Synchronization Mechanisms for Modular Programming Languages

by

Toby Bloom

January, 1979

© Massachusetts Institute of Technology

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 MCS74-21892.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts

02139

This empty page was substituted for a blank page in the original document.

Synchronization Mechanisms for Modular Programming Languages

by

Toby Bloom

Submitted to the Department of Electrical Engineering and Computer Science

on January 26, 1979, in partial fulfillment of the requirements

for the degree of Master of Science

ABSTRACT

Any programming language that supports concurrency needs a synchronization construct with which to express access control for shared resources. This thesis examines synchronization constructs from the standpoint of language design for reliable software. The criteria a synchronization mechanism must satisfy to support construction of reliable, easily maintainable concurrent software are defined. Some of these criteria, such as expressive power, can be defined only with respect to the set of problems the mechanism is expected to handle. A definition of the range of problems considered to be synchronization problems is therefore needed. Such a definition is provided by describing the possible types of constraints that may be imposed on access to shared resources. We then use this taxonomy of synchronization constraints to develop techniques for evaluating how well synchronization constructs meet the criteria discussed. These techniques are then applied to three existing synchronization mechanisms: monitors, path expressions, and serializers. Evaluations are presented, and the three mechanisms compared.

Thesis Supervisor: Barbara H. Liskov

Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: synchronization, concurrency, modularity, data abstractions, programming methodology

ACKNOWLEDGMENTS

I would like to express my appreciation to Professor Barbara Liskov, my thesis supervisor, for her guidance in my research, as well as for her patience and support while I was writing this thesis.

Craig Schaffert provided many useful insights and helped in clarifying many of the ideas in the thesis. Russ Atkinson and Mark Laventhal provided a great deal of encouragement and support, as well as technical help; without them I would have given up long ago.

I would especially like to thank Tim Anderson for reading the many drafts of this thesis, and for helping to put it into readable form. Bob Scheifler also helped edit the final draft.

Finally, I would like to thank my roommates, Margery Colten, Vicki Bier, and Dale Hattis, for giving me the encouragement I needed, and for putting up with me while I was writing this thesis.

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 MCS74-21892.

CONTENTS

1. Introduction 7

1.1 Background and Motivation 7
1.2 Research Goals and Outline of the Thesis 8
1.3 Related Work 9

2. Criteria and Evaluation Techniques 11

2.1 Modularity 12
2.2 Categorizing Synchronization Problems 13
2.3 Expressive Power 20
2.4 Ease of Use 22
2.5 Modifiability 24
2.6 Correctness 25
2.7 Summary 26

3. Monitors 28

3.1 Expressive Power 31
3.2 Modularity 44
3.3 Ease of Use and Modifiability 49
3.4 Correctness 53
3.5 Conclusions 54

4. Path Expressions 55

4.1 Expressive Power 58
4.2 Modularity 75
4.3 Ease of Use and Modifiability 80
4.4 Correctness 82
4.5 Conclusions 84

5. Serializers 85

5.1 Mechanism Description 85
5.2 Expressive power 89
5.3 Modularity 99
5.4 Ease of Use and Modifiability 101
5.5 Correctness 104
5.6 Conclusions 106

6. Summary and Evaluation 107

6.1 Summary and Conclusions 107
6.2 Evaluation and Extensions of this Work 109

Appendix I. Specification of Synchronization Problems 111

FIGURES

Figure 1. Bounded Buffer using Monitors 31
Figure 2. Readers_Priority Monitor 33
Figure 3. Readers_Priority Protected_Resource Module 35
Figure 4. First_Come_First_Serve Monitor 36
Figure 5. Writers_Exclude_Others Monitor 39
Figure 6. Alarmclock Monitor 40
Figure 7. Disk_Scheduler Monitor 42
Figure 8. Protected Resource Structure 45
Figure 9. Bounded Buffer Monitor 48
Figure 10. Writers_Priority Monitor 51
Figure 11. Fair_Readers_Priority Monitor 52
Figure 12. First_Come_First_Serve using Path Expressions 60
Figure 13. Readers_Priority Database using Path Expressions 63
Figure 14. Alarmclock 67
Figure 15. One_Slot Buffer using Path Expressions 70
Figure 16. Bounded Buffer using Path Expressions 71
Figure 17. First_Come_First_Serve Synchronization Module 77
Figure 18. Hierarchical Deadlock in Path Expressions 78
Figure 19. Writers_priority Database using Path Expressions 81
Figure 20. Structure of Serializer Objects 85
Figure 21. First_Come_First_Serve Serializer 88
Figure 22. Writers_Exclude_Others Serializer 90
Figure 23. Readers_Priority Serializer 91
Figure 24. Bounded Buffer Serializer 92
Figure 25. One_Slot Buffer Serializer 94
Figure 26. Disk Scheduler Serializer 96
Figure 27. Comparison of Monitor and Serializer Structures 100
Figure 28. Fair_Readers_Priority Serializer 103

1. Introduction

1.1 Background and Motivation

In recent years there has been great interest in development of high-level language constructs to support parallel programming. Numerous synchronization constructs have been proposed since Dijkstra introduced the semaphore[12]. These include conditional critical regions[5], monitors[18,7], path expressions[8], and serializers[3].

In addition, we have come to realize the importance of the role programming languages play in the development of reliable, high quality software. Languages that support good program structure significantly enhance programmer effectiveness in producing reliable software. One methodology for improving software quality is the use of modular programming techniques and abstraction mechanisms. Languages such as CLU[25] and Alphard[35] support this methodology.

The need for reliable, easily maintainable software is even greater when concurrency is involved. Parallel programs are more complex and harder to understand than sequential ones because processes interact more, and time-dependent errors, which are not susceptible to traditional debugging techniques, are much more likely. It is therefore imperative that the language constructs used to implement parallelism support good program design.

While a synchronization mechanism that supports modular programming and the use of data abstractions would certainly contribute to the reliability and quality of concurrent software, no clear description of the requirements that such a mechanism must satisfy has been established. Attempts to evaluate existing synchronization mechanisms usually depend on the rather ad hoc technique of attempting to implement numerous synchronization schemes using the mechanism. Unfortunately, one can never tell, when using this method, whether the analysis is complete. If the analysis reveals a weakness in the mechanism, the construct is modified or extended to handle the one case found. The result has been the development of

numerous constructs, each designed to correct one flaw in a previous version, with no standard criteria for deciding when a mechanism is satisfactory.

The aim of this thesis is to state as explicitly as possible the criteria a synchronization mechanism must meet if it is to support construction of reliable, well-structured concurrent software. We will develop techniques to evaluate how well mechanisms satisfy these requirements. The criteria and evaluation techniques presented can then be used, not only to evaluate existing mechanisms, but as a basis for defining new mechanisms.

1.2 Research Goals and Outline of the Thesis

Our intention is to develop a methodology for evaluating the effectiveness of synchronization mechanisms in supporting the development of quality concurrent software. The first step in this process is to identify the function synchronization mechanisms serve in programming languages, that is, we must identify the class of problems to which these mechanisms will be applied. We accomplish this in Chapter 2 by developing a taxonomy of the synchronization constraints.

The first criterion we establish is that a mechanism be able to express straightforward solutions to any problem that can be defined in terms of the constraints described. A mechanism is said to have sufficient expressive power if it satisfies this property. Any construct designed to support reliability must satisfy certain other basic criteria also. These include ease of use, modifiability, modularity, and correctness. None of these has a precise definition, and we must decide how each applies to synchronization. In the remainder of Chapter 2, we define these criteria with respect to synchronization and develop techniques for assessing how well each is supported by a given construct.

In Chapters 3, 4 and 5, we examine three synchronization mechanisms: monitors, path expressions and serializers. The use of each mechanism is illustrated by a set of examples chosen to represent each class in our taxonomy of synchronization constraints. These examples

are then used in applying the evaluation techniques developed in Chapter 2. This analysis indicates whether a given mechanism satisfies our requirements and can be incorporated into a language designed to support software reliability without undermining the goals of that language. Furthermore, it provides information as to which problems can be easily implemented using a given mechanism.

1.3 Related Work

Most of the research directly related to this thesis has been mentioned in the previous sections. It falls into two basic categories: the development of synchronization constructs for high-level languages, and evaluations of these mechanisms.

The monitor construct was developed independently by Hoare[18] and Brinch Hansen[7] as an extension of Dijkstra's secretary concept[13].

The path expression mechanism was first developed by Habermann and Campbell[8], and has since been extended and modified several times [15, 14]. The mechanism is intended to provide a means of specifying synchronization non-procedurally, as a set of relationships among the operations used to access the shared resource. It thus appears to be a higher level construct than monitors.

Serializers are the most recent of the mechanisms discussed. They are based on the monitor mechanism and were developed by Atkinson and Hewitt[3] to eliminate certain characteristics of monitors that were thought to be detrimental to good program structure.

Several other, less extensive, proposals have been made to change specific features of monitors. Among these are the automatic signalling mechanism suggested by Kessler[23], and the manager construct of [21], which are aimed specifically at improving the signalling mechanism in monitors (see Chapter 3). These are not discussed in the thesis; serializers are a more extensive revision of the monitor construct and cover the changes made by these proposals.

Few papers exist on techniques for evaluation of synchronization mechanisms according to the criteria mentioned earlier. Andler[1] presents a comparison of semaphores, conditional critical regions, monitors and path expressions. The comparison is based on solutions to the bounded buffer problem, and focuses on correctness issues. While we are concerned with correctness, our interest is primarily in how well a mechanism supports construction of correct programs, rather than with proof techniques for the mechanism.

In [20], Howard has compared several versions of monitors. Howard is primarily interested in equivalence of internal specifications of the various versions, and does not address issues of expressibility or ease of use.

The work on the "nested monitor call" problem by Lister[28], and the responses to his initial presentation of the problem [29, 31, 22] are also relevant to our research. Further discussion of this work appears in later chapters.

A brief comparison of monitors with serializers appears in [3]. Some discussion of the differences between monitors and path expressions also appears in [15].

2. Criteria and Evaluation Techniques

In this chapter, we present the criteria to be used in the evaluation of synchronization mechanisms. Techniques for measuring how well these criteria are supported by various synchronization constructs are also presented.

Our principal concerns in this evaluation focus on programming methodology and the ways in which the addition of synchronization to a language influence software quality and reliability. We are therefore interested in such properties of synchronization mechanisms as expressive power, ease of use, modularity, modifiability, and correctness. These terms are sufficiently vague to make evaluation according to these criteria extremely difficult.

We will attempt to clarify the definitions of these properties with respect to synchronization. This chapter is divided into several sections. The first deals with modularity; it applies the concept of abstraction mechanisms to the problem of modularizing the implementation of shared resources. The following section is devoted to defining a method for classifying synchronization problems and describing the range of problems that synchronization mechanisms will be expected to satisfy. This classification of problems will be needed in later sections to describe techniques for evaluating expressive power, ease of use, and modifiability, since these properties are meaningful only with respect to a given set of problems. The final section discusses correctness, and the properties of a synchronization mechanism that influence how easily a program can be written correctly and how easily it may be proved correct. We will not actually discuss proof techniques.

Thus, this chapter is devoted to establishing the definitions and techniques necessary for evaluating how well synchronization mechanisms support production of reliable, high quality software. It is a first attempt at establishing some standard criteria for evaluating

properties long held to be very important for programming language constructs, but which have only intuitive, imprecise definitions.

2.1 Modularity

By modularizing programs, we limit the complexity the programmer must deal with at any given time, thus making it easier to write correct programs. The increase in complexity of software due to the presence of concurrency makes modularization essential for maintaining correctness. In this section we describe the ways in which software used to access or control access to shared resources should be modularized. This modularization is based primarily on the use of abstraction mechanisms[28].

There are two distinct modularity requirements for concurrent programs accessing shared resources. The first follows from the principle that the definition of an abstraction should be separated from its use. We consider a shared resource to be a data abstraction. The definition of the synchronization for a shared resource should be part of the definition of that resource, rather than being associated with each resource access. Thus our first modularity requirement is that users[1] see a shared resource abstraction that can be assumed to be properly synchronized. No synchronization code need be included in programs accessing the resource.

Our other modularity requirement has to do with the shared resource definition. Within the module that implements the shared resource, we have the definition of the structure and operations on the resource, as well as the definition of the synchronization scheme for the resource. These two parts actually serve different functions and should be separable into different subsidiary abstractions of the shared resource.

---

1. "User" of a resource refers to systems or applications software that accesses the resource.

We thus have a model of shared resources that consists of two levels of abstraction. At the higher level we have a protected resource abstraction with the operations that users may perform in accessing the resource. At the lower level, we have the resource abstraction, with the access operations that may be performed after a synchronizer ensures that access is safe, and a "synchronization abstraction", which contains state information necessary for synchronization, but not conceptually meaningful as a part of the resource, as well as synchronization operations.

In examining synchronization constructs, we will be attempting to determine whether they automatically provide this modularization, and if not, whether they allow the resource implementor to easily modularize the design in this manner.

2.2 Categorizing Synchronization Problems

As stated earlier, expressive power, ease of use, and modifiability can only be evaluated relative to a specific set of problems. A synchronization mechanism need only be powerful enough to easily express solutions to those problems we consider to be valid synchronization problems. We therefore need a way to describe the range of problems in which we are interested. In this section, we identify a set of properties of synchronization schemes by which we can classify these problems. We will later use the idea that, since synchronization schemes have various combinations of these properties, testing whether the mechanism can express schemes with each property, and whether it allows us to easily combine properties, will indicate the power and usability of the mechanism.

2.2.1 Categorization of Constraints

Synchronization mechanisms serve two main functions with respect to shared resources. One is excluding certain processes from the resource, under given circumstances; the other is scheduling access to the resource according to given priorities. Synchronization schemes are thus composed of a set of constraints, each having the form:

if condition **then** process A **is excluded from** the resource

or:

if condition **then** process A **has priority over** process B

We will refer to constraints of the first type as exclusion or concurrency constraints and the second as priority constraints. Within these two main classes, constraints differ in the kinds of information referred to in the conditional clause. The information that should be available to the synchronizer, and thus the information that can appear in constraints, falls into several categories:

1. the procedure(s) requested :[2]

   The resource is a data abstraction, so access to it is always obtained through operations of the resource type. In some synchronization schemes, the constraints depend on the operation requested. In stating, for instance, that readers of a data base have priority over writers, we are giving a constraint in terms of the types of procedures requested. In contrast, a strict first_come_first_serve ordering uses no information about the procedures requested.

2. the time at which requests were made:

---

2. We will often refer to this information as the "type" of the request.

Though it is rarely necessary to know exact times of requests, the time of a request relative to other events is often important. The most frequent use of time information is the determination of the order of requests. In addition, it is sometimes necessary to determine the synchronization state(see below) at the time of a request.

3. arguments passed with requests:

In many cases, the arguments passed with a request for resource access are needed to determine the order in which processes should be admitted to the resource.

4. the "synchronization state" of the resource:

Synchronization state includes all local data and state information needed only for synchronization purposes. Included in this category is information about the processes currently accessing the resource, and the procedures those processes are executing.

5. the local state of the resource :

Local state includes information that would be present regardless of whether the resource were being accessed concurrently or sequentially. It is information meaningful to the actual unsynchronized resource abstraction. Though local state information is used in many synchronization schemes, its use often causes problems because it interferes with modularity requirements. (The local state information belongs in the resource module, and thus a synchronizer will not have automatic access to it. Several options for handling this problem are discussed in later chapters.)

6. history information:

History information is concerned with whether or not a given event has occurred, such as whether a specific procedure has been executed. This information type differs from synchronization state in that it refers to resource operations that have already completed, as opposed to those still in progress. It is often interchangeable with local state

information, since past events in which we are interested will most likely have left some noticeable change in the state of the resource. It is convenient to treat it as a separate category because it may be easier for the synchronizer to keep track of the history of operations executed than to obtain the required state information from the resource.

We have thus identified two major types of constraints, and several classes of information that distinguish different kinds of constraints within the two major categories. To be sufficiently powerful, a synchronization mechanism must provide a means of expressing exclusion and priority; it must also enable the resource implementor to express those constraints in terms of any of the information types described.

In the next section, examples that use various combinations of constraint types will be given. The way in which a mechanism makes use of different types of information, and how easily it can get access to this information are very important in determining how easily well-structured, reliable solutions can be developed.

2.2.2 Examples

The following are standard examples of synchronization problems. This set was chosen to cover all of the information types presented. Only informal descriptions of the problems are given. Formal specifications seem unnecessary for our purposes, but to avoid any ambiguity, the appendix contains formal specifications using notation from [24].

The bounded buffer problem

The bounded buffer problem assumes there is a fixed size buffer, of length n, into which producer processes are placing data, and from which consumer processes are retrieving it. The constraints specified are that only one process may access the buffer at a time, that the producer may store in the buffer only if it is not full, and that a

consumer may retrieve information from the buffer only if it is not empty. Thus, the constraints make use of information on synchronization state, resource state, and the procedure requested.

Readers_Writers Problems

There are several readers_writers problems[10] that illustrate the use of different types of information. The readers_writers problems assume there is a shared data base having read and write operations. All of the versions used here have the same set of exclusion constraints: reads may occur in parallel, but a write operation excludes both readers and other writers. The priority constraints are different in each version. The similarity of the various forms of the problem makes this set of problems especially useful in evaluating modifiability.

Writers_exclude_others

This version of the problem uses the exclusion constraints mentioned above but imposes no priority constraints. This synchronization scheme illustrates an important type of problem that synchronization mechanisms should be able to handle. The user may not care about the order in which operations are executed in certain cases. There may be external constraints that guarantee that eventually every request will be served, and the order is unimportant. Many mechanisms force the programmer to define an ordering when the specification has none. The inability to leave specifications nondeterministic is a weakness in the expressive power of the mechanism.

Readers_priority (or writers_priority)

In this version a priority constraint is added. If both a read request and a write request are pending, then the read (or in writers_priority, the write) is always given priority. The exclusion constraints remain the same. The priority is now based on the

operation requested. Notice that this scheme allows starvation.[3]

First_come_first_serve (fcfs)

In this version of the readers_writers problem, the type of operation requested is not used at all in the specification of priority constraints. Instead, priority is based entirely on order of request.

Fair_readers_priority

The fair_readers_priority scheme gives readers some priority over writers but limits that priority enough to be sure writers will eventually be served. One way of fulfilling this requirement is by use of the following constraints. If there are writers waiting when a read is requested, then the read must wait until one write completes. All reads waiting at the termination of that write may proceed. These constraints imply that only a finite number of readers have priority over a given writer. The writer that has been waiting longest will have priority over any readers not yet in the resource. The priority constraints for this scheme use a combination of request time and operation type.

The One_slot buffer

The one_slot buffer[8] problem assumes there is a message buffer with room for exactly one message. Users may insert and remove messages. The synchronizer must guarantee that a message is inserted before any process executes a remove, and that no message may be inserted before the previous one has been removed. Thus, an insert may occur only if the previous operation was a remove or a create, and remove may

---

3. Starvation means that a process waiting to access a resource may wait forever and never be granted access. In the readers_priority scheme, since readers have higher priority than writers, if reads are requested often enough a writer may wait forever.

occur only when the previous operation was an insert. Operations occurring out of order must wait until these constraints are satisfied. This example therefore illustrates the use of history information. The synchronizer must keep track of the operations already executed to determine whether a process may enter the resource.[4]

The disk scheduler

The disk scheduler [18] is a scheme to control access to a disk by using an "elevator" algorithm. The disk head moves in one direction until there are no more requests for tracks in that direction; then the direction is reversed. The access request contains the track number as an argument. The algorithm works as follows. If the head is currently moving up (toward higher-numbered tracks) then requests for tracks at the current track or lower must wait for the return pass. Requests that arrive for higher-numbered tracks will be serviced when the head reaches that track on the current sweep. Thus, it is the parameter of the request, and the state of the resource(i.e. current head position and direction) that determine the priority. The exclusion constraint allows only one process at a time to use the disk.

The alarmclock

The alarmclock is a system facility that allows processes to block themselves and request to be restarted after a specified period of time. Thus, granting the "resource request" means restarting the process. The order in which requests are served is based on the argument telling the alarmclock when to grant the request. The alarmclock example itself may not be a realistic use of synchronization. However, it is felt that it illustrates a class of problems that a synchronizer should be able to handle.

---

4. This problem can be restated using local state information if there is some way to determine whether the buffer contains an unread message.

Both the alarmclock and the disk scheduler represent examples of synchronization problems using arguments passed with requests as the basis for determining priorities. The primary difference between them is that the disk scheduler has a fixed number of possible parameter values on which to base the ordering, while the alarmclock may take any integer value as an argument. It therefore may require more mechanism to handle the type of problem illustrated by the alarmclock. Either the disk scheduler or the alarmclock can be used to represent the class of problems using arguments as a basis for priority.

All of the examples given deal with single resources and single accesses in each call to the synchronized resource. We have assumed throughout this thesis that the correct level of synchronization is at the point of access to the resource. One may, in addition, want synchronization at a level encompassing several resource accesses in the course of executing a synchronized operation. Some of the problems resulting from this extension are discussed in the section on correctness of hierarchically structured resources.

We have presented a method for categorizing synchronization problems according to their function and the types of information needed to express their solutions. This categorization will be used in the following section to develop methods for evaluating the expressive power of synchronization mechanisms. Evaluation techniques for ease of use and modifiability also make use of this problem classification.

2.3 Expressive Power

In evaluating the expressive power of a synchronization mechanism, we will be attempting to decide whether the mechanism provides straightforward methods for expressing priority and exclusion constraints, and whether one has the ability to express those constraints in terms of any of the information types described earlier.

One test of expressive power is to use the mechanism to implement solutions to the examples given in the previous section. If there is no direct way to use a certain kind of information, it should become obvious when an attempt is made to implement a solution requiring it. While testing one example from each class of information may be insufficient to guarantee that a mechanism is actually powerful enough, it does provide us with some indication of a mechanism's power, and will at least point out any large gaps in power.

A more general way to measure expressive power is simply to examine each mechanism and attempt to determine what features it has that will enable it to deal with each type of constraint. For example, we will see that monitor queues are a construct for handling request time information, while serializer crowds retain synchronization state information. Some data manipulation technique must be available for each type of information. The ability to identify the particular way in which to handle each information type will also make a mechanism easier to use because the structure of a solution will be indicated by the kinds of information referred to in the specification.

One technique that is often used for comparing the computational power of language constructs, and that has recently been used to compare several versions of monitors [20], is translation between solutions using different mechanisms. In comparing computational power, this technique is useful because if one mechanism can be implemented in terms of another, then the implementing mechanism must be at least as powerful as the one implemented. If the translation is possible in both directions, the two mechanisms must be equally powerful. This technique has been used to show that monitors, serializers and path expressions are all as powerful as semaphores. Since semaphores are considered to be sufficiently powerful as a synchronization construct, all three higher level constructs must have sufficient computational power.

It has been suggested that this translation technique can be used in comparing *expressive* power as well. If there is a *straightforward, simple* translation from one mechanism to another, then the one translated to must have at least the expressive power of the other. We have chosen not to employ this technique because the results of such a translation are unclear. It is too difficult to judge how simple and straightforward a translation algorithm is, or whether the translations in each direction are equivalent in complexity. If the translation in one direction varies slightly in complexity from the one in the other direction, the mechanisms probably vary slightly in power. Though the methods presented earlier for analyzing expressive power seem less algorithmic than the translation technique, we feel that by defining the set of properties we expect a mechanism to express, and then testing for the ability to do so, we have in fact used a more objective approach than translation.

2.4 Ease of Use

In analyzing expressive power, we determine whether a synchronization mechanism allows the straightforward implementation of the synchronization constraints described earlier. Whether or not a mechanism is easy to use depends not only on the ability to easily construct solutions to individual constraints, but on the ability to easily construct implementations of complex synchronization schemes made up of many such constraints.

Given that our requirements for expressive power are satisfied, complex synchronization schemes will be easy to implement only if they can be decomposed into individual constraints that can then be realized independently. If the implementation of any one constraint is dependent upon the other constraints present, solutions quickly become very difficult to construct as the number of constraints increases. Since the implementor must be aware of the entire set of constraints, and make sure that each constraint is consistent with every

other constraint present, the complexity of *constructing* the solution (not the complexity of the solution itself) increases with the number of combinations of constraints present. It is therefore far more difficult to construct a solution than if it were possible to implement each constraint separately, regardless of which other constraints were present.

One way to test whether a mechanism allows independent implementation of constraints is to examine solutions to two similar synchronization problems. If the solutions share some constraints, but differ in others, then the common constraints should be similarly implemented in both solutions. Differences in the way a given constraint is implemented in two different synchronization schemes, or solutions in which the implementations of each individual constraint are not even identifiable as separate parts of the solution, indicate that our independence criterion for constraints is being violated.

Among the examples presented earlier in this chapter, there are several readers_writers problems having a common exclusion constraint. The problems differ in the priority constraints used. These examples provide a good basis for examining independence of constraints. If the implementation of the exclusion constraint cannot be isolated in each mechanism, or if the implementation in each mechanism differs, it is an indication that the mechanism is hard to use. Conversely, if the implementation of this constraint is the same or very similar in each solution, we have a fairly strong indication that each constraint is independent of other constraints in the synchronization scheme.

Assuming a mechanism satisfies this constraint independence property, if it is easy to express solutions to each individual constraint, it will be easy to express solutions to more complex synchronization problems. Our evaluation of expressive power should indicate how easily individual constraints can be expressed. Mechanisms that are easiest to use will be those for which there is a particular structure or method for handling each information class and

constraint type.

2.5 Modifiability

We define modifiability to mean that a small change in a synchronization specification will result in a similarly minor change in its implementation. Like ease of use, modifiability is primarily dependent on the constraint independence property discussed in the previous section. If each constraint is implemented independently, a modification to one constraint should affect only the part of the solution implementing that constraint. If we have shown in our evaluation of expressive power that each type of constraint is easily implementable, then a small change in the specification should be easy to implement.

We can also evaluate modifiability by looking at modifications that might typically be made to some synchronization schemes, and judging whether the extent of the change required in the implementation was consistent with the size of the change in specifications. We would expect that a modification to one constraint that did not affect the type of the constraint or the kinds of information used, would be simple to implement. The structure of the modified solution should be similar to that of the original.

Modifications involving many constraints, or those involving changes in the types of constraints or kinds of information used, are more extensive, and can be expected to require more significant changes to the implementation. However, if it is extremely difficult to change an implementation when a realistic change in specifications has been made, the mechanism may not be consistent with our goals. Such a weakness in modifiability is usually indicative of a weakness in understandability, expressive power, or ease of use as well.

We would like to analyze and compare the ease with which modifications may be made both within a constraint class and between constraint classes. To do so, we will examine several

versions of the readers_writers problem: readers_priority, writers_priority, and fair_readers_priority. The readers_priority and writers_priority examples can be used to evaluate modifiability for the case in which the constraint types are not changed, since both use priority constraints based on procedure requested. The fair_readers_priority problem combines priority based on procedure type with that based on order of request. We would thus expect a change from readers_priority to writers_priority to be easier than a change from readers_priority to fair_readers_priority.

Thus, we can measure the "size" of a modification in terms of the number and types of constraints changed, and use this metric in evaluating how well synchronization mechanisms support modifiability. In this thesis, we will use transformations between various versions of the readers_writers problem to test modifiability.

2.6 Correctness

In the area of correctness, we are concerned primarily with the ability to write correct programs, rather than with techniques for verifying those programs. In the sections on correctness in the following chapters, we will concentrate on two main topics. One is whether there are specific features of each mechanism that will either aid or impede the production of correct programs. Highly structured mechanisms that perform a great deal of syntactic checking