MIT/LCS/TR-229

AUTOMATIC VERIFICATION OF SERIALIZERS

RUSSELL ROGER ATKINSON

Automatic Verification of Serializers

Russell Roger Atkinson

March 1980

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 MCS 74-21892 A01.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge **Massachusetts 02139**

Automatic Verification of Serializers

Russell Roger Atkinson

Abstract

This thesis is concerned with the problem of controlling concurrent access to shared data. A language construct is proposed to enforce such control; a specification language is defined to describe the formal requirements of such control; and verification techniques are given to prove that instances of the construct satisfy their specifications. The techniques are justified in terms of the definition of the construct and the definition of the specification language. Results are given for a program that implements a number of the techniques, illustrated by verifying several versions of the readers-writers problem. Interactions between instances of the construct are discussed in the context of a simple file system.

Thesis Supervisor: Barbara H. Liskov
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: verification, concurrency, monitors, serializers, specification

Acknowledgements

I would like to especially thank my advisor, Professor Barbara Liskov, for her help and encouragement during the past six and a half years of graduate study, and in particular for her work in supervising this thesis.

My readers, Professor John Guttag and Professor Carl Hewitt, also deserve much credit. John is largely responsible for whatever respectability the mathematics in this thesis may contain (the omissions are, of course, my own). Carl is responsible for the basic theme of this research. He proved to be an excellent listener during my first tentative steps toward a topic.

Deepak Kapur and Craig Schaffert deserve credit for many useful suggestions during the course of this work.

The verification program described in this thesis is the descendent of the earliest work that I performed in this area. I especially credit Barbara Liskov, the CLU programming language, and the LCS computing facilities for providing fertile ground for this research. My thanks, also, to the people who were dragged over to watch the program work (or fail to work).

I chose to use "we" rather than "I" in the remainder of this thesis in part to recognize that most of the ideas in this thesis are a result of effort by many people in the Computer Science community at large. My apologies to those who might consider their ideas misrepresented; my thanks to those who contributed.

CONTENTS

FIGURES

1. Introduction

This thesis is concerned with the problem of controlling concurrent access to shared resources. In systems where several processes may attempt to concurrently access the same resource, there is a need to impose some order on those accesses. If certain orders are not enforced, certain classes of access to the resource may conflict and cause erroneous results. Other classes of access to the same resource may proceed concurrently without conflict. This is true whether the resource is a data base, a printer spooler, a file system, or a communications network, although the definition of the classes of access may be specific to the resource.

Given this framework, we can informally define a few terms. Two accesses are *concurrent* if both accesses have started, yet neither has completed. Typically, concurrent access is controlled through *exclusion*, where a process executing one class of access prevents the initiation of another access from any of a set of classes. When one access excludes another, the latter must wait for the former to complete. If one access is waiting for another, which is waiting for the first to complete, then no progress can be made on either, which is called *deadlock*. If two processes are ready to initiate accesses, yet one access excludes the other, then the process that proceeds is said to have *priority* over the other. A process that is ready to proceed, yet is continually denied progress, suffers from *starvation*.

We wish to ensure that programs executing concurrently on shared resources obtain correct results, where correctness is defined in terms of programs meeting their specifications. We wish to show, for properly designed programs, that certain accesses exclude others, that the proper accesses are granted priority, that appropriate access may proceed concurrently, that there is no deadlock, and that there is no starvation. We limit this concern to the issues that are specific to concurrency, and not those that apply to determining whether the access, executed by itself, has the correct effect on the resource or returns the correct information. Also, we are not concerned with concurrency issues unrelated to accessing resources, such as process creation and deletion.

1.1 Initial decisions

Our first decision is that it is desirable to have a separate programming language construct to realize reliable control of concurrent access. We believe it insufficient to simply propose a construct and present some examples of its use. A language designer should also provide tools that increase the utility and reliability of a language construct. Consequently, this thesis presents:

- A language construct to control concurrent accesses to shared resources.
- A definition of the semantics of the construct.
- A specification language to describe properties of concurrency control that are to be realized through this construct.
- A verification methodology that is used to prove that instances of the construct satisfy their specifications.

- The design of a program to make use of this methodology and perform verification.

One of the contributions of this thesis is that all of these elements are presented together for a single construct.

Our approach to concurrency control is heavily influenced by by the *monitor* construct of [Brinch Hansen 72] and [Hoare 74], and the programming languages CLU [Liskov et. al. 77, Liskov 79a] and Alphard [Wulf 78], which in turn owe much to Simula [Dahl 72]. In these languages, access to data objects is achieved through a limited set of operations, which are generally implemented as procedures. Just as CLU and Alphard separate implementation details from the abstract appearance of data objects, our objective is to separate concurrency control from access to data objects. The monitor construct has a similar goal, although a slightly different view of data. The connection between concurrency control and data abstraction is a key issue in defining our construct and in our verification techniques.

Verification does not prove that programs operate correctly, in the sense that a verified program performs exactly as desired. There is often no reason to believe that the specifications are better than the program text for describing the desired behavior for the program. Verification performs the task of taking two different descriptions of a problem solution and showing that the descriptions agree, in the sense that every behavior that the program exhibits is allowed by the specifications. The two descriptions are quite different in kind: the code is an algorithmic description, and the specifications describe the effects of executing the code. The confirmation of arriving at the same answer through two different methods ought to increase confidence in the solution.

We wish our techniques to be valid whether there is true concurrency, using multiple processors, or simulated concurrency, using a multiplexed single processor, or a mixture of the two. To accomodate this range of behavior, we have described accesses as being concurrent if both accesses start before either ends. This definition may seem overly broad, since two accesses are considered to be concurrent if one access occurs as part of the other. We choose to make a conservative decision: two accesses are potentially concurrent if the start of either access can occur between the start and finish of the other.

1.2 Modularity

Large programs are usually difficult to understand and modify not because of their size, but because of their complexity. This complexity is far more often due to interactions between parts of programs than it is to inherent complexity in the task being performed. The notion of *modularity* is widely accepted as a means of limiting these interactions, although the term is defined in various ways. This principle is useful in constructing programs, in modifying programs, and in verifying programs. Modularity in verification has also been called the *independence principle*:

> *The proof of a routine may only depend upon its own specifications and implementation, and upon the* **external** *specifications of the routines to which it textually refers.* [Good, Cohen and Keeton-Williams 79, p.45]

We propose to make use of the following kinds of modularity:

- * *Data abstraction* is the organization of data into distinct objects, where each object belongs to a distinct data type, and direct access to the objects of any type is limited to the operations of the type. This definition of data abstraction follows the lead of the CLU programming language.

- * Concurrency control is separated from data access. The implementation of concurrency control is kept distinct from the implementation of data access, although the external interface of the two implementations may be similar.

- * Specifications of concurrency control are separated from specifications of other properties of a program. Further, these specifications are meant to be independent of any implementation.

- * Verification of concurrency control is separated from other program verification techniques. In particular, the verification of access to a resource and the verification of the concurrency control for such access are independent, although each may assume the specifications of the other (we will assume an absence of circularity, since it is a separable issue).

It is possible to find fault with modularity, since the kinds of separation we have described may make it more difficult to acheive other desirable properties.

- * The principle of modularity can be misapplied: the wrong kind of separation prevents necessary data from being communicated from one place to another. We hope to show through the use of examples that the kinds of modularity we propose to use do not prohibit necessary information from being in the appropriate places.

- Modularity can be inefficient: the mechanism for transferring from one context to another, as in a procedure call or process switch, can be expensive. Further, by limiting access to certain data, certain computations may be redundant. We will not address this issue directly in this thesis, but will return to this objection in our conclusions.

1.3 Related work

Much of the initial work on the construct we propose was done in conjunction with Carl Hewitt [Hewitt and Atkinson 79]. Since then, there has been a divergence in our efforts; this thesis explores issues of automatic verification of concurrency control, while Hewitt has concentrated on more primitive control of concurrency in a context where programs communicate by passing messages. Some of this work can be found in [Hewitt, Attardi, and Lieberman 79].

Below we briefly discuss related work on language constructs, concurrency specifications, semantic models, and some differences in our approach from other work.

1.3.1 Related language constructs

Most authors in this area note the importance of limiting the interactions between concurrent processes through the use of language constructs specifically designed for this purpose. We have a similar approach in this thesis, with the addition that we attempt to relate concurrency control to abstract (user-defined) data types.

We have already noted the intellectual debt owed to the monitors of Brinch Hansen and Hoare. For now, we characterize the monitor approach by noting that concurrency is controlled by only allowing one process at a time to execute an operation that belongs to a monitor. Given that initial exclusion, further execution orders may be imposed by the monitor operations. We will present a more detailed comparison of our construct with monitors in Chapter 2.

Another line of thought in concurrency control is to limit parallel processes to communicating through the passing of messages. Various authors have proposed such an approach, among them [Good, Cohen and Keeton-Williams 79, Hoare 78, Feldman 79]. Concurrent actions only proceed when a process that is sent a message chooses to receive it. Exclusion for a class of access derives from a refusal to accept a message of thct class. This approach is particularly well suited to distributed systems, where different processes may reside on widely separated processors.

These two approaches are not as different as they might initially appear. Although our presentation will follow the first approach, we will argue in this thesis that our techniques are valid for the second approach as well.

1.3.2 Concurrency specifications

Our work on specifications is strongly influenced by Greif [Greif 75]. In this approach, certain events related to an access are identified: access request, access start, and access finish. Specifications are given by indicating which orders of these events are required. For example, suppose that the execution of one kind of access (call it X) prevents another kind of access (call it Y) from starting. We can specify this requirement by stating that no Y access start event can occur between any X access start event and the corresponding X access finish event.

A similar approach to specifications appears in [Laventhal 78], in which such specifications are used to synthesize implementations to realize concurrency control.

1.3.3 Related semantic models

Various models have been used to describe concurrent execution of programs. In the models we discuss here, a program proceeds from state to state by atomic actions.

* In [Howard 76, Good, Cohen and Keeton-Williams 79], and in our work, actions that take place are recorded in sequences called *histories*, and program semantics are described by giving predicates that must be satisfied for histories.

* In [Greif 75], actions are related by partial orders called *behaviors*. Program behavior is given by predicates on these partial orders.

- In temporal logic (a survey-level explanation of this model appears in [Lamport 80]) the model uses sequences of states, rather than actions. Predicates that describe program behavior may be applied to sequences of states, for a *linear time* theory, or to all sequences of states with a common sequence of states as a prefix, for a *branching time* theory.

- Another related model, based on trees of states, is presented in [Owicki 75]. Given an initial state and a program, the behavior of the program is characterized by a tree of states, where the arcs represent execution of an action that leads to the next state.

All of the above models use some structure to relate either states or actions, and describe program behavior by giving predicates on such structures.

It is possible to discuss states in terms of equivalence classes of histories (or behaviors). For example:

> [There] *is a correspondence between states and behavior that allows one to define the states of a system as an equivalence relation over the possible behaviors.* [Greif 75, p. 72]

We believe it better to think of predicates on histories rather than to attempt to regard states as equivalence classes. The distinction lies in our concern with certain properties of objects at any particular time, rather than the entire state of the object.

1.3.4 Differences in our work

We approach concurrency control not just by defining a language construct, but also by providing specification and verification methods for the construct. Further, these methods are actually demonstrated in a simple automatic verifier. By providing a wide range of support for a relatively narrow construct we hope to illustrate the benefits of a unified approach to controlling concurrent access to resources.

We have attempted a greater use of modularity than is commonly found in other works. In particular, we couple control of concurrent access to the principles of data abstraction with strong typing, while maintaining separation of concurrency control specification and verification from data access specification and verification.

1.4 Plan of thesis

Chapter 2 introduces the *serializer* language construct, which is a method for controlling concurrent access. An informal presentation is made of the syntax and semantics of the construct. An example, based on the readers-writers problem, is discussed in detail. A simplification of the serializer construct is defined for use in later chapters. A translation of serializers into clusters and semaphores is given as a possible implementation strategy.

Chapter 3 presents a simple semantic model that supports concurrency, and uses it to define more precisely the simplified serializer construct. A definition language based on first-order predicate calculus is used to describe serializers as enforcing limitations on the execution order of programs.

Chapter 4 discusses the four kinds of concurrency control specifications used in this thesis. A simple specification language for concurrency control is defined. Specifications are given for the readers-writers problem, with several variations, and the bounded buffer problem.

Chapter 5 presents and justifies rules that are used to verify that serializers meet their specifications. Although the definition of serializer semantics and the definition of the specification language are sufficient to allow us to verify serializers, it would be difficult to write an automatic verifier that directly uses these definitions. Therefore we define and prove a number of inference rules that allow us to infer specification clauses given the assumption (or proof) of other specification clauses,. An example is given of how the rules allow verification in a simple mechanical fashion.

Chapter 6 describes a program that uses the verification rules to establish that a serializer meets its specifications. We first describe how the structure of the program incorporates the verification rules, and then present examples of proofs that the program has performed.

Chapter 7 discusses issues related to interaction of serializers, and presents an extended example of serializer usage: a simple hierarchical filing system. Guidelines are given for providing serializers for data types that are originally used in a single-process environment.

Chapter 8 contains a discussion of how the work in the previous chapters can be extended to cover more complex problems and more complex serializers.

Several examples of serializers are presented in the appendices, and are referred to from time to time in the body of the thesis. The last appendix presents a table showing where the various definitions and rules used in this thesis are defined.

2. Serializers

This chapter introduces the *serializer* construct, which is intended to provide a modular method of concurrent access to shared data objects. Related programming language constructs are monitors [Brinch Hansen 72, Hoare 74], path expressions [Campbell and Habermann 74], and communicating sequential processes [Hoare 78].

We treat the serializer construct as an extension to the CLU programming language [Liskov et. al. 77, Liskov 79a]. However, the basic ideas behind serializers go beyond any particular programming language. Earlier versions of the serializer construct were presented in [Hewitt and Atkinson 77] and [Hewitt and Atkinson 79] using a significantly different language.

In this chapter we describe the rationale for the design of the serializer construct, informally define the syntax and semantics of serializers, and present an example of a serializer. Then we describe the limited version of serializers that we will be using in the remaining chapters, give a possible implementation of serializers in terms of semaphores, and compare the serializer and monitor constructs.

2.1 Serializer design issues

We believe that a language construct for controlling concurrent access to shared objects should have the following qualities:

- The shared objects should be separated into identifiable sets of objects, each set being a *resource*. A resource should also be treated as an object, allowing resources to be composed from other resources. Each resource can only be directly accessed through a set of operations associated with the resource.

- The construct should separate control of concurrency from the algorithms that access the resource. This separation simplifies both the concurrency control and the resource access. Some concurrency may be lost by requiring complete separation, since it is likely to be difficult to partially overlap operations. However, we believe that the added simplicity is well worth the reduced concurrency.

- To aid reliability and verifiability, the shared resource should not be accessed except through an object that controls access to the resource. The concurrency control construct should enforce this restriction, since relying on programmers to follow conventions is not satisfactory.

- To ease the writing of programs that access resources, operations that access the object controlling the resource should appear to be, as nearly as practical, the same as the operations that access the resource. That is, the construct that controls concurrency should have the same appearance to the user as the construct used for the resource.

Based on these criteria, we designed the serializer construct to have the following characteristics:

- Like the **cluster** construct of CLU, the serializer construct is used to define data types by defining a set of operations for each type. The objects of a data type defined by the serializer construct are called *serializer objects.* Each serializer object is used to control a separate resource object. The operations of the data type are *serializer operations.* For the sake of modularity, serializer objects can only be accessed through the appropriate serializer operations.

- The execution of protected parts of a serializer operation for a particular serializer object precludes the simultaneous execution of protected parts of any serializer operation on the same serializer object. The process executing a protected part of an operation is said to have *possession* of the serializer object.

- During the execution of a serializer operation, possession of the serializer object can be released and regained. It is particularly useful to release possession while accessing the resource, thereby permitting concurrent activity involving the serializer object. After the resource access, possession is regained to indicate that the access is complete. This temporary release of possession permits external procedures to be invoked from a serializer operation while allowing other serializer operations to continue.

- During the execution of a serializer operation, it may become necessary to suspend execution to wait for some condition to become true. For example, if some operation needs exclusive access to the resource, it must wait until no other resource accesses are in progress. During this pause, possession of the serializer object is released to allow other requests to proceed concurrently as far as they are able.

Figure 1. A picture of a serializer object

```
        +--------------------------------------+
        |                                      |
        |    Serializer object                 |
        |                                      |
        |                     +------------+  |
        |                     |            |  |
Request -->  (Pause)  Request'-->          |  |
        |                     |            |  |
        |                     |  Resource  |  |
        |                     |            |  |
Reply <---   (Pause)  Reply'<---           |  |
        |                     |            |  |
        |                     +------------+  |
        |                                      |
        +--------------------------------------+
```

A graphical description of how the serializer construct is used is shown in Figure 1. A `Request` is the start of an operation, and a `Reply` its termination (possibly passing back information). The intended effect of the serializer is to impose an ordering on the requests and replies as they are transmitted between the resource and the requesters. The `(Pause)` is optional, based on whether the resource access requested can be performed immediately when the request enters the serializer. In most cases, a serializer operation passes the information it receives from the caller to the corresponding resource operation, and passes the information it receives from the resource operation to the caller.

2.2 Serializer syntax and mechanism

This section gives a brief syntax for the serializer construct and the statements used only by serializers. We also give an informal description of what each form is used for and how it works.

The syntax used for a serializer is similar to the syntax used for a CLU cluster. The header names the serializer and lists the externally available operations. Then the representation type for the serializer is given, which determines the names to be used for the components of the serializer object. Then the operations are given as procedures. The form of a serializer is:

```
name = serializer is operation_name_list

    rep = representation_type

    operation_name = proc ( formal_arguments )
                 optional_return_list
                 optional_exception_list
        procedure_body
        end operation_name

    . . . % other operations

    end name
```

We have used italics to informally indicate syntactic quantities.

As with clusters, the serializer construct defines a new data type, where the type is denoted by *name*. Certain of the operations are used to create new serializer objects of the named type, while other operations are used to access the serializer objects. Operations named in the *operation_name_list* are the *externally available* operations, and may be used by code outside of the serializer. Operations not named in the *operation_name_list* may only be used internally. Starting the execution of any externally available operation that directly uses the serializer object requires that the executing process gain possession of the serializer object (starting execution is shown as `Request` in Figure 1). Termination of an operation that has possession releases possession (termination is shown as `Reply` in Figure 1). To reduce the likelihood of deadlock, an operation that has possession of a serializer object is prohibited from directly calling another operation that requires possession of the same serializer object.[1]

We have also added two new kinds of statements that can only be used in a serializer. The **enqueue** statement is used to suspend execution (and release possession) until some condition is satisfied (shown as `(Pause)` in Figure 1). The statement has the form:

`enqueue` *queue_expression* `until` *boolean_expression*

The *queue_expression* denotes a queue that is used to impose a first-in-first-out discipline on processes waiting for conditions. The *boolean_expression* denotes the condition that is required to be true before a process can continue execution. Such a condition is called a *guarantee*. When a process is waiting for the condition to be true, we say that the process is *waiting in* the queue, since some identification of the process is stored in the queue. When a process waiting in a queue is allowed to proceed, it regains possession of the serializer object, the process identification is removed from the queue, and the **enqueue** statement terminates.

[1] In practice, it may not be possible to detect when this occurs. This does not affect our objective, which is to reduce the chances for errors. We do not believe that it is possible for a language restriction to completely eliminate this kind of error without unduly affecting the expressive power of the language.

The queues used in serializers are first-in-first-out unless otherwise specified:[2] If some process starts execution of an **enqueue** statement before another process starts execution of an **enqueue** statement for the same queue, the first process will complete execution of the **enqueue** statement before the second process, provided that either statement terminates.

The **join** statement is used to perform some body of statements that should be executed while *not* in possession of the serializer object. The statement has the form:

```
join crowd_expression
     body_of_statements
     end
```

A *crowd_expression* denotes a set used to identify the processes that have started executing a **join** statement but not completed it. There may be several such sets, called *crowds*, so that different classes of access can be distinguished.[3] The **join** statement starts by placing some identification of the executing process into the specified crowd and releasing possession (shown as `Request'` in Figure 1). After possession is released, the *body_of_statements* is executed. Finally, possession is regained (shown as `Reply'` in Figure 1), the process identification is removed from the crowd, and execution continues after the **end** of the **join** statement. Typically, a **join** inside of an operation is performed to invoke the corresponding operation of the resource.

[2] An example of the use of priority queues appears in Appendix 1.

[3] The **join** statement is so called because the process executing the statement *joins* a crowd of similar processes. It not be confused with fork and join primitives used for process creation and termination in other languages.

A process attempting to start or continue execution of an operation on a serializer object must wait until there is no other process that has possession of the serializer object. If the process is waiting for some condition to be satisfied, it does so in an explicitly named queue of an **enqueue** statement. If the process is waiting to gain possession at the start of an operation or at the end of a **join** statement, it does so in an implicit queue called the *external queue*, which is serviced in first-in-first-out order.[4]

Possession of the serializer object is released at the start of an **enqueue** statement (after the process is placed on the queue), the start of a **join** statement (after the process is placed in the crowd), and at the end of an operation. Whenever possession is released, the explicit serializer queues are examined to determine whether any queue has a process at its head with a true guarantee. If any of the guarantees are true, then one of those associated waiting processes will get possession of the serializer, and be removed from its queue. Then the process can proceed with the execution of the operation. In evaluating the guarantees, there is no assurance that the guarantees will be evaluated in any particular order, or that they will all be evaluated unless all evaluate to false. If all guarantees are false, then the process on the external queue that has waited the longest (if any) is removed from the queue and gains possession.

[4] We have chosen to use a single external queue for simplicity of explanation. Using a single external queue is a valid implementation, although it is not the only valid implementation.

2.3 An example: the readers-writers problem

The general readers-writers problem [Courtois, Heymans and Parnas 71] presents a simple resource that is to be accessed by concurrent processes. There are two operations on the resource, *read* and *write*. A process performing a read operation is called a *reader*, while a process performing a write operation is called a *writer*. In keeping with the serializer methodology, we have split the problem into writing a cluster to implement the resource and constructing a serializer that encapsulates such a resource. The basic constraint on concurrency is that readers should not access the resource concurrently with writers, and writers should not access the resource concurrently with other writers. The general readers-writers problem imposes no further requirement on the order of processing for operations.

The example we present in Figure 2 has the requirement that if a read operation on the serializer starts before a write operation on the serializer, the reader will access the resource before that writer, and that this first-in-first-out (FIFO) ordering is also imposed on writers with respect to readers, and on writers with respect to other writers. This variant of the readers-writers problem is discussed in [Greif 75].

In the FIFO serializer, there are three operations, one to create a new serializer object (and new resource), one to read a value associated with a key in the resource, one to write a value associated with a key in the resource. Only the serializer operations that access the representation (**rep**) of a serializer object argument need to

Figure 2. FIFO serializer

```
% The following serializer is a first-in-first-out solution to the
% readers-writers problem.

FIFO = serializer is
        create,    % Create a new serialized resource object
        read,      % Read a value from the resource given a key
        write      % Write a value to the resource given a key

    % Each serializer object has the following representation -
    rep = record [rc: crowd,      % readers' crowd
                  wc: crowd,      % writers' crowd
                  xq: queue,      % common queue
                  res: resource]  % unserialized resource

    create = proc () returns (cvt)
        return (rep${rc: crowd$create (),
                     wc: crowd$create (),
                     xq: queue$create (),
                     res: resource$create () })
        end create

    read = proc (x: cvt, k: key) returns (value)

        % Wait until there are no active writers
        enqueue x.xq until crowd$empty (x.wc)

        % Become an active reader & perform the read
        join x.rc
            return (resource$read (x.res, k))
            end
        end read

    write = proc (x: cvt, k: key, v: value)

        % Wait until there are no active writers or readers
        enqueue x.xq until crowd$empty (x.rc) & crowd$empty (x.wc)

        % Become an active writer & perform the write
        join x.wc
            resource$write (x.res, k, v)
            end
        end write

    end FIFO
```

gain possession of the serializer object.[5] The use of **cvt** as a type declaration for arguments to operations indicates which arguments are serializer objects viewed as their representations. The use of **cvt** follows the CLU usage, in that it represents a type conversion between abstract type and representation type that is performed at the interface of an operation. Each serializer operation is limited to one **cvt** argument, since there is no provision for gaining simultaneous possession of multiple serializer objects. There is no restriction on the use of **cvt** used as a return type[6] (even if we allow multiple serializer objects to be returned).

In the read operation of the FIFO serializer, the guarantee is `crowd$empty(x.wc)`. Therefore, no readers will begin to read from the resource until there are no writers accessing the resource. Similarly, in the write operation, the guarantee is `crowd$empty(x.rc) & crowd$empty(x.wc)`, which prevents a writer from proceeding until neither readers nor writers are accessing the resource.

The importance of having sole possession of the serializer object can be illustrated by examining Figure 2 and considering the consequences of *not* having such a restriction. For example, if a writer did not have sole possession of the serializer object after it performed its **enqueue**, another writer could access the resource between the first writer's execution of the **enqueue** statement and the **join** statement. This would

[5] The `create` operation does not need to gain possession, since no processes other than the process executing the `create` operation could access the object.

[6] Note that as an argument type description, **cvt** requires a conversion from abstract to representation type, and as a return type description, the conversion is from representation to abstract type.

allow simultaneous access to the resource by two writers, which violates our initial requirements for the serializer.

2.4 Simple serializers

It is infeasible to present definition, specification, and verification techniques for general serializers in this thesis. Therefore, we will restrict our attention to a limited version called *simple serializers.* A simple serializer has the following restrictions:

* The representation object (of type **rep**) for a simple serializer is a record that may only contain a single resource object and a fixed number of statically named queues and crowds.

* All queue and crowd expressions are limited to selection of representation components.

* The guarantees on the **enqueue** statements can only test for queue$empty, crowd$empty, the logical *and* (x & y) of guarantees, and the logical *or* (x | y) of guarantees.

* Only **enqueue** and **join** statements may be executed while in possession of the serializer object.

* Each serializer operations must correspond exactly in number, name, and interface to a corresponding resource operation. No statements may be executed inside a **join** statement except to invoke the corresponding resource operation, returning its results if there are any. This restriction also precludes the handling of exceptions.

* Inside of a simple serializer operation, the **return** statement does *not* immediately return an object from the operation, as it would in a normal operation. Instead, it is used to indicate the object to be returned when the serializer operation terminates. This restriction is present to simplify

the semantic model in the next chapter.

While the above restrictions may seem severe, they allow us to keep our presentation of details not associated with concurrency control to a reasonable level. Simple serializers are sufficient to solve the readers-writers problem, as well as some more involved examples.

In several places throughout the thesis we will indicate how extensions to simple serializers can be handled. These extensions include cases where more complicated computation must occur to determine the order of processing requests, where the interface to the serializer differs from that of the underlying resource, and where the serializer and the resource are implemented together.

2.5 Using semaphores to implement serializers

In this section we present a possible implementation of simple serializers using fair semaphores and clusters. We do this for two reasons:

1: To show that the serializer mechanism is realizable.

2: To give further insight into the semantics of serializers by giving a translation into a more commonly understood mechanism.

The semaphores that we use can be freely created, and obey a FIFO discipline when multiple processes request the same semaphore. We also describe the operations on the queue and crowd data types used in this implementation of serializers.

We assume that the semaphore data type has the following operations:

create ( ) returns (semaphore)
    returns a new semaphore with count = 0.

P (S: semaphore)
    Atomically tests and sets the count of the given semaphore. If count > 0, the count is decremented and the operation completes. If count = 0, then it stays 0 and the process performing the P operation does not proceed until the count becomes positive. Once the count becomes positive, the process waiting the longest decrements the count and completes the P operation.

V (S: semaphore)
    Atomically increments the count. Note that a P operation on an initially created semaphore must wait for a corresponding V operation.

We assume that the queue data type has the following operations:

create ( ) returns (queue)
    creates a new, empty, queue.

enq (Q: queue, T: semaphore, G: guar)
    adds the T, G pair to the queue, making the queue non-empty. The type of G, the guarantee expression, is assumed to be a predicate to indicate whether the guarantee is true.

deq (Q: queue) signals (empty)
    removes the head pair if the queue is not empty, otherwise signals empty.

empty (Q: queue) returns (bool)
    returns true if the queue is empty, false otherwise.

get_guar (Q: queue) returns (guar) signals (empty)
    returns the guarantee evaluation procedure at the head of the queue if the queue is not empty, otherwise signals empty. Note that queue$get_guar(Q) can also be written as Q.guar.

get_sem (Q: queue) returns (semaphore) signals (empty)
    returns the semaphore at the head of the queue if the queue is not empty, otherwise signals empty. Note that queue$get_sem(Q) can also be written as Q.sem.

We assume that the crowd data type has the following operations:

create ( ) returns (crowd)
    returns a new, empty, crowd.

insert (C: crowd, T: semaphore)
    inserts a semaphore into a crowd.

remove (C: crowd, T: semaphore) signals (absent)
    removes a semaphore from a crowd if present, otherwise signals absent.

empty (C: crowd) returns (bool)
    returns true if the crowd is empty, false otherwise.

Implementing a serializer as a cluster that uses semaphores is a translation that has the following cases:

1: The serializer becomes a cluster, and the representation object is extended to include a *sem* component, which is of type semaphore; and an *eval* component, which is of type sequence[queue]. The *sem* component is called the *external semaphore*, and the *eval* component is called the *queue list*.

2: The create operation initializes the external semaphore to a newly created semaphore, and performs semaphore$V on it. The queue list X.eval is initially the sequence of all queues in the representation.

3: Each operation that requires possession is given the following prolog:

```
semaphore$P(X.sem)
T: semaphore := semaphore$new( )
```

where X is the name of the cvt argument, and T is a unique local variable used to hold a newly created semaphore for the transaction. T is used to represent the process in queues and crowds.

4: A `return` statement is translated into an assignment to a temporary variable (or a multiple assignment if multiple return values are present). This requires such variables to be declared in the prolog, and their values returned in the epilog.

5: Each operation that requires possession is given the following epilog:

```
Eval(X)
```

where the Eval procedure is an internal operation used to select the next process to proceed, and will be detailed below.

6: Each statement of the form:

```
enqueue Q until G
```

is translated into:

```
queue$enq(Q, T, G')  % place self in queue
Eval(X)              % release possession
semaphore$P(Q.sem)   % regain possession
queue$deq(Q)         % remove self from queue
```

where Q is the queue to use in the expression, T is the local semaphore variable introduced in the prolog, and G' is a procedure (described as type *guar*) used to evaluate G.[7]

7: Each statement of the form:

```
        join C
             Body
             end
```

is translated into:

```
        crowd$insert(C, T)     % place self in crowd
        Eval(X)                % release possession
        Body                   % execute body
        semaphore$P(X.sem)     % regain possession
        crowd$remove(C, T)     % remove self from crowd
```

where `C` is the crowd to join, and `Body` is the body of statements to execute while not in possession.

The Eval procedure selects the next process to receive possession. It first checks (in some unspecified order) the non-empty queues to determine whether the guarantee at the head of the queue is true. The first non-empty queue found with a true guarantee has V performed on its head semaphore, and Eval returns. If no non-empty queues are found with true guarantees, V is performed on the external semaphore. Eval can be written as:

[7] A reader familiar with CLU may notice that we have taken some liberties in using G', and have not fully defined the type *guar*. In general, it is necessary to use a *closure* of procedure and data to properly define G'. We have avoided these issues for the sake of simplicity; they do not affect our approach to concurrency control.

```
Eval = proc (X: rep)

    % examine all queues for true guarantees
    for q: queue in sequence[queue]$elements(X.eval) do
        if queue$empty(q) then % if queue is empty
           continue            % then examine next queue
           end
        if q.guar(X) then      % if guarantee is true
           semaphore$V(q.sem)  % then allow that process
           return              %   to continue execution
           end
        end

    % no non-empty queues have true guarantees
    semaphore$V(X.sem)        % serve the external queue

    end Eval
```

The above version of Eval always checks the queues in some particular order. It would be equally valid to check the queues in any order, even if non-deterministic.

An example of how a serializer is implemented using clusters and semaphores is shown in Figure 3. We have omitted the write operation, since there is little difference from the read operation; and the Eval operation, since it was shown above.

2.6 A comparison of serializers with monitors

The unrestricted serializer construct has many similarities to the monitor construct [Brinch Hansen 72, Hoare 74]. Both serializers and monitors deal with synchronization by encapsulating details of concurrency control within a set of procedures. We present a brief comparison of the serializer and monitor constructs

Figure 3. Semaphore implementation of FIFO

```
FIFO = cluster is create, read, write

    elist = sequence[queue]
    rep = record [rc: crowd,      % readers' crowd
                  wc: crowd,      % writers' crowd
                  xq: queue,      % common queue
                  res: resource,  % unserialized resource
                  eval: elist,    % the queue list
                  sem: semaphore] % the external semaphore

    create = proc () returns (cvt)
        E: semaphore := semaphore$create()
        semaphore$V(E)
        Q: queue := queue$create()
        return ( rep${rc: crowd$create (),
                      wc: crowd$create (),
                      xq: Q,
                      res: resource$create (),
                      eval: elist$[Q],
                      sem: E } )
        end create

    read = proc (x: cvt, k: key) returns (value)

     % Prolog
        semaphore$p(x.sem)
        T: semaphore := semaphore$create()
        v: value
     % enqueue x.xq until crowd$empty (x.wc)
        queue$enq(x.xq, T, crowd$empty)
        Eval(x)
        semaphore$P(x.xq.sem)
        queue$deq(x.xq)
     % join x.rc; return (resource$read (x.res, k)); end
        crowd$insert(x.rc, T)
        Eval(x)
        v := resource$read(x.res, k)
        semaphore$P(x.sem)
        crowd$remove(x.rc, T)
     % Epilog
        Eval(x)
        return (v)
        end read

    % The write operation is not shown.

    end FIFO
```

below.[8] Except where noted, properties of the monitor construct are taken from [Hoare 74].

A serializer abstraction is intended to have the same interface as the protected resource, while the monitor appears to be a lock on access to the resource. The serializer construct has the expressive power to be used as a lock, but the monitor does not have the expressive power to mimic the resource (without serious loss of concurrency).[9] The serializer and monitor constructs both protect the underlying resource by controlling concurrent access to it, providing that the only access is through the serializer or monitor. The serializer construct further protects the underlying resource by allowing the programmer to prevent access to the resource except through the serializer. This protection can be achieved with monitors by having a data abstraction encapsulating a monitor, such that both the resource and the monitor can only be accessed through the data abstraction. Our preference is to provide this appearance through a single construct.

The serializer construct allows possession of the serializer object to be released and regained in a controlled manner within a serializer operation. In the monitors presented in [Hoare 74] there is no such provision. In an extension to monitors [Lampson and Redell 79] it is possible to write operations that do not require possession

[8] A comparison of an earlier version of serializers with monitors appears in [Hewitt and Atkinson 79]. An evaluation of serializers, monitors, and path expressions appears in [Bloom 79].
[9] Extensions which alleviate this problem have been made for the monitors presented in [Lampson and Redell 79].

of the monitor. This allows an operation to be written that requires possession of the monitor only for parts of the operation. These protected parts are required to be invocations of monitor operations that require possession. This solution is slightly more complicated to use than the serializer **join** statement, but is otherwise similar.

Serializers use explicit guarantees at the point in the procedure where a process waits on a queue. That guarantee is true when the process proceeds (providing that removing the process from the queue did not change the guarantee). Monitors also have first-in-first-out queues (called *conditions*), but the expressions that determine which queues are to be serviced next are distributed throughout the various procedures of the monitor, which complicates the verification task.

As mentioned briefly above, there is a basic difference the use of queues in monitors and serializers. Processes in the same queue in serializers can be waiting for different guarantees. Although the same effect can be achieved in monitors, it usually requires extra code to do so, and is difficult to write and understand.

The serializer construct, like the CLU cluster construct, supports sets of objects belonging to an abstract type. The monitors proposed in [Hoare 74] tend to support one-of-a-kind encapsulation. This difference is more a reflection of the base language used than a basic difference between serializers and monitors. We mention this difference because we believe that supporting sets of objects is a better choice to make, since there is more potential concurrency in a system where data is partitioned into separate objects.

2.7 Opportunities for optimization

One objection that might be raised to serializers is that they are inherently inefficient: at every release of possession the queues must be checked to determine whether the condition at the head of each queue is satisfied.[10] For this objection we have two answers:

1: It is unlikely that the evaluation of such conditions will be expensive compared to the execution of resource operations.

2: In the event of the guarantee checking being a significant cost in a program, optimization techniques are especially applicable in this limited context.

As an example of how we might optimize the checking of guarantees, consider the FIFO example. When a writer leaves the writers crowd, it is easy to prove that both the readers and writers crowds are empty. This knowledge allows an optimizing compiler to immediately dequeue the next transaction in the queue (if any) whenever a writer completes. In such a case, no guarantee evaluation takes place. When a reader leaves the readers crowd it is easy to prove that the writers crowd is still empty, which allows the compiler to simply check the head of the queue for a reader, thus avoiding any more complex evaluation. Whenever a writer joins the writers crowd all guarantees are known to be false, and do not need to be checked at all. In short, we have shown that intermediate steps of the verification program can lead to sufficient information to

[10] A similar objection is actually raised in [Hoare 74, p. 556].

perform optimizations that can significantly reduce overhead for checking guarantees.

We have advocated designing, verifying, and implementing serializers and data abstractions independently. This independence can lead (especially in CLU) to many levels of procedure calls, where each procedure performs an extremely small part of the computation. When the overhead for procedure calls costs on the same order as the rest of the computation, it becomes desirable to substitute the bodies of procedures for their invocations [Atkinson 76, Scheifler 77]. For serializers in the style we have advocated, it is generally both simple and beneficial to perform this substitution. We note that the simplicity of the substitution is greatly aided by our initial requirement that the serializer present the same interface as the underlying resource.

3. Semantic Model

In this chapter we present an abbreviated semantic model for concurrent execution of programs, and use it to define serializer semantics. In the next chapter, we use the model to define a small specification language for serializers.

The semantic model we use to define serializers is intended to be embedded within a larger semantic model, just as the serializer construct is embedded in a larger programming language. We will not be concerned initially with which larger model is used, although we will return to the issue later. Whatever larger model is used, there must be support for shared objects, side-effects, and concurrency.

We will first give an overview of the semantic model for serializers, assuming a particular larger semantic model. Then we discuss the various components of .he model in detail. Then we give the meaning of the serializer construct by giving predicates that all serializers must satisfy. Finally, we discuss the role of induction in the serializer model, and outline how the model might be embedded in a different larger semantic model based on message-passing between processes.

3.1 Overview of serializer semantics

Informally, the text of a serializer is a set of statements that describe what happens when serializer operations are executed in a system with concurrent processes. To give the semantics of the serializer construct, we require a definition of "serializer operations", a definition of "execution", a definition of "process", and a definition of "what happens".

The model we choose can be viewed as an interpreter. Each procedure is represented by a graph composed of basic instructions that indicate which actions to perform and arcs between the instructions to indicate the order of execution. There is a global state, consisting of a set of shared objects and a set of processes. Each process has a local state, which includes a set of local objects, a stack of procedure activations, and a program counter that indicates the instruction that the process is to execute next. Each instruction represents some basic action. Executing an instruction modifies the global or local state. The execution of an instruction always indicates the next instruction in the process by modifying the program counter. A process where the next instruction is permitted to occur is called *active*. Executing certain instructions may cause a process to become *inactive* until certain conditions hold.

For simple serializers, the only components of the global state modelled are the state of the queues and crowds for the serializer object, and the state of serializer possession. The only component of the local state modelled is the program counter within a serializer operation.

The interpreter proceeds by choosing an active process, and executing the instruction indicated by the program counter of that process. Although the choice of process is non-deterministic, no process that is active may be indefinitely denied execution. We call the sequence of instructions executed by the interpreter a *history*.

We can give the semantics of this informal model through a predicate that takes a history, an initial global memory state, an initial set of processes (and their local states), and a set of graphs representing the procedures in the system, and returns a boolean indicating whether the history could be produced by the interpreter we have described. We will call this predicate the *global legality predicate*.

In this thesis we are discussing a single language construct. In this context, presenting a complete definition for a language would occupy more space and attention than it merits. The semantics of a language construct can be defined through a *partial legality predicate* that partially determines the global legality predicate. For the serializer construct, this predicate is false for histories that are prohibited due to serializer semantics, and true for others. We will not present a definition of a larger language, nor formally state the interactions between the serializer construct and the other language features.

3.2 Nodes

In defining what is meant by "execution of serializer operations", we first need to define a representation for an operation and its associated data. Since we are dealing with only one serializer object at a time, it is convenient to regard the serializer operations and the serializer object as being inextricably bound together into a single unit. For brevity in this chapter, we will use the term serializer object to refer to this unit.

Each serializer operation (bound to an associated serializer object) is composed of *nodes*. A node is just (informally speaking) an instruction at some location in a program with its associated data. A *node graph* is used to represent a serializer operation, where the arcs in the graph represent sequential execution. For simple serializers, the node graph is degenerate, since there is a linear order to the nodes. We have used the term *graph* to ease the discussion of extensions to this model.

The following kinds of nodes are involved with synchronization in a simple serializer. At such a node, possession of the serializer object may be gained or released.

enter (*operation_name(formal_arguments)*): This node represents the initial entry to an operation that requires possession of the serializer object. After this node is executed, the executing process has possession.

exit: This node represents the epilog to an operation that requires possession. Executing this node releases possession.

enqueue (*queue, guarantee*): This node represents the first part of an **enqueue** statement. Executing this node places the process in the specified queue with the specified guarantee and releases possession.

dequeue (*queue, guarantee*): This node represents the second part of the **enqueue** statement. Executing this node regains possession and removes the executing process from the queue.

join (*crowd*): This node represents the start of the **join** statement. Executing this node places the process in the crowd and releases possession.

leave (*crowd*): This node represents the end of the **join** statement. Executing this node regains possession through the external queue and removes the process from the crowd.

The following kinds of nodes are used for other primitive actions that can occur in a simple serializer.

invoke (*invocation*): This node represents the termination of execution of the specified invocation. For simple serializers it will only appear once, and must appear in the body of a **join** statement.

return (*invocation*): As with the **invoke** node, the **return** node represents the termination of execution of the specified invocation. Executing the **return** node also designates the object to be returned when the serializer operation terminates at the exit node.

The use of **invoke** and **return** nodes in simple serializers is limited to showing where the operations of the underlying resource are called.

Each node N has the following structure:

* N.kind - an identifier (one of **enter, exit, enqueue, dequeue, join, leave, invoke, return)** indicating the kind of node.

* N.next - empty for **exit** nodes; otherwise the next node in the execution sequence. Note that the next node for any **return** node is an **leave** node if the return is performed while in a **join** statement, otherwise the next node is a **leave** node.

* N.mob - for **enqueue** and **dequeue** nodes, the queue used; for **join** and **leave** nodes, the crowd used; otherwise empty.

* N.expr - for **enqueue** and **dequeue** nodes, the condition to guarantee; for **return** and **invoke** nodes, the expression to evaluate; for an **enter** node, the operation name and its formal arguments; otherwise empty. Note that for an **invoke** or **return** node the information about which procedure is executed and which arguments are used is contained in the expression.

* N.match - for an **enqueue** node, the corresponding **dequeue** node; for a **join** node, the corresponding **leave** node; otherwise empty.

The transformation of a serializer operation to nodes will be given by example.

Suppose we have the following operation in a serializer:

```
change = proc (x: cvt, d: data) returns (value)
    enqueue x.q until crowd$empty(x.c)
    join x.c
         return (resource$change(x.r, d))
         end
    end change
```

The node graph for the above operation can be represented as:

```
N1:  enter (change(x, d))
N2:  enqueue (x.q, crowd$empty(x.c))
N3:  dequeue (x.q, crowd$empty(x.c))
N4:  join (x.c)
N5:  return (resource$change(x.r, d))
N6:  leave (x.c)
N7:  exit
```

In the above graph, N1.next = N2, N2.next = N3, and so on. N7.next is empty. The queues, crowds, and expressions are indicated.

N2.mob = N3.mob = *x.q*

N4.mob = N6.mob = *x.c*

N2.expr = N3.expr = *crowd$empty(x.c)*

The reader should be cautioned that the description we have given for nodes and node graphs is incomplete. We have not discussed conditional statements, assignment, exceptions, or iteration. In later chapters, we will describe how extended node graphs would be handled.

3.3 Events

Informally, an *event* is the completion of execution of a node in a process. For our purposes, the important features of an event are:

* An event is atomic. An event takes no time to occur, although the amount of time between events is always positive and finite.

* An event is associated with a single node of a serializer.

* An event is associated with a single "process". We assume that the reader has some intuitive idea of process. We will introduce a more exact definition of a specialization of the process notion in the next section.

It has been proposed [Greif75] that an event is a state transition. The state of a simple seria!izer consists of the state of the serializer queues (not including the external queue), the state of the serializer crowds, and the state of the serializer possession. Only the simple serializer events (**enter**, **exit**, **enqueue**, **dequeue**, **join**, **leave**) change the state of possession. Changes in possession that do not alter internal queues or crowds result from **enter** and **leave** events. Changes to internal queues result from **enqueue** and **dequeue** events. Changes to crowds result from **join** and **leave** events. We will return to this point in a later chapter.

In a full semantic model we would have to show where an invocation started and where it terminated. For simplicity, we have chosen to not represent the event that marks the start of an invocation. The **invoke** and **return** events are sufficient to indicate where the resource operations are called, which is all that we need at this point in our discussion.

A **dequeue** event marks a change in state of the indicated queue, and a change in the possession of the serializer. A **dequeue** event for some process will not occur until after the corresponding **enqueue** event, and not until that process is at the head of its queue and the guarantee evaluates to true. The evaluation of guarantees takes place immediately prior to every event that releases possession (**enqueue**, **join**, and **exit** events release possession). For any event E that releases possession, we will assume that evaluation of the guarantees takes place between E and the serializer event immediately preceding E. For simple serializers, where the guarantees are limited to side-effect free evaluation of expressions involving the serializer state, no further events need to be introduced to represent the evaluation of guarantees. If more involved expressions are allowed, events representing such evaluation mus: be introduced.

3.4 Transactions

For a serializer, a *transaction* is a sequence of serializer events that occur for some process in the execution of a serializer operation for some serializer object. The order of events in a transaction is the same as the order in which those events occur in the execution of the serializer operation. Each **enter** event for some serializer object is the first event in some transaction, and each **exit** event is the last event in some transaction. We assign a unique *transaction identifier* at the occurrence of an **enter** **event**.

A transaction may also be viewed as a segment of a process. There may be many transactions involving a serializer object for any particular process, but a transaction can only belong to a single process. The intent of transactions is to capture only the amount of detail about a process necessary to define serializer semantics. Where we formerly used the term process, we will now use the term transaction.

Now that we have identified events as being associated with transactions and nodes, it is notationally convenient to give events a structure. Each event E has several components:

* E.trans - the transaction identifier for the event.

* E.node - the node associated with the event.

* E.kind - the same as E.node.kind.

We can associate possession of the serializer object with a transaction by noting that if there have been more gaining than releasing events for some transaction in some finite history (the difference can only be 0 or 1), then the transaction has possession of the serializer from the last releasing event for that transaction up to the last event in that history.

3.5 Histories

For a serializer, a *history* is a sequence (possibly infinite) of events that represents all events that occur for a particular serializer object. For a given serializer object, there are infinitely many possible histories, depending on the requests sent to that serializer object and on the arbitrary choices possible in selecting dequeue events when several queues are ready.

A history can be viewed as being some interleaving of the transactions involving a serializer object. Every event in a history also belongs to some transaction. The reverse is not true, our model includes histories with incomplete transactions.

Serializer semantics is defined by stating which histories can be produced for any given serializer object. We define a predicate that, given a representation of serializer code and a serializer history, will be true if and only if the history could be produced by the serializer. A history that satisfies that predicate is called a *legal history* for that serializer code. A more complete definition of a legal history occurs later in this chapter.

We assume that the following functions are defined on serializer histories:

Finite(H)
is true if the history is finite; otherwise false.

Size (H)
returns the number of elements in H if H is finite; otherwise is undefined.

Index_set (H)

if H is infinite, returns the set of positive integers; otherwise returns the set of integers $\{N \mid 1 <= N <= Size(H)\}$.

Nth (H, N)

returns the Nth element of H if $N \in$ Index_set(H); otherwise is undefined.

Head (H, N)

returns a prefix of H that is the first N elements of H, provided that $N \in$ Index_set(H); returns the empty sequence if N is 0; otherwise is undefined.

For simplicity, we have chosen to model only those operations that accept a serializer object as an argument. We assume that the serializer object is initially in some initial state, such as that obtained by executing its *create* operation: the resource object is in its initial state, no transaction has possession, and all queues and crowds are empty. The model we have presented is only sufficient to represent operations where possession of the serializer object is gained. For example, the FIFO serializer presented in the previous chapter has three operations; the model we have presented is only sufficient to represent two of them: *read* and *write*.

3.6 Definitions

Predicates will be defined in a dialect of first-order predicate calculus. Functions are defined using a similar syntax, but avoid the use of quantifiers. We call this language the *definition language*, and will refer to it as such in later chapters.

Many of the following definitions are more easily expressed if we have a notation for conditional expressions. The expression "if X then Y else Z" is taken to be Y if X is true (even if Z is undefined), and Z if X is false (even if Y is undefined), and undefined if X is undefined. We also use the "elseif" extension to this notation, as in CLU, to allow convenient syntax for multiple cases. In cases where the "else" clause is omitted, "else true" is assumed (which implies that only boolean conditional expression may omit the "else" clause).

Many of the functions and predicates given below are defined only for finite histories. In our definitions, these functions and predicates are never applied to infinite histories, so there is no need to define them for those cases.

Event E occurs in history H if there is some integer index N such that E is the Nth event of H. Event El precedes event E2 in history H if both El and E2 occur in H, and the index where El occurs is less than the index where E2 occurs.

$$
\begin{aligned}
&\text{Occurs}\,(E, H) \equiv \\
&\quad \exists\, I \in \text{Index\_set}(H):\ E = \text{Nth}(H, I)
\end{aligned}
$$

$$
\begin{aligned}
&\text{Precedes}\,(E1, E2, H) \equiv \\
&\quad \exists\, I, J \in \text{Index\_set}(H): \\
&\qquad\quad I < J \;\&\; E1 = \text{Nth}(H, I) \;\&\; E2 = \text{Nth}(H, J)
\end{aligned}
$$

Note that we have assumed that an event can only occur once in a history. This is implied by later definitions.

As a notational convenience, we introduce the predicate Same_trans(H, I, J), which is true if the Ith and Jth events in history H are from the same transaction. The predicate is undefined if the integers I or J do not belong to Index_set(H).

$$
\begin{aligned}
&\text{Same\_trans}\,(H, I, J) \equiv \\
&\quad \text{Nth}(H, I).\text{trans} = \text{Nth}(H, J).\text{trans}
\end{aligned}
$$

We often need to express the idea that a particular event, or all events for a given node, cannot occur between two given events.

```
Excludes (E1, E2, E, H) ≡
  Precedes(E, E1, H) | Precedes(E2, E, H) | E = E1 | E = E2
```

```
Excludes_node (E1, E2, N, H) ≡
  ∀ I ∈ Index_set(H):
      if Nth(H, I).node = N
        then Excludes(E1, E2, Nth(H, I), H)
```

A slightly more complicated predicate will be needed to specify a more general exclusion predicate (to be used in later chapters). Node_excludes_node (N1, N2, N, H) is true iff no event for a given node N can occur between any two events E1 and E2, where E1.node = N1, E2.node = N2, and E1.trans = E2.trans.

```
Node_excludes_node (N1, N2, N, H) ≡
  ∀ I,J ∈ Index_set(H):
        if( Nth(H, I).node = N1
          & Nth(H, J).node = N2
          & Same_trans(H, I, J) )
        then Excludes_node(Nth(H, I), Nth(H, J), N, H)
```

Intuitively, Node_excludes_node(N1, N2, N, H) expresses the restriction that no event generated by node N occurs between events generated by nodes N1 and N2, where the events from N1 and N2 belong to the same transaction.

We are often interested in the last event of a finite history, or in a history that lacks only the last event of a given finite history. The functions Last and Front are used for notational convenience.

Last (H) ≡ Nth(H, Size(H))

Front (H) ≡ Head(H, Size(H) - 1)

Certain events gain exclusive possession of the serializer, while other events release possession of the serializer. Still other events do not change possession. Gains(E) is true only if the event E gains possession, while Releases(E) is true only if E releases possession.

Gains (E) ≡
  E.kind = **enter** | E.kind = **leave** | E.kind = **dequeue**

Releases (E) ≡
  E.kind = **exit** | E.kind = **join** | E.kind = **enqueue**

A finite serializer history is *busy* if its last event gained possession of the serializer, or if its last event did not release the serializer and the history before that event was busy.

```
Busy (H) ≡
  if Size(H) = 0 then false
    elseif Releases(Last(H)) then false
    else Gains(Last(H)) | Busy(Front(H))
```

The functions Qsize and Csize return the number of transactions using a queue or crowd given the queue or crowd and a finite history.

```
Qsize (Q, H) ≡
  if Size(H) = 0 then 0
    elseif Last(H).kind = enqueue & Last(H).mob = Q
    then Qsize(Front(H)) + 1
    elseif Last(H).kind = dequeue & Last(H).mob = Q
    then Qsize(Front(H)) - 1
    else Qsize(Front(H))
```

```
Csize (C, H) ≡
  if Size(H) = 0 then 0
    elseif Last(H).kind = join & Last(H).mob = C
    then Csize(Front(H)) + 1
    elseif Last(H).kind = leave & Last(H).mob = C
    then Csize(Front(H)) - 1
    else Csize(Front(H))
```

In certain serializer specifications, the *rank* of an event is important. The rank of an event E is an integer that represents the order of E relative to other events occurring at E.node. The first event to occur at a node has rank 1, the second has rank 2, and so on. The rank of an event that does not occur in a history is 0.

```
Rank (H, E) ≡
   if Occurs(H, E)
     then 1 + Rank_scan(H, E, 1)
     else 0
```

In defining Rank, we made use of Rank_scan(H, E, I), which returns the number of events occurring in H at or after event Nth(H, I) and before E with the same node as E.

```
Rank_scan (H, E, I) ≡
   if Nth(H, I) = E then 0
     elseif Nth(H, I).node = E.node
     then 1 + Rank_scan(H, E, I+1)
     else Rank_scan(H, E, I+1)
```

3.6.1 Evaluation of guarantees

Whenever a serializer is released, the guarantees of the non-empty queues are evaluated. The following functions define such evaluation given a finite history and an expression to be evaluated. The notation {G} is used to represent the expression G occurring in serializer code, and distinguishes the expression from our definition notation, since the syntax for expressions and definitions is often similar.

Eval is defined by cases, each case being based on the syntax for boolean expressions. For simple serializers, Eval returns a boolean value, since guarantees are limited to boolean expressions involving tests on the emptiness of queues and crowds.

Eval (H, {G1 & G2}) ≡ Eval(H, {G1}) & Eval(H, {G2})

Eval (H, {G1 | G2}) ≡ Eval(H, {G1}) | Eval(H, {G2})

Eval (H, {~ G}) ≡ ~Eval(H, {G})

Eval (H, {crowd$empty(C)}) ≡ Csize(Var({C}), H) = 0

Eval (H, {queue$empty(Q)}) ≡ Qsize(Var({Q}), H) = 0

Eval (H, {false}) ≡ false

Eval (H, {true}) ≡ true

The Var function (in Var({Q}) and Var({C})) is a mapping from syntactic expressions for queues and crowds to some semantic representation for queues and crowds. We require that the mapping produced by Var is the same mapping that is used to produce the N.mob component of any node N in the history H.

The above definition of Eval is tailored to the needs of defining the semantics of simple serializers. There is no provision for local variables, which would be transaction specific. There is no provision for guarantees with side effects, exceptions, or non-termination, which would require the use of events to mark the state transitions. Further, such provisions would also complicate the definition of the Var function.

3.6.2 Legal histories

A history is legal if it can be produced by some execution of a serializer. Legal(H, S) takes a history and a set of nodes that represent the code for a serializer, and returns true if the history could have been produced from the serializer code. A legal history must be composed of legal steps. That is, each prefix of the history can only be followed by an event that represents a permitted state transition of the serializer.

For a finite history H to be legally followed by the event E, the following rules must be satisfied:

- For E to gain possession of the serializer, then there can be no transaction in possession of the serializer (~Busy(H)).

- If there is a transaction in possession of the serializer, then E must belong to that transaction.

- If E is a dequeue event, its transaction must be at the head of its queue and the guarantee must be true.

- If E is an enter or leave event, there may be no queues such that the front transaction in the queue has a true guarantee.

- All events from a single transaction must occur in the order dictated by legal execution of the code for the operation executed by that transaction. In particular, an **enter** event must be the first event in its transaction.

Note that there are no restrictions explicitly involving **join** and **exit** events. The only restrictions that we impose for these events are expressed by the requirement for "legal execution" of the node graph.

The above conditions lead to the following definitions of Legal and Legal_step, where H is a history, and S is the set of **enter** nodes for the operations of the serializer that require possession.

```
Legal (H, S) ≡
  ∀ N ∈ Index_set(H): Legal_step(Head(H, N-1), Nth(H, N), S)

Legal_step (H, E, S) ≡
  ( (if Gains(E) then ~Busy(H))
   & (if Busy(H) then Last(H).trans = E.trans)
   & (E.kind = dequeue ⊃ Legal_dequeue(H, E))
   & (if E.kind = enter | E.kind = leave then None_ready(H))
   & Legal_transaction_step(H, E)
   & (E.kind = enter ⊃ E.node ∈ Nodes(S)) )
```

The event E is a legal **dequeue** event after the end of history H if the guarantee is true, and the corresponding **enqueue** event is is at the head of its queue in history H.

```
Legal_dequeue (H, E) ≡
  ( Eval(H, E.expr)
   & ∃ I ∈ Index_set(H):
           ( Nth(H, I).node.next = E.node
            & Nth(H, I).trans = E.trans
            & Head_enqueue(H, I) ) )
```

The transaction for the **enqueue** event Nth(H, I) is at the head of its queue if Nth(H, I) is the last event in H for the transaction, and every other **enqueue** event occurring in H before Nth(H, I) has a corresponding **dequeue** event.

```
Head_enqueue (H, I) ≡
  ( In_queue(H, I)
   & ∀ J ∈ Index_set(H):
           if J < I then ~In_same_queue(H, I, J) )
```

In_queue(H, I) is true only if Nth(H, I) is an **enqueue** event that is the last event in H for its transaction.

```
In_queue (H, I) ≡
  ( Nth(H, I).kind = enqueue
   & ∀ J ∈ Index_set(H):
           if J > I then ~Same_trans(H, I, J) )
```

In_same_queue(H, I, J) is true iff Nth(H, I) and Nth(H, J) are **enqueue** events that are the last events in their transactions and the transactions are in the same queue.

```
In_same_queue (H, I, J) ≡
  ( In_queue(H, I)
   & In_queue(H, J)
   & Nth(H, I).node.mob = Nth(H, J).node.mob )
```

None_ready(H) is true if for a particular finite history there is no explicit serializer queue such that the front transaction in the queue has a guarantee that evaluates to true. This predicate is used to define the priority of explicit queues over the single external queue of a serializer.

```
None_ready (H) ≡
  ∀ I ∈ Index_set(H):
        if Head_enqueue(H, I)
          then ~Eval(H, Nth(H, I).node.expr)
```

An event E can be a legal step after some history H only if it can be produced by sequential execution of some transaction. There must not be an event in H with the same transaction and the same node as E; and if E is not an **enter** node, then there must be an event in H from the same transaction as E that results from executing a node for which E.node is the next node.

```
Legal_transaction_step (H, E) ≡
  ( ∀ I ∈ Index_set(H):
        (if E.trans = Nth(H, I).trans
           then E.node ≠ Nth(H, I).node)
    & if E.kind ≠ enter
     then ∃ I ∈ Index_set(H):
              ( E.trans = Nth(H, I).trans
                & E.node = Nth(H, I).node.next) )
```

3.6.3 Complete histories

The set of legal histories for a serializer includes histories where transactions have been started but not completed. Any finite legal history where the serializer state requires further events to occur is termed *incomplete*. All other legal histories are *complete*. A complete finite history is one where no further events are required to occur. Events are required to occur according to the following rules:

The serializer specification language will be interpreted as defining *specification predicates* on complete histories. Serializer code is said to meet its specifications if the specification predicates are true for every complete history of that code.

For a complete history, all events that are required to occur in the history must occur.

- * Whenever a releasing event occurs and there are ready queues, a **dequeue** event from one of those queues is required. Therefore, if H is finite, and the last event in H released possession, then H is only complete if no queues are ready.

- * For every event that gains possession of the serializer, a corresponding event that releases the serializer is required. For simple serializers, every gaining event will be followed by a releasing event. Note that this condition implies that if H is finite and not empty, then Last(H) was a releasing event.

- * For every **join** event, a corresponding **leave** event is required. We assume that every operation of the underlying resource used in a **join** statement will terminate. Such an assumption is part of a modular proof of termination for programs involving serializers.

These conditions lead to the following definition for Complete, where H is a history for some serializer, and S is the set of **enter** nodes for operations of that serializer that require possession.

```
Complete (H, S) ≡
  ( Legal(H, S)
   & (if Finite(H) then None_ready(H))
   & Gain_complete(H)
   & Join_complete(H) )
```

Gain_complete(H) is true if for every gaining event there is a corresponding releasing event that occurs after the gaining event.

```
Gain_complete (H) ≡
  ∀ I ∈ Index_set(H):
       if Gains(Nth(H, I))
         then ∃ J ∈ Index_set(H):
                    Corresponding_release(H, I, J)
```

Corresponding_release (H, I, J) is true if Nth(H, J) is the releasing event that corresponds to the gaining event at Nth(H, I). A releasing event corresponds to a gaining event if both events are in the same transaction, and there are no intervening releasing events for the same transaction.

```
Corresponding_release (H, I, J) ≡
  ( Release_follows(H, I, J)
   & ∀ K ∈ Index_set(H):
            if K < J then ~Release_follows(H, I, K) )
```

Release_follows (H, I, J) is true iff Nth(H, J) is a releasing event that follows the event Nth(H, I); and belongs to the same transaction as Nth(H, I).

```
Release_follows (H, I, J) ≡
  I < J & Same_trans(H, I, J) & Releases(Nth(H, J))
```

Join_complete(H) is true if every **join** event has a corresponding **leave** event. A **leave** event corresponds to a **join** event iff it belongs to the same transaction as the **join** event and there are no intervening **leave** events for the same transaction.

```
Join_complete (H) ≡
  ∀ I ∈ Index_set(H):
      if Nth(H, I).kind = join
        then ∃ J ∈ Index_set(H):
                  ( Leave_follows(H, I, J)
                   & ∀ K ∈ Index_set(H):
                           if K < J
                             then ~Leave_follows(H, I, K) )
```

Leave_follows (H, I, J) is true iff Nth(H, J) is a **leave** event that follows the event Nth(H, I), and belongs to the same transaction as Nth(H, I).

```
Leave_follows (H, I, J) ≡
  I < J & Same_trans(H, I, J) & Nth(H, J).kind = leave
```

3.7 Serializer Induction

In CLU, a cluster that implements a data type does so by providing operations that manipulate objects of a representation type. For every abstract object, there is a representation object. In designing and verifying clusters, it has been found to be useful to make use of a *representation invariant* [Guttag, Horowitz and Musser 78] that must hold for all objects supported by the cluster. This representation invariant should be true whenever a representation object is created, and it should be maintained by all operations.

To prove that the representation invariant holds, we need to use induction on the sequence of operations performed. The induction principle we use is that if P is true at the start of the abstract object's lifetime, and assuming P for an object at the start of an operation implies that P is true at the end of the operation, then P is true of that object before and after every operation. As in [Guttag, Horowitz and Musser 78], we will call this *data type induction.*[11]

To show the soundness of data type induction, we need to show that if P is true of an object after any operation of the cluster, then P is true of the object before any other operation of the cluster, provided that there were no intervening operations of the cluster. Informally, to use data type induction using some predicate P, it should not be possible for actions of other programs to make P invalid. It is possible in CLU to write clusters such that data type induction can be used to prove reasonable predicates about their objects. A cluster with this property is said to have an *isolated representation* [Atkinson 76]. While the cluster construct is not strictly necessary if one wishes to use data type induction, it facilitates the determination of an isolated representation.

As presented in this thesis, the serializer construct is quite similar to the cluster construct. Both can implement abstract types, and both do so by manipulating objects of a representation type through operations that can have sole access to the

[11] Also know as *generator induction* in [Wegbreit and Spitzen 76].

representation objects. Since serializers provide the same kind of representation protection as clusters do, we can use data type induction, in part, to verify serializers.

We call the application of data type induction to histories *serializer induction.* For any complete history H, serializer induction can be expressed as:

```
if
  ( P(Head(H, 0))
   & ∀ I,J ∈ Index_set(H):
           (if (Gains(H, I)
              & Corresponding_release(H, I, J)
              & P(Head(H, I-1))
             then P(Head(H, J))) )

then
  ∀ K ∈ Index_set(H):
        if Gains(Nth(H, K)) then P(Head(H, K-1))
```

The predicate P is intended to be defined on finite histories where no transaction is in possession of the serializer at the end of the history.

History induction is applicable for any serializer where the predicate P will hold from the event where possession is released to the next event where possession is gained. We can express this condition as:

```
∀ I,J ∈ Index_set(H):
        if ( Gains(Nth(H, I))
           & Releases(Nth(H, J))
           & Nth(H, J).node.next = Nth(H, I).node
           & P(Head(H, J-1)) )
        then P(Head(H, I))
```

We call this the *isolation condition.* Just as the cluster construct facilitates but does not fully enforce an isolated representation, the serializer construct does not necessarily enforce the isolation condition.

The serializers we will be specifying and proving satisfy the isolation condition. In view of this, there is no provision in the histories for events that occur external to serializers. We have not provided for situations that we have been unable to prohibit in the programming language, but believe to be bad practice.

An example of serializer induction is the use of a representation invariant for the FIFO readers-writers problem presented in the previous chapter. A simple invariant for an object X of type **rep** for any finite history H is:

$$\mathrm{Csize}(X.rc, H) = 0 \mid \mathrm{Csize}(X.wc, H) = 0$$

While this invariant is not the strongest we can prove, it is a useful property that can be proven simply.

As a reminder, the code for the read operation is (briefly):

```
enqueue x.q until crowd$empty(x.wc)
join x.rc; ... end
```

while the code for the write operation is:

```
enqueue x.q until crowd$empty(x.wc) & crowd$empty(x.rc)
join x.wc; ... end
```

Informally, we can prove the invariant by cases. First, suppose that we have

$$C1 \equiv \text{Csize}(H, X.rc) > 0 \supset \text{Csize}(H, X.wc) = 0,$$
$$C2 \equiv \text{Csize}(H, X.wc) > 0 \supset \text{Csize}(H, X.rc) = 0,$$

where the history prefix is understood. Since Csize always results in a non-negative integer, the condition C1 & C2 implies the invariant. Initially, both crowds are empty, so the invariant is trivially true. To prove C1, we assume that C1 is true immediately prior to some gaining event, and show that it is maintained immediately after any releasing event. An examination of the code shows that the only sequence of events that can increase Csize(X.wc) is where some writer dequeues and joins the writer crowd. Therefore, the only way that C1 could be false is to allow some writer to dequeue when Csize(X.rc) > 0. However, the guarantee for the writer transaction prohibits the event from occurring until Csize(X.rc) = 0. Therefore, C1 is maintained. Condition C2 is proved similarly. Therefore the invariant is maintained.

3.8 Comments on enter and leave events

One simplification made in the model is based on the use of **enter** and **leave** events. A reasonable requirement on **enter** events is that they will occur if they have been requested. The only requirement that we have on **leave** events is that they will eventually occur if the corresponding **join** has occurred. Yet after completing the resource operation, the **leave** event must be requested, since some other transaction may be in possession. The simplification we have made is not to represent requests for **enter** or **leave** events as separate events.

One requirement that this places on serializers is that code executed while a transaction has possession of the serializer must terminate, since otherwise a request for possession could not be satisfied. Termination while in possession is trivially satisfied for simple serializers.

We have also assumed that there is some scheduling discipline on requests for possession of the serializer so that a request for an **enter** or **leave** event will not be forever delayed by other such requests. A FIFO discipline on all such requests may be overly strict in some systems, and we do not require it. Any discipline that guarantees service to requests for possession will be satisfactory. We make no attempt to prove this requirement in general.

Adding specific events to the model to indicate when **enter** and **leave** events have been requested is only necessary to represent undesirable cases such as non-termination while in possession, or a pathological scheduler. Further, it is not reasonable to include such events in the specifications or proof techniques, since their order of occurrence is not affected by possession of the serializer object.

3.9 Message passing semantics

The model we have presented in this chapter has been deliberately incomplete. The larger semantic model we have assumed uses procedure calls and processes, and is well-suited for describing the use of serializers in a system where multiple processes communicate through shared memory. While having a certain intuitive appeal, particularly to those familiar with monitors, the techniques we have used (and will use) are applicable when a larger programming language and larger semantic model are used.

In this section we will sketch a model based on *message passing*. Such a model has been proposed by various people [Greif and Hewitt 75, Hewitt and Baker 77, Good, Cohen and Keeton-Williams 79]. A similar model is used to describe distributed systems [Svobodova, Liskov and Clark 79, Liskov 79]. We believe that the structure of serializers is quite useful in organizing programs in these distributed systems, and will address some further implications of serializers in such an environment in our conclusions.

In the message-passing model, separate entities communicate by passing messages rather than by sharing memory among many processes. Of course, when the same physical entity receives messages from various sources, the effect of a shared memory is achieved. We can think of a serializer object as one such entity, the resource object as another entity, and the originators of messages to the serializer as other entities. In such a model, serializer objects are message switchers: They affect when a message gets passed to a resource, but not the message itself, nor its reply.

We imagine that serializers are used in a programming language that supports a logical network, where there are logical *sites*, each of which has its own local objects. Each site can communicate with another site only by sending messages to that other site. We assume that each site can send messages to any other site without regard to physical connections. Unlike physical sites in a network, logical sites can be freely created at relatively low cost, up to the limitations of the implementation.

In such a logical network, each serializer object is a separate site. Further, each resource object is a separate site. Instead of saying that a process is executing serializer code, however, we say that a site executes code for some transaction. Local variables are associated with the transaction, and representation components are associated with the site.

The following description of the serializer construct in a message passing model gives an outline of an abstract implementation for serializers. At serializer object creation, the representation object is initialized, and the serializer site waits for external messages to arrive. We describe the serializer events as follows:

* **enter** - An **enter** event represents the acceptance of an initial request message for service at the serializer site. At this acceptance, a unique transaction identifier is generated to name the transaction that this event starts. The request message identifies the operation to execute, the arguments to that operation, and the destination for the reply. A

destination is a site name and a transaction identifier relative to that site.

- **enqueue** - The **enqueue** event represents the completion of a series of actions. First, the transaction identifier, the guarantee, and the continuation point are placed in the named queue. Then the guarantees at the head of the internal queues are evaluated to determine the next transaction to service. If there are ready queues, the serializer site selects one of them as the next to process and releases possession. If there are no ready queues, the serializer site releases possession and accepts the next external message.

- **dequeue** - After the **dequeue** event, possession has been regained by the transaction, the enqueued information has been removed from the queue, and the serializer site will continue to execute code for that transaction at the given continuation point.

- **join** - The **join** event also represents completion of a series of actions. First, the transaction identifier and the continuation point are placed in the named crowd. Then a message is sent to the resource site,[12] requesting the operation and arguments desired. The message sent to the resource site indicates the serializer site as the destination, and also names the transaction being processed. Finally, as for the **enqueue** event, the guarantees are examined and possession is released.

- **leave** - A **leave** event represents an acceptance of a reply message from the resource site. Possession is regained by the transaction named in the reply. The information associated with that transaction in the named crowd is removed from that crowd. The serializer site continues to execute code for the transaction at the continuation point.

- **exit** - An **exit** event represents the completion of a series of actions. First, a reply message is sent to the destination given in the **enter** event. For simple serializers, the information in this reply is taken from the reply received at the **leave** event. Then the guarantees are evaluated and

[12] For simplicity, we will assume that the only code that can appear in the body of a **join** statement will be an invocation of a resource operation.

possession released, as for the **enqueue** and **join** events.

The above discussion has presented a very simple view of serializers in a distributed system. However, we believe that extensions to this model will not greatly affect our description of serializer events. For example, we have assumed that there is no more than one request outstanding at a time, so that the site name and transaction identifier are sufficient to specify a destination. A natural extension would be to allow several requests to be outstanding. In such a case, a request number relative to the transaction can be included in the destination.

3.10 Infinite histories revisited

We noted in our introduction that states can be regarded as equivalence classes of histories, a view advocated in [Greif75] (although Greif discusses partial orders of events rather than sequences of events). However, this approach does not easily deal with infinite histories, since the state predicates (such as Csize and Qsize) are not defined on infinite histories. It would be convenient if we could avoid introducing infinite histories, but we have not yet discovered a method that does not require them. We introduced infinite histories to model what happens to a serializer object over its entire lifetime. Some serializer objects are intended to have unbounded lifetimes, even though any physically realizable system must have a finite lifetime.

If we reject the use of infinite histories, then we consider the specification clauses to be requirements that all finite complete histories must satisfy. Unfortunately, this leads to difficulties with showing that the "starving" readers-writers solution could not satisfy the guaranteed service specifications, since the counterexamples involve infinite histories where certain events are not required to occur. If the only histories considered to be complete are finite histories where after the last event all crowds are empty and no queues are ready, then the starving readers-writers solution can be proven to guarantee service. The system designer who relied on this proof would be unpleasantly surprised to discover that starvation actually occurred under heavy loads.

4. Specification language

One method of specifying a programming language is to provide rules for translating programs written in that language into functions on some mathematical domain. This method can also be applied to specification languages. The specification language for serializers is composed of clauses in which certain relations between serializer events imply other relations between serializer events. The meaning of specification clauses is given by stating rules for transforming the clauses into *specification predicates* on histories.

Serializer code is said to meet its specifications if every complete history that can be legally generated by the serializer code (according to the partial legality predicate discussed in the previous chapter) satisfies all of the specification predicates that result from the specification clauses for that serializer code.

It is *not* our intention to require that the specification language have sufficient power to define abstract data types. We are only concerned with specifying concurrency control. We believe that the difficulty of arriving at good specification methods dictates that we attack a tractable problem, and integrate the various approaches as they are sufficiently well understood.

In this chapter we discuss the kinds of serializer specifications supported, and present the syntax and semantics of the specification language. Then we give a full specification for the FIFO readers-writers serializer, some specifications for variations on the readers-writers problem, and a partial specification for the bounded buffer problem.

4.1 Kinds of serializer specifications

The specification language is a notation for requiring a serializer abstraction to have certain properties. These properties are classified as:

- Exclusion - where one kind of access excludes another, such as readers excluding writers in a simple data base. This kind of specification is necessary to prevent concurrent requests from interfering with each other.

- Priority - where one transaction is served preferentially over another. This may occur because of the order of **enter** events, the kind of transaction, or other reasons or combinations of reasons.

- Concurrency - where some accesses are required to be served concurrently. The presence of concurrent processing for requests often affects the performance of system, and may even affect the correctness.

- Service - where some (or all) accesses are required to run to completion (analogous to requiring termination for sequential programs).

We make no claim that all interesting synchronization properties fall into the above categories, although many do. We also make no claim that all properties in the above classes can be expressed in the specification language, or that the specifications are especially concise in our language. The classes we have chosen are not necessarily distinct; some properties may be considered to be in more than one class. We are more interested in making the specification language usable by both programmers and verification systems than attaining some kind of formal completeness.

The specification language has nothing to say about performance, either for real time, computation time or storage. Although performance characteristics can be inferred from some of our specifications, specifications and proofs of performance are beyond the scope of this thesis.

The simple form of the specification language does not deal with the values passed to or from serializer operations. This simplification has been made to avoid discussing what the exact meaning of "value" is in the language. The form of the specification language in this chapter has events, nodes, boolean and integer values. We also include limited predicates on these values, and simple arithmetic expressions as functions on integers. It is possible to extend the specification language that the user sees to include further values and functions, but such extensions involve more of the semantics of the complete programming language than we wish to handle in this thesis. In the next chapter, certain extensions are made to the specification language to support our verification techniques, but these extensions are still quite limited, and do not support user-defined values and functions.

4.2 Specification language

The specification language is defined by specifying a mapping from specification clauses to *unbound specification predicates.* Each unbound specification predicate takes a *symbol map* and a history into a boolean that indicates whether the specification clause is satisfied for that symbol map and that history.

A symbol map is a function from event symbols to events, and from node symbols to nodes. It provides an *interpretation* in our semantic model of the symbols in the specification clause. A *valid symbol map* provides a consistent interpretation of symbols for a given history, and will be discussed further later in this chapter. The symbol map is an important distinction between the specification language and the definition language.

Each specification clause defines a specification predicate, which maps histories to boolean values: true if the clause is satisfied for that history, and false if it is not. The specification predicate for a clause is the value of the unbound specification predicate for that clause taken over every valid symbol map for a given history.

4.2.1 Syntax of specification language

The specification language has a simple syntax. The specifications for serializer code are expressed as a set of clauses, each clause being expressed as an implication. The syntax of the specification language is given informally below, issues of parenthesization and precedence being neglected.

```
Clause = Clause "⊃" Clause
     | Ordering_clause
     | Clause "&" Clause
     | Clause "|" Clause
     | "~" Clause
     | "GX" "(" Event_symbol "," Event_symbol "," Node_symbol ")"
     | "GX" "(" Event_symbol "," Event_symbol "," Event_symbol ")"
     | "@" Event_symbol
     | Expr Order_Op Expr

Ordering_clause = Event_symbol "<" Event_symbol
     | Event_symbol "<" Ordering_clause

Order_op = "<" | ">" | "≤" | "≥" | "=" | "≠"

Expr =  literal
     | Expr "-" Expr
     | Expr "+" Expr
     | Expr "*" Expr
     | Expr "/" Expr
     | "#" Event_symbol
```

An event symbol (Event_symbol above) is written by writing a transaction symbol followed by the event kind followed by optional information indicating other components of the event (with optional digits for further disambiguation). A transaction symbol is written by giving the first letter of the operation name (or enough letters to be unambiguous) followed by optional digits if more than one transaction for that operation is needed in the clause. Examples of event symbols for an operation whose name starts with 'X' are:

* X-enter: This symbol denotes an **enter** event for transaction X. By convention, if there is only one transaction appearing in a specification clause for the operation, no digits are necessary in the transaction symbol. There can be only one **enter** event for any transaction.

* X-join: This symbol denotes a **join** event for transaction X. For simple serializers, this **join** event is associated with performing the corresponding operation on the resource. Also, for simple serializers, we are limited to having one **join** event for any given transaction.

* X1-exit: This symbol denotes an **exit** event for transaction X1. Note the use of the digit '1' to indicate a transaction that is distinct from X (or X2). By convention, we give different transactions different digits in specification clauses where more than one transaction for an operation is mentioned.

* X2-enqueue(s.q): This symbol denotes a **enqueue** event for transaction X2, where the queue denoted by *s.q* is used.

A node symbol (Node_symbol above) is written by giving the first letter(s) of the transaction name, followed by a "*", followed by the event kind. For example, the **enter** node for operation X is written as X*-enter. Any further information given is the same as the corresponding event.

4.2.2 Semantics of specification language

We first must describe the domains over which the specification language is defined.[13] The syntax given above mentions event and node symbols, but does not explicitly demand that the symbols apply to a single serializer. Therefore, we need to limit ourselves to nodes and events chosen from some particular serializer, S. We name these domains (and representative elements) by:

$\tilde{n} \in \tilde{N}_S$ -- node symbols for S

$\tilde{e} \in \tilde{E}_S$ -- event symbols for S

$c \in C_S$ -- specification clauses for S

$x \in X_S$ -- expressions for S

Note that we have provided single character names for sample elements of the domains. We will follov. the leading character convention used in naming events for naming elements of these domains in the later equations, including using trailing digits where more than one element is desired.

The semantic domains are those domains described in the previous chapter on the semantic model.

$n \in N_S$ -- nodes for S

$e \in E_S$ -- events for S

[13] Although the denotational method used in this thesis to define the specification language owes much to work by Scott and Strachey [Scott and Strachey 71, Strachey and Wadsworth 74], the domains we use are simply sets, not lattices.

$h \in H_S$ -- complete histories for S

$$(H_S\text{: Int} \rightarrow E_S)$$

In specifying the meaning of the specification language it is necessary to provide a symbol map that takes node and event symbols into their meanings. We will discuss this function at greater length below.

$p \in P_S$: maps symbols to events or nodes

$$(P_S\text{: } \tilde{N}_S \cup \tilde{E}_S \rightarrow (N_S \cup E_S))$$

The following functions take syntactic values into semantic values. We say that they define the *meaning* of the syntactic constructs in the specification language. We have avoided parsing and precedence issues to more clearly present these functions. Note that the braces "{ }" are used to bracket syntactic constructs and distinguish them from the semantic expressions.

$\boldsymbol{E}(\{\tilde{e}\},p)$ -- event corresponding to $\tilde{e}$ in map p

$\boldsymbol{E}$: $(\tilde{E}_S, P_S) \rightarrow E_S$

$\boldsymbol{N}(\{\tilde{n}\},p)$ -- node corresponding to $\tilde{n}$ in map p

$\boldsymbol{N}$: $(\tilde{N}_S, P_S) \rightarrow N_S$

$\boldsymbol{C}(\{c\},p,h)$ -- validity of specification clause c in map p, history h

(true if c is satisfied, false if not)

$\boldsymbol{C}$: $(C_S, P_S, H_S) \rightarrow$ Bool

$\boldsymbol{X}(\{x\},p,h)$ -- value of expression x in map p, history h

(an integer value)

$\mathcal{X}$: (X_S, P_S, H_S) -> Int

$\mathcal{O}(\{op\},p)$ -- binary predicate corresponding to *op*

$(Op = \{<, >, \leq, \geq, =, \neq\})$

$\mathcal{O}$: Op -> ((Int, Int) -> Bool)

The definition of $\mathcal{C}(\{C\},p,h)$ for specification clause C is given below by cases.

$$\begin{aligned}
\mathcal{C}(\{c1 \supset c2\},p,h) &= \mathcal{C}(\{c1\},p,h) \supset \mathcal{C}(\{c2\},p,h) \\
\mathcal{C}(\{\tilde{e1} < \tilde{e2}\},p,h) &= \text{Precedes}(\mathcal{E}(\{\tilde{e1}\},p), \mathcal{E}(\{\tilde{e2}\},p), h) \\
\mathcal{C}(\{c1 \,\&\, c2\},p,h) &= \mathcal{C}(\{c1\},p,h) \,\&\, \mathcal{C}(\{c2\},p,h) \\
\mathcal{C}(\{c1 \mid c2\},p,h) &= \mathcal{C}(\{c1\},p,h) \mid \mathcal{C}(\{c2\},p,h) \\
\mathcal{C}(\{\sim c\},p,h) &= \sim\mathcal{C}(\{c\},p,h) \\
\mathcal{C}(\{GX(\tilde{e1}, \tilde{e2}, \tilde{n})\},p,h) &= \text{Excludes\_node}(\mathcal{E}(\{\tilde{e1}\},p), \mathcal{E}(\{\tilde{e2}\},p), \mathcal{N}(\{\tilde{n}\},p), h) \\
\mathcal{C}(\{GX(\tilde{e1}, \tilde{e2}, \tilde{e})\},p,h) &= \text{Excludes}(\mathcal{E}(\{\tilde{e1}\},p), \mathcal{E}(\{\tilde{e2}\},p), \mathcal{E}(\{\tilde{e}\},p), h) \\
\mathcal{C}(\{@\tilde{e}\},p,h) &= \text{Occurs}(\mathcal{E}(\{\tilde{e1}\},p), h) \\
\mathcal{C}(\{x1 \; op \; x2\},p,h) &= \mathcal{O}(\{op\},p)(\mathcal{C}(\{x1\},p,h), \mathcal{C}(\{x2\},p,h))
\end{aligned}$$

The definition of $X(\{x\},p,h)$ is given below by cases:

$$\begin{aligned}
X(\{x1 + x2\},p,h) &= X(\{x1\},p,h) + X(\{x2\},p,h)\\
X(\{x1 - x2\},p,h) &= X(\{x1\},p,h) - X(\{x2\},p,h)\\
X(\{x1 * x2\},p,h) &= X(\{x1\},p,h) * X(\{x2\},p,h)\\
X(\{x1 / x2\},p,h) &= X(\{x1\},p,h) / X(\{x2\},p,h)\\
X(\{\mathit{literal}\},p,h) &= \mathit{constant}\\
X(\{\#\tilde{e}\},p,h) &= \mathrm{Rank}(h, E(\{\tilde{e}\},p))
\end{aligned}$$

As a notational convenience, the clause "E1 < E2 < E3" is equivalent to "E1 < E2 & E2 < E3". Longer clauses of the same form are defined similarly.

Some examples of specification clauses follow:

X1-join < X2-join ⊃ X1-leave < X2-join

> This clause mentions two transactions, X1 and X2. The intention is to specify that having transaction X1 access the resource prohibits X2 from accessing the resource.

@X-enter ⊃ @X-exit

> This clause is a specification of service for transaction X. The occurrence of the X-enter event implies that the X-exit event occurs in any complete history.

@G-enter & (#G-enter ≤ #P-enter) ⊃ @G-exit

> If the **enter** event for transaction G occurs, and the rank of G-enter is not greater than the rank of the **enter** event for transaction P, then the **exit** event for transaction G must occur. In (slightly) more intuitive terms, a transaction for operation G is only required to receive service if there are at least as many transactions for operation

P as transactions for operation G.

4.3 The symbol map

Mapping symbols in the specification clauses to mathematical entities is a necessary part of translating specification clauses into functions on histories. It is necessary to map event symbols into events, node symbols into nodes, and syntactic expressions into their value domains.

The meaning of a specification clause is taken to be a predicate that, given a history, returns true if a history satisfies the specification, and false if it does not. Serializer code is said to satisfy a specification clause if, for every complete history and every valid symbol map for that history, the specification predicate defined by that clause is true for the history.

A valid symbol map for serializer S must satisfy the following restrictions:

- Distinct event symbols must map to distinct events, and distinct node symbols must map to distinct nodes.

- Event symbols must be consistent with node symbols. For example, the event symbol "R-enter" must map to an event that is consistent with the node symbol "R*-enter".

- Event and node symbols map to events and nodes that are consistent in kind to the symbol kinds. For example, the node symbol "R*-enter" must map to a node that is an **enter** node in the serializer S.

- * Event and node symbols map to events and nodes that are consistent in transactions to the transaction symbols. For example, the event symbols "R1-enter" and "R1-exit" must map to events with the same transaction.

- * Event symbols mentioned in ordering clauses (E1 < E2) and GX clauses (GX(E1, E2, E)) must map to events that actually occur in the history. Event symbols mentioned in rank expressions (#E) and occurrence clauses (@E) need not occur in the history.

The last restriction on symbol maps needs further explanation. The motivation for introducing it is to keep specifications of order separate from specifications of service. For example, suppose that we are attempting to specify a readers-writers serializer where writers are given priority over other writers solely on the basis of when enter events occurred. To do this, we use the following specification:

W1-enter < W2-enter ⊃ W1-exit < W2-exit

However, if the last restriction does not hold, and we therefore allow symbol maps where the events corresponding to W1-enter and W2-enter occur in the given order for some history, but either of the events corresponding to W1-exit or W2-exit have not occurred, then the specification clause will have a much different meaning. If the event occurrence is optional for the symbol map, then a serializer will satisfy the clause if the given order holds, *and* the serializer guarantees service to writers, but *not* if writers can starve. In this rather surprising way, a priority specification has implied a service specification.

We believe that keeping the specification of order separate from the specification of service simplifies both specifications and proofs. Therefore, we have required that a symbol map is valid for some history only if an event symbol in an ordering or GX clause maps to an event that actually occurs in the history.

4.4 Readers-writers specifications

Our first examples deal with the readers-writers problem. In this problem, a serializer abstraction should allow concurrent access to a simple data base for transactions that simply read from the data base, but should not allow transactions that write to the data base to overlap, since that could destroy the integrity of the data.

The same exclusion specifications apply to all versions of the readers-writers problem.

- Readers exclude Writers - A reader accessing the resource prevents a writer from accessing the resource.

  $\text{R-join} < \text{W-join} \supset \text{R-leave} < \text{W-join}$

- Writers exclude Readers - A writer accessing the resource prevents a reader from accessing the resource.

  $\text{W-join} < \text{R-join} \supset \text{W-leave} < \text{R-join}$

- Writers exclude Writers - A writer accessing the resource prevents another writer from accessing the resource.

  $\text{W1-join} < \text{W2-join} \supset \text{W1-leave} < \text{W2-join}$

For the FIFO readers-writers serializer shown in Chapter 2, the priority given to a transaction is based on when it arrived with respect to other transactions. We expect strict FIFO ordering between readers and writers, and between writers and writers. Strict priority between readers is not required, because readers may access the resource concurrently. Therefore, we have the following priority specifications:

* Readers not pre-empted by writers.

$\text{R-enter} < \text{W-enter} \supset \text{R-join} < \text{W-join}$

* Writers not pre-empted by readers.

$\text{W-enter} < \text{R-enter} \supset \text{W-join} < \text{R-join}$

* Writers not pre-empted by other writers.

$\text{W1-enter} < \text{W2-enter} \supset \text{W1-join} < \text{W2-join}$

The above priority specifications only require the order of requests to be preserved from **enter** events to **join** events, not from **leave** events to **exit** events. If the order of service matters after the resource operation is performed, then we would include the following clauses:

$\text{R-enter} < \text{W-enter} \supset \text{R-exit} < \text{W-exit}$

$\text{W-enter} < \text{R-enter} \supset \text{W-exit} < \text{R-exit}$

$\text{W1-enter} < \text{W2-enter} \supset \text{W1-exit} < \text{W2-exit}$

In the readers-writers case, we specify concurrency for readers by the following specification:

$$\begin{aligned}&\mathrm{GX(R1\text{-}enter,\ R2\text{-}enter,\ W{*}\text{-}enter)\ \&\ R2\text{-}enter} < \mathrm{R1\text{-}leave}\\ &\supset \mathrm{R2\text{-}join} < \mathrm{R1\text{-}leave}\end{aligned}$$

This clause is interpreted as requiring that for any two readers, R1 and R2, that enter the resource without a writer entering the resource between R1 and R2, if R2 enters before R1 has completed accessing the resource, then R2 will begin to access the resource before R1 completes its access.

We cannot require that two readers are actually concurrently executing resource operations, since actual concurrency may depend on the scheduling policy followed on a multi-processed machine, or on the relative speeds of two processors if the requests are executed by separate machines, or on further concurrency limitations imposed by the resource. The kind of specification that we must settle for is to require that both requests are sent to the resource (in **join** events) before either reply from the resource is acknowledged (in **leave** events). A concurrency specification only requires the opportunity for concurrent execution, unhindered by the serializer.

The specifications of service for readers and writers are simply that for every **enter** event there should be a corresponding **exit** event, and that this should hold for both readers and writers. The specification clauses are:

$$\begin{aligned}&\mathrm{@R\text{-}enter} \supset \mathrm{@R\text{-}exit}\\ &\mathrm{@W\text{-}enter} \supset \mathrm{@W\text{-}exit}\end{aligned}$$

4.5 Variations of the readers-writers problem

Other versions of the readers-writers problem exist [Courtois, Heymans and Parnas 71, Greif 75]. Aside from differences based on the programming language used, the versions differ mostly because of the kinds of priority they give to readers or writers and the presence or absence of starvation.

The simplest priority specifications often conflict with other specifications. For example, suppose that the person specifying the serializer wants to give writers priority. The intention might be: "whenever a writer enters a serializer before a reader has been serviced, the writer should be serviced before the reader." This specification can be written as:

$$\text{W-enter} < \text{R-join} \supset \text{W-join} < \text{R-join}$$

Further, we can write serializer code that will realize this specification. Unfortunately, if writers arrive at the serializer at a sufficiently high rate with respect to the length of time the resource$write takes, readers can be indefinitely prohibited from joining the resource. This would conflict with the guaranteed service requirement given above, since there can be no specification that prohibits writers from arriving at the resource.

A more reasonable specification of writer's priority is to require "if a reader and a writer enter the serializer while a particular other writer is being serviced, then the writer will be serviced before the reader." This specification can be written as:

$$(\text{W1-join} < \text{W2-enter} < \text{W1-leave} \ \& \ \text{W1-join} < \text{R-enter} < \text{W1-leave})$$
$$\supset \text{W1-join} < \text{R-join}$$

This specification does not conflict with our service specifications. Regardless of the number of writers that enter while resource$write is being performed for W1, the readers that entered in that period need not be delayed for any writers arriving after that period.

The guaranteed concurrency specifications may also differ from serializer to serializer. We may wish to require for the readers-priority serializer that all readers that enter while a writer is accessing the resource will be allowed to concurrently access the resource. This specification can be written as:

$$(\text{W-join} < \text{R1-enter} < \text{W-leave} \ \& \ \text{W-join} < \text{R2-enter} < \text{W-leave})$$
$$\supset (\text{R2-join} < \text{R1-leave} \ \& \ \text{R1-join} < \text{R2-leave})$$

This clause requires that for every pair of readers, R1 and R2, entering the serializer while a writer is accessing the resource, that both readers begin to access the resource before either reply is acknowledged.

4.6 Bounded Buffer Specifications

The bounded buffer problem[14] is based on operating system I/O buffering. We assume that there is a producer of information, and a consumer of information. The producer issues *put* requests to the system to pass the information to the consumer, and the consumer issues *get* requests to obtain the items of information from the system. In order to allow both producer and consumer to operate in parallel, the system provides a bounded buffer of length N to store items of information that the producer has delivered to the system before the consumer has requested them. The producer can proceed as long as it is no more than N items ahead of the consumer.

We have somewhat generalized the problem by allowing multiple consumer and producer processes for each bounded buffer. If the producer consists of several processes, then each process can proceed until it performs a *put* request where the request is made on a full buffer. Similarly, each consumer process can proceed until it performs a *get* request on an empty buffer.

We assume that the resource acts as a bounded sequence of information items,[15] where the sequence cannot be more than N items long. The *put* operation appends an item to the head of the sequence, while *get* operation removes an item from the tail of the sequence.

[14] A monitor approach to this problem appears in [Howard 76]. Serializer code for this problem appears in the appendix to this thesis, and is discussed in our conclusions.

[15] Although this kind of sequence is also known as a queue, we avoid the use of the term to distinguish between the queues used by the serializer code for scheduling, and the queue used for the data.

The following specifications are conditional service specifications for the bounded buffer problem.

$$((\#\text{G-enter} + N \geq \#\text{P-enter}) \;\&\; @\text{P-enter}) \supset @\text{P-exit}$$

$$((\#\text{P-enter} \geq \#\text{G-enter}) \;\&\; @\text{G-enter}) \supset @\text{G-exit}$$

The G-enter event is the initial event of some *get* transaction, and the P-enter event is the initial event of some *put* transaction. We require that the P transaction complete if there have been enough G transactions to use the data, or if there is sufficient room in the buffer to store the data. If the G-enter event is the i-th event using the G*-enter node, and the P-enter event is the j-th event using the P*-enter node, then P must complete if $j \leq i+N$. Similarly, we require that a G transaction complete if there have been enough P transactions started to supply the data. Therefore, G will complete if $i \leq j$.

Note that the above specifications need to use @G-enter and @P-enter because we only automatically require events appearing in ordering specifications to occur in the histories. This choice was made based on the convenience of writing certain examples. To illustrate, if the use of #G-enter required @G-enter, then the specification of service for P transactions above would have been written as two clauses:

$$(\sim @\text{G-enter} \;\&\; (\#\text{P-enter} \leq N)) \supset @\text{P-exit}$$

$$(\#\text{G-enter} + N \geq \#\text{P-enter}) \supset @\text{P-exit}$$

Another specification of the bounded buffer problem is that the order of *get* requests and *put* requests cannot be interchanged, either in forwarding the request to the resource, or in returning the result. These specifications are similar to the FIFO readers-writers priority specifications.

G1-enter $<$ G2-enter $\supset$ (G1-join $<$ G2-join & G1-exit $<$ G2-exit)

P1-enter $<$ P2-enter $\supset$ (P1-join $<$ P2-join & P1-exit $<$ P2-exit)

We have chosen the exclusion specifications to be quite simple: accessing the resource is exclusive. The exclusion specifications are expressed by the following four clauses.

G1-join $<$ G2-join $\supset$ G1-leave $<$ G2-join

G-join $<$ P-join $\supset$ G-leave $<$ P-join

P1-join $<$ P2-join $\supset$ P1-leave $<$ P2-join

P-join $<$ G-join $\supset$ P-leave $<$ G-join

We have said that the serializer operations should, as far as practical, have the same effect as the resource operations. In the bounded buffer problem, the serializer operations have the same effect as the cluster operations *provided* that the cluster operations return normally. In executing a *put* operation for the serializer, if there is no room in the bounded buffer for the item, the operation pauses until there is room. In executing a *get* operation, the operation will not proceed until an item is available. For the operations of the resource, however, an exception is signalled if there is no room in the buffer when executing a *put* operation, or if no item is present when executing a *get* operation. The signals of the resource operations have become the non-terminations of the serializer operations. This raises the question of how well we have separated concurrency control from data access. We will discuss this question in the conclusions.

We have presented the bounded buffer problem as an illustration of the specification language and as an example of a serializer that is slightly beyond simple serializers. We will return to this example to illustrate how we can perform extensions in the program proving domain as well.

5. Verification Rules

In previous chapters we have used a definition language based on first-order predicate calculus to give the meaning of both the serializer construct and the serializer specification language. In theory, we need nothing else to verify that a serializer meets its specifications. In practice, a certain amount of intermediate work is necessary.

We have chosen to build a verifier that operates in a restricted domain. The verifier applies rules that are specific to this domain to data it has describing a serializer and specifications for that serializer. This chapter states and proves those rules. Our choice of rules is based on their utility in verifying a number of variations of the readers-writers problem (these examples are presented in the next chapter). No claims will be made for their completeness. Other classes of problems would most likely lead to different sets of rules, although we would expect most such rule sets to have substantial intersections with the set we have chosen.

In this chapter, we first argue that proofs can be reasonably performed in an extended specification language. We then state and prove a number of verification rules expressed in the extended specification language. These rules are used in a program that performs automatic verification of serializers, to be discussed in the next chapter. A method for proving service specifications is then presented that is partially based on these rules, and its correctness argued. To illustrate the use of the verification rules, an example of a rule-based proof is given. Finally, certain weaknesses of our methods are examined.

5.1 Proving in the specification language

In proving that a serializer meets its specifications we start with the text for a serializer and a number of specification clauses. In proving that serializer code meets its specifications we need to state intermediate propositions about the serializer code and the specifications. To do so we need a language to state the propositions and rules of inference that can be used for the language.

One candidate for such a language is the dialect of predicate calculus that we used to define serializer semantics. If we used this definition language as the proof language of the verification program, then we would be faced with the following tasks: translating specifications into their meanings, reasoning in the definition language about propositions expressed in the definition language, and translating the results into some humanly readable form. The translation from specification language into definition language is relatively easy: we have already described it in the previous chapter. The translation from definition language into specification language is more difficult.

We considered it to be preferable to carry out our reasoning, as far as practical, in the specification language. It is the language that the user is most likely to understand. Further, we find that most of the inference rules are easier to state and manipulate in the specification language than in the definition language.

The verification program can be simply viewed as a data base about the serializer code, a set of algorithms that are used to examine and modify the data base, and a set of specification clauses to prove about the serializer. The data base can be expressed as a set of node graphs representing the serializer operations, and a set of assertions about the serializer, expressed as specification clauses. The algorithms are largely rule-driven, where a rule is used to infer a specification clause from known clauses. The rules we present in this chapter are treated as axioms by the verification program; this chapter states and proves the rules.

5.2 Extensions to the specification language

As it stands, the specification language presented in the previous chapter is oriented towards describing external properties of serializers. It has no constructs for describing the internal structure of a serializer. The rules we define in this chapter require a means for describing the node graphs for the operations, and relating events to the node graphs. Therefore, we propose extensions to the specification language.

5.2.1 New symbols and clauses

The extensions to the specification language pose no special problems. They extend the domain of discourse for the language to include symbols that can represent any event (or node), and to include components of events and nodes. For the sake of simplicity, we will not formally define these extensions, although we could do so.

- * general event symbols - E, E1, E2,. . . are event symbols that can be associated with any serializer event through the symbol map.

- * general node symbols - N, N1, N2,. . . are node symbols that can be associated with any serializer node in the node graphs.

- * extended expressions - E.trans, E.node, E.kind are added as expressions that represent the components of events. N.kind, N.next, N.expr, and N.mob expressions are also added. An extension to the domain of expression values to include events, transaction identifiers, nodes, syntactic expressions, and node kinds is necessary. We also include literals for node kinds.

- * GX (Guarantee Exclusion) specification extensions - GX(Node, Node, Node) is added as a syntactic form. The function Node_excludes_node is used as its meaning. GX(N1, N2, N3) expresses the restriction that no transaction can execute N3 while some other transaction is executing between N1 and N2 (inclusive).

- * PX (Possession Exclusion) specification clauses - We use PX(Node, Node) clauses to represent possession exclusion. PX(N1, N2) expresses the restriction that no transaction can execute any node while some other transaction is executing between N1 and N2 (inclusive). We will define the meaning of PX clauses below.

5.2.2 Marked and unmarked events

In defining the verification rules in this chapter we have occasionally found it necessary to write ordering clauses where one or more of the events appearing in those clause are *not* required to occur. To achieve this, we introduce the notation

!E

to indicate a *marked* event symbol in the specification clause. We then modify the definition of a valid symbol map to require that all *unmarked* event symbols appearing in ordering clauses and GX clauses must map to events that occur in the complete history for which the map is defined. In all other respects, a marked event symbol is the same as an unmarked event symbol.

The alternative to introducing the !E notation is to *not* require a valid symbol map for some history to take event symbols appearing in ordering and GX clauses into events that must occur in the history. We would then explicitly require the use of @E to require event occurrence in clauses where such occurrence was important. We have previously rejected such an approach because it leads to surprising implications for some specifications. We believe that it is still the right choice; we prefer to have some additional complication in the language for defining the verification rules so we can retain some simplicity in the specification language at the user level.

We note here that the Precedes predicate used to give the meaning of ordering clauses is well-defined even when the events do not occur in the histories. Note that the clause

$$!E1 < !E2$$

can only be true for some history if both events denoted occur in that history. This can be stated as the clause:

$$!E1 < !E2 \supset @E1 \ \& \ @E2$$

Also note that if an ordering clause mentioning two events that need not occur is false, it could be due to either the opposite order holding, the two events being the same, or non-occurrence of either event, as is expressed by:

$$\sim(!E1 < !E2) \supset (!E2 \leq !E1) \mid \sim@E1 \mid \sim@E2$$

5.3 Some simple inference rules

In this section we present proofs for several inference rules stated in the specification language. These rules are presented as specification clauses where one sub-clause implies another. Note that the rules are actually rule generators: free variables are permitted to appear to denote nodes and events. The free node symbols are chosen from the set {N, N1, N2, ...}, and the free event symbols are chosen from the set {E, E1, E2, ...}.

5.3.1 Transaction order

Events belonging to the same transaction must occur in the order prescribed by the node graph for that transaction. We can write this restriction as an inference rule:

Transaction order rule:

E1.node.next = E2.node & E1.trans = E2.trans
$\supset$ E1 $<$ E2

Proof: For every valid symbol map p and complete history h, since E1 and E2 are mentioned in an ordering clause, p maps E1 and E2 to events that occur in h. Therefore, there must be events e1 and e2 (with indices I and J), such that the above rule is equivalent to:

(e1 = Nth(h, I) = $\mathcal{E}$({E1},p)
& e2 = Nth(h, J) = $\mathcal{E}$({E2},p)
& Same_trans(I, J, h)
& e1.node.next = e2.node)
$\supset$ I $<$ J

Since an **enter** node can not be the next component of any node, e2.kind $\neq$ **enter**. Therefore, by the definition of Legal_transaction_step, there must be some index K $\in$ Index_set(h) such that

(K $<$ J
& Nth(h, K).node.next = e2.node
& Nth(h, K).trans = e2.trans)

Further, K = I by Legal_transaction_step, which proves that I $<$ J.

5.3.2 Transitivity

The event ordering is transitive. This can be expressed by the following rule:

Transitivity rule:

$$(E1 < E2 \;\&\; E2 < E3) \supset E1 < E3$$

Proof: By the definitions given in chapter 3, the above specification clause is defined to be equivalent to:

$$\begin{aligned}
&(\,\mathrm{Precedes}(\mathit{E}(\{E1\},p), \mathit{E}(\{E2\},p), h)\\
&\;\;\&\; \mathrm{Precedes}(\mathit{E}(\{E2\},p), \mathit{E}(\{E3\},p), h)\,)\\
&\supset \mathrm{Precedes}(\mathit{E}(\{E1\},p), \mathit{E}(\{E3\},p), h)
\end{aligned}$$

where p is any valid symbol map for the complete history h. By the definition of a valid symbol map, there must be three distinct events (e1, e2, e3) that occur in h, which implies that there are three distinct indices (I, J, K) such that the above rule is equivalent to:

$$\begin{aligned}
&(\,e1 = \mathrm{Nth}(h, I) = \mathit{E}(\{E1\},p)\\
&\;\;\&\; e2 = \mathrm{Nth}(h, J) = \mathit{E}(\{E2\},p)\\
&\;\;\&\; e3 = \mathrm{Nth}(h, K) = \mathit{E}(\{E3\},p)\\
&\;\;\&\; \mathrm{Precedes}(e1, e2, h) \;\&\; \mathrm{Precedes}(e2, e3, h)\,)\\
&\supset \mathrm{Precedes}(e1, e3, h)
\end{aligned}$$

By the definition of Precedes and the existence of the indices I and J, Precedes(e1, e2, h) is equivalent to $I < J$. The other Precedes expressions have similar simplifications. Therefore, the specification clause is equivalent to

$$(I < J \;\&\; J < K) \supset (I < K)$$

which is true by the axioms of integer ordering. Therefore, the specification clause is a true statement.

5.3.3 PX clauses

A PX clause is used to specify possession exclusion. The meaning of a PX clause is given by:

$$C(\{\widetilde{PX}(\widetilde{n1,n2})\},p,h) = \text{PX\_def}(\boldsymbol{N}(\{\widetilde{n1}\},p), \boldsymbol{N}(\{\widetilde{n2}\},p), h)$$

where

```
            PX_def(N1, N2, H) ≡
            ∀ I,J,K ∈ Index_set(H):
  if (Nth(H, I).node = N1 & Nth(H, J).node = N2
     & Same_trans(H, I, J))
     then Excludes(Nth(H, I), Nth(H, J), Nth(H, K))
```

The clause PX(N1, N2) specifies that a transaction executing nodes N1 and N2 has possession (of the serializer containing N1 and N2) after executing N1 and up to the completion of executing N2, and that N1.next = N2. Note that while a transaction has possession no events from another transaction may occur. There are two rules used to imply PX clauses:

PX from gain rule:

```
( N1.next = N2
 & ( N1.kind = enter
   | N1.kind = dequeue
   | N1.kind = leave) )
⊃ PX(N1, N2)
```

PX from PX rule:

(PX(N1, N2)

& N2.next = N3

& N2.kind ≠ **join**

& N2.kind ≠ **enqueue**)

⊃ PX(N2, N3)

Proof: By contradiction. For the first rule, suppose that the precondition implies ~PX(N2, N3). By the definition of a valid symbol map, there must be three distinct events (el, e2, e3) that occur in any complete history h, which implies that there are three distinct indices (I, J, K) such that:

el = Nth(h, I) & e2 = Nth(h, J) & e3 = Nth(h, K)

& el.node = ***N***({N1},p) & e2.node = ***N***({N2},p)

& el.trans = e2.trans & el.node.next = e2.node

& (el.kind = **enter** | el.kind = **dequeue** | el.kind = **leave**)

& ~Excludes(el, e2, e3, h)

At the finite history Head(h, I), which is the smallest prefix of h that contains el, we know that Legal_step(Head(h, I), e2, S) is true (where S is the set of node graphs for the serializer operations). Further, because Busy(Head(h, I)) is true (by the definition of Busy and Gains), e2 is the only event that is a legal step. Therefore, no events can occur between el and e2, which contradicts ~Excludes(el, e2, e3, h). Therefore, the PX from gain rule is true. A similar proof holds for the PX from PX rule.

The PX clauses are useful as intermediate steps that imply event ordering. The following rule is used to imply an event ordering from a PX rule and other preconditions.

Event before PX rule:

(PX(N1, N2) & E < E2 & E1.trans = E2.trans
& E1.node = N1 & E2.node = N2)
⊃ E < E1

Proof: The above clause is equivalent to the following (for every valid symbol map p and complete history h):

(PX_def(*N*({N1},p), *N*({N2},p), h)
& Precedes(*E*({E},p), *E*({E2},p), h)
& *E*({E1},p).trans = *E*({E2},p).trans
& *E*({E1},p).node = *N*({N1},p)
& *E*({E2},p).node = *N*({N2},p))
⊃ Precedes(*E*({E},p), *E*({E1},p), h)

Because E, E1, and E2 are mentioned in ordering clauses, there must be three distinct events (e1, e2, e) that occur in h, which implies that there are three distinct indices (I, J, K) such that, by the definition of PX_def:

(e1 = Nth(h, I) = *E*({E1},p)
& e2 = Nth(h, J) = *E*({E2},p)
& e = Nth(h, K) = *E*({E},p)
& Precedes(e, e2, h)
& Excludes(e1, e2, e, h))

which implies Precedes(e, e1, h), which implies that the rule is true.

The other PX rule is quite similar, and can be stated as:

Event after PX rule:

$(\, PX(N1, N2)\ \&\ E1 < E\ \&\ E1.trans = E2.trans$
$\ \&\ E1.node = N1\ \&\ E2.node = N2\,)$
$\supset E2 < E$

Proof: Similar to proof for Event before PX.

5.3.4 GRE clauses

The GRE (Guarantee Requires Empty) clause is an intermediate step used to infer GX (Guaranteed Exclusion) clauses. The definition of the GRE clause is:

$$C(\{GRE(N1, N2)\},p,h) = GRE\_def(\boldsymbol{N}(\{N1\},p), \boldsymbol{N}(\{N2\},p), h)$$

where

```
GRE_def(n1, n2, h) ≡
  ∀ I,J,K ∈ Index_set(h):
          if( Nth(h, I).node = n2
            & Nth(h, J).node = n2.match
            & I ≤ K < J
            & Same_trans(h, I, J) )
           then ~Eval( Head(h, K), n1.expr )
```

The intuitive meaning of GRE(N1, N2) is that the queue or crowd denoted by N2.mob must be empty in order for the expression N1.expr to be true.

There are two rules that can be used to infer GRE clauses:

GRE from empty rule:

N1.expr = Empty_expr(N2.mob)
⊃ GRE(N1, N2)

GRE from expression rule:

(N1.expr = And_expr(Empty_expr(N2.mob), G)
| N1.expr = And_expr(G, Empty_expr(N2.mob)))
⊃ GRE(N1, N2)

Note that we have had to add some *ad hoc* extensions to the specification language. G denotes a boolean-valued expression, Empty_expr(N.mob) denotes either queue$empty(N.mob) or crowd$empty(N.mob), as appropriate, and And_expr(G1, G2) denotes the expression that is the conjunction of the two guarantees.

Proof: By definition of GRE_def and the Eval function. For the first rule, suppose that the guarantee is crowd$empty(C). Then for any history that contains a **join** event for that crowd but does not contain the corresponding **leave** event the guarantee will evaluate to false, which proves the rule. Similar reasoning holds for the first rule if the guarantee is queue$empty(Q). A similar proof holds for the GRE from expression rule.

5.3.5 Using GX clauses

GX clauses are used to indicate where events are excluded because of guarantees being false. For example, if a guarantee for a queue is crowd$empty(C), where C is a crowd, then a **dequeue** event with that guarantee is prohibited from occurring between a **join** and a **leave** event for any transaction for that crowd. The following rule is used to infer GX clauses.

GX from GRE rule:

(N1.match = N2 & N2 ≠ N
& (N1.kind = **join** | N1.kind = **enqueue**)
& N.kind = **dequeue**
& GRE(N.expr, N2.mob))
⊃ GX(N1, N2, N)

The clause GRE(N1, N2) used above is true ii the expression N1.expr requires the queue or crowd N2.mob to be empty for the expression to be true.

Proof: By contradiction. Suppose that GX(N1, N2, N) is not true, yet the preconditions are met. By the definition of a valid symbol map, there must be three distinct events (e1, e2, e) that occur in any complete history h, which implies that there are three distinct indices (I, J, K) such that:

(e1 = Nth(h, I) = *E*({N1},p)
& e2 = Nth(h, J) = *E*({N2},p)
& e = Nth(h, K) = *E*({N},p)
& e1.node.match = e2.node
& (e1.kind = **join** | e1.kind = **enqueue**)
& e.kind = **dequeue**
& Precedes(e1, e, h) & Precedes(e, e2, h))

Further, from the GRE clause we know that the guarantee for event e must be false for any prefix of h that contains e1 but does not contain e2. Since e occurs after e1, we have a contradiction (due to Legal_dequeue), since e is a **dequeue** event that occurs when its guarantee is false. Therefore, the GX from GRE rule is true.

GX clauses are a useful intermediate step that can be used to infer event orderings.

Event before GX rule:

(GX(N1, N2, N) & E < E2 & E1.trans = E2.trans
& E.node = N & E1.node = N1 & E2.node = N2)
⊃ E < E1

Proof: Because E, E1, and E2 are mentioned in ordering clauses, for any valid symbol map p and complete history h, there must be events (e1, e2, e) occurring at distinct indices (I, J, K) such that:

(e1 = Nth(h, I) = *E*({E1},p)
& e2 = Nth(h, J) = *E*({E2},p)
& e = Nth(h, K) = *E*({E},p)
& e.node = ***N***({N},p)
& e1.node = ***N***({N1},p)
& e2.node = ***N***({N2},p)
& Precedes(e, e2, h)
& Same_trans(h, I, J)
& Node_excludes_node(e1.node, e2.node, e.node, h))

By the definition of Node_excludes_node we can infer:

Excludes(e1, e2, e) & Precedes(e, e2, h) & e ≠ e1

which implies that Precedes(e, e1, h), which implies that the clause E < E1, and therefore the rule, is true.

As with the PX clause, there is a symmetric rule to Event before GX.

Event after GX rule:

(GX(N1, N2, N) & E1 < E & E1.trans = E2.trans
& E.node = N & E1.node = N1 & E2.node = N2)
⊃ E2 < E

Proof: Similar to proof for Event before GX.

5.3.6 FIFO queues

Serializer queues are served strictly first-in-first-out. The following rule is used to infer event orders from the use of FIFO queues in serializers.

Event from FIFO rule:

(E1 < E2 & E1.kind = **enqueue** & E2.kind = **enqueue**
& E1.node.mob = E2.node.mob
& E3.trans = E1.trans & E4.trans = E2.trans
& E3.node = E1.node.next & E4.node = E2.node.next)
⊃ !E3 < E4

Proof: By contradiction. First, suppose that E3 occurs (we are not required to do so by the clause). As in the above proofs, E1, E2 and E4 are unmarked events mentioned in ordering clauses; so they must occur. There must be four events (e1, e2, e3, e4) with distinct indices (I, J, K, L) such that:

(e1 = Nth(h, I) = *E*({E1},p)
& e2 = Nth(h, J) = *E*({E2},p)
& e3 = Nth(h, K) = *E*({E3},p)
& e4 = Nth(h, L) = *E*({E4},p)
& Precedes(e1, e2, h)
& e1.kind = **enqueue** & e2.kind = **enqueue**
& Same_trans(I, K, h) & Same_trans(J, L, h)
& e3.node = e1.node.next & e4.node = e2.node.next)

We need to prove that Precedes(e3, e4, h), which we do by assuming Precedes(e4, e3, h), and finding a contradiction. By the definition of Legal_transaction_step we know that Precedes(e1, e3, h) and Precedes(e2, e4, h). Let h1 be the largest prefix of h that does not contain e4. We will show the contradiction by considering the predicate Legal_step(h', e4, S), where S is the set of node graphs for the serializer.

Since e4.kind = **dequeue,** Legal_step(h1, e4, S) requires that Legal_dequeue(h1, e4) be true, which requires that Eval be true for the guarantee, and that Head_enqueue(h1, J) be true. Head_enqueue(h1, J) is only true if every other transaction with an **enqueue** event for the queue e4.node.mob that occurred in h1 prior to e4 has a corresponding **dequeue** event that has occurred in h1. However, we know that e3 has not occurred in h1 by our assumption of Precedes(e4, e3, h). Therefore, either Precedes(e3, e4, h), or e3 does not occur.

The proof that e3 occurs is simple. We know that e4 occurs in h, since it is denoted by an unmarked event mentioned in an ordering clause. Therefore, when e4 occurs, e3 must have occurred in the history h1 by the definition of Legal_dequeue.

5.4 Evaluation of guarantees

In further rules we will need to express the evaluation of guarantees. The clause EVT(G, E) is used to specify that expression G always evaluates to **true** immediately before event E. The clause EVF(G, E) is used to specify that expression G always evaluates to **false** immediately before event E. In translating from specification language to definition language we will assume that, if the event denoted by E occurs at index I in history h, then

$$\mathcal{C}(\{\text{EVT}(G, E)\}, p, h) = \text{Eval}(\text{Head}(h, I\text{-}1), \{G\})$$
$$\mathcal{C}(\{\text{EVF}(G, E)\}, p, h) = \sim\text{Eval}(\text{Head}(h, I\text{-}1), \{G\})$$

When the event denoted by E does not occur, the EVT and EVF clauses are undefined. We are careful to only use these clauses in contexts where such an event does occur.

The following rule can be used to infer EVF clauses:

EVF rule:

```
( (E1.kind = enqueue | E1.kind = join)
 & E1.node.next = E2.node
 & E1.trans = E2.trans
 & E1 < E < E2 )
⊃ EVF(Empty_expr(E1.mob), E)
```

Proof: Suppose that M is a queue. By the definition of Legal_transaction_step, there can never be more **dequeue** events than **enqueue** events for any transaction. Therefore, by the definition of Csize, the queue is empty (Csize(M) = 0) only if all transactions have the same number of **enqueue** events as **dequeue** events immediately preceding E. However, the transaction E1.trans has an **enqueue** event (E1) that has occurred without the matching **dequeue event** (E2). Therefore, the queue must not be empty. A similar proof holds if M is a crowd.

The following rule can be used to infer EVT clauses:

EVT rule:

```
(∀ E1,E2:
     if ( E1.trans = E2.trans & E1.node.mob = M
        & E1.node.match = E2.node )
       then E ≤ E1 | !E2 < E )
⊃ EVT(Empty_expr(M), E)
```

Proof: First, we note that within the quantification the events E and E1 are required to occur, yet the event E2 is not required to occur, since it is marked. The condition that we are expressing with the quantified clause is that for every pair of events denoted by E1 and E2 the event denoted by E either occurs before (or is the same as) E1, or occurs after E2. Note that if E1 < E is true, then !E2 < E is false if E2 does not occur. In order for Empty_expr(M) to be false when evaluated immediately before E there must be some transaction that is **in** M immediately before E, which means that the **enqueue** (or **join**) event (call it E1) occurs before E, but the **dequeue** (or **leave**) event (call it E2) does not occur before E. We can express this requirement as

$E1 < E \leq !E2$

which is prohibited by the precondition

$E \leq E1 \mid !E2 < E$

and therefore the clauses always evaluates to **true** immediately before E.

The above clause uses internal quantification over **all** events, which is another extension to the specification language. It is difficult to use the above rule as it is in a verification program due to the internal quantification. The set of all events is infinite, and cannot be enumerated. We can prove that the quantification clause is satisfied by contradiction: proving that there can not exist a transaction with events E1 and E2 (as given above) where the clause within the quantification is not satisfied. This method will be further discussed in the next chapter.

The following rules can be used for guarantees that are conjunctions or disjunctions. These rules are sufficiently simple that we will omit the proofs.

EVT from conjunction rule:

(G = And_expr(G1, G2)
& EVT(G1, E) & EVT(G2, E))
⊃ EVT(G, E)

EVT from disjunction rule:

$$
\begin{aligned}
&(G = \mathrm{Or\_expr}(G1, G2) \\
&\ \ \& \ (\mathrm{EVT}(G1, E) \mid \mathrm{EVT}(G2, E))) \\
&\supset \mathrm{EVT}(G, E)
\end{aligned}
$$

EVF from conjunction rule:

$$
\begin{aligned}
&(G = \mathrm{And\_expr}(G1, G2) \\
&\ \ \& \ (\mathrm{EVF}(G1, E) \mid \mathrm{EVF}(G2, E))) \\
&\supset \mathrm{EVF}(G, E)
\end{aligned}
$$

EVF from disjunction rule:

$$
\begin{aligned}
&(G = \mathrm{Or\_expr}(G1, G2) \\
&\ \ \& \ \mathrm{EVF}(G1, E) \ \& \ \mathrm{EVF}(G2, E)) \\
&\supset \mathrm{EVF}(G, E)
\end{aligned}
$$

We have used G, G1, and G2 to denote guarantees, and And_expr and Or_expr to denote conjunctions and disjunctions of guarantees.

5.5 Priority of dequeue over enter and leave

If there are queues with true guarantees when possession is released, a **dequeue** event for one of those queues will occur before an **enter** or **leave** event.

Suppose we know that an **enqueue** event E1 occurs before an external gaining event E. To show that E must occur after the **dequeue** event E2 corresponding to E1, we must know that the guarantee for E1 is true immediately prior to E, and that there can be no transaction with a false guarantee that is in the queue ahead of the transaction for E1 when E occurs.

Event from ready queue rule:

```
((E.kind = enter | E.kind = leave)
 & E1.node.next = E2 & E1.trans = E2.trans
 & E1.kind = enqueue
 & EVT(E1.expr, E) & E1 < E
 & ∀ E3,E4:
        if( E3.kind = enqueue & E3.mob = E1.mob
          & E3.trans = E4.trans
          & E3.node.next = E4.node
          & E3 < E1 )
        then EVT(E3.expr, E) | !E4 < E )
⊃ !E2 < E
```

Proof: We will outline a proof by contradiction. Assume that the gaining event E precedes the **dequeue** event E2, such that E1 < E < E2. The quantification over E3 and E4 is a precondition that requires every transaction that has entered the queue before E1.trans to either have a true guarantee (immediately before E) or to have left the queue before the gaining event E. Therefore, there can be no transaction with a false guarantee in the queue ahead of E1.trans. However, the gaining event E cannot occur while there is a queue with a true guarantee, which is true for E1.mob. This is a contradiction, so we can infer that if E2 occurs, it must occur before E. By similar reasoning, E2 must occur, since if it does not occur there will be a ready queue when E occurs (E must occur, since it is an unmarked event).

Note that the above rule was expressed as implying !E2 < E, which not only implies an ordering between events, but also implies that the event denoted by E2 occurs, since any event the precedes an event that occurs must also occur.

The above rule is admittedly long and complex. We can shed some more light on the reasoning behind its form by considering some examples.

- Suppose that there are events E3 and E4 such that E3 < E1, and E4 does not occur (using E, E1, E2, E3, and E4 as in the above rule). Then the precondition expressed by the quantification must be false, which means that we cannot infer E2 < E. This should seem reasonable, since by the FIFO queue rule we know that E4 must precede E2 if E2 occurs, which implies that E2 does not occur.

- Suppose that there are events E3 and E4 such that EVF(E3.expr, E) and E3 < E1. Then it is possible for E3.trans to be at the head of the queue when E is ready to occur, which would imply that E < E4, or that E4 did not occur at all.

The reader may note that we have only considered a single queue in the above rule. It may be imagined that all of the preconditions were met for two queues, yet one queue was arbitrarily chosen to proceed, which then made the head guarantee of the other queue false, which then allowed the gaining event E to occur. Such a situation *is* covered by our rule, since we do not specify evaluation of the guarantee at any particular time, but rather *immediately before the event E in any context.* Intervening **dequeue** events from other queues are unimportant, since they will only postpone the occurrence of E, not change the precondition EVF(E1.expr, E).

5.6 A method for proving service

A service specification typically states that for every complete history and valid symbol map, the occurrence of an **enter** event for some transaction implies the occurrence of the **exit** event for that transaction. In proving this, we typically need to prove that the occurrence of any event (**exit** events excluded) in a transaction implies the occurrence of the next event in the transaction. Another way to state that the occurrence of one event implies the occurrence of another is to say that every complete history that contains the first event contains the second.

For most events in a transaction, if an event occurs, the successor event in that transaction must occur. For simple serializers, the occurrence of an event that gains possession implies the occurrence of a corresponding event that releases possession. Further, we have assumed that accesses to the resource terminate, so the occurrence of a **join** event implies the occurrence of the corresponding **leave** event. There are only two kinds of events where the occurrence of an event does not imply the occurrence of the successor: **exit** events, because they have no successors; and **enqueue** events, because they might never have true guarantees whenever possession is released, or because there might always be another queue ready whenever possession is released.

The method we propose for proving that an **enqueue** event requires a **dequeue** event is to first suppose that the **dequeue** event does not occur, then prove a contradiction: that a complete finite history exists where there is a ready queue at the end of the history.

Suppose that we want to prove $@E1 \supset @E2$, where E1 and E2 belong to the same transaction, and E1 precedes E2 if both events occur (which can be written as $@E1 \,\&\, @E2 \supset !E1 < !E2$). We need to show for every **enqueue** event E3 with corresponding **dequeue** event E4 that if E3.trans = E1.trans then the occurrence of E3 implies the occurrence of E4 ($@E3 \supset @E4$).

If an **enqueue** event occurs for some queue and the **dequeue** event does not occur, then we say that its queue is blocked. If a queue is blocked, then we can infer the following:

- If every **join** event for some crowd requires a preceding **dequeue** event from a blocked queue, then the crowd will eventually become empty. This is true because when the queue is blocked, there can be no further **join** events, and every **join** event requires that a **leave** event occur.

- If every **enqueue** event for some queue Q requires that a **dequeue** event for a blocked queue B must occur (because the **enqueue** event must follow some other **dequeue** event that is waiting for B to empty), then Q will eventually become either blocked or empty. Since the **enqueue** event for Q will not occur, then no new transactions will be added to Q, which implies that only **dequeue** events for Q can possibly occur. Eventually either Q is empty or a transaction with a false guarantee is at the head of Q.

- If every occurrence of an **enqueue** event for some queue implies the occurrence of a corresponding **dequeue** event, and the queue will eventually become either blocked or empty, then the queue will eventually become empty.

By saying that a condition "eventually becomes" true, we mean that for every complete history there is a event where the condition is true at every event after that event.

The method is now clear: to prove the contradiction, we assume that the **dequeue** event (E) does not occur, that certain queues and crowds will become empty, and that certain queues will become either empty or blocked. If these additional assertions are sufficient to prove that the guarantee for E is true, and that there is no other **dequeue** event with a false guarantee that is blocking E, then we have found a contradiction, and actually proved that E must occur.

We will not present rules for proving service. The number of supporting rules is relatively high, and the additional material would not introduce any new concepts. The method of proving service will be further explained in the next chapter.

5.7 Rule-based proving of FIFO priority specification

In this section we present a proof based on successive applications of the rules we have presented in this chapter. As presented in the previous chapter, the FIFO readers-writers problem has the following (partial) priority specification:

$$\text{R1-enter} < \text{W1-enter} \supset \text{R1-exit} < \text{W1-exit}$$

A rule-based proof of the above clause takes two stages: derivation of intermediate clauses (such as PX, GRE, and GX clauses), and use of the rules that imply event orders. Note that the first stage need only be performed once for any particular serializer, while the second stage is usually different for every specification clause.

In the first stage, we examine the node graphs and use the PX from gain rule to derive the following PX clauses, which indicate possession exclusion:

PX(R*-enter, R*-enqueue(x.xq))
PX(R*-dequeue(x.xq), R*-join(x.rc))
PX(R*-leave(x.rc), R*-exit)
PX(W*-enter, W*-enqueue(x.xq))
PX(W*-dequeue(x.xq), W*-join(x.wc))
PX(W*-leave(x.wc), W*-exit)

We then examine the node graphs and use the GRE from empty rule and the GRE from expression rule to derive the following GRE clauses:

GRE(R*-dequeue, W*-join)
GRE(W*-dequeue, R*-join)
GRE(W*-dequeue, W*-join)

Using the GRE clauses and the GX from GRE rule, we derive the following GX clauses:

GX(W*-join, W*-leave, R*-dequeue)
GX(R*-join, R*-leave, W*-dequeue)
GX(W*-join, W*-leave, W*-dequeue)

In the second stage of the proof, we prove the implication by assuming the precondition, and deriving the consequence. We use the Transaction order rule to derive:

(R1-enter < R1-enqueue < R1-dequeue
< R1-join < R1-leave < R1-exit)
&
(W1-enter < W1-enqueue < W1-dequeue
< W1-join < W1-leave < W1-exit)

Then we perform the following inferences, using the indicated rules:

Event order	**Rule applied**
R1-enter < W1-enter	Assumed
R1-enqueue < W1-enter	Event after PX
R1-enqueue < W1-enqueue	Transitivity
R1-dequeue < W1-dequeue	Event from FIFO
R1-join < W1-dequeue	Event after PX
R1-leave < W1-dequeue	Event after GX
R1-exit < W1-dequeue	Event after PX
R1-exit < W1-exit	Transitivity

5.8 Comments on the verification rules

While the intent of defining inference rules in the specification language is to simplify verification, one unfortunate side-effect has been to add numerous clauses to the specification language. These additions have made the specification language far closer to our definition language than we would like. As we add more extensions we begin to lose the simplicity that proofs in the specification language have over proofs in the definition language. Despite these misgivings, the rules do appear to work at a higher level than could be obtained from the definition language.

We have added a means for avoiding the requirement that every event mentioned in the ordering clauses must map (via the symbol map) to an event that occurs in the complete history on which the map is based. There is no inherent reason why this ability should not be extended to the user, although we have chosen not to do so. This feature is only rarely used, and continues to have potentially surprising interpretations, as evidenced by the Event from ready queue rule, where the occurrence of an event was proved without resorting to the @E notation.

6. Automatic Serializer Prover

The previous chapter presented verification rules that were defined in an extended specification language. This chapter describes a program that makes use of those rules. While limited to dealing with simple serializers and specification clauses that do not mention the rank of an event, many of the principles used are applicable to more general serializers. The program, called ASP (Automatic Serializer Prover), has been tested on a number of versions of the readers-writers problem.

In this chapter, we discuss the structure of ASP, first by giving an overview, then by detailing some of the algorithms used. The results for the readers-writers examples are given, and we discuss how ASP could be extended to accommodate various extensions to simple serializers.

6.1 Overview of ASP

The input to ASP is a description of each operation of a serializer and the specification clauses for the serializer. We use ASP interactively to prove that the specification clauses are satisfied, or to examine why they are not. The execution of ASP has the following phases:

* Initialization: This phase builds representations of the node graphs for the serializer operations given the text for the operations.[16] In the

[16] In the actual program, the text must undergo an initial translation by hand in order to be processed. This allowed us to concentrate our efforts on verification rather than parsing.

remainder of this chapter, we will make no distinction between the node graph representations used by the program and the node graphs used in the semantic model.

- Static analysis: This phase examines the node graphs to determine possession exclusion, represented by the PX clauses mentioned in the previous chapter, and guarantee exclusion, represented by GX clauses. Note that we also make no distinction between the specification clause representations used by the program and the actual clauses.

- Verification: In this phase we attempt to prove each specification clause given. Typically, a specification clause is given as an implication consisting of a precondition clause and a consequent clause. Proving such a clause involves assuming the precondition and using the inference rules described in the previous chapter to derive the consequent clause. When a consequent clause is derived, further rules may be applied to derive new clauses.

The node graphs, specification clauses, and other data are kept in a structure called the *data base*, which is composed of the following parts:

- Node graphs: There is a node graph for each operation of the serializer. Each node has a structure as described in Chapter 3. Data structures representing expressions (as in N.expr), queues and crowds (as in N.mob), and kinds (as in N.kind) are referred to by the node graphs.

- Transaction stack: There is a stack of transactions that represent the transactions mentioned in the specification clauses. Each transaction symbol in a specification clause has a corresponding transaction in this stack. Further transactions may be added to this stack due to attempted proof by contradiction, as mentioned in the previous chapter. When such an attempt succeeds or fails, such a transaction is removed from the stack.

- Assertion stack: There is a stack of specification clauses that have been asserted and the rules used to assert the clauses. The asserted clauses are those that have been assumed to be true or have been added by application of the inference rules to the clauses in the assertion stack. This stack provides a record of which rules led to particular event orderings, as well as an efficient mechanism for removing assertions.

- Event stack: There is a stack of the events that exist (although do not necessarily occur) for the transactions in the transaction stack. This stack is closely coupled to the stack of known transactions, since each event in this stack must have a known transaction. Whenever a transaction is added to the transaction stack, an event for every node that the transaction *may* execute is added to the event stack. When a transaction is removed from the transaction stack, all events for that transaction are removed from the event stack.

- Event order matrix: There is an extensible square matrix used to represent event orders. There is a row and a column for each event, with the entries indicating the ordering between the events. The row and column index for a particular event are identical, and the index for an event in this matrix corresponds to the index in the event stack for the event. The matrix is extended or retracted (in both dimensions) as the event stack is extended or retracted.

6.2 Static analysis phase

The static analysis phase inserts PX and GX clauses into the data base according to the node structure of the operations. It is performed in advance of examining the specification clauses. The purpose of the static analysis phase is to perform steps that can be done once for a given serializer, and avoid performing these steps for every clause we wish to prove.

The PX (Possession Exclusion) clauses are generated by examining the node graph to determine when a transaction is in possession of the serializer. For simple serializers only the PX from gain rule is needed.

The GX (Guarantee Exclusion) clauses are generated by examining the guarantees on **enqueue** statements during the initial pass over the serializer. They are generated according to the GX from GRE rule, which depends on the GRE from empty rule and the GRE from expression rule. As long as the guarantees only involves testing the emptiness of crowds or queues, or conjunctions (G1 & G2) of tests for emptiness, GX clauses can be generated for the guarantees during static analysis. Guarantees that are disjunctions (G1 | G2) or negations (~G) do not generate GX clause during static analysis.

6.3 Verification phase

A specification clause is usually written as $P \supset Q$, where P and Q are specification clauses that do not use implication clauses. Verifying that $P \supset Q$ is satisfied involves assuming that the precondition clause P is true, and showing that the consequent clause Q is therefore true. Note that the clause P is assumed to be true for a *particular* choice of complete history and valid symbol map. The verification methodology allows us to prove:

$\forall$ p,h: $(P \supset Q)$

The assumption and proof should *not* be viewed as:

$(\forall\ p,h: P) \supset (\forall\ p,h: Q)$

When a clause not previously in the assertion stack is asserted, we say that it is *inserted* into the data base. When a clause is inserted, ASP checks certain rules to determine whether they are immediately applicable. These rules are called *insertion rules*, and are: Transitivity, Event before PX, Event after PX, Event before GX, Event after GX, and Event from FIFO. If any are applicable, we assert the event order clauses they imply. This, in turn, may lead to the assertion of further clauses, and so on. This process is complete when no further insertion rules are applicable.

In asserting an event ordering, we need to have computer representations of events. In order to have event representations, we need transaction and node representations. The initialization phase built the nodes. The transactions and events are built by examining the specification clause to determine which transactions are mentioned in the clause. These transactions, and their associated events, are added to the data base.

For each transaction that is added due to being explicitly named in the specification clause, the Transaction order rule is used to determine the order of the events that belong to the transaction. This leads to the insertion of event order clauses, but does not immediately lead to the application of any rules other than the transaction order rule and the transitivity rule, since there is no known initial ordering between events from different transactions.

To prove an implication, we assert the precondition and attempt to derive the result. The precondition for a specification clause is asserted by performing operations on the data base to assume the various parts of the clause. For example, one component of the specification clause may be an event ordering, E1 < E2. This clause is asserted by calling the *add_order* operation of the data base. If this clause was not previously asserted, the insertion rules are applied by this operation.

6.4 Evaluation of guarantees and anonymous transactions

In several places in ASP it is necessary to evaluate a guarantee to determine if a queue is ready. The EVT and EVF clauses mentioned in the previous chapter are used to indicate the evaluation of guarantees. EVT(G, E) is true for some history that contains E if the guarantee G evaluates to true in the largest prefix of the history not containing E. EVF(G, E) is true if G evaluates to false in that prefix. For example, if the event E occurs between corresponding **enqueue** and **dequeue** events for some transaction, as in:

X-enqueue(Q) < E < X-dequeue(Q)

then we can assert the clause

EVT(queue$empty(Q), E)

In some cases, it is not sufficient to simply use the EVT and EVF rules presented in the previous chapter. Consider the following concurrency specification for the FIFO readers-writers serializer:

R1-enter < R2-enter < R1-leave & GX(R1-enter, R2-enter, W*-enter)
⊃ R2-join < R1-leave

In proving this specification, we need to prove

EVT({crowd$empty(x.wc)}, R1-leave)

The insertion rules are sufficient to prove that the writers crowd (x.wc) is empty when the readers crowd (x.rc) is not empty. However, the rules we have presented do not immediately allow us to conclude that the EVT clause above is true, since we must prove the clause for *all* writers.

A more general method of proof is available to us, based on proof by contradiction. If we assume that a writer is in the writers crowd, and that leads to a contradiction, then the writers crowd must be empty. To be exhaustive in choosing the writer, we have two cases:

1: The writer can be a writer that already exists in the transaction stack. To assume that some writer W is in the writers crowd when R1-leave occurs, we assert:

W-join < R1-leave < W-leave

and apply the insertion rules as necessary. A contradiction occurs if this leads to E < E being asserted for any event E (cyclic event orders are prohibited by Legal_transaction_step). If no contradiction occurs, then we cannot prove the EVT clause. If all writer transactions in the transaction stack cannot be in the writers crowd, it is necessary to apply the second case.

2: If no writer in the transaction stack can be assumed to be in the writers crowd, it is still possible that there is some other writer that can be in the crowd. Therefore, we invent an *anonymous transaction* and place it in the transaction stack, and assume that the new writer is in the crowd, as in the first case. If assuming that the anonymous transaction in in the crowd leads to a contradiction, then we can assume that the writers crowd is empty at R1-leave, and therefore the EVT clause is true.

The above method is easily generalized to proving any queue or crowd empty.

6.5 Checking for ready queues

The Event from ready queue rule is difficult to apply, since there is nested quantification. We start by examining the data base for **dequeue** events where the guarantees are true immediately preceding **enter** or **leave** events. Consider some transaction X, where X-dequeue has a true guarantee immediately before some **enter** or **leave** event, which we will call E. If E is known to occur after X-enqueue, then the only way that E can occur before X-dequeue is for there to be a transaction in the same queue, ahead of X, with a false guarantee. If such a transaction exists, we say that it *blocks* X-dequeue.

If no known transaction can block X-dequeue, it may still be possible that some other transaction not mentioned in the specification clause can block X-dequeue. Therefore, we create an anonymous transaction Z for an operation (provided that that transaction can have an **enqueue** event for the same queue as X-dequeue), and assert that

Z-enqueue < X-enqueue < Z-dequeue

where X-enqueue and Z-enqueue occur for the same queue. If the guarantee for Z-dequeue is true immediately before E, then Z cannot block X. Further, if asserting that Z-dequeue occurs after E causes a conflict, then there can be no such transaction Z. If there is no Z, for any operation of the serializer, that can block X, then X-dequeue must occur before E.

6.6 Proving by cases

One potential drawback of using the insertion rules is that some relatively simple proofs will be unachievable because there are not enough assertions. In particular, if **enter** events E1 and E2 are known to occur, yet the order of E1 and E2 is unknown, we may be able to prove a clause if we assume either E1 < E2 or E2 < E1, yet be unable to prove the clause if no order is assumed. ASP can perform some of these proofs by cases: where the order of E1 and E2 is unknown, first assume E1 < E2 and perform the proof, then retract the assumption of E1 < E2, assume E2 < E1, and perform the proof. If the desired result is obtained in both cases, the proof is valid, *provided* that E1 and E2 are known to occur.

The concurrency specification clause given for the FIFO serializer was overly restrictive, since it specified that

R1-enter < R2-enter < R1-leave

and the result (R2-join < R1-leave) can be shown to be true even if R2-enter < R1-enter. The following clause is a stronger version of the concurrency specification that requires proof by cases:

GX(R1-enter, R2-enter, W*-enter) & R2-enter < R1-leave
⊃ R2-join < R1-leave

Note that the GX clause does not specify that R1-enter < R2-enter, although the GX clause is trivially satisfied if R2-enter < R1-enter. Initially the precondition is asserted. Then ASP first assumes R1-enter < R2-enter, proves the consequent clause, retracts the assumption, then assumes R2-enter < R1-enter, and proves the consequent clause. That R1-enter and R2-enter occur can be shown in two ways: they are mentioned in a GX clause, and events subsequent to them (by Legal_transaction_step) are mentioned in an ordering clause.

6.7 Proving guaranteed service

In many serializers we would like to prove that every transaction receives service, i.e., for every **enter** event there is an **exit** event. The following is a typical service specification clause:

@T-enter ⊃ @T-exit

Proving guaranteed service for a transaction is performed by proving that each **dequeue** event that the transaction can execute is guaranteed to occur, since we have assumed for simple serializers that all other kinds of events will occur in complete histories given their predecessors.

Proving that a **dequeue** event occurs is largely done by contradiction: We assume that the **dequeue** event does not occur, which implies that its queue is not empty, and that any crowds that require **dequeue** events from that queue will empty.

This is generally enough to show that the guarantee for the **dequeue** event is true. The **dequeue** event must occur if no other queue is ready.

In this method, evaluating the guarantees must take place immediately prior to some event, since that is the basis of our evaluation mechanism. But there may be no actual event occurring, especially if no further **enter** events occur. Therefore, we invent a fictitious event with certain properties. We assume that some "quiet point" event QP occurs, such that the event QP gains possession of the serializer only when no queues are ready, and QP occurs late enough such that every crowd or queue that must empty has emptied. If the guarantee for the **dequeue** event in question is true at QP, and there can be no blocking of the **dequeue** event, then the **dequeue** event must precede QP, provided that QP does occur. We can guarantee that QP does occur if every other queue is not ready at QP. At this point we have proved that QP does occur, and the **dequeue** event precedes QP, but we assumed that the **dequeue** event does not occur. This is the contradiction that proves that the **dequeue** event does occur.

For extended serializers, it is possible for a request kind to have guaranteed service, yet the quiet-point method is too weak. To illustrate, suppose a serializer has the following operation:

```
op = proc (x; cvt)
    if queue$empty(x.q)
       then % O-enq1
            enqueue x.q until crowd$empty(x.c)
       else % O-enq2
            enqueue x.q until crowd$empty(x.c) & ~crowd$empty(x.cc)
       end
    join x.c % O-join1
         end
    join x.cc % O-join2
         end
    end op
```

For simplicity, we will suppose that *op* is the only operation of the serializer that can get sole possession (uses cvt). The QP event will not occur until x.c is empty and x.cc is empty. However, at QP the guarantee for O-enq2 is false. Therefore, it seems possible for QP to occur before O-enq2, so guaranteed service cannot be proven.

One way to prove guaranteed service for the above serializer is to split the proof into two cases dependent on the test *queue$empty(x.q)* in the if statement. If the test was true, the QP method will work. If the test is false just before O-enq2 occurs, then there must be at least one other transaction, call it O1, that is in *x.q* when the O-enq2 occurs. But then there are two more cases, based on whether or not *crowd$empty(x.c)*. If *x.c* is empty, then the guarantees for *x.q* must be true, and O-deq2 must occur before O1-leave, which must occur before QP, which guarantees service. If *x.cc* is not empty, then there is yet another transaction, call it O2, such that *x.c* will be empty at O2-join2, which implies that the guarantees for *x.q* will be true before O2-leave2, which must precede QP. Although this analysis by case would be expensive, it would be possible to add to ASP.

The reader might object that the above example is quite contrived, and we would agree. We have discovered no convincing realistic examples that require more than the simple QP method, even when extensions to serializers are considered. For this reason, ASP supports only the simple QP method.

6.8 A sample verification

This section presents a sample verification performed by ASP. Figure 4 figure shows the results produced by using ASP to verify a priority clause for the FIFO readers-writers serializer presented in Chapter 2. Input from the user is indicated by underlining. The user starts the session by typing in the name of the serializer that should be used. That name is interpreted as a file name, where the file should contain a description of the serializer in the format required by ASP. Then the user types the clause to be verified.

The response from ASP indicates whether the clause could be proved, and shows the assertion stack after the insertion rules have been applied (the first clause printed is the most recently asserted clause). This information is usually sufficient to enumerate the steps of the proof, or to demonstrate why the clause could not be proved. While we will not describe them in this thesis, additional aids are present for more detailed inspection of the steps that ASP uses to prove clauses.

Figure 4. A sample verification by ASP

```
Name of serializer: FIFO
1.012 seconds to setup.

Specification clause: R1-enter < W1-enter => R1-exit < W1-exit
Proved Implies(R1-enter < W1-enter,
               R1-exit < W1-exit)

base[39:
     R1-exit < W1-dequeue-xq: Possession exclusion,
     R1-leave-rc < W1-dequeue-xq: Guarantee exclusion,
     R1-join-rc < W1-dequeue-xq: Possession exclusion,
     R1-dequeue-xq < W1-dequeue-xq: FIFO queues,
     R1-enqueue-xq < W1-enter: Possession exclusion,
     R1-enter < W1-enter: Assumed,
     TR: W1-enter: From clause,
     TR: R1-enter: From clause]
1.376 seconds.
```

Note in Figure 4 that not all of the rules are shown. The default used is to omit showing the clauses asserted in the static analysis phase, and use of the Transaction order and Transitivity rules. The notation "base[39:" appearing in the middle of the figure indicates that the assertion stack has 39 members. At the end of the figure the amount of processor time needed for the proof is given. This figure includes the processor time necessary to parse the expression, apply the verification rules, and to print the results. The notation "TR: W1-enter: From clause" is used to indicate that the transaction W1 was added to the transaction stack since the transaction was mentioned in the specification clause (for uniformity in the program this is treated as an assertion).

6.9 Performance results

In this section we present a number of verifications performed by ASP on variation of the readers-writers problem. Each test is given as a specification clause to be verified (or not verified) for different readers-writers serializers. Figure 5 presents these specifications, most of which have been mentioned in previous chapters as specifications of different properties for the readers-writers problem.

Figure 5. Readers-writers tests for ASP

```
Wpri: Writer's priority
    R1-join < W1-enter < R2-enter < W2-enter < R1-leave
    ⊃ W2-join < R2-join

(NWPRI): Modified Writer's priority
    W1-enter < R1-enter < W2-enter < W1-leave
    ⊃ w2join < r1join

Rpri: Reader's priority
    W1-enter < W2-enter < R1-enter < W1-join
    ⊃ R1-join < W2-join

(NRPRI): Modified Reader's priority
    R1-enter < W1-enter < R2-enter < R1-leave
    ⊃ r2join < w1join

|| R: Concurrency for Readers
    GX(R1-enter, R2-enter, W*-enter) & R2-enter < R1-leave
    ⊃ R2-join < R1-leave

XexY: X busy excludes Y busy
    X-join < Y-join ⊃ X-leave < Y-leave

XpoY: X not by-passed by Y
    X-enter < Y-enter ⊃ Xexit < Yexit

GS(X): Guaranteed service for X
    @X-enter ⊃ @Xexit
```

An abbreviation for each specification is given prior to each clause. The Wpri and Rpri clauses specify writer's and reader's priority properties. The (NWPRI) and (NRPRI) clauses specify alternate versions of these properties to be proved for the NWPRI and NRPRI serializers (to be shown below). The XexY clause actually denotes three clauses: RexW, WexR, and W1exW2, where appropriate substitutions apply. The XpoY clause also denotes three clauses, with the same substitutions.

Figure 6 presents the code, in abbreviated form, for each of the seven readers-writers serializers tested. The create operations and headers have been omitted, as is the trailing code after any **join**. The use of crowd$empty and queue$empty is implicit where empty is used. There is one FIFO serializer, two readers priority serializers (RPRI & NRPRI), three writers priority serializers (WPRI1, WPRI2 & NWPRI), and one serializer that allows starvation (STARVE). Note that the priority specifications for RPRI and NRPRI differ, and that there are also two distinct writers priority specifications.

The various serializers above were developed at different times. In particular, NRPRI and NWPRI were written after ASP had become relatively reliable. We originally attempted to prove the Rpri specification clause for the NRPRI serializer. The attempt was made much more difficult by a preconception (due to a faulty informal proof) that the clause could be proved. After much effort to determine the cause of the fault in the program, we finally noticed that the program was correct: not only was the clause not satisfied, but the intermediate steps followed by ASP provided a counterexample. It was this example more than any other that convinced us of the worth of automatic verification aids.

The modified writers priority specification came about as a test of the speculation that NWPRI satisfied a priority clause that was symmetric to NWPRI, since the serializers were (roughly) symmetric. The unmodified writers priority clause is also satisfied by the NWPRI serializer.

Figure 6. Code for test serializers

```
Name    Oper  Code

FIFO    R     enqueue xq until empty(wc); join rc
        W     enqueue xq until empty(wc)&empty(rc); join wc

RPRI    R     enqueue rq until empty(wc); join rc
        W     enqueue wq until empty(rq)
              enqueue rq until empty(wc)&empty(rc); join wc

WPRI1   R     enqueue rq until empty(wq)|empty(rc)
              enqueue wq until empty(wc); join rc
        W     enqueue wq until empty(rc)&empty(wc); join wc

WPRI2   R     enqueue rq until empty(rc)
              enqueue wq until empty(wc)&empty(rq); join rc
        W     enqueue wq until empty(rc)&empty(wc); join wc

STARVE  R     enqueue rq until empty(wc); join rc
        W     enqueue wq until empty(wc)&empty(rc); join wc

NRPRI   R     enqueue xq until empty(wc); join rc
        W     enqueue xq until empty(wc)&empty(rc)
              enqueue xq until empty(wc)&empty(rc); join wc

NWPRI   R     enqueue xq until empty(wc);
              enqueue xq until empty(wc); join rc
        W     enqueue xq until empty(wc)&empty(rc); join wc
```

The results in Figure 7 were obtained on 23 August 1979. The times given are

Figure 7. CPU times for ASP tests

Name	Time	WexR	WexW	RexW	RpoW	WpoR	WpoW	Wpri	Rpri	|| R	GS(R)	GS(W)
FIFO	21	T	T	T	T	T	T	F	F	T	T	T
RPRI	35	T	T	T	T	?	T	F	T	T	T	T
WPRI1	47	T	T	T	?	T	T	T	F	?	T	T
WPRI2	67	T	T	T	?	T	T	T	F	?	T	T
STARVE	24	T	T	T	?	?	T	F	?	T	?	?
NRPRI	36	T	T	T	T	?	T	F	T	T	T	T
NWPRI	30	T	T	T	?	T	T	T	F	T	T	T

Time is given in CPU seconds.
T indicates a proved clause, F indicates a disproved clause.
? indicates a clause not proved or disproved.

CPU seconds for running all of the tests shown.[17] The test cases are explained in detail at the bottom of the figure. Each column after the Time column represents a different test, given by a specification clause. A T represents a proven specification clause. An F represents a specification clause proven to be always false. A ? represents a specification that could not proven true or false. In the serializers represented in the table below there were no cases where the program was not capable enough to prove or disprove a clause that was always true or false. In general, if the program can not prove or disprove a result, it is either due to a clause that is true for some histories and false for others, or it is due to a weakness in the verification methodology, and ASP will be

[17] These tests were performed on 23 August 1979, using a Decsystem-2060T. ASP occupies about 100K 36-bit words of memory, of which about 60K words are due to the CLU support system. No appreciable paging activity took place.

unable to distinguish the two.

6.10 Summary of methods used

This section provides a concise summary of the methods we have used in ASP. In this summary we follow the order of steps used in ASP, rather than precisely following the order of presentation for this chapter.

- Static analysis is performed once for any given serializer code to determine initial clauses that are derivable solely from the node graphs for the serializer operations. The remainder of the steps are performed for any given specification clause.

- Representations are introduced for the transactions mentioned in the specification clause.

- For any specification clause of the form P ⊃ Q, the clause P is asserted, and we attempt to derive Q through use of the insertion rules, which are the rules Transitivity, Event before PX, Event after PX, Event before GX, Event after GX, and Event from FIFO. If these rules are not sufficient to prove Q, further methods must be used.

- The Event from ready queue rule, which reflects the priority of service given to internal queues over the external queue, is applied where feasible. This is known as "checking for ready queues." This rule may result in the invention of anonymous transactions, which are essential to the proof by contradiction that the preconditions for the rule are met. Anonymous transactions may also be used in the EVT rule, which is subsidiary to the checking for ready queues.

- When the clause Q is still not proved, and the order of certain enter events is not known, although the events are known to occur, ASP tries all permutations of such events. If Q can be proven for every such permutation, then P ⊃ Q has been proved.

* Proof of guaranteed service is performed by assuming that a transaction is blocked in a queue, then proving that a ready queue must result at some "quiet point." Although this method is limited, it can be proven to be correct, and works for a variety of cases.

7. Interaction of Serializers

In previous chapters, we introduced the serializer construct, presented a specification language for serializers, and demonstrated some verification techniques. Our discussion has been limited to single instances of simple serializers. Yet if we are to reach our objective of modularity, we must examine how serializers interact.

In this chapter we present an application of serializers that incorporates the use of multiple serializers. We are especially concerned that serializer use can be nested, so that the techniques for modular decomposition of programs in a single process domain can be applied to a multiple process domain.

The example we have chosen is the use of serializers to control concurrent access to a simple file system. For this example we will assume that objects in primary memory can be shared by several processes running on a single processor. This choice is made to keep the example simple enough to be tractable, since presenting a distributed version of a filing system involves issues well beyond the scope of this thesis.

We start this chapter with a presentation of the simple file system, including a discussion of the abstractions involved. We then show two of the serializers used to control concurrent access to the file system, and show how the specifications are similar to the readers priority variant of the readers-writers problem. Further sections concern methods for introducing serializers for abstractions that were written for single process environments, and a discussion of higher-level transactions.

7.1 The file system

The structure of the file system is based on directories and files. A directory is a map from names (expressed by strings) to entries, which are either files or directories. If directory Y is named in directory X, then Y is a *child* directory of X, and X is the *parent* directory of Y. There is a single directory, called the *root directory*, that has no parent directory. Files and child directories may be added to or deleted from directories. A simple provision is made for iterating over the names of a directory. It is possible to get the number of entries in a directory, and to determine which directory (if any) is the parent of a given directory. For most operations, a directory must be *open* for the user to perform those operations. Opening a directory is accomplished by the *directory$open_dir* operation. The directory structure is acyclic.

A file is an array of pages, where a page is some fixed length unit of data. Pages on primary memory may be read from or written to any existing page in a file. Pages may be added to or removed from the end of a file. A file may be named by only one directory. It is possible to get the number of pages in a file, and to determine which directory names the file. As with directories, a file must be *open* for the user to perform most operations. A file opened by *directory$open_private* can only be accessed by a single process, while a file opened by *directory$open_public* can be accessed by any number of processes (although a practical system might impose some reasonable limit). A file is closed by the *file$close* operation.

At this point, some additional explanation of the open and close operations is in order. First, we have made the open operations work on directories, since directories are the logical means for initially accessing files and child directories. We have made the close operation work only on the object that the open provides, which prevents users from closing a file (or directory) except when they have acquired that file or directory object through an open operation. Second, we have two different kinds of open operation on files: *open_public*, for simultaneous access among several processes (or users), and *open_private*, for sole access. We can associate an *open count* with each file or directory object. This count is increased for every open operation, and decreased by every close operation. The directory$open_private operation will only succeed when the count is zero, and upon successful completion, prevents any increase in the count. The directory$open_dir operation opens a child directory such that multiple processes can access it concurrently.

In presenting the file system example we will concentrate on showing the interface of the file and directory data abstractions and the code for the file and directory serializers. It will not be necessary to show the implementation of the file and directory data abstractions, although we will discuss some of the details as necessary.

Figures 8 and 9 present the interface specifications for the directory and file clusters. As a first approximation, these are the same interface specifications that are used for the corresponding directory and file serializers. Each operation interface names the operation, the types of the arguments, the types of the returned objects, and the types of exceptions that can be signalled. We include some comments that indicate

Figure 8. File interface

A file may be described as an array of pages that exists on remote storage. It can be randomly accessed, and can be extended or retracted at one end. An open file can only be obtained through use of a directory open_private or open_public operation. No operations can be performed on a closed file except for is_open. The following file operations are available to the user (others will be discussed later in the chapter):

get_parent (file) returns (directory) signals (file_closed)

Get parent directory of file if file is open, otherwise signal file_closed.

get_name (file) returns (string) signals (file_closed)

Get name of file as a string if file is open, otherwise signal file_closed.

get_size (file) returns (int) signals (file_closed)

Get number of pages in the file if it is open, otherwise signal file_closed.

is_open (file) returns (bool)

Return true if file is open, false if it is not.

read_page (file, int, page) signals (file_closed, bounds)

Copy a page of information from the given location in the file into the given page in primary memory, provided that the file is open. Signal bounds if the location is invalid (less than 0, greater than or equal to the size). Signal file_closed if the file is closed.

write_page (file, int, page) signals (file_closed, bounds)

Copy a page of information from the given page in primary memory to the given location in the file. Signal bounds if the location is invalid, file_closed if the file is closed.

close (file) signals (file_closed)

Close file if it is open, otherwise signal file_closed.

add_page (file, page) signals (file_closed, no_room)

Add a page to end of file, signalling if the file is closed or there is insufficient room to complete.

rem_page (file) signals (file_closed, empty)

Remove a page from the end of the file, signalling if the file is closed or the file has no pages.

For concurrent access, there are the following classes of operations:

Info:	can overlap with any but sole access
Read:	can overlap with read or info access
Write:	can overlap with info access
Sole:	can not overlap

The operations in each class are:

Info:	get_parent, get_name, get_size, is_open
Read:	read_page
Write:	write_page
Sole:	close, add_page, rem_page

Figure 9. Directory interface

A directory functions as a symbol table of entries, where each entry is either a file or another directory. Entries can be created, deleted or opened using the directory. The following operations are publicly available:

root () returns (directory)
Get root directory, which is always open (this operation does not require possession).

get_parent (directory) returns (directory) signals (none, dir_closed)
Get parent directory, signalling none if the given directory is the root directory, and dir_closed if the given directory is closed.

get_size (directory) returns (int) signals (dir_closed)
Get number of entries in the given directory, signalling if the directory is closed.

get_name (directory) returns (string) signals (dir_closed)
Get name of the given directory, signalling if the directory is closed.

is_open (directory) returns (bool)
Return true if the given directory is open, false if it is not.

info (directory, string) returns (bool, int, bool)
signals (none, dir_closed)
Return information about the named entry: a boolean indicating the kind of entry (true if entry is a file, false if not), the size (in pages if a file, number of entries if a directory), and a boolean indicating whether the entry is open. Signal appropriate errors if they occur.

next (directory, string) returns (string) signals (none, dir_closed)
Get next entry name after named entry, using string ordering.

open_private (directory, string) returns (file)
signals (none, opened, dir_closed)
Open named file in given directory for sole use, signalling appropriate errors if they occur.

open_public (directory, string) returns (file)
            signals (none, locked, dir_closed)

Open named file in given directory for shared use, signalling appropriate errors if they occur (locked is signalled if entry is open for sole use).

open_dir (directory, string) returns (directory)
          signals (none, dir_closed)

Open named child directory in given directory, signalling appropriate errors if they occur.

close (directory) signals (dir_closed, open_entries, root)

Close the given directory, signalling if it is the root, or it is already closed, or open entries exist.

add_dir (directory, string)
          signals (no_room, duplicate, bad_name, dir_closed)

Add new (empty) child directory entry with given name. Signal if there is insufficient room, an existing file or directory of the same name, a bad directory name given, or the directory is closed.

add_file (directory, string)
          signals (no_room, duplicate, bad_name, dir_closed)

Add new (empty) file entry to directory. Signal if there is insufficient room, an existing file or directory of the same name, a bad file name given, or the directory is closed.

delete (directory, string) signals (none, opened, dir_closed)

Delete named entry in given directory, signalling appropriate errors. If entry is a directory, all of its entries are deleted as well.

There are four classes of operations requiring possession:

Fixed info:	can overlap with any but sole access
Variable info:	can overlap with variable or fixed info access
Opening:	can overlap with fixed info access
Sole:	can not overlap

The operations in each class are:

Fixed info:	get_parent, get_name, is_open
Variable info:	get_size, info, next
Opening:	open_private, open_public, open_dir
Sole:	close, add_dir, add_file, delete,

the intended effects of the operation. After the operations have been described, we divide the operations into classes based on which operations may overlap in execution with which other operations (when executed on the same serializer object).

One way to design a system that involves concurrency is to design it for a single-process system first, then add multiple processes for portions of tasks that can be performed concurrently, and add serializers to control access to shared objects. In the file system example, however, we have assumed that the file system would be accessed by multiple processes. This assumption has influenced the choice of operations, especially in providing for opening and closing of files. Even so, the single-process model of design is useful. Concurrent execution of operations is only permitted where the effects on the state of the files are the same as some serial execution of operations where concur-ent execution is prohibited. It may not be possible to obtain the maximum concurrency in this fashion, since certain operations could be allowed to execute concurrently in part. But increased concurrency is purchased at the cost of increased complexity.

One simplifying assumption has been made regarding file objects that may appear to be unrealistic. That is, a file on secondary memory has at most one file object in primary memory controlling access (this is also true for directories). Unfortunately, this allows a user to open a file once to obtain the controlling object, then close the file several times, thereby completely closing the file to access by other processes. To remedy this, in a real system it would be desirable to have a second level of indirection for files such that every successful execution of an *open_public* operation returned a unique controlling file object. The additional level of file object would be used to create a separate file object for each *open_public* operation, such that the file abstraction presented to the user would only allow a file object to be closed once. A full presentation of both levels of file has no advantage over a presentation of a single level, so we only discuss the system_file version of files, which is supported by the file cluster and its associated serializer.

7.2 File and directory serializers

Figures 10 and 11 on the following pages present the directory and file serializers. Note that we have added several operations that are "hidden" to the "normal" user. We would expect access to these operations to be regulated through some library mechanism, such that a normal user would see a subset of the interface of an abstraction, while a "privileged" user would be allowed to access more of that interface. In some cases, and in particular for this file system, access to privileged operations would be restricted to only allowing use by implementations of particular abstractions, rather than allowing access based on the identity of the person using the system.[18]

[18] Such protection could also be provided to some extent by establishing a block structure for clusters and serializers. We have chosen to retain CLU's approach to modules, and assume that protection is accomplished by other means.

Figure 10. File serializer

```
file = serializer is

    % The following operations are publically available.
    get_parent, % get parent directory
    get_name,   % get name of file
    get_size,   % get # of pages in file
    is_open,    % test open-ness of file
    read_page,  % read a page
    write_page, % write a page
    close,      % close file
    add_page,   % add a page to end of file
    rem_page,   % remove a page from end of file

    % Note: delete can only be called from _directory$delete
    delete,     % delete the contents of a file

    % The wrap operation can only be used by the _file cluster
    % to turn a _file object into a file serializer object.
    wrap

    % The operations with cvt arguments can be split into four
    % classes, depending on which operations can overlap in
    % execution with which other operations.

    %   - Class -       - Overlap -
    %   Info:           Info, Read, Write
    %   Read:           Info, Read
    %   Write:          Info
    %   Sole:           -

    %   - Class -       - Members -
    %   Info:           get_parent, get_name, get_size, is_open
    %   Read:           read_page
    %   Write:          write_page
    %   Sole:           close, add_page, rem_page, delete

    rep = struct[slow_q, fast_q: queue,
                 sole_c, write_c, read_c, info_c: crowd,
                 f: _file]

    wrap = proc (_f: _file) returns (cvt)
        return (rep${f: _f, fast_q, slow_q: queue$create(),
                     sole_c, info_c, read_c, write_c: crowd$create()})
        end wrap

    get_parent = proc (f: cvt) returns (directory)
            signals (file_closed)
        enqueue f.fast_q until crowd$empty(f.sole_c)
        join f.info_c
```

```
         return (_file$get_parent(f.f)) resignal file_closed
         end
    end get_parent

get_name = proc (f: cvt) returns (string)
        signals (file_closed)
    enqueue f.fast_q until crowd$empty(f,sole_c)
    join f.info_c
         return (f.f.name) resignal file_closed
         end
    end get_name

get_size = proc (f: cvt) returns (int)
        signals (file_closed)
    enqueue f.fast_q until crowd$empty(f.sole_c)
    join f.info_c
         return (f.f.size) resignal file_closed
         end
    end get_size

is_open = proc (f: cvt) returns (bool)
    enqueue f.fast_q until crowd$empty(f.sole_c)
    join f.info_c
         return (_file$is_open(f.f))
         end
    end is_open

read_page = proc (f: cvt, index: int, p: page)
        signals (file_closed, bounds)
    enqueue f.fast_q until crowd$empty(f.sole_c)
        & crowd$empty(f.write_c)
    join f.read_c
         _file$read(f.f, index, page) resignal file_closed, bounds
         end
    end read_page

write_page = proc (f: cvt, index: int, p: page)
        signals (file_closed, bounds)
    enqueue f.slow_q until queue$empty(f.fast_q)
    enqueue f.fast_q until crowd$empty(f.sole_c)
        & crowd$empty(f.read_c) & crowd$empty(f.write_c)
    join f.write_c
         _file$write(f.f, index, p) resignal file_closed, bounds
         end
    end write_page

close = proc (f: cvt) signals (file_closed)
    enqueue f.slow_q until queue$empty(f.fast_q)
    enqueue f.fast_q until crowd$empty(f.sole_c)
        & crowd$empty(f.info_c) & crowd$empty(f.read_c)
        & crowd$empty(f.write_c)
    join f.sole_c
         _file$close(f.f) resignal file_closed
         end
    end close

add_page = proc (f: cvt, p: page)
        signals (file_closed, no_room)
    enqueue f.slow_q until queue$empty(f.fast_q)
    enqueue f.fast_q until crowd$empty(f.sole_c)
        & crowd$empty(f.info_c) & crowd$empty(f.read_c)
        & crowd$empty(f.write_c)
    join f.sole_c
         _file$add_page(f.f, p) resignal file_closed, no_room
         end
    end add_page

rem_page = proc (f: cvt)
        signals (file_closed, no_room)
    enqueue f.slow_q until queue$empty(f.fast_q)
    enqueue f.fast_q until crowd$empty(f.sole_c)
        & crowd$empty(f.info_c) & crowd$empty(f.read_c)
        & crowd$empty(f.write_c)
    join f.sole_c
         _file$rem_page(f.f, p) resignal file_closed, no_room
         end
    end rem_page

% Note: called by _dir$delete

delete = proc (f: cvt)
        signals (file_open, file_deleted)
    enqueue f.slow_q until queue$empty(f.fast_q)
    enqueue f.fast_q until crowd$empty(f.sole_c)
        & crowd$empty(f.info_c) & crowd$empty(f.read_c)
        & crowd$empty(f.write_c)
    join f.sole_c
         % Note: use hidden _file$delete operation
         % to delete contents of file. _file$delete is
         % only used by file$delete.
         _file$delete(f.f, p) resignal file_open, file_deleted
         end
    end delete

end file
```

Figure 11. Directory serializer

```
directory = serializer is
    root,           % get root directory
    get_parent,     % get parent directory
    get_name,       % get name of directory
    is_open,        % test open-ness of directory
    get_size,       % get # of entries
    info,           % return info about named entry
    next,           % get next entry name after named entry
    open_private,   % open file for sole use
    open_public,    % open file for sharing
    open_dir,       % open sub-directory
    close,          % close this directory
    add_dir,        % add new sub-directory entry
    add_file,       % add new file entry
    delete,         % delete named entry

    % The wrap operation can only be used by the _directory cluster
    % to turn a _directory object into a directory serializer object.
    wrap

    % The operations can be split into six classes, depending on
    % which operations can overlap in execution with which other
    % operations.

    % - Class -        - Overlap -
    % Root:            Root, Fixed, Variable, Opening, Sole
    % Fixed info:      Root, Fixed, Variable, Opening
    % Variable info:   Root, Fixed, Variable
    % Opening:         Root, Fixed
    % Sole:            Root

    % - Class -           - Members -
    % Root:               root
    % Fixed info:         get_parent, get_name, is_open, get_size
    % Variable info:      info, next
    % Opening:            open_private, open_public, open_dir
    % Sole:               close, add_dir, add_file, delete

    rep = struct[slow_q, fast_q: queue,
                 sole_c, open_c, var_c, fixed_c: crowd,
                 dir: _directory]

    % The wrap procedure is used by the _directory cluster
    % to turn a _directory object into a directory serializer
    % object.  This operation can only be used by the
    % _directory$root and _directory$add_dir operations.

    wrap = proc (d: _directory) returns (cvt)
        return (rep$create(dir: _d,
                       slow_q, fast_q: queue$create(),
                       sole_c, open_c, var_c, fix_c:
                             crowd$create()})
    end wrap

root = proc () returns (directory)
    % note: _directory$root uses the wrap operation
    return (_directory$root())
    end root

get_parent = proc (d: cvt) returns (directory)
        signals (none, dir_closed)
    enqueue d.fast_q until crowd$empty(d.sole_c)
    join d.fix_c
        return (_directory$get_parent(d.dir))
            resignal none, dir_closed
        end
    end get_parent

get_name = proc (d: cvt) returns (string)
        signals (dir_closed)
    enqueue d.fast_q until crowd$empty(d.sole_c)
    join d.fixed_q
        return (_directory$get_name(d.dir)) resignal dir_closed
        end
    end get_name

is_open = proc (d: cvt) returns (bool)
    enqueue d.fast_q until crowd$empty(d.sole_c)
    join d.fixed_q
        return (_directory$is_closed(d.dir))
        end
    end is_open

get_size = proc (d: cvt) returns (int)
        signals (dir_closed)
    enqueue d.fast_q until crowd$empty(d.sole_c)
    join d.var_c
        return (_directory$get_size(d.dir)) resignal dir_closed
        end
    end get_size

info = proc (d: cvt, name: string)
        returns (bool, int, bool) signals (none, dir_closed)
    enqueue d.fast_q until crowd$empty(d.sole_c)
        & crowd$empty(d.open_c)
    join d.var_c
        file_ness: bool, size: int, open_ness: bool
            := _directory$info(d.dir) resignal dir_closed, none
        return (file_ness, size, open_ness)
        end
```

```
  end info

next = proc (d: cvt, name: string) returns (string)
    signals (none, dir_closed)
  enqueue d.fast_q until crowd$empty(d.sole_c)
    & crowd$empty(d.open_c)
  join d.var_c
    return (_directory$get_next(d.dir))
      resignal dir_closed, none
    end
  end next

open_private = proc (d: cvt, name: string) returns (file)
    signals (none, opened, dir_closed)
  enqueue d.slow_q until queue$empty(d.fast_q)
  enqueue d.fast_q until crowd$empty(d.sole_c)
    & crowd$empty(d.open_c)
  join d.open_c
    return (_directory$open_private(d.dir, name))
      resignal dir_closed, none, locked
    end
  end open_private

open_public = proc (d: cvt, name: string) returns (file)
    signals (none, locked, dir_closed)
  enqueue d.slow_q until queue$empty(d.fast_q)
  enqueue d.fast_q until crowd$empty(d.sole_c)
    & crowd$empty(d.open_c)
  join d.open_c
    return (_directory$open_public(d.dir, name))
      resignal dir_closed, none, locked
    end
  end open_public

open_dir = proc (d: cvt, name: string) returns (directory)
    signals (none, dir_closed)
  enqueue d.slow_q until queue$empty(d.fast_q)
  enqueue d.fast_q until crowd$empty(d.sole_c)
    & crowd$empty(d.open_c)
  join d.open_c
    return (_directory$open_dir(d.dir, name))
      resignal dir_closed, none
    end
  end open_dir

close = proc (d: cvt)
    signals (dir_closed, open_entries)
  enqueue d.slow_q until queue$empty(d.fast_q)
  enqueue d.fast_q until crowd$empty(d.sole_c)
    & crowd$empty(d.var_c) & crowd$empty(d.fix_c)
    & crowd$empty(d.open_c)
    join d.sole_c
        _directory$close(d.dir) resignal dir_closed, open_entries
        end
    end close

add_dir = proc (d: cvt, name: string)
        signals (no_room, duplicate, bad_name, dir_closed)
    % note: _directory$add_dir uses the wrap operation
    enqueue d.slow_q until queue$empty(d.fast_q)
    enqueue d.fast_q until crowd$empty(d.sole_c)
        & crowd$empty(d.var_c) & crowd$empty(d.open_c)
    join d.sole_c
        _directory$add_dir(d.dir)
            resignal no_room, duplicate, bad_name, dir_closed
        end
    end add_dir

add_file = proc (d: cvt, name: string)
        signals (no_room, duplicate, bad_name, dir_closed)
    enqueue d.slow_q until queue$empty(d.fast_q)
    enqueue d.fast_q until crowd$empty(d.sole_c)
        & crowd$empty(d.var_c) & crowd$empty(d.open_c)
    join d.sole_c
        _directory$add_file(d.dir)
            resignal no_room, duplicate, bad_name, dir_closed
        end
    end add_file

delete = proc (d: cvt, name: string)
        signals (none, opened, dir_closed)
    enqueue d.slow_q until queue$empty(d.fast_q)
    enqueue d.fast_q until crowd$empty(d.sole_c)
        & crowd$empty(d.var_c) & crowd$empty(d.fix_c)
        & crowd$empty(d.open_c)
    join d.sole_c
        _directory$delete(d.dir) resignal dir_closed, open_entries
        end
    end delete

end directory
```

To distinguish between the data abstractions and the serializer abstractions of the same interface, we will use the names *directory* and *file* for the serializer abstractions, and *_directory* and *_file* for the data abstractions. The user in a multiple process system would only be allowed to access the operations of the serializer abstractions, which would utilize the operations of the data abstractions.

In the above two serializers, there are classes of operations that can be strictly ordered on the basis of the execution of any operation from one class excluding the execution of any operation from another class. The order is from most permissive to least permissive, with operations that return information generally being the most permissive, since they can be executed concurrently. This ordering allows us to construct serializers that follow the general plan of the readers-writers problem. If an operation can execute concurrently with another invocation of the same operation, it is considered to be a reader; otherwise it is a writer. In the above serializers, we have adopted a readers priority approach, with the information gathering operations having higher priority. It would be equally correct to adopt a FIFO approach or a writers priority approach, but different performance would result.

The restrictions on simple serializers must be relaxed slightly to allow us to write the file and directory serializers. The most important addition is the exception mechanism, which includes a **signals** clause in the operation interface and a **resignal** clause at the end of any statement. This addition does not greatly add to the complexity of our model, since we only use the exception mechanism in the same manner as the

return statement.[19]

We retain the important limitation, which is to return or signal directly after invoking the operation of the data abstraction. The other addition is to allow local variables, which we use in *directory$info* to hold the results of an invocation that returns multiple objects. The effect of this addition is also minor, since we immediately return those results unchanged.

7.3 Specifications for file and directory serializers

The specifications for the file and directory serializers are similar to the readers priority readers-writers problem. Therefore, we will only present illustrative examples, rather than full specifications. One useful abbreviation is to use the first letters of the operation classes, rather than the operations, to name transactions. This gives us the following transaction names for file operation classes:

I: an Info class transaction

R: a Read class transaction

W: a Write class transaction

S: a Sole class transaction

For directory operation classes, we can use the same specifications, except that the

[19] In CLU, when an operation signals an exception, the invocation terminates, and the *immediate* caller is given the opportunity to handle the exception. A common method of handling an exception is to reflect it to yet another level via **resignal**. An invocation that signals an exception is not resumed. For further details, see [Liskov 79a].

transaction symbols have the following interpretation:

I: a Fixed Info class transaction

R: a Variable Info transaction

W: an Opening class transaction

S: a Sole class transaction

In the remainder of this section we use the class names of the file serializer (Info, Read, Write, and Sole) with the understanding that the remarks also apply to the corresponding directory classes.

The most important specifications are those that relate to the exclusion of certain operations by others. If these specifications are violated we obtain invalid result values. The complete exclusion specifications are:

I-join < S-join ⊃ I-leave < S-join

R-join < W-join ⊃ R-leave < W-join

R-join < S-join ⊃ R-leave < S-join

W-join < R-join ⊃ W-leave < R-join

W1-join < W2-join ⊃ W1-leave < W2-join

W-join < S-join ⊃ W-leave < S-join

S-join < I-join ⊃ S-leave < I-join

S-join < R-join ⊃ S-leave < R-join

S-join < W-join ⊃ S-leave < W-join

S1-join < S2-join ⊃ S1-leave < S2-join

A number of priority specifications might be proposed. The readers priority specification used in Chapter 6 is:

$$\text{W1-enter} < \text{W2-enter} < \text{R1-enter} < \text{W1-join} \supset \text{R1-join} < \text{W2-join}$$

The same specification clause holds for the file and directory serializers. To give more complete priority specifications, we introduce two new classes of transactions: SW, which contains all Sole and Write transactions; and IR, which contains all Info and Read transactions. Using these new classes, the priority specification becomes:

$$\begin{aligned}&\text{SW1-enter} < \text{SW2-enter} < \text{IR1-enter} < \text{SW1-join}\\&\supset \text{IR1-join} < \text{SW2-join}\end{aligned}$$

The following specification specifies concurrency for Read transactions, and is a slight adaptation of the concurrency specification in Chapter 6:

$$\begin{aligned}&\text{R1-enter} < \text{R2-enter} < \text{R1-leave}\\&\ \ \&\ \text{GX(R1-enter, R2-enter, W*-enter)}\\&\ \ \&\ \text{GX(R1-enter, R2-enter, S*-enter)}\\&\supset \text{R2-join} < \text{R1-leave}\end{aligned}$$

The difference lies in the addition of the exclusion of enter events from the Sole class of transactions. The above specification can also be proven for Read and Info transactions by substituting R for R1 and I for R2 to get one clause, and I for R1 and R for R2 to get the other. Finally, the following specification indicates where a Write transaction *must* overlap with an Info transaction:

$$\begin{aligned}&\text{W-enter} < \text{I-enter} < \text{W-leave}\\&\ \ \&\ \text{GX(W-enter, I-enter, S*-enter)}\ \&\ \text{GX(W-enter, I-enter, W*-enter)}\\&\supset \text{I-join} < \text{W-leave}\end{aligned}$$

The service specifications are as simple as for the readers-writers problem: each request must receive a reply. The service specifications are:

@I-enter ⊃ @I-exit
@R-enter ⊃ @R-exit
@W-enter ⊃ @W-exit
@S-enter ⊃ @S-exit

We have shown that the specifications for the file and directory serializers are similar to the readers priority example used in Chapter 6. This may not be surprising, since the problems and solutions are similar, but the lack of such a surprise is precisely one of our goals.

One point about the specifications that we have discovered through the above example is the usefulness of dividing the operations into classes, and providing the specifications for the classes rather than for the single operations. Using class-oriented specifications promises to provide more concise specifications while retaining the precision we desire.

The verification techniques we discussed in Chapter 5 and Chapter 6 remain valid for both the file and directory serializers. The only additions we would make would be to introduce classes of operations into the verification as we have for the specification. When two serializer operations are sufficiently similar it should be possible to use the proof of one in the proof of the other, as is the case for file operations in the same specification class. We will not propose techniques for determining how much similarity is sufficient, although we regard the issue as being worthy of further research.

7.4 Guidelines for addition of serializers

In a system where data abstractions are used, we believe it likely that some library of abstractions will become useful, and eventually indispensable. Further, we consider it likely that many of these abstractions will be initially designed for a single-process environment.[20] If we are to use these data abstractions in a multiple-process environment, and the corresponding objects are to be shared between processes, we can either rework the abstractions for that purpose, or we can provide a mechanism for controlling concurrent access that requires no change to the data abstractions. The serializer construct was designed along the latter lines. This section discusses how that approach could be made largely automatic.

As a first approximation, we assume that each operation has exclusive use of the resource, then introduce serializer abstractions as replacements for data abstractions in order to permit concurrency while prohibiting conflict and deadlock. This is a simple strategy, and is not intended to cover all situations, although we believe it to be an important first step.

When a serializer abstraction is substituted for a data abstraction in a program, yet the data abstraction is retained as part of the implementation of the serializer

[20] Even if for no other reason than programmer inertia.

abstraction, we may be faced with problems that result from having two abstractions in the place of one. If we wish to integrate a newly serialized abstraction into a system that has been created with the old data abstraction, we need a linking mechanism that will allow the operations of the serializer abstraction to be substituted for operations of the original abstraction in old user programs. If the interface to the serializer abstraction is compatible with the interface of the original data abstraction, and both abstractions have isolated representations, then this linkage mechanism allows graceful upgrading of programs that use the original data abstraction.

However, the representation of the original data abstraction *is* exposed to the operations of that data abstraction. Here the splitting of the original abstraction is more difficult. In most cases, we expect that an automatic "rewrite" of the data abstraction would be easi'y made by a program. If we call the type introduced by the data abstraction DA, and the type introduced by the serializer abstraction SA, then the following rules allow such an automatic rewrite:

- Occurrences of DA in the cluster for DA are changed to SA, including occurrences of DA in the interface of operations of DA, provided that they do not result from uses of cvt. Thus, a component of the representation of DA that was an object of type DA would become an object of type SA. In the file system example, this would be true for the case of the get_parent operation of the directory abstraction, since the get_parent operation of _directory (DA) must return a directory object (SA), and not a _directory object (DA). This is also true of the open_private, open_public, and open_dir operations.

- Operations of DA that have cvt appearing in their headers must have DA appear in the interface specifications where a corresponding cvt appears in the operation header. These are operations that explicitly access the representation of DA, so a conversion of DA to SA is not reasonable.

- The **up** and **down** operations convert between the representation and the DA type, not the SA type. This is consistent with the treatment of **cvt**.

- We introduce an operation, called *wrap*, that takes a DA object and returns a newly created SA object that encapsulates the DA object. The wrap operation is used to create a new SA object in operations that create new DA objects and need (due to our first transaformation) to use SA objects.

If the above translation results in a type error then the automatic rewrite is not performed, and a manual rewrite must be performed. Such a case could arise from an operation that accepted an argument of type DA, then explicitly used **down** to attempt to access the representation. The transformation would have changed the use of DA into SA, but the **down** operation would only work for an object of type DA, and fails (due to static type checking) with an SA object.

In addition, a data abstraction may have to be rewritten if it supports cyclic objects. If operations of DA call operations of SA, which in turn call operations of DA, a cyclic data structure can cause deadlock by having access to an object being blocked by an incomplete access to the same object by the same process. Access to cyclic objects is discussed later in this chapter.

There are two reasons to believe that a rewrite of the original data abstraction will not be a difficult process even if it cannot be done automatically. First, the amount of detail to be changed is likely to be small. After all, the intent of the data abstraction has not changed. There is only the additional distinction between serializer abstraction and data abstraction. Second, we believe that it will be rare that any code except for the implementation of the serializer and data abstractions will be allowed to use the data abstraction. The intention of this transformation is to make the rest of the system use the serializer abstraction. Therefore, the number of places to be changed is also likely to be small.

In the file system example, there is a case where the use of the automatic splitting of types may provide serializers where none are needed. In particular, if the directory information is implemented using a file, then the serializer for the directory may provide sufficient protection for the file object used to implement the directory. In such a case, the transformation from DA to SA would provide an unnecessary level of serializer. A rewrite of the _directory cluster would then be desirable to promote efficiency. This efficiency argument actually works in favor of our separation of data and serializer abstractions, since if they were inextricable, the optimization described could not be performed.

The above rewrite process has been applied to the _file and _directory serializers. In particular, the operations _directory$open_private and _directory$open_public now return file objects, which are supported by the file serializer. Further, the operation _directory$open_dir returns a directory object, which is supported by the directory serializer. The *wrap* operations shown in the file and directory serializers are used to enclose a _file or _directory object in a file or directory serializer. The wrap operations are used whenever a new _file or _directory object is created.

In any reasonable implementation of the _directory cluster there will be a list of the open files and child directories for any _directory object. In this case, the automatic rewrite we mentioned above informs us of a type conflict: the list of open files and directories must be for the file and directory objects supported by the serializers, and not the _file and _directory objects supported by the clusters.

7.5 Higher-level transactions

Suppose procedures P and Q use operations on a shared data object X of type T. We have recommended that a serializer object should be introduced for X to ensure that the operations of T performed on X do not interfere with each other. However, the user may intend that P and Q do not overlap. The serializer for object X does not enforce this restriction. One solution is to introduce a further encapsulation of X in order to perform operations P and Q such that they do not overlap.

A difficulty with the introduction of further abstraction levels is that the designer of a system may not know how the user will be using the system, and cannot provide the appropriate abstractions in advance. This inability to forecast is certainly present in our file system example, since the user may wish to have a process perform several operations on a file (or on several files) such that no other process will access the file (or files) while those operations are being performed. The file system example provides no solution to this problem in general, although we can attack certain special cases.

A limited solution to the above problem can be achieved by adding a new operation, *update*, to the file serializer. The text of this operation is shown in Figure 12. The update operation performs a sequence of read operations on a file, then performs a computation supplied as a procedure by the user on data supplied by the user, then performs a sequence of writes on the same file. In our simple solution, the entire update operation is performed without allowing overlapping reads or writes on the file. If more concurrency is desired, update operations that do not have overlapping sets of pages can be permitted to proceed in parallel, providing that the underlying _file abstraction will permit this.

Figure 12. Update operation

```
% The update operation is intended to perform a sequence of
% reads, an arbitrary computation, and a sequence of writes.
% The entire procedure should executed without overlapping
% other write operations or other update operations.  This
% procedure resignals an error on reading or writing, or an
% abort error from the arb procedure.  An error that is
% resignalled after the first write has been finished will
% leave the writes only partially completed.

update = proc [dt: type]
      (f: cvt, reads, writes: spair, arb: pt, data: dt)
           signals (file_closed, bounds, abort)

    pair = struct [pgnum: int, pg: page]
    spair = sequence [pair]
    pt = proctype (dt, spair, spair) signals (abort)

    % wait for write access to resource to be OK
    enqueue f.slow_q until queue$empty (f.fast_q)
    enqueue f.fast_q until crowd$empty (f.sole_c)
                             & crowd$empty (f.write_c)
                             & crowd$empty (f.read_c)

    % join the crowd to show that we are going to write
    join f.write_c

         % perform the reads into the given memory pages
         % from the given file pages
         for p: pair in spair$elements(reads) do
             _file$read(f.f, p.pgnum, p.pg)
             end

         % perform the arbitrary computation
         % (modifying the given memory pages)
         arb(data, reads, writes)

         % perform the writes from the given memory pages
         % into the given file pages
         for p: pair in spair$elements(writes) do
             _file$write(f.f, p.pgnum, p.pg)
             end
         end resignal file_closed, bounds, abort

    end update
```

8. Conclusions

In this thesis we have been concerned with verifiable control of concurrent access to resources. In this pursuit we have presented a language construct for controlling concurrent access, a definition of the semantics of this construct, a specification language for describing varieties of concurrency control for instances of the construct, methods to verify that instances of the construct satisfy their specifications, a program for performing this verification automatically, and a discussion of some of the interactions possible between instances of this construct.

In separating the control of concurrency from the data access, we have attempted to apply this separation to the programming language, the semantic model, the specifications, and the verification system. The objective has been to modularize the construction and verification of programs involving concurrency. By this modularization, the problems associated with construction and verification become more tractable. The results of our research indicate that this modularity can be achieved, at least for the simple serializers we have discussed.

In this chapter we discuss how extensions to serializers require extensions to our verification techniques. Most of these extensions require significant further research. Then we present closing remarks to sum up the contributions of this thesis.

8.1 Verification of serializer extensions

In this section we briefly consider how extensions to serializers affect our semantic model and verification methods. This is the area where further research is most necessary and most difficult. Our success in verifying simple serializers can be largely attributed to the limitations we have imposed. We believe that further success in verifying concurrency control lies in selective relaxation of these limitations.

8.1.1 Adding boolean variables and boolean expressions

To add simple boolean variables and boolean expressions to serializers requires the following changes to the semantic model:

- * The node graphs must be extended to handle declaration and assignment of boolean variables. These variables must further be distinguished as either local variables, which are instantiated on each transaction; or global variables, which are components of the serializer representation.

- * The semantic equations must be extended to handle evaluation of boolean expressions. This will require examining finite histories for the last assignment to any boolean variable. One of the most important changes to evaluation is that evaluation must take place in the context of a transaction, since expressions may involve local variables.

- * There must be some indication of the initial state of a serializer object. This is easily accomplished by representing the serializer state as the result of some initial assignments to representation components.

To illustrate the kinds of serializers and verifications that are possible with the addition of boolean variables, consider the case where we are limited to boolean variables as part of the representation, and the only legal boolean expressions are **true**, **false**, and simple components of the representation. As an example, we present the following abbreviated serializer:

```
xop = proc (x: cvt, ...)
        enqueue x.q1 until x.b & crowd$empty(x.c)
        join x.c; ...; end
        x.b := false
        end xop

yop = proc (x: cvt, ...)
        enqueue x.q2 until ~x.b & crowd$empty(x.c)
        join x.c; ...; end
        x.b := true
        end yop
```

Suppose that x.b is initially true. We would like to prove that the number of executions of xop is equal to or one greater than the number of executions of yop. This specification could be written as:

$$(\#X\text{-exit} = \#Y\text{-exit}) \,|\, (\#X\text{-exit} = \#Y\text{-exit} + 1)$$

Informally, suppose that the above specification is not satisfied, and that it is due to $\#X\text{-exit} > \#Y\text{-exit} + 1$. Then there must be two events X1-exit < X2-exit that occur without an intervening Y-exit. Note that the x.b is set to false after X1.leave, and remains false until after some Y-leave. If no such Y-leave event occurs, then the guarantee remains false, and X2-dequeue cannot occur. Therefore, there can be no such events. To prove that #Y-exit cannot exceed #X-exit, we note that the only way that #Y-exit could exceed #X-exit is for the initial exit event to be some Y-exit. However, we assumed that the variable x.b was initially true, which prohibits

Y-dequeue from occurring.

The addition of boolean variables provides additional information about the past execution of operations. As the above informal proof shows, the semantic model can capture this information as well. Extending the verification rules to handle such situations is left as a topic for future research.

8.1.2 Conditionals

The addition of boolean variables and expressions is of limited usefulness if the only test of a boolean expression remains limited to the guarantee on a queue. Another extension that can be added at this point is conditional statements, with the form

```
if expression
  then body_of_statements
  else body_of_statements
  end
```

The **else** part is optional. In the semantic model we need to introduce a new kind of node, the **if** node. The **if** node tests the results of the boolean expression (we will discuss a more general model for evaluation below), and conditionally executes the appropriate body of statements based on the result. The next node after the last node of either the **then** body or the **else** body is the node that corresponds to the statement directly following the **if** statement. By the introduction of conditionals, the "node graph" has become a true directed graph.

Although the modelling of conditionals poses no severe difficulties, the addition of conditionals complicates the specification language. Consider the following operation (we have also relaxed our requirement for a strict correspondence between serializer and resource operations):

```
xct = proc (x: cvt, d: data)
    enqueue x.q until crowd$empty(x.c)
    if data$cond(d)
       then join x.c
                 resource$fast_xct(x.res, d)
                 end
       else join x.c
                 resource$slow_xct(x.res, d)
                 end
       end
    end xct
```

What event does X-join denote? There are potentially two different events, and the event to occur depends on the data presented to the operation.

The solution we recommend is simple: for every test in a conditional statement, assume that the test evaluates to a particular boolean value (true or false). If the specification clause can be verified for every permutation of the conditional tests, then it is verified for the operation. In the above example, we would effectively need to verify two operations: one where data$cond(d) was true immediately after the enter event, and one where data$cond(d) was false.

8.1.3 Loops in serializer operations

Just as conditional statements introduce ambiguity about which nodes can be executed, iteration and recursion introduce ambiguity about how often a node is executed. The doubt is significantly worse, however, since the number of possible executions of a loop is not bounded.

When a point in a serializer operation can be passed many times during the execution of a transaction, an event is not just an execution of a node for that transaction, but a particular execution of that node. We can adapt the method of handling conditionals to handling loops by assuming particular numbers of iterations for each loop. If the specifications can be shown to hold for any choice of such numbers, then the specifications are verified for the operation as a whole, provided that all of the loops terminate. Induction can be used by assuming that the specification holds for some particular number N of executions around a loop, then showing that the specification holds for N+1 executions (plus a basis proof for N = 0). In order to prove service specifications, an additional proof that each loop terminated would be necessary.

8.1.4 Arbitrary expressions and invocations

The introduction of arbitrary expressions into serializers has the following effects:

- The semantic model must include arbitrary types and values of those types, *including* user-defined types.

- The semantic model must be provided with events to mark both the start and the end of an invocation.

- The specification language must be merged with a larger specification language. Values must be named and functions on those values defined. Concurrency specifications, data abstraction specifications, and procedural specifications may be mutually interdependent.

- The serializer verification system must be joined to a more general verification system. While it is our hope that the two kinds of verification systems can be kept modular, we have no evidence at this time to support this hope.

With arbitrary expressions and invocations, some of the verification techniques we have described may be invalid for some situations, some of which are:

- Some invocations may not always terminate. If we use such invocations, then we must be prepared to prove service where applicable. If we cannot prove service, then we are faced with a new potential source of lack of service: indefinite possession of the serializer object. In terms of our current model we would be faced with a finite *complete* history (since it would be possible for no further serializer events to occur) where a transaction would be in possession at the end of the history. Since many of our verification rules depend on *no* transaction being in possession at the end of a finite complete history, and *no* crowds being occupied, our techniques are not applicable where termination cannot be proved. The problems of combining our techniques with proofs of termination for invocations remain for future research.

- If we allow side effects in the evaluation of guarantees evaluation it becomes necessary to introduce events to model the beginning of such evaluation, and to indicate the order in which guarantee evaluation is performed.

- Recursive operations provide one more problem. When we assume that an invocation used by a serializer terminates, and thereby prove service for the serializer operation, such a proof must not be circular. If the invocation termination depends on the service proof, then the service proof is not valid unless one can prove that the level of recursion is bounded.

All of the above issues are left for further research.

8.1.5 Priority queues

The monitor construct presented in [Hoare 74] permits the use of *priority queues*, which obey a "first in, best out" discipline. A serializer example that makes use of priority queues is presented in Appendix III.

In using priority queues, we do not (usually) wish to allow the addition of requests to a queue to indefinitely postpone the progress of earlier requests. For the disk serializer we can prove that the *request* operation guarantees service since, when we are serving one queue, its size decreases with every fulfilled request, and we assume that the resource operation terminates. Therefore, the queue being served must empty, the direction must change, and the other queue becomes the served queue. Another proof of service can be based on never adding requests to a queue at a priority number less than or equal to the lowest number request in the queue.[21] We can still prove service even if we allow a bounded number of requests to be added at a lower or equal number priority.

8.2 Closing remarks

This thesis has presented a wide range of aspects of a single language construct, including programming language design, formal specifications for programming languages, and verification techniques. We were able to cope with such a wide range because we were interested in limited techniques for a limited construct, and our design philosophy emphasizes minimal interference between constructs. We believe that our results show that such an approach has merit.

In several places we have mentioned that it is possible to view serializer operations either as procedures or as message handlers. This flexibility is made possible through the design of the serializer construct, and through the use of a semantic model that is limited to describing serializers. Even though details may change as serializers are embedded in a procedure-oriented or a message-passing language, the basic approach to proving serializers should remain sound.

[21] This is the approach that Hoare takes in [Hoare 74].

We have only attempted to verify automatically a number of variants of the readers-writers problem. Partially due to this limitation we have been able to handle several important specifications regarding concurrency control. Even though the specification categories have been chosen for use with access to resources, properties such as exclusion, priority, and termination are generally recognized as important in dealing with concurrent programs.

We have demonstrated the feasibility of proving a form of termination that is applicable to transactions, rather than programs or objects. This technique is especially useful when resources (or objects in general) have unbounded lifetimes and the number of active transactions (or processes) is unbounded.

Our approach to verification has not been oriented toward presenting either a minimal or a complete set of axioms and inference rules. Rather, we have identified some higher-level theorems, expressed as inference rules, that are useful in proving serializers, and have justified these theorems by direct appeal to the semantic model. Should further examples identify other useful theorems, more justification through the model is called for. While the study of the completeness of an axiom system is intersting in its own right, it is rare for a verifier (either automatic or manual) to appeal to the axioms if more general and more powerful theorems are known. The test we value most for such a selection of theorems is their utility in verification, a test that our theorems have passed.

Bibliography

Atkinson 76

R. Atkinson, Optimization Techniques for a Structured Programming Language, S.M thesis, Massachusetts Institute of Technology, May 1976.

Baker 78

H. Baker, Actor Systems for Real-Time Computation, M.I.T. Laboratory for Computer Science TR 197 (Ph.D. thesis), March 1978.

Bloom 79

T. Bloom, Synchronization Mechanisms for Modular Programming Languages, S.M. Thesis, Massachusetts Institute of Technology, January 1979.

Boyer and Moore 75

R. Boyer, J. Moore, Proving Theorems About LISP Programs, JACM, vol. 22, January 1974, 129-144.

Campbell and Habermann 74

R. Campbell, A. Habermann, The Specification of Process Synchronization by Path Expressions, Lecture Notes in Computer Science 16, Springer-Verlag 1974, 89-102.

Courtois, Heymans and Parnas 71

P. Courtois, F. Heymans, D. Parnas, Concurrent Control with Readers and Writers, CACM 14, 10, October 1971, 667-668.

Dahl 72

O. Dahl, Hierarchical Program Structures, Structured Programming, Academic Press, New York, 1972.

Deutsch 73

P. Deutsch, An Interactive Program Verifier, Ph.D. thesis, University of California at Berkely, Berkeley CA, 1973.

Dijkstra 68

E. Dijkstra, Cooperating Sequential Processes, Programming Languages, Academic Press, New York, 1968.

Dijkstra 71

E. Dijkstra, Hierarchical Ordering of Sequential Processes, Acta Informatica, vol. 1, 1971, 115-138.

Dijkstra 75

E. Dijkstra, Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. CACM 18, 8, August 1975, 453-457.

Eswaren et. al. 76

K. Eswaren, J. Gray, R. Lorie, I. Traiger, The Notion of Consistency and Predicate Locks in a Database System, CACM 19, 11, November 1976, 624-633.

Feldman 79

J. Feldman, High Level Programming for Distributed Computing, CACM 22, 6, June 1979, 353-367.

Good, London and Bledsoe 75

D. Good, R. London, W. Bledsoe, An Interactive Program Verification System, Proceedings of the International Conference on Reliable Software, Los Angeles CA, April 1975, 482-492.

Good, Cohen and Keeton-Williams 79

D. Good, R. Cohen, J. Keeton-Williams, Principles of Proving Concurrent Programs in Gypsy, Sixth ACM Symposium on Principles of Programming Languages, San Antonio, January 1979, 42-52.

Greif and Hewitt 75

I. Greif, C. Hewitt, Actor Semantics of PLANNER-73, Proceedings of ACM SIGPLAN-SIGACT Conference, Palo Alto CA, January 1975.

Greif 75

I. Greif, Semantics of Communicating Parallel Processes, M.I.T. Laboratory for Computer Science TR 154 (Ph.D. thesis), September 1975.

Guttag, Horowitz and Musser 78

J. Guttag, E. Horowitz, D. Musser, Abstract Data Types and Software Validation, CACM 21, 12, December 1978, 1048-1064.

Brinch Hansen 72

P. Brinch Hansen, Structured multiprogramming, CACM 15, 7, July 1972, 574-577.

Brinch Hansen 78

P. Brinch Hansen, Distributed Processes: A Concurrent Programming Concept, CACM 21, 11, November 1978, 934-941.

Hewitt and Atkinson 77

C. Hewitt, R. Atkinson, Synchronization in Actor Systems, Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, January 1977, 267-280.

Hewitt and Baker 77

C. Hewitt, H. Baker, Actors and Continuous Functionals, M.I.T. Laboratory for Computer Science TR 194, December 1977.

Hewitt, Attardi, and Lieberman 79

C. Hewitt, G. Attardi, H. Lieberman, Specifying and Proving Properties of Guardians for Distributed Systems, A. I. Memo 505, M.I.T. Artificial Intelligence Laboratory, June 1979.

Hewitt and Atkinson 79

C. Hewitt, R. Atkinson, Specification and Proof Techniques for Serializers, IEEE Transactions on Software Engineering, January 1979, 10-23.

Hoare 74

C. Hoare, Monitors: An Operating System Structuring Concept, CACM 17, 10, October 1974, 549-557.

Hoare 78

C. Hoare, Communicating Sequential Processes, CACM 21, 8, August 1978, 666-677.

Howard 76

J. Howard, Proving Monitors, CACM 19, 5, May 1976.

Ingalls 78

D. Ingalls, The Smalltalk-76 Programming System Design and Implementation, Fifth ACM Symposium on Principles of Programming Languages, Tucson, January 1978, 9-15.

Igurashi, London, and Luckham 72

S. Igurashi, R. London, D. Luckham, Automatic Program Verification, AIM-200, Stanford Artificial Intelligence Project, Stanford University, Stanford CA, 1974.

Lamport 80

L. Lamport, "Sometime" is sometimes "not never" - On the temporal logic of programs, Seventh ACM Symposium on Principles of Programming Languages, Las Vegas, January 1980, 174-185.

Lampson and Redell 79

B. Lampson, D. Redell, Experience with monitors and processes in Mesa, CACM 22, 2, February 1980.

Laventhal 78

M. Laventhal, Synthesis of Synchronization Code for Data Abstractions, M.I.T. Laboratory for Computer Science TR 203 (Ph.D. thesis), June 1978.

Liskov et. al. 77

B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, Abstraction Mechanisms in CLU, CACM 20, 8, August 1977, 564-576.

Liskov 79

B. Liskov, Primitives for Distributed Computing, Computation Structures Group Memo 175, Massachusetts Institute of Technology Laboratory for Computer Science, May 1979.

Liskov 79a

B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Scheifler, A. Snyder, CLU Reference Manual, M.I.T. Laboratory for Computer Science TR 225, October 1979.

Metcalfe and Boggs 76

R. Metcalfe, D Boggs, Ethernet: Distributed Packet Switching for Local Computer Networks, CACM 19, 7, July 1976, 395-404.

Morris 74

J. Morris, Towards More Flexible Systems, Lecture Notes in Computer Science 19, Springer-Verlag, 377-383, 1974.

Owicki 75

S. Owicki, Axiomatic Proof Techniques for Parallel Programs, Ph. D. thesis, Department of Computer Science, Cornell University, Cornell NY, July 1975.

Owicki and Gries 76

S. Owicki, D. Gries, Verifying Properties of Parallel Programs: An Axiomatic Approach, CACM 19, 5, May 1976, 279-285.

Reed 78

D. Reed, Naming and Synchronization in a Decentralized Computer System, M.I.T. Laboratory for Computer Science TR 205 (Ph.D. thesis), September 1978.

Scott and Strachey 71

D. Scott and C. Strachey, Toward a Mathematical Semantics for Computer Languages, Proceedings of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, 1971.

Strachey and Wadsworth 74

C. Strachey and C. Wadsworth, Continuations - A Mathematical Semantics for Handling Full Jumps, Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

Scheifler 77

R. Scheifler, An Analysis of Inline Substitution for a Structured Programming Language, CACM 20, 9, September 1977, 647-654.

Suzuki 74

N. Suzuki, Verification of Programs by Algebraic and Logical Reduction, AIM-255, Stanford Artificial Intelligence Project, Stanford University, Stanford CA, 1974.

Svobodova, Liskov and Clark 79

L. Svobodova, B. Liskov, D. Clark, Distributed Computer Systems: Structure and Semantics, M.I.T. Laboratory for Computer Science TR 215, March 1979.

Waldinger and Levitt 74

R. Waldinger, K. Levitt, Reasoning About Programs, Artificial Intelligence 5,3, Fall 1974, 235-316.

Wegbreit and Spitzen 76

B. Wegbreit, J. Spitzen, Proving properties of complex structures, JACM 23, 2, April 1976, 389-396.

Wulf 78

W. Wulf, et. al., An Informal Definition of Alphard (preliminary), Carnegie-Mellon University, Computer Science Department, Report CMU-CS-78-105, Pittsburgh PA, February 1978.

Yonezawa 77

A. Yonezawa, Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics, M.I.T. Laboratory for Computer Science TR 203 (Ph.D. thesis), December 1977.

Appendix I - Bounded buffer serializer

A bounded buffer is intended to smooth variations in processing speed between a producer and a consumer of items of information, and thereby afford more concurrency between the two processes.[22] A bounded buffer is accessed by *get* and *put* operations, where the Nth get operation retrieves the information that the Nth put operation deposited. A bounded buffer object is constructed by calling the create operation with a positive number specifying the number of items of information to buffer. The buffered information is transferred by copying the contents (via *item$move*) from one item to another. We assume that this copying takes some significant amount of time.[23] Partial specifications for this problem appear in Chapter 4.

The *bounded_buffer* serializer given below uses only slight extensions over serializers. We assume that performing a put operation on a full buffer causes an exception to be signalled for the data abstraction (called *bbuf* in this example), but that the serializer operation simply pauses until the buffer is not full. If several processes perform get operations, there is no overlap between the operations, since a modification to the buffer is made in the data abstraction, and the modifications made by two invocations could conflict. A similar conflict arises for put operations.

[22] A solution to this problem using monitors appears in [Hoare 74]. A verification of a similar monitor appears in [Howard 76].

[23] Although such copying is normally foreign to CLU, we have used copying in an attempt to remain comparable to the monitor statement of the problem.

The *combined_bounded_buffer* serializer shown in Appendix II combines the function of the bounded_buffer serializer and the bbuf cluster. The interface remains the same, but the implementation does not use the bbuf cluster. Besides the obvious savings afforded by the elimination of operation calls from the serializer to the cluster, there is additional concurrency possible because get operations are allowed to overlap with other get operations, and put operations are allowed to overlap with other put operations.

We have presented this problem as an illustration of how the modularity provided by serializers allows such optimization without changing the interface that the user sees. Further, any verification of programs that *use* the bounded buffer serializer remain valid, provided that they are unaffected by the additional concurrency.

```
% The bounded_buffer serializer protects the bbuf abstraction
% against damaging concurrent access. Get and Put operations
% may only overlap with get_size operations. All copying of
% item to item is done in the bbuf cluster.

bounded_buffer = serializer is
        create, get_size, get, put

    rep = struct[res: bbuf, c; crowd, max: int,
                 gq, pq: queue]

    create = proc (n: int) returns (cvt) signals (bad_size)
        return (rep${res: bbuf$create(n),
                     max: n,
                     c: crowd$create(),
                     gq, pq: queue$create()})
           resignal bad_size
        end create

    get_size = proc (x: cvt) returns (int)
        return (x.res.size)
        end get_size

    get = proc (x: cvt, dst: item)
        enqueue x.gq until crowd$empty(x.c) & x.res.size > 0
        join x.c
             bbuf$get(x.res, dst)
             end
        end get

    put = proc (x: cvt, src: item)
        enqueue x.pq until crowd$empty(x.c) & x.res.size <= x.max
        join x.c
             bbuf$put(x.res, src)
             end
        end put

    end bounded_buffer
```

Appendix II - Combined bounded buffer serializer

```
% The combined bounded buffer permits get operations to overlap with
% other get operations, and put operations to overlap with other put
% operations, but get and put operations cannot overlap.  Get_size
% operations can overlap with either get or put operations.

combined_bounded_buffer = serializer is
    create, get_size, get, put

    buf = array[item]
    rep = struct[res: buf, gc,pc: crowd,
                 next, size, max: int,
                 sq, gq, pq: queue]

    create = proc (n: int) returns (cvt) signals (bad_size)
        if n < 1 then signal bad_size end
        return (rep${res: buf$fill_copy(0, n, item$create()),
                     next: 1, size: 0, max: n,
                     gc, pc: crowd$create(),
                     gq, pq, sq: queue$create()})
        end create

    get_size = proc (x: cvt) returns (int)
        return (x.size)
        end get_size

    get = proc (x: cvt, dst: item)
        enqueue x.gq until x.size > 0 & crowd$empty(x.pc)
        src: item := x.res[x.next]
        x.size := x.size - 1
        x.next := (x.next+1) // x.max   % take increment mod N
        join x.gc
            item$move(dst, src)    % copy data from src to dst
            end
        end get

    put = proc (x: cvt, src: item)
        enqueue x.pq until crowd$empty(x.gc) & x.size <= x.max
        dst: item := x.res[(x.next+x.size) // x.max]
        x.size := x.size + 1
        join x.pc
            item$move(dst, src)
            end
        end put

    end combined_bounded_buffer
```

Appendix III - Disk head scheduler

In [Hoare 74], the disk head scheduler problem is discussed for monitors. Below we give a serializer solution to the problem, which uses the *priority_queue* type. A priority_queue is a queue where the order of dequeue events is dependent on the priority. We will assume that the lowest numerical value of the priority is served before any others. Equal priorities are served FIFO.

The algorithm used depends on having two queues, one which is served in increasing order of disk address, called x.up_q; and one which is served in decreasing order of disk address, called x.down_q. Our algorithm works by adding requests to one queue, and serving the other. We change direction whenever the queue for the current direction is empty and the other queue is not empty.

```
disk = serializer is
        create,
        request

    rep = record[increasing: bool,
                 up_q, down_q: priority_queue,
                 disk: _disk]

    create = proc () returns (cvt)
        return (rep${increasing: true,
                     up_q, down_q: priority_queue$create(),
                     disk: _disk$create()})
        end create

    request = proc (d: cvt, address: int, kind: int, p: page)
                signals (bad_address, disk_error)

        if d.increasing
           then enqueue d.down_q
                    until crowd$empty(d.c) &
                          (~d.increasing |
                           priority_queue$empty(d.up_q))
                    priority address
                d.increasing := false
           else enqueue d.up_q
                    until crowd$empty(d.c) &
                          (d.increasing |
                           priority_queue$empty(d.down_q))
                    priority -address
                d.increasing := true
           end

        join d.c
             _disk$request(d.disk, address, kind, p)
             end resignal bad_address, disk_error

        end request

    end disk
```

Appendix IV - Table of definitions

Biographical Note

Russell Roger Atkinson

a: was born in New York City in 1950.

b: was raised in Fanwood, NJ.

c: received a B.S. degree in Physics from the University of Illinois in 1972.

d: received a S.M. degree in Computer Science from M.I.T. in 1976, thesis title: *Optimization Techniques for a Structured Programming Language.*

e: received a Ph.D. degree in Computer Science from M.I.T. in 1980, thesis title: *Automatic Verification of Serializers.*

f: will be working at Xerox Palo Alto Research Center.

g: is interested in programming languages, optimization, verification, computer architecture, and distributed systems.

h: all of the above.

Answer: h.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
MIT/LCS/TR-229

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Automatic Verification of Serializers

5. TYPE OF REPORT & PERIOD COVERED
Ph.D. Thesis, March 1980

6. PERFORMING ORG. REPORT NUMBER
MIT/LCS/TR-229

7. AUTHOR(s)
Russell Roger Atkinson

8. CONTRACT OR GRANT NUMBER(s)
N00014-75-C-0661
MCS 74-21892 A01

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
ARPA/Department of Defense / NSF/Associate Program
1400 Wilson Boulevard / Director/Office Comp.
Arlington, VA 22209 / Activities
/Washington, D. C.20550

12. REPORT DATE
March 1980

13. NUMBER OF PAGES
205

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
ONR/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)

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

verification
concurrency
monitors
serializers
specification

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

This thesis is concerned with the problem of controlling concurrent access to shared data. A language construct is proposed to enforce such control; a specification language is defined to describe the formal requirements of such control; and verification techniques are given to prove that instances of the construct satisfy their specifications. The techniques are justified in terms of the definition of the construct and the definition of the specification language. Results are given for a program that implements a number of the techniques, illustrated by verifying several versions of the readers-writers