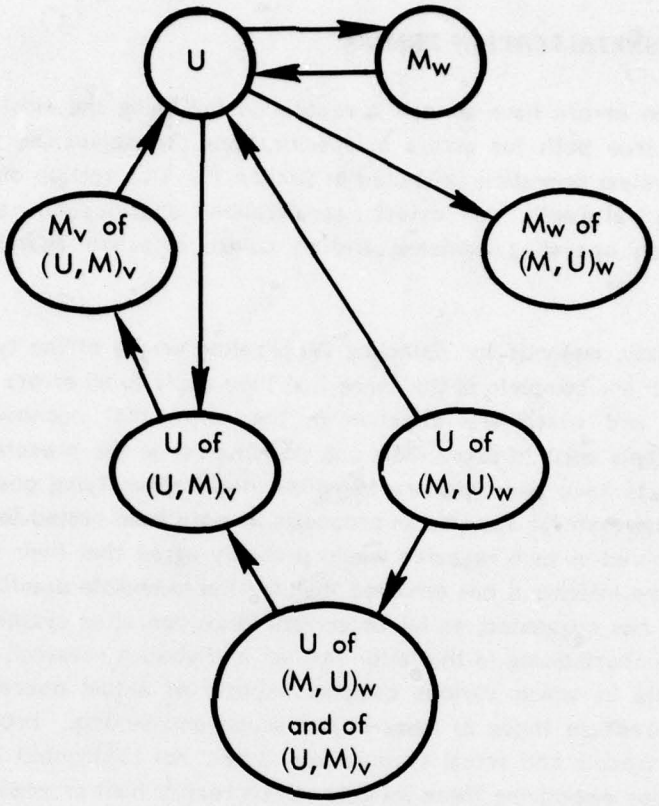

Figure 3. Compatibility phase transitions

The phases and phase allocations/transitions identified here apply only to objects as a whole. As noted, other phases and transitions would be derived if possibilities for access interleaving were exploited by recognizing component accessing orders. A somewhat similar scheme, but one which allows allocating both hierarchically-structured objects and their components to compatible access sequences, is presented in [Gray 75], where, however, only two basic phases or "modes" are provided: "share," allowing use but no modification, and "exclusive," allowing both use and modification by a particular operation.

Just as the operation types involved in determinations of mutual compatibility can transcend operator boundaries in the form of transoperators (as noted in Section 6 previous), so also the individual accesses of critical access sequences can be distributed among a number of operators, i.e., critical intervals may also transcend operation boundaries. For example, the U in a U-M sequence may occur in one operation, and the corresponding M in a later one. Operations containing such accesses must be scheduled or programmed to occur in the proper order.

8. *DETECTING SERIALIZATION ERRORS*

Serialization errors have earned a reputation for being the most difficult type to detect. This is true both for errors in specifications themselves, as well as for their effects during system operation (as noted in Section 2). This section discusses problems associated with attempts to detect serialization specification errors in large current-generation operating systems, and to devise effective search strategies and methods.

Unfortunately, methods for detecting serialization errors of the types described in this report, which are complete in the sense that they apply to all errors of these types in entire systems, and which are effective in the sense that economical and reliable techniques and tools exist to carry them out, do not exist at the present time. A number of research efforts have been and are being devoted to identifying possibilities for such tools, but none have so far resulted in proposals of more than limited feasibility. Most of the persons involved in such research would probably agree that their work has been of value more for the insights it has achieved than for the immediate practicality of any tools or techniques it has suggested, as far as current large operating systems are concerned. Its most common shortcoming in the latter respect is that such research is typically based on formal models in which various complex aspects of actual operating systems are abstracted, for example those of name-object scope and binding. Problems of mapping between formal models and actual systems are usually not confronted to the extent that tools or techniques embodying these insights can be readily built or employed.

For example, Alexander [Alexander 74] shows how to prove that only allowable sequences of state transitions can occur in a program, by comparing the state graph of the program with a prototype state graph of the class of allowable sequences. The method is extended to a system of processes by including process-local state identifiers as state information within a single system state graph. However, no general procedure to produce state graphs for an actual system is given; without such a procedure the method is practical only when a few manually-identifiable variables and states are involved.

Howard [Howard 76] shows how Hoare's formal methods for program verification can be applied to prove the correctness of the management of queues (message channels) by monitors, with various types of signalling operators defined and compared. These methods would apply only in operating systems where such monitors are used.

Keller [Keller 76] uses an "induction principle," defined in terms of an abstract conceptual model and applied to a formal model of parallel programs, to show how certain properties of such programs can be proved, including that of mutual exclusion. However, the technique is demonstrated only for very simple examples; the problem of choosing invariant conditions for actual implementations is not confronted.

Lipton [Lipton 75] defines a technique called "reduction," by which the simpler analysis of sets of noninterruptible program units can be substituted for the more difficult analysis of interruptible ones. However, the method is applied only to prove that sample communicating operations, based on operators specified in a given "parallel programming language," cannot deadlock (although it appears applicable to the verification of other serialization properties as well).

Owicki and Gries [Owicki 76] show how Hoare's methods can be applied to prove various properties of parallel programs, including mutual exclusion. Again, the assumption is that (a) the target system contains identifiable synchronization mechanisms corresponding to two types of statements in a given parallel programming language, one for initiating parallel operations (RESOURCE r: COBEGIN...COEND) and one for synchronizing on shared objects (WITH r WHEN b DO s); and of course that (b) the proper invariants (integrity conditions) for shared objects can be determined. It may well be that neither of these assumptions holds for an actual system.

A strategy for detecting serialization errors must be based on the policy or policies violated by such errors. Policies R2-R8 in Sections 5-8 have the following aspects in common:

** Objects potentially accessible by two or more concurrent operations.

** Phases or classes of states of such objects, during which accesses of only certain types are proper and also mutually compatible.

As usual, the word "object" is used in its extended sense to include environments. In the above, access sequences and operations are included in the meaning of "access". From now on, unless otherwise stated, this word will be used in the static sense, to mean "access specification"; an *access interval* is then a control path segment (see below) in which one or more accesses to a given object occur. Use and modification accesses are represented in target system programs by appearances of variables in expressions, assignment statements, and parameter lists.

To enforce these policies, an operating system must provide for the following:

1. Distinction of (concurrently) sharable from nonsharable objects.

2. Classification of sharable objects on the basis of the different types of accesses to them.

3. Recognition of types of access phases for various types of objects (objects having the same types of accesses will have the same types of access phases).

4. Means of establishing or recognizing the transitions among access phases.

5. Means of recognizing candidate accesses (in the dynamic sense), and allowing to proceed only those that are proper and compatible for given phases. proceed.

Serialization errors can occur in the design or implementation of any of these provisions. The rest of this section comments briefly and in general terms on possible approaches to the detection problem.

At least three approaches can be taken. One is to analyze the target system macroscopically and informally for the adequacy of each of the above provisions. How are sharable objects distinguished? In what ways can objects become known to and hence shared by operations? What types of accesses to shared objects exist, and how are these distinguished for protection purposes in general and serialization purposes in particular? Are objects themselves classified by the types of accesses allowed? What types of access phases are defined for the various types of objects? How are the beginnings and ends of the various phases determined or recognized? How are accesses or attempted accesses monitored so as to allow only those that are proper and compatible? On each of these questions is superimposed the following test: "Are the definitions and distinctions adequate; are any possibilities overlooked or not anticipated in the design?" Careful analysis of this type can certainly lead to the discovery of serialization errors. The problem with this approach is that no actual algorithm is suggested for deciding when serialization errors do or do not exist.

A more methodical strategy is one which starts by attempting to determine potential concurrencies, and given these, attempts to determine whether any of them (e.g., taken pairwise) represent access conflicts. To describe this strategy it is necessary first to define the term *potential concurrency*. The execution by the processing system along a *control path* from instruction to instruction within and among operators, is denoted by the generic term *process*; a control path can be regarded as consisting of a set of *control points* comprising the locus of a potential process. At any instant, an actual process *occupies* a particular control point and all control path segments containing it. Control segments of various types (e.g., a segment beginning with an entry point of a procedure and ending with a return point of that procedure) define potential *subprocesses* of corresponding types. Two control segments, i.e., possibly overlapping segments of the same or different control paths, represent potential concurrencies if processes can occupy them both during the same interval of time.

It is not necessary, of course, to identify all potential concurrencies, but only those representing possible conflicts. The problem is to be able to answer the question,

> "Given an access interval to some object, or two or more access intervals to the same object, can more than one process occupy it or them simultaneously?"

To identify potential conflicts the following kinds of information are necessary:

1. The control points at which processes can originate.

2. The conditions under which they can originate.

3. The conditions governing their movements along control paths.

4. The ways in which those conditions can arise.

These kinds of information can be obtained from program text and associated documentation. The first two kinds come from specifications for externally invocable operators, namely user-callable and command procedures, and operators that receive control as a result of "asynchronous" events, for example interrupt handlers and PL/1 "on" routines.

The third kind comes from control constructs found in target programs. These are of three types:

** Invocation, branching, and iteration statements (e.g., "call", "if-then" and "do") that specify initiation of processes and subprocesses at designated control points.

** Process synchronization statements (e.g., "wait") and implicit delay specifications that determine the relative rates of processes along control paths.

** Process and subprocess termination statements (e.g., "exit" and "return").

The fourth kind of information comes from specifications for data objects involved in the descriptions of the conditions referred to in (2) and (3), together with program statements (e.g., "assignment" statements) in which the values of those objects are modified.

To determine potential concurrencies, however, it is necessary not only to obtain the above types of information, but to actually determine whether and when the conditions involved in (2) and (3) can actually be achieved. Questions of "when" must be answered in terms of potential process occupancies in various control segments--just the type of problem being attempted to be solved in the first place.

The potential concurrency problem can also be characterized as that of determining classes of states achievable by a system in which various local state transitions take place whenever prerequisite conditions occur. A tool that suggests itself for problems of this

type is the Petri net. [Peterson 77] Control path segments can be represented in Petri nets as places, processes occupying them as tokens, and control points at which processes are synchronized as transitions. However, no general and effective method exists for representing object states and conditions in a Petri net; it cannot currently be recommended as an immediately practical tool.

A third strategy is to take a conservative view by assuming that all access sequences to sharable objects are critical and represent potentially conflicting concurrencies, unless these are clearly made impossible either by explicit invocations of serialization mechanisms or by other serializing program logic. The steps to be taken in this approach are the following:

1. Identify access intervals to sharable objects.

2. For each such access interval determine whether adequate serialization specifications exist to insure that the accesses specified will always be consistent with propriety and compatibility phases of the objects accessed.

The first step may proceed either by first identifying actual access intervals in the target system programs and eliminating those that involve only accesses to obviously nonsharable objects, or by first identifying all specifications for sharable objects themselves (e.g., from data declarations) and then all access intervals to those objects. The problem of finding all accesses to given objects is made difficult in current operating systems by possibilities for dynamic bindings and address calculations and by the use of pointers. Apparently the most straightforward method is simply to examine access intervals directly.

The conservative approach turns out to be too conservative, however. One finds that a great many of the access intervals examined are not serialized in any explicit or obvious manner, and that one must resort to deeper analysis of the type indicated with the second strategy above, to determine whether explicit serialization is needed, or whether program logic, possibly distributed over a number of procedures, proscribes all possible access conflicts. For example, analysis might show that process concurrencies in one or more access intervals to a given object are in fact impossible because of conditions governing the movement of processes toward or around such intervals. Or, explicit invocations of serialization mechanisms (e.g., locking of objects, or access via monitors) may occur only in connection with modification accesses, but may result in all attempted use accesses being automatically suspended during periods of modification, without any associated explicit serialization at all.

Where critical intervals are identified which have been explicitly serialized by specifications for invocations to serialization mechanisms provided by the underlying system, these specifications must be determined to be correct (not to mention the

mechanisms themselves). In cases where critical intervals to the same objects should be mutually serialized, errors can occur because of improper object designations, e.g., designations that actually translate to different objects, or that refer to different structural levels of the same object, or to different levels of abstraction.

In some cases it may be difficult with any of the above strategies to distinguish potential sharable objects in the first place. An object is concurrently sharable if it exists during the lifetimes of two or more concurrent processes or subprocesses, and if it can be designated by more than one such process or subprocess. Operating systems provide many ways for objects to be bound into the name or address spaces of processes, or for processes to explicitly obtain the names or descriptions of objects, or pointers to them.

Finally, in general it can be very difficult to identify sequences of accesses intended to be completed as a unit, the individual accesses of which may be distributed over a number of procedures, or even intended to be executed as a part of more than one process.

Many of the difficulties of detecting serialization errors could be alleviated if operating systems implementation languages provided, and required the use of, more constrained constructs to name objects and control the bindings of names to objects; to specify the beginnings and ends of access sequences of various types; and, in general, to specify relative orderings among events, making it easier to answer questions of the form

"Can event e1 occur before event e2?"

A general method for detecting serialization errors is equivalent to a general method for answering such questions.

In view of the lack of a complete and effective strategy and the shortage of tools, any search for serialization errors in a large operating system that is intended to be exhaustive, would require an immense effort and would most likely result in the detection of only a fraction of the existing errors. Nevertheless, in many situations in which security has a high value, the expected payoff might be sufficient to make worthwhile a limited search, one that concentrates on the most critical areas of the target system.

REFERENCES

[Alexander 74] Alexander, William Preston III, *Analysis of Sequencing in Computer Programs and Systems*, University of Texas at Austin, Department of Computer Sciences, TR-35, August 1974.

[Bisbey 75] Bisbey, Richard II, Gerald Popek, and Jim Carlstedt, *Protection Errors in Operating Systems: Inconsistency of a Single Data Value Over Time*, USC/Information Sciences Institute, ISI/SR-75-4, December 1975.

[Bisbey 76] Bisbey, Richard II, Jim Carlstedt, Dale Chase, and Dennis Hollingworth, *Data Dependency Analysis*, USC/Information Sciences Institute, ISI/RR-76-45, February 1976.

[Campbell 74] Campbell, R.H., and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," in *Lecture Notes in Computer Science*, Vol. 16: *Operating Systems*, Springer-Verlag, 1974, pp. 89-102.

[Carlstedt 75] Carlstedt, Jim, Richard Bisbey II, and Gerald Popek, *Pattern-Directed Protection Evaluation*, USC/Information Sciences Institute, ISI/RR-75-31, June 1975.

[Carlstedt 76] Carlstedt, Jim, *Protection Errors in Operating Systems: Validation of Critical Conditions*, USC/Information Sciences Institute, ISI/SR-76-5, May 1976.

[Chow 76] Chow, Tsun S., "A Generalized Assertion Language," *Proceedings of the Second International Conference on Software Engineering*, October, 1976. [IEEE Cat. No. 76CG1125-4-C 392-399]

[Courtois 71] Courtois, P.J., F. Heymans, and D.L. Parnas, "Concurrent Control with 'readers' and 'writers'," *Communications of the ACM*, Vol. 14, No. 10, October 1971, 667-668.

[Dijkstra 68] Dijkstra, E.W., "Cooperating Sequential Processes," in F. Genuys (ed.), *Programming Languages*, Academic Press, New York, 1968.

[Dijkstra 71] Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, 1(1971), 115-138. Also in Hoare & Perrot (eds.), *Operating Systems Techniques*, Academic Press, 1972, pp. 72-93.

[Eswaran 76] Eswaran, K.P., J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Data Base System," *Communications of the ACM*, Vol. 19, No. 11 (November 1976), 624-633.

[Gilbert 72] Gilbert, Philip, and W.J. Chandler, "Interference Between Communicating Parallel Processes," *Communications of the ACM*, Vol. 15, No. 6 (June 1972), 427-437.

[Graham 68] Graham, Robert M., "Protection in an Information Processing Utility," *Communications of the ACM*, Vol. 11, No. 5, May 1968, 365-369.

[Gray 75] Gray, J.N., R.A. Lorie, G.R. Putzolu, and I.L. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, IBM Research Report RJ 1654 (24264), September 19, 1975.

[Habermann 75] Habermann, A.N., *Path Expressions*, Carnegie-Mellon University, Department of Computer Science, AFOSR-TR-75-1292, June 1975. [NTIS: AD-A015 842/8GA]

[Hoare 74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol. 17, No. 10 (October 1974), 549-557.

[Horning 73] Horning, J.J., and B. Randell, "Process Structuring," *ACM Computing Surveys*, Vol. 5, No. 1 (March 1973) 5-30. [Errata: Vol. 5, No. 3, September 1973, p.198]

[Hollingworth 76] Hollingworth, Dennis, and Richard Bisbey II, *Protection Errors in Operating Systems: Allocation/Deallocation Residuals.* USC/Information Sciences Institute ISI/SR-76-7, June 1976.

[Howard 76] Howard, John H., "Proving Monitors," *Communications of the ACM*, Vol. 19, No. 5 (May 1976), 273-279.

[Keller 76] Keller, Robert M., "Formal Verification of Parallel Programs," *Communications of the ACM*, Vol. 19, No. 7 (July 1976), 371-384.

[Lipton 75] Lipton, Richard J., "Reduction: A Method of Proving Properties of Parallel Programs," *Communications of the ACM*, Vol. 18, No. 12 (December 1975), 717-721.

[Lomet 77] Lomet, D.B., "Process Structuring, Synchronization, and Recovery Using Atomic Actions," *SIGPLAN Notices*, Vol. 12, No. 3 (March 1977). [Also, *Proceedings of ACM Conferences on Language Design for Reliable Software*, 128-137]

[Neumann 77] Neumann, Peter G., Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson, *A Provably Secure Operating System: The System, Its*

Applications, and Proofs, Stanford Research Institute Final Report, Project 4332, February 11, 1977.

[Owicki 76] Owicki, Susan, and David Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, Vol. 19, No. 5 (May 1976), 279-285.

[Peterson 77] Peterson, James L., "Petri Nets," *ACM Computing Surveys*, Vol. 9, No. 3 (September 1977), 223-252.

[Popek 75] Popek, Gerald J., and Charles S. Kline, "A Verifiable Protection System," *SIGPLAN Notices*, Vol. 10, No. 6, June 1975. [Also, *Proceedings of the 1975 International Conference on Reliable Software*, 294-304]

[Saltzer 75] Saltzer, Jerome H., and Michael D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, Vol. 63, No. 9, September 1975, 1278-1308.

[Schroeder 75] Schroeder, Michael D., "Engineering a Security Kernel for Multics," *Operating Systems Review*, Vol. 9, No. 5 (Special Issue: Proc. of the Fifth Symposium on Operating Systems Principles), 1975, 25-32.