MIT/LCS/TR-203

SYNTHESIS OF SYNCHRONIZATION CODE FOR DATA ABSTRACTIONS

by

Mark Steven Laventhal

June, 1978

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant DCR74-21892.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts
02139

MIT/LCS/TR-203

SYNTHESIS OF SYNCHRONIZATION CODE FOR DATA ABSTRACTIONS

by

Mark Steven Laventhal

June, 1978

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant DCR74-21892.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts
02139

SYNTHESIS OF SYNCHRONIZATION CODE FOR DATA ABSTRACTIONS

by

Mark Steven Laventhal

Submitted to the Department of Electrical Engineering and Computer Science on June 23, 1978 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

ABSTRACT

Synchronization code is necessary to control shared access of an abstract data object in a parallel-processing environment. This thesis explores an approach in which a synchronization property can be specified in a high-level nonprocedural language, and an implementation for the specified property can be synthesized algorithmically. A *problem specification* language is introduced in which synchronization properties can be expressed in a structured but natural manner. A method is then presented for synthesizing an implementation. An intermediate form, called a *solution specification*, is first derived, representing an abstract solution to the problem. The derivation of the solution specification accomplishes the transformation of the specification from nonprocedural to procedural form. The solution specification can be translated directly into a source language synchronization mechanism, such as a monitor.

Specifications for common synchronization properties, such as the readers-writers and bounded buffer problems, are expressed in the problem specification language. Corresponding implementations are then synthesized for these problems. In addition, the derived solution specification can be used in analyzing the soundness of the original problem specification with respect to criteria such as freedom from deadlock and starvation.

THESIS SUPERVISOR: Barbara H. Liskov
TITLE: Associate Professor of Electrical Engineering and Computer Science

Keywords: synchronization, synthesis, data abstractions, abstract data types, concurrency, interprocess communication, monitors, deadlock, starvation

Acknowledgments

I wish to thank a number of people who have contributed in various ways to my completing this thesis. First of all, I want to express my appreciation to my thesis supervisor, Barbara Liskov, for all the help she has given me. Not only has her technical advice invariably been sound, but her patience, encouragement, and support during my many years as a graduate student have been invaluable.

Each of my three readers, Irene Greif, Carl Hewitt, and Liba Svobodova, has contributed important insights to different aspects of both the research and the presentation of this thesis. My sincere gratitude goes to all three of them.

Many of the graduate students in the M. I. T. Laboratory for Computer Science have helped to create an interesting, stimulating, often diverting, and always supportive atmosphere in which to work. I want to thank in particular my officemates Dean Brock and Toby Bloom.

Finally, I wish to thank my wife Carol for her deep and constant support and encouragement. It is she who has enabled me to persevere throughout my graduate school career, and from whom I derive my inspiration.

Table of Contents

Chapter 1

Introduction and Background

1.1 Goals of the thesis

This thesis is concerned with the problem of synchronizing accesses by concurrently executing processes on a shared data object. Overall the thesis has two major goals. One is to design a high-level language in which synchronization properties can be specified in a nonprocedural form. The other is to devise a method for translating such specifications into actual source language code that implements the specified properties.

The reliability of computer software has received a great deal of attention in recent years. The reasons are both economic and intellectual. Rapid advances in hardware technology have dramatically decreased the cost of hardware relative to software, as well as expanded the range of complex computer applications for which new software is required. As a result, the cost of producing and maintaining software has become more than ever a major concern. Since testing and debugging incorrect programs consume a large share of total software costs, methods for improving the reliability of software are increasingly important from an economic viewpoint. At the same time, the intellectual difficulty of producing high quality software has become more generally appreciated. The study of how to produce complex yet reliable software systems represents a fertile area for research.

One productive approach in this area has been the study of language support to enhance software reliability. The range of current work in the area is quite broad, as illustrated by [LDRS77]. A particular aspect of this approach that has received wide attention has been the idea of abstract data types [Lis74]. Language support for abstract data types gives programmers a facility for implementing data abstractions analogous to the capability provided by procedures for functional abstractions. Following a methodology using data abstractions has been found to be a significant aid in producing reliable software.

A number of languages have been developed, and in many cases implemented, that include mechanisms supporting the concept of abstract data types (e.g. [Lis77], [Sha77], [Ges77]). Because of a lack of facilities in these languages for creation of multiple concurrent processes and interprocess communication, their range of programs until recently has been restricted to single-process computations. However, it is obvious that many of the kinds of applications for which the reliability provided by data abstractions are needed, such as operating systems, require such multiprocessing capabilities. In introducing facilities for concurrency and interprocess communication into these languages, it is necessary to do so in a manner that maintains the philosophy and methodology that such languages support.

This thesis explores a particular approach to a key problem in this area. The issue is the proper synchronization mechanism for a language that supports an abstract data type mechanism. Specifically, it is assumed that objects of abstract types in the language are shared among different processes and can be accessed concurrently. This means that some sort of synchronization mechanism is required to regulate these concurrent accesses. Synchronization may be required both to maintain the internal consistency of the objects and to implement higher-level scheduling decisions.

The approach taken here involves specifying synchronization properties in a high-level nonprocedural language, and obtaining automatically an implementation for the specified property. Synchronizing concurrent accesses to data can be a complex, error-prone task. Since the reliability of programs that access shared data depends upon the correctness of the synchronization, it is highly desirable that the synchronization itself be implemented as reliably as possible. If a specification language can be developed that is powerful enough to express synchronization properties of interest, and for which implementations can be synthesized automatically without too much effort, then it can be incorporated into a source language that supports data abstractions. Programs in the source language can specify synchronization properties nonprocedurally at a high level, and the compiler can produce the actual code using the synthesis algorithm. This would be a very attractive alternative to the range of synchronization mechanisms currently available, some of which are surveyed in the next section.

1.2 Synchronization mechanisms

Whenever concurrent processes share access to common resources, it is necessary that accesses by different processes be coordinated. The purpose of synchronization code, in the broadest sense, is to bring about this coordination. One kind of coordination involves limiting the combinations of simultaneous accesses allowed on a resource. That is, it is sometimes necessary for certain accesses to exclude others from taking place at the same time. This may be because the resource can inherently support only a limited number of concurrent accesses. For instance, a physical device such as a card reader must be devoted to a single process at a time. Alternatively, the nature of the accesses may be such that certain kinds of accesses performed concurrently would lead to inconsistent results, such as the case of two simultaneous updates to a database.

When certain accesses are prevented from occurring immediately, provision must be made for these deferred accesses eventually to take place. This is another aspect of coordination that must be handled by the synchronization code. Not only must a mechanism exist for deferring accesses. Decisions must be made as to when deferred accesses should occur, and these accesses must be activated in some way.

In working on synchronization problems, it has been found that writing synchronization code is a conceptually difficult task, more difficult in general than writing sequential programs. This difficulty arises from the non-intuitive nature of many problems that arise in synchronization, and the combinatorial problem associated with different possible sets of concurrent accesses on a resource. Therefore, several generations of synchronization mechanisms have evolved, reacting to the increasing complexity of concurrent programming applications, and to the resulting need for better, more well-structured synchronization mechanisms.

Originally, concurrent processes communicated through common shared storage. Access to this common storage was usually controlled by "locks", which were set prior to accesses and reset afterwards. Setting a lock was accomplished by means of an indivisible "test and set" instruction, usually implemented in hardware. This mechanism was quite unstructured, and certainly did not provide great confidence in its reliability. In addition, locking protocols involved "busy waiting", so that a process prevented from performing an access because of an already set lock was forced to perform essentially useless computation while waiting for the lock to be reset. With the advent of multiprocess time-sharing systems, this became unacceptable.

An important step forward was the development of the *semaphore* mechanism [Dij68], on which two operations are possible. Operation P accomplishes a "test and decrement" instruction, similar to setting a lock. However, the result of an unsuccessful "test" is to block the given process and place it on a queue associated with the semaphore. This eliminates the need for busy waiting. Operation V increments the semaphore and dequeues a process from the associated queue. With processes communicating via semaphores and using just these two operations, nearly all common synchronization problems can be solved. In addition to solving the busy waiting problem, semaphores, unlike locks, can be required to be fair. This means that service is granted in such a way that a given process is not kept waiting indefinitely while an arbitrary number of other processes proceed.

A complete generation of alternative mechanisms then appeared, all of them in some way variations on the semaphore concept. The proposed alternatives were designed to improve somewhat on the power of the semaphore mechanism. A difficulty common to semaphores and these alternative mechanisms became apparent, however. They are at too low a level, comparable to *goto* statements in the area of control structures. While sufficiently powerful to solve synchronization problems, they do not provide the programmer with enough structure to make these solutions easy to construct and reliable in operation.

Recent emphasis on "structured programming" [Dij72a] and language constructs suitable for producing reliable software has resulted in a new generation of synchronization mechanisms. Many of these new constructs attempt to internalize well-structured disciplines developed for the use of semaphore-style mechanisms, in much the same way that the *while* statement internalizes a structured style of writing loops originally developed using *goto* statements. Among the noteworthy mechanisms in this group are conditional critical regions [Bri72] and monitors [Hoa74], both of which embody the idea of accessing shared data only in indivisible segments of code. Both also seek to relate the scheduling mechanism for deferred accesses directly to properties of the shared data as another step toward better structure. More recent alternatives have attempted to improve further on these mechanisms. For example, serializers [Hew77] have drawn on experience with the use of monitors to build even more structure into the mechanism, and thereby correct certain perceived deficiencies in the monitor construct.

It is certainly easier to program solutions to non-trivial synchronization problems using these well-structured mechanisms than with semaphores or the like. However, synchronization remains an area of great complexity, and thus unreliability, in any large concurrent system such as an operating system or database management system. There is still a large conceptual gap between one's understanding of a synchronization problem and the code one must write to solve it. This has motivated recent work whose goal is to allow the expression of synchronization problems in a more natural form, and in some cases, to obtain automatically an implementation for the specified property. Some of this work, and its relationship with this thesis, is discussed in the next section.

1.3 Specifications and synthesis

Originally, synchronization problems were expressed simply in natural language. The informality of such descriptions was a contributing factor to the unreliability of the "solutions" proposed, as well as a source of controversy over just what a problem description "really" meant. After the widespread acceptance of semaphores, many problems were expressed via a representative program using semaphores. The circularity inherent in such a description is obvious, since the solutions to the synchronization problems also used code involving semaphores, and the distinction between "problem" and "solution" became negligible. More importantly, the expression of a synchronization problem at the level of actual code, while bridging the gap between specification and program, left the same gap between people's intuitive understanding and the specification. The "correctness" of specifications remained problematic.

A number of informal arguments about the correctness of an algorithm or the meaning of a mechanism have relied on the notion of "state" to reason indirectly about the effect of synchronization code (e.g. [Hab72], [Bri72], [Owi75]). This approach was used by Hoare in constructing formal proof rules for monitors in [Hoa74]. However, such an approach does not really formalize the meaning of synchronization code and synchronization problems themselves, but only in their relation to a program or system as a whole. Issues of modularity make it desirable to formally specify synchronization behavior in isolation from the procedures being synchronized..

Recent efforts to create structures through which to express synchronization problems include [Rob75], [Owi76] and [Gri76]. [Gri76] contains in addition a system for synthesizing solutions from the specification language automatically. However, in all these cases what can be expressed is not a synchronization problem itself, but rather the abstract solution to the problem. This is an improvement over a "specification" in the form of a concrete program using semaphores, but it still does not allow the specification of a synchronization problem independent of its solution. In order to do so, it is necessary to have a nonprocedural language for describing synchronization behavior that is independent of notions of how to implement that behavior.

Path expressions [Cam74] are a nonprocedural language for expressing synchronization problems. In addition, implementations can be derived directly from path expression specifications. Path expressions represent the most nearly comparable work to this thesis, both in overall goals and in basic approach. A discussion and evaluation of path expressions will be deferred until the approach of the thesis has been fully presented.

A comparison of this approach with that of path expressions is presented in Section 8.4.

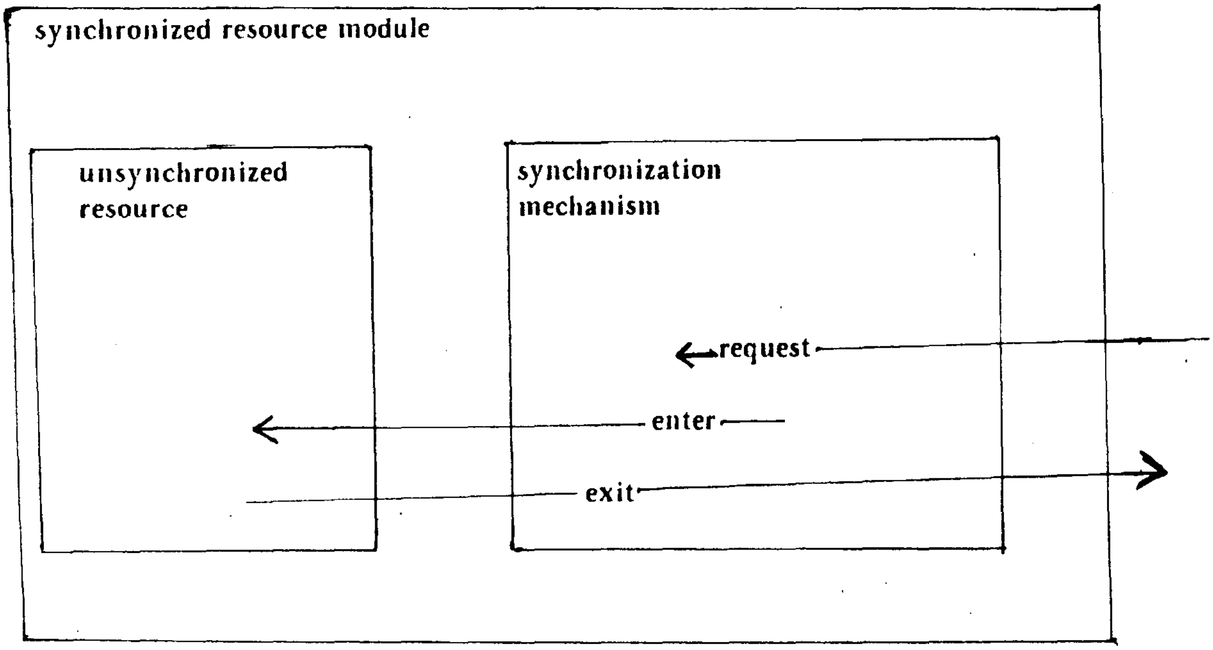
[Gre75] introduces a theory and notation for describing system behavior, including synchronization behavior. This theory involves the notion of events, over which a time ordering relation is defined. The notation introduced in [Gre75] is very general, in keeping with the abstract level at which events are discussed. The specification language used in this thesis represents one approach toward refining and structuring that notation.

1.4 Overview of the thesis

The view of synchronization taken in this thesis is illustrated in Figure 1.1, which illustrates the sequence of events involved in accessing a synchronized shared resource. This view shares with a number of other recent approaches the importance of *encapsulation.* The unsynchronized resource to be shared and the synchronization mechanism for that resource are encapsulated into a single "synchronized resource" module. The details of the coordination between the two are hidden from the outside world, which can only access the resource through this higher-level module.

The distinguishing features of the approach here concern the structure imposed on synchronized accesses of the resource. As indicated in the figure, every access involves a certain fixed sequence of events. The process wishing to make an access first communicates this desire to the synchronization mechanism, and this is denoted as the "request" for the access. When the synchronization mechanism permits the initiation of the access on the actual resource, the "enter" event occurs. The termination of the access is communicated to the synchronization mechanism in the "exit" event.

Figure 1.1. Accessing a synchronized resource



The specification language of this thesis is designed to describe properties concerning the time order of these abstract events. Chapter 2 presents this language, both its syntax and semantics, and includes a number of examples of its use. The synthesis of an implementation for the specified property is described in Chapters 3 through 5. Chapter 3 describes the abstract solution specification structure, in which events are implemented by abstract notions called "gates". The algorithm for deriving an equivalent solution specification from a problem specification is presented in Chapter 4. Chapter 5 explains the implementation of a solution specification in actual code, where the abstract gates are replaced by procedures of a monitor. Several examples of complete synthesis for well-known synchronization problems are presented in Chapter 6. The detection of certain types of erroneous specifications, those that permit deadlock and starvation, is discussed in chapter 7. A summary and evaluation of the thesis is contained in Chapter 8.

Chapter 2

The Problem Specification Language

2.1 Introduction

The focus of this chapter is on the language used for expressing synchronization constraints on accesses to an abstract data object. Before the language itself can be presented, however, it is necessary to "set the scene" in terms of exactly what kind of data objects are being treated, what the nature of accesses to these objects is, and what kind of synchronization of these accesses is possible. These issues are discussed in the first two sections of this chapter. Then an overview of the language is presented, including some motivation. This overview should make it easier to understand the following two sections, which formally define the syntax and semantics of the language, respectively. The chapter concludes with some examples of using the language to express common synchronization problems.

2.2 Data abstractions and synchronization

The data objects with which this thesis is concerned are of the sort that are handled in a language supporting the notion of abstract data types, such as CLU([Lis77]) or Simula([Dah72]). A data object in one of these languages is strongly typed, which is to say that its data type is an integral part of the object, and represents a severe restriction on how the object can be used. In particular, there is associated with the abstract data type a set of basic procedures, or *operations*. An object of the type can only be accessed through these operations, or through higher-level procedures that themselves make use of the operations. Furthermore, it is only these operations that are allowed to manipulate the lower-level representation of the abstract object.

In general, an abstract object can be either mutable or immutable. An object is *mutable* if it has state, so that its behavior can change over time. Immutable objects do not have state, and once they are created they are fixed for all time. Thus they are not useful for communication between parallel processes, and consequently are not of great interest with regard to synchronization. The data objects treated throughout this thesis are generally mutable.

An operation of a data type whose objects are mutable can have the function of creating an object of the type with some (possibly parameterized) initial state, of accessing the object's state without modifying it, or of accessing and updating the state. Assignment of the object to a variable is not considered to be an operation on the object, but instead constitutes a (temporary) binding of the variable to the object. See [Sch78] for a more detailed discussion of the semantics of a language such as CLU.

Synchronization is considered here to impose a constraint on the otherwise unconstrained time ordering of accesses to an individual data object. By this model, the ordering among accesses to different objects is completely unconstrained, except for the normal sequencing order within each individual process. This means that if synchronization is required among accesses to several objects, then these objects must be collected together into a single composite object, with the synchronization applying to this new higher-level object. It is important to keep in mind that it is the accesses on an object itself, not on any particular variable that happens to be bound to that object, that are of interest. Concurrent processes that share access to a data object presumably employ different variables for the purpose of referring to it, but it is over the total set of all these accesses that synchronization is required.

This thesis will not be concerned at all with the exact mechanism by which there come to be concurrent processes, or with how such processes gain joint access to a shared data object. It is not important whether the processes represent concurrent users of a time-sharing system, or are created from one process by some sort of fork-join mechanism in the language. Nor does it matter if the shared object resides in some form of central library to which all processes have access, or if a reference to the object must be explicitly passed to each one. The issue of synchronizing accesses to an object by concurrent processes is independent of such concerns, and the work here applies regardless of how these issues are handled. The important point is that there are processes executing in parallel that concurrently access the shared object. Consequently constraints must be put on the time ordering of accesses to the data object, and this is the purpose of the synchronization.

A basic assumption in the approach of the thesis is that the units upon which synchronization should be performed are the basic operations of the abstract data type. It is felt that the type's operations are the right level at which to impose synchronization constraints. Only these operations are allowed to access and manipulate the more concrete data representation of the abstract object, and so it is here that decisions by the implementer of the abstraction as to what pattern of accesses is necessary to maintain internal consistency make sense. The centralization of these operations in a type module (such as a CLU cluster) permits a single expression of constraints to cover all accesses of the object. Since the language ensures that all accesses to the object are made through the basic type operations, the discipline required for synchronization can be enforced universally, which would not be true necessarily if higher-level procedures were chosen for synchronizing. On the other hand, to the user of an abstraction these operations are basic and the details of their implementation are unknown (and in fact can be changed without his/her knowledge). Synchronization constraints at any lower level, i.e. involving code *internal* to these operations, therefore would not be meaningful to the user. It is exactly at the level of the basic operations of a data abstraction that the two viewpoints of the implementer and of the user can and should be resolved in a smooth interface. This is true for the synchronization component of the interface just as much as for the data component.

A very strict division is assumed between the synchronization and data accessing functions involved in accessing a shared data object. This is based on the philosophy that the task of synchronization belongs in a separate language construct, whose sole function is synchronization. The operations of the abstract data type, on the other hand, should be completely unconcerned with this synchronization, and written assuming that synchronization exists that is sufficient to prevent any conflicts between concurrent operation activations. Synchronization is taken to be uniform across all objects of the same type, reflecting the belief that a type consists not only of data accessing operations but the synchronization on them as well. That is, all objects of a given type are synchronized in the same way. This means that the same (sequential) implementation of a data type and its operations can be used with different synchronization constraints, perhaps embodying alternative scheduling policies or maintaining different levels of consistency, to create different data types.

2.3 The guardian model of synchronization

The model of synchronization that I use assumes there to be an abstract protection mechanism that conceptually surrounds each abstract data object on which accesses must be synchronized. (Recall the picture given in Figure 1.1.) This mechanism ensures that the encapsulated synchronization mechanism, which I call the *guardian* of the data abstraction, monitors all communication with the object, in a similar manner to the "secretary" concept proposed in [Dij72b]. Through this monitoring, the guardian is able to maintain the *synchronization state* of the resource, an abstract representation of the history of accesses to the object. (This is to be contrasted with the "data state" of the abstract object, which is the state explicitly manipulated by the operations accessing the object.) The guardian uses the synchronization state information to temporarily block any process attempting an access that the guardian deems to be unsafe given its current state. The blocked process is allowed to proceed when the synchronization state has changed in such a way that the access can safely occur.

Accessing an abstract data object consists of invoking a procedure implementing one of the operations of the type to which the object belongs. A given procedure activation generates three distinct events that the guardian includes in the synchronization history of the abstract object. The first event occurs when the guardian first receives notice of the invocation of the given procedure by the user process. I term this the request event for the given procedure activation. A request event can be likened to the act of "taking a number" in a crowded bakery, and represents the very first externally visible occurrence associated with the particular procedure activation.

The next event occurs when the process actually gains access to the object by beginning execution of the invoked procedure. I call this the enter event for the activation. It is this event that often must be delayed by the guardian until it can safely occur. Once it has occurred, the process may be assumed to be executing the body of the procedure. No assumptions can be made as to the relative execution speeds of different activations.

When the process has completed execution of the procedure, it indicates this fact to the guardian and exits from the resource. This is the exit event, the last event involved in the activation. Frequently it is the exit event for one activation that triggers a delayed enter event for some other activation.

This model of synchronization, of course, was not conceived in a vacuum. It is the result of a careful study of the kinds of synchronization properties that appear in the literature, which presumably reflect the nature of real-world concerns. Procedure entry and exit are natural concepts to use, since the basis of many synchronization problems is specifying which combinations of procedure activations can be allowed to execute concurrently. Clearly the solution of such problems requires that a record be kept of which procedure activations are currently executing, that is to say, which activations have entered but not exited. Another large class of synchronization properties, constituting what are usually regarded as "scheduling" properties, involve decisions as to which of a collection of processes each waiting to execute some procedure is allowed to proceed first. In order to deal with such properties, it is important to keep track of what activations have been requested, hence the need for request events. My investigation of synchronization problems has failed to discover any other distinguished events associated with operation activations that are as fundamental as these three. Since this model appears adequate for capturing synchronization properties of interest, there seems to be no need for using a more complicated one. The examples at the end of this chapter, written in the problem specification language that is based on the guardian model, testify to its generality.

The guardian model assumes that the set of all events concerning a particular data object is totally ordered. That is to say, while many procedure activations can be executing concurrently, only one request, enter, or exit event associated with a given object can occur at a time. This total ordering property is comparable to the fact that the "arrival ordering" for any particular actor in [Hew73] is total, and relies ultimately on some sort of "arbiter" mechanism for each data object.

2.4 Overview of the language

The purpose of the problem specification is to express, in a clear and concise manner, an imposed constraint on the temporal order of accesses to abstract data objects of a particular type. To facilitate this goal, the language for expressing the specification has been designed to be as general as possible, subject to the requirement that it be compatible with the guardian synchronization model. That is, the guardian model paradigm of request - enter - procedure body execution - exit forms the basis of the language, but beyond this, the complete freedom of first-order predicate calculus with equality and ordering among integers is available. Because of the power of predicate calculus, any meaningful synchronization constraint that operates on the level of the time ordering of individual events can be expressed.

This power, in fact, permits specifications to be written that must be judged erroneous. Such an invalid specification may, for instance, place a constraint on the circumstances under which a particular request event can occur, which would be incompatible with the guardian model. For certain kinds of erroneous specifications, the invalidity can be discovered in attempting to apply the synthesis algorithm presented in Chapter 4. The detection of other undesirable properties, namely deadlock and starvation, can take place after the synthesis is performed, and this is the subject of Chapter 7.

A specification is written for an abstract data type, and is intended to apply independently to every object of that type. The specification expresses a constraint on the ordering of accesses to the object, and represents the only such constraint. This means that any ordering of events that is consistent with the specification is valid, and in particular that procedure activations are allowed to execute in parallel unless constrained otherwise by the specification.

The distinctive elements of the specification language concern events and their ordering in time. Time ordering between events is embodied in the "temporally precedes" relation, which is denoted by the infix symbol " $\Rightarrow$ ", and which is adapted from [Gre75]. This relation is a strict partial order, transitive and anti-symmetric. The parallelism in a computation prevents the ordering from being total, but the set of events associated with accesses of a particular abstract data object is assumed to be totally ordered, as explained previously.

Each activation of a basic operation on a given abstract data object is identified by the name of the procedure being called and the activation number. Procedure activations are numbered uniquely for each data object according to the (total) ordering of the request events associated with the activations. The convention used here is that activation numbers are written as subscripts to the procedure name. The sixth activation of procedure p (i.e. the activation associated with the sixth request for p) therefore is denoted "p_6".

A particular event associated with an access is denoted by adjoining to the procedure activation formula the event type (request, enter, or exit) as a superscript. For example, the exit event associated with procedure activation p_6 is denoted "p_6^{exit}." Every event belongs to an *event class*, e.g. the p^{enter} event class consists of the events p_1^{enter}, p_2^{enter}, etc.

Activation numbers appearing in a specification can be any integer expressions, with important special cases being integer constants and variables. Constant activation numbers can be used to refer to a specific event of a particular class, such as the first one in a history. Variable activation numbers are more generally useful, though, since they allow reference to a general member of an event class. In the absence of explicit quantification, activation number variables are assumed to be universally quantified. This is a useful convention, permitting a specification that refers to event p_i^{enter}, for example, to represent a constraint on the enter event of *any* activation of procedure p. The use of expressions as activation numbers allows a specification to deal with related activations, such as p_i and p_{i+1}.

It is possible, but not necessary, to include the arguments to procedure activations. If not included, they are assumed to be unimportant, and the specification applies to any activation of the particular procedure. Including the arguments to an activation can be useful for constraining these arguments in some way, and thereby limiting the applicability of the specification to those activations whose arguments meet the constraint. The identifier of the process making the procedure activation can be used as one of the arguments of the procedure, so that if the identity of the particular process is important, it can be included in this way.

The actual abstract data object on which the synchronization is being performed is *not* included as an explicit argument to any of the procedures operating on it. In this respect, this kind of specification resembles the "state machine" specifications used by Parnas for specifying the behavior of the operations of an abstract data type (see [Par72], e.g.). It can be assumed that operations are called by a mechanism such as the "dot" notation of Simula ([Dah72]), by which operation p on abstract object x with arguments a and b is called via the statement "x.p(a,b)". A specification referring to operation p might list arguments a and b explicitly, but no reference would be made to object x. The specification would implicitly apply independently to each object x of the given type.

As an example of a specification expressed in this language, consider the following expression, which also appears as example 1 in Section 2.7:

$$(p_i^{enter} \Rightarrow q_j^{enter}) \supset (p_i^{exit} \Rightarrow q_j^{enter})$$

This specification refers to two procedure activations, p_i (the i-th activation of procedure p) and q_j (the j-th activation of procedure q). Variables i and j appear free in the expression and therefore are universally quantified, and since no constraints are placed on the arguments to the procedure activations, the specification in fact applies to *any* activations of procedures p and q.. The specification states that if the enter event for q_j is preceded by the enter event for p_i, then it is also preceded by the exit event for the same activation of p. That is, a currently executing activation of procedure p (on a given object) excludes a subsequent activation of procedure q (on the same object) until the activation of p is completed. Notice, though, that concurrent activations of p and q are allowed, as long as the activation of q begins (i.e. enters) first.

2.5 Syntax of the language

This section presents the syntactic rules for well-formed specifications. The notions *identifier* and *arithmetic expression* are assumed to be basic. An *arithmetic expression* is a series of one or more identifiers and/or integer constants separated by the usual arithmetic operations. The other notions are defined in terms of these two and each other. In each rule the concept being defined appears in italics:

(1) A *procedure name* is an identifier.

(2) A *term* is an arithmetic expression.

(3) An *activation number* is a term.

(4) An *activation name* is a procedure name, subscripted with an activation number.

(5) An *activation expression* is either an activation name, or an activation name followed by a left parenthesis, followed by one or more terms separated by commas, followed by a right parenthesis.

(6) An *event type* is one of the elements of the set {**request** , **enter** , **exit**}.

(7) An *event expression* is an activation expression superscripted with an event type.

(8) An *ordering clause* is an event expression followed by the symbol $\Longrightarrow$ followed by another event expression.

(9) An *arithmetic relation* is one of the elements of the set

$$\text{Rel} = \{=\,,\neq\,,<\,,>\,,\leq\,,\geq\}$$

(10) An *argument constraint* is a term followed by an arithmetic relation followed by another term.

(11) A *clause* is either an ordering clause or an argument constraint.

(12) A *specification* is defined by:

(a) A clause is a specification.

(b) If S is a specification, then $(\neg S)$ is a specification.

(c) If S_1 and S_2 are specifications and op is an element of the set

$$\text{Op} = \{\wedge\,,\vee\,,\supset\,,\leftrightarrow\},$$

then $(S_1 \text{ op } S_2)$ is a specification.

(d) If S is a specification and i is an identifier, then $\forall\, i\, (S)$ and $\exists\, i\, (S)$ are specifications.

The "argument constraints" defined in rule (10) may refer to the activation numbers and/or to the arguments to the activations (which are the "terms" in rule (5)). They may *not* refer to the actual abstract data object in question, however, since it does not appear as an explicit argument to any of the procedures. In fact, a general rule is that the arguments of procedure activations to which predicates may refer are limited to immutable objects, such as integers. The interpretation of a relation on a mutable object would depend upon the point in time at which the relation is taken to apply, and might itself require synchronization on the given object. Rather than becoming involved in questions such as these, I choose to limit the predicates on activation arguments to immutable objects. This restriction does not appear to be severe.

2.6 Semantics of the language

The definition of the language whose syntax has been formally defined in the previous section can now be completed by means of a formal definition of its semantics. The purpose of the language is to express synchronization properties, that is, to constrain the order of accesses on an abstract data object. The semantics of the language therefore can be defined by specifying the collection of access histories that are valid with respect to any given specification in the language. This is accomplished by defining a predicate Valid(h, s), which decides for any history h and specification s whether h is a valid history with respect to the constraint expressed in s. First, however, it is necessary to define the concept of a history, and to restrict the concept to histories that are physically possible.

The first step in this process is to define the notion of "event". An *event* is a 5-tuple <p, t, x, n, a>, such that:

(1) $p \in P$, the set of basic operations of all types.

(2) $t \in ET$, the set of event types, where ET = {request, enter, exit}.

(3) $x \in Ob$, the set of all data objects in the system, and p is a basic operation for the type of x. x is the data object on which the access is taking place.

(4) n $\in$ N*, the set of positive integers. n represents the activation number.

(5) a is a vector [a_1, ... , a_m], where each element $a_i \in$ Ob. a is the vector of arguments to p.

The types of the objects a_1, ... , a_m must match the types of the parameters to operation p.

A partially ordered set of events forms a *computation history*, provided that the partial order fulfills the condition that each object history is totally ordered. An *object history* for data object z is a subset of a computation history, consisting of all events <p, t, x, n, a> in the computation history such that x = z. All events in an object history are on the same data object, so that the third component x of each event tuple can be eliminated, and each element of an object history is simply a 4-tuple <p, t, n, a>. Throughout the rest of this section, we will be concerned exclusively with object histories, though the simple term "history" will be used.

Since the events in a history are totally ordered, the history may be considered to be a sequence of events. A *sequence* over a domain D can be defined as either the empty sequence [ ], or else the result of adding an element d $\in$ D to the end of a sequence s, which is given by the expression "add(s, d)".

Not all histories are actually possible. In order to define what class of histories are possible, some further definitions are required. An *event class* for a data type dt is a pair <p, t>, where p $\in$ P and t $\in$ ET, and p is a basic operation of data type dt. The set of *occurrences* of an event class <p, t> in a history h is a set of pairs of the form <n, a>, where n is an activation number and a is a vector of arguments, such that an event of the form

<p, t, n, a> occurs in history h. Formally, this is given by Occurrences(h, <p, t>), where:

$$\begin{aligned}
&\text{Occurrences}([\,], \langle p, t\rangle) \;=\; \{\,\} \\
&\text{Occurrences}(\text{add}(h, \langle p_1, t_1, n, a\rangle), \langle p, t\rangle) \;= \\
&\qquad\qquad \text{if } (p = p_1 \wedge t = t_1) \text{ then Occurrences}(h, \langle p, t\rangle) \cup \{\langle n, a\rangle\} \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{else Occurrences}(h, \langle p, t\rangle)
\end{aligned}$$

With the aid of these definitions, we can now define when an history is possible. The predicate Possible requires a request event to precede the corresponding enter event, which in turn must precede the corresponding exit event. Also the ordering of request events for a given procedure must determine the numbering of invocations.

$$\begin{aligned}
&\text{Possible}([\,]) \;=\; \text{TRUE} \\
&\text{Possible}(\text{add}(h, \langle p, t, n, a\rangle)) \;= \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{Possible}(h) \;\wedge \\
&\quad ((t = \text{request} \;\wedge\; \text{Occurrences}(h, \langle p, \text{request}\rangle) = \{\langle i, a_i\rangle \mid 1 \leq i < n\}) \;\vee \\
&\qquad\qquad (t = \text{enter} \;\;\wedge\; \langle n, a\rangle \in \text{Occurrences}(h, \langle p, \text{request}\rangle)) \;\vee \\
&\qquad\qquad\quad (t = \text{exit} \;\;\wedge\; \langle n, a\rangle \in \text{Occurrences}(h, \langle p, \text{enter}\rangle)))
\end{aligned}$$

A few more definitions are required before the validity of a possible history with respect to a specification s can be defined. An *event expression* is a 4-tuple <p, t, exp, v>, where $p \in P$, $t \in ET$, exp is an arithmetic expression, and **v** is a vector of arithmetic expressions, possibly empty. (The concept of *arithmetic expression* can be defined formally in the obvious manner.) Let the set of arithmetic relations Rel = $\{=, \neq, <, >, \leq, \geq\}$ and the set of logical binary operators Op = $\{\wedge, \vee, \supset, \leftrightarrow\}$. Then the set of event expressions in a specification s is given by Evexp(s), which is defined in the obvious manner:

$\mathrm{Evexp}(e_1 \Rightarrow e_2) = \{ e_1, e_2 \}$

$\mathrm{Evexp}(\mathrm{exp}_1 \text{ rel } \mathrm{exp}_2) = \{ \}$, for rel $\in$ Rel

$\mathrm{Evexp}(\neg s) = \mathrm{Evexp}(s)$

$\mathrm{Evexp}(s_1 \text{ op } s_2) = \mathrm{Evexp}(s_1) \cup \mathrm{Evexp}(s_2)$, for op $\in$ Op

$\mathrm{Evexp}(\exists x (s)) = \mathrm{Evexp}(s)$

$\mathrm{Evexp}(\forall x (s)) = \mathrm{Evexp}(s)$

An *interpretation* is a mapping f from expressions to data objects that preserves the meaning of all constants and operations. That is:

(1) f maps every constant expression to the corresponding constant object,

e.g. $f(1) = 1$.

(2) f is consistent with every operation,

e.g. $f(\mathrm{exp}_1 + \mathrm{exp}_2) = f(\mathrm{exp}_1) + f(\mathrm{exp}_2)$.

(3) f maps a vector of expressions into the corresponding vector of objects,

e.g. $f(<\mathrm{exp}_1, ..., \mathrm{exp}_m>) = <f(\mathrm{exp}_1), ..., f(\mathrm{exp}_m)>$.

An event e and an event expression ee *match* under an interpretation f if e and ee are of the same event class, and f maps the activation number expression and parameter vector expression (unless the latter is empty) of ee to the corresponding components of e. Formally, Match(e, ee, f) is defined as:

$$\mathrm{Match}(<p_1, t_1, n, a>, <p_2, t_2, \mathrm{exp}, v>, f) =$$
$$(p_1 = p_2) \wedge (t_1 = t_2) \wedge (f(\mathrm{exp}) = n) \wedge (v = [\,] \vee f(v) = a).$$

The validity of a history with respect to a specification s can now be defined by a predicate Valid. The definition of Valid recursively determines when a history is valid with respect to a specification. For a history to be valid, the previous history consisting of all but the last event must first be valid. Furthermore the last event in the history must satisfy the specification for all interpretations under which the event matches some event expression in the specification.

Whether or not an event added onto a valid history *satisfies* a specification under an interpretation is defined by another predicate Sat. The definition of Sat for a complicated specification is basically just a matter of breaking down the structure of the specification, by removing each logical operator and applying it to the recursive applications of the definition, until one reaches the level of a simple clause. Satisfaction of an argument constraint is determined solely by how the components of the clause are embodied by the given interpretation, not by the event in question. Whether an event satisfies an ordering clause depends upon whether the event matches one of the event expressions in the clause under the interpretation. If the event matches the first event expression under the given interpretation, then it is necessary that no event matching the second event expression occurs in the previous history. If the event matches the second event expression, though, then some event matching the first event expression must occur in the history.

Formally, if h is a possible history and s is a specification, then h is *valid* with respect to s if and only if Valid(h, s), where:

$$\begin{aligned}
&\text{Valid}([\,], s) = \text{TRUE} \\
&\text{Valid}(\text{add}(h, e), s) = \text{Valid}(h, s) \wedge \\
&\qquad\qquad \forall\, (ee, f)\ (ee \in \text{Evexp}(s) \wedge f \text{ is an interpretation} \\
&\qquad\qquad\qquad \wedge\ \text{Match}(e, ee, f) \supset \text{Sat}(h, e, s, f))
\end{aligned}$$

The predicate Sat(h, e, s, f) determines whether event e added to history h satisfies specification s under interpretation f. It is defined by the following equations, giving all possible cases for specification s:

$$\begin{array}{l}
\mathrm{Sat}(h, e, (\langle p_1, t_1, exp_1, v_1\rangle \Longrightarrow \langle p_2, t_2, exp_2, v_2\rangle), f) = \\
\qquad (\mathrm{Match}(e, \langle p_1, t_1, exp_1, v_1\rangle, f) \supset \\
\qquad\qquad ((v_2 \neq [\,] \wedge \langle f(exp_2), f(v_2)\rangle \notin \mathrm{Occurrences}(h, \langle p_2, t_2\rangle)) \vee \\
\qquad\qquad (v_2 = [\,] \wedge \forall a\, (\langle f(exp_2), a\rangle \notin \mathrm{Occurrences}(h, \langle p_2, t_2\rangle))))) \\
\qquad \wedge\ (\mathrm{Match}(e, \langle p_2, t_2, exp_2, v_2\rangle, f) \supset \\
\qquad\qquad ((v_1 \neq [\,] \wedge \langle f(exp_1), f(v_1)\rangle \in \mathrm{Occurrences}(h, \langle p_2, t_2\rangle)) \vee \\
\qquad\qquad (v_1 = [\,] \wedge \exists a\, (\langle f(exp_1), a\rangle \in \mathrm{Occurrences}(h, \langle p_1, t_1\rangle)))))
\end{array}$$

$$\mathrm{Sat}(h, e, exp_1\ rel\ exp_2, f) = (f(exp_1)\ \ rel\ \ f(exp_2)), \quad \text{for } rel \in \mathrm{Rel}$$

$$\mathrm{Sat}(h, e, \neg\, s, f) = \neg\, \mathrm{Sat}(h, e, s_1, f)$$

$$\mathrm{Sat}(h, e, s_1\ op\ s_2, f) = \mathrm{Sat}(h, e, s_1, f)\ \ op\ \ \mathrm{Sat}(h, e, s_2, f), \quad \text{for } op \in \mathrm{Op}$$

$$\mathrm{Sat}(h, e, \exists\, i\ (s), f) = \exists\, m\ \mathrm{Sat}(h, e, s[m/i], f)$$

$$\mathrm{Sat}(h, e, \forall\, i\ (s), f) = \forall\, m\ \mathrm{Sat}(h, e, s[m/i], f)$$

The notation s[m/i] in the last two equations represents the expression resulting from substituting m for all free occurrences of i in s.

2.7 Examples

This section presents a series of examples of the use of the problem specification language. These examples have been chosen with two criteria in mind. First, together they illustrate the range of features that the language offers. Second, they specify realistic and representative properties, covering a significant portion of the classic synchronization problems that appear in the literature.

Example 1: Exclusion

$$(p_i^{enter} \Longrightarrow q_j^{enter}) \supset (p_i^{exit} \Longrightarrow q_j^{enter})$$

This specification has been discussed previously in Section 2.4. It states that an activation of procedure p excludes a subsequent activation of procedure q until the activation of p is completed.

Example 2: Mutual exclusion

$$(p_i^{exit} \Rightarrow q_j^{enter}) \vee (q_j^{exit} \Rightarrow p_i^{enter})$$

This specification is similar to example 1, except that it is symmetric between procedures p and q. That is, an activation of either p or q excludes any concurrent activation of the other.

Example 3: Readers-writers property

$$((write_i^{enter} \Rightarrow write_j^{enter}) \supset (write_i^{exit} \Rightarrow write_j^{enter})) \wedge$$
$$((write_i^{exit} \Rightarrow read_k^{enter}) \vee (read_k^{exit} \Rightarrow write_i^{enter}))$$

The so-called *readers-writers* property concerns two operations, "read" and "write". It states that activations of "read" exclude those of "write", and that an activation of "write" excludes all other activations of either operation. This has been re-shaped into an instance of example 1 (an activation of "write" excludes all other activations of "write"), and an instance of example 2 (activations of "read" and "write" mutually exclude one another). By combining this specification with an instance of example 4, giving one of the operations priority over the other, or of example 5, requiring an equal-priority first-come-first-served discipline, one can obtain any of the classic versions of the readers-writers problem (as found, for example, in [Gre75]).

Example 4: Priority

$$(p_i^{request} \Rightarrow q_j^{enter}) \supset (p_i^{enter} \Rightarrow q_j^{enter})$$

This specification gives priority to activations of procedure p over those of procedure q. It does this by requiring that so long as the activation of q has not yet entered, then any activation of p that has been requested must enter first, regardless of whether the request event for p came after the request event for the activation of q. This is an example of a scheduling property making use of a request event.

Example 5: FCFS scheduling

$$(p_i^{\text{request}} \Longrightarrow q_j^{\text{request}}) \leftrightarrow (p_i^{\text{enter}} \Longrightarrow q_j^{\text{enter}})$$

This specification represents an alternative to giving either of a pair of operations priority over the other. Instead it requires a strict first-come-first-served discipline between them, by stating that whichever activation is requested first is the one to enter first.

Example 6: LCFS scheduling

$$(p_i^{\text{request}} \Longrightarrow p_j^{\text{request}}) \wedge (p_j^{\text{request}} \Longrightarrow p_i^{\text{enter}}) \supset (p_j^{\text{enter}} \Longrightarrow p_i^{\text{enter}})$$

Here another alternative scheduling policy, though probably a less likely one, is specified. This "last-come-first-served" property requires that of all the requested and pending activations of a given operation p, the one most recently requested is allowed to enter.

Example 7: Operation pairing

$$(a_i^{\text{enter}} \Longrightarrow b_j^{\text{enter}}) \leftrightarrow (c_i^{\text{enter}} \Longrightarrow d_j^{\text{enter}})$$

This specification requires that whichever order occurs between the entry of an activation of "a" and one of "b", the same order must hold for the corresponding activations of "c" and "d", respectively. Illustrated is the use of the same activation number for activations of different procedures, i for procedures "a" and "c", and j for procedures "b" and "d". The specification could be used for a data type in which operations a and b conflict, in the sense of updating the same part of the object's state, as do operations c and d. If operations a and c, taken as a pair, update the state consistently, and operations b and d do likewise, then the constraint specified here might be necessary to prevent an inconsistent update.

For example, in [Esw76], an example is given for which the operations have the following meanings:

```
a: x := x+10;
b: x := x*2;
c: y := y+10;
d: y := y*2;
```

If the predicate (x = y) is the criterion for consistency of the data object, then this would be part of the specification required. (Other constraints also would be necessary.)

Example 8: Producer-consumer (single buffer)

$$(\mathrm{dep}_i^{\mathrm{exit}} \Rightarrow \mathrm{rem}_i^{\mathrm{enter}}) \wedge (\mathrm{rem}_i^{\mathrm{exit}} \Rightarrow \mathrm{dep}_{i+1}^{\mathrm{enter}})$$

The "producer-consumer" problem is that producers and consumers must alternate in depositing and removing messages, respectively, in a shared buffer. This means that each deposit, represented here by an activation of procedure "dep", must precede the corresponding removal, or activation of procedure "rem". On the other hand, the removal must take place before the next deposit can occur. This specification again illustrates the use of the same activation number for activations of two different procedures, as well as the use of an expression ("i+1") as an activation number. Notice that this specification could be rewritten so as to make the relationships between activation numbers more explicit by means of predicates on the activation numbers:

$$(i = j) \supset (dep_i^{exit} \Rightarrow rem_j^{enter}) \wedge (rem_j^{exit} \Rightarrow dep_{i+1}^{enter})$$

This specification is exactly equivalent to the original; it makes no difference whether such relationships are represented explicitly or implicitly.

Example 9: Bounded buffer

$$(dep_i^{exit} \Rightarrow rem_i^{enter}) \wedge (rem_i^{exit} \Rightarrow dep_{i+N}^{enter}) \wedge$$
$$(dep_i^{exit} \Rightarrow dep_{i+1}^{enter}) \wedge (rem_i^{exit} \Rightarrow rem_{i+1}^{enter})$$

This example is a generalization of the previous one, in that the activation number of the dep^{enter} event has been changed from i+1 to i+N, for some integer N. The specification is for the same problem, except that the size of the buffer is now N. This means that up to N messages can be deposited in the buffer before filling it, so that up to N successive "dep" operations can be allowed before one has to wait for a "rem" operation. The last two clauses state that the individual "dep" activations must be mutually exclusive and execute in first-come-first-served order, as must the individual "rem" activations.

Example 10: Intervening activation

$$(p_i^{exit} \Rightarrow p_j^{enter}) \supset (\exists\, k\, (p_i^{exit} \Rightarrow q_k^{enter} \wedge q_k^{exit} \Rightarrow p_j^{enter}))$$

This specification represents a weaker property that is implied by the producer-consumer constraint of example 8. It requires that between any two activations of procedure "p" there must be an activation of procedure "q". This shows the use of an existential quantifier in a specification to require a particular kind of event to occur at a given point in the history.

Example 11: Threshold of requests

$$\forall\, i\, ((k \le i) \wedge (i < k+N) \supset (p_i^{request} \Rightarrow p_k^{enter}))$$

This specification places a threshold of N request events for activations of procedure "p" before the first one can execute. Since this applies to any value of k, the result is that whenever an activation of procedûre "p" is currently executing, there must be at least N processes that are waiting on requests to execute "p".

Example 12: Exclusion on a restricted class of accesses

$$(p_i(a)^{enter} \Rightarrow q_j(a)^{enter}) \supset (p_i(a)^{exit} \Rightarrow q_j(a)^{enter})$$

This specification is identical to example 1, except that a parameter has been given to each of the two procedure activations. By providing the same identifier as the argument to both activations, this specification conveys the information that the arguments to the two procedure activations are equal. Therefore the exclusion constraint expressed by this specification is restricted to activations with equal parameters.

Example 13: Predicate locks

$$C(a,b) \wedge (p_i(a)^{enter} \Rightarrow q_j(b)^{enter}) \supset (p_i(a)^{exit} \Rightarrow q_j(b)^{enter})$$

This specification again represents a restriction of the exclusion constraint of example 1. Here, though, the restriction is represented by a general predicates C on the parameters to activations p_i and q_j. This suggests how a simple version of the concept of "predicate locks" might be specified. A specification of this form can be used to state the synchronization constraint, as long as the predicate C for which exclusion is required is known ahead of time.

For example, suppose that the abstract data object on which procedures "p" and "q" operate is a hierarchically organized database. The database consists of a collection of files, each of which in turn consists of a collection of records. The predicate C might express the relation that records a and b are elements of the same file. Therefore, procedure "p" would exclude procedure "q" only when they were operating on records in the same file.

The general notion of "predicate locks" was introduced in [Esw76]. The more complicated versions of the concept discussed there would require more complex specifications.

Example 14: Disk head scheduling

$$\begin{gathered}
((a_z{}^{enter} \Longrightarrow a_y{}^{enter}) \supset (a_z{}^{exit} \Longrightarrow a_y{}^{enter})) \quad \wedge \\
((a_i(x2)^{request} \Longrightarrow a_k(x1)^{exit} \Longrightarrow a_i(x2)^{enter}) \wedge \\
(a_j(x3)^{request} \Longrightarrow a_k(x1)^{exit} \Longrightarrow a_j(x3)^{enter}) \wedge \\
(a_m(x0)^{exit} \Longrightarrow a_k(x1)^{exit})) \wedge \\
\neg \exists(n)\ ((a_m(x0)^{exit} \Longrightarrow a_n{}^{exit} \Longrightarrow a_k(x1)^{exit})) \quad \wedge \\
((x0 < x1 < x2 \ \wedge (x2 < x3 \ \vee \ x3 < x1)) \ \vee \\
(x0 > x1 > x2 \ \wedge (x2 > x3 \ \vee \ x3 > x1))) \\
\supset (a_i(x2)^{enter} \Longrightarrow a_j(x3)^{enter}))
\end{gathered}$$

The final example is the "disk head scheduler" problem, which appears in [Hoa74], among other places. The problem is to schedule disk accesses so as to minimize average waiting time. The way this is done is to have the disk head sweep in one direction, accessing each track it encounters for which an access has been requested, until no more requested tracks remain in the direction in which it is sweeping. The head then reverses direction and sweeps back, again accessing requested tracks as it encounters them. The essential idea is that at any given point, the next track to be accessed is the one closest to the currently accessed track in the direction currently being swept.

The specification for this problem concerns four activations of an access procedure "a" on a disk, with the parameter (x0, x1, x2, or x3) representing the number of the track being accessed. The constraint expressed is that of the two activations (a_i and a_j) requested during the time that another activation (a_k) is executing, the activation allowed to execute first is the one accessing the track nearest to the track currently being accessed (track x1) in the direction currently being swept. The direction is indicated by the inequality between x0, the track that most recently accessed, and x1. Track x2 is accessed before track x3 either because it is closer to track x1 (either $x1 < x2 < x3$ or $x1 > x2 > x3$), or else because it is in the right direction and x3 is not ($x3 < x1 < x2$ or $x3 > x1 > x2$).

Chapter 3

The Solution Specification

3.1 Introduction

There is a vast conceptual distance separating, on the one hand, a problem specification written in the language described in Chapter 2, and on the other, the synchronization code that implements the specification. This is because the specification is a non-procedural, requirements-oriented expression of *what* should happen with no indication of the means by which this behavior should be realized. Determination of the procedural mechanism, that is *how* to accomplish the desired constraint on the time order of accesses, requires a fundamental transformation in concepts. Once this determination has been made, there are still a number of details that need to be worked out, but the remaining work is basically that of the back end of a compiler, translating from an intermediate language into actual code (though the target code in this case is still in a high-level language, not machine language).

I have chosen to split the derivation process into two stages. The first stage is the transformation from procedural to nonprocedural form. It can be described without reference to the exact details of particular source language constructs. The second stage constructs an actual implementation. The intermediate form into which the problem specification is transformed by the first stage is called the *solution specification*. This chapter presents an informal description of solution specifications, followed by a formal definition of their semantics. The method for transforming a problem specification into an equivalent solution specification is the subject of Chapter 4. The translation of the solution specification into synchronization code is treated in Chapter 5.

Section 3.2 presents the "basic" structure of the solution specification, which is only a first approximation to the actual structure. The basic structure described is quite simple and elegant, and in fact the solutions to many synchronization problems can be expressed within it. Unfortunately, this simple structure lacks sufficient expressive power for certain important classes of problems. For this reason, it is necessary to augment the basic structure with additional features, which are described in Section 3.3. The formal semantic definition of the solution specification appears in Section 3.4.

3.2 The basic solution specification structure

The structure of the solution specification, as of the problem specification, is dictated to some extent by the guardian synchronization model. That is, the solution specification must contain features corresponding to those events associated with procedure activations that the guardian model distinguishes. Beyond this, there is some choice as to how rigid a structure to impose on the solution specification. Since the solution specification is an intermediate form between the problem specification and the generated code, the degree of flexibility represents to some extent where it lies on the spectrum between these two structures. A very general solution specification structure, corresponding to the generality of the problem specification language, would represent a decision that the solution specification be relatively close to the problem specification. The price paid for this generality would lie in the difficulty of translating such a solution specification into target code.

The alternative choice made here is for the solution specification to have a rather rigid structure. This means that, as indicated in the introduction to this chapter, the fundamental transformation takes place in deriving the solution specification from the problem specification.

The basic structure of the solution specification is for each guardian to consist of a collection of *gates* through which processes accessing the abstract data object must pass. The use of the term "gate" is taken from [Rob75], though the concept as used in this thesis differs somewhat from the one introduced there. Specifically, the guardian for an object of abstract data type t contains a gate for each event class of t. This means that for each operation p of the abstraction, there are gates $p^{request}$, p^{enter}, and p^{exit}. Each event associated with an object corresponds to the passage through a gate in its guardian. For a process to access the data object by activating procedure p, the process first must pass through the $p^{request}$ gate, then through the p^{enter} gate. At this point it executes the body of procedure p, after which it must pass through the p^{exit} gate.

Each passage through a gate by a process produces a (conceptually instantaneous) change in the state of the guardian. Because of the total ordering on the events associated with an object, the gate passages for a particular guardian are totally ordered. The ordering of processes passing through any single gate is first-come-first-served. This means that unless a specification explicitly requires a particular scheduling policy for activations of a given operation, the default policy assumed is first-come-first-served. The order of service among different gates of a guardian is assumed to be fair, in the sense that processes at different gates have equal chances of being chosen for service. That is, a requirement in the implementation is that a process cannot starve because of lack of attention from the scheduling mechanism.

Gates for request and exit event classes are unconditional, so that processes cannot be blocked in passing through these gates. A gate for an enter event class is conditional, however. Associated with each enter gate there is some condition on the guardian state. This condition must be satisfied in order for the process making the activation to pass through the gate. If a process attempts to pass through an enter gate whose condition is not satisfied, then the process is blocked, and must wait until the condition becomes true before proceeding through the gate.

Schematically, then, an activation of operation p on a data object is implemented by the abstract program below. Since gate passages represent events, which are totally ordered, the abstract code representing each gate can be considered an indivisible operation.

```
$p^{request}$: update guardian state
$p^{enter}$: wait until entry condition is satisfied,
        then update guardian state
execute body of operation p
$p^{exit}$: update guardian state
```

It would appear that to represent a given solution specification, it would be necessary to specify for each operation p the specific entry condition on gate p^{enter}, and the particular updates to the guardian state accomplished in each of the three gates. In fact, the form chosen for the synchronization state of a data object defines *a priori* the nature of the updates within all gates.

The history of a data object, and of the guardian for the object, consists of the totally ordered sequence of events associated with all accesses of the object in the entire computation. The state of the object represents some abstraction from the history that is sufficient for predicting its future behavior. An alternative way of saying this is the definition in [Gre75] that a state is an abbreviation for a class of histories. The *synchronization state* of the object is the synchronization component of the state, which is sufficient for the prediction of its future synchronization behavior.

The decision made here is to express the synchronization state of an object as the number of events that have occurred at each gate of its guardian. The notation used is that count(g) denotes the number of events at gate g. So count($p^{request}$) is the number of activations of procedure p that have been requested, whether or not those requests have been granted; count(p^{enter}) is the number of activations of p that have entered, whether or not they have exited; and count(p^{exit}) is the number that have exited.

This decision has a number of ramifications. The implications for the expressive power of the solution specification are discussed in the next section. The decision to use counts forms the basis for the method of deriving a solution specification from a problem specification, as will be apparent in the description of the derivation algorithm in Chapter 4. With respect to the basic struçture of the solution specification, it means that in the schematic abstract program representing an activation of operation p, each update to the guardian state now can be defined to be simply incrementing the proper count. The abstract program therefore becomes:

```
$p^{request}$: increment count($p^{request}$) by 1
$p^{enter}$: wait until entry condition is satisfied,
        then increment count($p^{enter}$) by 1
execute body of operation p
$p^{exit}$: increment count($p^{exit}$) by 1
```

That is, the update to the synchronization state within each gate consists simply of incrementing the count of events at that gate by 1. (The quantity **count**(g) is similar to, and in fact can be implemented by, the "eventcount" notion introduced in [Ree77]).

This means that the representation of a particular solution specification can consist simply of the entry condition on gate p^{enter}, for each operation p of the abstract type. Each entry condition on the synchronization state must take the form of a predicate on the counts of gates. The other (non-enter) gates in the solution specification are indicated implicitly by the appearance of quantities of the form **count**(g) within the entry conditions.

For example, consider an abstraction with one operation "op". Suppose that the synchronization constraint for this abstraction requires activations of op to be mutually exclusive, that is, at most one activation is allowed to be executing at a time. Then the solution specification for the abstraction can be expressed by stating the condition for gate op^{enter} to be

$$\mathbf{count}(op^{enter}) = \mathbf{count}(op^{exit}).$$

This is a shorthand way of saying that the abstract program for accessing an abstract data object via operation op is:

```
$op^{request}$: increment count($op^{request}$) by 1
$op^{enter}$: wait until count($op^{enter}$) = count($op^{exit}$),
        then increment count($op^{enter}$) by 1
execute body of operation op
$op^{exit}$: increment count($op^{exit}$) by 1
```

As a second example, consider an abstraction with two operations f and g. Assume that an activation of operation f is allowed to begin execution only if no activations of g have been requested and are waiting. Also, let an activation of g be able to enter only if exactly one activation of f is actively being executed. Then the solution specification for this abstraction consists of the two entry conditions:

For gate f^{enter}: count($g^{request}$) = count(g^{enter})

For gate g^{enter}: count(f^{enter}) - count(f^{exit}) = 1

In other words, the following are the abstract programs for activations of f and g:

Abstract program for f:

```
$f^{request}$: increment count($f^{request}$) by 1
$f^{enter}$: wait until count($g^{request}$) = count($g^{enter}$),
        then increment count($f^{enter}$) by 1
execute body of operation f
$f^{exit}$: increment count($f^{exit}$) by 1
```

Abstract program for g:

```
    $g^{request}$: increment count($g^{request}$) by 1
    $g^{enter}$: wait until count($f^{enter}$) - count($f^{exit}$) = 1,
            then increment count($g^{enter}$) by 1
    execute body of operation g
    $g^{exit}$: increment count($g^{exit}$) by 1
```

3.3 Additional features of the solution specification

As indicated in the introduction to this chapter, the basic structure presented thus far for the solution specification lacks sufficient power for expressing solutions to a wide class of synchronization problems. Two new features must be added to this basic structure in order to achieve the required expressive power. These additional features, which are the subject of this section, provide the ability to save and use previous state information, and the ability to use properties of parameters to operation activations. The first to be discussed is the use of previous state information.

In the previous section, the synchronization state was defined as some abstraction from the history of a data object containing sufficient information for the prediction of the future synchronization behavior of the object. Unfortunately, the counts of all event classes do not provide sufficient information. Sometimes it is necessary to know not only *how many* events of each class have taken place previously, but *in what order* certain of these events occurred.

There are a number of advantages to using integer-valued counts to represent the synchronization state. As illustrated in the previous section, it makes the abstract state update within each gate of the guardian particularly simple. As a result, the actual implementation of a solution specification in terms of a source language synchronization mechanism, which is the subject of Chapter 5, can be both simple and efficient. This efficiency is important in ensuring that the synchronization code itself does not significantly affect the concurrency of the computation. The use of counts is also important in terms of the algorithm presented in Chapter 4 for deriving a solution specification from a problem specification. For these reasons, it is desirable to remedy the lack of expressive power in a way that does not sacrifice the advantages of using counts of events as the basic form of the synchronization state.

The way to accomplish this is to add to the basic solution specification structure the ability to save the synchronization state at the time of an event. The state of the guardian then includes not only the current synchronization state, but also each previous state that has been saved. Conditions on enter gates can be expressed in terms of both the current synchronization state and any information saved from previous states. All the information that is lost by abstracting from the complete sequence of events within the history to the counts of event classes can be regained by using the state at the time of prior events as well as the current state. Basically the reason for this is that when it is necessary to know whether some particular event e_1 has preceded some other event e_2 in the preceding sub-history, this information can be obtained by comparing information in the states when e_1 and e_2 occurred with the current state and/or each other. In Chapter 4 it is explained how previous state information is derived to express properties for which the current state is insufficient.

A notational extension is needed to represent previous state information. Unless indicated otherwise, quantities appearing in a condition represent current state values. When a quantity is meant to represent a value in the state at some previous event, the notation "@ g" appended to the quantity is employed, where g is the name of some gate. This means that the quantity refers to the state saved just prior to the most recent event occurring at gate g. For example, the number of activations of p that had been requested at the point at which the most recent exit event for procedure q has occurred is denoted $[\text{count}(p^{request}) @ q^{exit}]$. Notice that since the state is saved just before the indicated event, a quantity such as $[\text{count}(q^{exit}) @ q^{exit}]$ does not include the q^{exit} event actually occurring at the point at which the state is saved.

As an example of a solution specification that uses previous state information, consider an abstraction with two operations u and v. Suppose that it is desired not only that activations of operation u be mutually exclusive, but that between any two successive activations of u, an enter event for operation v must occur. This can be expressed by the condition

$$\text{count}(u^{enter}) = \text{count}(u^{exit}) \wedge [\text{count}(v^{enter}) @ u^{exit}] < \text{count}(v^{enter})$$

for gate u^{enter}. The second conjunct of the condition says that $\text{count}(v^{enter})$ must increase between the exit event for the most recent activation of u and the time the next activation of u is allowed to enter. The corresponding abstract program for an activation of u is:

```
$u^{request}$: increment $\mathrm{count}(u^{request})$ by 1
$u^{enter}$: wait until $\mathrm{count}(u^{enter}) = \mathrm{count}(u^{exit}) \wedge$
                    $[\mathrm{count}(v^{enter}) \,@\, u^{exit}] < \mathrm{count}(v^{enter})$,
        then increment $\mathrm{count}(u^{enter})$ by 1
execute body of operation u
$u^{exit}$: save the guardian state, in particular the quantity $\mathrm{count}(v^{enter})$,
        and increment $\mathrm{count}(u^{exit})$ by 1
```

Each event at gate u^{enter} uses the value of $\mathrm{count}(v^{enter})$ saved at the most recent u^{exit} event in its entry condition.

As before, a solution specification is represented simply by the entry conditions that apply to all enter gates in the guardian. The state information that must be saved is not listed explicitly. Instead it is indicated implicitly by the appearance of quantities of the form $[\mathrm{count}(ec) \,@\, g]$, where ec is an event class and g is a gate, within entry conditions.

There is another aspect of information that is lost by abstracting from the history of an object to simply the count of events in each event class. The history is a sequence of events, each of which is described not only by its event class, which is to say the operation name and event type, but also by the vector of parameters passed to the operation. All information concerning the values of these parameters is lost when considering only the counts of event classes. For instance, it may be necessary for activations of an operation to be mutually exclusive only if an integer parameter of each activation is non-negative. Such a property can be expressed in the problem specification language of Chapter 2, but not in a solution specification with the structure presented thus far.

The solution is to "qualify" gates in the solution specification. A gate is *qualified* by the attachment of some predicate on the parameters of the associated procedure activation. Only if the parameters of an activation satisfy the predicate does the process making the activation pass through that gate. An unqualified gate, which applies to all activations of the given procedure, may be considered to be simply a special case of a qualified gate, with a qualifying predicate that is identically TRUE for all parameter values.

Some new notation is needed in order to refer to gates. An unqualified gate, as before, is indicated simply by the event class it is in, such as the p^{enter} gate. A qualified gate is denoted by appending the qualifying predicate to the procedure activation expression. The notation used is similar to that employed in set theory, with a vertical bar used to separate the predicate from the activation expression. Therefore, $[p(v) \mid C(v)]^{enter}$ denotes a gate in the p^{enter} event class that is qualified by the predicate C on the vector of parameters v to procedure p.

As an example, consider the following situation. Let an abstraction have one operation h, taking a single integer parameter x. Let all activations of h with non-negative parameter values be mutually exclusive. Then the solution specification contains the condition

$$\text{count}([h(x) \mid (x \geq 0)]^{enter}) = \text{count}([h(x) \mid (x \geq 0)]^{exit})$$

for gate $[h(x) \mid (x \geq 0)]^{enter}$. This means that the gates for both the h^{enter} and h^{exit} event classes are qualified with the predicate $(x \geq 0)$, and that any activation of h whose parameter does not satisfy this predicate need not pass through these gates. That is, the abstract program for an activation of h with parameter x is:

```
$h^{request}$: increment count($h^{request}$) by 1
$h^{enter}$: if $x \geq 0$ then
            wait until count($[h(x) \mid (x \geq 0)]^{enter}$) = count($[h(x) \mid (x \geq 0)]^{exit}$),
            and then increment count($[h(x) \mid (x \geq 0)]^{enter}$) by 1
execute body of operation h with parameter x
$h^{exit}$: if $x \geq 0$ then
            increment count($[h(x) \mid (x \geq 0)]^{exit}$) by 1
```

Since gate $h^{request}$ is not qualified, all activations must pass through it, regardless of their parameters.

Allowing only one qualifying predicate for an event class would be overly restrictive. It may be necessary to maintain counts of several different subsets of events in an event class, where each subset is distinguished by a different predicate on the operation parameters. These subsets may either be disjoint or overlap. Also, different entry conditions may be required for different subsets of the total set of activations of an operation, and again these subsets may be disjoint or overlap. It is therefore necessary to generalize the above structure by allowing more than one gate for each event class. Each gate in an event class is distinguished by a different qualifying predicate, and each gate of an enter class may have a different entry condition as well. When there is more than one gate for an event class, a process passes through exactly that set of gates whose qualifying predicates are satisfied by the parameters of the activation it is making. These gate passages are assumed to all occur in parallel. It is this simultaneous passage through a subset of the gates in an event class that implements the abstract notion of an event.

The implementation of each event class by a whole set of gates is a fundamental change in the structure of the solution specification. It is perhaps best understood by looking at the new abstract program for an activation of operation p with parameter vector v:

```
$p^{request}$: in parallel for all gates g in event class $p^{request}$,
        if v satisfies the qualifying predicate of g,
            then increment count(g) by 1
$p^{enter}$: in parallel for all gates g in event class $p^{enter}$,
        if v satisfies the qualifying predicate of g,
            then wait until the entry condition of g is satisfied,
            and then increment count(g) by 1
execute body of operation p
$p^{exit}$: in parallel for all gates g in event class $p^{exit}$,
        if v satisfies the qualifying predicate of g,
            then increment count($p^{exit}$) by 1
```

Since the events in an object history are totally ordered, each event must be an indivisible operation. This means that all gate passages making up an event occur, at least in a conceptual sense, in parallel and simultaneously. In particular, it means that a process may not pass through an enter gate unless it can pass through *all* of the enter gates for the given event class whose qualifying predicates are satisfied by its parameters. Only when all the entry conditions on these gates are satisfied may the enter event, in the form of the parallel passage through all these gates, take place.

As before, the processes that are blocked at a given enter event class are queued up in FIFO order. However, they need not be unblocked in this same order. Each process in the queue is waiting on one or more conditions, depending upon which qualifying predicates on gates apply to the activation. The process that proceeds first is the one closest to the front of the queue for which all entry conditions are satisfied. This may not be the one at the head of the queue, since that process may be waiting at a different set of gates than other processes further back in the queue.

It is important that the distinction between qualifying predicates and conditions on gates be clear. A qualifying predicate can be attached to a gate of any event class, and represents a constraint on the parameters of the associated procedure activation. If the predicate is satisfied for a particular activation, then the process making the activation passes through the gate, while if it is not satisfied, the process bypasses the gate. A condition, on the other hand, applies only to an enter gate. This condition is on synchronization states, the current state and perhaps also one or more previous states. If the condition is true, then the process may pass through the gate. If it is not, then the process becomes blocked, and must wait in a queue for the condition to be true.

As an example of a solution specification employing multiple gates, consider the abstraction discussed above with one operation h. Assume now, though, that h takes two integer parameters x and y. As before, activations of h for which parameter x is non-negative must be mutually exclusive. In addition, though, we want activations for which parameter y = 5 to be excluded whenever there is an activation currently executing for which y > x. The solution specification for this example consists of the following two conditions:

For gate $[h(x,y) \mid (x \geq 0)]^{enter}$:

$$\mathbf{count}([h(x,y) \mid (x \geq 0)]^{enter}) = \mathbf{count}([h(x,y) \mid (x \geq 0)]^{exit})$$

For gate $[h(x,y) \mid (y = 5)]^{enter}$:

$$\mathbf{count}([h(x,y) \mid (y > x)]^{enter}) = \mathbf{count}([h(x,y) \mid (y > x)]^{exit})$$

These conditions require two gates with entry conditions for event class h^{enter}, with qualifying predicates $(x \geq 0)$ and $(y = 5)$. There must be gates in both the h^{enter} and h^{exit} event classes to maintain counts for the qualifying predicates $(x \geq 0)$ and $(y > x)$. The abstract program for an activation of h with parameters x and y consists of:

```
$h^{request}$: increment $count(h^{request})$ by 1
$h^{enter}$: in parallel,
        if $(x \geq 0)$, wait until
            $count([h(x,y) \mid (x \geq 0)]^{enter}) = count([h(x,y) \mid (x \geq 0)]^{exit})$,
        and if $(y = 5)$, wait until
            $count([h(x,y) \mid (y > x)]^{enter}) = count([h(x,y) \mid (y > x)]^{exit})$,
    and then in parallel,
        if $(x \geq 0)$,
            increment $count([h(x,y) \mid (x \geq 0)]^{enter})$ by 1
        and if $(y > x)$,
            increment $count([h(x,y) \mid (y > x)]^{enter})$ by 1
execute body of operation h
$h^{exit}$: in parallel,
        if $(x \geq 0)$,
            increment $count([h(x,y) \mid (x \geq 0)]^{exit})$ by 1
        and if $(y > x)$,
            increment $count([h(x,y) \mid (y > x)]^{exit})$ by 1
```

That is, if both qualifying predicates $(x \geq 0)$ and $(y = 5)$ are satisfied for an activation, then both entry conditions must be simultaneously satisfied before its enter event. If only one qualifying predicate is satisfied, then only the entry condition corresponding to that qualified gate must be true. If neither predicate is satisfied, then the enter event can occur without delay. In any of these cases, $count([h(x,y) \mid (y > x)]^{enter})$ is incremented if and only if $(y > x)$.

3.4 Semantics of the solution specification

Thus far, the discussion in this chapter has relied on an informal, intuitive idea of the meaning of the solution specification. This section presents the formal definition of the semantics of solution specifications. As was the case for the problem specification language, whose formal definition was presented in Section 2.6, the semantics of the solution specification structure are defined by specifying which histories are valid with respect to any particular solution specification.

A *qualification* is a predicate on a vector of parameters. The domain of qualifications is denoted Q. One particular element of Q is the predicate that always returns TRUE. By considering this special predicate to be the qualification associated with what until now has been called an "unqualified" gate, we are able to consider all gates to be qualified. So, a *gate* is a pair <ec, q>, whose first component ec is an event class and whose second component q is a qualification.

A *state* is a function from gates to non-negative integers. A state maps each gate into the count of the number of passages through it. A *condition* is a predicate on a set of states. If the condition refers only to the current state, then the argument to the condition is a singleton set containing only the current state. When a condition refers to previous states as well, each of these states must also be in the set.

A *solution specification* consists of a set of gates, and a condition on each one of these gates. (It is simplest to take the view that a solution specification assigns each request and exit gate, and every enter gate not explicitly given an entry condition, the condition that is identically TRUE.) The set of gates in solution specification ss is given by the expression Gates(ss). For every gate g ∈ Gates(ss), the condition assigned to g in ss is given by Cond(ss, g). The set of previous states that the condition on gate g in solution specification ss refers to is given by PrevStates(ss, g).

A history is *valid* with respect to a solution specification if for each event in the history, every solution specification condition that applies to the event is satisfied at the point in the history at which the event occurs. (Actually, only enter events have non-trivial conditions, but for the sake of uniformity, it is easier to define the concept in terms of all events in the history.) To define this formally, it is necessary to have functions that map histories into states, i.e. into functions from gates into counts. The function CurSt maps an object history, the sequence of events associated with a given object, into the current state of that object. Recall that an object history is either the empty sequence [ ], or else is obtained by adding an event onto some other history. An event is represented by a four-tuple of the form <p, t, n, a>, where p is the operation name, t is the event type, n is the activation number, and a is the vector of arguments. The definition of CurSt is:

$$\mathrm{CurSt}([\ ]) = \lambda\,(ec, q).\ 0$$

$$\begin{aligned}\mathrm{CurSt}(\mathrm{add}(h, \langle p, t, n, a\rangle) = \lambda\,(ec, q).\ (&\text{if } \langle p, t\rangle = ec \ \wedge\ q(a)\\ &\text{then } (\mathrm{CurSt}(h)(ec, q) + 1)\\ &\text{else } \mathrm{CurSt}(h)(ec, q))\end{aligned}$$

The notation used here is taken from λ-calculus. The formula "λ (x, y). F" represents the function of arguments x and y whose body is given by F.

The function MosRecSt (Most Recent State) maps an object history and a gate into the state of an object at the time of the most recent event at that gate:

$$\mathrm{MosRecSt}([\ ], \langle ec, q\rangle) = \lambda\,(ec, q).\ 0$$

$$\begin{aligned}\mathrm{MosRecSt}(\mathrm{add}(h, \langle p, t, n, a\rangle), \langle ec, q\rangle) = \ &\text{if } \langle p, t\rangle = ec \ \wedge\ q(a)\\ &\text{then } \mathrm{CurSt}(h)\\ &\text{else } \mathrm{MosRecSt}(h, \langle ec, q\rangle)\end{aligned}$$

The current state after history h becomes the most recent state for any gate that applies to the event added onto h.

It is now possible to formally define the validity of a history h with respect to a solution specification ss. This is given by ValidSS(h, ss), where:

$$\mathrm{ValidSS}([\ ], ss) = \mathrm{TRUE}$$

$$\begin{aligned}\mathrm{ValidSS}(\mathrm{add}(h, \langle p, t, n, a\rangle), ss) = \mathrm{ValidSS}(h, ss)\ \wedge&\\ \forall\,(ec, q)\ (\langle ec, q\rangle \in \mathrm{Gates}(ss)\ \wedge\ ec = \langle p, t\rangle\ \wedge\ q(a)&\\ \supset\ \mathrm{SatSS}(h, ss, \langle ec, q\rangle)&\end{aligned}$$

SatSS(h, ss, <ec, q>), defined below, is a predicate that determines whether the state represented by history h satisfies the condition in solution specification ss for gate <ec, q>.

Therefore, the definition of ValidSS simply states that a history is valid with respect to a solution specification if it was valid before the last event occurred, and if the history satisfies the conditions for all gates that apply to the last event.

The predicate SatSS is easy to define. A history *satisfies* a condition simply if the current state plus the relevant most recent states of the history satisfy the condition. Recall that the condition on gate g in solution specification ss is given by Cond(ss, g), and that this condition is simply a predicate on a set of states. Formally, then,

$$\mathrm{SatSS}(h, ss, g) \;=\; C(\mathrm{States}),$$

$$\text{where } C = \mathrm{Cond}(ss, g)$$

$$\text{and } \mathrm{States} \;=\; \{\mathrm{CurSt}(h)\} \;\cup\; \{\mathrm{MosRecSt}(h, g') \mid g' \in \mathrm{PrevStates}(ss, g)\}$$

The subject of the next chapter is the method for deriving an equivalent solution specification from a problem specification. Section 4.6 justifies the method presented. This justification relies on both the formal definition of the problem specification language given in Section 2.6, and the formal definition of the solution specification in this section.

Chapter 4

Derivation of the Solution Specification

4.1 Introduction

The subject of this chapter is the algorithm for analyzing a problem specification and deriving from it an equivalent solution specification. There are two aspects to the construction of a solution specification. Identifying the gates required in the solution specification is relatively straightforward. This simply involves identifying the event classes appearing in the problem specification. For qualified gates to be identified correctly, however, this must be done *after* all argument constraints have been incorporated into the ordering clauses of the specification, as explained in Section 4.5.

Constructing appropriate conditions to attach to the gates associated with **enter** event classes is the formidable task. The algorithm for constructing these entry conditions is the subject of this chapter. As explained in Chapter 3, the set of conditions on all **enter** gates is sufficient to represent the complete solution specification. The other gates in the solution specification and the saving of previous state information are indicated implicitly by the quantities appearing in the entry conditions.

In constructing a condition for an **enter** gate, the basic strategy employed is to determine, in terms of the synchronization state, what distinguishes points in a computation at which an event at that gate should or should not occur. "Should occur" here can be interpreted formally as satisfying the predicate Sat, which was defined in Chapter 2, relative to the given specification. In making this determination, it is necessary to consider all relevant subsequences of histories, specifically those subsequences containing the events mentioned explicitly in the specification. Each of these subsequences, or "orderings", can be classified as either valid or invalid with respect to the specification. At each point in an ordering at which an event occurs at the gate in question, it is possible to characterize the synchronization state. These individual characterizations can then be combined appropriately, based on the validity of the orderings, to form an overall condition for the gate.

The paragraph above summarizes the main phase of the derivation algorithm. The result of this phase, which is presented fully in Section 4.2, is the derivation for each gate of a "preliminary condition". For cases where the correct condition for a gate can be expressed solely in terms of the current state, the preliminary condition is correct. When this is not so, the preliminary condition can be refined by iterating over another phase of the algorithm. This phase, which is presented in Section 4.3, uses information saved at previous states in the orderings as well as the current state. Section 4.4 contains an example of applying the algorithm of Sections 4.2 and 4.3. The one other aspect of the algorithm is some initial processing designed to make the specification suitable for analysis. Section 4.5 describes this processing, in which argument constraints are incorporated into the specification so that the transformed specification consists entirely of ordering clauses. The algorithm is summarized in its entirety in Section 4.6, and there a justification is presented for why it works. The last section of this chapter, Section 4.7, discusses the class of specifications for which the algorithm fails.

An important feature of the approach to be presented is a property that I call *extensibility*. This means that the algorithm can be applied to each conjunct in a problem specification individually. If the specification s is of the form

$$s_1 \wedge s_2 \wedge \ldots \wedge s_m,$$

then for each conjunct s_i of the specification, the algorithm derives one or more conditions for gates in the solution specification. For each gate, the condition required for the entire specification s is simply the conjunction of the conditions obtained separately from the conjuncts s_i. This property can be proved in terms of the formal semantic definitions of the problem specification language and the solution specification. Informally, it is true because each conjunct in a specification represents a separate constraint that must be met by any valid history, so that the overall specification represents a set of constraints, all of which must be met. If each constraint is implemented by a different set of solution specification conditions, then the joint overall constraint must be implemented by conjoining all these conditions. This is because an event may validly occur only if it does not violate any of the individual constraints. For this reason, the analysis of specification s can take place on each relatively simple conjunct separately, rather than on the entire, more complex specification. With regard to any reference in this chapter to specification s, the reader should understand that s can represent a single conjunct that is being analyzed individually.

4.2 The derivation algorithm

This section describes the essence of the derivation algorithm. It is assumed that the problem specification consists exclusively of ordering information, in that all clauses, as defined in Section 2.5, are ordering clauses of the form ($e_1 \Rightarrow e_2$), where events e_1 and e_2 refer to procedure activations for which arguments are not listed. That is to say, there are no argument constraint clauses, nor are arguments explicitly given for any procedure activations. The conditions derived for the solution specification in this phase of the algorithm refer only to the current synchronization state, and not to any previous states. When any of the preliminary conditions derived by this phase is inadequate, then previous state information must be used in order to refine it. The method for doing so is presented in the section following this one.

The algorithm is presented here on a step-by-step basis. Each step first is described as it works on a general specification s, and then illustrated on a particular specification. The specific example used for illustration purposes is example 4 from Section 2.7, which will be denoted here as specification s_1:

$$p_i^{request} \Rightarrow q_j^{enter} \supset p_i^{enter} \Rightarrow q_j^{enter}.$$

As discussed in Chapter 2, the effect of this specification is to give executions of procedure p priority over those of procedure q.

Given a problem specification s, the first step in deriving the equivalent solution specification is to identify Evexp(s), the set of event expressions appearing in s. Informally, this set can be constructed simply by noting which event expressions are contained in the specification. The recursive definition of Evexp(s), which was presented in Section 2.6 and is repeated in Figure 4.1 below, can be used to formally construct Evexp(s) for any specification. For the example specification,

$$\text{Evexp}(s_1) = \{p_i^{\text{request}}, p_i^{\text{enter}}, q_j^{\text{enter}}\}.$$

Once Evexp(s) has been constructed, the next step is to construct the set of possible time orderings among the events represented by these expressions. Suppose a history contains events that correspond to the event expressions in the specification. Formally, using the definitions of Section 2.6, this means that there is some interpretation mapping the event expressions in Evexp(s) into a subset of the events in the history. Then whether or not the history satisfies the specification under this interpretation depends upon the order among exactly these events. To analyze all possible histories that involve events corresponding to the expressions in the specification, it is sufficient to analyze all possible subsequences of these events. A subsequence of events in a history is called a *sub-history*.

Figure 4.1. Definition of Evexp(s)

$$\text{Evexp}(e_1 \Rightarrow e_2) = \{ e_1 , e_2 \}$$
$$\text{Evexp}(\text{exp}_1 \text{ rel } \text{exp}_2) = \{ \}, \quad \text{for rel} \in \text{Rel}$$
$$\text{Evexp}(\neg s) = \text{Evexp}(s)$$
$$\text{Evexp}(s_1 \text{ op } s_2) = \text{Evexp}(s_1) \cup \text{Evexp}(s_2), \quad \text{for op} \in \text{Op}$$
$$\text{Evexp}(\exists x (s)) = \text{Evexp}(s)$$
$$\text{Evexp}(\forall x (s)) = \text{Evexp}(s)$$

Since each relevant event is represented by an event expression appearing in specification s, the sub-histories of interest correspond to the possible sequences of the expressions in Evexp(s). Each sequence of event expressions that represents a possible sub-history is called an *ordering*. Every history containing events represented by the event expressions of Evexp(s) corresponds to exactly one of the orderings.

If the size of Evexp(s) is n, then there are n! permutations of these n events, but not all of the corresponding sequences are necessarily possible time orderings. To be a possible ordering, a sequence must obey the basic constraint

$$u_m^{request} \Rightarrow u_m^{enter} \Rightarrow u_m^{exit},$$

for every procedure activation u_m. For example, consider a case where

$$Evexp(s) = \{x_a^{request}, x_a^{enter}, x_a^{exit}, y_b^{request}, y_b^{enter}, y_b^{exit}\}.$$

While there are 720 permutations of these six events, only 20 sequences represent possible time orderings. An additional constraint that must be met by any ordering is that

$$(m < n) \supset (u_m^{request} \Rightarrow u_n^{request}),$$

since the numbering of procedure activations is based on the order of the respective request events. Thus, for a specification in which $x_i^{request}$ and $x_{i+1}^{request}$ both appear, $x_i^{request}$ must precede $x_{i+1}^{request}$ in every ordering. These constraints are exactly the ones embodied in the predicate Possible defined in Section 2.6. Ruling out all orderings that are impossible corresponds to restricting attention to object histories that are possible according to that definition.

Formally, the construction of the possible orderings among the elements of Evexp(s) can be carried out in two stages. The first stage consists of generating all permutations of the elements of Evexp(s). Then every permutation that violates one of these basic constraints is eliminated.

For the example specification s_1, Evexp(s_1) contains three events, as already noted. Although there are six permutations of these three events, only three are possible time orderings, since the other three violate the constraint that $p_i^{request} \Longrightarrow p_i^{enter}$. These three possible orderings are:

(1) $p_i^{request} \Longrightarrow p_i^{enter} \Longrightarrow q_j^{enter}$

(2) $p_i^{request} \Longrightarrow q_j^{enter} \Longrightarrow p_i^{enter}$

(3) $q_j^{enter} \Longrightarrow p_i^{request} \Longrightarrow p_i^{enter}$

That is, in any possible history in which there are events corresponding to the three event expressions in Evexp(s_1), these events must occur in exactly one of these three orders.

Once the possible orderings of the events associated with specification s have been constructed, it is necessary to separate them into two classes. Those that satisfy the specification s are termed *valid* orderings, while the rest are *invalid*. Validity of an ordering with respect to a specification s can be determined by simply evaluating the formula s. In this evaluation, either TRUE or FALSE is substituted for each expression of the form ($e_1 \Longrightarrow e_2$), depending upon whether or not event e_1 precedes event e_2 in the given ordering. Since it is assumed that by this point the specification consists entirely of ordering information, the result of this evaluation must equal either TRUE or FALSE.

The ordering is valid when the formula evaluates to TRUE, and invalid when it is FALSE. In terms of the formal semantics of the problem specification language presented in Chapter 2, this corresponds to evaluating the predicate Sat for an otherwise valid history that contains the given ordering as a sub-history under an arbitrary interpretation.

For the example, substitution of ordering (1) into specification s_1 yields the formula

$$\text{TRUE} \supset \text{TRUE},$$

which evaluates to TRUE. Substituting ordering (3) into s_1 results in the formula

$$\text{FALSE} \supset \text{FALSE},$$

which also evaluates to TRUE. Orderings (1) and (3) are therefore both valid with respect to s_1. Substituting ordering (2) into s_1, however, yields

$$\text{TRUE} \supset \text{FALSE},$$

which is FALSE, so ordering (2) is invalid.

In describing the next step of the algorithm, some definitions are needed. A *prefix* of a sequence is simply any initial subsequence. A special case is the empty sequence, which is a prefix of every sequence. Any two sequences have a unique *longest matching prefix* which they share. Given two different orderings of n events, there is a unique k, where $0 < k < n$, such that each of the first $(k - 1)$ events in the two orderings are identical, and the k-th events differ. The shared prefix of length $(k - 1)$ is the longest matching prefix of the two orderings.

It is necessary to compare each invalid ordering with all of the valid orderings in turn. In each case, there will be a longest matching prefix that the two orderings share, which may be the empty sequence. Of all these longest matching prefixes, we choose the one with the greatest length. If this prefix is of length (k - 1), then the k-th event (more precisely, the k-th event expression) in the invalid ordering is the *offending* event of that ordering. The offending event is the one at which the invalid ordering first "goes wrong" in the sense of violating the specification. That is, it is at this point in the history that the Sat predicate is first violated for the specification. Assuming that the offending event is in an enter event class, a condition must be attached to the gate for that event class in the solution specification, so that the SatSS predicate for the solution specification is also violated at this point.

If the offending event in the invalid ordering is not an enter event, then the specification is illegal, in that it does not agree with the basic guardian model being employed here. According to the model, only enter events can be conditional and so be delayed from immediately taking place. If a specification implies that some request or exit event should be delayed, then it represents a property that is incompatible with this model. Such a specification cannot be analyzed by the method presented here. (These cases are discussed in section 4.7.)

Returning to the example specification s_1, orderings (1) and (3) have already been shown to be valid, and ordering (2) to be invalid. For orderings (1) and (2), the longest matching prefix consists of the sequence of length one whose only element is $p_i^{request}$; for orderings (2) and (3), the longest matching prefix is the empty sequence. The longest prefix of ordering (2) that matches some valid ordering is therefore the one-element sequence $[p_i^{request}]$. The offending event in (2) is the event immediately following this prefix, namely q_j^{enter}. Thus a condition is required on the gate for the q^{enter} event class to prevent this invalid ordering.

In the general case, a condition must be derived for each event class that contains an offending event in one or more invalid orderings. When this condition is placed on the gate for that event class in the solution specification, it must prevent any sub-history corresponding to one of these invalid orderings, but allow any of the valid orderings as sub-histories. The derivation of the condition requires the state, i.e. the synchronization state of the object, to be characterized for each invalid ordering at the point at which the offending event occurs, so long as the offending event belongs to the given event class. The method for characterizing the state is explained below. A disjunction of these state characterizations is formed, to be denoted here as D_i. D_i represents a general state characterization of when the occurrence of an event in the given event class would fail to satisfy the specification. Similarly, the state must be characterized for each valid ordering at the point at which an event in the class occurs. The disjunction of these characterizations is denoted D_v, which is a general characterization of when the occurrence of such an event would satisfy the specification.

The expression given by the formula $(D_v \wedge (\neg D_i))$ represents a preliminary possibility for the condition required in the solution specification. The term $(\neg D_i)$ guarantees that the expression is strong enough to exclude every invalid ordering. Conjoining the term D_v aids in the simplification of the formula. Since any conditions that are trivially true in all orderings of interest appear both in D_V and in D_I, such conditions cancel out in the conjunction of D_V with the negation of D_I. These conditions may arise from the fact, for instance, that at the point just before an event in the p^{enter} class occurs, it is always true that $count(p^{request}) > count(p^{enter})$, since there is at least one activation (the one under consideration) for which the request event, but not the enter event, has occurred. Thus, this clause is a component of every state characterization, whether the ordering is valid or invalid. The conjunct D_V guarantees that the negation of this clause is eliminated from the condition.

The preliminary condition given by $(D_V \wedge (\neg D_I))$ is known to be at least as strong as the condition required, since the term $(\neg D_I)$ excludes all invalid orderings, i.e. all histories with sub-histories corresponding to an invalid ordering. The condition must be tested against all the valid orderings, however, to check that it is weak enough to allow all of them as sub-histories. This checking is accomplished by determining that the condition is satisfied at the point at which the appropriate event occurs in each valid ordering. If the condition is satisfied at all these points, then the condition is correct, and the task is completed. If this is not so, then the condition is too strong, in that it rules out some orderings that are valid according to the specification. When this happens, steps must be taken to refine the condition by weakening it appropriately. This weakening process will be described in the next section.

In characterizing the synchronization state of the object at a point in an ordering, the ordering must be considered to represent a sub-history that is embedded within some possible history. Except for what can be deduced from the ordering itself, nothing can be assumed about the history or about the interpretation by which the event expressions in the ordering are mapped into the events in the history. There may be an arbitrary number of events in the history preceding the sub-history, and between any two events in the sub-history. It is known, however, that the history is possible. Also, the history can be assumed to be compatible with the solution specification structure, since if it is not, then the algorithm cannot succeed in any case (see Section 4.7).

The characterization of the state therefore relies entirely on the other events in the sub-history represented by event expressions in the ordering. Since the characterization involves actual events in a history, rather than the event expressions in an ordering, each event expression conceptually is replaced by a real event, so that every variable within an expression is replaced by an actual value. Since the interpretation for making these replacements is arbitrary, however, nothing can be assumed about the values. All that is known is that for any given history and interpretation, there is some particular value for each variable. For this reason, in the state characterization each variable is existentially quantified. That is, every state characterization formula is of the form

$$\exists (i_1, \ldots, i_m) (S),$$

where $\{i_1, \ldots, i_m\}$ is the set of variables appearing free in formula S.

The body S of the state characterization formula consists of placing bounds on the counts of event classes, based on which of these events occur before and after the point at which the characterization is being made. It is assumed that the characterization is made *just before* the enter event of interest occurs, so that this event itself has not yet taken place, but every preceding event has occurred. The characterization contains a clause corresponding to each event in the ordering, that is, to each element of Evexp(s). For each $e \in$ Evexp(s), the count of the event class containing e is given either a lower bound if e occurs prior to this point in the ordering, or an upper bound if e occurs subsequent to this point. The bound in either case is the invocation number of e.

For example, let e be the event expression x_m^{enter}. If event x_m^{enter} occurs prior to the enter event in the ordering being considered, then the state characterization contains the conjunct

$$\mathrm{count}(x^{enter}) \geq m.$$

The reasoning is that if x_m^{enter} has already occurred, then so have each of x_k^{enter} for $(1 \leq k < m)$, so that count(x^{enter}) is at least as great as m. The count may be greater than m, as other events in the x^{enter} class may have taken place in between event x_m^{enter} and the current point, but it is not less than m. On the other hand, if x_m^{enter} occurs after the point at which the characterization is made, then the clause becomes instead

$$\mathrm{count}(x^{enter}) < m.$$

If x_m^{enter} has not yet occurred, then neither has x_k^{enter} for any $k > m$, so that count(x^{enter}) must be less than m. Again, other x^{enter} events may occur in between the point of the characterization and x_m^{enter}, so that the count may be less than (m - 1), but it is certainly less than m.

This method of state characterization relies on a first-come-first-served scheduling discipline at each gate. That is, it assumes that any history occurring prior to event x_m^{enter} contains exactly

$$[x_1^{enter}, x_2^{enter}, \dots, x_{m-1}^{enter}]$$

as the subsequence of events occurring at the x^{enter} gate. This scheduling policy is built into the structure of the solution specification, and so it may be assumed that if a correct solution specification can be derived for a specification, then it must fit this structure. There are specifications with which this first-come-first-served scheduling policy is not compatible, and the derivation algorithm fails to derive a solution specification in such cases. This point is discussed more fully in Section 4.7.

Since every state characterization formula is of the form

$$\exists (i_1, \dots, i_m) (S),$$

the construction and manipulation of the formulas D_v and D_i must make use of logical properties of existentially quantified expressions. Because of the negation of D_i in the preliminary condition, universally quantified expressions must also be manipulated. A summary of the important logical properties used for simplifying these formulas appears in Figure 4.2. Properties (E1) through (E6) are equivalences applicable to existentially quantified expressions, and properties (A1) through (A6) are their dual forms for universally quantified expressions. (Q1) and (Q2) apply to formulas involving both types of quantifiers, and (D1) and (D2) are the distributive laws for $\wedge$ and $\vee$.

Figure 4.2. Logical properties of quantified expressions

(E1)	$\exists\, i\, (S_1) \lor \exists\, i\, (S_2)$	$\leftrightarrow$	$\exists\, i\, (S_1 \lor S_2)$
(E2)	$\exists\, (i,j)\, (A(i) \land B(j))$	$\leftrightarrow$	$\exists\, i\, (A(i)) \land \exists\, j\, (B(j))$
(E3)	$\exists\, (i,j)\, (A(i))$	$\leftrightarrow$	$\exists\, i\, (A(i))$
(E4)	$\neg\, (\exists\, i\, (S))$	$\leftrightarrow$	$\forall\, i\, (\neg\, S)$
(E5)	$\exists\, i\, (x \geq i \land y < i)$	$\leftrightarrow$	$(x > y)$
(E6)	$\exists\, i\, (x < i)$	$\leftrightarrow$	TRUE
(A1)	$\forall\, i\, (S_1) \land \forall\, i\, (S_2)$	$\leftrightarrow$	$\forall\, i\, (S_1 \land S_2)$
(A2)	$\forall\, (i,j)\, (A(i) \lor B(j))$	$\leftrightarrow$	$\forall\, i\, (A(i)) \lor \forall\, j\, (B(j))$
(A3)	$\forall\, (i,j)\, (A(i))$	$\leftrightarrow$	$\forall\, i\, (A(i))$
(A4)	$\neg\, (\forall\, i\, (S))$	$\leftrightarrow$	$\exists\, i\, (\neg\, S)$
(A5)	$\forall\, i\, (x < i \lor y \geq i)$	$\leftrightarrow$	$(x \leq y)$
(A6)	$\forall\, i\, (x \geq i)$	$\leftrightarrow$	FALSE
(Q1)	$\exists\, i\, (S) \land \forall\, i\, (\neg\, S)$	$\leftrightarrow$	FALSE
(Q2)	$\exists\, i\, (P \land S) \land \forall\, i\, (Q \lor \neg\, S)$	$\leftrightarrow$	$\exists\, i\, (P \land Q \land S)$
(D1)	$((x \land y) \lor z)$	$\leftrightarrow$	$((x \lor z) \land (y \lor z))$
(D2)	$((x \lor y) \land z)$	$\leftrightarrow$	$((x \land z) \lor (y \land z))$

Let us return to the example for an illustration of the above discussion. Recall that the offending event in the invalid ordering is q_j^{enter}, and so a condition must be derived for the q^{enter} gate. In ordering (1), the event q_j^{enter} is preceded by events $p_i^{request}$ and p_i^{enter}, and has no events following it. Therefore, the state characterization c_1 is:

$$\exists\, (i,j)\, (count(p^{request}) \geq i \land count(p^{enter}) \geq i \land count(q^{enter}) < j),$$

where the first two terms in the body are obtained from the events preceding q_j^{enter}, and the last term from the fact that q_j^{enter} itself has not yet occurred at the point at which the characterization is made. In ordering (2), the event q_j^{enter} is preceded by $p_i^{request}$ and followed by p_i^{enter}, so the state characterization c_2 is:

$$\exists\, (i,j)\, (count(p^{request}) \geq i \land count(p^{enter}) < i \land count(q^{enter}) < j).$$

In ordering (3), q_j^{enter} precedes both $p_i^{request}$ and p_i^{enter}, and the state characterization c_3 is:

$$\exists\,(i,j)\,(\text{count}(p^{\text{request}}) < i \;\wedge\; \text{count}(p^{\text{enter}}) < i \;\wedge\; \text{count}(q^{\text{enter}}) < j).$$

These individual characterizations can now be combined to form the terms D_v and D_i. The disjunction for the valid orderings D_v is equal to $(c_1 \vee c_3)$, or

$$\begin{gathered}
\exists\,(i,j)\,(\text{count}(q^{\text{enter}}) < j \;\wedge \\
((\text{count}(p^{\text{request}}) \geq i \;\wedge\; \text{count}(p^{\text{enter}}) \geq i) \;\vee \\
(\text{count}(p^{\text{request}}) < i \;\wedge\; \text{count}(p^{\text{enter}}) < i))).
\end{gathered}$$

The disjunction for the invalid orderings D_i is simply c_2, so that $(\neg\, D_i)$ becomes

$$\forall\,(i,j)\,(\text{count}(p^{\text{request}}) < i \;\vee\; \text{count}(p^{\text{enter}}) \geq i \;\vee\; \text{count}(q^{\text{enter}}) \geq j).$$

The formula for the preliminary condition is therefore given by $D_v \wedge (\neg\, D_i)$, or

$$\begin{gathered}
\exists\,(i,j)\,(\text{count}(q^{\text{enter}}) < j \;\wedge \\
((\text{count}(p^{\text{request}}) \geq i \;\wedge\; \text{count}(p^{\text{enter}}) \geq i) \;\vee \\
(\text{count}(p^{\text{request}}) < i \;\wedge\; \text{count}(p^{\text{enter}}) < i))) \;\;\wedge \\
\forall\,(i,j)\,(\text{count}(p^{\text{request}}) < i \;\vee\; \text{count}(p^{\text{enter}}) \geq i \;\vee\; \text{count}(q^{\text{enter}}) \geq j).
\end{gathered}$$

This formula can be simplified. Since the terms involving i and j are independent in both of the quantified expressions, they can be separated, using logical properties (E2) and (A2) from Figure 4.2. This yields the formula:

$$\begin{gathered}
\exists\, i\,((\text{count}(p^{\text{request}}) \geq i \;\wedge\; \text{count}(p^{\text{enter}}) \geq i) \;\vee \\
(\text{count}(p^{\text{request}}) < i \;\wedge\; \text{count}(p^{\text{enter}}) < i)) \;\;\wedge \\
\exists\, j\,(\text{count}(q^{\text{enter}}) < j) \;\wedge \\
(\forall\, j\,(\text{count}(q^{\text{enter}}) \geq j) \;\vee \\
\forall\, i\,(\text{count}(p^{\text{request}}) < i \;\vee\; \text{count}(p^{\text{enter}}) \geq i)).
\end{gathered}$$

By distributivity property (D2), this is equivalent to

$$\begin{gathered}
(\exists\, i\, ((\mathrm{count}(p^{request}) \geq i \;\wedge\; \mathrm{count}(p^{enter}) \geq i) \;\vee \\
(\mathrm{count}(p^{request}) < i \;\wedge\; \mathrm{count}(p^{enter}) < i)) \;\;\wedge \\
\exists\, j\, (\mathrm{count}(q^{enter}) < j) \;\wedge \\
\forall\, j\, (\mathrm{count}(q^{enter}) \geq j)) \\
\vee \\
(\exists\, i\, ((\mathrm{count}(p^{request}) \geq i \;\wedge\; \mathrm{count}(p^{enter}) \geq i) \;\vee \\
(\mathrm{count}(p^{request}) < i \;\wedge\; \mathrm{count}(p^{enter}) < i)) \;\;\wedge \\
\exists\, j\, (\mathrm{count}(q^{enter}) < j) \;\wedge \\
\forall\, i\, (\mathrm{count}(p^{request}) < i \;\vee\; \mathrm{count}(p^{enter}) \geq i)).
\end{gathered}$$

The first disjunct is simply FALSE, since it contains the conjunction of

$$\exists\, j\, (\mathrm{count}(q^{enter}) < j)$$

and

$$\forall\, j\, (\mathrm{count}(q^{enter}) \geq j).$$

This means that the formula reduces to the second disjunct,

$$\begin{gathered}
\exists\, i\, ((\mathrm{count}(p^{request}) \geq i \;\wedge\; \mathrm{count}(p^{enter}) \geq i) \;\vee \\
(\mathrm{count}(p^{request}) < i \;\wedge\; \mathrm{count}(p^{enter}) < i)) \;\;\wedge \\
\exists\, j\, (\mathrm{count}(q^{enter}) < j) \;\wedge \\
\forall\, i\, (\mathrm{count}(p^{request}) < i \;\vee\; \mathrm{count}(p^{enter}) \geq i).
\end{gathered}$$

Each of the first two conjuncts simplifies to TRUE, so the entire formula reduces to

$$\forall\, i\, (\mathrm{count}(p^{request}) < i \;\vee\; \mathrm{count}(p^{enter}) \geq i),$$

which is equivalent to

$$\mathrm{count}(p^{request}) \leq \mathrm{count}(p^{enter})$$

by property (A5). Using the *a priori* fact that **$\mathrm{count}(p^{request}) \geq \mathrm{count}(p^{enter})$, the preliminary** condition can be simplified finally to:

$$\mathbf{count}(p^{request}) = \mathbf{count}(p^{enter}).$$

To determine whether the preliminary condition is indeed correct and not overly strong, it is necessary to test it at the appropriate point in each of the valid orderings. The valid orderings are (1) and (3). At the point of event q_j^{enter} in each of these orderings, the condition

$$\mathbf{count}(p^{request}) = \mathbf{count}(p^{enter})$$

is satisfied, showing that it is weak enough to permit both valid orderings. Because of the conjunct $(\neg D_i)$ in the condition, it is guaranteed to be strong enough to prevent the invalid ordering. Therefore, it is exactly the condition required for gate q^{enter}, and a correct solution specification has been constructed.

4.3 Use of previous states

In the example presented in the last section, the current state alone was sufficient to derive the condition required in the solution specification. The purpose of this section is to explain the method employed when this is not the case, and one or more previous states must be used as well. Information from previous states is used to refine a preliminary condition that is too strong so that one or more valid orderings do not satisfy it.

An overly strong preliminary condition is weakened by disjoining one or more terms to it. The new condition that results is strictly weaker than the preliminary condition, since it is the disjunction of the preliminary condition and other terms. All valid orderings that satisfy the preliminary condition therefore automatically satisfy the new condition. The purpose of the weakening terms is to include the remaining valid orderings as well. For this reason the analysis for constructing a weakening term can disregard the valid orderings satisfying the preliminary condition. Only the remaining valid orderings not permitted by the preliminary condition need be considered, along with all invalid orderings for which the event in question is the offending event.

Each weakening term shares the property with the preliminary condition that it is at least strong enough to exclude every invalid ordering. Therefore, all that need be checked for each weakening term is which valid orderings that have thus far been excluded are permitted by the given term. The method terminates when the condition is weakened so that all valid orderings are allowed, or else when no further weakening terms can be constructed.

In deriving a weakening term, it is necessary first to find some event that precedes the enter event in question in each ordering being considered, i.e. all of the valid orderings not satisfying the preliminary condition plus all of the invalid orderings in which the enter event is the offending event. This event may be in any event class, and is not limited to enter events. Once such an event is found, the weakening term is constructed in much the same way as the preliminary condition, but using state characterizations at this previous event. The state is characterized at the point of the preceding event in each of these orderings (but not in any of the other valid orderings). Notice that each of these characterizations, rather than involving ordinary counts of event classes, concerns quantities of the form [count(ec) @ g], i.e. counts of event classes saved at the event at gate g.

At this point the characterizations from the valid orderings are disjoined to form a new expression D_v', and the characterizations from the invalid orderings are disjoined to form D_i'. The formula $(D_v' \wedge (\neg D_i'))$ is constructed and used as a weakening term by disjoining it to the preliminary condition to form a new condition. This new condition is tested to determine whether the valid orderings excluded by the preliminary condition are allowed as a result of the weakening term. If all these orderings are permitted by the weakening term, then the new condition constitutes the solution specification condition.

If there are still some valid orderings not allowed, then the process is repeated on the valid orderings still excluded. Here, however, each characterization refers to both the current state and the previous state. That is, each characterization involves both current counts and counts in the previous state. The weakening term $(D_v' \wedge (\neg D_i'))$ is formed in the same way. This term is again tested on the excluded valid orderings, and disjoined to the condition if it is satisfied by any of the excluded orderings.

For example, consider the specification

$$(p_i^{exit} \Rightarrow p_j^{enter}) \supset$$
$$\exists\, k\, (p_i^{exit} \Rightarrow q_k^{enter} \Rightarrow p_j^{enter}).$$

When the preliminary condition is formed for gate p^{enter}, it is found not to satisfy the valid ordering

$$(1)\ p_i^{exit} \Rightarrow q_k^{enter} \Rightarrow p_j^{enter}.$$

A weakening term must therefore be constructed for this ordering. The two invalid orderings are

$$(2)\ p_i^{exit} \Rightarrow p_j^{enter} \Rightarrow q_k^{enter}$$

$$(3)\ q_k^{enter} \Rightarrow p_i^{exit} \Rightarrow p_j^{enter}$$

in both of which the offending event is p_j^{enter}. The one event that precedes p_j^{enter} in each of these three orderings is p_i^{exit}. The state characterization at this event in each of the three orderings is:

$$c_1\text{:}\ \exists\,(i, j, k)\ ([count(p^{exit}) \bullet p^{exit}] < i\ \wedge\ [count(q^{enter}) \bullet p^{exit}] < k$$
$$\wedge\ [count(p^{enter}) \bullet p^{exit}] < j)$$

$$c_2\text{:}\ \exists\,(i, j, k)\ ([count(p^{exit}) \bullet p^{exit}] < i\ \wedge\ [count(q^{enter}) \bullet p^{exit}] < k$$
$$\wedge\ [count(p^{enter}) \bullet p^{exit}] < j)$$

$$c_3\text{:}\ \exists\,(i, j, k)\ ([count(p^{exit}) \bullet p^{exit}] < i\ \wedge\ [count(q^{enter}) \bullet p^{exit}] \geq k$$
$$\wedge\ [count(p^{enter}) \bullet p^{exit}] < j)$$

However, the formula $(D_v' \wedge (\neg\ D_i'))$ given by

$$c_1 \wedge (\neg\ (c_2 \vee c_3))$$

is equivalent to FALSE, which is obviously useless as a weakening term.

Therefore, it is necessary to form new characterizations of both the current and previous states. These are given by:

$$c_1'\text{:}\ \exists\,(i, j, k)\ (count(p^{exit}) \geq i\ \wedge count(q^{enter}) \geq k\ \wedge\ count(p^{enter}) < j$$
$$\wedge\ [count(p^{exit}) \bullet p^{exit}] < i\ \wedge\ [count(q^{enter}) \bullet p^{exit}] < k$$
$$\wedge\ [count(p^{enter}) \bullet p^{exit}] < j)$$

$$c_2' \text{:}\ \exists\,(i, j, k)\,(\text{count}(p^{exit}) \geq i \wedge \text{count}(q^{enter}) < k \wedge \text{count}(p^{enter}) < j$$
$$\wedge\ [\text{count}(p^{exit}) \bullet p^{exit}] < i \wedge [\text{count}(q^{enter}) \bullet p^{exit}] < k$$
$$\wedge\ [\text{count}(p^{enter}) \bullet p^{exit}] < j)$$

$$c_3' \text{:}\ \exists\,(i, j, k)\,(\text{count}(p^{exit}) \geq i \wedge \text{count}(q^{enter}) \geq k \wedge \text{count}(p^{enter}) < j$$
$$\wedge\ [\text{count}(p^{exit}) \bullet p^{exit}] < i \wedge [\text{count}(q^{enter}) \bullet p^{exit}] \geq k$$
$$\wedge\ [\text{count}(p^{enter}) \bullet p^{exit}] < j)$$

The new weakening term $(D_v' \wedge (\neg D_i'))$ is equal to

$$c_1' \wedge (\neg (c_2' \vee c_3')),$$

which simplifies to

$$\text{count}(q^{enter}) > [\text{count}(q^{enter}) \bullet p^{exit}],$$

which does satisfy ordering (1). As a result, the solution specification condition is given by disjoining this term to the preliminary condition.

If neither of the weakening terms obtained as a result of a given previous state is sufficient to include all of the remaining orderings, then another previous event must be found and the entire weakening process is repeated using the state at that event. Since this may involve using the next-to-most recent, etc. event at a particular gate, a notational extension is needed to refer to such quantities, such as [count(ec) •• g], etc.

The idea behind the method is to find some property that distinguishes the valid orderings from the invalid ones. Unless the specification is one that violates the underlying model, it is always possible to find such a property. A valid ordering that cannot be distinguished on the basis of the preliminary condition must differ from an invalid ordering by the exact ordering of previous events, rather than by their absolute number. At some previous event, then, certain other events must have occurred in the valid ordering but not in the invalid one, or vice versa. Using the state at that point allows the two to be distinguished from each other. Using only the previous state allows a weakening term to be constructed that involves only the relationships among quantities at that previous event. When this is not sufficient to distinguish all valid orderings, then characterizing both the current and the previous state permits relations to be formed between current and previous quantities.

The weakening process is repeated until one of two things happens. If every valid ordering is allowed, by either the preliminary condition involving the current state or else by a weakening term involving some previous state as well, then a correct solution specification condition is thereby obtained. If instead, one or more valid orderings are still disallowed, and no event can be found that precedes the enter event in question in both the disallowed valid ordering(s) and all the invalid orderings, then the algorithm fails in constructing a condition. A discussion of situations in which the method fails will be postponed until Section 4.7.

4.4 An example using a previous state

This section contains an example of applying the algorithm as it has been presented in Sections 4.2 and 4.3. The example chosen is one for which the current state is insufficient for expressing the solution specification conditions, and previous states must be used. The specification to be analyzed here is example 7 from Section 2.7, to be denoted s_2:

$$(a_i^{enter} \Rightarrow b_j^{enter}) \leftrightarrow (c_i^{enter} \Rightarrow d_j^{enter})$$

The first step in the derivation process is to identify the set of event expressions in the specification. The set of event expressions in this case is given by

$$\mathrm{Evexp}(s_2) = \{a_i^{enter}, b_j^{enter}, c_i^{enter}, d_j^{enter}\}.$$

The next step is to construct all possible orderings among these event expressions. In this example there are no two events associated with the same procedure activation (such as $p_i^{request}$ and p_i^{enter}), nor are there two request events for the same procedure (such as $p_i^{request}$ and $p_{i+1}^{request}$). Therefore any of the 24 permutations of the four events in Evexp(s_2) represents a possible ordering among them. These 24 orderings are listed in Figure 4.3 and numbered for the sake of future reference.

Each of the constructed orderings is tested against the specification to determine whether it satisfies the specification and is therefore valid, or fails to satisfy it and is invalid. For example, in ordering (6), $(a_i^{enter} \Rightarrow b_j^{enter})$ and $(c_i^{enter} \Rightarrow d_j^{enter})$ are both FALSE, so that specification s_2 evaluates to the expression

$$\mathrm{FALSE} \leftrightarrow \mathrm{FALSE},$$

which is equal to TRUE. Ordering (6) therefore satisfies the specification. When the specification is evaluated for each of the first 12 orderings, it evaluates to TRUE, showing each of these orderings to be valid. Each of the last 12 orderings causes s to evaluate to FALSE, though, so that these orderings are invalid.

Figure 4.3. Possible orderings for specification s_2

(1) $a_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow d_j^{enter}$

(2) $a_i^{enter} \Longrightarrow c_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow d_j^{enter}$

(3) $a_i^{enter} \Longrightarrow c_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow b_j^{enter}$

(4) $b_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow c_i^{enter}$

(5) $b_j^{enter} \Longrightarrow d_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow c_i^{enter}$

(6) $b_j^{enter} \Longrightarrow d_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow a_i^{enter}$

(7) $c_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow b_j^{enter}$

(8) $c_i^{enter} \Longrightarrow a_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow b_j^{enter}$

(9) $c_i^{enter} \Longrightarrow a_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow d_j^{enter}$

(10) $d_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow a_i^{enter}$

(11) $d_j^{enter} \Longrightarrow b_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow a_i^{enter}$

(12) $d_j^{enter} \Longrightarrow b_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow c_i^{enter}$

(13) $a_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow d_j^{enter} \Longrightarrow c_i^{enter}$

(14) $a_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow b_j^{enter} \Longrightarrow c_i^{enter}$

(15) $a_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow b_j^{enter}$

(16) $b_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow c_i^{enter} \Longrightarrow d_j^{enter}$

(17) $b_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow a_i^{enter}$

(18) $b_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow a_i^{enter} \Longrightarrow d_j^{enter}$

(19) $c_i^{enter} \Longrightarrow d_j^{enter} \Longrightarrow b_j^{enter} \Longrightarrow a_i^{enter}$

(20) $c_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow d_j^{enter} \Longrightarrow a_i^{enter}$

(21) $c_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow d_j^{enter}$

(22) $d_j^{enter} \Longrightarrow c_i^{enter} \Longrightarrow a_i^{enter} \Longrightarrow b_j^{enter}$

(23) $d_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow c_i^{enter} \Longrightarrow b_j^{enter}$

(24) $d_j^{enter} \Longrightarrow a_i^{enter} \Longrightarrow b_j^{enter} \Longrightarrow c_i^{enter}$

The offending event in each invalid ordering can be identified by comparing the ordering with all the valid orderings to determine at what point the invalid ordering first fails to satisfy the specification. For example, invalid ordering (13) matches valid ordering (1) as far as the first two events are concerned. Since this is the longest prefix that does match the prefix of some valid ordering, the next event in (13), namely d_j^{enter}, is the offending event. When this is done for each of the 12 invalid orderings, it is found that the offending event is d_j^{enter} in orderings (13) through (15), c_i^{enter} in (16) through (18), b_j^{enter} in (19) through (21), and a_i^{enter} in (22) through (24).

A condition is needed for each of the four enter gates mentioned in the specification. Here the condition for the a^{enter} gate will be derived. To determine the condition for this gate, it is necessary to characterize the state at event a_i^{enter} in each of the 12 valid orderings as well as in each of the orderings in which it is the offending event, namely (22), (23), and (24). The characterizations one obtains for all of these orderings, using the characterization method described in the previous section, are listed in Figure 4.4, with characterization c_i applying to ordering i.

The formula that is obtained from disjoining the characterizations c_1 through c_{12} is given by

$$\begin{gathered}\exists\,(i,j)\;(\mathrm{count}(a^{enter}) < i \;\wedge\\ ((\mathrm{count}(b^{enter}) < j \;\wedge\; \mathrm{count}(c^{enter}) \geq i) \;\vee\\ (\mathrm{count}(b^{enter}) \geq j \;\wedge\; \mathrm{count}(d^{enter}) \geq j) \;\vee\\ (\mathrm{count}(c^{enter}) < i \;\wedge\; \mathrm{count}(d^{enter}) < j))).\end{gathered}$$

This formula is D_V, which represents a characterization of when the occurrence of such an

Figure 4.4. State characterizations at event a_i^{enter}

Valid orderings

c_1: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) < j)$

c_2: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) < j)$

c_3: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) < j)$

c_4: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) \geq j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) < j)$

c_5: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) \geq j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

c_6: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) \geq j \ \wedge\ \mathrm{count}(c^{enter}) \geq i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

c_7: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) \geq i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

c_8: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) \geq i \ \wedge\ \mathrm{count}(d^{enter}) < j)$

c_9: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) \geq i \ \wedge\ \mathrm{count}(d^{enter}) < j)$

c_{10}: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) \geq j \ \wedge\ \mathrm{count}(c^{enter}) \geq i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

c_{11}: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) \geq j \ \wedge\ \mathrm{count}(c^{enter}) \geq i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

c_{12}: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) \geq j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

Invalid orderings

c_{22}: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) \geq i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

c_{23}: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

c_{24}: $\exists\ (i,j)\ (\mathrm{count}(a^{enter}) < i \ \wedge\ \mathrm{count}(b^{enter}) < j \ \wedge\ \mathrm{count}(c^{enter}) < i \ \wedge\ \mathrm{count}(d^{enter}) \geq j)$

event satisfies the specification. The disjunction of c_{22}, c_{23}, and c_{24} is D_i, representing a general characterization of when occurrence would not satisfy the specification. This formula is equal to:

$$\exists\,(i,j)\ (\mathrm{count}(a^{enter}) < i\ \wedge\ \mathrm{count}(b^{enter}) < j\ \wedge\ \mathrm{count}(d^{enter}) \geq j).$$

The body of this expression contains the three conjuncts that appear in all three characterizations, whereas $\mathrm{count}(c^{enter})$ is greater than or equal to i in c_{22}, but less than i in the other two, so that these terms cancel out.

The preliminary condition that one obtains then is given by $(D_v \wedge (\neg\ D_i))$, which equals

$$\begin{gathered}\exists\,(i,j)\ (\mathrm{count}(a^{enter}) < i\ \wedge \\ ((\mathrm{count}(b^{enter}) < j\ \wedge\ \mathrm{count}(c^{enter}) \geq i)\ \vee \\ (\mathrm{count}(b^{enter}) \geq j\ \wedge\ \mathrm{count}(d^{enter}) \geq j)\ \vee \\ (\mathrm{count}(c^{enter}) < i\ \wedge\ \mathrm{count}(d^{enter}) < j)))\ \wedge \\ \forall\,(i,j)\ (\mathrm{count}(a^{enter}) \geq i\ \vee\ \mathrm{count}(b^{enter}) \geq j\ \vee\ \mathrm{count}(d^{enter}) < j).\end{gathered}$$

The terms involving i and j in the universally quantified expression can be separated, applying logical property (A2) from Section 4.2. This results in the formula

$$\begin{gathered}
\exists\,(i,j)\ (\mathrm{count}(a^{\mathrm{enter}}) < i\ \wedge \\
((\mathrm{count}(b^{\mathrm{enter}}) < j\ \wedge\ \mathrm{count}(c^{\mathrm{enter}}) \geq i)\ \vee \\
(\mathrm{count}(b^{\mathrm{enter}}) \geq j\ \wedge\ \mathrm{count}(d^{\mathrm{enter}}) \geq j)\ \vee \\
(\mathrm{count}(c^{\mathrm{enter}}) < i\ \wedge\ \mathrm{count}(d^{\mathrm{enter}}) < j)))\ \wedge \\
(\forall\, i\ (\mathrm{count}(a^{\mathrm{enter}}) \geq i)\ \vee \\
\forall\, j\ \ (\mathrm{count}(b^{\mathrm{enter}}) \geq j\ \vee\ \mathrm{count}(d^{\mathrm{enter}}) < j)).
\end{gathered}$$

Using distributivity, this can be expanded into

$$\begin{gathered}
(\exists\,(i,j)\ (\mathrm{count}(a^{\mathrm{enter}}) < i\ \wedge \\
((\mathrm{count}(b^{\mathrm{enter}}) < j\ \wedge\ \mathrm{count}(c^{\mathrm{enter}}) \geq i)\ \vee \\
(\mathrm{count}(b^{\mathrm{enter}}) \geq j\ \wedge\ \mathrm{count}(d^{\mathrm{enter}}) \geq j)\ \vee \\
(\mathrm{count}(c^{\mathrm{enter}}) < i\ \wedge\ \mathrm{count}(d^{\mathrm{enter}}) < j)))\ \wedge \\
\forall\, i\ (\mathrm{count}(a^{\mathrm{enter}}) \geq i)) \\
\vee \\
(\exists\,(i,j)\ (\mathrm{count}(a^{\mathrm{enter}}) < i\ \wedge \\
((\mathrm{count}(b^{\mathrm{enter}}) < j\ \wedge\ \mathrm{count}(c^{\mathrm{enter}}) \geq i)\ \vee \\
(\mathrm{count}(b^{\mathrm{enter}}) \geq j\ \wedge\ \mathrm{count}(d^{\mathrm{enter}}) \geq j)\ \vee \\
(\mathrm{count}(c^{\mathrm{enter}}) < i\ \wedge\ \mathrm{count}(d^{\mathrm{enter}}) < j)))\ \wedge \\
\forall\, j\ \ (\mathrm{count}(b^{\mathrm{enter}}) \geq j\ \vee\ \mathrm{count}(d^{\mathrm{enter}}) < j)).
\end{gathered}$$

The first disjunct reduces to FALSE, due to the conjunction of

$$\exists\, i\ (\mathrm{count}(a^{\mathrm{enter}}) < i)$$

and its negation. This leaves the formula

$$\exists\,(i,j)\ (count(a^{enter}) < i\ \wedge$$
$$((count(b^{enter}) < j\ \wedge\ count(c^{enter}) \geq i)\ \vee$$
$$(count(b^{enter}) \geq j\ \wedge\ count(d^{enter}) \geq j)\ \vee$$
$$(count(c^{enter}) < i\ \wedge\ count(d^{enter}) < j)))\ \wedge$$
$$\forall\, j\ \ (count(b^{enter}) \geq j\ \vee\ count(d^{enter}) < j).$$

This can be simplified to

$$\forall\, j\ \ (count(b^{enter}) \geq j\ \vee\ count(d^{enter}) < j),$$

or simply

$$count(b^{enter}) \geq count(d^{enter})$$

using logical property (A5). This is the preliminary condition in simplified form. However, when one checks this condition against each of the valid orderings, one finds that there is one ordering, namely (7), that violates the condition. This means that the preliminary condition is too strong, and must be weakened sufficiently so as to permit ordering (7).

It is at this point that the weakening method described in Section 4.3 must be employed. An event must be found that precedes a_i^{enter}, the enter event in question, in ordering (7) as well as in each of the invalid orderings for which a_i^{enter} is the offending event, those being (22), (23), and (24). The single event that occurs before a_i^{enter} in all these orderings is d_j^{enter}. Thus, an attempt is made to find a condition at this event that distinguishes the valid from the invalid orderings.

The state characterization at d_j^{enter} in ordering (7) is given by:

$$\exists\ (i,j)\ ([count(a^{enter})\ @\ d^{enter}] < i\ \wedge\ [count(b^{enter})\ @\ d^{enter}] < j\ \wedge$$
$$[count(c^{enter})\ @\ d^{enter}] \geq i\ \wedge\ [count(d^{enter})\ @\ d^{enter}] < j).$$

Since all quantities refer to the state at the d^{enter} event, the notation "$@\ d^{enter}$" is used on all counts. This becomes the term D_v', the disjunction of previous state characterizations for valid orderings. The characterization for each of orderings (22), (23), and (24) is the same, namely

$$\exists\ (i,j)\ ([count(a^{enter})\ @\ d^{enter}] < i\ \wedge\ [count(b^{enter}\ @\ d^{enter}] < j\ \wedge$$
$$[count(c^{enter})\ @\ d^{enter}] < i\ \wedge\ [count(d^{enter})\ @\ d^{enter}] < j),$$

so the disjunction of characterizations D_i' is equal to this as well. The proposed weakening term is given by (D_v' $\wedge$ ($\neg$ D_i')), which equals

$$\exists\ (i,j)\ ([count(a^{enter})\ @\ d^{enter}] < i)\ \wedge\ [count(b^{enter})\ @\ d^{enter}] < j\ \wedge$$
$$[count(c^{enter})\ @\ d^{enter}] \geq i\ \wedge\ [count(d^{enter})\ @\ d^{enter}] < j)\ \wedge$$
$$\forall\ (i,j)\ ([count(a^{enter})\ @\ d^{enter}] \geq i\ \vee\ [count(b^{enter})\ @\ d^{enter}] \geq j\ \vee$$
$$[count(c^{enter})\ @\ d^{enter}] \geq i\ \vee\ [count(d^{enter})\ @\ d^{enter}] \geq j).$$

Simplifying, this formula becomes

$$\exists\ i\ ([count(a^{enter})\ @\ d^{enter}] < i)\ \wedge\ [count(c^{enter})\ @\ d^{enter}] \geq i),$$

which reduces to

$$[count(a^{enter})\ @\ d^{enter}] < [count(c^{enter})\ @\ d^{enter}]$$

by logical property (E5).

When this condition is tested in ordering (7), it is found to be satisfied. Therefore, this term is disjoined to the preliminary condition to obtain the final solution specification condition:

$$\mathrm{count}(b^{enter}) \geq \mathrm{count}(d^{enter}) \ \vee \ [\mathrm{count}(a^{enter}) \bullet d^{enter}] < [\mathrm{count}(c^{enter}) \bullet d^{enter}],$$

The method illustrated in deriving the condition for gate a^{enter} must be applied again for each of the other gates b^{enter}, c^{enter}, and d^{enter}. Due to the symmetry of the specification, these derivations are completely isomorphic.

4.5 Incorporating argument constraints

The previous sections have presented the method for deriving a solution specification from the problem specification, under the assumption that each clause in the specification is an ordering clause of the form $e_1 \Rightarrow e_2$, for some events e_1 and e_2. When a specification also contains other clauses in the form of argument constraints, these constraints must first be incorporated into the ordering clauses of the specification before the algorithm described previously can be used.

To simplify the discussion, it will be assumed that argument constraint clauses appear only as conjuncts in the hypothesis of an implication. A specification that does not satisfy this condition can be transformed into an equivalent one that does as follows: Any specification can be put into conjunctive normal form (CNF) by well-known techniques of first-order logic. Each conjunct (which is analyzed separately, as explained previously) then consists of a series of disjuncts, some of which may be argument constraint clauses and at least one of which must be an ordering clause. The general form of such a conjunct is therefore:

$$N_1 \vee N_2 \vee ... \vee N_j \vee O_1 \vee ... \vee O_k,$$

where each N_i is a (possibly negated) argument constraint clause and each O_i is a (possibly negated) ordering clause, and $j \geq 0$ and $k \geq 1$. This can be transformed, using the tautology $(x \supset y) \leftrightarrow (\neg x \vee y)$, into:

$$((\neg N_1) \wedge (\neg N_2) \wedge ... \wedge (\neg N_j)) \supset (O_1 \vee ... \vee O_k).$$

In this way, all of the argument constraint clauses of the specification, some in negated form, are brought into the hypothesis of the implication, while all ordering clauses are in the conclusion of the implication.

An argument constraint clause can involve either invocation number variables or arguments to procedure activations. When a clause involves invocation number variables, it simply represents a constraint on those variables appearing in the specification. This constraint must be incorporated into every state characterization. Otherwise, the clause can be ignored in the other steps of the derivation process.

As an example, consider the first conjunct of the alternative producer-consumer specification of example 8 in Chapter 2:

$$(i = j) \supset (dep_i^{exit} \Rightarrow rem_j^{enter}).$$

The clause $(i = j)$ is ignored for the moment, and the ordering clause is analyzed by the regular method. Of the two orderings possible on the two events in the conjunct, the ordering $(dep_i^{exit} \Rightarrow rem_j^{enter})$ is valid, while the other ordering $(rem_j^{enter} \Rightarrow dep_i^{exit})$ is not. The offending event is clearly rem_j^{enter}, and a condition must be constructed for the rem^{enter} gate. The state characterization at event rem_j^{enter} in the valid ordering would be

$$\exists\,(i,j)\,(\mathrm{count}(\mathrm{rem}^{\mathrm{enter}}) < j \;\wedge\; \mathrm{count}(\mathrm{dep}^{\mathrm{exit}}) \geq i),$$

except that here the clause $(i = j)$ must be added as a conjunct of the characterization, giving:

$$\exists\,(i,j)\,(\mathrm{count}(\mathrm{rem}^{\mathrm{enter}}) < j \;\wedge\; \mathrm{count}(\mathrm{dep}^{\mathrm{exit}}) \geq i \;\wedge\; (i = j)).$$

This expression represents D_V.

For the invalid ordering, the state characterization at event $\mathrm{rem}_j^{\mathrm{enter}}$ also must include the clause $(i = j)$. This formula is:

$$\exists\,(i,j)\,(\mathrm{count}(\mathrm{rem}^{\mathrm{enter}}) < j \;\wedge\; \mathrm{count}(\mathrm{dep}^{\mathrm{exit}}) < i \;\wedge\; (i = j)),$$

which constitutes the formula D_I. The preliminary condition is $D_V \wedge (\neg\, D_I)$, or:

$$\exists\,(i,j)\,(\mathrm{count}(\mathrm{rem}^{\mathrm{enter}}) < j \;\wedge\; \mathrm{count}(\mathrm{dep}^{\mathrm{exit}}) \geq i \;\wedge\; (i = j)) \;\wedge$$

$$\forall\,(i,j)\,(\mathrm{count}(\mathrm{rem}^{\mathrm{enter}}) \geq j \;\vee\; \mathrm{count}(\mathrm{dep}^{\mathrm{exit}}) \geq i \;\vee\; (i \neq j)).$$

When this condition is simplified, it reduces to:

$$\mathrm{count}(\mathrm{rem}^{\mathrm{enter}}) < \mathrm{count}(\mathrm{dep}^{\mathrm{exit}}),$$

which is the condition on the $\mathrm{rem}^{\mathrm{enter}}$ gate required in the solution specification. This same condition is obtained when analyzing the specification

$$\mathrm{dep}_i^{\mathrm{exit}} \Longrightarrow \mathrm{rem}_i^{\mathrm{enter}},$$

in which the same property is specified, with the equality between the invocation numbers of the two activations indicated implicitly.

This illustrates the general technique for handling relational clauses that involve invocation numbers. As it shows, such clauses are integrated in a relatively simple manner into the method previously given for constructing a solution specification, since they simply represent additional information that must be included in each state characterization. For predicates on arguments to procedure activations, the matter is not quite so simple. The rest of this section is devoted to discussing how to handle such clauses.

An additional assumption that will be made concerning relations involving the arguments to procedure activations is that all such relations are made explicit. An example of an implicit relationship is a specification involving two procedure activations $p_i(x)$ and $q_j(x)$. Here the implicit relationship is that of equality of the arguments to the two activations. This can be made explicit by changing the argument of q_j to some new identifier y, and adding the predicate $(x = y)$ as a hypothesis of the specification. The situation would be handled in a similar manner if the argument to q_j were not x but instead $(x+1)$ or any other function of x.

Argument constraint clauses are incorporated into the ordering clauses of a specification by qualifying all affected procedure activations. Once a clause has been incorporated by means of qualification, it can be eliminated from the specification, so that the result of the qualification phase of the algorithm is to transform the specification into one involving only ordering clauses. After this transformation has been accomplished, the specification contains some procedure activations that are qualified. Qualified activations in a specification result in a solution specification containing qualified gates. Specifically, a qualified gate is required in an event class for each event expression in that class appearing in the specification and involving a qualified activation. The conditions required on all enter gates in the solution specification, qualified or unqualified, can be derived by the method already presented. In the derivation of these conditions, the qualifying predicates on procedure activations are transferred to the associated gates. Both the enter gates for which conditions are constructed, and the gates on which counts are taken, may be qualified.

The general form of a qualified procedure activation is:

$$[p_i(v) \mid C(v, t_1, \ldots, t_m)],$$

where v is the vector of parameters to procedure activation p_i, and C is some predicate involving these parameters and also possibly some new variables t_1 through t_m that do not appear in the specification. (The use of these "new" variables is explained below.) The qualifying predicate C represents an implicit restriction on the universal quantification of the invocation number i in the expression, restricting i to those invocation numbers for which the corresponding activations satisfy condition C. This means that this event expression can only represent events whose arguments satisfy predicate C.

Each clause that involves only the argument to a single procedure activation is incorporated into the specification by attaching the clause to the given activation as a qualifying predicate. For example, let v_1 be the vector of arguments to procedure p, and v_2 be the vector of arguments to procedure q. Consider the following specification, where C_1 is a predicate only involving v_1 and C_2 is a predicate only involving v_2:

$$(C_1(v_1) \wedge C_2(v_2)) \supset$$
$$((p_i(v_1)^{request} \Longrightarrow q_j(v_2)^{enter}) \supset (p_i(v_1)^{enter} \Longrightarrow q_j(v_2)^{enter})).$$

Predicate C_1 can be incorporated into the specification by qualifying procedure activation p_i, so that $p_i(v_1)$ becomes

$$[p_i(v_1) \mid C_1(v_1)].$$

Predicate C_2 can be incorporated by qualifying activation q_j to

$$[q_j(v_2) \mid C_2(v_2)].$$

This transforms the specification itself to:

$$([p_i(v_1) \mid C_1(v_1)]^{request} \Rightarrow [q_j(v_2) \mid C_2(v_2)]^{enter}) \supset$$
$$([p_i(v_1) \mid C_1(v_1)]^{enter} \Rightarrow [q_j(v_2) \mid C_2(v_2)]^{enter}).$$

The meaning of this specification is that any activation of p satisfying qualifying predicate C_1 and any activation of q satisfying C_2 must obey the ordering constraint given, but other activations of these operations need not. This is exactly the meaning of the original specification: If $C_1(v_1)$ and $C_2(v_2)$ are both true, then the events must satisfy the ordering constraint in order for the history containing those events to be valid. If either of the qualifying predicates is not true, then the history is valid according to the specification regardless of the order among the events.

In deriving a solution specification for this specification, there must be gates with qualifying predicate $C_1(v_1)$ in the $p^{request}$ and p^{enter} event classes, and a gate with qualifying predicate $C_2(v_2)$ in the q^{enter} event class. The entry conditions in the solution specification are derived just as if the activations were unqualified, except that the enter gates for which the conditions are derived, and the gates on which counts are taken, must be qualified appropriately. Without the argument constraint predicates, this specification would be s_1, the example analyzed in Section 4.2, where the condition

$$count(p^{request}) = count(p^{enter}),$$

was derived for gate q^{enter}. With the predicates included in the specification, the same analysis results in the condition

$$\mathrm{count}([p(v_1) \mid C_1(v_1)]^{request}) = \mathrm{count}([p(v_1) \mid C_1(v_1)]^{enter}),$$

for the qualified gate $[q(v_2) \mid C_2(v_2)]^{enter}$. That is, the qualification $C_1(v_1)$ on activation $p_i(v_1)$ results in qualifying the gates $p^{request}$ and p^{enter}, on which the counts are taken, with this same predicate. The qualification $C_2(v_2)$ on activation $q_j(v_2)$ is attached to the q^{enter} gate for which the condition is derived.

A predicate involving arguments to more than one procedure activation is converted into a conjunction of different predicates, each of which only involves the arguments to a single activation. This is accomplished by parameterizing the original predicate in terms of some new variable t. Once this is done, then the same method of qualification as discussed above can be used. For example, the predicate $(x = y)$, where x and y are arguments to different procedure activations, is transformed into the two predicates $(x = t)$ and $(y = t)$. Each of these two predicates is then incorporated into the specification by using it to qualify the appropriate activation.

As a result, the specification

$$(x = y) \supset$$
$$((p_i(x)^{request} \Rightarrow q_j(y)^{enter}) \supset (p_i(x)^{enter} \Rightarrow q_j(y)^{enter}))$$

is transformed into

$$((x = t) \wedge (y = t)) \supset$$
$$((p_i(x)^{request} \Rightarrow q_j(y)^{enter}) \supset (p_i(x)^{enter} \Rightarrow q_j(y)^{enter}))$$

by parameterizing the predicate $(x = y)$. Incorporating the two predicates $(x = t)$ and $(y = t)$ into the appropriate procedure activations further transforms the specification into

$$([p_i(x) \mid (x=t)]^{request} \Longrightarrow [q_j(y) \mid (y=t)]^{enter}) \supset$$

$$([p_i(x) \mid (x=t)]^{enter} \Longrightarrow [q_j(y) \mid (y=t)]^{enter})$$

Since this is again simply specification s_1 with qualifying predicates on the procedure activations, the resulting solution specification contains condition

$$count([p(x) \mid (x=t)]^{request}) = count([p(x) \mid (x=t)]^{enter}),$$

for the qualified gate $[q(y) \mid (y=t)]^{enter}$.

The meaning of this solution specification is the following: For whatever value of t is equal to parameter y of an activation of operation q, the enter event for that activation passes through the gate $[q(y) \mid (y=t)]^{enter}$. The condition for that gate is given by

$$count([p(x) \mid (x=t)]^{request}) = count([p(x) \mid (x=t)]^{enter}),$$

for this same value of t. Therefore, the "gate" $[q(y) \mid (y=t)]$ actually represents an entire set of gates, one for each value of t, which is to say each possible value of y.

An argument constraint predicate can always be parameterized into several predicates, each of which involves only the arguments to one procedure activation. In fact, many such ways of parameterizating a given predicate are possible. For reasons having to do with the implementation that are discussed in Chapter 5, it is desirable that at most one of the new parameterized predicates be a non-functional relation between the activation parameters and the parameterizing variable(s), and furthermore that this possibly non-functional relation apply to the arguments of the activation whose enter event is the offending event. This restriction can always be followed in practice.

Once a predicate has been parameterized, the resulting predicates then can be used to qualify the corresponding procedure activations. When all predicates have been so incorporated, the specification consists entirely of ordering clauses involving qualified procedure activations. This specification can be analyzed by the method presented previously, resulting in a solution specification containing qualified gates.

4.6 Justification of the derivation method

Both the problem specification language and the solution specification structure have been defined formally in terms of a common basis, the validity of histories. This means that the equivalence of a problem specification and the solution specification that is derived from it can be discussed in terms of the same set of histories being valid with respect to each. Rather than attempt a formal proof of correctness for the derivation method, this section will present an informal justification of the method. The justification will rely, however, on the formal definitions given for validity of histories. The complete derivation algorithm is presented in Figure 4.5, with the individual steps numbered for ease of reference throughout this section.

In discussing the validity of histories with respect to both problem specification s and solution specification ss, we can refer to the definitions of the predicates Valid from Chapter 2 and ValidSS from Chapter 3. They are repeated here:

Figure 4.5. Derivation of solution specification ss from problem specification s

(1) Transform s into a logically equivalent specification in which all argument constraint clauses are in the hypothesis of an implication and all ordering clauses are in the conclusion.

(2) Parameterize each predicate on the arguments to more than one procedure activation into two or more predicates, each of which applies only to the arguments of a single activation.

(3) Incorporate each argument constraint clause that applies to the arguments to a procedure activation by qualifying each appearance of that activation using the given clause as the qualifying predicate. The result is a transformed specification, to be denoted s'. Specification s' consists entirely of ordering clauses on qualified events, except possibly for clauses involving invocation number variables only, appearing in the hypothesis of the implication. These clauses are ignored until step (8).

(4) Construct the set Evexp(s') consisting of all event expressions, including qualifying predicates, that appear in s'. The set of (possibly qualified) event classes associated with these event expressions represents the set of gates required in solution specification ss.

(5) Construct all possible orderings of the elements of Evexp(s'), by generating all permutations of this set and then eliminating all those that are not possible.

(6) Evaluate specification s' for each ordering, denoting each ordering that evaluates s' to TRUE as valid, and each that evaluates it to FALSE as invalid.

(7) For each invalid ordering, find the longest matching prefix that it shares with some valid ordering, and identify the event following this prefix in the ordering as the offending event. If the offending event is not an enter event, then the specification is regarded as erroneous, and the algorithm terminates without being able to derive a solution specification.

(8) For each enter gate (either qualified or unqualified) that applies to the offending event in at least one invalid ordering, characterize the state at each event to which the gate applies that appears in a valid ordering, and disjoin these characterizations to form D_V. Also, characterize the state at each offending event in an invalid ordering to which the gate applies, and disjoin these characterizations to form D_I. Any clauses in s' constraining invocation number variables must be included in each state characterization.

(9) For each enter gate for which step (8) is carried out, form the preliminary condition given by $(D_V \wedge \neg(D_I))$. Test whether this condition is satisfied at every event to which the gate applies that appears in a valid ordering. If so, then the preliminary condition is the condition for that gate in solution specification ss. If not, then proceed to step (10).

(10) Find an event that precedes the given enter event in every valid ordering that is excluded by the condition so far, and also in every invalid ordering whose offending event applies to the given gate.

(11) Characterize the state at each of these points, and form disjunctions D_v' and D_i' of these characterizations analogous to those formed in step (8).

(12) Test all valid orderings still excluded to determine which satisfy the term ($D_v' \wedge \neg(D_i')$). If at least one such ordering does satisfy this term, disjoin the term to the current condition.

(13) If some orderings still do not satisfy the condition, then repeat steps (11) and (12) but using the characterizations both for the current state and at the previous event.

(14) Repeat steps (10) through (13) until either all valid orderings satisfy the condition or the weakening term in step (12), or no previous event can be found in step (10). If the former, then the condition formed by disjoining the preliminary condition and all the weakening terms from step (12) is correct and is attached to the gate in solution specification ss. If the latter, then the method fails to derive a solution specification for problem specification s.

$$\begin{aligned}
&\text{Valid}([\,], s) = \text{TRUE}\\
&\text{Valid}(\text{add}(h, e), s) = \text{Valid}(h, s) \wedge\\
&\qquad\qquad \forall\,(ee, f)\,(ee \in \text{Evexp}(s) \wedge f \text{ is an interpretation}\\
&\qquad\qquad\qquad \wedge \text{Match}(e, ee, f) \supset \text{Sat}(h, e, s, f)
\end{aligned}$$

$$\begin{aligned}
&\text{ValidSS}([\,], ss) = \text{TRUE}\\
&\text{ValidSS}(\text{add}(h, <p, t, n, a>), ss) = \text{ValidSS}(h, ss) \wedge\\
&\qquad\qquad \forall\,(ec, q)\,(<ec, q> \in \text{Gates}(ss) \wedge ec = <p, t> \wedge q(a)\\
&\qquad\qquad\qquad \supset \text{SatSS}(h, ss, <ec, q>)
\end{aligned}$$

It is straightforward to compare these two definitions. They are both recursive formulas in which the basis case is the empty history [ ] and yields a value of TRUE. Also, both of the terms for the inductive case, which is add(h, e), are a conjunction of the given predicate applied to history h, and some term involving h and the last event e. Therefore, by recursion induction ([McC62]), the two definitions are equivalent if and only if these last terms are equivalent for all histories h and all events e = <p, t, n, a>. That is, it must be the case that

$$\begin{aligned}
&\forall\,(ee, f)\,(ee \in \text{Evexp}(s) \wedge f \text{ is an interpretation} \wedge \text{Match}(<p, t, n, a>, ee, f)\\
&\qquad \supset \text{Sat}(h, <p, t, n, a>, s, f)
\end{aligned}$$

if and only if

$$\begin{aligned}
&\forall\,(ec, q)\,(<ec, q> \in \text{Gates}(ss) \wedge ec = <p, t> \wedge q(a)\\
&\qquad \supset \text{SatSS}(h, ss, <ec, q>).
\end{aligned}$$

The first term requires predicate Sat to be true for all interpretations under which the event matches an expression in the specification. The second one states that for all gates in the solution specification "matched" by the event, predicate SatSS is true. These two terms must be equivalent for problem specification s and solution specification ss to be equivalent, in the sense that they allow the exact same subset of possible object histories to be valid.

Steps (1) through (3) of the derivation method transform the original specification s into a new specification s'. To justify this transformation, it must be shown that specifications s and s' are equivalent with respect to the validity predicate Valid, which really means with respect to the satisfaction predicate Sat. Step (1), in which all argument constraint clauses are brought into the hypothesis of an implication simply involves properties of first-order logic. Step (2), in which predicates involving arguments to different activations are parameterised, is also mathematically straightforward.

To justify step (3), let us look at the transformation that it accomplishes. We start from a specification of the form $Q(v) \supset s_1$, where Q is a qualifying predicate on some parameter vector v and s_1 is some specification involving only ordering clauses. According to the definition of Sat,

$$Sat(h, e, Q(v) \supset s_1, f) = (Sat(h, e, Q(v), f) \supset Sat(h, e, s_1, f)).$$

Furthermore, Q must be some combination of arithmetic relations, which are invariant under the Sat predicate, since

$$Sat(h, e, exp_1 \text{ rel } exp_2, f) = (f(exp_1) \text{ rel } f(exp_2)).$$

This means that if the interpretation of v by f satisfies the qualifying predicate Q, then the ordering specification s_1 must be satisfied by event e and history h under interpretation f. If Q is not satisfied by the interpretation of v by f, then it does not matter whether s_1 is satisfied under f, since the overall specification is satisfied regardless. This is exactly the result of qualifying the appropriate procedure activation in s_1 with predicate Q on v. The constraint represented by s_1 must be satisfied only if the qualifying predicate itself is satisfied. Therefore, the transformation resulting from step (3) is consistent with preserving the meaning of the specification, and the value returned by the Sat predicate is the same when applied to the transformed specification s' as to the original s.

The next steps of the algorithm construct the possible orderings of the events for which the specification contains expressions. These orderings represent sub-sequences within general histories. The history in which each ordering is embedded is assumed to be otherwise valid with respect to the specification. For this reason, an invalid orderings results in a history that is not valid, while a valid ordering maintains the validity of the overall history. Therefore, a condition that distinguishes the valid from the invalid orderings is required to distinguish all valid histories from invalid ones.

Step (4) of the derivation algorithm consists of the construction of the set Evexp(s') of event expressions in the specification. This can be accomplished by using the formal definition of this set in Chapter 2. The gates required in the solution specification are exactly the gates associated with this set of event expressions. If the specification refers to a certain set of qualified event classes, the solution specification must contain exactly this set of gates, since it is these classes of events that the guardian must keep track of in order to implement the specified constraint.

In step (5) all possible orderings among the elements of Evexp(s') are constructed. Each ordering actually represents a sub-sequence of a history, containing exactly those events that are represented by the event expressions in s' under some interpretation. Since there is no restriction at all on the quantification, the range of the interpretation is the complete set of all interpretations. This means that the orderings together constitute the entire class of possible sub-histories that consist of the events represented in the specification under any interpretation.

An ordering is considered possible unless (a) there is some procedure activation whose enter event precedes its request event, or whose exit event precedes its enter event; or (b) there are two request events for the same procedure such that the invocation number of the earlier one is greater than the invocation number of the later one under all interpretations. This step corresponds to the restriction of the domain of Valid to histories satisfying predicate Possible, which embodies these same restrictions.

In Step (6), each ordering is used to evaluate specification s', resulting in a classification of each ordering as either valid or invalid. Since the implicit interpretation by which the event expressions correspond to actual events is unrestricted, an ordering is valid only if, under any interpretation whatsoever, each event in it satisfies the specification. An invalid ordering, on the other hand, represents a sub-history which under some interpretation does not satisfy the specification. This is equivalent to the definition of the Valid predicate, where for a history to be valid, each event in it must satisfy the specification for *all* interpretations.

The identification of the offending event for each invalid ordering in step (7) is straightforward. The validity of a history with respect to a specification is defined in terms of each successive event in the history satisfying the specification. Since the history in which the ordering is embedded is valid otherwise, the first event at which an invalid ordering fails to match some valid ordering is the "offending" one. All events preceding this one must satisfy the specification according to the predicate Sat. The definition of the validity of a history with respect to a solution specification similarly is in terms of each event satisfying the solution specification conditions. This means that the offending event must be the point at which SatSS is first not satisfied, and therefore a condition must exist that is violated here.

Step (8) requires the state to be characterized at each point representing either a valid or offending occurrence of an event of the given (qualified) event class. As described in Section 4.2, this characterization is made by existentially quantifying all variables and putting bounds on the counts of all gates involved in an ordering. The existential quantification of variables signifies the fact that the event expressions correspond to actual events under some unknown interpretation, and that every variable is therefore replaced by some unknown value. Each bound on the count of passages through a gate is either a lower or upper bound depending upon whether the event at that gate precedes or follows the point at which the characterization is made. If event x_m^{enter} precedes this point, then count(x^{enter}) is presumed to be at least m, while if x_m^{enter} follows this point, then count(x^{enter}) is presumed to be less than m. For an event involving a qualified activation, it is the count of the appropriately qualified event class that is bounded.

This characterization is accurate because the scheduling at each gate is first-come-first-served. According to the solution specification structure, two activations whose parameters satisfy the same set of qualifying predicates must pass through exactly the same set of gates. Since the queue for each event class is FIFO, these activations must proceed in first-come-first-served order. The rest of the state characterization method simply involves introducing the existential quantification on invocation number values explicitly, and including any explicit constraints on these values that may appear as clauses in s'.

The characterization that is formed for each ordering represents the most general expression possible of the current state following the occurrence of a sub-history corresponding to the given ordering. Nothing is assumed about the rest of the history except what can be deduced directly from the events in the ordering itself. All unknown values in the formula are existentially quantified, so the formula simply states that there exist *some* values for which its body is true. That is to say, there exists *some* interpretation causing a sub-history to correspond to this ordering. Therefore, D_V, the disjunction of the characterizations from all the valid orderings, represents the most general expression of when an event in the given qualified class can validly occur. Using the formal semantic definitions of Sections 2.6 and 3.4, it is the most general characterization of CurSt(h) for histories h which, when followed by some event e in the given class, satisfies the specification s' (by the definition of Sat) for any interpretation f. Similarly, D_I, the disjunction of the characterizations from all the invalid orderings, represents the most general expression of when such an event cannot validly occur. This means that it is the most general characterization of CurSt(h) for histories that under some interpretation do *not* satisfy the specification when followed by an event in the class.

The preliminary condition formed by $(D_V \wedge \neg(D_I))$ in step (9) represents an attempt to incorporate all histories with which an event of the given class satisfies the specification for all interpretations, and to rule out all those with which it does not. It is the conjunction of two terms, one of which is the negation of D_I, the expression of when the event cannot occur. For this reason, it is guaranteed to be a strong enough condition to exclude all invalid orderings, and therefore all histories that do not satisfy the Sat predicate for the specification. Therefore, no history that does not satisfy Sat will satisfy SatSS for the solution specification containing this condition for the given gate. Testing the condition against all valid orderings determines whether or not it is weak enough to allow all histories satisfying Sat. If so, then it is the correct condition, in that it causes exactly the correct set of histories to satisfy the SatSS predicate as well. If not, then there are some histories that satisfy Sat but would not satisfy SatSS if ss contains the given condition.

Progressively weakening the condition allows more histories to satisfy SatSS. This weakening is accomplished by repeating steps (10) through (13) using previous states, each time disjoining the resulting terms to the previous condition if they allow more valid orderings to satisfy the condition. The weakening term constructed from the first application of steps (11) and (12) involves only quantities in the previous state. If this is found in step (13) to be not sufficient, then repeating steps (11) and (12) allows a term to be constructed that involves relations between quantities in the previous state and those in the current state. Since each weakening term is of the form $(D_V' \wedge \neg(D_I'))$, just as the preliminary condition is, no invalid orderings can become allowed as a result of this process. By choosing each time an event that precedes the given point in all remaining valid orderings still not allowed by the condition, the weakening terms constructed have a good chance of including most if not all of the remaining valid orderings. Therefore, steps (10) through (13) in practice rarely need to be repeated more than once or twice. Eventually, all valid orderings must be included, unless the algorithm fails due to an inability to find a previous state in step (10) to use in constructing new weakening terms. Specifications for which this the algorithm fails are the subject of the last section of this chapter.

4.7 Failure of the derivation algorithm

The structure of the solution specification is flexible enough to express the solutions to a large class of synchronization problems. However, certain features do limit somewhat the range of synchronization constraints that can be expressed. The solution specification structure is less general than the problem specification language, so that for some specifications the derivation algorithm is unable to construct equivalent solution specifications. As noted in Section 4.2, this sometimes is manifested by finding the offending event in an ordering to be other than an enter event. Since this would imply a condition on a request or exit gate, such a specification is incompatible with the solution specification structure that only places conditions on enter gates. The algorithm therefore fails whenever an invalid ordering is found for which the offending event is not an enter event.

The other manifestation of incompatibility with the structure of the solution specification is an inability to find sufficient previous states at which terms can be constructed to weaken conditions. An example of such an incompatible specification is the "last-come-first-served" (LCFS) scheduling specification of Example 6 in Section 2.7:

$$(p_i^{\text{request}} \Rightarrow p_j^{\text{request}} \Rightarrow p_i^{\text{enter}}) \supset (p_j^{\text{enter}} \Rightarrow p_i^{\text{enter}}).$$

When the derivation algorithm is applied to this specification, the following preliminary condition is first constructed for gate p^{enter}:

$$\text{count}(p^{\text{request}}) = \text{count}(p^{\text{enter}}) + 1$$

This condition is found not to be satisfied by one of the events occurring in a valid ordering, however, namely p_j^{enter} in the valid ordering

$$p_i^{\text{request}} \Rightarrow p_j^{\text{request}} \Rightarrow p_j^{\text{enter}} \Rightarrow p_i^{\text{enter}}.$$

This must be distinguished from the offending event p_i^{enter} in the invalid ordering

$$p_i^{\text{request}} \Rightarrow p_j^{\text{request}} \Rightarrow p_i^{\text{enter}} \Rightarrow p_j^{\text{enter}}$$

on the basis of previous state information. Since these orderings differ only in the identity of which of the two p^{enter} events occurs first, and the identity is not reflected in any predicate on the parameters of the two activations, it is obvious that the two cannot be distinguished. In applying the algorithm, there are two previous events at which possible weakening terms can be constructed: the most recent, and next-to-most recent, p^{request} events. However, the state characterizations for the two orderings are identical in each case, resulting in potential weakening terms that are identically FALSE and thus not useful. As a result, the derivation ends in failure, since no other possible weakening terms are available.

The reason for the failure of the algorithm on this specification is that the property specified requires two different activations to be distinguished, not on the basis of their parameters, but simply by their identity. A solution specification condition for this constraint would have to depend on not only the *number* of previous events, which would involve the current synchronization state, or even the *order* of these events, since this information can always be obtained from previous state information, as explained in Section 3.3. Instead, the constraint relies on distinguishing the *identity* of two different activations. However, since there is no parameter-related property by which to distinguish the two activations, the structure of the solution specification requires that the activations pass through the same gate or set of gates for the p^{enter} event class in FIFO order. The requirement in the specification of non-FIFO scheduling is in direct contradiction with the solution specification structure. This is why the derivation algorithm cannot possibly succeed in deriving a solution specification for this specification.

Synchronization constraints such as the LCFS specification that rely on the identity of particular events are rather unusual in practice, and their incompatibility with the solution specification structure is not terribly distressing. A second kind of incompatibility, though, is demonstrated by a very commonly desired property, the first-come-first-served (FCFS) specification of Example 5 in Section 2.7:

$$(p_i^{request} \Rightarrow q_j^{request}) \leftrightarrow (p_i^{enter} \Rightarrow q_j^{enter}).$$

This specification, somewhat surprisingly, is also one for which the derivation of a solution specification fails. The reason is that this synchronization constraint cannot be implemented using one queue for each event class and one entry condition for each queue. An implementation using the serializer construct appears in [Hew77] for a FCFS scheduling property on two operations "read" and "write", but this relies on the two operations sharing the same queue, though with different entry conditions. A monitor implementation was devised in an unpublished note [Bro76], but here again the two operations shared a single queue, with one of the operations using a second auxiliary queue as well.

The reason that a solution specification cannot be constructed for this property is that it would be necessary to save information at a previous state that is arbitrarily far back in the history. The solution specification structure allows states to be saved at the most recent event at a gate, and by extension, at the next-to-most recent, etc. However, the FCFS constraint requires that each enter event use information from the point in the history at which the corresponding request event took place, which may be arbitrarily far back. That is, the condition for enter events by different processes must involve information saved at previous states individually applicable to each process. Specifically, let

$$[\text{count(ec) private} \bullet p^{\text{request}}]$$

be a quantity that for any particular activation of operation p represents the value of count(ec) saved at its request event. Then the conditions on gates p^{enter} and q^{enter} could be expressed as:

$$p^{\text{enter}}\text{:}\;\; [\text{count}(q^{\text{request}}) \text{ private} \bullet p^{\text{request}}] = \text{count}(q^{\text{enter}})$$

$$q^{\text{enter}}\text{:}\;\; [\text{count}(p^{\text{request}}) \text{ private} \bullet q^{\text{request}}] = \text{count}(p^{\text{enter}})$$

That is, there must be as many q^{enter} events at the time of a p^{enter} event as there were q^{request} events when the given activation of p was requested.

The use of this kind of information that is "private" to each process appears in [Owi76] to specify solutions to synchronization problems. Interestingly, the LCFS property can also be expressed with the use of private information. The condition on gate p^{enter} becomes:

$$(\mathrm{count}(p^{request}) - [\mathrm{count}(p^{request}) \text{ private} \bullet p^{request}]) =$$
$$(\mathrm{count}(p^{enter}) - [\mathrm{count}(p^{enter}) \text{ private} \bullet p^{request}])$$

In other words, all requests for p since this activation of p was requested must first be fulfilled.

The solution specification can only save states at a "fixed" distance back from the current state, where "fixed" is relative to the number of events at a gate. Information privately saved by each process must be saved at states arbitrarily far back in the history. Without such privately saved information, the solution specification structure is unable to express certain properties, including the rather straightforward FCFS property. This must be considered a weakness of the solution specification and therefore of the synthesis method. However, it is nevertheless true that most specifications are compatible with the solution specification structure, so that the derivation algorithm does succeed in constructing equivalent solution specifications in most cases. The next chapter describes the last step in the synthesis for these cases, the implementation of the solution specification in actual code.

Chapter 5

The Source Language Implementation

5.1 Introduction

The derivation of an equivalent solution specification from a problem specification, using the algorithm presented in Chapter 4, constitutes the major conceptual task involved in synthesizing actual synchronization code. The derived solution specification is a procedural representation of the same ordering constraint that is expressed non-procedurally by the problem specification. The final step in the synthesis is implementing the solution specification in terms of an appropriate source language synchronization mechanism. The translation from solution specification to source language is the subject of this chapter, and while relatively straightforward, it is not completely obvious for all cases.

The structure of the solution specification is general enough for it to be translated into any one of a wide range of source language synchronization mechanisms. For purposes of explaining and illustrating the translation technique, the monitor construct of Hoare ([Hoa74]) will be used throughout the thesis. An implementation using an alternative high-level synchronization mechanism such as conditional critical regions ([Bri72]) or serializers ([Hew77]) would be quite similar. If a lower-level mechanism such as semaphores ([Dij68]) is preferable, then an algorithm given in [Hoa74] can be used to further translate the monitor implementation given here into semaphore code.

A fundamental assumption of the model used here is that all synchronization for a data object takes place through a single centralized mechanism associated with that object. This does not cause any problems with an implementation in terms of monitors, or any of the other constructs cited above. However, it does make the solution specification structure somewhat incompatible with situations in which a data object is distributed throughout some decentralized system, and where it is desirable for the synchronization control similarly to be distributed. The structure of the solution specification does not give much aid in deciding how to perform the message passing required in a distributed system to implement the synchronization constraint. For centralized synchronization mechanisms such as monitors, though, the implementation is not too difficult, as will be demonstrated once the monitor construct itself has been introduced in the next section.

5.2 Monitors

The *monitor* is a synchronization mechanism that was first described by Brinch Hansen in [Bri73] and defined more formally by Hoare in [Hoa74]. It grew out of the "secretary" concept proposed by Dijkstra in [Dij72b]. A monitor is an extension of the *class* construct of Simula [Dah72], with one important difference. A monitor, like a Simula class, consists of some local data and a collection of procedures for manipulating that data. The major difference is that executions of the procedures of a monitor are mutually exclusive, in order to protect the integrity of the local data. Processes attempting concurrent executions of a monitor's procedures must wait to gain exclusive access to the monitor. This waiting is defined by Hoare to be fair, and can be assumed to follow a first-come-first-served discipline.

Monitors also contain features for explicit process synchronization. As defined by Hoare, this takes the form of a *condition* data type, which represents a FIFO queue of waiting processes. Two operations are defined on a condition for queuing and dequeuing processes: "wait", which causes the process executing the operation to enter the queue; and "signal", which dequeues the process at the head of the queue; if any. Both operations cause the process executing the operation to relinquish possession of the monitor. A process on a queue that is dequeued via a "signal" operation by some other process regains possession of the monitor. It resumes execution of the monitor procedure it was executing at the point immediately following the "wait" operation that it performed. An additional operation "queue" returns a boolean value, indicating whether any processes are on the queue.

The notation used here will be based on the language CLU [Lis76] rather than the Simula-based notation introduced by Hoare. Thus, a "wait" operation on condition variable c is written

condition$wait(c);

rather than

c.wait;

as in [Hoa74].

Hoare advocates associating informally with each condition variable a boolean predicate on the local data of the monitor. This predicate indicates what condition on the monitor state a process on the queue is awaiting. Making this association aids in proving properties of monitors. As indicated in the next section, this association makes condition variables suitable for representing the entry conditions in the solution specification being implemented.

5.3 The basic monitor implementation

A comparison of monitors with the solution specification structure discussed in Chapter 3 reveals a close correspondence between features of one and the other. The local data of a monitor is sufficient for representing the state information required in a solution specification, since this state information can be represented by a collection of integer-valued quantities. A condition variable in a monitor is a FIFO queue of waiting processes, and as advocated by Hoare, has associated with it informally a boolean predicate on the monitor data. These are exactly the features required for conditions associated with enter gates in a solution specification. Passage through a set of gates associated with a given event class must be indivisible and produce a state change in the system. Monitor procedures are ideal for implementing gates, in that they manipulate the local data of the monitor, and because the enforced mutual exclusion on their executions makes them indivisible operations. Monitor procedures can take parameters, which is important since the behavior of gates sometimes depends on the arguments to the associated procedure activation.

It should be emphasized here that the monitor is being used to implement only the synchronization code, not the abstract data type as a whole. The monitor was originally conceived in [Hoa74] to implement a shared data abstraction itself. Criticism of the monitor construct has appeared in some recent technical literature (e.g. [Hew77], [Had77], [Jam77]). The basis of this criticism has been that the use of monitors to implement abstract data types leads to such problems as reduced concurrency, lack of modularity, and a potential for deadlock through hierarchical monitor calling. As used here, however, the monitor is employed *within* a data abstraction, for the sole purpose of implementing the synchronization code required by the operations of the abstraction. The monitor procedures are kept small in size, so that their use does not significantly affect the degree of concurrency possible. Modularity is enhanced by implementing the synchronization code separately from the abstract data operations. Since lower-level abstractions are called from the bodies of the operations, not from the synchronization code, the problem of hierarchical use of monitors is avoided. (See [Blo78] for the advocacy of a similar discipline in the use of monitors.)

The monitor for a data type contains three procedures for each operation p of the type. These procedures represent the three event classes associated with p, and are named p_request, p_enter, and p_exit. It is necessary that the procedures of the derived monitor be called at the proper points within the data abstraction operations, in order to ensure that the monitor is used properly and the synchronization constraint is embodied in the data abstraction. The form that operation p must take is illustrated below in Figure 5.1. The identifier "m" is the name of the constructed monitor, and v is the vector of parameters to operation p. This vector of parameters actually must be passed to the monitor procedures only for implementations involving qualified gates, as explained in Section 5.5.

The monitor implementation of a "basic" solution specification that involves neither previous state information nor qualified gates is straightforward. Recall that the abstract program for an activation of operation p of the data abstraction in such cases is given by:

Figure 5.1. Monitor calls within operation p

```
p = proc ... ;
      call m.p_request(v);
      call m.p_enter(v);
        .
        . (body of p)
        .
      call m.p_exit(v);
  end p;
```

```
$p^{request}$: increment count($p^{request}$) by 1
$p^{enter}$: wait until entry condition is satisfied,
         then increment count($p^{enter}$) by 1
execute body of operation p
$p^{exit}$: increment count($p^{exit}$) by 1
```

For each quantity of the form count(ec) that appears in one or more entry conditions in the solution specification, there is a corresponding variable of type *integer* in the monitor. This variable is initialized to 0, and is incremented by 1 in the procedure that represents event class ec.

An alternative implementation could employ instead a separate variable for each quantity of the form (count(ec_1) - count(ec_2)), since a condition almost always concerns the difference between two counts. The implementation chosen here is somewhat simpler for purposes of explanation. It does, however, incur the possibility of integer overflow, since each variable is constantly increasing over time. Although techniques can be used to avoid overflow by dynamically extending the precision of integers, the alternative might be preferable in practice.

For each enter gate with an entry condition in the solution specification, the monitor contains a condition variable. The boolean predicate informally associated with this variable is exactly the same as the entry condition, with each quantity of the form count(ec) replaced by the corresponding variable. Let the condition variable corresponding to gate p^{enter} be pentry, and denote the predicate associated with it as C_p. Then the first statement in procedure p_enter is

```
    if (¬ $C_p$) then condition$wait(pentry); end;
```

Whenever control of the monitor is relinquished, it is necessary to check the predicates associated with all condition variables on which processes are queued. If one or more of these predicates are satisfied, then a "signal" operation is performed on one of the conditions. The condition to be signalled must be chosen in a fair manner, so that no process starves because the condition on which it is queued is never chosen for signalling. This can be accomplished by using a variation of Dijkstra's "guarded commands" [Dij75] to implement a new kind of statement called a "choice" statement. Changing Dijkstra's notation so as to distinguish choice statements from ordinary *if* statements, a choice statement looks like:

```
choose
    $B_1$: $s_1$;
    $B_2$: $s_2$;
    . . .
    $B_n$: $s_n$;
end;
```

where the number of guarded commands $n \geq 1$. The meaning of this statement is the following: The "guards" B_i are simply boolean expressions. If one or more of these guards are true, then one of the true guards B_j is (non-determinately) selected and the corresponding statement s_j is executed. There are two important differences between this statement and Dijkstra's version. The method for making the selection between several true guards is unspecified but must be fair. Also, if none of the guards is true, then the statement is simply skipped.

If the condition variables in the monitor are pentry, qentry, etc. with corresponding boolean predicates C_p, C_q, etc., then the following choice statement must appear at the end of every monitor procedure:

```
choose
        condition$queue(pentry) ∧ C_p: condition$signal(pentry);
        condition$queue(qentry) ∧ C_q: condition$signal(qentry);
                    . . .
end;
```

This ensures that whenever one or more waiting processes can be dequeued, due to the satisfaction of the predicates on which they are waiting, one of them will in fact be dequeued. The fact that the predicates in the guards include the conjunct of the form *condition*$queue(pentry) ensures that the condition that is signalled does in fact have a waiting process. As long as the selection is made fairly, the monitor will be a faithful implementation of the solution specification.

A property of the monitor construct that is used here is that a process that is dequeued from a condition variable via a "signal" operation gains possession of the monitor ahead of any process that is attempting to call a monitor procedure. This ensures that a process that has been waiting for an entry condition to become satisfied is allowed to proceed as soon as the condition is in fact satisfied, and is not overtaken by a later-arriving process. This property is necessary for the faithful implementation of the FIFO scheduling that is part of the solution specification structure.

In practice, it is often possible to optimize the signalling statement by eliminating some of the options in the choice statement. The basis for such eliminations is that the corresponding guards cannot possibly be satisfied at the given point in the monitor, due to the rest of the monitor code. In fact, for many simple examples, at most one guard in the *choose* statement can ever be true at any given point. However, in general the analysis required to perform this optimization is difficult. Rather than becoming involved in the details of when a given option can or cannot be eliminated, the simple-minded implementation of always testing all conditions will be used here. (In practice, it might be simpler to make a separate procedure internal to the monitor for this signalling code. Each of the regular monitor procedures could then call on this internal procedure.)

An optimization that can be made easily is the elimination of unnecessary monitor procedures. If no reference is ever made to the quantity **count**($p^{request}$) or **count**(p^{exit}), then the body of the corresponding procedure is empty. The procedure itself, along with the call to it within the data abstraction operation p, then can be eliminated. Similarly, if there is no entry condition associated with gate p^{enter}, and no reference to the quantity **count**(p^{enter}), then procedure p_enter can be eliminated.

As a concrete example of a monitor implementation, consider the following specification:

$$((p_i^{request} \Rightarrow q_j^{enter}) \supset (p_i^{enter} \Rightarrow q_j^{enter})) \wedge$$

$$((q_j^{enter} \Rightarrow p_i^{enter}) \supset (q_j^{exit} \Rightarrow p_i^{enter})).$$

There are two clauses, one giving operation p priority over operation q, the other excluding new activations of operation p during active executions of operation q. The solution specification for this example consists of the conditions

For gate q^{enter}: $count(p^{request}) = count(p^{enter})$

For gate p^{enter}: $count(q^{enter}) = count(q^{exit})$

The monitor implementation of this solution specification contains four integer variables, representing $count(p^{request})$, $count(p^{enter})$, $count(q^{enter})$ and $count(q^{exit})$. These variables are named pr, pn, qn, and qx, respectively. Each variable must be initialized to 0, and incremented by 1 in the corresponding monitor procedure. There are two condition variables, pentry and qentry, for the entry conditions on gates p^{enter} and q^{enter}. The predicates associated with these condition variables are the analogues in terms of monitor variables to the solution specification entry conditions: (pr = pn) for qentry, and (qn = qx) for pentry. The monitor "ex" that is obtained for this example appears in Figure 5.2. The monitor procedures p_exit and q_request have been eliminated as unnecessary. Operation p of the data abstraction must call monitor procedures ex.p_request and ex.p_enter (in that order) before executing its body, while operation q must call procedure ex.q_enter before executing its body, and ex.q_exit afterwards.

Figure 5.2. Monitor for example

```
ex = monitor;
    pr, pn, qn, qx: integer;
    pentry, qentry: condition;

    p_request = procedure;
        pr := pr + 1;
        choose
            condition$queue(pentry)  ∧  qn = qx: condition$signal(pentry);
            condition$queue(qentry)  ∧  pr = pn: condition$signal(qentry);
        end;
    end p_request;

    p_enter = procedure;
        if qn ≠ qx then condition$wait(pentry); end;
        pn := pn + 1;
        choose
            condition$queue(pentry)  ∧  qn = qx: condition$signal(pentry);
            condition$queue(qentry)  ∧  pr = pn: condition$signal(qentry);
        end;
    end p_enter;

    q_enter = procedure;
        if pr ≠ pn then condition$wait(qentry); end;
        qn := qn + 1;
        choose
            condition$queue(pentry)  ∧  qn = qx: condition$signal(pentry);
            condition$queue(qentry)  ∧  pr = pn: condition$signal(qentry);
        end;
    end q_enter;

    q_exit = procedure;
        qx := qx + 1;
        choose
            condition$queue(pentry)  ∧  qn = qx: condition$signal(pentry);
            condition$queue(qentry)  ∧  pr = pn: condition$signal(qentry);
        end;
    end q_exit;

    pr, pn, qn, qx := 0, 0, 0, 0;
end ex;
```

5.4 Previous state information

When a solution specification contains references to quantities not only in the current state but also in previous states, these quantities must be maintained in the monitor in a different manner. Specifically, a separate monitor variable is required for each quantity of the form "[count(ec) $\bullet$ g]", where g is some gate. This variable of type *integer* saves the current value of the variable representing count(ec) in the monitor procedure corresponding to gate g. That is, it is set in the procedure representing gate g by assigning to it the current value of the variable representing count(ec). It can be used in the boolean predicates associated with condition variables in the same way as a variable that represents a quantity in the current state.

Consider example 7 from Chapter 2, the specification for "operation pairing":

$$(a_i^{enter} \Longrightarrow b_j^{enter}) \leftrightarrow (c_i^{enter} \Longrightarrow d_j^{enter}).$$

The derivation of the solution specification for this example was started in Section 4.2. The overall solution specification is:

For gate a^{enter}:

$$(count(b^{enter}) \geq count(d^{enter})) \lor ([count(a^{enter}) \bullet d^{enter}] < [count(c^{enter}) \bullet d^{enter}])$$

For gate b^{enter}:

$$(count(a^{enter}) \geq count(c^{enter})) \lor ([count(b^{enter}) \bullet c^{enter}] < [count(d^{enter}) \bullet c^{enter}])$$

For gate c^{enter}:

$$(count(d^{enter}) \geq count(b^{enter})) \lor ([count(c^{enter}) \bullet b^{enter}] < [count(a^{enter}) \bullet b^{enter}])$$

For gate d^{enter}:

$$(\mathrm{count}(c^{enter}) \geq \mathrm{count}(a^{enter})) \vee ([\mathrm{count}(d^{enter}) @ a^{enter}]) < [\mathrm{count}(b^{enter}) @ a^{enter}])$$

Since the entry conditions do not involve any request or exit gates, only four procedures are needed in the monitor, one for each enter event class. Each of the operations a, b, c, and d must call the appropriate monitor procedure prior to executing its body. The variables an, bn, cn, and dn can be used to represent the current counts of the four enter event classes. In addition, eight other variables are needed to save the values of counts in previous states. Variable amrd, for example, represents the count of gate a^{enter} saved at the most recent d^{enter} event. This variable is set in monitor procedure d_enter to the value of an, which represents the current value of count(a^{enter}). Similarly, variable cmrd represents the count of gate c^{enter} saved at the most recent d^{enter} event. The predicate for the condition variable aentry on which procedure a_enter performs a wait operation is:

$$bn \geq dn \;\vee\; amrd < cmrd.$$

The predicates for the other condition variables bentry, centry, and dentry are analogous. The complete monitor appears in Figure 5.3.

5.5 Qualified gates

The remaining issue to be handled is the implementation of qualified gates, which arise in a solution specification from the presence in the problem specification of predicates on the arguments to procedure activations. Recall from Chapter 3 the abstract program for an activation of operation p in a situation involving qualified gates:

Figure 5.3. Monitor for operation pairing example

```
pairs = monitor;
    an, bn, cn, dn: integer;
    amrd, cmrd, bmrc, dmrc: integer;
    amrb, cmrb, bmra, dmra: integer;
    aentry, bentry, centry, dentry: condition;

    a_enter = procedure;
        if (bn < dn ∧ amrd ≥ cmrd) then condition$wait(aentry); end;
        an := an + 1;
        bmra := bn;
        dmra := dn;
        choose
          condition$queue(aentry) ∧ (bn ≥ dn ∨ amrd < cmrd): condition$signal(aentry);
          condition$queue(bentry) ∧ (an ≥ cn ∨ bmrc < dmrc): condition$signal(bentry);
          condition$queue(centry) ∧ (dn ≥ bn ∨ cmrb < amrb): condition$signal(centry);
          condition$queue(dentry) ∧ (cn ≥ an ∨ dmra < bmra): condition$signal(dentry);
        end;
    end a_enter;

    b_enter = procedure;
        if (an < cn ∧ bmrc ≥ dmrc) then condition$wait(bentry); end;
        bn := bn + 1;
        amrb := an;
        cmrb := cn;
        choose
          condition$queue(aentry) ∧ (bn ≥ dn ∨ amrd < cmrd): condition$signal(aentry);
          condition$queue(bentry) ∧ (an ≥ cn ∨ bmrc < dmrc): condition$signal(bentry);
          condition$queue(centry) ∧ (dn ≥ bn ∨ cmrb < amrb): condition$signal(centry);
          condition$queue(dentry) ∧ (cn ≥ an ∨ dmra < bmra): condition$signal(dentry);
        end;
    end b_enter;

    c_enter = procedure;
        if (dn < bn ∧ cmrb ≥ amrb) then condition$wait(centry); end;
        cn := cn + 1;
        bmrc := bn;
        dmrc := dn;
        choose
          condition$queue(aentry) ∧ (bn ≥ dn ∨ amrd < cmrd): condition$signal(aentry);
          condition$queue(bentry) ∧ (an ≥ cn ∨ bmrc < dmrc): condition$signal(bentry);
          condition$queue(centry) ∧ (dn ≥ bn ∨ cmrb < amrb): condition$signal(centry);
          condition$queue(dentry) ∧ (cn ≥ an ∨ dmra < bmra): condition$signal(dentry);
        end;
    end c_enter;

    d_enter = procedure;
        if (cn < an ∧ dmra ≥ bmra) then condition$wait(dentry); end;
        dn := dn + 1;
        amrd := an;
        cmrd := cn;
        choose
          condition$queue(aentry) ∧ (bn ≥ dn ∨ amrd < cmrd): condition$signal(aentry);
          condition$queue(bentry) ∧ (an ≥ cn ∨ bmrc < dmrc): condition$signal(bentry);
          condition$queue(centry) ∧ (dn ≥ bn ∨ cmrb < amrb): condition$signal(centry);
          condition$queue(dentry) ∧ (cn ≥ an ∨ dmra < bmra): condition$signal(dentry);
        end;
    end d_enter;

    an, bn, cn, dn := 0, 0, 0, 0;
    amrd, cmrd, bmrc, dmrc := 0, 0, 0, 0;
    amrb, cmrb, bmra, dmra := 0, 0, 0, 0;
end pairs;
```

```
$p^{request}$: in parallel for all gates g in event class $p^{request}$,
        if v satisfies the qualifying predicate of g,
            then increment count(g) by 1
$p^{enter}$: in parallel for all gates g in event class $p^{enter}$,
        if v satisfies the qualifying predicate of g,
            then wait until the entry condition of g is satisfied,
            and then increment count(g) by 1
execute body of operation p
$p^{exit}$: in parallel for all gates g in event class $p^{exit}$,
        if v satisfies the qualifying predicate of g,
            then increment count($p^{exit}$) by 1
```

How this abstract program is implemented in a monitor depends to some extent upon the nature of the qualifying predicates. In all cases, though, it is necessary that each of the monitor procedures p_request, p_enter, and p_exit take the same vector of arguments as the data abstraction operation p itself does. This allows the monitor procedures to test the qualifying predicates on the arguments, thereby determining which gates apply to an operation activation. Each monitor procedure implements the entire set of gates for the given event class.

Qualified request and exit gates are easier to implement than qualified enter gates. Since these gates consist only of incrementing integer variables, it is merely necessary to test the qualifying predicate before incrementing. The simplest case involves a predicate concerning only the arguments to the associated data type operation. A qualified count, like an unqualified one, is represented by an integer variable initialized to 0. The update to this variable is preceded by a test of the qualifying condition, and is only made if the condition is true. For example, let the qualifying predicate be Q(v), i.e. the quantity to be updated is something like count([p(v) | Q(v)]^{exit}). If x is the monitor variable representing this qualified count, then the update statement in procedure p_exit is

```
if Q(v) then x:=x+1; end;
```

There may be more than one qualified gate for an event class, in which case a separate state variable is required for each gate. The update of each state variable x_i must be preceded by a test of its corresponding condition Q_i. Because more than one of these conditions may be simultaneously satisfied, it is important that the tests be made in a series of statements of the form

```
if $Q_i(v)$ then $x_i := x_i + 1$ end;
```

rather than in one statement such as

```
if $Q_1(v)$ then $x_1 := x_1 + 1$
elseif $Q_2(v)$ then $x_2 := x_2 + 1$
elseif ... end;
```

that could only increment one variable at most.

A qualifying predicate may be parameterized, and so involve not only the arguments to the associated operation, but also a parameterizing variable t. (There actually may be several parameterizing variables t_i, but they can be combined into one composite variable t = $<t_1, ..., t_n>$.) For each possible value of t there is conceptually a separate gate, which means there must be a separate quantity in the state. For example, suppose that a solution specification contains a quantity of the form count([p(v) | Q(v, t)]^{exit}). If the parameterizing variable t were of type *integer* and could only take values from a restricted range, say 1 to

100, then this quantity could be implemented by an array with that subscript range. The n-th element in the array would represent the quantity $count([p(v) \mid Q(v, n)]^{exit})$.

In general, of course, variable t is not necessarily an integer, and it is impossible to know ahead of time all possible values for t. The same idea can be used in the implementation, though, by employing an abstraction that captures this same effect. The parameterized type "counts[T]", where T represents the type of t, contains counts for all possible values of t, at least conceptually. These counts are all set to 0 initially when a new object of type counts[T] is created, and the count corresponding to a particular value t_0 is incremented by the operation "incr" with t_0 as argument.

In the actual implementation of the type counts[T], the count for any particular value of t is created and added to the object of type counts[T] only as it becomes needed. The implementation of this type in a language with dynamic arrays such as CLU is straightforward. However, the dynamic creation of counts as they are needed is an implementation detail; users of the type can ignore this and use the abstract conception of all counts that are needed being created as part of the object initially.

For the purpose of translating a solution specification into a monitor, each state variable representing a qualified count whose predicate is parameterized by variable t must be implemented by an object of type counts[T]. A "create" operation for this object is required in the initialization code of the monitor. The qualifying predicate must take the form of a functional relation between variable t and the arguments of the procedure activation, i.e. the qualified quantity must be of the form $count([p(v) \mid t = f(v)]^{exit})$. It is always possible to parameterize a predicate so that at most one activation is non-functionally related to the parameterizing variable t. This activation should be chosen to be the one for whose enter event the condition is derived. This means that only one value of t can apply to any given activation, and so only one count needs to be incremented. Incrementing the proper count is accomplished by the statement

$$\text{counts[T]\$incr(cou, f(v))}$$

where T is the type of t and cou refers to the object of type counts[T].

There must also be an operation "get", analogous to the "fetch" operation on arrays, by which the count for any particular value of t can be retrieved. This operation is used within the predicates for conditions associated with parameterized enter gates, as explained below. The quantity

$$\mathrm{count}([p(v) \mid (t = f(v))]^{exit})$$

in a solution specification entry condition is implemented by the operation call

$$\text{counts[T]\$get(pexitcounts, f(v)),}$$

where the object referred to by variable pexitcounts represents $\mathrm{count}([p(v) \mid (t = f(v))]^{exit})$.

Qualified enter gates are more complicated to implement than other types of gates. Not only must a quantity of the form $\mathrm{count}([p(v) \mid Q(v)]^{enter})$ be updated, but first some entry condition must be satisfied, which means that waiting must be implemented. The simplest case is when there is a single qualified gate for the event class, and where the qualifying predicate is only on the parameters to operation p. Then there is a single condition variable "cond", just as for an unqualified enter gate. The wait operation on "cond" is preceded by a test of the qualifying predicate Q(v) as well as the associated predicate C:

if Q(v) $\wedge$ ($\neg$ C) *then condition*$wait(cond); *end*;

When an enter event class contains more than one gate, each with a different qualifying predicate and entry condition, then there must be a separate condition variable for each possible subset of gates whose qualifying predicates may be satisfied by an activation. The boolean predicate associated with each condition variable consists of the conjunction of the entry conditions on all gates in the subset of gates to which the condition variable corresponds. An unqualified gate, of course, applies to every activation, so if there is an unqualified gate, its condition must be part of every predicate. In cases where two qualifying predicates are contradictory, or where one implies another, some subsets of gates will be impossible and can be eliminated from consideration.

For example, assume a solution specification contains the following entry conditions:

For gate p^{enter}: count(a^{enter}) = count(b^{enter})

For gate $[p(v) \mid Q1(v)]^{enter}$: count($a^{request}$) = count($a^{enter}$)

For gate $[p(v) \mid Q2(v)]^{enter}$: count($b^{exit}$) = count($c^{enter}$)

Assuming that predicates Q1 and Q2 are not contradictory, and that neither one implies the other, then there must be four separate condition variables. These must cover the activations satisfying neither Q1 nor Q2, both Q1 and Q2, and either one but not the other. The unqualified gate applies to all four cases, of course. Let variables ar, an, bn, bx, and cn represent the quantities count($a^{request}$), count(a^{enter}), count(b^{enter}), count(b^{exit}), and count(c^{enter}), respectively. The predicates associated with the condition variables are then

c0: an = bn

c1: $an = bn \wedge ar = an$

c2: $an = bn \wedge bx = cn$

c3: $an = bn \wedge ar = an \wedge bx = cn$

The code involving these variables at the beginning of monitor procedure p_enter(v) is:

```
if $Q2(v) \wedge Q1(v) \wedge (an \neq bn \vee ar \neq an \vee bx \neq cn)$
        then condition$wait(c3);
    elseif $Q2(v) \wedge (\neg Q1(v)) \wedge (an \neq bn \vee bx \neq cn)$
        then condition$wait(c2);
    elseif $(\neg Q2(v)) \wedge Q1(v) \wedge (an \neq bn \vee ar \neq an)$
        then condition$wait(c1);
    elseif $(an \neq bn)$
        then condition$wait(c0); end;
```

If Q1 and Q2 are contradictory, then condition c3 may be eliminated, while if one implies the other, then either c1 or c2 is not needed.

A qualifying predicate on an enter gate that involves a parameterizing variable t presents the most difficult implementation problem. Since this construct actually represents a separate gate for each possible value of t, a separate condition variable is needed for each possible value of variable t. To implement this, what is required is something like an array of conditions, but with a dynamic range, so that new conditions can be created and added to it.

The implementation uses a type called "conditions[T]". An object of this type contains an object of type *condition* for each value in its domain. The initial domain of the object returned by the "create" operation is empty. In general, the domain consists of the set of values of t that have been explicitly added by means of the "add" operation. The "add" operation creates a new condition only if one does not yet exist for the given value of t, so that subsequent calls on "add" with the same value of t have no effect. The predicate associated with each condition is parameterized by the associated value of t, and so is of the form $C_p(t)$. Notice that while the exact set of conditions is determined dynamically by the "add" operations, the form of the predicate for each one is fixed, except for the value of the parameterizing variable t.

The first step in implementing parameterized enter gates is to combine all the enter gates in the event class into all possible combinations of satisfiable qualifying predicates. If there is an unqualified gate for the event class, then its entry condition becomes a conjunct of the parameterized condition $C_p(t)$. When there are individual (non-parameterized) qualified gates, then the same analysis as to possible subsets of satisfied gates must be made as was discussed above and illustrated by the example involving predicates Q_1 and Q_2. As before, there must be a condition representing each possible subset of gates through which a given procedure activation may pass. Since there are parameterized gates, this requires a separate object of type conditions[T] for each combination of gates including a parameterized gate. The remaining discussion focuses on a single object of type conditions[T], but notes how to generalize to cases involving many such objects.

Given an enter gate qualified by some predicate parameterized by variable t, the relation R between variable t and parameter vector v of the operation being qualified may or may not be a function. If it is a function, the qualifying predicate takes the form t = f(v). The condition on which to possibly wait is then found by calling the "get" operation on the object conds of type conditions[T] with argument f(v). The "get" operation, similar to the "get" operation on counts[T], retrieves the condition corresponding to the value of its second argument. The code for waiting in the monitor "enter" procedure is therefore:

if (¬ C_p(f(v))) *then condition*$wait(conditions[T]$get(conds, f(v))); *end*;

To guarantee that the object conds does in fact contain a condition for the associated value of t, the statement

conditions[T]$add(conds, f(v));

is used to add the appropriate condition to the set of conditions in object conds if it is not already there. This operation must precede the waiting statement, a fact that can be ensured by placing it at the beginning of the "enter" procedure.

An optimization that is possible is to only add the condition to conds if a "wait" is actually performed. That is, instead of locating the "add" operation at the beginning of the monitor procedure, instead it can be placed inside the *then* clause of the *if* statement immediately preceding the "wait".. In addition, after the process finishes waiting on the condition, i.e. after being signalled, the condition created may no longer be needed. If no other processes are waiting on the condition, then it could be deleted from object conds. These .optimizations would increase efficiency by keeping the size of conds as small as possible. However, they will not be performed in the examples here.

Note that in certain situations, the range of possible values of the parameterizing variable t may be quite limited. If this is so, then it might be more efficient to add all possible values to the domain of conds initially, and eliminate the need for adding (and deleting) new conditions dynamically. This optimization, however, relies on extra information that is not contained in the specification but would have to be supplied in addition by the specifier. In an actual system, this might be accomplished by having the system interact with the user to find out about the range of values of a given parameter.

In order to signal the conditions contained in an object of type conditions[T], the type must have an iterator "domain" for accessing one by one (in an unspecified order) all values of t for which conditions exist. By iterating through these values, all conditions on which processes may be waiting are tested. The code for signalling that appears at the end of each monitor procedure is:

```
for t:T in conditions[T]$domain(conds) do
    if condition$queue(conditions[T]$get(conds, t)) $\wedge$ $C_p(t)$
        then condition$signal(conditions[T]$get(conds, t)); end;
end;
```

This serves to signal a process on any of the condition queues in conds whose predicates are true. Where there are several different objects of type conditions[T], due to different combinations of gates with satisfiable qualifying predicates, then this must be generalized so that the conditions contained in all of them are tested and signalled. (Notice that if the optimization mentioned earlier of deleting unneeded conditions were applied, then the implementation of the "domain" iterator would have to function correctly in a situation in which conditions could be deleted while the iterator was suspended due to a "signal" operation.)

For an example to illustrate the above discussion, suppose that the solution specification consists of the following parameterized entry condition:

For gate $[p(y) \mid (y + 1 = t)]^{enter}$:

$$count([q(x) \mid (x = t)]^{enter}) = count([q(x) \mid (x = t)]^{exit})$$

where variable t is an integer. Then the parameterized quantities $\mathrm{count}([q(x) \mid (x = t)]^{enter})$ and $\mathrm{count}([q(x) \mid (x = t)]^{exit})$ would be implemented by objects of type counts[*integer*], as explained previously, named qentercounts and qexitcounts. They are created conceptually containing counts for all possible values of t, with all counts initialized to 0, and are updated by "incr" operations in procedures q_enter and q_exit, respectively.

An object pentry of type conditions[*integer*] can be used to hold the conditions required. The predicate corresponding to the condition for any t_0 is given by

$$\text{counts}[\textit{integer}]\$\text{get}(\text{qentercounts}, t_0) = \text{counts}[\textit{integer}]\$\text{get}(\text{qexitcounts}, t_0)$$

Conditions are added to pentry by the statement

```
conditions[integer]$add(pentry, y+1)
```

appearing at the start of monitor procedure p_enter, which takes the same parameter y as operation p. The waiting in procedure p_enter then is accomplished by a wait on the appropriate condition, retrieved via a "get" operation:

```
if counts[integer]$get(qentercounts, y+1) ≠ counts[integer]$get(qexitcounts, y+1)
    then condition$wait(conditions[integer]$get(pentry, y+1); end;
```

The signalling code at the end of each monitor procedure is:

```
for t:integer in conditions[integer]$domain(pentry) do
    if condition$queue(conditions[integer]$get(pentry, t)) ∧
        counts[integer]$get(qentercounts, t) = counts[integer]$get(qexitcounts, t)
          then condition$signal(conditions[integer]$get(pentry, t)); end;
end;
```

The overall monitor for this example appears in Figure 5.4.

Figure 5.4. **Monitor for functional parameterized example**

```
parafun = monitor;

    qentercounts, qexitcounts: counts[integer];
    pentry: conditions[integer];

    q_enter = procedure(x:integer);
        counts[integer]$incr(qentercounts, x);
        for t:integer in conditions[integer]$domain(pentry) do
            if condition$queue(conditions[integer]$get(pentry, t)) ∧
                counts[integer]$get(qentercounts, t) = counts[integer]$get(qexitcounts, t)
                  then condition$signal(conditions[integer]$get(pentry, t)); end;
        end;
    end q_enter;

    q_exit = procedure(x:integer);
        counts[integer]$incr(qexitcounts, x);
        for t:integer in conditions[integer]$domain(pentry) do
            if condition$queue(conditions[T]$get(pentry, t)) ∧
                counts[integer]$get(qentercounts, t) = counts[integer]$get(qexitcounts, t)
                  then condition$signal(conditions[integer]$get(pentry, t)); end;
        end;
    end q_exit;

    p_enter = procedure(y:integer);
        conditions[integer]$add(pentry, y+1);
        if counts[integer]$get(qentercounts, y+1) ≠ counts[integer]$get(qexitcounts, y+1)
            then condition$wait(conditions[integer]$get(pentry, y+1); end;
        for t:integer in conditions[integer]$domain(pentry) do
            if condition$queue(conditions[integer]$get(pentry, t)) ∧
                counts[integer]$get(qentercounts, t) = counts[integer]$get(qexitcounts, t)
                  then condition$signal(conditions[integer]$get(pentry, t)); end;
        end;
    end p_enter;

    qentercounts := counts[integer]$create();
    qexitcounts := counts[integer]$create();
    pentry := conditions[integer]$create();
end parafun;
```

If the relation R(t, v) that qualifies an enter gate is not functional, then the enter event must wait until the entry condition represented by predicate $C_p(t)$ is satisfied for all values of t such that R(t, v) is true. That is, the entry condition for an activation of p with argument vector v is given by the formula

$$\forall t\, (R(t, v) \supset C_p(t)).$$

This is considerably more complex than the entry condition $C_p(f(v))$ for the case where the relation between t and v was of the functional form t = f(v), as discussed above.

As discussed above in connection with parameterized qualifying predicates that are functions, information about the range of possible values of the parameters could be used to optimize the implementation. Such information would make a much greater difference here where the predicate is a nonfunctional relation. In the absence of such information, which would have to be supplied by the user in addition to the specification, the implementation to be presented here must work under the assumption that the range of possible values of each parameter is infinite. The result is a severe penalty in both complexity and efficiency. It will be noted where user-supplied range information could be used to simplify and optimize the implementation.

An assumption is made here that the predicate $C_p(t)$ is initially true for all values of t. That is, it is assumed to be something like

$$\text{count}([q(x) \mid (x = t)]^{\text{enter}}) = \text{count}([q(x) \mid (x = t)]^{\text{exit}}),$$

rather than

$$\text{count}([q(x) \mid (x = t)]^{\text{enter}}) > \text{count}([q(x) \mid (x = t)]^{\text{exit}}).$$

If this were not the case, and assuming there are an infinite number of possible values of t satisfying predicate $R(t, v)$, then the entry condition

$$\forall t\, (R(t, v) \supset C_p(t))$$

could never be satisfied, since there would always be some values of t (in fact, an infinite number) not satisfying the body of the quantified formula. Here is one example of where information as to the range of possible values of t would be helpful, since in fact t might assume only a small number of possible values. In the absence of an explicit range, however, the range must be assumed to be infinite. Analysis to determine what subset of the range could satisfy the relation R is clearly beyond the scope of this work. The assumption made here appears to be satisfied for all cases of interest, such as the disk head scheduler discussed in Chapter 6, and therefore not to be limiting.

In implementing a solution specification in which an enter gate is qualified with a nonfunctional parameterized predicate, we again use the type conditions[T]. The type T by which this type is parameterized, however, is not the type of the parameterizing variable t, but rather the type of the argument vector v, or more precisely of some sub-vector of v. The specific sub-vector chosen consists of exactly those components of v that are involved in relation R, which can be determined by syntactic inspection of R. The type of this sub-vector will be denoted "vtype".

Because of the solution specification structure, there must be a separate condition variable for each subset of gates that could apply to a given activation. If processes making different activations pass through the same subset of gates, then they must do so in FIFO order. This is implemented by having the processes wait on the same condition, thus ensuring FIFO order. In general, two activations $p(v_1)$ and $p(v_2)$ wait on the same subset of gates when

$$\forall t\,(R(t, v_1) \leftrightarrow R(t, v_2)).$$

Ideally, this formula should determine whether two activations wait on the same condition variable. However, the logical power necessary to perform this analysis in general is beyond the scope of this thesis. Here again, information about the range of parameter values could overcome the problem.

The implementation therefore makes a simplifying assumption, which is that two activations of an operation pass through the same set of gates only if the sub-vector of components involved in relation R are equal for the two activations. When the argument vectors to different activations share the sub-vector to which R refers, though possibly differing in other components, then they must pass through the exact same set of gates. This means that in the implementation they must wait on the same condition. For this reason, there is one condition for each value of the sub-vector of arguments involved in relation R. What is assumed here is that two activations with different sub-vectors always pass through different, through possibly overlapping, subsets of gates, so that in the implementation they can wait on different conditions. This assumption is true for the disk head scheduler of Chapter 6, for instance, and where the relation R is something like

$$t < x,$$

where x is one of the arguments in v. This is because if two values of x are unequal, then there exists some value of t that is less than one but not the other. An example of where the assumption breaks down is if R is of the form

$$t = \text{absolute\_value}(x),$$

since x and (-x) satisfy this relation for the exact same set of values of t.

The object conds of type conditions[vtype] is different from the corresponding object of type conditions[T] in the case of a functional relation. As before, the object is created initially empty, and conditions are added to it dynamically in the monitor "enter" procedure prior to the code for waiting. However, since the predicate associated with each condition in conds is of the form

$$\forall t\,(R(t, v) \supset C_p(t)),$$

it is necessary also to maintain a record of those values of the parameterizing variable t that have occurred, since these are the values for which $C_p(t)$, which by assumption is initially true, may have become false. This is accomplished by saving the set of all relevant values of t in an object "tset" of type set[T] (where T is again the type of t). The object tset is initially created as the empty set. Elements are added to the set by the "insert" operation. An "insert" operation must be performed in each monitor procedure in which quantities involved in the predicate $C_p(t)$ are updated. There is also an iterator "elements" for accessing the elements of the set.

As was the case mentioned earlier for type conditions[T], information from the user as to the range of possible values of t would permit an optimization to be performed with respect to the object tset. If the range is relatively small, then all relevant values can be inserted into the set beforehand. This would eliminate the need to dynamically insert values. Note that another optimization mentioned in connection with conditions[T], that of deleting elements when no longer needed, cannot be applied to tset, since any value of t that has occurred may be relevant and must therefore be saved.

The code in procedure p_enter(v) for testing and waiting on the condition in object conds is given by:

```
for t:T in set[T]$elements(tset) do
    if $R(t, v) \land (\neg C_p(t))$ then
        condition$wait(conditions[vtype]$get(conds, v)); end;
end;
```

This code implements waiting on the entry condition

$$\forall t\,(R(t, v) \supset C_p(t)).$$

The required condition is added to conds by the statement

```
conditions[vtype]$add(pentry, v);
```

at the start of the monitor procedure.

Notice that the "elements" iterator may be suspended in the middle of execution due to the execution of a "wait". While it is suspended, new values of t may be added to tset by other monitor procedures. The iterator must be implemented so as to function correctly in such a situation.

Signalling at the end of each monitor procedure is complicated. The signalling code must iterate through all values of z (a sub-vector of v) in conds, for each one testing whether its predicate is true by iterating through all values of t in tset. This code involves an iterative loop within an iterative loop, with a "signal" operation performed at the completion of the inner loop if all values of t for which R(t, z) are true satisfy the predicate $C_p(t)$. (We take the liberty of saying "R(t, z)" rather than "R(t, v)", since z contains all the components of v that are involved in R.) The code is of the form:

```
for z:vtype in conditions[vtype]$domain(conds) do
    if condition$queue(conditions[vtype]$get(conds, z)) then
        ok:boolean := true;
        for t:T in set[T]$elements(tset) do
            if R(t, z) ∧ (¬ C_p(t)) then
                ok := false; end;
        end;
        if ok then condition$signal(conditions[vtype]$get(conds, z)); end;
    end;
end;
```

The boolean variable ok keeps track of whether the predicate $C_p(t)$ is true for all values of t for which R(t, z) is satisfied. If ok is still true after the end of the inner loop, then

$$\forall\, t\, (R(t, z) \supset C_p(t)),$$

is true for the given value of z, and therefore the condition should be signalled. Notice that if there is a process waiting on the condition queue for z, there must be at least one value of t for which R(t, z) is true, because otherwise there would have been no reason for the process to have performed a "wait". As before, in a situation in which there is more than one object of type conditions[T], the conditions in each such object must be tested and signalled by code of the above form.

As an example, consider a solution specification consisting of the condition:

For gate $[p(y) \mid (y < t)]^{enter}$:

$$count([q(x) \mid (x = t)]^{enter}) = count([q(x) \mid (x = t)]^{exit})$$

where variable t is an integer. Then as in the previous example, $count([q(x) \mid (x = t)]^{enter})$ and $count([q(x) \mid (x = t)]^{exit})$ are implemented by objects of type counts[*integer*], named qentercounts and qexitcounts, respectively. An object pentry of type conditions[*integer*] is used to hold the conditions required, the single argument y serving as the sub-vector of v involved in relation R. An object tset of type set[*integer*] holds the values of t, to which values are added by the statement

```
set[integer]$insert(pentry, x);
```

in monitor procedures q_enter and q_exit. The code for waiting in procedure p_enter is given by

```
for t:integer in set[integer]$elements(tset) do
    if  y < t  ∧  counts[integer]$get(qentercounts, t) ≠
                              counts[integer]$get(qexitcounts, t)
    then
        condition$wait(conditions[integer]$get(pentry, v)); end;
end;
```

As before, the required condition is added to pentry at the start of the p_enter procedure by the operation

```
conditions[integer]$add(pentry, y);
```

The signalling code at the end of each monitor procedure is:

```
for z:integer in conditions[integer]$domain(pentry) do
    if condition$queue(conditions[integer]$get(pentry, z)) then
        ok:boolean := true;
        for t:integer in set[integer]$elements(tset) do
            if  z < t  ∧  counts[integer]$get(qentercounts, t) ≠
                                counts[integer]$get(qexitcounts, t)
            then ok := false; end;
        end;
        if ok then condition$signal(conditions[integer]$get(pentry, z)); end;
    end;
end;
```

The monitor for this example appears in Figure 5.5.

A number of examples of the translation techniques discussed here appear in Chapter 6. These examples actually illustrate the entire synthesis process, starting with problem specifications of the type described in Chapter 2, proceeding to the construction of equivalent solution specifications via the method presented in Chapter 4, and finally translating these solution specifications into monitors as discussed in this chapter. In particular, the last example of Chapter 6, the "disk head scheduler", illustrates the implementation of qualified gates involving parameterized predicates.

Figure 5.5. Monitor for nonfunctional parameterized example

```
paranon = monitor;

     qentercounts, qexitcounts: counts[integer];
     pentry: conditions[integer];
     tset := set[integer];

     q_enter = procedure(x:integer);
          counts[integer]$incr(qentercounts, x);
          set[integer]$insert(tset, x);
          for z:integer in conditions[integer]$domain(pentry) do
               if condition$queue(conditions[integer]$get(pentry, z)) then
                    ok:boolean := true;
                    for t:integer in set[integer]$elements(tset) do
                         if  z < t  ∧  counts[integer]$get(qentercounts, t) ≠
                                             counts[integer]$get(qexitcounts, t)
                         then ok := false; end;
                    end;
                    if ok then condition$signal(conditions[integer]$get(pentry, z)); end;
               end;
          end;
     end q_enter;

     q_exit = procedure(x:integer);
          counts[integer]$incr(qexitcounts, x);
          set[integer]$insert(tset, x);
          for z:integer in conditions[integer]$domain(pentry) do
               if condition$queue(conditions[integer]$get(pentry, z)) then
                    ok:boolean := true;
                    for t:integer in set[integer]$elements(tset) do
                         if  z < t  ∧  counts[integer]$get(qentercounts, t) ≠
                                             counts[integer]$get(qexitcounts, t)
                         then ok := false; end;
                    end;
                    if ok then condition$signal(conditions[integer]$get(pentry, z)); end;
               end;
          end;
     end q_exit;

     p_enter = procedure(y:integer);
          conditions[integer]$add(pentry, y);
          for t:integer in set[integer]$elements(tset) do
               if  y < t  ∧  counts[integer]$get(qentercounts, t) ≠
                                        counts[integer]$get(qexitcounts, t)
```

```
            then condition$wait(conditions[integer]$get(pentry, v)); end;
        end;
        for z:integer in conditions[integer]$domain(pentry) do
            if condition$queue(conditions[integer]$get(pentry, z)) then
                ok:boolean := true;
                for t:integer in set[integer]$elements(tset) do
                    if  z < t  ∧  counts[integer]$get(qentercounts, t) ≠
                                  counts[integer]$get(qexitcounts, t)
                    then ok := false; end;
                end;
                if ok then condition$signal(conditions[integer]$get(pentry, z)); end;
            end;
        end;
    end p_enter;

    qentercounts := counts[integer]$create();
    qexitcounts := counts[integer]$create();
    pentry := conditions[integer]$create();
    tset := set[integer]$create();
end paranon;
```

Chapter 6

Complete Examples of Synthesis

6.1 Introduction

This chapter presents a series of examples of the complete synthesis method. Each example starts with a problem specification, and derives an equivalent solution specification via the method presented in Chapter 4. This solution specification is then translated into a monitor implementation in the manner outlined in Chapter 5. The examples chosen for this chapter are problems that commonly are addressed in technical literature on synchronization. These are the bounded buffer, two different versions of the readers-writers problem, with writers' priority and alternating priority, and the disk head scheduler.

6.2 Bounded buffer

The first example in this chapter is the specification of example 9 from Section 2.7, the "bounded buffer". The problem specification given in Chapter 2 is repeated here, to be denoted bb:

$$(\text{dep}_i^{\text{exit}} \Rightarrow \text{rem}_i^{\text{enter}}) \wedge (\text{rem}_i^{\text{exit}} \Rightarrow \text{dep}_{i+N}^{\text{enter}}) \wedge$$
$$(\text{dep}_i^{\text{exit}} \Rightarrow \text{dep}_{i+1}^{\text{enter}}) \wedge (\text{rem}_i^{\text{exit}} \Rightarrow \text{rem}_{i+1}^{\text{enter}}).$$

The specification bb consists of four conjuncts, and the solution specification is constructed by analyzing each conjunct separately. Since each individual conjunct is quite simple, the analysis is straightforward. For purposes of reference, the four conjuncts are denoted bb_1, bb_2, bb_3, and bb_4.

The first conjunct to be analyzed is bb_1,

$$(dep_i^{exit} \Rightarrow rem_i^{enter}).$$

This conjunct specifies that the i-th "deposit" activation must finish before the i-th "remove" activation can start. This constraint ensures that no attempt is ever made to remove a message from the buffer before it has been deposited in. Since there are no argument constraints in the conjunct, the first step in the analysis is the identification of which event expressions are mentioned. The set of event expressions in the conjunct is given by

$$Evexp(bb_1) = \{dep_i^{exit}, rem_i^{enter}\}.$$

The next step is to construct the possible orderings among the events represented in the set $Evexp(bb_1)$. With just two such events, only two orderings are possible:

$$(1)\ (dep_i^{exit} \Rightarrow rem_i^{enter})$$

$$(2)\ (rem_i^{enter} \Rightarrow dep_i^{exit})$$

In evaluating whether each is valid or invalid, it is obvious that the first is valid, while the second is not. Equally obvious is the fact the the offending event in ordering (2) must be the first event, namely rem_i^{enter}. This means that a solution specification condition must be derived for the rem^{enter} gate.

Characterizing the state at each event in the rem^{enter} event class, one obtains characterizations c_1 and c_2 for event rem_i^{enter} in orderings (1) and (2), respectively:

$$c_1\!: \ \exists\, i\ (count(dep^{exit}) \geq i\ \wedge\ count(rem^{enter}) < i)$$

$$c_2\!: \ \exists\, i\ (count(dep^{exit}) < i\ \wedge\ count(rem^{enter}) < i)$$

With only one valid ordering, the disjunction of valid ordering characterizations D_v is simply c_1. Similarly, the disjunction of invalid ordering characterizations D_i is c_2. The preliminary condition, given by $(D_V \wedge (\neg D_i))$, then becomes

$$\exists\, i\ (\mathrm{count}(dep^{exit}) \geq i \ \wedge\ \mathrm{count}(rem^{enter}) < i) \ \ \wedge$$

$$\forall\, i\ (\mathrm{count}(dep^{exit}) \geq i \ \vee\ \mathrm{count}(rem^{enter}) \geq i),$$

which reduces to

$$\exists\, i\ (\mathrm{count}(dep^{exit}) \geq i > \mathrm{count}(rem^{enter})).$$

The quantified variable i can be eliminated, resulting in the simplified formula

$$\mathrm{count}(dep^{exit}) > \mathrm{count}(rem^{enter}).$$

When tested, this condition is found to satisfy the single valid ordering, ordering (1), showing it to be the correct condition obtainable from conjunct bb_1.

Each of the other three conjuncts can also be analyzed quite easily. The second conjunct is bb_2,

$$(rem_i^{exit} \Longrightarrow dep_{i+N}^{enter}),$$

This prohibits more than N consecutive "deposit" operations without at least one "remove" operation, preventing overflow of the buffer. The set of event expressions for this conjunct is

$$\mathrm{Evexp}(bb_2) = \{rem_i^{exit}, dep_{i+N}^{enter}\}.$$

The two possible orderings are:

$$(1)\ (rem_i^{exit} \Longrightarrow dep_{i+N}^{enter})$$

$$(2)\ (dep_{i+N}^{enter} \Longrightarrow rem_i^{exit})$$

Of these, the first is valid, while the second is invalid, with the offending event in (2) being dep_{i+N}^{enter}. A condition must be derived for gate dep^{enter}.

The state characterizations for event $\mathrm{dep}_{i+N}^{\mathrm{enter}}$ in orderings (1) and (2), respectively, are given by c_1 and c_2:

$$c_1\text{: } \exists\, i\ (\mathrm{count}(\mathrm{rem}^{\mathrm{exit}}) \geq i \ \wedge\ \mathrm{count}(\mathrm{dep}^{\mathrm{enter}}) < i+N)$$

$$c_2\text{: } \exists\, i\ (\mathrm{count}(\mathrm{rem}^{\mathrm{exit}}) < i \ \wedge\ \mathrm{count}(\mathrm{dep}^{\mathrm{enter}}) < i+N)$$

The preliminary condition, $(D_v \wedge (\neg D_1))$, is equal to $(c_1 \wedge (\neg c_2))$:

$$\exists\, i\ (\mathrm{count}(\mathrm{rem}^{\mathrm{exit}}) \geq i \ \wedge\ \mathrm{count}(\mathrm{dep}^{\mathrm{enter}}) < i+N) \ \ \wedge$$

$$\forall\, i\ (\mathrm{count}(\mathrm{rem}^{\mathrm{exit}}) \geq i \ \vee\ \mathrm{count}(\mathrm{dep}^{\mathrm{enter}}) \geq i+N),$$

which simplifies to

$$\mathrm{count}(\mathrm{rem}^{\mathrm{exit}}) > \mathrm{count}(\mathrm{dep}^{\mathrm{enter}}) - N.$$

This is the correct solution specification condition for conjunct bb_2. Notice that variable N in the above formulas is treated as a constant, since it is the parameter to the abstract data type itself. For this reason, it is not quantified and cannot be eliminated as variable i is.

The last two conjuncts are identical, except that bb_3 applies to operation "dep" and bb_4 to operation "rem". Therefore, whatever condition is obtained from bb_3 for gate $\mathrm{dep}^{\mathrm{enter}}$ applies in corresponding form for $\mathrm{rem}^{\mathrm{enter}}$ due to bb_4. The constraint specified by each is that activations of the given operation must be mutually exclusive and must proceed in first-come-first-served order. This prevents interference by concurrent activations of the same operation manipulating the same local data, and guarantees that messages are deposited and removed in the proper order. For conjunct bb_3,

$$\mathrm{Evexp}(bb_3) = \{\mathrm{dep}_i^{\mathrm{exit}}, \mathrm{dep}_{i+1}^{\mathrm{enter}}\}.$$

The two possible orderings are:

$$(1)\ (\mathrm{dep}_i^{\mathrm{exit}} \Longrightarrow \mathrm{dep}_{i+1}^{\mathrm{enter}})$$

$$(2)\ (dep_{i+1}^{enter} \Rightarrow dep_i^{exit})$$

Ordering (1) is valid, but (2) is not. The offending event in (2) is dep_{i+1}^{enter}, so a condition is required for gate dep^{enter}.

The state characterizations for event dep_{i+1}^{enter} in orderings (1) and (2), respectively, are given by c_1 and c_2:

$$c_1\text{:}\ \exists\, i\ (count(dep^{exit}) \geq i \ \wedge\ count(dep^{enter}) < i+1)$$

$$c_2\text{:}\ \exists\, i\ (count(dep^{exit}) < i \ \wedge\ count(dep^{enter}) < i+1)$$

The preliminary condition, $(D_v \wedge (\neg D_i))$, is given by $(c_1 \wedge (\neg c_2))$:

$$\exists\, i\ (count(dep^{exit}) \geq i \ \wedge\ count(dep^{enter}) < i+1)\ \wedge$$

$$\forall\, i\ (count(dep^{exit}) \geq i \ \vee\ count(dep^{enter}) \geq i+1).$$

This reduces to simply

$$count(dep^{enter}) = count(dep^{exit}),$$

which is the correct solution specification condition for conjunct bb_3. Analogously, the correct condition for bb_4 is

$$count(rem^{enter}) = count(rem^{exit})$$

for gate rem^{enter}.

The overall solution specification for specification bb is constructed by conjoining for each gate the conditions obtained separately from the individual conjuncts. This obtains the following overall conditions:

For gate dep^{enter}:

$$count(rem^{exit}) > count(dep^{enter}) - N \ \wedge\ count(dep^{enter}) = count(dep^{exit})$$

For gate rem^{enter}:

$$count(dep^{exit}) > count(rem^{enter}) \wedge count(rem^{enter}) = count(rem^{exit})$$

The monitor to implement this solution specification must have four integer variables, depn, depx, remn, and remx, to represent the quantities $count(dep^{enter})$, $count(dep^{exit})$, $count(rem^{enter})$, and $count(rem^{exit})$. There also must be two condition variables, depentry and rementry, corresponding to the entry conditions for gates dep^{enter} and rem^{enter}, respectively. The boolean predicates associated with these conditions are

depentry: $remx > depn - N \wedge depn = depx$

rementry: $depx > remn \wedge remn = remx$

Since the request events for the two operations are not used in the specification, there is no need for procedures to implement the corresponding gates. The monitor for the bounded buffer is presented in Figure 6.1. Since the intention is for the monitor to be contained *within* the type module for the abstract type buffer(N), the variable N inside the monitor is bound to the parameter of the type.

6.3 Writers' priority database

The second example in this chapter is a problem that was introduced in [Cou71]. The data abstraction in question is a database, on which two operations are defined: "read" accesses the database without changing it at all, and "write" updates the database. In order to ensure consistent accessing and updating, these two operations must obey the "readers-writers" property embodied in example 3 of Section 2.7. In addition, the scheduling policy desired is for activations of operation "write" to have absolute priority over those of

Figure 6.1. Monitor for bounded buffer

```
bb = monitor;
    depn, depx, remn, remx: integer;
    depentry, rementry: condition;

    dep_enter = procedure;
        if ($remx \le depn - N \lor depn \ne depx$) then condition$wait(depentry); end;
        depn := depn + 1;
        choose
            condition$queue(depentry) $\land remx > depn - N \land depn = depx$:
                    condition$signal(depentry);
            condition$queue(rementry) $\land depx > remn \land remn = remx$:
                    condition$signal(rementry);
        end;
    end dep_enter;

    dep_exit = procedure;
        depx := depx + 1;
        choose
            condition$queue(depentry) $\land remx > depn - N \land depn = depx$:
                    condition$signal(depentry);
            condition$queue(rementry) $\land depx > remn \land remn = remx$:
                    condition$signal(rementry);
        end;
    end dep_exit;

    rem_enter = procedure;
        if ($depx \le remn \lor remn \ne remx$) then condition$wait(rementry); end;
        remn := remn + 1;
        choose
            condition$queue(depentry) $\land remx > depn - N \land depn = depx$:
                    condition$signal(depentry);
            condition$queue(rementry) $\land depx > remn \land remn = remx$:
                    condition$signal(rementry);
        end;
    end rem_enter;

    rem_exit = procedure;
        remx := remx + 1;
        choose
            condition$queue(depentry) $\land remx > depn - N \land depn = depx$:
                    condition$signal(depentry);
            condition$queue(rementry) $\land depx > remn \land remn = remx$:
                    condition$signal(rementry);
            *end*;
        *end* rem_exit;

        depn, depx, remn, remx := 0, 0, 0, 0;
*end* bb;
```

operation "read", in order to ensure that each "read" operation accesses the most current version of the database. Therefore, to the "readers-writers" specification of example 3 must be added an instantiation of the priority specification embodied in example 4 of Section 2.7. The overall specification is the following, to be denoted wpdb:

$$((\text{write}_i^{\text{enter}} \Rightarrow \text{write}_j^{\text{enter}}) \supset (\text{write}_i^{\text{exit}} \Rightarrow \text{write}_j^{\text{enter}})) \quad \wedge$$

$$((\text{write}_i^{\text{exit}} \Rightarrow \text{read}_k^{\text{enter}}) \vee (\text{read}_k^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}})) \quad \wedge$$

$$((\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}}) \supset (\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}})).$$

The specification contains three conjuncts to be analyzed. Of these, the third conjunct has already been treated in detail in Section 4.2, with the names "p" and "q" used for the operations instead of "write" and "read". By the analysis in that section, this conjunct contributes the condition

$$\text{count}(\text{write}^{\text{request}}) = \text{count}(\text{write}^{\text{enter}})$$

to the gate $\text{read}^{\text{enter}}$.

The other two conjuncts of the specification remain to be analyzed. They will be referred to as wpdb_1 and wpdb_2, respectively. The first conjunct wpdb_1 is

$$((\text{write}_i^{\text{enter}} \Rightarrow \text{write}_j^{\text{enter}}) \supset (\text{write}_i^{\text{exit}} \Rightarrow \text{write}_j^{\text{enter}})).$$

As with the bounded buffer example, there are no argument constraints in this or any other conjunct. The set of event expressions contained in the conjunct is

$$\text{Evexp}(\text{wpdb}_1) = \{\text{write}_i^{\text{enter}}, \text{write}_i^{\text{exit}}, \text{write}_j^{\text{enter}}\}.$$

There are three possible orderings among these three events:

$$(1)\ (\text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{write}_j^{\text{enter}})$$

$$(2)\ (\text{write}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}})$$

$$(3)\ (\text{write}_i^{\text{enter}} \Rightarrow \text{write}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}})$$

When ordering (1) is substituted into the conjunct wpdb_j, the result is the formula (TRUE ⊃ TRUE), or simply TRUE, so that ordering (1) is valid. Ordering (2) is also valid, since it evaluates wpdb_j to the formula (FALSE ⊃ FALSE), which similarly reduces to TRUE. Ordering (3) substituted into wpdb_j evaluates to (TRUE ⊃ FALSE), or FALSE, so that ordering (3) is invalid.

Comparing invalid ordering (3) with the valid orderings (1) and (2), the longest matching prefix is the one-element sequence $[\text{write}_i^{\text{enter}}]$, matching ordering (1). The offending event in (3) is therefore the event following this prefix, which is $\text{write}_j^{\text{enter}}$. This means that a condition must be derived for the $\text{write}^{\text{enter}}$ gate.

The state must be characterized at the point of each event in the $\text{write}^{\text{enter}}$ class that either occurs within a valid ordering or is the offending event in an invalid ordering. There are five such events, $\text{write}_i^{\text{enter}}$ and $\text{write}_j^{\text{enter}}$ in each of the two valid orderings (1) and (2), and $\text{write}_j^{\text{enter}}$ in invalid ordering (3), where it is the offending event. Denoting the characterization at event $\text{write}_i^{\text{enter}}$ in ordering (1) as c_{1i}, etc.:

c_{1i}: $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) < i \ \wedge\ \text{count}(\text{write}^{\text{enter}}) < j \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i)$

c_{1j}: $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{enter}}) < j \ \wedge\ \text{count}(\text{write}^{\text{exit}}) \geq i)$

c_{2j}: $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) < i \ \wedge\ \text{count}(\text{write}^{\text{enter}}) < j \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i)$

c_{2i}: $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) < i \ \wedge\ \text{count}(\text{write}^{\text{enter}}) \geq j \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i)$

c_{3j}: $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{enter}}) < j \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i)$

The four characterizations from the valid orderings are disjoined to form D_V:

$$\exists\,(i, j)\,((\text{count}(\text{write}^{\text{enter}}) < i \;\wedge\; \text{count}(\text{write}^{\text{exit}}) < i) \;\vee\; \text{count}(\text{write}^{\text{enter}}) < j)$$

Since there is only one invalid ordering, the disjunction of the invalid ordering characterizations D_i is simply c_{3j}. The preliminary condition is given by $(D_V \wedge \neg\,(D_i))$:

$$\exists\,(i, j)\,((\text{count}(\text{write}^{\text{enter}}) < i \;\wedge\; \text{count}(\text{write}^{\text{exit}}) < i) \;\vee\; \text{count}(\text{write}^{\text{enter}}) < j) \;\;\wedge$$
$$\forall\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) < i \;\vee\; \text{count}(\text{write}^{\text{enter}}) \geq j \;\vee\; \text{count}(\text{write}^{\text{exit}}) \geq i).$$

This reduces to

$$\forall\, i\,(\text{count}(\text{write}^{\text{enter}}) < i \;\vee\; \text{count}(\text{write}^{\text{exit}}) \geq i),$$

which in turn simplifies to

$$\text{count}(\text{write}^{\text{enter}}) = \text{count}(\text{write}^{\text{exit}}).$$

When this condition is tested for both $\text{write}^{\text{enter}}$ events in each of the two valid orderings, it is found to be satisfied in all cases, showing that it is the correct condition.

The other conjunct in the specification is wpdb_2:

$$((\text{write}_i^{\text{exit}} \Longrightarrow \text{read}_k^{\text{enter}}) \;\vee\; (\text{read}_k^{\text{exit}} \Longrightarrow \text{write}_i^{\text{enter}})).$$

The set of event expressions contained within wpdb_2 is given by

$$\text{Evexp}(\text{wpdb}_2) = \{\text{write}_i^{\text{enter}}, \text{write}_i^{\text{exit}}, \text{read}_k^{\text{enter}}, \text{read}_k^{\text{exit}}\}.$$

There are six possible orderings of these four events:

(1) $\text{write}_i^{\text{enter}} \Longrightarrow \text{write}_i^{\text{exit}} \Longrightarrow \text{read}_k^{\text{enter}} \Longrightarrow \text{read}_k^{\text{exit}}$

(2) $\text{read}_k^{\text{enter}} \Longrightarrow \text{read}_k^{\text{exit}} \Longrightarrow \text{write}_i^{\text{enter}} \Longrightarrow \text{write}_i^{\text{exit}}$

(3) $\text{write}_i^{\text{enter}} \Longrightarrow \text{read}_k^{\text{enter}} \Longrightarrow \text{write}_i^{\text{exit}} \Longrightarrow \text{read}_k^{\text{exit}}$

(4) $\text{write}_i^{\text{enter}} \Longrightarrow \text{read}_k^{\text{enter}} \Longrightarrow \text{read}_k^{\text{exit}} \Longrightarrow \text{write}_i^{\text{exit}}$

(5) $\text{read}_k^{\text{enter}} \Longrightarrow \text{write}_i^{\text{enter}} \Longrightarrow \text{read}_k^{\text{exit}} \Longrightarrow \text{write}_i^{\text{exit}}$

$$(6)\ \text{read}_k^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_k^{\text{exit}}$$

When wpdb_2 is evaluated for each of these orderings, the results for orderings (1) and (2) are (TRUE ∨ FALSE) and (FALSE ∨ TRUE), respectively, each of which equals TRUE. This means that orderings (1) and (2) are valid. For each of the other four orderings, the resulting formula is (FALSE ∨ FALSE), which equals FALSE, showing each of these orderings to be invalid.

The next step is to identify the offending event in each of the four invalid orderings. Both orderings (3) and (4) match valid ordering (1) as far as the first event, $\text{write}_i^{\text{enter}}$. The offending event in each is the second event, which in both cases is $\text{read}_k^{\text{enter}}$. Similarly, orderings (5) and (6) both match ordering (2) as far as the first event, $\text{read}_k^{\text{enter}}$, so that the offending event in each case is $\text{write}_i^{\text{enter}}$, the second event in the ordering. Solution specification conditions must be derived for two gates, $\text{read}^{\text{enter}}$ and $\text{write}^{\text{enter}}$.

In order to derive the condition for gate $\text{read}^{\text{enter}}$, it is necessary to characterize the state at certain events in the $\text{read}^{\text{enter}}$ class. The events in the class occurring in valid orderings are the $\text{read}_k^{\text{enter}}$ events in orderings (1) and (2). The offending events in the class are the occurrences of $\text{read}_k^{\text{enter}}$ in orderings (3) and (4). Denoting these characterizations as c_{1r}, c_{2r}, etc., they are:

$$c_{1r}\text{: } \exists (i, k)\ (\text{count}(\text{write}^{\text{enter}}) \geq i \wedge \text{count}(\text{write}^{\text{exit}}) \geq i \wedge$$
$$\text{count}(\text{read}^{\text{enter}}) < k \wedge \text{count}(\text{read}^{\text{exit}}) < k)$$

$$c_{2r}\text{: } \exists (i, k)\ (\text{count}(\text{write}^{\text{enter}}) < i \wedge \text{count}(\text{write}^{\text{exit}}) < i \wedge$$
$$\text{count}(\text{read}^{\text{enter}}) < k \wedge \text{count}(\text{read}^{\text{exit}}) < k)$$

$$c_{3r}\text{: } \exists\,(i, k)\,(\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i \ \wedge$$
$$\text{count}(\text{read}^{\text{enter}}) < k \ \wedge\ \text{count}(\text{read}^{\text{exit}}) < k)$$

$$c_{4r}\text{: } \exists\,(i, k)\,(\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i \ \wedge$$
$$\text{count}(\text{read}^{\text{enter}}) < k \ \wedge\ \text{count}(\text{read}^{\text{exit}}) < k)$$

The two disjunctions are given by $D_v = (c_{1r} \vee c_{2r})$, and ${}_i = (c_{3r} \vee c_{4r})$:

$$D_v\text{: } \exists\,(i, k)\,(((\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{exit}}) \geq i) \ \vee$$
$$(\text{count}(\text{write}^{\text{enter}}) < i \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i)) \ \wedge$$
$$\text{count}(\text{read}^{\text{enter}}) < k \ \wedge\ \text{count}(\text{read}^{\text{exit}}) < k)$$

$$D_i\text{: } \exists\,(i, k)\,(\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i \ \wedge$$
$$\text{count}(\text{read}^{\text{enter}}) < k \ \wedge\ \text{count}(\text{read}^{\text{exit}}) < k)$$

The preliminary condition is formed by the expression $(D_v \wedge (\neg D_i))$,

$$\exists\,(i, k)\,(((\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{exit}}) \geq i) \ \vee$$
$$(\text{count}(\text{write}^{\text{enter}}) < i \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i)) \ \wedge$$
$$\text{count}(\text{read}^{\text{enter}}) < k \ \wedge\ \text{count}(\text{read}^{\text{exit}}) < k) \ \ \wedge$$
$$\forall\,(i, k)\,(\text{count}(\text{write}^{\text{enter}}) < i \ \vee\ \text{count}(\text{write}^{\text{exit}}) \geq i \ \vee$$
$$\text{count}(\text{read}^{\text{enter}}) \geq k \ \vee\ \text{count}(\text{read}^{\text{exit}}) \geq k).$$

This can be simplified to

$$\forall\, i\,(\text{count}(\text{write}^{\text{enter}}) < i \ \vee\ \text{count}(\text{write}^{\text{exit}}) \geq i),$$

which in turn is equivalent to

$$\text{count}(\text{write}^{\text{enter}}) = \text{count}(\text{write}^{\text{exit}}).$$

This condition satisfies both valid orderings (1) and (2), and so is correct.

Because of the symmetry of the specification wpdb_2, and therefore of the orderings, the derivation of the condition for gate $\mathrm{write}^{\mathrm{enter}}$ is completely isomorphic to the above derivation. Rather than repeat essentially the same derivation, I will simply state the result, that the condition for gate $\mathrm{write}^{\mathrm{enter}}$ as a result of this conjunct is

$$\mathrm{count}(\mathrm{read}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{read}^{\mathrm{exit}}).$$

The overall solution specification for specification wpdb is constructed by conjoining the conditions from the individual conjuncts. The composite conditions are:

For gate $\mathrm{read}^{\mathrm{enter}}$:

$$\mathrm{count}(\mathrm{write}^{\mathrm{request}}) = \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \land \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{write}^{\mathrm{exit}})$$

For gate $\mathrm{write}^{\mathrm{enter}}$:

$$\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{write}^{\mathrm{exit}}) \land \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{read}^{\mathrm{exit}})$$

In the monitor into which this solution specification is translated, there must be integer variables wr, wn, wx, rn, and rx, representing $\mathrm{count}(\mathrm{write}^{\mathrm{request}})$, $\mathrm{count}(\mathrm{write}^{\mathrm{enter}})$, $\mathrm{count}(\mathrm{write}^{\mathrm{exit}})$, $\mathrm{count}(\mathrm{read}^{\mathrm{enter}})$, and $\mathrm{count}(\mathrm{read}^{\mathrm{exit}})$, respectively. There must also be condition variables writeentry and readentry corresponding to the conditions in the solution specification. Their associated boolean predicates are

readentry: $\mathrm{wr} = \mathrm{wn} \land \mathrm{wn} = \mathrm{wx}$

writeentry: $\mathrm{wn} = \mathrm{wx} \land \mathrm{rn} = \mathrm{rx}$

Notice that $\mathrm{count}(\mathrm{read}^{\mathrm{request}})$ does not appear in the solution specification, so that no variable is needed for it, and thus a procedure read_request is not required. The resulting monitor appears in Figure 6.2.

Figure 6.2. Monitor for writers' priority database

```
wpdb = monitor;
      wr, wn, wx, rn, rx: integer;
      readentry, writeentry: condition;

      write_request = procedure;
            wr := wr + 1;
            choose
                  condition$queue(readentry) ∧ wr = wn ∧ wn = wx:
                              condition$signal(readentry);
                  condition$queue(writeentry) ∧ wn = wx ∧ rn = rx:
                              condition$signal(writeentry);
            end;
      end write_request;

      write_enter = procedure;
            if wn ≠ wx ∨ rn ≠ rx then condition$wait(writeentry); end;
            wn := wn + 1;
            choose
                  condition$queue(readentry) ∧ wr = wn ∧ wn = wx:
                              condition$signal(readentry);
                  condition$queue(writeentry) ∧ wn = wx ∧ rn = rx:
                              condition$signal(writeentry);
            end;
      end write_enter;

      write_exit = procedure;
            wx := wx + 1;
            choose
                  condition$queue(readentry) ∧ wr = wn ∧ wn = wx:
                              condition$signal(readentry);
                  condition$queue(writeentry) ∧ wn = wx ∧ rn = rx:
                              condition$signal(writeentry);
            end;
      end write_exit;

      read_enter = procedure;
            if wr ≠ wn ∨ wn ≠ wx then condition$wait(readentry); end;
            rn := rn + 1;
            choose
                  condition$queue(readentry) ∧ wr = wn ∧ wn = wx:
                              condition$signal(readentry);
                  condition$queue(writeentry) ∧ wn = wx ∧ rn = rx:
                              condition$signal(writeentry);
```

```
        end;
    end read_enter;

    read_exit = procedure;
        rx := rx + 1;
        choose
            condition$queue(readentry)  ∧  wr = wn  ∧  wn = wx:
                    condition$signal(readentry);
            condition$queue(writeentry)  ∧  wn = wx  ∧  rn = rx:
                    condition$signal(writeentry);
        end;
    end read_exit;

    wr, wn, wx, rn, rx := 0, 0, 0, 0, 0;
end wpdb;
```

6.4 Alternating priority database

The next example is a variation of the previous one. Again the data abstraction is a database, with operations "read" and "write" obeying the "readers-writers" property. In this case, though, the relative priority of the two operations is to alternate, so that in a situation in which activations of both operations are being continually requested, the result is that first a single "write" activation executes, then all waiting "read" activations, then the next "write", etc. This scheduling policy is the one followed by the readers-writers example in [Hoa74], and is referred to as the "fair database" in [Gre75].

The specification for the "alternating priority" database is given by:

$$((\text{write}_i^{\text{enter}} \Rightarrow \text{write}_j^{\text{enter}}) \supset (\text{write}_i^{\text{exit}} \Rightarrow \text{write}_j^{\text{enter}})) \;\wedge$$

$$((\text{write}_i^{\text{exit}} \Rightarrow \text{read}_k^{\text{enter}}) \vee (\text{read}_k^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}})) \;\wedge$$

$$((\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{exit}}) \supset (\text{read}_j^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}})) \;\wedge$$

$$((\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}}) \supset$$

$$\exists\, m\, (\text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}})).$$

The first two conjuncts express the "readers-writers" property and are the same as for the previous example wpdb. The analysis of the previous section need not be repeated here. The last two conjuncts state the "alternating priority" property. The third conjunct apdb_3 requires a "write" activation to wait to enter until all "read" activations that were requested during the execution of the previous "write" have done so first. The fourth conjunct apdb_4 prevents an activation of "read" from entering until an activation of "write" has exited, assuming that there is at least one "write" that is waiting at the point at which the "read" is requested. This prevents new "read" activations from continually entering. Solution specification conditions must be derived for these two conjuncts.

The first conjunct to be analyzed is apdb_3:

$$((\mathrm{write}_i^{\mathrm{enter}} \Longrightarrow \mathrm{read}_j^{\mathrm{request}} \Longrightarrow \mathrm{write}_i^{\mathrm{exit}}) \supset (\mathrm{read}_j^{\mathrm{enter}} \Longrightarrow \mathrm{write}_{i+1}^{\mathrm{enter}}))$$

The set of event expressions in the conjunct is

$$\mathrm{Evexp}(\mathrm{apdb}_3) = \{\mathrm{write}_i^{\mathrm{enter}}, \mathrm{write}_i^{\mathrm{exit}}, \mathrm{write}_{i+1}^{\mathrm{enter}}, \mathrm{read}_j^{\mathrm{request}}, \mathrm{read}_j^{\mathrm{enter}}\}.$$

With these five events to be ordered, there are eighteen possible orderings. They appear in Figure 6.3.

When each of these orderings is used to evaluate the specification apdb_3, orderings (1) through (15) are found to be valid, while orderings (16) through (18) are invalid. Since ordering (16) matches ordering (1) through the first three events in each, the offending event in (16) is the fourth event $\mathrm{write}_{i+1}^{\mathrm{enter}}$. Each of the other two invalid orderings (17) and (18) matches orderings (1) through (3) as far as the first two events, so the offending event in each is the third event, also $\mathrm{write}_{i+1}^{\mathrm{enter}}$. This means that a condition must be found for gate $\mathrm{write}^{\mathrm{enter}}$.

The characterization of the state at the point of the offending event $\mathrm{write}_{i+1}^{\mathrm{enter}}$ in ordering (16) is given by:

$$\exists\,(i, j)\,(\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \geq i \;\wedge\; \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) < (i+1) \;\wedge\; \mathrm{count}(\mathrm{write}^{\mathrm{exit}}) \geq i \;\wedge$$
$$\mathrm{count}(\mathrm{read}^{\mathrm{request}}) \geq j \;\wedge\; \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) < j).$$

The characterizations for orderings (17) and (18) are identical, namely

$$\exists\,(i, j)\,(\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \geq i \;\wedge\; \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) < (i+1) \;\wedge\; \mathrm{count}(\mathrm{write}^{\mathrm{exit}}) < i \;\wedge$$
$$\mathrm{count}(\mathrm{read}^{\mathrm{request}}) \geq j \;\wedge\; \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) < j).$$

Figure 6.3. Possible orderings for apdb_3

(1) $\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}}$

(2) $\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{write}_{i+1}^{\text{enter}}$

(3) $\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}}$

(4) $\text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}}$

(5) $\text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(6) $\text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}}$

(7) $\text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{write}_{i+1}^{\text{enter}}$

(8) $\text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}}$

(9) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{write}_{i+1}^{\text{enter}}$

(10) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}}$

(11) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}}$

(12) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}}$

(13) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}}$

(14) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(15) $\text{write}_i^{\text{enter}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}}$

(16) $\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(17) $\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}}$

(18) $\text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_{i+1}^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{exit}}$

The disjunction of these two characterizations is equal to D_i:

$$D_i\text{:}\ \exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) = i\ \wedge\ \text{count}(\text{read}^{\text{request}}) \geq j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) < j).$$

The state also must be characterized for events $\text{write}_i^{\text{enter}}$ and $\text{write}_{i+1}^{\text{enter}}$ in each of the 15 valid orderings. This means that 30 separate characterizations must be formed. However, many of the characterizations for different orderings are identical. In fact there are only nine distinct characterizations, which are listed here:

(a) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) \geq i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) \geq i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) \geq j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) \geq j)$

(b) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) \geq i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) \geq i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) \geq j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) < j)$

(c) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) \geq i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) \geq i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) < j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) < j)$

(d) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) < i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) < j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) < j)$

(e) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) < i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) \geq j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) < j)$

(f) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) < i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) \geq j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) \geq j)$

(g) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) \geq i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) \geq j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) \geq j)$

(h) $\exists\,(i, j)\,(\text{count}(\text{write}^{\text{enter}}) \geq i\ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1)\ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i\ \wedge$
$\text{count}(\text{read}^{\text{request}}) \geq j\ \wedge\ \text{count}(\text{read}^{\text{enter}}) < j)$

(i) $\exists\, (i, j)\, (\text{count}(\text{write}^{\text{enter}}) \geq i \ \wedge\ \text{count}(\text{write}^{\text{enter}}) < (i + 1) \ \wedge\ \text{count}(\text{write}^{\text{exit}}) < i \ \wedge$

$\text{count}(\text{read}^{\text{request}}) < j \ \wedge\ \text{count}(\text{read}^{\text{enter}}) < j)$

The disjunction of these nine charácterizations is D_V, which reduces to:

$$D_V\!:\ \exists\, (i, j)\, ((\text{count}(\text{write}^{\text{enter}}) \geq i \ \vee\ \text{count}(\text{write}^{\text{exit}}) < i) \ \wedge$$

$$(\text{count}(\text{read}^{\text{request}}) \geq j \ \vee\ \text{count}(\text{read}^{\text{enter}}) < j)).$$

The preliminary condition is $(D_V \wedge (\neg\, D_i))$,

$$\exists\, (i, j)\, ((\text{count}(\text{write}^{\text{enter}}) \geq i \ \vee\ \text{count}(\text{write}^{\text{exit}}) < i) \ \wedge$$

$$(\text{count}(\text{read}^{\text{request}}) \geq j \ \vee\ \text{count}(\text{read}^{\text{enter}}) < j)) \ \wedge$$

$$\forall\, (i, j)\, (\text{count}(\text{write}^{\text{enter}}) \neq i \ \vee\ \text{count}(\text{read}^{\text{request}}) < j \ \vee\ \text{count}(\text{read}^{\text{enter}}) \geq j),$$

which when simplified reduces to:

$$\text{count}(\text{read}^{\text{request}}) = \text{count}(\text{read}^{\text{enter}}).$$

This condition must be tested for both write$^{\text{enter}}$ events in each of the fifteen valid orderings. In doing so, it is discovered that the condition is not satisfied for the following events:

$\text{write}_i^{\text{enter}}$ in orderings 13 and 14

$\text{write}_{i+1}^{\text{enter}}$ in orderings 5, 11, 12 and 13

An event must be found that precedes each of these events, as well as the offending event in each of the invalid orderings. The only such event is $\text{read}_j^{\text{request}}$. The state is therefore characterized at this event in each of the orderings in question. In ordering (5), the characterization at event $\text{read}_j^{\text{request}}$ is:

$$\begin{gathered}
\exists\,(i, j)\;([\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] \geq i \;\wedge \\
[\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < (i + 1) \;\wedge \\
[\mathrm{count}(\mathrm{write}^{\mathrm{exit}}) \bullet \mathrm{read}^{\mathrm{request}}] \geq i \;\wedge \\
[\mathrm{count}(\mathrm{read}^{\mathrm{request}}) \bullet \mathrm{read}^{\mathrm{request}}] < j \;\wedge \\
[\mathrm{count}(\mathrm{read}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < j).
\end{gathered}$$

In each of the other valid orderings in question, it is:

$$\begin{gathered}
\exists\,(i, j)\;([\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < i \;\wedge \\
[\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < (i + 1) \;\wedge \\
[\mathrm{count}(\mathrm{write}^{\mathrm{exit}}) \bullet \mathrm{read}^{\mathrm{request}}] < i \;\wedge \\
[\mathrm{count}(\mathrm{read}^{\mathrm{request}}) \bullet \mathrm{read}^{\mathrm{request}}] < j \;\wedge \\
[\mathrm{count}(\mathrm{read}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < j).
\end{gathered}$$

This means that the formula D_V' is given by the disjunction of these two, or:

$$\begin{gathered}
D_V\text{'}\!:\;\; \exists\,(i, j)\;(([\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] \geq i \;\wedge \\
[\mathrm{count}(\mathrm{write}^{\mathrm{exit}}) \bullet \mathrm{read}^{\mathrm{request}}] \geq i) \;\vee \\
([\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < i \;\wedge \\
[\mathrm{count}(\mathrm{write}^{\mathrm{exit}}) \bullet \mathrm{read}^{\mathrm{request}}] < i) \;\wedge \\
[\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < (i + 1) \;\wedge \\
[\mathrm{count}(\mathrm{read}^{\mathrm{request}}) \bullet \mathrm{read}^{\mathrm{request}}] < j \;\wedge \\
[\mathrm{count}(\mathrm{read}^{\mathrm{enter}}) \bullet \mathrm{read}^{\mathrm{request}}] < j).
\end{gathered}$$

The characterization at $\mathrm{read}_j^{\mathrm{request}}$ in each of the three invalid orderings is the same,

$$\exists\, (i, j)\ (([\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] \geq i \ \wedge$$

$$[\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}] < i) \ \wedge$$

$$[\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] < (i + 1) \ \wedge$$

$$[\text{count}(\text{read}^{\text{request}}) \bullet \text{read}^{\text{request}}] < j \ \wedge$$

$$[\text{count}(\text{read}^{\text{enter}}) \bullet \text{read}^{\text{request}}] < j).$$

This formula is therefore D_f'. The weakening term is formed by $D_v' \wedge (\neg D_v')$:

$$\exists\, (i, j)\ (([\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] \geq i \ \wedge$$

$$[\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}] \geq i) \ \vee$$

$$([\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] < i \ \wedge$$

$$[\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}] < i) \ \wedge$$

$$[\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] < (i + 1) \ \wedge$$

$$[\text{count}(\text{read}^{\text{request}}) \bullet \text{read}^{\text{request}}] < j \ \wedge$$

$$[\text{count}(\text{read}^{\text{enter}}) \bullet \text{read}^{\text{request}}] < j) \ \wedge$$

$$\forall\, (i, j)\ (([\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] < i \ \vee$$

$$[\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}] \geq i) \ \vee$$

$$[\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] \geq (i + 1) \ \vee$$

$$[\text{count}(\text{read}^{\text{request}}) \bullet \text{read}^{\text{request}}] \geq j \ \vee$$

$$[\text{count}(\text{read}^{\text{enter}}) \bullet \text{read}^{\text{request}}] \geq j).$$

When simplified, this reduces to:

$$[\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] = [\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}].$$

This weakening term is tested in each of the six events in valid orderings for which the preliminary condition is not satisfied. It is found that the weakening term is satisfied in each case. Therefore, disjoining the weakening term to the preliminary condition obtains the correct condition. The condition for gate $\text{write}^{\text{enter}}$ from conjunct apdb_3 is:

$$\text{count}(\text{read}^{\text{request}}) = \text{count}(\text{read}^{\text{enter}}) \lor$$

$$[\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] = [\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}].$$

This condition makes sense intuitively. The first disjunct states that there are no unfulfilled requests for activations of "read". The second says that the most recent request event for "read" took place at a point at which no activation of "write" was active. Therefore, a "read" activation is allowed to proceed ahead of the next waiting "write" activation if it was requested during the previous "write" activation.

There remains conjunct apdb_4 to analyze:

$$((\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}}) \supset$$

$$\exists\, m\ (\text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}})).$$

Unfortunately, the analysis of this conjunct is even more complicated than that of the previous conjunct, owing to the fact that there are 30 possible orderings of the 5 events contained within it. These 30 orderings are listed in Figure 6.4.

Rather than go through the details of the derivation, the complete process will simply be summarized. Of the 30 orderings, the orderings numbered (1) through (23) are found to be valid. Orderings (24) through (30) are invalid, with the offending event in each being $\text{read}_i^{\text{enter}}$. A condition must therefore be derived for gate $\text{read}^{\text{enter}}$. When the preliminary

Figure 6.4. Possible orderings for apdb_4

(1) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}}$

(2) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}}$

(3) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}}$

(4) $\text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}}$

(5) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(6) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(7) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}}$

(8) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}}$

(9) $\text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}}$

(10) $\text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}}$

(11) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}}$

(12) $\text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}}$

(13) $\text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}}$

(14) $\text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}}$

(15) $\text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(16) $\text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}}$

(17) $\text{write}_i^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}}$

(18) $\text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}}$

(19) $\text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}}$

(20) $\text{write}_i^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}}$

(21) $\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}}$

(22) $\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(23) $\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}}$

(24) $\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{enter}}$

(25) $\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}}$

(26) $\text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_m^{\text{exit}}$

(27) $\text{write}_i^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(28) $\text{write}_i^{\text{request}} \Rightarrow \text{write}_m^{\text{exit}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}}$

(29) $\text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{write}_i^{\text{enter}} \Rightarrow \text{read}_j^{\text{enter}}$

(30) $\text{write}_m^{\text{exit}} \Rightarrow \text{write}_i^{\text{request}} \Rightarrow \text{read}_j^{\text{request}} \Rightarrow \text{read}_j^{\text{enter}} \Rightarrow \text{write}_i^{\text{enter}}$

condition is formed, it reduces to FALSE. This is the extreme case of an overly strong condition, in that *none* of the 23 valid orderings is allowed.

The only event that precedes $\text{read}_i^{\text{enter}}$ in all 30 orderings is $\text{read}_i^{\text{request}}$. The weakening term obtained by considering the state at event $\text{read}_i^{\text{request}}$ alone is:

$$[\text{count}(\text{write}^{\text{request}}) \bullet \text{read}^{\text{request}}] = [\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}].$$

This condition is satisfied by valid orderings (1) through (20), but not (21) through (23). This means that the characterizations of both the current state and the previous state at $\text{read}_i^{\text{request}}$ must be used at the same time to obtain another weakening term for these three orderings. The weakening term obtained is:

$$[\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}] < \text{count}(\text{write}^{\text{exit}})$$

which is satisfied by each of the orderings (21) through (23).

The solution specification condition for gate $\text{read}^{\text{enter}}$ from this conjunct is therefore the disjunction of the two weakening terms:

$$[\text{count}(\text{write}^{\text{request}}) \bullet \text{read}^{\text{request}}] = [\text{count}(\text{write}^{\text{enter}}) \bullet \text{read}^{\text{request}}] \quad \vee$$

$$([\text{count}(\text{write}^{\text{exit}}) \bullet \text{read}^{\text{request}}] < \text{count}(\text{write}^{\text{exit}})).$$

Again this condition makes intuitive sense. The first disjunct states that no activations of "write" were requested but waiting at the point at which the "read" under consideration was requested. The second says that some activation of "write" has exited since the point at which this "read" was requested. One of these must be true before the "read" can enter.

The overall solution specification for specification apdb is given by the conjunction of the individual conditions obtained for each of the four conjuncts:

For gate read^{enter}:

$$(\mathrm{count}(\mathrm{write}^{enter}) = \mathrm{count}(\mathrm{write}^{exit})) \quad \wedge$$

$$(([\mathrm{count}(\mathrm{write}^{request})\ @\ \mathrm{read}^{request}] = [\mathrm{count}(\mathrm{write}^{enter})\ @\ \mathrm{read}^{request}]) \quad \vee$$

$$([\mathrm{count}(\mathrm{write}^{exit})\ @\ \mathrm{read}^{request}] < \mathrm{count}(\mathrm{write}^{exit})))$$

For gate write^{enter}:

$$(\mathrm{count}(\mathrm{write}^{enter}) = \mathrm{count}(\mathrm{write}^{exit})) \quad \wedge$$

$$(\mathrm{count}(\mathrm{read}^{enter}) = \mathrm{count}(\mathrm{read}^{exit})) \quad \wedge$$

$$((\mathrm{count}(\mathrm{read}^{request}) = \mathrm{count}(\mathrm{read}^{enter})) \quad \vee$$

$$([\mathrm{count}(\mathrm{write}^{enter})\ @\ \mathrm{read}^{request}] = [\mathrm{count}(\mathrm{write}^{exit})\ @\ \mathrm{read}^{request}])).$$

The monitor implementation of this solution specification requires three variables wr, wn, and wx, to represent the current values of count(write^{request}), count(write^{enter}), and count(write^{exit}), and three variables rr, rn, and rx, to represent the values of count(read^{request}), count(read^{enter}), and count(read^{exit}). In addition, three variables are required to save values at a previous state: wrrr for [count(write^{request}) @ read^{request}], wnrr for [count(write^{enter}) @ read^{request}], and wxrr for [count(write^{exit}) @ read^{request}]. The values of these variables are set in the monitor procedure read_request corresponding to gate read^{request} by saving the values of the variables representing the corresponding current quantities. For instance, variable wrrr, representing [count(write^{request}) @ read^{request}], saves the value of wr, which represents count(write^{request}). The two condition variables, and their associated predicates, are:

readentry: $wn = wx \wedge (wrrr = wnrr \vee wxrr < wx)$

writeentry: $wn = wx \wedge rn = rx \wedge (rr = rn \vee wnrr = wxrr)$

The monitor that is obtained appears in Figure 6.5.

6.5 Disk head scheduler

The final example of this chapter is the "disk head scheduler" problem. Actually, the specification used here is a simplification of the actual disk head scheduling specification that appears as Example 14 in Section 2.7. The real disk head scheduler keeps the disk head sweeping in one direction until all requested accesses in that direction have been made, then reverses the direction and repeats. Accesses are made as the requested tracks are encountered in the sweep, so that the next track to be accessed is the one that is closest to the currently accessed track in the direction being swept. The simplification here involves disregarding the direction in which the disk head is sweeping. We simply wish that the next track to be accessed is closest to the currently accessed track of all requested tracks in a given direction. The requirement that the disk head sweep continuously in one direction until no further accesses have been requested in that direction is omitted. This allows the specification to be considerably simplified (though it also introduces the possibility of starvation, as noted in Chapter 7).

We assume here that accessing a disk track is accomplished by means of an operation named "a" on the "disk" data type. This operation takes a single parameter x of type "track_no", giving the value of the track number being accessed. Activations of "a" must be mutually exclusive, since only one access can occur at a time. The first conjunct of

Figure 6.5. Monitor for alternating priority database

```
apdb = monitor;
    wr, wn, wx, rr, rn, rx: integer;
    wrrr, wnrr, wxrr: integer;
    readentry, writeentry: condition;

    write_request = procedure;
        wr := wr + 1;
        choose
            condition$queue(readentry)  ∧
                wn = wx  ∧  (wrrr = wnrr  ∨  wxrr < wx):
                    condition$signal(readentry);
            condition$queue(writeentry)  ∧
                wn = wx  ∧  rn = rx  ∧  (rr = rn  ∨  wnrr = wxrr):
                    condition$signal(writeentry);
        end;
    end write_request;

    write_enter = procedure;
        if wn ≠ wx  ∨  rn ≠ rx  ∨  (rr ≠ rn  ∧  wnrr ≠ wxrr)
            then condition$wait(writeentry); end;
        wn := wn + 1;
        choose
            condition$queue(readentry)  ∧
                wn = wx  ∧  (wrrr = wnrr  ∨  wxrr < wx):
                    condition$signal(readentry);
            condition$queue(writeentry)  ∧
                wn = wx  ∧  rn = rx  ∧  (rr = rn  ∨  wnrr = wxrr):
                    condition$signal(writeentry);
        end;
    end write_enter;

    write_exit = procedure;
        wx := wx + 1;
        choose
            condition$queue(readentry)  ∧
                wn = wx  ∧  (wrrr = wnrr  ∨  wxrr < wx):
                    condition$signal(readentry);
            condition$queue(writeentry)  ∧
                wn = wx  ∧  rn = rx  ∧  (rr = rn  ∨  wnrr = wxrr):
                    condition$signal(writeentry);
        end;
    end write_exit;
```

```
          read_request = procedure;
                rr := rr + 1;
                wrrr := wr;
                wnrr := wn;
                wxrr := wx;
                choose
                      condition$queue(readentry)  ∧
                            wn = wx  ∧  (wrrr = wnrr  ∨  wxrr < wx):
                                  condition$signal(readentry);
                      condition$queue(writeentry)  ∧
                            wn = wx  ∧  rn = rx  ∧  (rr = rn  ∨  wnrr = wxrr):
                                  condition$signal(writeentry);
                end;
          end read_request;

          read_enter = procedure;
                if wn ≠ wx  ∨  (wrrr ≠ wnrr  ∧  wxrr ≥ wx)
                      then condition$wait(readentry); end;
                rn := rn + 1;
                choose
                      condition$queue(readentry)  ∧
                            wn = wx  ∧  (wrrr = wnrr  ∨  wxrr < wx):
                                  condition$signal(readentry);
                      condition$queue(writeentry)  ∧
                            wn = wx  ∧  rn = rx  ∧  (rr = rn  ∨  wnrr = wxrr):
                                  condition$signal(writeentry);
                end;
          end read_enter;

          read_exit = procedure;
                rx := rx + 1;
                choose
                      condition$queue(readentry)  ∧
                            wn = wx  ∧  (wrrr = wnrr  ∨  wxrr < wx):
                                  condition$signal(readentry);
                      condition$queue(writeentry)  ∧
                            wn = wx  ∧  rn = rx  ∧  (rr = rn  ∨  wnrr = wxrr):
                                  condition$signal(writeentry);
                end;
          end read_exit;

          wr, wn, wx, rr, rn, rx := 0, 0, 0, 0, 0, 0;
          wrrr, wnrr, wxrr, := 0, 0, 0;
    end apdb;
```

specification dh specifies this mutual exclusion, and the second specifies the scheduling policy desired:

$$((a_m{}^{enter} \Rightarrow a_n{}^{enter}) \supset (a_m{}^{exit} \Rightarrow a_n{}^{enter})) \quad \wedge$$

$$((a_i(x2)^{request} \Rightarrow a_k(x1)^{exit} \Rightarrow a_i(x2)^{enter}) \wedge$$

$$(a_j(x3)^{request} \Rightarrow a_k(x1)^{exit} \Rightarrow a_j(x3)^{enter}) \wedge$$

$$(x1 < x2 < x3 \;\vee\; x1 > x2 > x3) \supset$$

$$(a_i(x2)^{enter} \Rightarrow a_j(x3)^{enter})).$$

The analysis of the first conjunct has been carried out already in Section 6.3, where the same property was specified for operation "write" as part of the "readers-writers" property. Here we will consider the scheduling property conjunct dh_2.

First, the argument constraint predicate $(x1 < x2 < x3 \;\vee\; x1 > x2 > x3)$ must be incorporated into the conjunct. The predicate already appears in the hypothesis of an implication. It can be incorporated by parameterizing it and then qualifying the appropriate procedure activations. The parameterized form of the predicate is

$$(x1 = u) \;\wedge\; (x2 = t) \;\wedge\; (u < t < x3 \;\vee\; u > t > x3).$$

This means that activation $a_k(x1)$ must be qualified with the predicate $(x1 = u)$, activation $a_i(x2)$ with the predicate $(x2 = t)$, and activation $a_j(x3)$ with the predicate $(u < t < x3 \;\vee\; u > t > x3)$. The transformed specification then becomes:

$$([a_i(x2) \mid (x2 = t)]^{request} \Longrightarrow [a_k(x1) \mid (x1 = u)]^{exit} \Longrightarrow [a_i(x2) \mid (x2 = t)]^{enter}) \wedge$$

$$([a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{request} \Longrightarrow [a_k(x1) \mid (x1 = u)]^{exit}$$

$$\Longrightarrow [a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{enter})$$

$$\supset \quad ([a_i(x2) \mid (x2 = t)]^{enter} \Longrightarrow [a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{enter}).$$

Now that the argument constraint information has been incorporated into the conjunct by means of qualification, the analysis can proceed normally. There are five events mentioned in the conjunct:

$$Evexp(dh_2) = \{[a_i(x2) \mid (x2 = t)]^{request},\ [a_i(x2) \mid (x2 = t)]^{enter}),\ [a_k(x1) \mid (x1 = u)]^{exit},$$

$$[a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{request},\ [a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{enter}\}.$$

There are 30 possible orderings among these five events. Rather than list all 30 of them, only the two invalid ones are given here:

$$(1)\ [a_i(x2) \mid (x2 = t)]^{request} \Longrightarrow [a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{request} \Longrightarrow$$

$$[a_k(x1) \mid (x1 = u)]^{exit} \Longrightarrow [a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{enter} \Longrightarrow$$

$$[a_i(x2) \mid (x2 = t)]^{enter}.$$

$$(2)\ [a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{request} \Longrightarrow [a_i(x2) \mid (x2 = t)]^{request}] \Longrightarrow$$

$$[a_k(x1) \mid (x1 = u)]^{exit} \Longrightarrow [a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{enter} \Longrightarrow$$

$$[a_i(x2) \mid (x2 = t)]^{enter}.$$

The offending event in each is $[a_j(x3) \mid (u < t < x3 \vee u > t > x3)]^{enter}$. This means that a condition must be derived for gate $[a(x) \mid (u < t < x \vee u > t > x)]^{enter}$. Since the state characterization is the same at the point of the offending event in both orderings, this characterization becomes the term D_i:

$$\exists\,(i, j, k)\,(\mathrm{count}([a(x) \mid (x = u)]^{exit}) \geq k \;\wedge$$

$$\mathrm{count}([a(x) \mid (u < t < x \vee u > t > x)]^{request}) \geq j \;\wedge\; \mathrm{count}([a(x) \mid (x = t)]^{request}) \geq i \;\wedge$$

$$\mathrm{count}([a(x) \mid (u < t < x \vee u > t > x)]^{enter}) < j \;\wedge\; \mathrm{count}([a(x) \mid (x = t)]^{enter})) < i)$$

The term D_V, being the disjunction of 23 characterizations, is quite complicated. However, when the expression $(D_V \wedge (\neg\, D_i))$ is constructed, the formula can be simplified considerably. The result of the simplification is to arrive at the following preliminary condition:

$$\forall\, i\,(\mathrm{count}([a(x) \mid (x = t)]^{request}) < i \;\vee\; \mathrm{count}([a(x) \mid (x = t)]^{enter})) \geq i).$$

This is equivalent to the even simpler

$$\mathrm{count}([a(x) \mid (x = t)]^{request}) = \mathrm{count}([a(x) \mid (x = t)]^{enter}).$$

This condition is found to satisfy all the valid orderings, and therefore is correct as it stands.

The overall solution specification for specification dh consists of the following gate conditions:

For gate a^{enter}: $\mathrm{count}(a^{enter}) = \mathrm{count}(a^{exit})$

For gate $[a(x) \mid (u < t < x \vee u > t > x)]^{enter}$:

$$\mathrm{count}([a(x) \mid (x = t)]^{request}) = \mathrm{count}([a(x) \mid (x = t)]^{enter})$$

where u is the parameter of the activation corresponding to the most recent a^{exit} event.

A monitor must now be constructed to implement this solution specification.

The monitor must contain three procedures a_request, a_enter, and a_exit, to correspond to the three event classes. Since there are qualified gates in the solution specification, each of the monitor procedures must take the same parameter x as operation a. There must be variables an and ax to represent count(a^{enter}) and count(a^{exit}), respectively. In addition, there must be a local variable u of type track_no, the same type as parameter t, representing the value of the parameter of the most recent call on procedure a_exit. This variable should be initialized to an appropriate value, such as the minimum possible track number.

In order to implement the parameterized counts, there must be two objects atreq and atent of type counts[track_no] to hold the values of count($[a(x) \mid (x = t)]^{request}$) and count($[a(x) \mid (x = t)]^{enter}$) for all values of x. Procedure a_request increments a count in atreq, and procedure a_enter increments a count in atent. Each of these objects must be created in the initialization code for the monitor.

The qualifying predicate on the p^{enter} gate

$$(u < t < x \lor u > t > x)$$

is a non-functional relation. The entry conditions must be implemented by an object aentry of type conditions[track_no], that holds the conditions for all relevant values of x. A condition for a given value of x is added to aentry by an "add" operation at the start of procedure a_enter. The predicate associated with the condition for value t is given by

$$an = ax \land \text{cnts\$get(atreq, t)} = \text{cnts\$get(atent, t)},$$

combining the predicates associated with the unqualified and qualified gates. It is also necessary to have an object "tracks" of type set[track_no] to maintain the set of relevant track numbers. Elements are added to track by "insert" operations within procedures a_request and a_enter. The resulting monitor appears in Figure 6.6.

Figure 6.6. Monitor for disk head scheduler

```
dh = monitor;

    an, ax: integer;
    u: track_no;
    atreq, atent: counts[track_no];
    aentry: conditions[track_no];
    tracks := set[track_no];

    a_request = procedure(x:track_no);
        counts[track_no]$incr(atreq, x);
        set[track_no]$insert(tracks, x);
        for z:track_no in conditions[track_no]$domain(aentry) do
            if condition$queue(conditions[track_no]$get(aentry, z)) then
                ok:boolean := true;
                for t:track_no in set[track_no]$elements(tracks) do
                    if (u < t < z  ∨  u > t > z)  ∧
                             (an ≠ ax  ∨  counts[track_no]$get(atreq, t) ≠
                                  counts[track_no]$get(atent, t))
                    then ok := false; end;
                end;
                if ok then condition$signal(conditions[track_no]$get(aentry, z)); end;
            end;
        end;
    end a_request;

    a_enter = procedure(x:track_no);
        conditions[track_no]$add(pentry, x);
        for t:track_no in set[track_no]$elements(tracks) do
            if condition$queue(conditions[track_no]$get(aentry, x))  ∧
             (an ≠ ax  ∨  counts[track_no]$get(atreq, t) ≠ counts[track_no]$get(atent, t))
            then condition$wait(conditions[track_no]$get(conds, v)); end;
        end;
        an := an + 1;
        counts[track_no]$incr(atent, x);
        set[track_no]$insert(tracks, x);
        for z:track_no in conditions[track_no]$domain(aentry) do
            if condition$queue(conditions[track_no]$get(aentry, z)) then
                ok:boolean := true;
                for t:track_no in set[track_no]$elements(tracks) do
                    if (u < t < z  ∨  u > t > z)  ∧
                             (an ≠ ax  ∨  counts[track_no]$get(atreq, t) ≠
                                  counts[track_no]$get(atent, t))
                    then ok := false; end;
                end;
                if ok then condition$signal(conditions[track_no]$get(aentry, z)); end;
            end;
        end;
    end a_enter;

    a_exit = procedure(x:track_no);
        ax := ax + 1;
        u := x;
        for z:track_no in conditions[track_no]$domain(aentry) do
            if condition$queue(conditions[track_no]$get(aentry, z)) then
                ok:boolean := true;
                for t:track_no in set[track_no]$elements(tracks) do
                    if  (u < t < z  ∨  u > t > z)  ∧
                            (an ≠ ax  ∨  counts[track_no]$get(atreq, t) ≠
                                 counts[track_no]$get(atent, t))
                    then ok := false; end;
                end;
                if ok then condition$signal(conditions[track_no]$get(aentry, z)); end;
            end;
        end;
    end a_exit;

    an, ax := 0, 0;
    u := track_no$min();
    atreq := counts[track_no]$create();
    atent := counts[track_no]$create();
    aentry := conditions[track_no]$create();
    tracks := set[track_no]$create();
end dh;
```

Chapter 7

Detecting Erroneous Specifications

7.1 Introduction

The flexibility of the problem specification language makes it possible to specify a wide variety of synchronization constraints. Unfortunately, this flexibility also permits erroneous specifications to be constructed. Certain kinds of errors in specifications can be detected in attempting to derive equivalent solution specifications. As noted in Chapter 4, if a specification constrains when in a history, say, a request event can occur, this results in an invalid ordering being found in the derivation algorithm for which the offending event is of type request. Since this is erroneous, in that the underlying model requires events other than enter events to be unconditional, the derivation algorithm detects this error and fails to construct an equivalent solution specification.

There are other kinds of erroneous specifications, however, for which equivalent solution specifications can be derived. These specifications are compatible with the underlying model, but the synchronization constraints they specify display certain forms of undesirable behavior. Two such forms of behavior are the potential for deadlock and starvation. Deadlock results from a situation being overconstrained, so that each of a set of waiting processes is prevented from proceeding by the presence of all the rest. Starvation means that the constraint that is specified may be too rigid, in that certain processes are prevented from proceeding indefinitely.

A problem specification that manifests one of these forms of behavior results in the derivation of a solution specification that does likewise. However, the form of the solution specification makes the analysis required to detect these erroneous behaviors much more tractable than for the problem specification itself. This chapter presents algorithms for performing such analysis. They can be used, once a solution specification has been derived from a problem specification, as a check on the soundness of the original specification.

By the argument in Chapter 4 justifying the derivation algorithm, the set of histories allowed by a derived solution specification is exactly equal to the set allowed by the original problem specification. This means that a potential for deadlock or starvation cannot be introduced into the solution specification by the derivation itself, since if this were possible, then there would have to be one or more histories valid with respect to the problem specification but not to the solution specification. Rather, since the solution specification corresponds exactly to the problem specification in history-theoretic terms, any potential for deadlock or starvation in the problem specification is mirrored in the solution specification.

For example, a potential for deadlock would be reflected by the existence in a valid history of request events for which the corresponding enter events could never satisfy the specification. Assume the existence of such a history and its validity with respect to a solution specification. Then this same history must be valid with respect to the problem specification, and the enter events must fail to satisfy the problem specification, as well. Of course, the reverse is similarly true. Thus the solution specification must contain exactly the same potential for deadlock as the problem specification. In a similar way, starvation implies that there are valid histories in which the request and enter events for a particular operation activation are separated by an arbitrary number of other request-enter event pairs. For a problem specification and a solution specification that are valid with respect to exactly the same set of histories, starvation in one implies starvation in the other.

Since the solution specification is state-oriented, it is a convenient form on which to perform the analysis for these properties. The solution specification can be used to determine under what conditions, if any, deadlock and starvation are possible. Such a possibility, though, arises due to the original problem specification, and it is there that a correction must be made.

7.2 Deadlock detection

In a survey paper ([Hol72]) on the subject, *deadlock* is defined as "the situation in which one or more processes in a system are blocked forever because of requirements that can never be satisfied." In the context of this thesis, deadlock arises when a problem specification overconstrains the order of events in certain situations so as to prevent any of a group of requested accesses from ever occurring. The entry conditions in the derived solution specification form a basis for characterizing possible deadlock situations in terms of the synchronization state of the object. If deadlock is impossible, then each such characterization can be proved to lead to a contradiction.

The problem of deadlock detection has been studied fairly extensively, particularly for operating systems (e.g. [Hav68], [Hab69]) and database systems ([Cha74]). The bulk of this work has used a common scenario for deadlock: Each process in a collection of concurrently executing processes holds exclusive access to one or more scarce resources, and is blocked because of a request for resources held by other processes in the collection. The scarce resources are commonly viewed as devices in the case of operating systems, and locks in database systems. Unfortunately, shared abstract data objects are not really similar to peripheral devices, which are serially reusable and must be "owned" by one process at a time. Nor is the database paradigm of setting and releasing locks on parts of the database very applicable to most situations involving data abstractions. Blocking of processes competing for access to an abstract data object more often results from calls on particular operations of the abstraction, rather than the subcomponents of the data object they access.

Closer to the mark, from this point of view, is the work by Holt ([Hol71], [Rob75]). Using a Petri net-based model, Holt views a system as a set of states with transitions between them. With this approach, a process is "blocked" in a state when there is no transition it can make to another state. Deadlock results from a process being blocked in all reachable states of the system. The approach to be described in this section is similar.

The solution specification into which the specification is transformed is a convenient form on which to perform deadlock analysis. The control points at which processes can be blocked are the enter gates, and the conditions the processes are awaiting to become unblocked are the corresponding entry conditions. A deadlock corresponds to one or more processes waiting at each of one or more gates, on conditions that can never become true. (It is assumed throughout that all operation activations terminate, so that processes can deadlock only via the synchronization code itself.)

For example, consider a data abstraction with two operations p and q. Suppose that in deriving the solution specification from the problem specification, it is discovered that a condition for passing through the p^{enter} gate is

$$count(q^{request}) = count(q^{enter}).$$

Now suppose also that a condition for the q^{enter} gate is

$$count(p^{request}) = count(p^{enter}).$$

Obviously then, whenever there is a process waiting at each of the two gates p^{enter} and q^{enter}, these two processes are deadlocked. Each prevents the other from proceeding and thereby enabling the condition that it itself is awaiting. This means that the original problem specification is in error, in that the constraint it expresses prevents either activation in the given situation from ever proceeding.

In the general case, a necessary but not sufficient condition for a collection of processes to deadlock over access to a shared data object is for each of these processes to be waiting at an enter gate for a condition to be satisfied. Whether or not this situation is a potential deadlock depends on whether the conditions on which the processes are waiting can be enabled by subsequent events associated with the shared object caused by other active processes. The idea behind the deadlock analysis technique to be described here is to characterize the synchronization state of the object at a potential deadlock point, a point at which processes are waiting at enter gates. This characterization then contains sufficient information for determining whether the entry conditions can be enabled by other active processes, or whether the waiting processes themselves prevent the conditions from ever becoming satisfied, in which case the situation represents a deadlock.

Each potential deadlock situation is distinguished by the subset of enter gates in the system at which one or more processes are waiting. The terminology used here is that an operation is *blocked* if there are processes waiting at the associated enter gate to execute it. If there are n operations defined on an abstract data type, then there are $(2^n - 1)$ potential deadlock situations, since any subset of the operations may be blocked, except the empty subset. An empty set of blocked operations could not, of course, represent a deadlock situation.

A complication arises from the use of qualified gates in solution specifications. When there are two or more enter gates for a particular operation, with a different qualifying predicate on each, the easiest point of view to take is that they behave like gates controlling completely different operations. In the context of deadlock analysis, it is simplest to consider two qualifications of an operation p, $[p(v) \mid Q_1(v)]$ and $[p(v) \mid Q_2(v)]$, as if they were separate operations p_1 and p_2, since each distinct qualification of p can independently be blocked, just as different operations can. The catch is that the qualifying predicates Q_1 and Q_2 may not be independent, and if, for example, $Q_1 \supset Q_2$, then whenever $[p(v) \mid Q_1(v)]$ is blocked, $[p(v) \mid Q_2(v)]$ must be as well. In general, however, it is not always possible to determine when one qualified class is a subset of another. Always treating different qualifications of an operation as separate operations is a conservative approach which is guaranteed not to overlook any potential case of deadlock. Throughout this chapter, therefore, when reference is made to a data abstraction having n operations, the reader should understand that the intention is for different qualifications of an actual operation to be treated as separate operations.

It is straightforward to characterize a situation in which an operation is blocked. If C(p) is the condition for gate p^{enter}, then the condition of operation p being blocked is expressed by the formula B(p):

$$(\neg C(p)) \wedge count(p^{request}) > count(p^{enter}) \wedge count(p^{enter}) = count(p^{exit}).$$

That is, when p is blocked, there are no active executions of p, but one or more activations have been requested and are waiting because the entry condition C(p) is not satisfied.

Assume that the potential deadlock situations are numbered 1, 2, ... , $(2^n - 1)$, and let W_i be the set of blocked operations in situation i. Formula U_i will denote the characterization of the synchronization state of an object in situation i, by expressing the fact that all operations in W_i are blocked.

$$U_i = \wedge (B(p) \mid p \in W_i).$$

If U_i is equivalent to FALSE, then there is a contradiction in the information in the formula. This means that the potential deadlock situation is impossible, and that a condition on which an activation of one of the blocked operations is waiting must be satisfied. If U_i is not equivalent to FALSE, then it represents a characterization of the circumstances under which the situation can occur.

For a potential deadlock situation that is possible, the formula U_i can be used to determine whether or not the situation in fact represents an actual deadlock. This determination can be made by checking whether any of the conditions on which blocked operations are waiting involve operations that are not blocked in the given situation. If not, then the conditions can never become satisfied, and the situation in fact does represent a deadlock. If one or more conditions involve non-blocked operations, however, then there is not a deadlock, since a subsequent event involving one of these operations can "unblock" the situation and enable one of the waiting processes. At the very worst, such an event may change the situation to a *different* potential deadlock situation, to be analyzed separately. Therefore, it is sufficient to find a single non-blocked operation that is involved in the waiting conditions to disprove deadlock for a given situation.

As an example of deadlock analysis, consider the solution specification for a writers' priority database given in Section 6.3. Since there are two operations, "read" and "write", there are three potential deadlock situations--processes waiting only at the $\text{read}^{\text{enter}}$ gate, only at the $\text{write}^{\text{enter}}$ gate, and at both gates. In the first situation, $W(I) = \{ \text{read} \}$. The description of this situation U_1 is given by the "blocked" condition on the "read" operation, B(read):

$$(\text{count}(\text{write}^{\text{request}}) \neq \text{count}(\text{write}^{\text{enter}}) \;\vee\; \text{count}(\text{write}^{\text{enter}}) \neq \text{count}(\text{write}^{\text{exit}})) \;\wedge$$
$$\text{count}(\text{read}^{\text{request}}) > \text{count}(\text{read}^{\text{enter}}) \;\wedge\; \text{count}(\text{read}^{\text{enter}}) = \text{count}(\text{read}^{\text{exit}}).$$

The condition on which "read" activations are waiting involves events associated with the non-blocked operation "write". This is not an actual deadlock situation, since the "read" activations themselves are not causing the blocking. This does not necessarily mean that the processes blocked at the $\text{read}^{\text{enter}}$ gate will eventually proceed. There may exist histories in which these processes are blocked forever, i.e. they may face the possibility of starvation (see the next section). What the analysis here shows is that circumstances exist, involving possible future events associated with operation "write", that make unblocking of these processes possible. Their being blocked need not be a permanent condition for all possible histories.

The second situation is when only "write" is blocked, i.e. W(2) = { write }. The description here is U_2 = B(write):

$$(\mathrm{count}(\mathrm{read}^{enter}) \neq \mathrm{count}(\mathrm{read}^{exit}) \vee \mathrm{count}(\mathrm{write}^{enter}) \neq \mathrm{count}(\mathrm{write}^{exit})) \wedge$$

$$\mathrm{count}(\mathrm{write}^{request}) > \mathrm{count}(\mathrm{write}^{enter}) \wedge \mathrm{count}(\mathrm{write}^{enter}) = \mathrm{count}(\mathrm{write}^{exit}),$$

which can be simplified to

$$\mathrm{count}(\mathrm{read}^{enter}) \neq \mathrm{count}(\mathrm{read}^{exit}) \wedge$$

$$\mathrm{count}(\mathrm{write}^{request}) > \mathrm{count}(\mathrm{write}^{enter}) \wedge \mathrm{count}(\mathrm{write}^{enter}) = \mathrm{count}(\mathrm{write}^{exit}).$$

Since the blocking condition involves the non-blocked operation "read", this is also not an actual deadlock.

The third potential deadlock situation for the abstract object involves waiting readers and writers, so that W(3) = { read, write }. This situation is characterized by U_3 = (B(write) $\wedge$ B(read)):

$$(\mathrm{count}(\mathrm{read}^{enter}) \neq \mathrm{count}(\mathrm{read}^{exit}) \vee \mathrm{count}(\mathrm{write}^{enter}) \neq \mathrm{count}(\mathrm{write}^{exit})) \wedge$$

$$\mathrm{count}(\mathrm{write}^{request}) > \mathrm{count}(\mathrm{write}^{enter}) \wedge \mathrm{count}(\mathrm{write}^{enter}) = \mathrm{count}(\mathrm{write}^{exit}) \wedge$$

$$(\mathrm{count}(\mathrm{write}^{request}) \neq \mathrm{count}(\mathrm{write}^{enter}) \vee \mathrm{count}(\mathrm{write}^{enter}) \neq \mathrm{count}(\mathrm{write}^{exit})) \wedge$$

$$\mathrm{count}(\mathrm{read}^{request}) > \mathrm{count}(\mathrm{read}^{enter}) \wedge \mathrm{count}(\mathrm{read}^{enter}) = \mathrm{count}(\mathrm{read}^{exit}).$$

Here there is a contradiction, between the first disjunctive clause on the one hand, and the third and last conjuncts on the other. This reduces the formula to FALSE, proving the situation to be impossible. Since this disposes of all three potential deadlock situations, deadlock is proved to be impossible for this abstraction.

As a second example, consider the bounded buffer example analyzed in Section 6.2. Once again, there are two operations, and therefore three potential deadlock situations for this abstraction. The first is when only operation "rem" is blocked, so that W(1) = { rem }. This is described by the formula $U_1 = B(rem)$:

$$(count(dep^{exit}) \leq count(rem^{enter}) \vee count(rem^{enter}) \neq count(rem^{exit})) \wedge$$
$$count(rem^{request}) > count(rem^{enter}) \wedge count(rem^{enter}) = count(rem^{exit}),$$

which reduces slightly to:

$$count(dep^{exit}) \leq count(rem^{enter}) \wedge$$
$$count(rem^{request}) > count(rem^{enter}) \wedge count(rem^{enter}) = count(rem^{exit}).$$

Since the formula is not equivalent to FALSE, the situation is possible. However, the condition on which "rem" activations are waiting, namely

$$count(dep^{exit}) \leq count(rem^{enter}),$$

involves operation "dep" that is not blocked in the situation. This means that the condition need not be prevented from ever being satisfied, and so this does not represent an actual deadlock.

The second situation is when only "dep" is blocked, and W(2) = { dep }. The characterization of this situation is given by $U_2 = B(dep)$:

$$(count(rem^{exit}) \leq count(dep^{enter}) - N \vee count(dep^{enter}) \neq count(dep^{exit})) \wedge$$
$$count(dep^{request}) > count(dep^{enter}) \wedge count(dep^{enter}) = count(dep^{exit}),$$

which simplifies to:

$$\text{count}(\text{rem}^{\text{exit}}) \le \text{count}(\text{dep}^{\text{enter}}) - N \ \wedge$$

$$\text{count}(\text{dep}^{\text{request}}) > \text{count}(\text{dep}^{\text{enter}}) \ \wedge \ \text{count}(\text{dep}^{\text{enter}}) = \text{count}(\text{dep}^{\text{exit}}).$$

This formula also is satisfiable, but once again, the waiting condition involves a non-blocked operation, in this case "rem". This means that the potential for deadlock is averted.

The third inactive situation involves both "dep" and "rem" being blocked. $W_3 = \{$ dep, rem $\}$, and $U_3 = (B(\text{rem}) \ \wedge \ B(\text{dep}))$:

$$(\text{count}(\text{dep}^{\text{exit}}) \le \text{count}(\text{rem}^{\text{enter}}) \ \vee \ \text{count}(\text{rem}^{\text{enter}}) \ne \text{count}(\text{rem}^{\text{exit}})) \ \wedge$$

$$\text{count}(\text{rem}^{\text{request}}) > \text{count}(\text{rem}^{\text{enter}}) \ \wedge \ \text{count}(\text{rem}^{\text{enter}}) = \text{count}(\text{rem}^{\text{exit}}) \ \wedge$$

$$(\text{count}(\text{rem}^{\text{exit}}) \le \text{count}(\text{dep}^{\text{enter}}) - N \ \vee \ \text{count}(\text{dep}^{\text{enter}}) \ne \text{count}(\text{dep}^{\text{exit}})) \ \wedge$$

$$\text{count}(\text{dep}^{\text{request}}) > \text{count}(\text{dep}^{\text{enter}}) \ \wedge \ \text{count}(\text{dep}^{\text{enter}}) = \text{count}(\text{dep}^{\text{exit}}).$$

For any value of $N > 1$, this formula reduces to FALSE, since it implies that

$$\text{count}(\text{dep}^{\text{enter}}) = \text{count}(\text{dep}^{\text{exit}}) \le \text{count}(\text{rem}^{\text{enter}}) =$$

$$\text{count}(\text{rem}^{\text{exit}}) \le \text{count}(\text{dep}^{\text{enter}}) - N.$$

Therefore, the situation is impossible. In conjunction with the previous analysis of the other two situations, this means that no deadlock is possible for the "buffer" type.

7.3 Starvation detection

A related problem to deadlock is the notion of *starvation*. Starvation means that while a process that is waiting to access an object is not necessarily blocked permanently, a pattern of accesses exists that prevents the process indefinitely from proceeding. The opposite of starvation is *fairness*, which indicates that every process is guaranteed eventually to have its request for access fulfilled. A method analogous to that used for deadlocks can indicate a large class of possible starvation situations, specifically those that are independent of parameter values.

Unfortunately, not all starvation possibilities can be easily detected. For example, the disk head scheduler example of Example 14 in Section 2.7 is starvation-free, but the simplified version analyzed in Section 6.5 is not. The fairness of the former specification depends upon (1) the range of track numbers being bounded, and (2) the set of track numbers being well-ordered. The proof that these are sufficient conditions for fairness involves non-trivial properties of well-ordered sets. In general, properties related to activation parameters, specifically to predicates qualifying gates in the solution specification, involve analysis that is too complex for the relatively simple starvation detection method outlined here. Such properties do not cause similar problems for deadlock analysis, since there the issue is simply whether *any* activations of an operation can proceed under *any* circumstances. Starvation analysis must determine whether an arbitrary activation eventually can proceed under *all* circumstances. This means that interactions among different activations of an operation become more important. For those starvation possibilities that can be detected by the method to be presented, the same approach to qualified gates is taken as for deadlocks. Different qualifications of an operation are treated as distinct operations, and each is analyzed independently for starvation.

The motivation for the starvation analysis presented below is as follows: For a process to starve, it must be kept waiting indefinitely at the enter gate for some operation. Since the synchronization mechanism itself is fair in scheduling activations whose entry conditions are satisfied, this can only happen if the condition on which the process is waiting is never allowed to be satisfied, due to the presence of other operation activations. (As before, all operation activations are assumed to terminate.) Therefore, it must be possible for processes executing other operations of the data abstraction to *overtake* the waiting process. "Overtaking" refers to the fact that even though the given process is waiting at an enter gate, processes making other activations whose request events occur later proceed through their respective enter gates ahead of it.

If operation q cannot overtake operation p, then whenever an activation of p is blocked, eventually all activations of q that were requested prior to the request for p must be completed. Under circumstances in which the activation of p starves, therefore, no subsequent activation of q can proceed either. Thus the first step in the starvation analysis for a particular operation is to determine which other operations of the abstract data type can and cannot overtake it. The characterization of a starvation situation then states that the given operation is blocked, and that no "non-overtaking" operations are currently active. This characterization reduces to FALSE if there is a contradiction in the situation, meaning that starvation is impossible.

Formally, the method of analysis for each operation p is the following: As before, B(p) denotes that p is blocked:

$$(\neg C(p)) \wedge count(p^{request}) > count(p^{enter}) \wedge count(p^{enter}) = count(p^{exit}).$$

For all q ≠ p, construct the formula T(q, p) given by

$$B(p) \wedge C(q) \wedge (count(q^{request}) > count(q^{enter})).$$

This formula indicates under what circumstances a process executing operation q can overtake the process blocked at gate p^{enter}, i.e. when there are requested activations of q and the entry condition for q is satisfied. It T(q, p) is other than false, then it is possible for an activation of q to overtake the waiting activation of p. Therefore nothing can be assumed about operation q in a starvation situation for p. If T(q, p) reduces to FALSE, however, then this overtaking cannot occur, and a process waiting at p^{enter} will cause a process subsequently arriving at q^{enter} to be blocked as well. This means that no activations of q can be active in a starvation situation for p. The starvation condition S(p) is constructed by conjoining to B(p) the formula

$$count(q^{enter}) = count(q^{exit})$$

for each q for which T(q, p) is FALSE. That is,

$$S(p) = \wedge (count(q^{enter}) = count(q^{exit}) \mid T(q, p) = FALSE) \wedge B(p).$$

This indicates that since q cannot overtake p, eventually no executions of q will be active. If S(p) is FALSE, then starvation of processes attempting to execute p is impossible, in that the hypothesized starvation situation for p contains a contradiction. Otherwise, S(p) characterizes a possible starvation situation.

Again, consider the writers' priority database as an example. The condition for gate write^{enter} is

$$(\text{count}(\text{read}^{enter}) = \text{count}(\text{read}^{exit}) \wedge \text{count}(\text{write}^{enter}) = \text{count}(\text{write}^{exit})),$$

so the blocked condition for operation "write" is B(write):

$$\begin{gathered}(\text{count}(\text{read}^{enter}) \neq \text{count}(\text{read}^{exit}) \vee \text{count}(\text{write}^{enter}) \neq \text{count}(\text{write}^{exit})) \wedge \\ \text{count}(\text{write}^{request}) > \text{count}(\text{write}^{enter}) \wedge \text{count}(\text{write}^{enter}) = \text{count}(\text{write}^{exit}).\end{gathered}$$

The condition C(read) is given by

$$\text{count}(\text{write}^{request}) = \text{count}(\text{write}^{enter}) \wedge \text{count}(\text{write}^{enter}) = \text{count}(\text{write}^{exit}).$$

This makes the overtaking condition T(read, write):

$$\begin{gathered}(\text{count}(\text{read}^{enter}) \neq \text{count}(\text{read}^{exit}) \vee \text{count}(\text{write}^{enter}) \neq \text{count}(\text{write}^{exit})) \wedge \\ \text{count}(\text{write}^{request}) > \text{count}(\text{write}^{enter}) \wedge \text{count}(\text{write}^{enter}) = \text{count}(\text{write}^{exit}) \wedge \\ \text{count}(\text{write}^{request}) = \text{count}(\text{write}^{enter}) \wedge \text{count}(\text{write}^{enter}) = \text{count}(\text{write}^{exit}) \wedge \\ \text{count}(\text{read}^{request}) > \text{count}(\text{read}^{enter}).\end{gathered}$$

Since the second and fourth clauses contradict each other, the formula reduces to FALSE. This means that the clause

$$\text{count}(\text{read}^{enter}) = \text{count}(\text{read}^{exit})$$

is conjoined to B(write) to form S(write), the starvation condition for operation "write":

$$\begin{gathered}(\text{count}(\text{read}^{enter}) \neq \text{count}(\text{read}^{exit}) \vee \text{count}(\text{write}^{enter}) \neq \text{count}(\text{write}^{exit})) \wedge \\ \text{count}(\text{write}^{request}) > \text{count}(\text{write}^{enter}) \wedge \text{count}(\text{write}^{enter}) = \text{count}(\text{write}^{exit}) \wedge \\ \text{count}(\text{read}^{enter}) = \text{count}(\text{read}^{exit}).\end{gathered}$$

This formula in turn is FALSE, since the last two conjuncts together contradict the first disjunctive clause. Starvation of writers is therefore impossible.

If a similar analysis is performed for the "read" operation, B(read) is constructed as:

$$(\mathrm{count}(\mathrm{write}^{\mathrm{request}}) \neq \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \vee \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \neq \mathrm{count}(\mathrm{write}^{\mathrm{exit}})) \wedge$$
$$\mathrm{count}(\mathrm{read}^{\mathrm{request}}) > \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) \wedge \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{read}^{\mathrm{exit}}).$$

The condition of "write" overtaking "read", T(write, read), is then formed:

$$(\mathrm{count}(\mathrm{write}^{\mathrm{request}}) \neq \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \vee \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \neq \mathrm{count}(\mathrm{write}^{\mathrm{exit}})) \wedge$$
$$\mathrm{count}(\mathrm{read}^{\mathrm{request}}) > \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) \wedge \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{read}^{\mathrm{exit}}) \wedge$$
$$\mathrm{count}(\mathrm{read}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{read}^{\mathrm{exit}}) \wedge \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{write}^{\mathrm{exit}}) \wedge$$
$$\mathrm{count}(\mathrm{write}^{\mathrm{enter}}) > \mathrm{count}(\mathrm{write}^{\mathrm{exit}}).$$

This formula is not identically FALSE, however, so that operation "write" can indeed overtake "read". This means that the starvation condition S(read) is simply equal to the blocked condition B(read). Since S(read) is not FALSE, starvation of readers is indeed a possibility, as expected, and can take place under the circumstances given by:

$$(\mathrm{count}(\mathrm{write}^{\mathrm{request}}) \neq \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \vee \mathrm{count}(\mathrm{write}^{\mathrm{enter}}) \neq \mathrm{count}(\mathrm{write}^{\mathrm{exit}})) \wedge$$
$$\mathrm{count}(\mathrm{read}^{\mathrm{request}}) > \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) \wedge \mathrm{count}(\mathrm{read}^{\mathrm{enter}}) = \mathrm{count}(\mathrm{read}^{\mathrm{exit}}).$$

That is, as long as there are activations of "write" that are either requested and pending, or active, then requested activations of "read" may starve.

Chapter 8

Summary and Evaluation

8.1 Summary of the thesis

This thesis has explored one approach to the problem of specifying synchronization properties and synthesizing source language code to implement them. The approach taken has depended on a basic model of abstract data objects and synchronization, which was described in Chapter 2. The principal features of this model are:

(1) Every data object is strongly typed, and any access of the object must be via a basic operation of the type of the object.

(2) Certain points in time, called *events*, are distinguished in a computation history involving accesses of a given data object. In particular, there are three types of events: request events, which denote processes making known their wish to gain access to the object; enter events, which denote successfully gaining access; and exit events, which denote relinquishing access.

(3) The temporal precedence relation among events associated with a given data object is a total ordering relation.

(4) The function of synchronization is to constrain in certain ways the time ordering relation on a data object, in particular the occurrence of enter events within the total ordering. This function is orthogonal to the meaning of the operations by which processes access the object, and therefore can and should be implemented separately from those operations.

(5) Individual synchronization constraints exist for each object in the system. Furthermore, a synchronization constraint is associated with a data type, and applies independently to each object of that type.

Using this model as a basis, a specification language was described in Chapter 2 for expressing synchronization properties of abstract data types. A notation was devised for denoting events, and the infix symbol "$\Rightarrow$" introduced for the time ordering relation. Specifications express constraints on this relation via predicate calculus formulas involving the time ordering between universally quantified event expressions. The quantification causes the constraint to apply to all events of a given class in a history. By explicitly stating the arguments to procedure invocations involved in a specification and using predicates to constrain these arguments, a constraint on the $\Rightarrow$ relation can be made to selectively apply to a sub-class of events. The formal semantics of this specification language consisted of defining the validity of histories with respect to a given specification. A number of examples of the use of the language to express synchronization constraints appeared at the end of Chapter 2.

To synthesize source language code implementing the specifications, it was found to be desirable to use an intermediate form. This form, called the *solution specification*, was described in Chapter 3. It is an abstract representation of the solution to a specification that is procedural in nature but independent of the particular construct used for implementation. A solution specification consists of a collection of *gates*, which are abstract implementations of event classes. Synchronization constraints are implemented by attaching conditions on the synchronization state to gates for enter event classes. Processes are only allowed to pass through gates when the corresponding conditions are satisfied. The semantics of a solution specification, as of the problem specification, were defined in terms of the validity of histories. Translating a solution specification into an implementation using a synchronization construct such as a monitor is quite straightforward, as explained in Chapter 5. Therefore, the difficulty in synthesis is deriving the solution specification from the problem specification.

This derivation was the subject of Chapter 4. Besides simply identifying which gates are needed for a specification, this consists of constructing appropriate conditions on the synchronization state to associate with enter gates in order to implement the specified constraint. The construction of these conditions is accomplished by an algorithm that can be broken into several phases. First, constraints on the arguments to activations are incorporated into the rest of the specification by a technique called "qualification". Once this has been done, all possible orderings of relevant events are formed, and each ordering is identified as either valid or invalid with respect to the specification. The synchronization state at particular events in both valid and invalid orderings is characterized, and these characterizations are combined to form a preliminary condition. This condition is tested among the valid orderings; it either succeeds in satisfying them all and is therefore correct, or else it fails in one or more cases, and must be weakened by disjoining to it one or more other terms. These weakening terms are derived in much the same way as the preliminary condition, except that a smaller class of orderings is used, and the characterizations involve synchronization states saved at previous points in the orderings.

Chapter 6 presented several examples of commonly addressed synchronization problems, which are specified and then synthesized by the approach described. These examples certainly do not constitute a complete test of an approach, but they do represent a fairly broad range of the kinds of synchronization properties found to be of real interest. The topic of Chapter 7 was the analysis of a synchronization constraint for possible deadlock and starvation. The solution specification is a convenient form on which to perform this analysis. Algorithms were presented that for any given specification can disprove the possibility of certain kinds of deadlock or starvation, or derive the conditions under which they can take place.

8.2 The specification language

There are a number of ways of evaluating the specification language described in Chapter 2. The example specifications in Section 2.7 attest to its power to express a wide range of synchronization properties. The derivation method discussed in Chapter 4 and further illustrated by the examples of Chapter 6 demonstrates its suitability as an input language for the synthesis algorithm. Two other related criteria are especially important, though subjective in nature: the *constructability* of the language, how easy is it to write specifications; and its *comprehensibility*, how easy is it to understand specifications.

Within the framework of the model of synchronization upon which the language is based, the language itself is quite convenient for writing synchronization specifications. Since all of the standard logical operators of predicate calculus can be used, and formulas of arbitrary complexity constructed, any constraint on time ordering can be expressed. These specifications are relatively easy to write and to understand, since each logical operator has a natural interpretation. The extensibility of the language permits a complex specification involving many constraints to be expressed as a conjunction of individual clauses, each one specifying a single constraint. This feature, illustrated by the different versions of the readers-writers problem considered in Chapter 6, enhances both constructability and comprehensibility.

There may exist grounds for criticizing the language based on disagreements with the underlying model. For example, consider the choice of which points in time to be designated as events. Each of the three event types request, enter, and exit has a uniform meaning, and each is necessary for expressing a wide class of synchronization properties. Properties concerning exclusion of operations involve enter and exit events, and scheduling properties use request and enter events.

Disagreement may exist, however, over whether these three types constitute a sufficient set. In particular, assume that some operation p may be blocked from proceeding, not initially before the activation begins, but rather at some point in the middle of execution. That is, suppose p performs a certain amount of computation, then must wait for some synchronization condition to be satisfied, after which it completes execution with some further computation. There is no straightforward mechanism in the model (and therefore the language) for denoting this "intermediate" event. Such a situation must be handled by splitting operation p into two subsidiary operations pl and p2, which when executed serially constitute the whole of operation p. The intermediate point within p is represented by the exit event for pl and request event for p2. The condition on which it may be blocked is an entry condition for gate $p2^{enter}$.

While this may not be considered an aesthetically satisfying solution to the problem, it can be justified. The event types request, enter, and exit were chosen in part because they possess a uniform interpretation independent of the meaning of the particular operation. If a new event type intermediate were employed, its meaning (the intermediate point at which the operation may pause) necessarily would be operation-dependent. Moreover, a single intermediate event type would not be sufficient for handling operations that may be blocked at more than one intermediate point. For the sake of generality, then, it would be necessary to have an unbounded number of event types intermediate-1, intermediate-2, ... Whatever such an approach might gain in constructability of the language would surely be lost in reduced comprehensibility. The solution chosen instead of splitting the operation p into component segments p1, p2, etc. seems at least as satisfactory.

There is another important aspect of the specification language used here. That is the ability to use synchronization specifications, along with the bodies of the operations, to prove properties of the data abstractions. One kind of proof is of the (serial) correctness of an operation, with the synchronization specification used to show that all possibly interfering operation activations are excluded from concurrent execution. The synchronization specification also can be used to demonstrate that certain types of exception handling are unnecessary. An example is the bounded buffer specification analyzed in Section 6.2, by which it can be shown that an activation of the "rem" operation never operates on an empty buffer.

One limitation of the specification language is an inability to refer to the state of the abstract data object to which a specification applies. There are good reasons for restricting the language in this way, as explained in Chapter 2. It is also true, at least theoretically, that any state information can be expressed in terms of events in the history. However, capturing state information via histories can make the specification of certain properties rather awkward. For example, the disk head scheduling specification of Example 14 in Section 2.7 could be simplified significantly if reference could be made to whether the disk head is moving up or down (at the point at which a certain event occurs). This limitation, however, does serve the purpose of maintaining a clean separation between the synchronization aspect of the data abstraction and the actual operations.

8.3 The synthesis method

The method for synthesizing synchronization code from specifications was presented in Chapters 4 and 5. The justification of the algorithm for deriving a solution specification, and a discussion of cases for which it fails, is presented at the end of Chapter 4. Failures of the algorithm really reflect an inability of the relatively rigid solution specification to capture certain synchronization properties of interest. For example, the algorithm fails on the first-come-first-served specification because this property cannot be implemented using a separate queue for each operation of the abstraction. On the whole, though, and particularly with the use of qualified gates to capture parameter-related properties, the solution specification structure is able to express the solutions to almost all synchronization problems that can be specified in the problem specification language.

The monitor implementation of the solution specification is relatively straightforward in most cases. The exception to this is the handling of parameterized gates using the types counts[T] and conditions[T]. The implementation of parameterized enter gates in particular, especially where the qualifying predicate is not a functional relation, becomes quite complicated. As noted in Chapter 5, a certain amount of simplification would be possible if the user were to supply the range of values that each parameter could assume. This information could also be used to prevent the decrease in expressive power that results from having to make certain assumptions about the solution specification conditions in order to construct a correct implementation.

Chapter 6 contains a small set of examples in which implementations are completely synthesized from problem specifications. In fact, a considerably larger number of examples have been worked out, including all of the specifications presented as examples in Section 2.7, with the exception of those explicitly cited in Chapter 4 as failures. The method appears to satisfactorily synthesize implementations for a wide class of specifications, except for those properties for which solution specifications cannot be obtained, as noted above.

Two other measures of the synthesis method are important to discuss here. The first of these, the practicality of the synthesis algorithm, appears open to question. In the derivation of the solution specification, all possible orderings of the event expressions contained in the specification must be considered, and since n events may have as many as n! orderings, the algorithm is necessarily exponential. In a less formal sense, the practicality is weakened by the complexity of some of the steps of the algorithm, particularly those requiring a logical simplification of formulas. Compensating somewhat is the fact that the formulas involved are of a restricted form. Therefore, a small collection of special-case simplifications, such as those appearing in Figure 4.2, rather than the power of a general-purpose logical simplifier, would probably be sufficient for implementing a system based on the method proposed here. Also, the ability to analyze each conjunct of the specification separately helps reduce the overall complexity.

Still, improvements in the algorithm are required to make it practical in, say, a compiler. The algorithm as it stands can be used manually by a person to implement a synchronization constraint expressed in the specification language, or to informally check a hand-coded implementation. Further work, as discussed at the end of this chapter, is needed to automate the algorithm.

With respect to the other measure of evaluation, the efficiency of the synthesized source code, the method can be judged to be quite respectable. There are certain inefficiencies that necessarily result from the use of a relatively fixed structure. Two aspects of the fixed structure here are particularly restrictive. One is the use of separate condition variables for different enter gates, which prevents the queuing of processes waiting to execute different operations on a common queue. The other is the derivation of a single entry condition applicable both initially when a process first attempts an access and subsequently when testing whether to allow the deferred access.

As a result, the synthesized monitor for the "alternating priority database" example of Section 6.4 is awkward compared to the rather elegant monitor coded by hand to solve the same problem in [Hoa74]. Much of this awkwardness, however, is due to the simple-minded implementation of testing for possible signalling all condition variables at the end of each monitor procedure. As indicated in Chapter 5, optimization of the signalling statements by eliminating provably unsatisfiable options is often possible.

On the whole, synthesized implementations approach hand-coded ones in terms of efficiency for a large class of problems. The fact that all synchronization code manipulates only integer-valued quantities, and that entry conditions always consist of linear equalities or inequalities of such quantities, keeps the implementations efficient. The efficiency can be enhanced if other obvious optimizations are applied to the results of the straightforward synthesis, such as using a single variable for a quantity of the form

count(ecl) - count(ec2),

rather than two separate variables for the two different counts.

Where the efficiency of the synthesized code becomes unacceptable is in cases involving parameterized gates, such as the disk head scheduler of Section 6.5. In order to accommodate the structure of the solution specification, the parameterized types counts[T] and conditions[T] must be employed to implement what amount to entire arrays of counts and conditions. Here, the fixed structure of the synthesized implementations becomes a real barrier to an efficient implementation, since "good" implementations of such properties make use of special mechanisms such as priority queues. With the exception of parameter-related properties, though, the performance penalties paid for most specifications seem to be within the limits of what can be reasonably expected from an automatic synthesis system.

8.4 Comparison with path expressions

As noted in the introductory chapter, the work on path expressions ([Cam74], [Hab75], [Flo76]) most nearly matches this thesis in terms of overall goals. In evaluating the thesis, then, it is instructive to compare it with the path expression work to see to what extent each meets these shared goals. In terms of this comparison, the path expression language is restricted to its original description in [Cam74]. Later versions have added successively more features to the language, with questionable results. The original language simply contains the basic features that make path expressions analogous to regular expressions, namely the sequencing operator " ; ", the alternation operator " , ", and the repetition operators "{ ... }" and "*path* ... *end*". The analogy with regular expressions embodies the basic philosophy underlying path expressions.

The approach both of this thesis and of path expressions is to constrain the ordering relation on accesses to some shared abstract data object. Access of the abstract object is limited to a collection of basic operations associated with the type of the object, and so each language specifies a subset of possible object histories involving these operations to be valid. For path expressions, activations of the operations are treated as units, while this thesis has denoted three particular points in time associated with each activation as events, and dealt with these events rather than the activation itself.

The path expression approach is to specify a global constraint for the complete sequence of accesses represented by the overall history. The specifications of this thesis, on the other hand, represent local constraints for individual operation activations; because the activations involved in a specification are quantified, the constraints apply individually to each activation in the history. My intuition is that local constraints are inherently simpler, both to construct and to comprehend, and that people must translate global constraints into local ones to understand them. This is a subjective judgement, however.

The path expression language uses as basic notions the concepts of mutual exclusion, sequencing, and concurrent repetition. These are at a higher level than the more primitive temporal ordering relation ⇒. Use of such higher-level concepts facilitates the expression of properties that are based closely on them. For example, the readers-writers property, appearing as Example 3 in Section 2.7 in the form

$$((\mathrm{write}_i^{\mathrm{enter}} \Rightarrow \mathrm{write}_j^{\mathrm{enter}}) \supset (\mathrm{write}_i^{\mathrm{exit}} \Rightarrow \mathrm{write}_j^{\mathrm{enter}})) \wedge$$
$$((\mathrm{write}_i^{\mathrm{exit}} \Rightarrow \mathrm{read}_k^{\mathrm{enter}}) \vee (\mathrm{read}_k^{\mathrm{exit}} \Rightarrow \mathrm{write}_i^{\mathrm{enter}})),$$

can be specified by the path expression

$$\textbf{\textit{path}}\ \{\ \mathrm{read}\ \},\ \mathrm{write}\ \textbf{\textit{end}}.$$

The gain in comprehensibility and constructability is obvious.

However, the same result can be achieved by using some sort of macro facility with the language of this thesis. For example, MUTEX(p, q) could be employed as a shorthand abbreviation for the mutual exclusion specification of Example 2 in Section 2.7:

$$(p_i^{\mathrm{exit}} \Rightarrow q_j^{\mathrm{enter}}) \vee (q_j^{\mathrm{exit}} \Rightarrow p_i^{\mathrm{enter}}),$$

and the readers-writers property then could be expressed as

$$\mathrm{MUTEX}(\mathrm{write}, \mathrm{read}) \wedge \mathrm{MUTEX}(\mathrm{write}, \mathrm{write}).$$

Such a macro facility would also be useful in identifying specifications for which implementations have already been derived in the past, thus eliminating replication of previous effort.

The use of higher-level concepts as basic to the path expression language has the disadvantage that properties not closely related to these basic ones can be rather difficult to specify. For example, consider the writers' priority database example analyzed in Section 6.3. There the property was specified by adding to the readers-writers specification above the following conjunct, giving priority to operation "write" over "read":

$$(\mathrm{write}_i^{\mathrm{request}} \Rightarrow \mathrm{read}_j^{\mathrm{enter}}) \supset (\mathrm{write}_i^{\mathrm{enter}} \Rightarrow \mathrm{read}_j^{\mathrm{enter}}).$$

The path expression specification for the same example appears in [Cam74] as:

```
path readattempt end
path requestread, { requestwrite } end
path { openread; read }, write end
where
    readattempt = begin requestread end
    requestread = begin openread end
    requestwrite = begin write end
    READ = begin readattempt; read end
    WRITE = begin requestwrite end
```

There is quite a lot of extra effort involved in adding the single property of priority to the readers-writers specification, and in terms of comprehensibility it leaves much to be desired. Even more discouraging is the fact that giving priority to "read" over "write" is done in a slightly different manner. Little wonder, then, that in the next version of path expressions, appearing in [Hab75], priority becomes another pre-defined operator in the specification language.

The languages of both this thesis and path expressions claim the virtue of extensibility, meaning that further constraints simply can be added onto previous ones without changing the existing specification. As the above example illustrates, this is not quite true of path expressions, since the addition of the writers' priority property requires a change in the expression of the readers-writers property as well. In this thesis, new constraints can always be conjoined to existing ones.

The writers' priority database example also illustrates the fact that with path expressions new operations sometimes must be invented for the specification of desired properties. In this thesis, this is also true, but here it is limited to the single category of breaking an operation into serial sections of code in between which the process executing the operation may be blocked, as explained in Section 8.2. With path expressions, blocking within operations must be handled in the same way, in fact. However, it may also be necessary to construct a new operation whose only purpose is to call an existing one, such as "requestwrite" in the example. Other examples in both [Cam74] and [Hab75] contain numerous other such "hidden" operations used in various ways. In general, a clean separation of synchronization code from the data abstraction operations themselves seems less feasible with path expressions.

The final comparison with respect to the specification languages themselves is that path expressions contain no facility for expressing properties that involve the parameters of operation activations. The only way to handle such properties would appear to be for the operation body to call different hidden procedures based on the satisfaction of different predicates by the parameters. Path expressions could then express synchronization constraints on these hidden procedures. There is no straightforward mechanism, however, as there is in the language of the thesis.

The main thrust of the discussion in this section so far has been that the specification language of this thesis is superior, particularly in terms of criteria such as constructability and comprehensibility, to the path expression language. With respect to synthesis, however, there is no question that the path expression approach is better. A simple recursive algorithm in [Cam74] can automatically implement any constraint specified by path expressions in terms of semaphores and integer counters.

In general, there is a tradeoff between expressive power of a specification language, and relative ease of synthesizing implementations from it. Because the path expression language is designed around a few built-in properties such as mutual exclusion, "canned" implementations of these properties can simplify the task of synthesis. The greater generality of the language of this thesis results in a far more difficult synthesis problem. It is interesting that in later versions of the path expression language ([Hab75], [Flo76]), additional features are added to increase the expressive power. These later papers do not include automatic implementation algorithms, and the problem of synthesis would appear far more difficult for these more complicated versions of the language.

8.5 Future work

There are a number of areas in which the work of this thesis could be extended in the future. Generally, the specification language itself seems sound as it stands, with the possible exception of the inability to refer to the data state of the resource, which is an issue that should be investigated. Further work is also needed on using specifications in proving properties of data abstractions.

As noted in Chapter 5, information about the range of values of certain parameters would be very helpful in constructing implementations of argument-related properties. An automated system could interactively ask for this information from the user. However, it could also be supplied as part of the original specification, if the specification language were extended to handle it.

The synthesis method described here can only be viewed as a starting point for pursuing this general approach. The synthesis algorithm is very complicated, and while this is dictated to some extent by the generality of the specification language, the complexity almost certainly could be reduced, perhaps dramatically, by looking at alternative strategies.

One area that could particularly benefit from a different approach is the use of qualified gates for argument-related properties. As indicated above in Section 8.3, the implementations resulting from such cases are unacceptably inefficient. It is unreasonable to have to perform a detailed search in determining the state variable to be updated or the condition on which to wait. A change in the basic solution specification structure would probably be necessary to achieve acceptably efficient implementations of argument-related properties. Unless some alternative were found, it might be better to eliminate argument-related predicates from the specification language entirely, even at the cost of reducing the power of the language.

The use of information private to each process, as discussed in Section 4.7, represents one possible direction for extending the power of the solution specification. Private information would permit each process to look back in the history to a point whose state is important only to that process. This would increase the range of applicability of the derivation algorithm. Of course, adding this feature to the solution specification requires modification of the algorithm so that such information can be derived. This issue would have to be investigated.

An alternative to private information would be a more flexible solution specification structure. As noted in Section 8.3, the ability to employ different queuing strategies and to have different entry conditions for a gate depending upon context would add expressive power to the solution specification. Again, the impact on the derivation algorithm would have to be considered.

Another idea that might bear exploring is the use of more powerful data types than simple integers in both the solution specification structure and the source code implementation. Specifically, sequences of events may be a more natural concept by which to translate properties from history-theoretic to state-theoretic terms. One potential difficulty is the fact that there is no theory of sequences as rich as number theory, and no good analogue for sequences to the < relation on integers, which is so basic to the synthesis algorithm. Also, the problem of source-level optimizations, which has been addressed briefly in the thesis, would become much more serious.

A limitation of the work here that has been mentioned earlier is its dependence on a centralized synchronization mechanism for each data object. This limits its applicability in situations where data objects may be distributed widely around a system of geographically distant processors. It would be interesting to explore to what extent this centralized-control bias is built into the underlying model, and see what problems have to be overcome in devising an implementation suitable for distributed systems.

An interesting problem growing out of the approach here is whether or not synchronization constraints for an abstract data type can be derived automatically from the implementation of the type. Obviously, questions such as whether one operation should have priority over another can only be decided by a person, since there is no inherent reason to choose one priority scheme over another. However, the code implementing the operations of a type, possibly augmented by some internal consistency requirement for the lower-level representation of objects of the type, can provide enough information to determine many classes of synchronization constraints. Which operations must be mutually exclusive of each other can often be determined by analyzing the manipulation of shared variables used in the implementation of the type. A number of techniques employed in optimizing compilers can also be used: Heuristics such as dead code elimination and requiring a variable to be initialized before being used can reveal certain required dependencies in the ordering of operations. Success in investigating this area could lead to the partial elimination of the need for synchronization code itself.

Of all the areas open for future work, however, the most obvious is the need to implement in an actual system a method such as the one described in this thesis. Many ideas look good on paper, only to founder when actually put into practice. A certain amount of system design has been done on paper, in order to help determine the feasibility of the system. Nothing has been actually run and tested, however, and only an actual implementation ultimately can be convincing as to the feasibility of automatic synthesis of synchronization code.

Bibliography

[Blo78] Bloom, T., "Synchronization Mechanisms for Data Abstractions", M. S. thesis (forthcoming), M. I. T., 1978.

[Bri72] Brinch Hansen, P., "A Comparison of Two Synchronizing Concepts", Acta Informatica 1, pp. 190-199.

[Bri73] Brinch Hansen, P., Operating System Principles, Prentice-Hall, Englewood Cliffs, N. J., 1973.

[Bro76] Brock, J. D., and Laventhal, M. S., unpublished note.

[Cam74] Campbell, R. H., and Habermann, A. N., "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science, Vol. 16, Springer Verlag, Heidelberg, 1974.

[Cha74] Chamberlin, D. D., Boyce, R. F., and Traiger, I. L., "A Deadlock-Free Scheme for Resource Locking in a Data-Base Environment", Information Processing 74, North-Holland, Amsterdam, 1974, pp. 340-343.

[Cou71] Courtois, P. J., Heymans, F., and Parnas, D. L., "Concurrent Control with 'Readers' and 'Writers'", Comm. ACM 14, 10, pp. 667-668.

[Dah72] Dahl, O. J., "Hierarchical Program Structures", Structured Programming, Academic Press, New York, 1972.

[Dij68] Dijstra, E. W., "Cooperating Sequential Processes", Programming Languages, Academic Press, New York, 1968.

[Dij72a] Dijstra, E. W., "Notes on Structured Programming", Structured Programming, Academic Press, New York, 1972.

[Dij72b] Dijstra, E. W., "Hierarchical Ordering of Sequential Processes", Operating Systems Techniques, Academic Press, New York, 1972.

[Dij75] Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", Comm. ACM 18, 8, pp. 453-457.

[Esw76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database System", Comm. ACM 19, 11, pp. 624-633.

[Flo76] Flon, L., and Habermann, A. N., "Towards the Construction of Verifiable Software Systems", Proc. ACM Conference on Data, SIGPLAN Notices 8, 2, pp. 141-146.

[Ges77] Geschke, C. M., Morris, J. H., and Satterthwaite, E. H., "Early Experience with Mesa", Comm. ACM 20, 8, pp. 540-553.

[Gre75] Greif, I., "Semantics of Communicating Parallel Processes", MAC-TR-154, M.I.T. Project MAC, 1975.

[Gri76] Griffiths, P., "SYNVER: An Automatic System for the Synthesis and Verification of Synchronous Processes", Ph. D. thesis, Harvard University, 1976.

[Hab69] Habermann, A. N., "Prevention of System Deadlocks" Comm. ACM 12, 7, pp. 373-377.

[Hab72] Habermann, A. N., "Synchronization of Communicating Processes", Comm. ACM 15, 3, pp. 171-176.

[Hab75] Habermann, A. N., "Path Expressions", Carnegie-Mellon University, 1975.

[Had77] Haddon, B. K., "Nested Monitor Calls", Operating System Review 11, 4, pp. 18-23.

[Hav68] Havender, J. W., "Avoiding Deadlock in Multi-Tasking Systems", IBM Systems J. 7, 2, pp. 74-84.

[Hew73] Hewitt, C., Bishop, P., and Steiger, R., "A Universal Modular Actor Formalism for Artificial Intelligence", Proc. IJCAI, 1973.

[Hew77] Hewitt, C., and Atkinson, R., "Parallelism and Synchronization in Actor Systems", Proc. ACM Conference on Principles of Programming Languages, 1977.

[Hoa74] "Monitors: An Operating System Structuring Concept", Comm. ACM 17, 10, pp. 549-557.

[Hol71] Holt, R. C., "On Deadlock in Computer Systems", CSRG Technical Report 6, Department of Computer Science, University of Toronto, 1971.

[Hol72] Holt, R. C., "Some Deadlock Properties of Computer Systems", ACM Computing Surveys 4, 3, pp. 179-196.

[Jam77] Jammel, A. J., and Stiegler, H. G., "Managers versus Monitors", Information Processing 77, North-Holland, Amsterdam, 1977, pp. 827-830.

[LDRS77] Proceedings of ACM Conference on Language Design for Reliable Software", SIGPLAN Notices 12, 3.

[Lis74] Liskov, B., and Zilles, S., "Programming with Abstract Data Types", SIGPLAN Notices 9, 4, pp. 50-59.

[Lis77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU", Comm. ACM 20, 8, pp. 564-576.

[McC62] McCarthy, J., "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Systems, North-Holland, Amsterdam, pp. 33-70.

[Owi75] Owicki, S. S., "Axiomatic Proof Techniques for Parallel Programs", TR75-251, Cornell University, 1975.

[Owi76] Owicki, S. S., "An Axiomatic Proof Technique for Parallel Programs II: Shared Data Abstractions", Stanford University, 1976.

[Par72] Parnas, D. L., "A Technique for Software Module Specification with Examples", Comm. ACM 15, 5, pp. 330-336.

[Ree77] Reed, D. P., and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers", M.I.T., 1977.

[Rob75] Robinson, L., and Holt, R. C., "Formal Specifications for .Solutions to Synchronization Problems", Stanford Research Institute, 1975.

[Sch78] Schaffert, J. C., "A Formal Definition of CLU", MIT/LCS/TR-193, M.I.T. Laboratory for Computer Science, 1978.

[Sha77] Shaw, M., Wulf, W. A., and London, R. L., "Abstraction and Verification in Alphard", Comm. ACM 20, 8, pp. 553-564.

Biographic Note

Mark Steven Laventhal was born on November 14, 1950, in Englewood, New Jersey. He grew up in Bergenfield, New Jersey, in Detroit, Michigan, and in Broomall, Pennsylvania. He graduated from Marple-Newtown High School in Newtown Square, Pennsylvania, in 1968. From 1968 to 1978, Mr. Laventhal has attended the Massachusetts Institute of Technology. He received the S. B. and S. M. degrees in the Department of Electrical Engineering and Computer Science in February, 1974. His S. M. thesis was entitled "Verification of Programs Operating on Structured Data". From 1972 through 1975, Mr. Laventhal received a National Science Foundation Graduate Fellowship. He served as a teaching assistant in the Department of Electrical Engineering and Computer Science from September, 1975, through January, 1977, and as a research assistant under Professor Barbara Liskov from January, 1977 through June, 1978.

Mr. Laventhal worked at the Thomas J. Watson Research Center of I. B. M. Corporation in Yorktown Heights, New York, during the summers of 1976 and 1977. He is a member of the Association for Computing Machinery, including its Special Interest Groups on Programming Languages and Software Engineering. He is also a member of the Tau Beta Pi engineering honorary society, and the Eta Kappa Nu electrical engineering honarary society.

In August, 1978, Mr. Laventhal will assume a position with the Data Systems Division of Hewlett-Packard Corporation in Cupertino, California. He is married to Carol J. Goodman.

CS-TR Scanning Project
Document Control Form

Date : 10/12/95

Report # LCS-TR-203

Each of the following should be identified by a checkmark:
Originating Department:

□ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) □ Technical Memo (TM)
□ Other:

Document Information

Number of pages: 236(245-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
□ Single-sided or
☒ Double-sided

Intended to be printed as :
□ Single-sided or
☒ Double-sided

Print type:
□ Typewriter □ Offset Press □ Laser Print
□ InkJet Printer ☒ Unknown □ Other:

Check each if included with document:

☒ DOD Form (2) □ Funding Agent Form ☒ Cover Page
☒ Spine ☒ Printers Notes □ Photo negatives
□ Other:

Page Data:

Blank Pages (by page number):

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:

UN# UN#
IMAGE MAP: (1-236) UN#'ED TITLE & BLANK, 2, BLANK, 3 BLANK,
4-16, UN#BLK, 17-41, UN#BLK, 42-62, UN#BLK,
63-189, UN#BLK, 190-224, UN#BLK, 225-228
(237-245) SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, DOD(2), TRGTS(3)

Scanning Agent Signoff:
Date Received: 10/12/95 Date Scanned: 10/30/95 Date Returned: 11/2/95

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
MIT/LCS/TR-203

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Synthesis of Synchronization Code for Data Abstractions

5. TYPE OF REPORT & PERIOD COVERED
Ph.D.Thesis - June 23, 1978

6. PERFORMING ORG. REPORT NUMBER
MIT/LCS/TR-203

7. AUTHOR(s)
Mark S. Laventhal

8. CONTRACT OR GRANT NUMBER(s)
N00014-75-C-0661
DCR74-21892

9. PERFORMING ORGANIZATION NAME AND ADDRESS
MIT/Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Advanced Research Projects Agency/Associate Program Director
Department of Defense /Office Computing Activities
1400 Wilson Boulevard /National Science Foundation
Arlington, VA 22209 /Washington, D.C. 20550

12. REPORT DATE
June 1978

13. NUMBER OF PAGES
231

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Office of Naval Research
Department of the Navy
Information Systems Program
Arlington, VA 22217

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

synchronization	interprocess communication
synthesis	monitors
data abstractions	deadlock
abstract data types	starvation
concurrency	

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

Synchronization code is necessary to control shared access of an abstract data object in a parallel-processing environment. This thesis explores an approach in which a synchronization property can be specified in a high-level nonprocedural language, and an implementation for the specified property can be synthesized algorithmically. A problem specification language is introduced in which synchronization properties can be expressed in a structured but natural manner. A method is then presented for synthesizing an implementation. An

20. intermediate form, called a solution specification, is first derived, representing an abstract solution to the problem. The derivation of the solution specification accomplishes the transformation of the specification from nonprocedural to procedural form. The solution specification can be translated directly into a source language synchronization mechanism, such as a monitor.

Specifications for common synchronization properties, such as the readers-writers and bounded buffer problems, are expressed in the problem specification language. Corresponding implementations are then synthesized for these problems. In addition, the derived solution specification can be used in analyzing the soundness of the original problem specification with respect to criteria such as freedom from deadlock and starvation.