CHAPTER 1
Motivations and Goals

Over the past decade, the paradigm of "object-oriented programming" has become increasingly popular with software engineers and computer science researchers. This popularity stems from the belief that object-oriented programming will lead to shorter program development times, less program maintenance and greater code reuse than do other software engineering approaches. Yet despite its promise, the relative youth of the paradigm leaves open the question of merit; there is too little data available to determine its strengths and weaknesses. One goal of this thesis is to shed light on the usefulness of object-oriented programming; this is done by evaluating the paradigm's ability to solve a well-known software problem, that of *concurrency control*. Concurrency control has particular relevance for object-oriented systems, leading to the second, primary, goal of this thesis: the development of principles for realizing object-oriented concurrency control schemes that fully use and support the paradigm.

Section 1.1 discusses *algorithmic decomposition*, a commonly-used design method, and reveals that it may be partly responsible for common software engineering problems. The features of the object-oriented programming paradigm are discussed in section 1.2; section 1.3 describes the problem of concurrency control and its relevance to the paradigm. Section 1.4 outlines the remaining thesis chapters.

1.1. Algorithmic Decomposition and its Problems

Humans have a limited ability to deal with complexity (Miller, 1956), yet many software engineering projects are very complex. There is general consensus that the only way to approach such projects is by dividing them into units small enough to be easily understood (Booch, 1990; Meyer, 1988; Stroustrup, 1991). *Algorithmic decomposition* (e.g., Wirth, 1971) adopts such a strategy. The algorithmic decomposition process starts by considering the functionality of the program to be developed, but only in terms of general requirements. These requirements are then broken into specific program functions, and each function is further decomposed into smaller units. This process continues until the units of functionality are simple enough to be directly implemented in code. Consider, for example, a program that reads an "import file" of data, written in a specified text format, and stores it in a database by converting the data to the form that the database expects. The general steps in the algorithm are:

(1) Get the next line of data from the import file.
(2) Convert it to the form expected by the database.
(3) Write it to the database.
(4) Repeat the above steps until there is no more data in the import file.

The three major steps above would each be decomposed into smaller and smaller steps. For example, the conversion step might involve doing error checks on the line of import data, converting the data to the units used by the database, and putting each converted data item into the appropriate field of a database record (Figure 1.1). These sub-functions would each in turn be subjected to the decomposition process. Algorithmic decomposition thus eases the complexity of software development by allowing the designer to concentrate on only one component of the system at a time, and by giving him or her the ability to further subdivide that component into more easily understood units. The paradigm also encourages the use of project teams, with

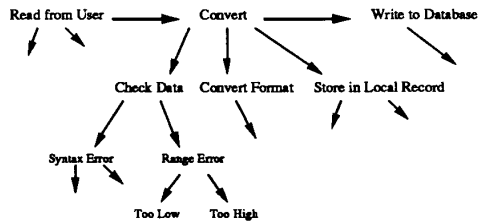


FIGURE 1.1. Storing User Data in a Database using Algorithmic Decomposition

each person concentrating on a single system component. Such a strategy can greatly decrease the time needed to create a finished program.

However, programs resulting from algorithmic decomposition, while perhaps designed and coded relatively quickly, may not fare as well in other parts of the software life cycle (Meyer, 1988). Although the lowest-level routines would seem to be prime candidates for reuse in later programs, in practice this rarely happens. Adding extra functionality to a "finished" program, as is bound to happen sooner or later, can be a lengthy, frustrating process. Meyer notes that these problems arise from two major sources: the rigid temporal ordering of functions, and the "knowledge" of data formats these functions possess.

The solution to a design or programming problem given by algorithmic decomposition is inherently temporal. In the example of the database import routine, the steps were specified in the order in which they would occur. This yields an unnecessarily brittle program: if a new major function is added later, the whole existing structure may have to be changed. As well, most of functions and sub-functions depend, directly or indirectly, on the format of the data with which they work. If this changes,

these functions will have to be modified. This is a major reason why functions are often not reused from one project to the next: although each project may require the same algorithms, these algorithms may operate on different types of data, rendering the code used in one project unsuitable for the next.

In light of these problems, there has been an increasing tendency to regard the *data* of a particular problem as being more important than the functions that manipulate it. The result of this view is the paradigm of object-oriented programming, discussed in the next section.

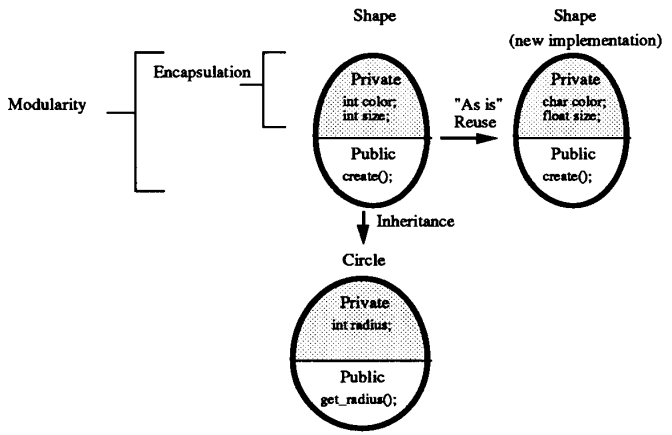


FIGURE 1.2. Object-Oriented Programming Concepts

1.2. Object-Oriented Programming

Object-oriented programming encourages hierarchical structuring, as does algorithmic decomposition, giving people the ability to deal with complex ideas and problems. However, the components of the structure are not the functions of the program, but the data. Object-oriented programs are composed of *objects*, with each object having an internal state and a set of well-defined behaviors. The variables and functions that implement the state and behaviors of an object are called *members* of the object, with all objects of a particular *class* having the same types of members. (In some object-oriented languages, objects are equivalent to classes; in others, a class exists at compile-time, defining what a set of objects is, while objects themselves are instantiations of classes existing only when the program runs.) The object-oriented paradigm allows several desirable design strategies, including modularity, encapsulation, natural units of design, reusability and inheritance (Figure 1.2).

1.2.1. Modularity. An object can be seen as an instance of an abstract data type, containing all the variables and functions needed to represent the behavior of that data type. For example, a *circle* object may have a function to return the radius of the circle. This sort of *modularity* encourages a logical organization of program components, because related functions and variables are grouped together in the object whose behavior they implement.

1.2.2. Encapsulation. Most object-oriented languages divide the functions and variables contained within an object class into at least two parts: private and public. *Public* members are those which other objects may use — they represent the object's interface to the rest of the program. Normally, the interface consists only of functions (commonly called *methods*). In the circle example above, the radius function would be public. *Private* members are used by the object to manipulate its internal state; though such manipulation may result from a call to one of the object's public methods, private members are not directly accessible to objects of other classes,

meaning that there can be a clear separation of the object's external behavior and its implementation. Such a separation allows the internal details of the object to be *encapsulated* — a client of the object's public members need not be aware of how they or the rest of the object is implemented. This allows such implementations to be changed without affecting how clients deal with the object, reducing the complexity with which the designer/programmer is faced. Equally important, it frees clients of the object from having to take into account the format of the object; as discussed above, changes in data format are common, and are a major reason why programs developed through algorithmic decomposition are hard to maintain.

1.2.3. "As Is" Reusability. Building systems out of objects leads directly to the notion of reusing the same object types in different systems. More specifically, libraries of object classes can be created to help the design and implementation of different pieces of software. Reuse of classes seems more plausible than reuse of simple functions because, as discussed previously, an object can hide the details of its implementation and thus save users of its public methods from the need to change when the implementation changes.

1.2.4. Inheritance. A powerful feature of object-oriented programming is that one object class may *inherit* properties from one or more other classes; such a class is called a *subclass* of the classes from which it inherits. For example, a *colored_point* class may inherit most of its members from class *point*. Functional *polymorphism* is often associated with inheritance — a subclass may specialize an inherited method to suit its own needs, yet the customized method can be invoked safely by clients of the superclass. In the example above, *colored_point* may redefine *point*'s "draw" method, but clients of *point* will be able to invoke a *colored_point* "draw" method without causing an error. Thus, inheritance actually serves two purposes — code can be shared between a class and its subclass, and the client-visible behavior of a class can be further specialized by a subclass, encouraging a hierarchic view of the problem

domain. These two roles of an inheritance mechanism, *code reuse* and *subtyping*, are not always compatible — the differences are discussed further in chapter 5.

1.2.5. Natural Design Units. Proponents of object-oriented programming believe that structuring programs as hierarchies of objects is more natural than as hierarchies of functions, because the former is how people tend to view objects in the world. For example, a person may think of a computer as consisting of "screen," "keyboard," and "hardware," with each component being further decomposed into its constituent parts. It is considerably less natural to view a computer from the perspective of the temporal order of its functionality (e.g., power switch is turned on, CPU fetches first instruction from memory, etc.). By corresponding to how humans view the world, object-oriented decomposition can mitigate the abstractness of program design. Because the interaction of two or more objects is not inherently temporal, it is relatively easy to add a new object to a program — such an addition will not compromise the whole program structure as it might if algorithmic decomposition were used.

These features of object-oriented programming are compelling, and the paradigm has many proponents (Booch, 1990; Coad & Yourdon, 1991; Meyer, 1988; Stroustrup, 1991). Despite its popularity, there is little evidence that it is superior to algorithmic decomposition — experience reports indicate mixed results (e.g., Barry, Altoft, Thomas, & Wilson, 1987; Harrison, Sweeney, & Shilling, 1989; Kiczales & Lamping, 1992), and many supporters of the paradigm offer differing strategies for its use (e.g., compare Meyer and Stroustrup). One goal of this thesis is to gain insight into the paradigm's performance. To this end, this thesis examines the effect object-oriented programming has had on a specific domain, *concurrency control*. Concurrency control is a well-known problem arising in many software systems, be they object-oriented or not, and has particular relevance for object-oriented programs. This relevance, and the nature of the problem, are discussed below.

1.3. Concurrency Control

Concurrency control is necessary in any system in which more than one thread of control is possible. Examples of these include the UNIX operating system, in which one or more processors split their time between an indeterminate number of *processes* (active programs), and *distributed systems*, in which many processes run on many processors, exchanging messages and often sharing data. The concurrent access of processes to shared resources has to be controlled, because in many cases access can only be reasonably granted to one process at a time. For example, if two processes try to add a new element to a linked list at the same time, disaster can result (see Figure 1.3). Thus, access to such resources must be mutually exclusive. As well, interprocess communication has to be synchronized: if process A is in a state where it can't accept a message from process B, process B must *block* (become suspended) until A is ready.

1.3.1. Standard Strategies.

Semaphores and Monitors. Semaphores and monitors are two of the most common concurrency control mechanisms. A *semaphore* (Dijkstra, 1965) is a construct with two operations, P and V, which can be used to control access to a resource shared among processes. When a process is about to enter a *critical section* of code (i.e., one where it manipulates a shared resource), it performs the semaphore's P operation. If the semaphore has not already been acquired by another process, this operation will grant it to the invoking process. Otherwise, the process blocks, and is put in a queue awaiting its turn at the semaphore. When the process holding the semaphore exits its critical section, it performs the V operation, freeing the semaphore for use by another process.

The semaphore approach requires that synchronization operations be coded as part of the program. In an attempt to make synchronization more transparent at the application level, Hoare (1974) introduced the *monitor*, a construct that controls

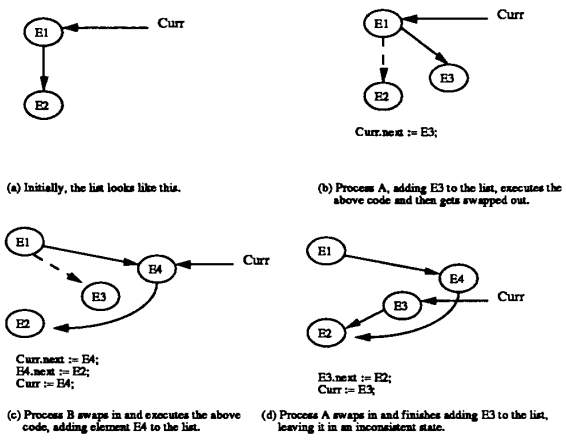


FIGURE 1.3. Uncontrolled Concurrent Access

access to a shared resource. Only one process can be active in a monitor at any time, guaranteeing mutual exclusion. Other processes wishing to access the resource are queued, waiting on *conditions* of the resource state (e.g., "buffer not full"). Such conditions determine what resource operations are accessible given the resource's current state. A process may thus also be queued if, when given access to the resource, conditions relevant to the operation it requested are not met. When a process is given access to the monitor and the appropriate conditions hold, the operation requested by the process executes. When the operation finishes, the process *signals* a condition, representing the new state of the resource (e.g., "buffer empty"). At this point, exactly one process that was waiting on the signalled condition will be immediately

restarted, and the cycle continues.

Transactions. As discussed by Chin and Chanson (1991), *transactions* are used to maintain consistency in situations where program actions may affect the state of more than one resource. Actions are considered transactions if their behavior has the following properties:

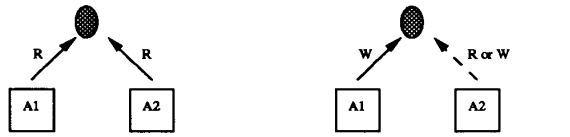
(1) Serializability. The interleaving of concurrent accesses to a resource is *serializable* if it is equivalent to some serial (sequential) access order. This effectively deals with the "critical section" problem discussed above, since a sequential access order guarantees that only one action at a time will be modifying the resource.

(2) Failure Atomicity. If for some reason an action is unable to complete all its operations, any partial results are completely undone. Such *failure atomicity* prevents the resources in the system from being left in an indeterminate state.

(3) Permanence of Effect. If an action successfully completes all its constituent operations, all changes it has made to resources become permanent when it *commits*. A committed action can be considered terminated.

Locks are often used to guarantee serializability. *Read* locks can be held on a resource by many processes at once, since reading a resource's state is assumed not to change that state. *Write* locks, however, can only be held by one process at a time, since such a process is assumed to modify the resource. This, the *readers/writers problem* (Courtois, Heymans, & Parnas, 1971), is illustrated in Figure 1.4.

1.3.2. Relevance to Object-Oriented Programming. The above discussion shows that concurrency control is an important, complex problem, well-suited to test the purported strengths of the object-oriented paradigm. It is also especially relevant to systems using the paradigm. Most objects in the real world interact concurrently, not linearly: an object-oriented system attempting to use objects as representations of real-world entities must take this concurrency into account, regardless of whether



(a) Any number of processes can hold Read locks on an object.

(b) If Process A1 has a Write lock on the object, A2 has to wait until the lock is is released to get its own lock, be it Read or Write.

FIGURE 1.4. Acquiring Read and Write Locks

system objects are *passive* (manipulated by exterior processes) or *active* (possessing a thread of control). Moreover, *message-passing*, normally used to invoke an object method and return the results of the method to the calling object, conforms well with communication in many distributed systems. Other distributed communication strategies, such as the remote procedure call protocol used by Aschmann, Giger, Hoepli, Janak, and Kirrmann (1991), can be simulated by message-passing. This leads to the notion of objects as active entities in the distributed system, perhaps in *client-server* relationships (e.g., Apple Computer, 1991). A server object, one that accepts and processes request messages from client objects, clearly requires some form of concurrency control, because requests may continue to arrive while the current request is being processed. Therefore, development of powerful, efficient methods of object concurrency control is a worthwhile goal.

1.4. Thesis Structure

To give a better understanding of object-oriented concurrency control, chapter 2 surveys current approaches to the problem, discusses how well each supports the

object-oriented paradigm and concludes that inheritance is the paradigm feature most difficult to support.

Chapter 3 discusses the importance of evaluation criteria, and applies such criteria to the surveyed approaches to determine their success as concurrency control mechanisms. The results of this evaluation are correlated with each approach's support of the object-oriented paradigm, revealing that good paradigm support is neither necessary nor sufficient for good concurrency control.

The widely varying concurrency control abilities of the surveyed approaches indicate that the term "concurrency control" means different things in different contexts. Chapter 4 explores four distinct types of control problem, shows that each is inseparable from attributes of the controlled object, and discusses why most approaches ignore these dependencies. Chapter 5 reveals that doing so causes problems with inheritance of control, and proposes requirements for disciplined control mechanisms that recognize their dependencies on object attributes. Splitting the concept of inheritance into subtyping and code reuse allows these requirements to be met elegantly. Chapter 6 gives an example of the resulting concurrency control model and discusses issues relevant to its use.

Based on the criteria of chapters 2 and 3 and the requirements laid out in chapter 5, chapter 7 evaluates the proposed scheme and shows that it compares favorably with the surveyed approaches. Chapter 8 discusses the feasibility of verifying the scheme's behavior using a process calculus that allows concurrent systems to be modeled and reasoned about. Chapter 9 reflects on the findings of the earlier chapters, and notes areas of possible future work.

CHAPTER 2

Approaches to Object Concurrency Control

Section 1.3.2 discussed the relevance of concurrency control to the object-oriented paradigm: objects often model concurrent real-world entities, and are also natural candidates for use in distributed systems, which by definition involve concurrency. However, the concurrency requirements of a distributed or parallel-processing system may differ from those of a single-processor, multi-threaded environment, and some applications treat objects as passive data, while others give them their own threads of control. Approaches to object concurrency control reflect these different needs and structures, making a clean taxonomy difficult to realize, though attempts have been made. Section 2.1 outlines the approaches surveyed in terms of how they can be classified; the approaches are described in detail in section 2.2. Section 2.3 discusses how well each approach supports the object-oriented paradigm.

2.1. Approach Overview and Classification

Eleven approaches to object concurrency control are considered in this chapter. Two major criteria were used when choosing approaches to survey:

- *recency*: All approaches dating from the last three years of which the author was aware and able to obtain information about were included.
- *relevancy*: Older approaches, if frequently cited in the literature, were also included.

This section discusses ways of classifying the approaches and gives a brief overview of each.

Based on the relation between an object and its control mechanism, Nierstrasz and Papathomas (1990) divide approaches to object concurrency control into three rough categories:

(1) *orthogonal*, in which concurrency control is separate from the object model. Traditional methods, such as semaphores and monitors, are used to guarantee consistency of object states. The *Trellis/Owl* language, which allows multi-threaded objects and uses a monitor-like construct for concurrency control, is an example of such an approach. Approaches using transactions and object-level locks have also been proposed (Parrington & Shrivastava, 1988; Detlefs, Herlihy, & Wing, 1988), but are not the focus of current research.

(2) *heterogeneous*, supporting both concurrent and sequential objects. Protection for concurrent objects may be provided by transactions or similar methods, or by giving them explicit control over their own concurrency. Such approaches include:
   - *CEiffel*, an Eiffel (Meyer, 1988) extension that uses concurrency annotations to describe the concurrent behavior of objects;
   - *Scheduling Predicates*, an approach using method entry guards to provide sophisticated concurrency control;
   - *Frølund's proposal*, based on the premise that synchronization constraints should become increasingly restrictive in subclasses;

(3) *homogeneous*, in which objects have threads of control, and are responsible for their own concurrency control. Homogeneous languages include:
   - *POOL*, an object-oriented language designed for parallel-processing that allows concurrency at the object level;

- *Hybrid*, an object-oriented language that puts flexible concurrency control within object methods;
- *ACT++*, *Rosette*, *Synchronizing Actions and Conditional Waits*, four similar strategies that achieve inheritable concurrency control by specifying what types of messages are receivable by an object, given its state;
- *DRAGOON*, an ADA-based approach using inheritable "behavior classes" for synchronization.

This classification is convenient because it partitions the spectrum of approaches, allowing easier comparisons and evaluations. However, other classifications are also useful. Kafura and Lee (1989) divide approaches based on where the concurrency control occurs: *centralized* approaches use a single procedure or algorithm for all methods of a given class, while *decentralized* approaches implement concurrency control at the method level. POOL and DRAGOON fall into the first category; the other approaches fall into the second. As well, Trellis/Owl provides control for passive objects (section 1.3.2), while CEiffel, Scheduling Predicates and Rosette support passive or active objects and the remaining approaches support active objects only.

Section 2.3 correlates the classifications into which each approach falls with its support of the object-oriented paradigm.

2.2. Description of Approaches

2.2.1. Orthogonal Approaches.

2.2.1.1. *Trellis/Owl.* The Trellis/Owl language (Moss & Kohler, 1987) allows objects to be multi-threaded by means of its *activity* class. Activity objects are created by a parent object when concurrency is needed, and terminate when their task is finished. Activity objects normally communicate via passive shared objects; concurrency control, using mutual exclusion locks and wait queues, is provided. Trellis/Owl's approach is thus very similar to monitors, but the encapsulation provided by a monitor

is not duplicated — operations such as lock acquisition are done from within the method code of a shared object, not transparently when the object method is requested. Figure 2.1 shows methods for a Trellis/Owl bounded buffer — the *insert* procedure may enqueue an item after receiving a mutual exclusion lock, while the *remove* procedure must also ensure that the buffer is not empty before dequeuing an item (the "me" in the code refers to the buffer object.)

```
operation insert (me, x : Char)
is begin
   lock me.mutex do
      insert (me.buffer, x);
      wakeup (me.nonempty);
   end lock;
end;
```

```
operation remove (me) returns (Char)
is begin
   lock me.mutex do
      if empty (me.buffer) then
         wait (me.nonempty);
      end if;
      return remove (me.buffer);
   end lock;
end;
```

FIGURE 2.1. Methods for a Trellis/Owl Bounded Buffer

2.2.2. Heterogeneous Approaches.

2.2.2.1. *CEiffel.* Löhr (1992), arguing that many concurrency control schemes do not capitalize on the commonalities between sequential and concurrent objects, proposes an approach allowing the same object to have either a sequential or concurrent interpretation, depending on information provided at compile-time. Such information, *concurrency annotations*, is in the form of Eiffel comments interpreted by a preprocessor. Most annotations apply to object methods, and are written after the method name, while others apply to classes or objects. The most important are:

- `--v--`, representing client asynchrony. Clients of methods annotated with `--v--` proceed immediately after making a method invocation, and may claim the result returned by the method, if any, at some later time.
- `-->--`, representing method autonomy. An autonomous method continually issues requests for itself (though only one request can be pending at a time), and is used to model continuous behavior of the object. Autonomous methods take no arguments.
- `--||name--`, signifying that the method is compatible with the method *name*. Compatible methods can be invoked concurrently. An extension of this annotation allows conditional compatibility based on comparisons of method parameter values. The annotation `--||--`, when applied to the head of a class, indicates that all class methods are compatible. When applied to a method name, `--||--` indicates that the method does not alter the object's state. If some but not all class methods are compatible, the class is automatically subject to a concurrency control scheme in which requests for methods are queued if they are incompatible with currently executing methods. If no compatibility annotations are present, each class method executes atomically, and a first-come, first-served priority scheme is used to queue methods.
- `--!--`, indicating that any other annotations are in effect. This annotation can thus be used to switch other annotations on, and is applicable to individual objects or to all objects of a class.

Figure 2.2 shows an example of CEiffel annotations. The *enqueue* method has annotations indicating both client asynchrony and compatibility with the *dequeue* method. The *dequeue* method does not have to specify compatibility with *enqueue*, because the relationship is reflexive. The `--||--` annotation marks the *in_buff* method as compatible with itself and all other class methods. Finally, the `--!--` annotation at the class level means that the other annotations are in effect for all *WeirdBuff*


```
class  WeirdBuff --!--
...
  enqueue (Item i)     is  --v--
                           --||dequeue--
  do ... end ;
  Item dequeue ()      is
  do ... end ;
  Boolean in_buff (Item i) is --||--
  do ... end ;
```

FIGURE 2.2. CEiffel Concurrency Annotations

objects.

Annotated routines can be inherited just as normal, sequential routines are, and thus can be redefined by a subclass. Particularly if multiple inheritance is involved, the concurrency properties of a subclass may not be immediately obvious — by default, routines inherited from different parents are assumed to be compatible, and this may conflict with the (class level) annotations of one or more of the parent classes. Moreover, annotations themselves may be redefined by the subclass. Thus, in terms of concurrency properties, an heir class's behavior may be incompatible with that of its parent, violating subtyping rules (section 1.2.4).

An additional concurrency control feature is based on the standard Eiffel *require* precondition: an expression evaluated to determine if a particular method request can be accepted. For example, *require size > 0* could be the precondition associated with a *dequeue* method. In a sequential program, precondition failure would generate an exception, but concurrent programs might simply delay an offending request until another type of request changes the object state such that the first request is legal. To allow maximum flexibility, the *delay* annotation `--@--` may be used to separate

a precondition into two parts, checker and guard. The optional *checker* checks for fatal errors (e.g., a request parameter having an invalid value); such errors cause an exception. The *guard* is the standard precondition; requests that pass the checker but fail the guard are delayed.

Finally, Löhr makes the important point that asynchrony and autonomy are *specification* properties, while compatibility (method concurrency) is an *implementation* property only. That is, clients of an object method have to know if that method is asynchronous, or if the object has any autonomous methods, but can ignore method concurrency in the object, since such concurrency should not affect how the object's methods behave from the client's point of view. CEiffel can be implemented on systems supporting threaded address spaces.

2.2.2.2. *Scheduling Predicates.* McHale, Walsh, Baker, and Donnelly (1992) propose a concurrency control mechanism using *scheduling predicates* to describe object synchronization state and request priorities. *Synchronization counters*, variables recording the number of current executions of a method, the number of pending executions of a method, etc., form the basis for many of the predicates. Figure 2.3 (a) shows a scheduling predicate for potentially concurrent *put* methods: a *put* request (referred to as *my_req* in the code) may be granted if there are no other *put* methods currently executing that use the same buffer location. The predicate *there_are_no ... executing* is based on the *exec* synchronization counter, which returns a count of the processes executing a particular method. Figure 2.3 (b) uses this counter directly, and conjuncts it with another *there_are_no* predicate to describe synchronization constraints for mutually-exclusive *get* requests that are serviced in FCFS order. (The *arrival* time of a request is logged by the system).

Scheduling predicates thus allow concurrency constraints to be expressed naturally: the names of predicates describe their functions, they can be combined through conjunction (or disjunction), and method parameters can be used in predicates. Inher-


```
put(item,pos) there_are_no [p executing put : p.pos =
                                                my_req.pos]
```

(a) Constraints based on request parameters and concurrent "put" methods.

```
get() exec(get) = 0 and there_are_no [p waiting get :
                        p.arrival < my_req.arrival]
```

(b) Constraints based on mutually exclusive "get" methods and FCFS priority.

FIGURE 2.3. Scheduling Predicates

itance of predicates is not yet supported, however, and would likely be problematic: as discussed in Saleh and Gautron (1992), adding a method to a class may require that synchronization counters used in super and subclasses also be modified. Since synchronization counters refer to specific methods, constraints or predicates built on top them will not take the new method into account. Equally serious, scheduling predicates cannot make use of the object's local state (e.g., "buffer empty"), because, to guarantee correct results, predicates must be evaluated if and when the object is in a consistent state. Though it is easy to determine when the *synchronization* state of the object is consistent, McHale et al. note that the same is not true for *local* state — object instance variables may be modified in ways and at times that are hard to predict.

2.2.2.3. *Frølund's Proposal.* Frølund (1992) proposes a control scheme in which subclasses automatically inherit the constraints of their superclasses. A class method may have *disabled* expressions associated with it, each of the form:

```
(condition) prevents (method name)
```

If the condition is satisfied, the method request cannot be granted. The format of the condition is flexible, and may refer to object state variables, request parameters, local functions and the delay status of other object methods.

When a class is subclassed, the subclass inherits all disabled expressions of the superclass, and may add more to new or inherited methods. However, inherited expressions cannot be modified; thus, concurrency control constraints become more restrictive as the class hierarchy is descended. Frølund argues that this allows easy integration of concurrency control and inheritance, and that such integration is difficult to achieve in approaches using conditional acceptance predicates. However, his proposal is too vague for rigorous analysis and would not support some views of subtyping, as will be seen in chapter 5.

2.2.3. Homogeneous Approaches.

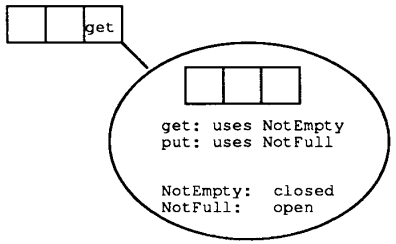
2.2.3.1. *POOL.* POOL (America, 1987; America & van der Linden, 1990) is an object-oriented language designed for parallel-processing applications. Each running object has a local process called a *body*, coded identically for all members of a class, that provides concurrency control by indicating if an object is ready to accept a message, or if it has sent one. For two objects to communicate, the sender must be able to send a message and the receiver must be able to receive it. An object can only recognize incoming messages that request one of the object's methods, and only one such message can be processed at a time.

Code-sharing and subtyping mechanisms are both available in POOL, but not applicable to class bodies. Although an inherited body could be put in parallel with the subclass object's body, for example, parallelism is already present between objects. Another practical use of inheritance, the sharing of body initialization code, would likely grow less important as bodies become more complex and specialized. Finally, America argues that subtyping is poorly-understood in sequential systems

and probably even more complex in concurrent ones; for such reasons, a subtyping structure for class bodies was rejected.

2.2.3.2. *Hybrid.* Hybrid (Nierstrasz, 1987) is an object-oriented programming language in which active objects communicate asynchronously by message-passing simulating the remote procedure call (RPC) paradigm. Though much of what is interesting about Hybrid centers on its run-time environment, the concern of this thesis is its concurrency control mechanism. Each active object normally processes one message at a time; messages recognized by an object are specified in the class definition. *Delay queues* allow an object to store messages that it cannot currently accept. A delay queue is a binary instance variable associated with a method; if *open*, the method request can be accepted, if *closed*, the request has to wait. For example, a pipeline object would have a *NotEmpty* delay queue associated with its *get* operation, and this queue would initially be closed (Figure 2.4). A *put* operation would open the queue.

2.2.3.3. *State-Based Synchronization Schemes.* Four similar object synchronization mechanisms — ACT++, Rosette, Synchronizing Actions and Conditional Waits — are motivated by the interference of concurrency control with inheritance in many approaches (Kafura & Lee, 1989). In POOL, for example, the *centralized* approach to concurrency control would make inheritance cumbersome: the body of an object is responsible for accepting or delaying messages, and would have to be modified to recognize methods declared in a new subclass. Hybrid, though providing *decentralized* concurrency control (each public method has a delay queue associated with it), cannot cleanly cope with subclass delay queues: any superclass method (e.g., a general clean-up routine) requiring access to all delay queues must be modified to recognize queues introduced in subclasses. Similar problems exist in class-based implementations of *actor* systems (see Agha, 1986; Nelson, 1991). In such systems, active objects are *actors*, capable of taking one message at a time out of a *mailbox*



The pipeline object is initially empty, so the NotEmpty delay queue is closed. Since *get* operations are associated with this queue, they are queued up pending the arrival of elements in the pipeline, at which point NotEmpty would be opened.

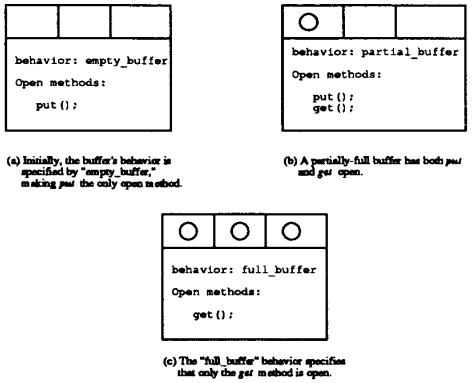
FIGURE 2.4. Hybrid Delay Queues

and processing it. The last step in processing a message specifies a *replacement* actor; the actor itself terminates after processing the message. (The replacement is often very similar to the original actor). For example, a *buffer* class, representing a bounded buffer with methods *put* and *get*, might have three subclasses: *empty_buffer*, *full_buffer* and *partial_buffer*. The first would not have the *get* method, and would be the replacement class specified by the initialization method. The second would not have the *put* method, while the third would have both methods. This is cumbersome, because all three subclasses are similar, and adding a new method to the base class effectively requires a new subclass. Allowing inheritance is even more costly: consider an *extended_buffer* class, identical to *buffer* with the addition of a *get_rear* method, returning the most recently entered item. If it were a subclass of *bounded_buffer*, as it naturally would be, replacement actors *full_buffer* and *partial_buffer* would be unsuitable, because they don't recognize the *get_rear* method. As well, *get* and *put*

methods cannot be inherited from *bounded_buffer*, because they use these (now invalid) replacement actors.

Kafura and Lee solve this problem in their class-based actor language ACT++ by associating an *object manager* with each object to enforce the object's interface. Object methods are either *open* or *closed* depending on whether the current interface can accept requests for their invocations. When a message acceptable to the current interface is received, the object manager invokes the appropriate method by starting a new thread, then closes all methods. Arriving messages are therefore queued, each awaiting the opening of the method it requests. Methods are opened by the *become* operation of the thread, which specifies its replacement behavior and can be executed once in the thread's lifetime. (Concurrency within an object is possible by having the thread execute the *become* operation before it terminates.) This operation specifies which methods the next version of the object's interface accepts.

The ACT++ scheme thus specifies a list of *methods* as the replacement behavior, while the conventional actor approach specifies a new actor. *Behavior names* are used to list the names of replacement methods. For example, the behavior name *partial_buffer* of class *bounded_buffer* would list the *get* and *put* methods, since both operations are legal on a partially-filled buffer (Figure 2.5). Inheritance of concurrency control is elegant because subclasses need only redefine *behavior names* to incorporate new methods. Subclass *extended_buffer*, for example, would redefine *partial_buffer* as *extended_partial_buffer* and add *get_rear* to the list of open methods. The new definition of a behavior name is automatically used by all relevant superclass methods (in this case, *get* and *put*), thus not requiring that they themselves be redefined. Kafura and Lee argue that behavior names facilitate incremental programming, because superclass methods automatically recognize subclass behavior name definitions, and also effectively decouple concurrency control from the sequential behavior of the object.



(a) Initially, the buffer's behavior is specified by "empty_buffer," making *put* the only open method.

(b) A partially-full buffer has both *put* and *get* open.

(c) The "full_buffer" behavior specifies that only the *get* method is open.

FIGURE 2.5. ACT++ Behavior Names

The Rosette system (Tomlinson & Singh, 1989) allows concurrency control to be inherited in parallel, object-oriented programs. The mechanism used, *enabled-sets*, is similar to the behavior names approach; enabled-sets specify which methods may be executed in the new state of the actor, but are implemented as objects rather than redefinable methods, as is the Rosette equivalent of an ACT++ object manager. Private methods are used by a public method to specify the enabled-set that holds at the end of the public method. For example, the *get* method of a bounded_buffer class would invoke a private method *empty* if the last item had been removed from the buffer. The empty method would then enable a set specifying only the public *put* method.

Inheritance of enabled-sets is augmented by a *composition operator* "+" allowing the subclass's inherited set to be a superset of its parent(s)'. For example, the union of two superclass enabled-sets can be taken: the *partial* enabled-set of the bounded_buffer class is precisely the composition of the *full* and *empty* sets [Figure 2.6 (a)]. This scheme facilitates inheritance: to synchronize the *get_rear* method of a new *extended_buffer* class, the bounded buffer class would be inherited, and, using the composition operator, *get_rear* would be added to the *full* enabled-set. The *partial* set will correctly use *get_rear* in the subclass, because it inherits from *full*. [Figure 2.6 (b)].


```
empty:    put()
full:     get()
partial:  (+ empty full)   /* partial = get (), put ()  */
```

(a) The *partial* enabled set of a *bounded_buffer* class is the composition of the *empty* and *full* sets. (NB: the syntax used is not that of Rosette)

```
full:  (+ super.full get_rear())

  /* full = get(), get_rear()              */
  /* partial = get(), put(), get_rear()    */
```

(b) *Extended_buffer*, which inherits *bounded_buffer*, redefines its local copy of the *full* set to include the *get_rear* method. Because *partial* inherits from *full*, it also gets the new method.

FIGURE 2.6. Inheritance of Enabled-Sets

Composition allows new methods added to the superclass enabled-set to be automatically inherited by the subclass, an advantage over Kafura and Lee's strategy, which requires an explicit list of relevant superclass methods to appear in each sub-

class behavior name. However, there is no operator analogous to composition that allows methods to be *subtracted* from an enabled-set; subclasses requiring such subtraction would have to list all desired superclass methods in their enabled-sets. Using different semantics and syntax, Rosette also allows priority schemes to be represented through enabled-sets.

Neusius's Synchronizing Actions (1991) extends the synchronization mechanisms of ACT++ and Rosette. While stressing the importance of *decentralized interface control* those approaches provide for object methods, Neusius argues that neither adequately separates concurrency control from the methods themselves. The methods do part of the work (by specifying behavior abstractions or enabled-sets) that determines what methods will be available in the next state of the object; thus, encapsulation of the concurrency control mechanism is weakened. In Synchronizing Actions, state transitions are handled by methods and variables accessed only by the Interface Control Space, a sub-object having its own thread of control. As in Figure 2.7, the control space uses *matching rules* associated with each method to determine if requests for that method can be accepted; *pre-actions* specify the consequences of invoking a given method (e.g., the locking of conflicting methods), while *post-actions* describe the conditions holding at method termination. Thus, the method itself does not interact with the object concurrency control. Moreover, potential method concurrency can be increased by putting all shared instance variables under the control of pre and post-actions — since these are only executed by the single-threaded Interface Control Space, such variables will be updated consistently.

Despite the complete encapsulation of concurrency control, Neusius admits that elements of the interface control space may have to be redefined if methods are added to the class or its subclasses. This redefinition, while restricted only to *behavior abstractions* specifying the names of excluded methods, is not as elegant as the composition scheme of Rosette, though perhaps more general.


```
class bounded_buffer;
...
concurrency_control:
   int N = 0;     // number of queue elements
   behavior abstraction
      op_on_head =    { get };
      op_on_tail =    { put };
public:
   method put (int elem);
      matching       ( N < MAX );
      pre            { exclude op_on_tail; }
      action         { ... }
      post           { N++; }
...
```

FIGURE 2.7. Synchronizing Actions Applied to a Bounded Buffer

```
class Buffer
{
private:
   full(int pos);
   empty();
   same_pos(int pos1, pos2);
delay:
   full(int pos)            delay put(item,pos);
   same_pos(int p1,int p2) delay put(item,p1),
                                 put(item,p2);
   empty ()                 delay get();
   get > put;
public:
   put(Item item,int pos);
   Item get();
};
```

FIGURE 2.8. C++ Conditional Waits

Though not pointed out in Neusius (1991), the incomplete encapsulation of synchronization in approaches such as ACT++ and Rosette prohibits direct reuse of uncontrolled classes, since class methods must be modified to incorporate part of the concurrency control mechanism. In an effort to allow easy reuse of sequential classes, Saleh and Gautron (1992) propose an extension to C++ that uses the concept of *conditional waits* (no relation to those used in Trellis/Owl) to synchronize method requests. A *delay* access specifier is added to the language (Figure 2.8). Declarations following the delay specifier are of the form *condition() delay method()*, where *condition()* is a private method assumed to return a boolean value, and *method()* is the name of a method that will be made inaccessible if *condition()* returns true. The (rather odd) buffer class shown in the figure requires that calls to *put* specify the buffer location in which an item is to be placed, while the *get* method returns an item through some (here unspecified) algorithm. The class has three private methods, *full*, *empty* and *same_pos*, used by the *delay* declarations: *full* delays calls to *put* if the desired location is already occupied; *same_pos* delays one of two (potentially concurrent) calls to *put* if they reference the same location; *empty* delays *gets* if the buffer is empty. Finally, the statement *get* > *put* signifies that *get* requests have priority over *put* requests.

Saleh and Gautron's approach offers several advantages over similar strategies. Sequential classes can be made concurrent by adding the *delay* declarations and appropriate condition methods — existing method code does not have to be changed. A simple request priority scheme is implemented using the > and < operators, and request parameters can be used by the concurrency control mechanism. However, the synchronization constraints provided by *delay* declarations are far less easily understood than those of Rosette, for example, and it is difficult to imagine how the *same_pos* method in Figure 2.8 would scale up to handle an arbitrary number of concurrent *put* requests. The approach requires a compiler extension to generate code

for *delay* statements and an underlying task manager to provide runtime support. Features such as asynchronous calls are provided by a template library.

2.2.3.4. *DRAGOON.* Reghizzi, Paratesi, and Genolini's ADA extension DRAGOON (1991) allows a group of object methods to be handled by identical synchronization rules, as do ACT++, Rosette and Synchronizing Actions, but uses inherited *behavioral classes* to specify these rules. Only *concrete* classes (those having code defined for all methods) may inherit behavioral classes, and only concrete behavioral classes may be defined. Furthermore, no behavioral class can be a subclass of another behavioral class, and a concrete class may inherit at most one behavioral class (the authors believe these restriction strike a fair balance between efficiency and simplicity). The addition of a behavioral class to a concrete class yields a *regulated* class.

Behavioral classes achieve method synchronization by specifying *deontic predicates* that must hold for a method to be activated. Deontic predicates use local object state information and synchronization counters; Reghizzi et al. demonstrate how these can be used to specify exclusion constraints and method request priorities. For example, Figure 2.9 shows how an *alternation* priority scheme would be implemented. Class methods are divided into two groups; a method in the first group is permitted to start executing if the number of past (not necessarily completed) activations of methods in that group is equal to the number of past activations of methods in the second group. A method in the second group, however, can only start executing if the number of *finished* activations of methods in the first group is greater than the number of past activations of methods in the second group. Thus initially, only a method from the first group is eligible to start executing, then only a method from the second group, then only a method from the first group, etc.

Behavioral classes can also be *generic* — applicable to a wide range of classes — and are combined at module link-time with the concrete classes that require them.

However, such a combination can only occur once: when behavioral class predicates are bound to concrete class methods, the resulting regulated class cannot be the basis of further inheritance. As well, the syntax used to specify deontic predicates is both low-level and cumbersome.

```
behavioral class ALTERNATION is
   ruled FOPS,SOPS
   -- class methods are divided into two
   -- categories, F(irst)_Op_Set and
   -- S(econd)_Op_Set.
where
   per (FOPS) <=> act (FOPS) = act (SOPS)
   -- a FOPS method can be activated iff.
   -- the number of past FOPS activations equals
   -- the number of past SOPS activations.

   per (SOPS) <=> fin (FOPS) > act (SOPS)
   -- a SOPS method can be activated iff.
   -- the number of finished FOPS activations
   -- exceeds the number of past SOPS activations.
end ALTERNATION;
```

FIGURE 2.9. Deontic Predicates [after Reghizzi et. al. (1991)]

2.3. Support of the Object-Oriented Paradigm

Two points must be considered when determining how well an approach to object concurrency control supports the object-oriented paradigm:

- How does the control mechanism affect the object-oriented properties of a class to which it's applied?
- Is the control mechanism itself object-oriented? More precisely, is the control mechanism's relationship with the class object-oriented?

The first point focuses on the changes perceived by a uncontrolled class's *clients* when that class is placed under control of the mechanism; the second considers how the

control mechanism interacts with the class methods and data, from the standpoint of a *developer* — someone who maintains the class code, or uses the class as a basis for inheritance. Although the two views are equivalent for parts of the object-oriented paradigm, enough differences exist to make consideration of both worthwhile. The attributes of the paradigm (section 1.2) are used as the evaluation criteria.

2.3.1. Modularity. Most approaches achieve modularity by concentrating concurrency control policies into distinct, explicit units. Trellis/Owl, however, uses a mix of queues and locks, and Rosette, Conditional Waits and Frølund's approach use functions not directly linked to the main control mechanism. Thus, these four approaches are not completely modular.

2.3.2. Encapsulation. Encapsulation is violated by approaches in two distinct ways. The visible behavior of controlled objects in CEiffel may be different than their uncontrolled behavior because of properties unique to active objects. Trellis/Owl, Hybrid, ACT++ and Rosette give object methods access to synchronization code, weakening encapsulation between the object and its control mechanism. The remaining approaches do not have either drawback.

2.3.3. "As Is" Reusability. From the client's perspective, a control mechanism provides good reusability if the interface of a class does not change when the mechanism is applied to it. Most approaches achieve this, though CEiffel may require client changes for the same reason it may violate encapsulation — certain types of concurrency, notably object autonomy and method asynchrony, cannot be shielded from the client (section 2.2.2.1).

From the standpoint of a person who actually incorporates the control mechanism into a class, reusability has two aspects:

- the ease with which uncontrolled classes can be put under control of the scheme;

- the ease with which the class implementation can be changed without affecting the concurrency control scheme.

Many approaches excel at the first type of reusability — CEiffel and Conditional Waits are explicitly designed to allow reusability between sequential and concurrent domains. DRAGOON achieves similar effects by creating controlled classes from the merging of sequential and behavioral classes, while the control "bodies" used in POOL could easily be added to sequential classes. However, such encapsulation is not present in Trellis/Owl, Hybrid, ACT++ or Rosette, all of which embed synchronization information in the method code. Thus, sequential classes cannot be used "as is" in these approaches. This is also true of Synchronizing Actions; the excellent synchronization encapsulation it provides is partially a result of hiding some instance variables (e.g., the count of buffer items) in the Interface Control Space. Since such variables would also be present in the sequential object, its method code must be modified if the approach is used.

The second type of reuse is more subtle. Just as a class interface should not change if its implementation does, the interaction of the concurrency control scheme with the class should be shielded from implementation changes. However, this is often not feasible: because intra-object concurrency is an implementation property (section 2.2.2.1), it is reasonable to expect that aspects of the control mechanism may change if the class implementation does. Approaches allowing intra-object concurrency, such as CEiffel and Synchronizing Actions, are sensitive to such changes, as might be any approach that makes use of object state (e.g., "buffer empty") to determine its concurrency control. Though such information is also relevant to a purely sequential object, and will likely be properly encapsulated if the uncontrolled class was well-designed, there is no guarantee that such encapsulation will exist.

2.3.4. Inheritance. The success of a control mechanism's support for inheritance from the client viewpoint is largely dependent on its success from the developer

viewpoint. Some approaches interfere with inheritance (Hybrid), disallow inheritance of control mechanisms (POOL and Trellis/Owl) or disallow inheritance in general once a control mechanism is in place (DRAGOON and Scheduling Predicates), and such limitations affect both clients and developers. However, even approaches that support inheritance may cause problems. ACT++, Rosette and Synchronizing Actions allow easy inheritance and incremental modification of concurrency control, but the scheme offered by Conditional Waits, while similar in power, is harder to understand. Frølund's automatic inheritance of constraints is elegant, but does not support some models of subtyping. Although CEiffel supports inheritance, subclasses may override superclass annotations that affect the client's view of the class — the decoupling of concurrency features from the behavior of class methods allows the former to be changed arbitrarily. On the other hand, some of these potential changes arise from CEiffel's support of multiple inheritance, a feature not provided by other approaches.

2.3.5. Logical Design Units. Whether an approach provides objects that are natural units of software design, from either the client or developer standpoint, is largely determined by its support of the above properties, and should not be measured separately. It is worth noting, however, that the approaches to concurrency control taken by some mechanisms are easier to understand than others. ACT++, Rosette and Synchronizing Actions allow concurrency to be seen as state transitions, while Scheduling Predicates and Frølund's approach provide intuitive ways to describe constraints.

2.3.6. Summary and Observations. The varying detail of available approach descriptions means that any quantification and comparison of the approaches will be both difficult and somewhat arbitrary. Nevertheless, Table 2.1 assigns rough numeric ratings to each approach's support of the object-oriented paradigm. Ratings in each category range from 0 to 100, with 0 signifying "no support" and 100 signifying "excellent support." An approach's score in each category is based on the discussion

Approach	Modular.			Encapsul.			Reusab.			Inher.			Avg.
	C	D	A	C	D	A	C	D	A	C	D	A	
Owl	100	60	80	100	60	80	100	60	80	50	50	50	73
CEiffel	100	100	100	80	100	90	60	80	70	90	90	90	88
Sch. Pred.	100	100	100	100	100	100	100	100	100	0	0	0	75
Frølund	100	70	85	100	100	100	100	80	90	60	80	70	86
POOL	100	100	100	100	100	100	100	80	90	50	50	50	85
Hybrid	100	100	100	100	80	90	100	60	80	100	50	75	86
ACT++	100	100	100	100	80	90	100	60	80	100	70	85	89
Rosette	100	70	85	100	80	90	100	60	80	100	80	90	86
Synch. Act.	100	100	100	100	100	100	100	80	90	100	80	90	95
Cond. Wait	100	70	85	100	100	100	100	80	90	100	80	90	91
DRAGOON	100	100	100	100	100	100	100	80	90	0	0	0	73
Abbreviations: C: Client D: Developer A: Average													

TABLE 2.1. Ratings of Approach Support for the Object-Oriented Paradigm

in sections 2.3.1 to 2.3.5; all scores were initially set at 100, and decremented in units of 10 to reflect the severity of flaws noted in the discussion.

Assuming that 50 is an average score, all approaches provide above-average support for the object-oriented paradigm — hardly a surprising result, since all are intended for use with objects. Nevertheless, some fare considerably better than others. Synchronizing Actions, Conditional Waits and ACT++ provide the best support, while Trellis/Owl, Scheduling Predicates and DRAGOON provide the weakest. While most approaches did well in most categories, many provide poor support for inheritance. Correlations between attribute scores are difficult to find; although there is a relation between an approach's modularity and its encapsulation, the high scores achieved on both attributes by most approaches make the significance of this hard to assess. Likewise, the strong inverse relation between reusability and inheritance exhibited by Trellis/Owl, POOL, DRAGOON and Scheduling Predicates is offset by positive correlations shown by Synchronizing Actions and Conditional Waits.

Comparing attribute support from the perspectives of class client and class developer yields more interesting results. In particular, one attribute, reusability, shows an inverse relation between support from the client's perspective and support from the developer's perspective; that is, good client support is often linked with poor developer support, and vice-versa. To a lesser extent, the same observation holds for encapsulation. Yet from both viewpoints, there is usually a correlation between encapsulation and reusability (e.g., from the client's perspective, the quality of support provided for encapsulation matches that provided for reusability, and the same holds true from the developer's perspective). Finally, support for inheritance, from the developer's standpoint, is usually equal to or inferior to that provided to the client.

Table 2.2, which compares the rating of an approach with some of its characteristics, shows that the paradigm support provided by an approach is not correlated with the type of objects it handles, their threading (for a passive object, the number of threads that may access it at once; for an active object, the number of threads it itself has), nor with the relation between the class and its control mechanism. There is some correlation, however, between the support provided and the level at which the control mechanism interacts with class methods — in general, decentralized approaches offer better support than centralized ones.

2.4. Conclusions

This chapter has shown that object concurrency control is not straightforward — many approaches to the problem exist, and none provides outstanding support for object-oriented programming. The results of the above evaluation, while tenuous at best, bring to light some interesting points:

- Concurrency control is difficult to reconcile with inheritance, especially from the standpoint of the class designer or implementor. This is a well-known problem (c.f., Kafura & Lee, 1989; Nierstrasz & Papathomas, 1990; Saleh & Gautron, 1992), though its causes have not been yet completely identified.

Approach	Objects	Threading	Relation	Level	Paradigm Support
Owl	P	S	Orth	Decen	73
CEiffel	P/A	M	Het	Decen	88
Sch. Pred.	P/A	M	Het	Decen	75
Frølund	A	M	Het	Decen	86
POOL	A	S	Hom	Cen	85
Hybrid	A	S	Hom	Decen	86
ACT++	A	M	Hom	Decen	89
Rosette	P/A	M	Hom	Decen	86
Synch. Act.	A	M	Hom	Decen	95
Cond. Wait	A	M	Hom	Decen	91
DRAGOON	A	M	Hom	Cen	73

Abbreviations:			
A: Active	S: Single-threaded	Orth: Orthogonal	Cen: Centralized
P: Passive	M: Multi-threaded	Het: Heterogeneous	Decen: Decentralized
		Hom: Homogeneous	

TABLE 2.2. Approach Characteristics and Paradigm Support Ratings

- Though reusability may be linked with encapsulation, neither attribute is easy to support concurrently from the standpoint of class client and class developer. This indicates that, at least in the domain of object concurrency control, the two views are sometimes irreconcilable.
- The support provided by an approach is largely independent of the types of objects for which it is designed, and its relationship with these objects.

To recognize the significance of these findings and explain their causes, one must also consider the other relevant feature of the surveyed approaches: their success as concurrency control mechanisms. This is done in Chapter 3.

CHAPTER 3

Evaluation of Approaches

Chapter 2 described various approaches to object concurrency control, discussed their classification, and determined how well they support the object-oriented programming paradigm. This chapter focuses on the success of each approach as a concurrency control mechanism. Section 3.1 describes the criteria used to evaluate the approaches, and how this evaluation will be carried out. Section 3.2 evaluates the approaches, section 3.3 summarizes the results and compares them to the approaches' support of the object-oriented paradigm (section 2.3.6), and section 3.4 discusses the implications of the results.

3.1. Evaluation Criteria

3.1.1. General Criteria. Determining the worth of a particular concurrency control mechanism is not easy; some approaches are designed for specific problems and environments, and perform well only in such situations, while others have more generality at the expense of efficiency or elegance. Thus, the concept of a "best" control scheme is perhaps specious. However, it is useful to discuss what properties such a scheme might have; this section describes such general, intuitive criteria.

3.1.1.1. *Flexibility.* The flexibility of an approach most obviously means its ability to cope with different types of synchronization problems: an approach that can deal equally well with standard read/write lock conflicts and synchronization based on resource state, for example, would be preferable to one handling only one of the two

problems. However, flexibility is also a function of the *environments* in which an approach can operate — flexible approaches are not tied to a particular communication mechanism, such as shared memory or message-passing.

3.1.1.2. *Development Facilitation.* An approach will not be widely-used if its solutions to synchronization problems are difficult to develop, understand, and modify. Such flaws limit the approach's usefulness in the software development cycle, because people other than those who originally used the approach to solve a problem will invariably need to understand and modify the solution.

3.1.1.3. *Ease of Integration.* Many existing approaches are part of a complete research system, implemented as a research language, or designed to run on experimental hardware. While these are not deficiencies *per se*, the suitability of an approach to conventional, widely-used systems must be considered.

3.1.1.4. *Efficiency.* Efficiency is essential to an approach's success, especially if the approach will be used to implement low-level, widely-used synchronization primitives. Three types of efficiency must be considered: *space* efficiency concerns how much code the approach requires to be implemented; *time* overhead concerns both the number of machine instructions executed by the approach, and any time-delays incurred by message-passing; *concurrency* efficiency reflects how much (safe) concurrent access an approach allows to protected objects.

3.1.2. Quantifying the Criteria. The above criteria give a general idea of the desirable properties of a concurrency control mechanism, but are too vague to be of much use. Even if they were specific, measuring an approach against them would require using that approach in a software engineering project: a costly, time-consuming task. Detailed criteria are thus necessary, but not sufficient; they must also be measurable, to a large extent, using only the technical description of an approach.

Bloom (1979) argues that the lack of such criteria leads most synchronization schemes to be tested on only a small sample of problems, then later modified or discarded when experience proves them unsuitable for other situations. To avoid such wasted effort, he offers a different evaluation approach.[1]

3.1.2.1. *Synchronization Constraints.* To facilitate thorough evaluations of synchronization mechanisms, Bloom discusses the two major purposes of any synchronization scheme: enforcement of exclusion and enforcement of priority. *Exclusion* is the denial of a resource to a process when that process would compromise work in progress on the resource, or when the resource state cannot accept the process's request; *priority* involves the scheduling of process accesses to a resource to allow maximal concurrency or some other efficiency. Both properties are *constraints* on processes wishing to access resources, and are normally qualified by one or more types of additional information:

- *requested operation type*: used to determine the priority of the process, or if the process would interfere with operations already in progress.
- *relative request times*: used in many scheduling algorithms (e.g., First-Come, First-Served).
- *request parameters*: parameters such as "disk track number," used to determine a process's priority.
- *synchronization state*: information pertaining to concurrent access of the resource, such as the number of processes accessing it.
- *local state of resource*: state of the "basic" resource, ignoring any variables concerning synchronization. Examples of such states include "buffer empty" and "disk full."
- *history information*: information on completed operations, often equivalent to the local state of the resource.

[1] A March, 1992 Science Citation Index search on Bloom's paper and synchronization keywords revealed no references that extend or surpass his concepts.

3.1.2.2. *Bloom's Criteria.* Bloom uses the above taxonomy of constraints, and the concept of encapsulation (section 1.2), to introduce four new criteria for synchronization mechanism evaluation:

(1) *encapsulation of synchronization*: Each resource should contain the code to handle its synchronization, thus making its concurrent nature transparent to accessing processes (i.e., each process can use the resource as if it had exclusive access).

(2) *isolation of synchronization code*: There should be complete separation of the code of the "actual" resource from the code that controls its synchronization, because these implement logically different functions.

(3) *expressive power*: A synchronization scheme should be capable of handling all varieties of exclusion and priority constraints discussed above. Bloom suggests that expressive power be measured by testing the resource against a series of synchronization problems covering all constraint variants.

(4) *ease of use*: Bloom argues that complex synchronization problems may involve many constraints: the synchronization mechanism should allow these to be implemented *independently*; that is, the solution to any one constraint should not require modification when a solution to a new constraint is added. This can be checked by examining how two similar synchronization problems are handled by the mechanism: constraints common to the two problems should be implemented identically.

3.1.3. Applying the Criteria. The concepts in Bloom (1979) allow some of the general criteria discussed in section 3.1.1 to be more rigorously defined, and also give a framework for applying them. This section outlines how this application will be done:

3.1.3.1. *Flexibility.* One aspect of an approach's flexibility — its ability to cope with different synchronization problems — is captured by Bloom's "expressive power"

criterion. To determine the expressive power of an approach, Bloom evaluates its ability to solve the following synchronization problems:

- synchronization of a *bounded-buffer* resource, requiring use of information about the resource's local state;
- a *First-Come-First-Served (FCFS)* priority scheme, requiring use of request times;
- a *readers-priority* scheme, requiring use of request types and the synchronization state of the resource;
- a *disk-scheduler*, requiring use of request parameters (the desired disk track number, so that requests needing little disk head movement can be given priority);
- a *one-slot buffer*, requiring use of history information to determine if requests can be granted.

These problems test an approach's ability to deal with different types of constraint information, and are used to test the approaches surveyed in Chapter 2.

Evaluating the other aspect of flexibility, that of environment, is less straightforward. In most cases, however, the given descriptions are detailed enough to determine how general each approach actually is.

3.1.3.2. *Development Facilitation.* The potential modifiability of an approach's solutions can be assessed by using Bloom's "ease of use" criterion; that is, by determining how much solutions to different synchronization constraints interfere when those constraints are combined in the same problem. Bloom accomplishes this by comparing solutions to readers-priority and writers-priority schemes, problems having different priority but identical exclusion requirements.

In Chapter 2, the support an approach provided for object-oriented programming was considered from two viewpoints: that of client objects and that of designers or

implementors of controlled classes. A similar dichotomy is reasonable when considering the *comprehensibility* of an approach. For this attribute, however, different views are appropriate — the idea of a class "developer" is divided into those who *use* a class with a control mechanism already in place, and those who *write* or modify such mechanisms for a class. The first type of developer must understand an existing control mechanism in order to use objects of the controlled class. Thus, control solutions understandable by examining a few program statements are likely to ease program development. From the standpoint of the mechanism writer, the effort required to arrive at such a solution and the effort required to change it are important.

Bloom's notions of "encapsulation of synchronization" and "isolation of synchronization" are also relevant to an approach's development facilitation. However, these criteria are equivalent to the definitions of encapsulation given in section 2.3: the first corresponds to the client's notion of encapsulation, while the second matches that of the developer. Therefore, an approach's encapsulation rating is considered to be part of its development facilitation rating. The significance of this is further discussed in section 3.4.

3.1.3.3. *Ease of Integration.* Ease of integration is difficult to measure without attempting to incorporate a given approach into the target computing environment. However, the descriptions of the approaches are used to get a general idea of how they could be integrated into conventional systems. For the purposes of this thesis, a "conventional system" is one using a common operating system (UNIX, DOS, or Macintosh), either networked or multi-threaded, and class-based, static-typed languages such as C++ or Eiffel.

3.1.3.4. *Efficiency.* Efficiency is also difficult to measure — some of the approaches are unimplemented, and performance measurements are generally unavailable for those that are. Rough estimates of efficiency are made by considering the following:

- the code (space) overhead incurred by incorporating an approach into a class, or by inheriting the class(es) that constitute the approach;
- the number of operations typically performed by an approach when handling incoming requests, and the amount of message-passing overhead incurred.
- the concurrency possible when using an approach.

3.2. Evaluation of Approaches

Based on the above strategy, this section evaluates the concurrency control abilities of the approaches.

3.2.1. Flexibility. Scheduling Predicates (section 2.2.2.2) is the only surveyed mechanism that directly uses Bloom's evaluation approach. As discussed in McHale et al. (1992), the predicates can be used to implement five of the six constraint types described by Bloom, the major exception being local resource state. This precludes the mechanism from being used to solve many common synchronization problems.

Other approaches come close to satisfying Bloom's criteria. The functionality of Trellis/Owl (section 2.2.1.1) is roughly equivalent to that of monitors. Bloom finds that monitors are able to handle most types of exclusion and priority constraints, resulting in good expressive power, and are not limited to a particular operating environment. CEiffel (section 2.2.2.1) also offers good flexibility. The standard Eiffel *require* statement used by CEiffel can handle constraints based on request parameters and local object state, while the *compatibility* annotation, specifying which methods may execute concurrently, is in effect a specification of the object's synchronization states. CEiffel's major limitation lies in its simple FCFS priority scheme — this cannot handle priority-based on request type and, more seriously, is not described at the level of either annotations or *require* statements. Frølund's *disabled* predicates (section 2.2.2.3), though potentially able to handle most synchronization constraints, are not described in enough detail for accurate evaluation.

Other approaches maintain object state consistency by specifying which methods can be validly invoked in the object's current state, and by accepting only such methods. In POOL (section 2.2.3.1) this is accomplished by the *body* process of each object, which explicitly lists the available methods. Hybrid (section 2.2.3.2) and ACT++, Rosette, Synchronizing Actions and Conditional Waits (section 2.2.3.3) associate queues with methods, allowing requests to be delayed until they can be processed, while DRAGOON (section 2.2.3.4) associates predicates with method names to specify their availability. Thus, such approaches primarily use the local state of the resource to achieve synchronization, and would be very suitable for objects such as bounded buffers. Priority based on relative request times is also implicitly handled, since acceptable requests are processed on a first-come, first-served basis.

Other constraints, such as request parameters and priority based on request type, are not handled by POOL, ACT++ or Synchronizing Actions, and cannot be implemented easily in Hybrid. While Rosette allows parameters to be matched against enabled-sets as patterns, the priority-handling scheme uses enabled-sets in a completely different, less-intuitive way. Conditional Waits and DRAGOON allow easier specification of both constraint types, and are the only homogeneous approaches that support synchronization state constraints.

3.2.2. Development Facilitation. The development facilitation of an approach is often a trade-off between potential for modifiability and ease of modification. Trellis/Owl's control system, for example, shares the ability of monitors to handle combinations of constraints (Bloom, 1979), and is thus very modifiable. However, although the initial control scheme would not be difficult to write, Bloom notes that certain constraint combinations require extra work to implement, and that some constraints cannot be implemented independently of others. Moreover, monitors do not directly provide isolation of synchronization, and this deficiency is worsened by Trellis/Owl's lack of automatic mutual exclusion.

CEiffel, Scheduling Predicates and Frølund's approach do not offer the same potential flexibility as Trellis/Owl, but may shield the class user from low-level details. Both approaches offer easily understood descriptions of concurrency control, and these can be combined as recommended by Bloom. The annotations and standard Eiffel assertions used in CEiffel are also easily understood, when considering a single class. However, the high level of abstraction provided by a given annotation obscures the reasons why that annotation was used in the first place, and thus increases the likelihood of incorrect modification. For example, subclass redefinition of an annotation may be arbitrary. Moreover, the class user may be confused by the roles of inherited annotations: when multiple inheritance is present, these may differ from their superclass meanings.

Most homogeneous approaches are not designed to allow customization of synchronization algorithms, but instead use the object's state to ensure its synchronization. However, ACT++, Rosette, Synchronizing Actions, Conditional Waits and Hybrid allow the *class* to be modified through inheritance, and ensure that the additions to or subtractions from the object's possible states resulting from this modification are mirrored in its concurrency control. The first three approaches accomplish this elegantly, while the inheritance scheme of Conditional Waits is more cumbersome and inheritance in Hybrid may conflict with object synchronization. Neither POOL nor DRAGOON allows inheritance of control mechanisms, though DRAGOON achieves reuse of synchronization code through its generic behavioral classes, and also allows some types of synchronization constraints to be combined, an advantage shared with Conditional Waits and Frølund's approach.

3.2.3. Ease of Integration. No approach requires special hardware, but software requirements vary widely. The monitor-based approach of Trellis/Owl, for example, is general enough for implementation in most existing languages, while CEiffel and Scheduling Predicates both require substantial preprocessors. Homogeneous ap-

proaches often require a fairly complex runtime environment: Hybrid's includes *object managers* to time share between active domains, implementable as UNIX processes. The *behavior names* used in ACT++ can be implemented by C++ virtual functions, as might the similar approaches used by Rosette and Synchronizing Actions. Conditional Waits is a C++ extension, and DRAGOON, written in ADA, could also be implemented in C++.

3.2.4. Efficiency. The three measures of efficiency — time, space and concurrency — differ widely from approach to approach. Trellis/Owl is time and space-efficient, but in most situations shares the mutual-exclusion property of monitor calls. Other approaches provide concurrency at the expense of other efficiencies. CEiffel allows both internal concurrency and asynchrony, but requires a threading library to realize these features. Scheduling Predicates also allows internal concurrency, but requires synchronization counters to implement the relevant predicates. The truth value of predicates must also be evaluated periodically; McHale et al. (1992) discusses how such evaluations can be optimized. Though details are not given, Frølund's approach would probably have similar requirements.

Homogeneous approaches, with the exception of POOL, entail substantial space overhead: DRAGOON objects must inherit a behavioral class for concurrency control, while the others require some form of object manager to deal with incoming messages. Such managers exist on a per-domain (functional object group) basis in Hybrid, and on a per-object basis in ACT++, Rosette, Synchronizing Actions and Conditional Waits, which also apparently requires each *delay* function be evaluated whenever a message is received. Synchronizing Actions, Conditional Waits and DRAGOON allow full intra-object concurrency, while Hybrid, ACT++ and Rosette offer a more limited form (objects may not share instance variables). Such concurrency is not permitted in POOL.

Approach	Flexibility		
	Environ.	Power	Average
Owl	100	100	100
CEiffel	100	80	90
Sched. Pred.	100	80	90
Frølund	100	90	95
POOL	100	40	70
Hybrid	100	60	80
ACT++	100	60	80
Rosette	100	80	90
Synch. Act.	100	60	80
Cond. Wait	100	80	90
DRAGOON	100	80	90

TABLE 3.1. Flexibility Ratings of Approaches

3.3. Summaries and Observations

3.3.1. Concurrency Control Abilities. Based on the above discussion, Tables 3.1 through 3.3 present rough numeric ratings of the approaches' abilities as concurrency control mechanisms, using the same rating system as in chapter 2. Table 3.4 summarizes the results of the other three tables, and gives an overall rating for each approach.

The tables indicate that most attribute correlations are tenuous, and weakened by exceptions. Table 3.1, summarizing approach flexibility, shows only that independence of operating environment is much easier to achieve than synchronization power; the two sub-attributes are not related. A fairly strong relation *is* found between comprehensibility of a control scheme from the user's standpoint and comprehensibility from the writer's standpoint (Table 3.2) — in many approaches, class users and concurrency control implementors have to understand similar things. A weaker relation between comprehensibility in general and encapsulation is contradicted by DRAGOON and Conditional Waits. Other sub-attributes affecting development facilitation exhibit few correlations, with some approaches having a positive correlation

Approach	Development Facilitation					
	Modifiab.	Comprehensibility			Encap.[a]	Average
		User	Writer	Avg.		
Owl	80	50	50	50	80	70
CEiffel	80	60	60	60	90	77
Sched. Pred.	100	100	100	100	100	100
Frølund	60	100	100	100	100	87
POOL	30	70	70	70	100	67
Hybrid	60	60	60	60	90	70
ACT++	70	90	70	80	90	80
Rosette	70	70	60	65	90	75
Synch. Act.	80	90	90	90	100	90
Cond. Wait	80	70	50	60	100	80
DRAGOON	80	60	40	50	100	77

[a]from section 2.3

TABLE 3.2. Development Facilitation Ratings of Approaches

between two given sub-attributes and others having a negative one. A weak correlation does exist between an approach's time and space efficiencies (Table 3.3), though this may only be indicative of lack of optimizations on either sub-attribute. Interestingly, there is a moderate inverse relation between time and space efficiencies and the concurrency allowed by an approach.

Table 3.4 takes the averages of sub-attributes to obtain full attribute scores, and reveals that correlations between attributes are also tenuous, with only flexibility and development facilitation showing a relation. However, the sub-attribute of synchronization power (Table 3.1) correlates positively with ease of integration, a somewhat surprising result.

3.3.2. Correlation with Other Attributes. Before attempting to draw conclusions from the analysis, it is useful to compare the approaches characteristics from chapter 2 with their overall ratings as concurrency control mechanisms (Table 3.5). Most notably, there is little correlation between an approach's support of the object-oriented paradigm and its success as a concurrency control mechanism. Although

Approach	Efficiency			Average
	Time	Space	Concurr.	
Owl	80	80	20	60
CEiffel	70	60	90	73
Sched. Pred.	70	70	100	80
Frølund	70	70	80	73
POOL	90	80	20	63
Hybrid	70	70	60	67
ACT++	70	60	40	57
Rosette	70	60	40	57
Synch. Act.	60	60	80	67
Cond. Wait	50	60	70	60
DRAGOON	70	70	80	73

TABLE 3.3. Efficiency Ratings of Approaches

Approach	Flexibility	Development Facilitation	Ease of Integration	Efficiency	Overall
Owl	100	70	80	60	78
CEiffel	90	77	60	73	75
Sched Pred.	90	100	70	80	85
Frølund	95	87	70	73	81
POOL	70	67	70	63	68
Hybrid	80	70	50	67	67
ACT++	80	80	60	57	69
Rosette	90	75	60	57	71
Synch. Act.	80	90	50	67	72
Cond. Wait	90	80	80	60	78
DRAGOON	90	77	70	73	78

TABLE 3.4. Overall Ratings of Approaches

Approach	Objects	Threading	Relation	Level	Scores	
					Paradigm Support	Concurrency Control
Owl	P	S	Orth	Decen	73	78
CEiffel	P/A	M	Het	Decen	88	75
Sch. Pred.	P/A	M	Het	Decen	75	85
Frølund	A	M	Het	Decen	86	81
POOL	A	S	Hom	Cen	85	68
Hybrid	A	S	Hom	Decen	86	67
ACT++	A	M	Hom	Decen	89	69
Rosette	P/A	M	Hom	Decen	86	71
Synch. Act.	A	M	Hom	Decen	95	72
Cond. Wait	A	M	Hom	Decen	91	78
DRAGOON	A	M	Hom	Cen	73	78

Abbreviations:
A: Active S: Single-threaded Orth: Orthogonal Cen: Centralized
P: Passive M: Multi-threaded Het: Heterogeneous Decen: Decentralized
Hom: Homogeneous

TABLE 3.5. Summary of Approach Characteristics

most approaches offering good paradigm support achieve good or acceptable concurrency control (Frølund's approach, Synchronizing Actions and Conditional Waits), the reverse is not always true — Trellis/Owl and Scheduling Predicates succeed as control mechanisms but do not offer good paradigm support, mostly because of poor inheritance capabilities.

Table 3.5 also shows that orthogonal and heterogeneous approaches generally provide better control mechanisms than do homogeneous ones. This may be a result of concurrency difficulties inherent in active objects: in particular, the tendency of some homogeneous approaches to prohibit or limit internal concurrency.

3.4. Discussion

Most of the attribute correlations discussed above are weak and have significant exceptions, meaning that conclusions about a particular pair of attributes cannot usually be made. Perhaps the most important correlation is one that does not exist

— a correlation, either positive or negative, between paradigm support and success as a concurrency control mechanism. Thus, while mechanism paradigm support does not preclude good object concurrency control, *neither does it guarantee such control*. Some approaches provide one without the other.

Significantly, however, one attribute of object-oriented programming, encapsulation, was also used to evaluate control mechanism success (in Table 3.2), because chapter 2's definitions of encapsulation were equivalent to Bloom's "encapsulation of synchronization" and "isolation of synchronization" criteria. If one subscribes to Bloom's arguments, this equivalency is a strong argument in favor of the paradigm.

Of course, one could argue that Bloom's criteria, the other criteria presented in this thesis, and the numeric scales used to evaluate the criteria are open to question. No theoretical underpinnings were provided for these, and some of the findings resulting from their application are troubling. In particular, the lack of noticeable correlations between attributes may indicate that neither the attributes themselves nor the methods used to evaluate them were sufficiently defined. Nonetheless, there is evidence that using criteria such as Bloom's allows the strengths and weaknesses of a control mechanism to be discovered without using test implementations (McHale et al., 1992). And while the numeric evaluations carried out here and in chapter 2 were not rigorous, they provided a framework by which approach merits could be discovered and quantified; this process revealed much about the nature of object concurrency and concurrency control problems.

In fact, the *absence* of strong attribute correlations is itself an important finding, indicating that many concurrency control approaches actually are designed to handle *different* problems. Some concurrency control mechanisms handle intra-object concurrency; some allow priority specification; some deal with object state; some consider the implications of active object systems. No approach deals with all problems, and those handling more than one often fail to distinguish between them. Without a criteria-based approach evaluation, this phenomena would be hard to recognize.

Finally, Bloom's criteria illuminate a paradox in concurrency control. As noted above, the "isolation of synchronization" criterion is equivalent to encapsulation of synchronization from the object implementor's point of view — Bloom argues that the module that synchronizes a resource should be separate from the resource itself, since the two perform logically distinct functions. However, Bloom also indicates that resource state is one of the six types of information a synchronizer might need, implying that complete separation of synchronizer and resource is not possible. Yet many approaches to object concurrency control, in an effort to achieve encapsulation, logically sever the synchronizer from the rest of the object.

Chapter 4 investigates these two findings — the distinct types of concurrency control problems and their relations to the controlled objects — and reveals why inheritance is difficult to reconcile with concurrency control.

CHAPTER 4

Concurrency Control Types and Dependencies

The evaluation performed in Chapter 3 showed that different approaches to "object concurrency control" may deal with different problems, and further indicated that one attribute of the object-oriented paradigm, encapsulation, may be incompatible with concurrency control. This chapter expands on these findings. Section 4.1 discusses four problems that fall into the category of concurrency control, and shows that all depend intrinsically on attributes of the controlled objects. These dependencies explain why inheritance and concurrency control are difficult to reconcile — section 4.2 analyses how inheritance affects concurrency control dependencies, and thus solutions to concurrency control problems, while section 4.3 argues that most approaches to object concurrency control fail to recognize the importance of these dependencies.

4.1. Concurrency Control Types and Dependencies

4.1.1. Problem Types. At least four different problems fall under the heading of object concurrency control:

Synchronization. Trivially, all concurrency control policies are synchronizers — they determine what requests a resource will accept at any given time. But such decisions are often based on information known only to the control mechanism and irrelevant to the resource itself. A more intuitive definition of synchronization focuses on resource state (one of Bloom's six information types). Object synchronization is thus the determining of whether a request can be accepted, given the object's

state. For example, an empty *buffer* object cannot accept requests to dequeue items. In sequential programs, requests for which the object is unsynchronized generally cause run-time exceptions; in concurrent systems, an object may delay such requests until the processing of other types of requests makes them acceptable. Monitors, ACT++, Rosette and Synchronizing Actions are examples of approaches to object synchronization.

Concurrent Access Control. Unlike synchronization, concurrent access control has no counterpart in sequential systems, being required only when multiple threads of control access the same data. As discussed in chapter 1, concurrent access control is needed in such cases to ensure that data is updated consistently; semaphores, CEiffel, Synchronizing Actions and DRAGOON offer ways of controlling concurrent access.

Priority. Like concurrent access control, priority is only an issue in concurrent systems — it seeks to order requests such that efficient use of the object is maximized. Because "efficiency" can have different meanings in different situations, flexible priority schemes allow ordering based on request type, request parameters and object state. Scheduling Predicates and DRAGOON have good support for priority.

Autonomy and Asynchrony. Autonomy and asynchrony pertain only to active object systems — those in which objects possess threads of control. Autonomous object methods may execute repeatedly to model continuous behavior, while asynchronous methods allow the requesting object to continue executing immediately after making the request. Although such active object features are better classified as concurrency, rather than concurrency control, problems, they are discussed here because they complicate concurrent systems in much the same way that synchronization does. CEiffel is the only surveyed approach that supports active object properties.

4.1.2. Problem Dependencies. The four types of concurrency control problem described above are logically distinct. However, they all depend on one or both aspects of the object being controlled, its interface and its implementation.

As discussed in chapter 1, a key feature of the object-oriented paradigm is the distinction between object interface and implementation. An object's *interface* consists of the methods visible to other objects; this visibility can be further divided into method signatures (the object types the method takes as arguments, if any, and the object type it returns, if any) and the behavior of the method. Specification of this behavior, and of the method's signature, can be made without reference to the code that implements the method. This allows the method implementation to be changed without affecting how clients of the method interact with it.

However, the concurrency control scheme of an object is, in many cases, not uniquely dependent on the object's interface — often, it cannot be considered separately from the implementation. Furthermore, solutions to one problem type may be affected by solutions to another:

Synchronization, concerned with what request types the object can accept, is dependent on the state of the object; this is often an interface property: states such as "buffer empty" are part of the buffer object's specification. However, object states may also reflect implementation; "buffer full," for example, is not part of the abstract data type specification of a buffer, because it only pertains to implementations imposing a limit on the buffer size. For these implementations, such a state does affect the synchronization of the object, though this effect may be invisible to the object's clients. Many synchronization approaches, including ACT++, Rosette and Synchronizing Actions, allow interface and implementation states to be freely mixed.

Concurrent Access Control is strictly an implementation property (Löhr, 1992) — implementation changes may completely alter the methods that can be safely invoked concurrently. Allowing concurrent access to an object may adversely affect solutions to other control problems — values of object instance variables may change at unpredictable times in the presence of concurrency, meaning that such values cannot be used reliably (McHale et al., 1992). This problem precludes, for example, synchronization based on object implementation as discussed above, and is a major

limitation of McHale et al.'s Scheduling Predicates. Approaches such as ACT++ and Rosette avoid the instance variable problem by putting severe restrictions on internal concurrency: only methods that do not access the same instance variables may be active concurrently. Synchronizing Actions explicitly makes such data accessible only to the single-threaded concurrency control mechanism of each object. While this allows more intra-object concurrency, it requires counter-intuitive modification to the internal structure of formerly-sequential classes.

Priority schemes are used to increase the efficient utilization of an object; thus, like concurrent access control, they are implementation-dependent. However, changing an object's implementation does not invalidate its priority scheme, but may make it undesirable. Thus, the dependencies of priority schemes on implementations are more abstract than other control dependencies.

Autonomy and asynchrony are somewhat different from the other control problems. Neither property is necessarily affected by implementation changes, but adding either to an existing method changes that method's client-visible behavior, and hence its interface.

Thus, concurrency control problems are closely linked to both the interface and implementation of the controlled object, and sometimes to each other as well. These links make solutions to such problems difficult to find and maintain, and these difficulties are increased by the presence of inheritance.

4.2. Effects of Inheritance

To discover what effect inheritance has on concurrency control solutions, one must consider what happens when a class is subclassed. Most object oriented languages allow the following:

- Behavioral redefinition of inherited methods.
- Changes to the implementations of inherited methods, leaving their client-visible behaviors unchanged.

- Addition of new methods to those inherited from the parents.

Only the first possibility affects interface dependencies, but all three affect dependencies on implementation. The first possibility does so because changes to behavioral specification (or argument types) invariably result in implementation changes, while the second possibility causes such changes by definition. The third, while not modifying existing method implementations, may still affect implementation dependencies: unless the new method(s) access only new data, their addition may change the class's control policy. Synchronization based on implementation, for example, may be altered by such methods; this is the issue Rosette-like approaches are primarily intended to address. New methods also have effects on superclass concurrent access control policies: because such policies require knowledge of which methods access what data, addition of a subclass method that accesses existing data will make that knowledge incomplete. This prevents DRAGOON and Scheduling Predicates from allowing inheritance, and approaches that do allow both internal concurrency and inheritance either require redefinition of superclass policies to cope with new subclass methods (Synchronizing Actions) or use constructs that are both cumbersome and of limited extensibility (Conditional Waits).

4.3. Recognizing the Dependencies

The preceding two sections reveal that concurrency control problems are inextricably linked with the interfaces and implementations of the object in question, and that inheritance affects these dependencies. Yet, particularly where inheritance is involved, most concurrency control approaches try to separate the control mechanism from the controlled object. There are several related reasons for this.

- *Encapsulation*: Chapter 3 noted a paradox in Bloom's criteria for evaluating synchronization mechanisms: the control mechanism should be separated from the resource, yet may need knowledge of resource state. Isolation of concurrency control is attractive — for most types of control problem, the controller

and resource perform logically distinct functions, and the controller's behavior is invisible to resource clients. Separating the controller and resource effectively *encapsulates* the controller from the point of view of the resource designer/implementor, and many approaches to object concurrency control try to provide such encapsulation.

- *Implementation independence*: Because the object-oriented paradigm encourages separation of interface and implementation, class implementations are considered ephemeral, and not alluded to in method specifications.
- *Sequential reuse*: Because the client-visible behavior of a class may not change when that class is moved from a sequential to a concurrent environment, some approaches view control mechanisms as "add-ons," to be used and discarded as required.

Although all three goals are defensible, they are dangerous, because *they do not recognize the fundamental dependencies of concurrency control solutions*. This lack of recognition is most obvious when inheritance is considered: approaches striving for one or more of the three goals generally offer inadequate discipline or guidelines for inheriting concurrency control:

- *Encapsulation* of the control mechanism encourages abstract strategies that obscure the mechanism's dependence on class attributes. The designer of a Rosette subclass, for example, bears full responsibility for deciding how new subclass methods affect the superclass's enabled-sets — the name of an enabled-set's creation function may have no obvious relation to the method names in that enabled-set, and neither is directly linked to the actual object properties. Similar arguments apply to Conditional Waits: the functions used to determine request acceptance/rejection may be redefined arbitrarily by subclasses.

- *Implementation independence* prevents recognition of the control mechanism's dependencies on class implementation. Compatibility annotations of a CEiffel class may be arbitrarily changed by subclasses, for example, and implementation changes in general will affect the control schemes of most approaches. In such cases the class designer/implementor is responsible for recognizing and dealing with such effects.
- *Sequential reuse* encourages undisciplined grafting of control mechanisms onto classes. Strategies supporting sequential reuse, such as CEiffel and Conditional Waits, offer no means of verifying the results of this grafting. Moreover, the added mechanisms may be freely redefined by subclasses.

Even in sequential systems, inheritance can be confusing; a lack of guidelines for inheriting concurrency control makes it even more so in concurrent ones. Chapter 5 discusses how such guidelines may be realized, and chapter 6 describes an example of their application.

CHAPTER 5

Disciplined Inheritance of Concurrency Control

Chapter 4 explained why concurrency control is difficult to reconcile with inheritance: most approaches try to separate the control policy from its object dependencies, meaning that few guidelines or constraints exist for modifying the policy when inheritance changes these dependencies. This chapter develops concurrency control policies whose inheritance and modification are constrained by their dependencies. Section 5.1 discusses some desirable properties of control mechanisms, while section 5.2 differentiates between subtyping and code reuse, arguing that the distinction eases inheritance of control mechanisms. Sections 5.3 and 5.4 revisit the four types of concurrency control problems identified in chapter 4, considering inheritance of their solutions from the standpoints of subtyping and code reuse, respectively.

5.1. Disciplined Control Mechanisms

Ideally, a concurrency control policy would adjust itself according to changes in its dependencies. This is not currently feasible — changes to a class's interface or implementation generally lead to manual modifications of its clients and descendents, and concurrency control policies are sensitive to both types of change. However, the concurrency control policies achieved through a carefully-designed mechanism can be modified easily and correctly. The properties of such a *disciplined* mechanism are:

- *Automatic inheritance*: The concurrency control policy of a class should be automatically inherited by its subclasses, just as other class properties are.

- *Explicit dependencies*: The mechanism's syntax should clearly indicate what class properties a policy depends upon, so that the developer can recognize the need to modify it when those properties change.
- *Easy modification*: The mechanism's syntax should allow policy modification to be incremental, because subclasses will often only require that *part* of their inherited policy be changed.
- *Correct modification*: Modification of a policy should be constrained by its dependencies, to prevent arbitrary and incorrect change.

Mechanisms having these properties encourage a disciplined approach to object concurrency control. By default, concurrency control policies are *automatically* inherited by subclasses; the need to change them can be recognized *easily* because their dependencies are clearly stated. Policy modification can be done *simply*, because policies are incrementally modifiable, and *correctly*, because the policies' dependencies must be respected.

Unfortunately, no surveyed approach meets all four criteria, and some meet none. The generally-poor recognition of concurrency control dependencies indicates why all but the first criterion are seldom achieved. Failure to meet the requirement of automatic inheritance is more puzzling, because inheritance is one of the fundamentals of the object-oriented paradigm. The next section explores the concept of inheritance in more detail, revealing that, especially when concurrency control is concerned, traditional approaches to inheritance complicate the problem.

5.2. Subtyping and Code Reuse

For the dependencies of object concurrency control policies — class interfaces and implementations — to be explicit, they must be easily recognizable. An important step lies in realizing that inheritance of these two dependencies is equivalent to subtyping and code reuse, respectively — concepts mentioned in section 1.2.4.

Subtyping (Cardelli & Wegner, 1985; Danforth & Tomlinson, 1988) is the preservation of characteristics from parent to child; its major characteristics are:

- *Is-a* relationships. Subtyped classes further specify the concepts their supertypes model; for example, a Dog object 'is-a' Mammal object, in some sense. Though subtypes normally add behavior to the concept, they also further constrain it — 'dog' is not as general a concept as 'mammal.'
- *Transparency*. A subtype can be used transparently in all contexts in which its supertype(s) can, because it has all their attributes. In practice, this transparency takes two forms: some typing perspectives require that methods redefined in the subtype maintain only type compatibility with the supertype method (Porter III, 1992), while others require that essential supertype semantics be maintained (Cardelli & Wegner, 1985; Meyer, 1988).
- *Implementation independence*. A subtyping relationship says nothing about how the relevant classes are implemented: a subtype may have an implementation completely different from those of its supertypes.

Code-reuse is the sharing of implementation between parent and child, with no obligation on the child to preserve method behavior (e.g., the behavior of some inherited methods may be redefined, perhaps by adding to the inherited code). The semantics of code-sharing are far less structured than those of subtyping, and Cargill (1992) argues that such inheritance is often used to model *has-a* relationships better handled by containing objects as instance variables. Nevertheless, code reuse is a common goal, and implementations *are* important: to be usable in a program, every class must have an implementation somewhere in its type hierarchy.

Most popular object-oriented languages combine both subtyping and code reuse in their inheritance mechanisms (Goldberg & Robson, 1983; Meyer, 1988; Stroustrup, 1991). Yet there is evidence that doing so unnecessarily complicates the design, implementation and modification of object-oriented programs, and that separating the

two hierarchies is advantageous (America & van der Linden, 1990; Danforth & Tomlinson, 1988). The OATH class library (Kennedy, 1991) and Portlandish language (Porter III, 1992) effect such a split, with type hierarchies representing class interfaces and code hierarchies providing class implementations. In both approaches, the hierarchies are parallel: normally, there is a one-to-one correspondence between types and implementations, though potentially many implementations may be available for a type and conversely, an implementation for a particular type provides code for that type's abstract supertypes. Figure 5.1 illustrates these concepts, using a *buffer* type and its associated *array* implementation. Inheritance results in subtypes or code reuse, depending on the hierarchy. In both cases, descendents may add new methods or redefine old ones.

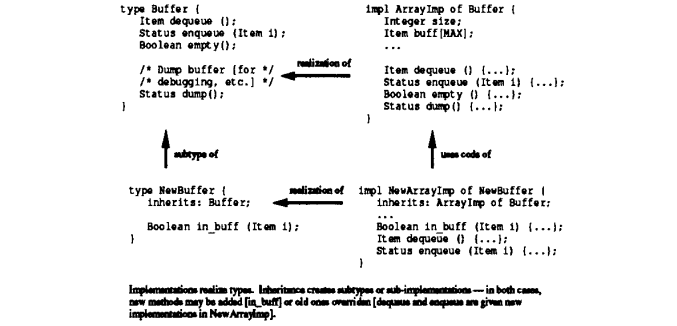


FIGURE 5.1. Subtyping and Code Reuse Hierarchies

A subtyping/code reuse split isolates the two major dependencies of concurrency control policies, allowing them to be made explicit. The following sections discuss

how this eases inheritance of concurrency control.

5.3. Concurrency Control through Subtyping

Interface synchronization constraints, method autonomy and client asynchrony are *specification* properties, affecting how clients deal with objects. Because most languages allow subclass objects to be used in place of superclass objects, subclassing should not alter such properties, a requirement that can be met by following the strong subtyping principle that subtypes respect essential supertype semantics.

5.3.1. Subtyping of Synchronization. For synchronization policies, strong subtyping implies that the subtype must respect both the conditions under which a supertype method may be invoked and the conditions that hold after it terminates. This can be realized by associating pre-conditions and post-conditions with methods. *Pre-conditions* specify the *domain* of object states under which the method can be invoked; post-conditions specify the *range* of object states that may hold after it terminates. Additionally, class *invariants* specify properties that hold whenever an object of that class is stable (e.g., before and after method executions). When a class is subclassed, invariants of all ancestors are implicitly composed with any new subclass invariant; redefined methods must be given new pre- and post-conditions. The new pre-condition may be *weaker* than the original, meaning that more object states allow the method to be invoked (Cardelli & Wegner, 1985); the new post-condition may be *stronger* than the original, meaning that fewer possible object states result from the method's execution. In practice, predicates are used to implement invariants, pre-conditions and post-conditions [c.f., Eiffel (Meyer, 1988)]; Figure 5.2 shows how this can be done.

Subtyping rules ensure that a redefined method will be usable by clients of the original method, and thus maintain the original method's synchronization constraints. As well, most requirements for disciplined concurrency control are met: invariants, pre-conditions and post-conditions make explicit the class's synchronization dependencies;

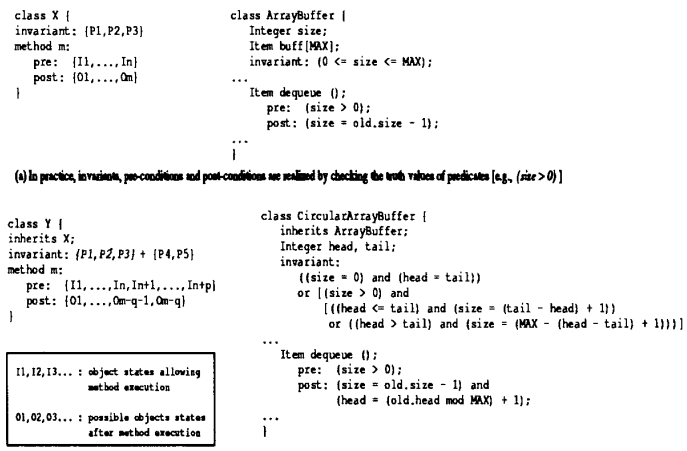


FIGURE 5.2. Application of Subtyping principles

invariants are inherited automatically, while pre-conditions and post-conditions are changed according to well-defined rules. However, such modification requires a complete *restatement* of the pre- or post-condition — these may be complex predicates, and thus difficult to change incrementally.

5.3.2. Subtyping of Active Object Properties. Autonomy and asynchrony, having effects visible to the object's clients, must be noted in the class interface. For example:

```
Boolean in_buff (Item i) asynch;
```

indicates that clients will proceed asynchronously after requesting the *in_buff* method. Because autonomy and asynchrony are binary properties, they cannot be changed incrementally — an autonomous or asynchronous supertype method must still be so if redefined by a subtype. It is thus surprising that CEiffel, an Eiffel extension, does not require subclasses to maintain the autonomy and asynchrony annotations of redefined superclass methods. Löhr (1992) argues that such annotations may need to be changed because of implementation changes, and that *deferred* routines (routines having an interface but depending on subclasses to provide an implementation) need not originally have annotations. Yet such annotations should be considered part of the method specification, and therefore immutable.

5.4. Concurrency Control through Code Reuse

Synchronization may depend on implementation as well as interface, and concurrent access and priority policies are completely dependent on implementation. To realize disciplined inheritance of such policies, implementations must be given some degree of permanence. This is implicit in languages like Eiffel that support pre- and post-conditions: because data structures can be referenced in the conditions, there is an assumption that these will not be changed by subclasses, since doing so would violate the inheritance rules of such conditions. Similar views are adopted by approaches that split the typing and code hierarchies (OATH, Portlandish) — implementations are often built around the data structures they use, suggesting that these should be the primary units of code reuse.[1] The following sections discuss how adopting such a strategy allows subtyping rules to be applied to implementation-dependent concurrency control problems.

5.4.1. Synchronization through Code Reuse. A class's synchronization policy may be partly determined by its implementation; as discussed in chapter 4, states

[1]Strict algorithm reuse is probably better achieved through generic classes or templates rather than inheritance.

like "buffer full" may have no counterpart in the class interface, but still concern the synchronizer. As well, the implementation is responsible for realizing synchronization constraints noted in the class interface. Fortunately, implementation-based pre-conditions state synchronization policies, just as interface-based pre-conditions do — in Figure 5.2, for example, the predicate ($size > 0$) ensures that *dequeue* will not be invoked if the buffer is empty. If a method is redefined by a sub-implementation, its modification should be constrained by invariants, pre-conditions and post-conditions. Thus, constrained algorithm changes are allowed, but the sub-implementation should not *override* implementation data structures — if this is necessary, a new implementation class, using a different data structure, is probably the appropriate solution.

Applying subtyping rules to implementations is not unduly restrictive. Consider an implementation class I, and a sub-implementation SI. By definition, SI inherits the code in I and, if unconstrained, may do any of the following:

(1) Add the implementation of a new method that
   (a) consists of entirely new data;
   (b) may have new data, but also accesses or modifies data inherited from I.
(2) Redefine an inherited method implementation such that
   (a) the method uses different data;
   (b) the algorithm used by the method to access its data is changed.

If subtyping rules are observed, possibility 2a is prohibited, for reasons discussed above. Possibility 1a does not affect, and is not affected by, inherited class invariants. Presumably, the sub-implementation's own invariant and the new method's pre- and post-conditions specify the desired semantics. Methods falling under possibility 1b *are* constrained by inherited invariants, and redefined methods are constrained by the rules for pre- and post-condition redefinition, ensuring that any change in algorithm (possibility 2b) does not reduce the conditions under which the method can be invoked.

At this point, it is instructive to compare the condition-based synchronization strategy to that of "state-based" approaches such as ACT++, Rosette and Synchronizing Actions. Such approaches synchronize an object by having the executing method specify a list of methods accessible in the next object state. While one could argue that pre- and post-conditions are more complex and less intuitive than these specifications, they offer rules for disciplined inheritance of synchronization, and make synchronization dependencies explicit. The designers of Rosette argue that assertion-based techniques are unsuitable in situations where execution of a subclass method puts the object into a state where a particular superclass method cannot be executed (Tomlinson & Singh, 1989). However, by the definition given in section 5.2, such a subclass would not be a subtype of its superclass. Moreover, addition of new methods always requires at least some of the method name lists to be updated, whereas existing pre-conditions are unaffected by new methods.

5.4.2. Concurrent Access Control through Code Reuse. Disciplined inheritance of code also helps the analysis of concurrent access policies, likewise dependent on implementation. Even if a sub-implementation does not change inherited data structures, such policies will often have to be modified. New methods that access inherited data make super-implementation policies incomplete, because these methods must be recognized by the policies. A redefined method algorithm may also invalidate inherited policies. For example, a new implementation of a buffer *dequeue* method may not interfere with the implementation of *enqueue*, meaning that the two methods can now be invoked concurrently. Only methods added by the sub-implementation, accessing only data structures added by the sub-implementation, do not change existing concurrent access policies. Even addition of a method to the super-implementation may invalidate policies of the sub-implementation, unless super-implementation policies are automatically inherited.

Fortunately, concurrent access policies are amenable to subtyping principles. Note that determining if two methods can safely execute concurrently is an example of the readers/writers problem (section 1.3.1): methods that read the same instance variables may be concurrent; any that *modify* (write) instance variables must execute atomically with respect to other readers and writers. The data dependencies of a method can be expressed as:

```
Item dequeue ();
   read:  size, buff;
   write: size, buff;
```

meaning that *dequeue* both reads and writes *size* and *buff*. If a sub-implementation redefines the method's algorithm, previous variable requirements might be dropped:

```
Item dequeue ();
   write: -buff;
```

signifying that *buff* is no longer written, but other data requirements remain unchanged. This corresponds to a weakening of pre-conditions, and may increase possible method concurrencies. The converse, adding new access requirements, is equivalent to a strengthening of pre-conditions, and should only be allowed if it does not introduce reader/writer conflicts in methods that formerly did not have them.

Listing variable access requirements may appear unduly low-level, but it is exactly what the developer must do when deciding which methods can execute concurrently, and makes concurrent access dependencies explicit, avoiding a potential source of error. More importantly, an automated tool can use the information to determine concurrent methods — addition or redefinition of a method only requires the developer to list the data access requirements; the tool can determine possible concurrencies and flag invalid redefinitions (e.g., introducing a readers/writers conflict for formerly concurrent methods).[2] Specification of data access requirements, coupled with use

[2]Automatic determination of the data access requirements themselves is probably not feasible

of such a tool, makes the scheme disciplined: the dependencies are explicit and solutions are automatically inherited, incrementally modifiable and constrained by the dependencies.

5.4.3. Priority through Code Reuse. In some respects, priority is the most difficult concurrency control problem. Though dependent on implementation, priority does not have an obvious criterion for correctness as safe concurrent access does. The goal of a priority scheme is to promote efficient use of the object, usually by increasing request throughput, and the designer of the scheme bears full responsibility for determining how such efficiency can be attained. Unlike synchronization or concurrent access schemes, an incorrect priority scheme will not cause an object to behave incorrectly, merely inefficiently, and this helps obscure the scheme's implementation dependencies. Moreover, the relationship of methods in a priority scheme is not nearly as simple as in a concurrent access scheme; although simple orderings of request types are often used, more advanced schemes exist (Reghizzi et al., 1991) and the lack of coupling between priority and behavioral correctness implies that schemes of arbitrary complexity are possible.

However, many priority conditions can be inherited automatically, and modified without violating existing conditions. For example, prioritizing requests for a particular method relative to other method requests can be specified at the method level:

```
Status enqueue (Item i);
   before: dequeue;
```

meaning that *enqueue* requests should be granted before *dequeue* requests. Often, only one method need be named in each *before* statement, because a transitive ordering of method requests can be automatically determined. Subclassing the implementation and adding code for a new method, with associated *before* statement, will

because of calls to library functions from within the object, etc.

not violate the priority relations of superclass methods — in particular, *enqueue* requests will still have priority over *dequeue* requests, regardless of the priorities of new methods. Note that *changing* an existing priority ordering should not normally be allowed, to preserve disciplined inheritance semantics.

Other priority schemes are also straightforward to express. For example:

```
Boolean in_buff (Item i);
   before: all;
   self_priority: (i <);
```

signifies that *in_buff* has priority over all other method types, and that multiple requests for *in_buff* are prioritized on their parameter values, with lower values conferring higher priority.

Admittedly, adding a new statement type to represent a new priority scheme is inelegant, and incremental modification of predicates (e.g., in the *self_priority* clause) would be difficult to achieve, just as modification of pre- and post-conditions is (section 5.3.1). But given the infinite number of priority schemes that could theoretically be devised, there is probably no one syntax that could represent all of them elegantly.

5.5. Summary

This chapter has shown that the four major types of concurrency control problems — synchronization, concurrent access control, active object properties and priority — can be viewed in a disciplined manner allowing easy recognition of problem dependencies, straightforward solution inheritance and incremental and correct modification of solutions. These goals were met by considering each problem in light of the two components of inheritance: subtyping and code reuse, and applying subtyping rules to both.

Chapter 6 presents an example illustrating the ideas of this chapter and discusses structural issues relevant to a system using those ideas.

CHAPTER 6

Structural Issues

Chapter 4 described the major types of concurrency control problems and their dependencies on class interfaces and implementations, while chapter 5 discussed how to achieve problem solutions that recognize those dependencies. Table 6.1 summarizes the four major problem types, their dependencies, and the solutions proposed for each problem. This chapter presents a complete example of the approach and discusses compilation, runtime and multiple inheritance issues.

Problem	Dependencies		Solution
	Interface	Implementation	
Synchronization	X	X	Method pre- and post-conditions
Concurrent Access Control		X	Inherited data access dependencies
Priority		X	Ordering inferred by info in method headers
Active Object Properties	X		Noted in class interface

TABLE 6.1. Summary of Problems, Dependencies and Solutions

6.1. Example

Figure 6.1 builds on the examples given in chapter 5 to illustrate how the proposed concurrency control policies would be used in practice. The *Buffer* type [Figure


```
type Buffer {
   Item dequeue ();
   Status enqueue (Item i);
   Boolean empty();

   /* Dump buffer [for */
   /* debugging, etc.] */
   Status dump();
}
```

```
impl ArrayImp of Buffer {
    Integer size;
    Item buff[MAX];
    invariant: (0 <= size <= MAX);
 Item dequeue ();
    pre: (size > 0);
    post: (size = old.size - 1);
    read: size, buff;
    write: size, buff;
 Status enqueue ();
    pre:  (size < MAX);
    post: (size = old.size + 1);
    read: size, buff;
    write: size, buff;
    before: dequeue;
 Boolean empty ();
    read: size;
    before: enqueue;
 Status dump ();
    post: buff = old.buff
    read: buff;
    before dequeue;
}
```

(a)

```
type NewBuffer {
   inherits: Buffer;

   Boolean in_buff (Item i) asynch;
}
```

```
impl CircularArrayImp of NewBuffer {
   inherits ArrayImp of Buffer;
   Integer head, tail;
   invariant:
      ((size = 0) and (head = tail))
      or [(size > 0) and
          [((head <= tail) and (size = (tail - head) + 1))
           or ((head > tail) and (size = (MAX - (head - tail) + 1)))]];

 Item dequeue ();
    pre:  (size > 0);
    post: (size = old.size - 1) and
          (head = (old.head mod MAX) + 1);
    read: head;
    write: ~buff, head;
 Status enqueue ();
    pre:  (size < MAX);
    post: (size = old.size + 1) and
          (tail = (old.tail mod MAX) + 1);
    read: tail;
    write: tail;
 Boolean in_buff (Item i) asynch;
    pre:
    post: (size = old.size);
    read: buff, head, size;
    before: dequeue;
    self_priority: (i <);
}
```

(b)

The sub-implementation automatically inherits the concurrency control properties of the super-implementation. Redefined methods must have pre- and post-conditions explicitly stated, and may add or subtract data access requirements if allowable concurrencies are maintained.

FIGURE 6.1. Concurrency Control for Types and Implementations

6.1 (a)], has four methods: *dequeue*, *enqueue* and *empty* are self-explanatory; *dump* is a debugging method, and displays the entire buffer implementation, regardless of how many positions actually have valid data. *ArrayImp* provides a crude array implementation for *Buffer*: after reading the first buffer item, *dequeue* explicitly moves all other items up one position; thus, it must both read and write the array *buff*. The pre-conditions and post-conditions for *dequeue* and the other methods, as well as the class invariant, should be easy to understand. Note that *dump* requires only read access to *buff*. Priority information is also given; this is discussed below.

Type *NewBuffer* [Figure 6.1 (b)] adds only the *in_buff* method, which determines whether a particular item is in the buffer. Because *in_buff* is marked as asynchronous, clients will continue executing immediately after requesting it, collecting the boolean result at some later point. *NewBuffer*'s implementation, *CircularArrayImp*, inherits *ArrayImp* but turns the basic array implementation into a circular array by using *head* and *tail* indices. This requires an addition to the implementation invariant, and redefinition of the *dequeue* and *enqueue* methods. In particular, note that *dequeue* no longer modifies *buff*, because it now updates *head* to indicate the head of the buffer.

The following sections use this example to discuss structural issues of the approach.

6.2. Compilation Issues

6.2.1. Determining Safe Concurrencies and Priorities. For a given implementation, the data that each method accesses determines which methods may execute concurrently (Figure 6.2). Methods calling other methods in the same class also have the access requirements of those methods. If a method is redefined in a sub-implementation, its dependencies are checked to ensure that they respect the rules for pre-condition redefinition. Priority orderings can be determined after allowable concurrencies have been calculated. Note that a priority relationship between two methods in a concurrent object only makes sense if those methods cannot safely execute concurrently — if an ordering is specified for safely concurrent methods,

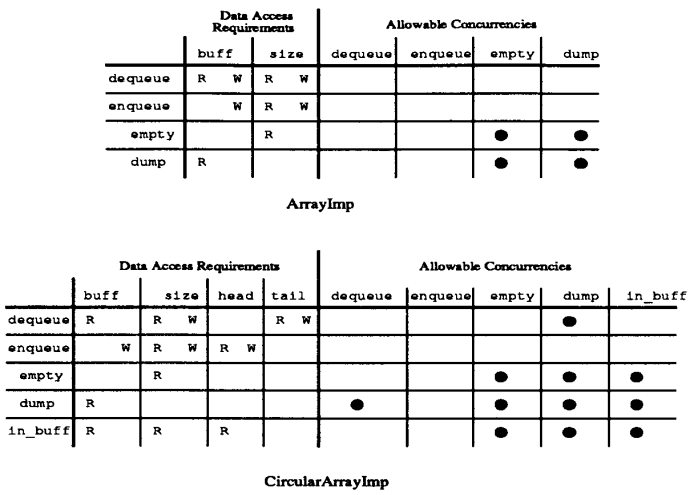


FIGURE 6.2. Data Access Requirements and Method Concurrencies

a warning is issued. Such a warning may also result if two methods that could not safely be concurrent in an implementation can be so in a sub-implementation — when this occurs, the affected priority ordering(s) can be changed. Normally, however, a sub-implementation may not change inherited orderings.

As discussed above, priority orderings may be transitive: all methods without priority information are assumed to have lower priority than those that do; those that have equal priority are subject to a first-come, first-served ordering. Figure 6.3 gives an example of priority orderings.

Relative Priorities	Equal Priorities	Meaningless Priorities
empty > enqueue > dequeue dump > dequeue	enqueue, dump	

ArrayImp

Relative Priorities	Equal Priorities	Meaningless Priorities
empty > enqueue > dequeue	enqueue, dump	dump > dequeue
dump > dequeue	enqueue, in_buff	[dump and dequeue are now safely concurrent]
in_buff > dequeue		in_buff (i1) > in_buff (i2) if (i1 < i2)
in_buff (i1) > in_buff (i2) if (i1 < i2)		[in_buff can be safely concurrent with itself]

CircularArrayImp

FIGURE 6.3. Priorities for Non-Concurrent Methods

6.2.2. Implementations as Types. Because concurrency and priority calculations can be done by automated tools, a profile of the implementation's concurrency and priority properties is easily generated. Use of subtyping principles ensures that these properties will not be violated by sub-implementations. Implementations can thus be used by developers as types are used by programs: a sub-implementation may be safely used in place of its super-implementation. This encourages developers to write implementations that have particular concurrency control properties, because incremental modification of an implementation through inheritance will yield a sub-implementation respecting those properties. Similarly, other developers can choose implementations for types based on the profile of concurrency control properties associated with each implementation. Thus, data structures need not be the only major units of reuse in the implementation hierarchy — a subtyping approach

allows concurrency control properties to be granted a fair degree of permanence.

The benefit of having multiple implementations for a single type, each with a different control policy, is an open question. However, Uhl and Schmid (1990) use a similar approach in a library of abstract data types. Only about ten data types are provided, but each type has nearly a thousand implementation variants, representing all data structures and data structure combinations commonly used to implement the types. Each type also has a nearly identical interface, partially to minimize the user learning curve — Uhl and Schmid claim that this is an important consideration if reuse is to be achieved. Stroustrup (1991) adopts a similar position, arguing that candidates for reuse must be easy to find and understand. Given these requirements, it is unclear what the best way of presenting an implementation's concurrency control profile would be. The format presented in Figures 6.2 and 6.3 is a reasonable first approximation, however.

6.2.3. Verification of Implementations. When an implementation is compiled, the signatures of its methods are verified against those of the type it implements. Verification of implementation invariants, pre-conditions and post-conditions against those of the type is less feasible; Meyer (1988) notes that many assertions can only be realized within the context of an implementation. Similar arguments preclude automatic verification of autonomous or asynchronous properties. Moreover, weakening of pre-conditions and strengthening of post-conditions, both in subtypes and sub-implementations, cannot be feasibly verified at compile-time. However, the proper subtyping of other attributes (priority and concurrent access control) *can* be verified, as noted above.

6.3. Runtime Issues

At run-time, each object has an associated single-threaded controller activated by method termination or request arrival [Figure 6.4]. Requests that cannot immediately be granted are queued. When activated, the controller:

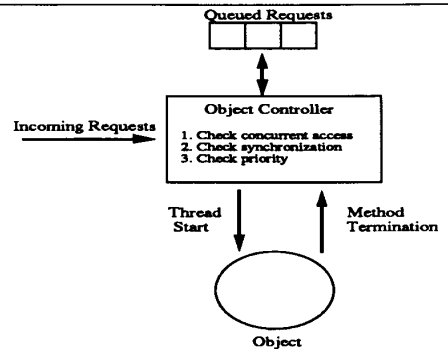


FIGURE 6.4. Object Controller

(1) checks the set of executing methods and the class's concurrent access profile to determine which requested methods could safely be started;

(2) for all such safe methods, checks if the object's synchronization state permits their activation;

(3) for all such synchronized methods, uses method priority ordering to select between those that cannot execute concurrently. Threads are started to execute the requested method with the highest priority, along with all other synchronized methods that can safely be concurrent with it and with each other.

Figure 6.5 gives an example of the algorithm's application. The ordering of the algorithm's steps is not arbitrary. Concurrent access constraints *must* be checked first — not doing so may result in the "instance variable" problem (McHale et al., 1992; Löhr, 1992). As discussed in section 2.2.2.2, method guard predicates should ideally be re-evaluated whenever the attributes they reference are modified, to maxi-

Queued	Executing	Candidates
dequeue, dump, empty, in_buff, enqueue	enqueue	None - no safe concurrencies.

Queued	Executing
dequeue, dump, empty, in_buff, enqueue	None. Buffer full

(a) While enqueue executes, other requests arrive. When enqueue terminates, the buffer is full.

Concurrent Access Candidates	Why
dequeue, dump, empty, in_buff, enqueue	No methods executing.

Synchronization Candidates	Why
dequeue, dump, empty, in_buff	Buffer full.

Highest Priority	Why
dump	(a) dequeue is first in queue --> default candidate. (b) empty > dequeue --> empty becomes candidate. (c) dump is before empty in queue.

(b) A method request is chosen.

Queued	Executing	Why
dequeue, enqueue	dump, empty, in_buff	(a) empty > dequeue. (b) empty and in_buff are safely concurrent with dump and each other.

(c) Threads are started to service the request, and other eligible requests.

FIGURE 6.5. Selecting Method Requests for CircularArrayImp

mize throughput of method requests. This is easily done for synchronization counters, which count the number of executing methods, terminated method executions, etc., but not for instance variables, which may change at unpredictable times. More seriously, the values of instance variables used in the evaluation of a predicate may be immediately invalidated by concurrently executing methods. This prevents McHale et al.'s Scheduling Predicates from allowing concurrency control based on object data, and also affects earlier approaches like Decouchant, Krakowiak, Meysembourg, Riveill, and de Pina's Guide (1989) and Nakajima, Yokote, Tokoro, Ochiai, and Nagamatsu's

Distributed Concurrent Smalltalk (1989). Both languages allow instance variables to appear in method guards, though these cannot be modified by the guards. The guards and other control constructs must also be written, however, to ensure that no executing method will invalidate the values of variables as seen by the guards.

Such problems can be avoided by evaluating concurrent access constraints before constraints involving instance variables, and proceeding no further if the concurrent access constraints are not satisfied. That is why allowable method concurrencies are checked before object synchronization state. Priority orderings, statements of preference rather than correctness, are checked only for requested methods that meet both concurrent access and synchronization requirements.

Active object properties — autonomy and asynchrony — are easily handled by the object controller. Following Löhr's definition, (section 2.2.2.1) autonomous methods issue requests for themselves before terminating, to model continuous behavior. Such requests are subject to the same algorithm applied to other method requests. Asynchronous method requests, allowing clients to continue executing immediately after the request, also require no special handling: asynchronous message passing is transparent at the level of the object controller.

6.4. Multiple Inheritance

Multiple inheritance of types or implementations follows the same strategy used for single inheritance, and generally poses no problems. Each parent has its own concurrency control properties, and these will not interfere in the subclass. If a child overrides a method inherited from one of its parents, this redefinition must respect subtyping rules. If two inherited methods have the same name but different signatures, they can be treated as distinct methods as in C++ (Stroustrup, 1991).

If the methods have the same signature as well, however, the situation is more complex. Renaming one or both methods is probably the most practical solution; this strategy is used by both Eiffel and Portlandish, but has the disadvantage that

sub-implementations may no longer observe subtyping rules — because methods may be renamed, sub-implementations cannot necessarily be used in place of their super-implementations. Alternately, the sub-implementation can provide its own method of the same name and signature, which may call one or both super-implementation methods. The new method should be constrained by the properties of the redefined methods:

- The new method's pre-condition should be the disjunction of the inherited pre-conditions, ensuring that the synchronization properties of the redefined methods are maintained. The new post-condition should be the conjunction of the old post-conditions.
- The new method should be safely concurrent with all methods the redefined methods were.
- Active object properties, if any, of the redefined methods should be maintained.
- The priorities of the redefined methods should be maintained.

Some of these requirements may be impossible to meet. For example, if method m in class X could safely be concurrent with itself, but method m in class Y could not be, what concurrent access policy should m have in a class that inherits both X and Y? Similar problems arise with active object properties: if the m method of only one parent had such a property, then the redefined m can also have that property, but if both inherited ms had conflicting properties (e.g., autonomy and asynchrony), which should the child's version of m have? Finally, although priorities orderings of two redefined methods can generally be respected, this may not be possible if the parents had *more than one* method with the same name and signature. Figure 6.6 illustrates this and the other multiple inheritance issues.

These problems indicate that disciplined inheritance of concurrency control policies is not always possible in the presence of multiple inheritance: although conflicts arising from it can be resolved, this normally involves an arbitrary selection of one

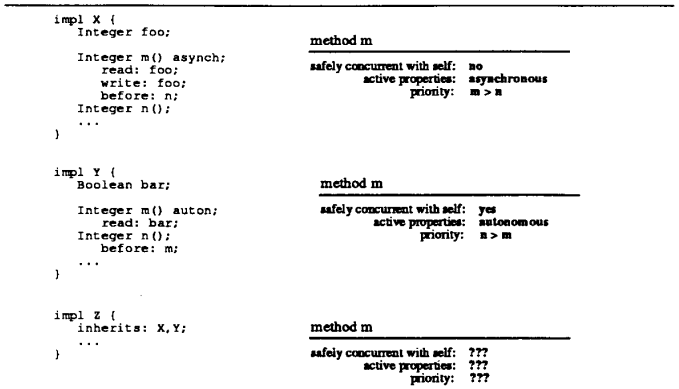


FIGURE 6.6. Multiple Inheritance Problems

of the conflicting properties. On the other hand, such problems are not limited to concurrency control: both the use and usefulness of multiple inheritance is an open question in languages that support it (Cargill, 1992; Meyers, 1992), and some popular languages disallow it (Goldberg & Robson, 1983).

6.5. Summary

This chapter has illustrated how the policies outlined in chapter 5 would be used in practice, and revealed a number of interesting issues. Just as concurrency control policies depend on class interfaces and implementations, they may depend on each other: priority orderings are meaningful only for methods that cannot safely execute concurrently and, to determine the eligibility of method requests at runtime, the

concurrent access policy of an object must be consulted before its synchronization policy. Using automated tools to determine concurrency and priority properties of an implementation, and applying subtyping rules to ensure that those properties are respected by sub-implementations, allow implementations and their concurrency control properties to be written and used as stable entities by developers. Verification of implementation properties against those of the type being implemented may not be feasible, however, and multiple inheritance causes problems not present with single inheritance. On the other hand, neither difficulty is specific to the domain of object concurrency control or the presented approach.

Chapter 7 evaluates the approach using the criteria of chapters 2 and 3 and the requirements for disciplined concurrency control mechanisms discussed in chapter 5.

CHAPTER 7
Evaluation

Chapters 4 through 6 developed an approach to object concurrency control aimed largely at avoiding the failings of previous approaches. This chapter evaluates the approach using the same criteria used on the surveyed approaches, and the criteria used to guide the approach's design. There are thus three major components to the evaluation:

- object-oriented properties (from chapter 2);
- concurrency control properties (from chapter 3);
- disciplined concurrency control mechanism requirements (from chapter 5).

For notational convenience, the approach proposed in this thesis is referred to as DOCC (Disciplined Object Concurrency Control). Using the classification of chapter 2, DOCC objects are *active* (have their own threads of control) and *multi-threaded* (allow more than one thread to be active at a time). DOCC itself is a *homogeneous* approach, because all DOCC objects are active; its policies are *decentralized*, because policies relevant to a method are specified at the level of that method.

7.1. Object-Oriented Properties

Following the format used in chapter 2, the approach's object-oriented properties are assessed from two perspectives: that of *clients* of an object controlled by the approach, and that of class *developers* who must incorporate the approach into an

object, and perhaps later modify it. Recall from chapter 1 that the aspects of the object-oriented paradigm are:

- *Modularity*: The grouping of data structures and the functions that manipulate those structures.
- *Encapsulation*: The use of a well-defined interface that shields clients from class information that may be both complex and subject to change.
- *"As Is" Reusability*: the applicability of a class to more than one specific problem or situation.
- *Inheritance*: The feasibility of a class to be the basis of another class that inherits its properties and may add to them or redefine them.
- *Logical Design Units*: The extent to which a class models a concept in the problem domain.

7.1.1. Modularity. As with most surveyed approaches, DOCC is modular: the descriptions of concurrency control requirements/policies are concentrated at the class or method level, as appropriate. Although one could argue that method pre- and post-conditions do "double-duty" — serving both to synchronize the object and enforce its semantics — neither of these roles is ambiguous, and neither interferes with the other.

7.1.2. Encapsulation. Chapter 2 noted that surveyed approaches violated encapsulation in two ways: one (CEiffel) allowed active object properties, and thus would change the client-visible behavior of objects it was applied to, while others (e.g., Rosette) required that method implementations be modified to include concurrency control code. DOCC does not have this latter requirement, but does support active object properties, and so may alter the visible behavior of formerly-sequential objects. However, chapter 2 noted that such changes are unavoidable if active object properties are to be supported, and thus are not a deficiency of the approach *per se*.

7.1.3. "As Is" Reusability. Because DOCC allows active object properties, it may compromise "as is" reusability from the client's perspective, for reasons noted above.

Reusability from the developer's perspective is more interesting. The concurrency control statements used by DOCC can easily be added to existing classes, and dealt with by a preprocessor — no modification of method code is necessary. However, chapter 2 also considered implementation-independence of control mechanisms as desirable, to minimize the developer's work when the class implementation changes. In light of the last three chapters, this requirement is seen as unachievable, and chapter 2 did note that it would be difficult to realize. On the other hand, DOCC allows implementations to be viewed as stable entities (section 6.2.2), thus easing the developer's job.

7.1.4. Inheritance. Chapter 2 revealed that many approaches either disallow policy inheritance (OWL, POOL, Scheduling Predicates, DRAGOON), may require substantial policy redefinition in subclasses (Hybrid, ACT++, Synchronizing Actions), or offer no guidelines for inheriting policies correctly (Conditional Waits, Rosette, CEiffel). DOCC's concurrency control policies are inheritable and modifiable in a reasonable, disciplined way, and so offer better support for inheritance than any surveyed approach. DOCC also allows multiple inheritance, a feature shared only with CEiffel.

7.1.5. Logical Design Units. Chapter 2 noted that the effect an approach had on a class's power as a logical design unit was largely determined by its support of other object-oriented properties. Some approaches (e.g., Scheduling Predicates and Rosette) also allowed concurrency control policies to be stated in an intuitive way. DOCC shares this property because of the profiles of an implementation's concurrency and priority properties that can be generated by automatic tools.

Modular.			Encapsul.			Reusab.			Inher.			Overall
C	D	A	C	D	A	C	D	A	C	D	A	
100	100	100	80	100	90	80	80	80	100	100	100	93
Abbreviations: C: Client D: Developer A: Average												

TABLE 7.1. Ratings of DOCC's Support for the Object-Oriented Paradigm

Table 7.1 summarizes DOCC's support of the object-oriented paradigm. As in chapters 2 and 3, ratings range from 0 to 100; all ratings were initially set to 100 and decremented in units of 10 based on flaws noted in the discussion.

7.2. Concurrency Control Properties

Chapter 3 outlined a number of criteria for evaluating concurrency control mechanisms:

- *Flexibility*: The mechanism's ability to cope with different problem types, using different information. Problem types include synchronization and priority; information includes resource state, synchronizer state, request type, request times and request parameters. The mechanism's suitability for different concurrent environments is also considered a measure of its flexibility.
- *Development facilitation*: How little the mechanism's solution to one type of problem interferes with those of other types. As well, the comprehensibility of the mechanism from the perspective of the developer who uses classes with the mechanism already in place and from the perspective of the developer who incorporates the mechanism into an object and maintains it.
- *Ease of integration*: The ease with which the mechanism may be incorporated into popular operating systems and object-oriented languages.
- *Efficiency*: The time overhead incurred by having the mechanism control an object; the code space required by the mechanism; the concurrency allowed by the mechanism.

Flexibility		
Environ.	Power	Average
100	90	95

TABLE 7.2. DOCC's Flexibility Ratings

The following sections evaluate DOCC based on these criteria.

7.2.1. Flexibility. DOCC can handle concurrency control problems using all information types: object synchronization uses request type and object state; concurrent access control uses request type and control mechanism state; priority uses request type, request times and request parameters. Chapter 6 noted that the strategy used by DOCC to represent priority schemes makes complex schemes cumbersome to state. However, it should also be noted that the two approaches that allow easier representation of such schemes, DRAGOON and Scheduling Predicates, do not support control mechanism inheritance.

DOCC is not dependent on a specific hardware platform or message-passing scheme. Table 7.2 summarizes DOCC's flexibility rating.

7.2.2. Development Facilitation. DOCC excels at this criterion. Because the four major types of concurrency control problems are treated distinctly, solutions to one can be specified orthogonally to those of another. Some solutions do intrinsically depend on others, but these dependencies can be checked by automated tools. Such tools also make DOCC's solutions easily understandable by developers: those responsible for writing a class's concurrency control policies need specify only a minimal amount of information, because the tools can use this to infer complete policies. Similarly, developers wishing to use an implementation can understand its concurrency control attributes by consulting the profile generated by the tools.

At a higher level, the utility of separating the typing and implementation hierarchies requires further investigation. In particular, developers writing classes must update both hierarchies, and developers wanting to use a class must consult both hierarchies

Development Facilitation					
Modifiab.	Comprehensibility			Encapsul.*	Average
	User	Writer	Avg.		
100	100	100	100	90	97

*from section 7.1

TABLE 7.3. DOCC's Development Facilitation Ratings

to select an implementation for a desired type. How easy this would be in practice is an open question.

Table 7.3 summarizes DOCC's development facilitation abilities.

7.2.3. Ease of Integration. The object controller required by DOCC at runtime does not depend on customized hardware, and uses straightforward algorithms. The separation of types and implementation, though not automatically done in languages like C++ and Eiffel, can be easily accomplished using "handle" classes that represent the class interface and contain an internal pointer to an implementation [c.f., (Meyers, 1992)]. The concurrency control statements used at the class and method levels can be embedded in the comment delimiters used by the language in question, and parsed by a preprocessor (e.g., as done in CEiffel). Although the approach is perhaps best suited to languages like Eiffel that support pre- and post-conditions, these would not be hard to model in other languages — for example, the *assert* macro in C++ plays a similar role.

7.2.4. Efficiency. Changes to a type or implementation will require re-running the automated tools; this can be done as part of the compilation process. The runtime space overhead of the object controller will be roughly equivalent to that of other approaches that use controllers (e.g., ACT++, Rosette, CEiffel); the time overhead for determining if requests can be granted will not be substantial.

The intra-object concurrency provided by DOCC will only be moderate, for three reasons. The first is that instance variables are viewed monolithically, even if methods

do not treat them as such. For example, in the *Buffer* implementations of chapter 6, methods *enqueue* and *dequeue*, having a reader/writer conflict on the array *buff*, could not execute concurrently, even though they would not access the same positions in the array. Such conflict would arise only when the buffer is either full or empty, and in such cases one of the two methods would be unsynchronized. Although both update the integer variable *size* in different ways (one increments it, the other decrements it), neither reads it after modifying it, and so would not be compromised by unexpected changes in its value.

DOCC's approach to concurrent access control does not allow such possibilities — this is the cost of using a simple syntax to represent data access requirements, and of having automated tools that use such information. Although the developer could be given the option of manually specifying allowable concurrencies, this would contradict the philosophy of a disciplined concurrency control policy: "customized" solutions would likely be both dependent on nuances of the method algorithm and difficult to modify incrementally and correctly.

A third, related, reason for disallowing such policies is that they preclude the use of post-conditions. Recall that many post-conditions refer to values instance variables had before the method was executed, e.g.:

```
Item dequeue ();
   post: (size = old.size - 1);
```

If an *enqueue* method is also modifying *size* while *dequeue* is executing, the post-condition can fail. Thus, to realize the semantic checking provided by post-conditions, such concurrencies must be disallowed.

Table 7.4 summarizes DOCC's efficiency ratings, while Table 7.5 gives DOCC's overall score as a concurrency control mechanism.

Efficiency			
Time	Space	Concurr.	Average
80	60	70	70

TABLE 7.4. DOCC's Efficiency Ratings

Flexibility	Development Facilitation	Ease of Integration	Efficiency	Overall
95	97	60	70	81

TABLE 7.5. DOCC's Overall Rating as Concurrency Control Mechanism

7.3. Disciplined Control Mechanism Properties

Chapter 5 presented the following requirements for disciplined concurrency control mechanisms:

- Automatic inheritance of concurrency control policies.
- Explicit statement of the class attributes (interface and/or implementation) that the policies depend on.
- Incremental modification of policies in subclasses.
- Correct modification in subclasses, respecting the dependencies of the policies and essential superclass semantics.

This section discusses how well DOCC's solutions to concurrency control problems meet these requirements.

7.3.1. Synchronization. Synchronization, both in types and implementations, is achieved using pre-conditions and post-conditions. Such conditions have two major strengths: they make synchronization dependencies explicit, and there are well-defined rules for their correct modification in subclasses. Because they may be complex, perhaps nested, predicates, they cannot be incrementally modified by subclasses, and thus also cannot be automatically inherited by a subclass that needs to modify them.

7.3.2. Active Object Properties. Autonomy and asynchrony are interface properties, and thus have no implementation dependencies. Because they are both binary properties, they must be automatically inherited and cannot be incrementally modified. Thus, DOCC's approach of stating these properties in the method header, and prohibiting their redefinition in subclasses, meets all applicable disciplined requirements.

7.3.3. Concurrent Access Control. DOCC represents concurrent access policies in a disciplined manner: data access requirements are explicitly stated at the method level, automatically inherited and incrementally modifiable. Automated tools can be used to both generate the concurrent access policies that the data access requirements allow and to verify that subclass modification maintains existing superclass concurrencies.

7.3.4. Priority. As noted in chapter 5, priority schemes have no obvious criterion for correctness: this obscures their implementation dependencies and makes the correctness of incremental modification difficult to verify. DOCC does ensure, however, that superclass policies are both automatically inherited and not compromised by subclasses.

Chapter 6 discussed problems specific to multiple inheritance that may make disciplined inheritance of concurrency control policies difficult. Given the poor support for multiple inheritance provided by other concurrency control approaches, and the debate surrounding multiple inheritance itself, this should not be seen as a major drawback.

7.4. Conclusions

Table 7.6 summarizes the characteristics of all approaches, along with their support of the object-oriented paradigm and success as concurrency control mechanisms.

Approach	Objects	Threading	Relation	Level	Scores	
					Paradigm Support	Concurrency Control
Owl	P	S	Orth	Decen	73	78
CEiffel	P/A	M	Het	Decen	88	75
Sch. Pred.	P/A	M	Het	Decen	75	85
Frølund	A	M	Het	Decen	86	81
POOL	A	S	Hom	Cen	85	68
Hybrid	A	S	Hom	Decen	86	67
ACT++	A	M	Hom	Decen	89	69
Rosette	P/A	M	Hom	Decen	86	71
Synch. Act.	A	M	Hom	Decen	95	72
Cond. Wait	A	M	Hom	Decen	91	78
DRAGOON	A	M	Hom	Cen	73	78
DOCC	A	M	Hom	Decen	93	81

Abbreviations:
A: Active — S: Single-threaded — Orth: Orthogonal — Cen: Centralized
P: Passive — M: Multi-threaded — Het: Heterogeneous — Decen: Decentralized
Hom: Homogeneous

TABLE 7.6. Summary of Approach Characteristics

DOCC compares favorably with the surveyed approaches. It supports the object-oriented paradigm better than all but one approach (Synchronizing Actions), and is a considerably better concurrency control mechanism than that approach. Likewise, although Scheduling Predicates scores higher than DOCC as a concurrency control mechanism, and Frølund's proposal scores equivalently, DOCC provides better support for the object-oriented paradigm than either of those approaches. These findings vindicate the disciplined approach to object concurrency control developed in this thesis — none of the surveyed approaches can match DOCC's success in reconciling concurrency control and the object-oriented paradigm.

Before moving on, however, it is interesting to compare DOCC to the two approaches that it most resembles: CEiffel and Frølund's proposal. CEiffel (section 2.2.2.1) is an Eiffel extension; it is the first approach to recognize properties specific to active objects (autonomy and asynchrony) and uses pre-conditions (built into the

Eiffel language) for synchronization. Admittedly, both ideas influenced the design of DOCC. However, the annotations used by CEiffel to represent concurrency control policies (active object properties, allowable method concurrencies, etc.) do not tie these policies to the class attributes they depend on and offer no guidelines describing when and how they should be changed in subclasses. Chapters 4 and 5 argued that such failings severely complicate the successful inheritance of concurrency control policies.

Frølund (1992) proposes that concurrency control policies be both automatically inherited and untouchable by subclasses (section 2.2.2.3). This strategy is attractive, because it avoids a failing of DOCC: as discussed above, pre- and post-conditions do not meet all requirements for a disciplined concurrency control mechanism, because they are not incrementally modifiable and consequently, not automatically inheritable either. But if subclasses are prevented from modifying superclass polices, all such policies — no matter how complex — can be automatically inherited. This would appear to greatly simplify the interaction of concurrency control and inheritance.

Nevertheless, there are strong reasons for questioning such an approach. Frølund only sketchily describes it, and incorrectly equates a weakening of pre-conditions with a weakening of constraints. More seriously, prohibiting *any* sort of policy modification also prohibits most implementation changes, a fact Frølund may have been unaware of. The examples discussed in chapters 5 and 6 showed that constrained implementation changes can be desirable; thus, constrained concurrency control policy changes should also be possible.

Chapter 8 discusses the feasibility of representing DOCC in CCS, a process calculus, and describes how such a representation can be used to examine the behavior of the mechanism.

CHAPTER 8
Towards a Verification

Throughout this thesis, more attention has been paid to design and evaluation of design than to implementation. The reasons for this are partly pragmatic: many of the eleven surveyed approaches to object concurrency control differ radically from each other; comparisons and contrasts would have been difficult to achieve without the evaluation criteria discussed in chapters 2 and 3. There is also evidence that focusing on evaluation criteria is particularly beneficial in the concurrency control domain: Bloom (1979), on whose work many of the criteria of chapter 3 are based, argues that the shortcomings of many implemented concurrency control mechanisms arise precisely because comprehensive evaluation criteria were not available to their designers. Given such criteria, Bloom believes that the strengths and weaknesses of mechanisms can be assessed primarily by referring to their designs — implementations are secondary.

This thesis bears out Bloom's arguments: evaluation of existing approaches revealed that all had flaws and, in many cases, the same flaws. Knowledge of such problems guided the design of DOCC, and this knowledge would not have been gained without evaluation criteria. But although these criteria were also used to evaluate DOCC, questions about its actual performance still remain. DOCC's evaluation determined the general strengths and weaknesses of its control mechanisms, but could not reveal specific problems with mechanism algorithms — especially in complex sys-

tems, such problems are often subtle and may only occur when different parts of the system interact. If system components are concurrent, the overall behavior becomes even more complex. Simulation frameworks may allow the behavior of such systems to be investigated without resorting to implementation and testing.

Milner's Calculus of Communicating Systems (CCS) is such a framework. This chapter examines the feasibility of simulating DOCC and its behavior in the calculus: section 8.1 introduces the calculus and shows how it can be used to model a buffer object. Sections 8.2 through 8.6 discuss how well the calculus can represent concurrency control and object-oriented properties.

8.1. Representing a Buffer

The Calculus of Communicating Systems (Milner, 1989) is a process calculus allowing the behavior of concurrent entities to be represented and reasoned about. The calculus may be well-suited to the domain of object concurrency control: Nierstrasz (Nierstrasz & Papathomas, 1990; Nierstrasz, 1992) is developing a calculus based on CCS to represent concurrent, object-oriented languages. This thesis uses the standard calculus for two reasons: it is fully-developed and a simulator, the Edinburgh Concurrency Workbench (Moller, 1991), is available to automatically reason about the behavior of systems represented in it. This section introduces CCS by showing how a version of the buffer object used in earlier chapters can be modeled.

Consider the following definition:

(1) $$Buffer \stackrel{def}{=} enqueue.\overline{status}.Buffer$$

This defines an *agent*, *Buffer*, that can perform two *actions*, *enqueue* and $\overline{status}$, then revert back to its original state. The agent models the *visible behavior* of a simple buffer — *enqueue* is an *input* action, representing an enqueue request arriving at the buffer. $\overline{status}$ is barred to indicate an *output* action — the returning of the completion status of the enqueue operation. The details of how the enqueue request

is handled are hidden; this is one way in which CCS encapsulates behavior of agents, and thus supports the object-oriented paradigm.

A client of the buffer can be defined as follows:

(2) $$Client \stackrel{def}{=} \overline{enqueue}.status.Client$$

This can communicate with the buffer because it has the appropriately named actions (e.g, output action $\overline{enqueue}$ matches the buffer's input action *enqueue*). Such action pairs represent a "hand-shake" or synchronization point between the two agents. A system consisting of the buffer and its client can be modeled as follows:

(3) $$Sys \stackrel{def}{=} (Buffer \mid Client) \setminus \{enqueue, status\}$$

Putting the agents inside parentheses, separated by vertical bars, define a *composition*, a new agent in which the two agents act concurrently. The actions in curly braces following the backslash are *restricted* to the new system — outside agents are prevented from performing an *enqueue* or *status* hand-shake with either of the system's components, meaning that only those components can perform the hand-shake. Such an "invisible" hand-shake is called a *tau* (τ) action. Thus, although *Client* repeatedly enqueues items into *Buffer*, the system has no externally visible behavior.

A more powerful buffer object can be created by first defining agents for some of the buffer methods described in chapters 5 and 6:

(4) $$Enqueue \stackrel{def}{=} enqueue.\overline{status}.Enqueue$$

(5) $$Dequeue \stackrel{def}{=} dequeue.\overline{value}.Dequeue$$

(6) $$Dump \stackrel{def}{=} dump.\overline{guts}.Dump$$

The buffer is simply the concurrent composition of these methods:

(7) $$AbstractBuffer \stackrel{def}{=} (Enqueue \mid Dequeue \mid Dump)$$

The buffer represents an abstract interface because it has no concurrency control (all methods may execute concurrently) and no synchronization constraints. This example shows that CCS allows *has-a* relationships among objects: the *Buffer* class contains "objects" that represent its three methods.[1]

8.2. Adding Concurrent Access Control and Synchronization

If the buffer is implemented by the *ArrayImp* that was used in Figure 6.1, none of three methods defined in the interface can safely execute concurrently with the other two, and only the *dump* method can safely execute concurrently with itself. Such constraints are represented by the following agents:

$$
\begin{aligned}
(8)\quad ArrayImp &\stackrel{def}{=} DumpAI + EnqueueAI + DequeueAI \\
DumpAI &\stackrel{def}{=} dump.(\overline{guts}.ArrayImp + dump.\overline{guts}.\overline{guts}.ArrayImp) \\
EnqueueAI &\stackrel{def}{=} enqueue.\overline{status}.ArrayImp \\
DequeueAI &\stackrel{def}{=} dequeue.\overline{value}.ArrayImp
\end{aligned}
$$

Substantial changes were necessary. The agents acting as operands to the + operator represent *branches*. A transition from the *ArrayImp* agent will follow only *one* branch; after the choice has been made, all other branches are discarded. This models the desired non-concurrency of the three methods, but has another implication: the three "method" agents had to be modified to become an *ArrayImp* agent after performing their respective operations. If the original definitions were used, the *ArrayImp* agent could get into states where it could only repeatedly perform *dequeues*, *dumps* or *enqueues*.

Dump methods can safely execute concurrently, and this is crudely simulated by the *DumpAI* agent: after receiving a *dump* request, *DumpAI* may either reply to it and evolve back to the buffer implementation, or accept another *dump* request and reply to both requests before becoming the buffer again. In principle, any number of *Dump*

[1] The *empty* method is not modeled here for simplicity's sake.

agents could be active, but CCS has no means of representing the dynamic creation of agents, and thus, no way of representing the creation of new method threads.

Given the representation of concurrent access policies, adding synchronization to the buffer implementation is straightforward. Different agents can be used to represent different synchronization states — the first agent, for example, would represent an empty buffer, and thus would have no *dequeue* method associated with it. Its *enqueue* branch would take it to a state where a *dequeue* branch would be provided. Using this strategy, a two-cell buffer can be represented as follows:

$$
\begin{aligned}
(9)\quad ArrayImp0 &\stackrel{def}{=} DumpAI0 + enqueue.\overline{status}.ArrayImp1 \\
ArrayImp1 &\stackrel{def}{=} DumpAI1 \\
&\quad + enqueue.\overline{status}.ArrayImp2 + dequeue.\overline{value}.ArrayImp0 \\
ArrayImp2 &\stackrel{def}{=} DumpAI2 + dequeue.\overline{value}.ArrayImp1
\end{aligned}
$$

The *dump* agents (*DumpAI0*, etc.) are assumed to evolve back to their respective buffer implementations, since they do not change the buffer state. This example shows that CCS models the synchronization policy of an object through state transitions, and therefore resembles approaches like ACT++ and Rosette more than it does DOCC. However, analogies can be drawn between DOCC's synchronization scheme and that offered by CCS:

- *Invariants*: are implicitly represented in agents that evolve back to their starting points (e.g., the buffer methods in equations 4 through 6), because such evolution indicates that the sequence of inputs and outputs performed by the agent does not change the agent. Agents that evolve to other agents do not have implicit invariants.
- *Pre-conditions*: are represented by hand-shakes (matching of input and output offers) between agents. For example, a client of the *ArrayImp0* agent would be blocked if it tried to dequeue an item from the buffer, because the needed handshake (*dequeue*) is not available. Object synchronization policies, repre-

sented by transitions from one agent to another, are effectively built up from these "lower-level" handshakes that model method synchronization.

- *Post-conditions*: are implicitly represented in an agent that evolves to another agent (e.g., the "enqueue" branch of *ArrayImp0*), because the new agent represents the state that holds after the old agent's sequence of inputs and outputs is performed.

CCS's adoption of state-based synchronization is largely due to the encapsulation provided by agents — because only the input and output offers made by an agent are considered its visible behavior, the internal state information needed to precisely model pre-conditions, post-conditions and invariants is not available.

8.3. Representing Priority

CCS is not designed to model priority schemes, and this can be easy illustrated. Consider the following agents.

$$
\begin{aligned}
Enqueue &\stackrel{def}{=} enqueue.\overline{status}.Enqueue \\
Client1 &\stackrel{def}{=} \overline{enqueue}.status.\overline{client1}.Client1 \\
Client2 &\stackrel{def}{=} \overline{enqueue}.status.\overline{client2}.Client2 \\
Client3 &\stackrel{def}{=} \overline{enqueue}.status.\overline{client3}.Client3 \\
Unfair &\stackrel{def}{=} (Enqueue \mid Client1 \mid Client2 \mid Client3) \backslash \{enqueue, status\} \qquad (10)
\end{aligned}
$$

Because the *enqueue* and *status* actions are restricted in the *Unfair* agents, its visible actions include only $\overline{client1}$, $\overline{client2}$ and $\overline{client3}$. The *Edinburgh Concurrency Workbench* (Moller, 1991) is a program that allows the behavior of CCS agents to be investigated. For example, the *visible sequences* of three actions that the *Unfair* agent can generate include the following:

```
=== 'client1 'client1 'client1 ===>
=== 'client1 'client1 'client2 ===>
=== 'client1 'client1 'client3 ===>
=== 'client1 'client2 'client3 ===>
```



```
=== 'client2 'client1 'client2 ===>
=== 'client2 'client2 'client2 ===>
=== 'client3 'client3 'client3 ===>
```

(Output actions are prefixed with a comma (e.g., 'out1)).

Most of these sequences clearly show that the *Unfair* agent does not model fairness — the same client could handshake with the *Enqueue* agents two or more times in a row. This is not a mistake; such behavior should be expected in concurrent systems composed of largely independent agents. However, CCS has no syntax for indicating an ordering among agents (e.g., that client 1 has priority over clients 2 and 3); If there is more than one agent in the system that is ready to perform an action (a visible action or a hand-shake with another agent), it is impossible to know *which* agent will be the next to do so.

The implications of this are immediate: CCS cannot model priority schemes well. More precisely, the author knows of no way to represent first-come, first-served schemes or orderings based on request type in CCS, for the reasons given above. Surprisingly, a more complex scheme *can* be modeled by the calculus. *Alternation* (Reghizzi et al., 1991) between two agents can be represented as follows:

$$
\begin{aligned}
A &\stackrel{def}{=} prea.\overline{a}.posta.A \\
B &\stackrel{def}{=} preb.\overline{b}.postb.B \\
Env &\stackrel{def}{=} \overline{prea}.\overline{posta}.\overline{preb}.\overline{postb}.Env \\
Alt &\stackrel{def}{=} (A \mid B \mid Env) \backslash \{prea, posta, preb, postb\} \qquad (11)
\end{aligned}
$$

The visible behavior of the *Alt* system will be a sequence of alternating $\overline{a}$s and $\overline{b}$s. The *environment* (*Env*) is needed to impose the desired ordering — each visible action has to be enclosed by "pre" and "post" actions that interact with the environment. This solution is both verbose and non-intuitive; although similar strategies might allow other priority orderings to be represented, the resulting models would probably be more complex and even less elegant.

Although CCS does support value-passing and conditional expressions, the "value-passing" calculus is difficult to use in the basic calculus supported by the Concurrency Workbench. This makes concurrency control constraints based on parameter values, for example, hard to model, and further limits CCS's ability to represent priority schemes.

8.4. Representing the Object Controller

Recall from chapter 6 that the each object in DOCC has an associated controller that implements its concurrency control policies. The controller is single-threaded, and activated by the arrival of a method request or the termination of an executing method thread. When active, the controller checks the object's concurrent access control, synchronization and priority policies, in that order, and starts threads to service all waiting method requests that those policies allow. Requests that cannot be serviced are queued.

8.4.1. Restrictions. Only portions of the controller's behavior can be represented in CCS. The main features that cannot be modeled are:

- *The queue of waiting requests.* Basic CCS's poor ability to represent value-passing would make the enqueuing and dequeuing of different request types difficult. More seriously, CCS's lack of fairness would preclude any ordering among queue elements, just as it prevents other priority schemes. (Recall that requests with the same priority level are effectively ordered by their positions in the queue.)
- *Checking of policies.* The representations of synchronization and concurrent access control discussed above both ultimately rely on hand-shakes; lack of a hand-shake needed by a method request is what enforces synchronization or concurrent access policies. However, this also effectively suspends the request until the needed hand-shake is available. Thus, the controller cannot "check" if a policy allows a request and then take some action based on the result, because doing the check may suspend it.
- *Lack of Dynamic Agent Creation.* As noted earlier, CCS cannot model the dynamic creation of agents. This prevents elegant simulation of the controller's ability to start an arbitrary number of method threads.

8.4.2. Representation. Given these restrictions, modeling of the controller proceeds as follows. First, some clients of the ArrayImp buffer are defined; each has a visible input action (e.g., *enqueue*) and a visible output action to indicate the completion of its request (e.g., $\overline{enqdone}$):

$$EnqC \stackrel{def}{=} enqueue.\overline{enqstart}.status.\overline{enqdone}.EnqC \quad (12)$$

$$DeqC \stackrel{def}{=} dequeue.\overline{deqstart}.value.\overline{deqdone}.DeqC \quad (13)$$

$$DumpC \stackrel{def}{=} dump.\overline{dumpstart}.guts.\overline{dumpdone}.DumpC \quad (14)$$

The buffer object controller requires a *receiver* to handle requests from buffer clients. Each request is received by a different branch to simulate that the controller is single-threaded. Each branch starts a thread to deal with the request, then evolves back to the receiver agent. Using threads to actually handle the request prevents the handler from blocking if the request cannot be granted.

$$\begin{aligned} Receiver \stackrel{def}{=}\ & enqstart.\overline{enqthread}.Receiver \\ +\ & deqstart.\overline{deqthread}.Receiver \\ +\ & dumpstart.\overline{dumpthread}.Receiver \end{aligned} \quad (15)$$

$$EnqT \stackrel{def}{=} enqthread.\overline{doenq}.EnqT \quad (16)$$

$$DeqT \stackrel{def}{=} deqthread.\overline{dodeq}.DeqT \quad (17)$$

$$DumpT \stackrel{def}{=} dumpthread.\overline{dodump}.DumpT \quad (18)$$

Each thread tries to handshake with the buffer by performing the appropriate output action (e.g., $\overline{doenq}$) — this is further discussed below.

The controller is simply the composition of the Receiver and the threads, with handshake actions restricted.

$$Controller \stackrel{def}{=} (Receiver \mid EnqT \mid DeqT \mid DumpT) \backslash \{enqthread, deqthread, dumpthread\} \quad (19)$$

A "buffer system" can be represented as the composition of a buffer implementation, the controller, and the buffer clients:

$$BuffSys \stackrel{def}{=} (ArrayImp0\,[doenq/enqueue, dodeq/dequeue, dodump/dump] \mid Controller \mid EnqC \mid DeqC \mid DumpC) \backslash \{doenq, status, enqstart, dodeq, ..., dodump, ...\} \quad (20)$$

The buffer implementation has its inputs *relabeled* so that they can handshake with the controller's threads (e.g., *enqueue* is relabeled *doenq*). All of the actions that hand-shake are restricted (made invisible). Greater possible concurrency could have been simulated by having more than one copy of each thread and client type, but for simplicity this was not done. Some of the system's visible sequences of length four are:

```
=== dequeue dump 'dumpdone dump ===>
=== dequeue dump enqueue 'deqdone ===>
=== dump 'dumpdone dump 'dumpdone ===>
=== enqueue dequeue 'deqdone dequeue ===>
=== enqueue 'enqdone dump enqueue ===>
```

Note the lack of fairness, but also that a dequeue request cannot be serviced until an enqueue request has begun.

8.5. Inheritance

In chapter 6, a *CircularArrayImp* implementation of buffer inherited from *ArrayImp* and added a new method, *in_buff*:

$$InBuff \stackrel{def}{=} in\_buff.\overline{answer}.InBuff \quad (21)$$

In_buff could safely execute concurrently with both *dump* and itself. As well, the method algorithms in *CircularArrayImp* were changed such that *dump* and *dequeue* could execute concurrently. The general form of the *CircularArrayImp* agent would thus be:

$$\begin{aligned} CircularArrayImp &\stackrel{def}{=} ConcThreads + Enqueue \\ ConcThreads &\stackrel{def}{=} dump.(\overline{guts}.CircularArrayImp \\ &\quad + dequeue.\overline{guts}.\overline{value}.CircularArrayImp \\ &\quad + in\_buff.\overline{guts}.\overline{answer}.CircularArrayImp \\ &\quad ...) \\ &\quad + in\_buff.(\overline{answer}.CircularArrayImp \\ &\quad\quad + dump.\overline{answer}.\overline{guts}.CircularArrayImp \\ &\quad\quad ...) \\ &\quad ... \end{aligned} \quad (22)$$

Although there are conceptually few differences between *CircularArrayImp* and *ArrayImp*, the CCS definition of the new implementation is very different from that of the old (equation 8). The syntax of CCS prevented the changed component relationships (compatibility of dequeue requests with dump requests) and the addition of new methods (*in_buff*) from being represented as incremental modifications to the original agent — a complete restatement was necessary.

The above equation also demonstrates how cumbersome the representation of method concurrencies can be — the *ConcThreads* agent does not cover all the possible concurrencies, but is hard to understand even as is. Note that such problems do not arise when modeling objects that allow all methods to be safely concurrent — such objects can be represented by agents like that in equation 3.

8.6. Active Object Properties

Representation of client asynchrony is trivial in the calculus. For example, assume that the *InBuff* agent (equation 21) allows such asynchrony; its client could be defined as follows:

$$InBuffClient \stackrel{def}{=} \overline{in\_buff}.\overline{stuff}.answer.InBuffClient \quad (23)$$

Where $\overline{stuff}$ represents some action the client takes between the time it makes the request and the time it collects the answer.

An autonomous method — one that continually issues requests for itself — is also easy to model. Effectively, the autonomous method is treated as a client of itself, as shown below:

$$\begin{aligned} AutoMethod &\stackrel{def}{=} \overline{req\_auto}.do\_auto.\overline{did\_auto}.AutoMethod \quad (24)\\ AutoReceiver &\stackrel{def}{=} req\_auto.\overline{auto\_thread}.AutoReceiver\\ AutoThread &\stackrel{def}{=} auto\_thread.\overline{do\_auto}.AutoThread\\ AutoSys &\stackrel{def}{=} (AutoMethod \mid AutoReceiver \mid AutoThread)\\ &\quad \backslash\{req\_auto, auto\_thread, do\_auto\} \end{aligned}$$

The visible behavior of *AutoSys* will be a sequence of $\overline{did_auto}$s.

8.7. Results of the Representation

Despite the limitations of CCS, it could be used to explore aspects of DOCC's behavior. The agents described in this chapter were loaded into the concurrency workbench, and various tests performed on them. For example, the "visible sequences" described earlier were generated from the workbench, and indicate that the agents' behavior is as expected. More importantly, the workbench's "find deadlocks" command revealed that none of the agents can get into a deadlock state (one in which no further actions are possible). This is an encouraging result, because absence of deadlock is by no means a given — CCS model checking has revealed such potential in

both electronic mail systems (Brebner, 1991) and the Macintosh threads and futures package (Kittlitz, 1991).

The behavior of the buffer system (equation 20) was explored in more detail. First, the agent representing the buffer system was *minimized* from its original size of 511 possible states to an agent with equivalent visible behavior having only 32 states. Then, the state space was examined using predicates of the *μ-calculus* (Kozen, 1983). A discussion of the calculus is behind the scope of this thesis, but the following examples give an overview of its capabilities. Consider the following predicate:

```
BOX (<->T)
```

This checks for deadlock. *BOX* is a macro such that `BOX P` means that for all states, `P` holds. In this case, `P` is `<->T`, meaning that some action is possible. Thus, if the whole predicate is true (which it is), then the system can make a move from every reachable state; that is, it cannot deadlock.

Another useful check is for *livelock* — a state in which the system can do nothing but invisible (τ) actions. The predicate:

```
~POSS(max.(X<t>.X))
```

was true, indicating that the system cannot livelock. The macro *POSS* means "there is a possible state in which." The workbench uses the "t" character to represent τ, so `max.(X<t>.X)` isolates all cycles of states which can be traversed only by internal handshakes (i.e., with no intervening input or output actions). However, applying `~POSS` to this resulted in a true predicate, meaning that the buffer system had no such possible cycles.

Other predicates are much simpler. For example:

```
NEC_FOR enqueue 'enqdone

CYCLE2 enqueue 'enqdone
```

help verify the integrity of the system. *NEC_FOR* checks that the action given as its first argument is a necessary condition for the action given as its second argument to occur. *CYCLE2* verifies that the two actions given as its arguments always occur in the specified order.

Finally, that the buffer system cannot guarantee fairness is illustrated by the predicate:

```
BOX [enqueue] (EVENT <'enqdone>T)
```

which states that "after all states in which an *enqueue* is performed, an $\overline{enqdone}$ will eventually be performed." This is not true — after a client makes an *enqueue* request, the controller may service a potentially infinite number of other request types. However, the weaker predicate:

```
BOX [enqueue] (BOX POSS <'enqdone>T)
```

is true, meaning that after an *enqueue* is performed, it is always possible to do an $\overline{enqdone}$.

8.8. Summary

Table 8.1 summarizes basic CCS's strengths and weaknesses as a framework for simulating concurrency control in object-oriented systems. Reasonable support is provided for most properties, but the lack of incremental agent modification and the poor support for some types of concurrency are irritating, and the lack of a means of specifying priority is a major drawback. As discussed earlier in the chapter, the absence of an action ordering prevents most sorts of priority schemes from being represented in the calculus, and also prevents the modeling of simple constructs like queues.

Property	Comments
Modularity	Yes — agents are modular.
Encapsulation	Yes — notion of visible vs. invisible action intrinsic to CCS. Restriction also useful.
Inheritance	No — but models "has-a" relationships well.
Synchronization	Yes — based on state transitions. Some similarity to pre-conditions, post-conditions and invariants.
Concurrent Access Control	Yes — modeled by composition vs. branching, but resulting agents may be complex.
Priority	No — made impractical by lack of fairness and cumbersome value passing syntax.
Active Object Properties	Yes — easily represented.
Dynamic Thread Creation	No — but can be simulated.

TABLE 8.1. Summary of CCS's Suitability to the Problem Domain

Other constraints forced the CCS representation of DOCC to be somewhat inverted — the synchronization and concurrent access policies of an agent were effectively built into its definition rather than that of its controller, and the threads used by agent controllers were responsible for both *checking* if a request could be granted, and for *invoking* the appropriate agent method if so. Such modifications were necessary largely because of the hand-shake/blocking paradigm used for communication between agents (section 8.4).

However, representing DOCC in CCS was still worthwhile — the predicates shown in section 8.7 helped clarify the system's behavior. As CCS's popularity continues to grow, and extensions for concurrent, object-oriented modeling are developed, its power as a verification tool will also increase.

CHAPTER 9

Conclusions and Future Work

This thesis had two major goals:

- An assessment of the merits of the object-oriented programming paradigm, using concurrency control as problem domain.
- The development of design principles allowing approaches to object-oriented concurrency control to fully support the paradigm.

This chapter presents the results of pursuing those goals. Section 9.1 discusses this thesis's findings on the strengths and weaknesses of the object-oriented programming paradigm, while section 9.2 summarizes desirable design principles for object-oriented concurrency mechanisms. Section 9.3 discusses the use of evaluation criteria and verification tools. Section 9.4 outlines areas of future work.

9.1. Object-Oriented Programming

Chapter 3 concluded that using the object-oriented programming paradigm is neither necessary nor sufficient for successful concurrency control. Some surveyed approaches offered good paradigm support but had poor concurrency control abilities, or vice-versa. Although some approaches did perform both roles well, none completely reconciled them. Failings differed from approach to approach, but inheritance of concurrency control policies seemed especially problematic.

Later chapters revealed the major reason for such problems: two attributes of the paradigm, encapsulation and the distinction made between interfaces and implementations, are ill-suited to the domain of concurrency control. A concurrency control mechanism cannot be encapsulated from the controlled object, nor can it be used as is with different implementations of the same class interface. Few approaches recognized these constraints, and thus could not provide control mechanisms that meshed well with the object-oriented paradigm.

The above arguments also indicate, however, that concurrency control may have been a poor test domain. If a concurrency control mechanism is in some sense a "by-product" of the object it controls, the relationship of that mechanism to the object cannot be considered a reasonable platform for evaluating the object-oriented paradigm. Properties such as encapsulation and implementation-independence refer primarily to how clients see the object, not necessarily to how parts of the same object see each other!

Nevertheless, the fact that many approaches tried to incorporate these properties into object concurrency control mechanisms suggests that better guidelines for the use of these properties, and perhaps the paradigm as a whole, have to be developed. This thesis has shown that attributes of the paradigm are not intrinsically desirable, but should be considered in light of the problem domain.

Other aspects of the paradigm indicate immaturity. Although the differences between subtyping and code reuse are now commonly recognized, most object-oriented languages do not treat the two types of inheritance separately. Not doing so may add unnecessary complexity — certainly, it seems to in the domain of object concurrency control. Multiple inheritance was also shown to cause unexpected complexities (chapter 6), and is still a subject of active debate in the literature.

Given these observations, one must conclude that the worth of the object-oriented programming paradigm has yet to be determined. Though many of the paradigm's aspects are intuitively appealing, others are neither mature nor well-understood. Until this situation changes, a thorough evaluation is impossible.

9.2. Object-Oriented Concurrency Control

9.2.1. Problem Types and Problem Dependencies. The survey and evaluation of approaches to object-oriented concurrency control done in chapters 2 and 3 showed that approach capabilities varied widely. Chapter 4 built on these results and found that the approaches were designed to solve one or more distinct types of problems:

- *Synchronization*: The determining of whether method requests can be accepted by the object in its current state.
- *Concurrent Access Control*: The denial or blocking of requests for methods that have readers/writers conflicts with executing methods.
- *Priority*: An ordering of method requests to increase efficient use of the object.
- *Active Object Properties*: The implementation of properties such as method autonomy and client asynchrony that only apply to active object systems.

Chapter 4 also determined that all such problems depend on the interfaces and/or implementations of the class in question. Concurrent access control and priority are implementation properties — their effects need not be visible to clients of the object. A class's synchronization policy may be partly stated in its interface, but also partly dependent on implementation details. Active object properties, having client-visible effects, must be stated in the class interface. Although inheritance can modify both interfaces and implementations, and thus invalidate existing concurrency control policies, few approaches recognized this. The lack of recognition is largely attributable to aspects of the object-oriented paradigm, as discussed above.

9.2.2. A Disciplined Approach. To reconcile concurrency control and inheritance, chapter 5 proposed requirements for *disciplined* object-oriented control mechanisms. The syntax of such mechanisms should make the interface and/or implementation dependencies of concurrency control policies explicit. Policies should be automatically inherited by subclasses, and incrementally modifiable to match the changes

in their dependencies that inheritance might cause. Such modification should not be arbitrary, but constrained by the dependencies.

Splitting the concept of inheritance into subtyping and code reuse separated the two dependencies of concurrency control mechanisms. Subtyping, which preserves behavior by requiring that subtypes be usable in the contexts that their supertypes are, was chosen as a model for disciplined concurrency control mechanisms. Applying subtyping principles to the code reuse hierarchy led to the conclusion that data structures should be preserved from implementation to sub-implementation, while algorithm changes, constrained by subtyping rules, should be allowed.

The following solutions to concurrency control problems were proposed:

- *Method pre-conditions and post-conditions*, to realize synchronization policies. Pre-conditions specify the conditions under which a method may invoked; post-conditions specify the conditions allowable after it terminates execution. In accordance with subtyping principles, the pre-condition of a redefined method may be weaker than the original, and the post-condition stronger; this permits constrained implementation changes, but also prevents the conditions, often implemented as complex predicates, from being automatically inheritable or incrementally modifiable.
- *Data access dependencies*, stating the instance variables a method needs to access or modify. Using this information, an automated tool can determine which methods can safely execute concurrently. Data access dependencies can be incrementally modified in sub-implementations to reflect code changes; the validity of modification can be checked by the tool.
- *Simple priority relations* at the method level, indicating the priority of the method relative to one or more other methods. An automated tool can determine priority orderings based on this. Sub-implementations may add orderings for new methods, but these will not affect inherited orderings.

- *Statement of active object properties* in the class interface. Because such properties are binary, they cannot be incrementally modified by subclasses.

Chapter 6 discussed how these solutions would interact at compile-time and at run-time, and revealed dependencies between the four types of concurrency control problems: priority schemes are only sensible for methods that cannot safely execute concurrently, and concurrent access policies must be checked first at runtime to ensure correct evaluation of synchronization policies. The ability of automated tools to determine the concurrent access and priority policies of a class based on information in the implementation led to the viewing of implementations as types having specific concurrency control profiles. Developers can select an implementation for a type depending on the desired concurrency control policies; using subtyping rules on the implementation hierarchy ensures that sub-implementations will maintain (or improve) those policies.

9.2.3. Conclusions. For the most part, this thesis has succeeded in its goal to develop design principles for concurrency control mechanisms that fully support the object-oriented paradigm. The requirements for disciplined control mechanisms allow concurrency control to be reconciled with inheritance — seemingly the most difficult task — and the concurrency control solutions proposed largely meet those requirements. There are two primary weaknesses: method pre- and post-conditions cannot be incrementally modified and complex priority schemes cannot be easily represented. These issues are further discussed in section 9.4.

9.3. Evaluation and Verification

This thesis has demonstrated the importance of evaluation criteria. The criteria used to evaluate the surveyed approaches in chapters 2 and 3 provided a context in which the approach strengths and weaknesses could be compared and contrasted. Without such a context, it is unlikely that the idea of disciplined object-oriented concurrency control could have been developed. The numeric ratings of the approaches

based on the criteria were somewhat arbitrary, but the author knows of no survey of object-oriented concurrency control approaches that is more rigorous — though the evaluation was imperfect, it proved extremely useful nonetheless.

The representation of this thesis's approach in CCS (chapter 8) was motivated by a similar philosophy. Given the considerable time that would have been required to develop a test implementation, more cost-effective evaluation approaches were desirable. CCS's lack of priority ordering prevented a complete representation of the system, but in many other respects it was quite well-suited to the domain. As more powerful extensions to the calculus are developed, it will become increasingly useful.

9.4. Future Work

The preceding sections have hinted at areas of future work. This section discusses these.

Perhaps the most important issue left unresolved is the practicality of a subtyping/code reuse hierarchy split. Such a split would seem to increase the complexity faced by the developer, because necessary components of a class would be in two places — techniques for matching types and implementations would be necessary for both easily navigating the hierarchies and for keeping them synchronized. In the domain of concurrency control, the usefulness of potentially having multiple implementations of a single type, differing mostly in concurrency control policies, needs to be explored.

The concurrency control approach developed in this thesis also suggests areas of future work. In particular, algorithms for incrementally modifying predicates (possibly complex and/or nested) would allow synchronization policies to be automatically inherited, and might also be applicable to CCS — recall that the calculus does not model inheritance well, partially because agent definitions cannot be incrementally modified. Investigation of priority schemes is also warranted, to determine if reasonable correctness conditions for priority exist. Such conditions could allow priority

policies to be stated and inherited more elegantly than is done in this thesis.

Finally, the suitability of evaluation and verification as replacements for test implementations could be investigated by comparing the results of an evaluation/verification to those of an implementation. This is an important issue — as concurrent systems become more and more complex, verification tools may be the only way to efficiently investigate their behavior.

Bibliography

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press.

America, P. (1987). Inheritance and Subtyping in a Parallel, Object-Oriented Language. In *Proc. ECOOP '87 (LNCS 276)*, pp. 234–242. Springer-Verlag.

America, P., & van der Linden, F. (1990). A Parallel, Object-Oriented Language with Inheritance and Subtyping. In Meyrowitz, N. (Ed.), *Proc. OOPSLA/ECOOP '90*, pp. 234–242. Addison-Wesley.

Apple Computer (1991). *Inside Macintosh, Volume VI.* Apple Computer, Inc.

Aschmann, H., Giger, N., Hoepli, E., Janak, P., & Kirrmann, H. (1991). Alphorn: A Remote Procedure Call Environment for Fault-Tolerant, Heterogeneous, Distributed Systems. *IEEE Micro, 11*(5), 16–19, 60–67.

Barry, B., Altoft, J., Thomas, D., & Wilson, M. (1987). Using Objects to Design and Build Radar ESM Systems. In Meyrowitz, N. (Ed.), *Proc. OOPSLA '87*, pp. 192–201. ACM Press.

Bloom, T. (1979). Evaluating Synchronization Mechanisms. In *Proc. Seventh ACM Symposium on Operating System Principles*, pp. 24–32.

Booch, G. (1990). *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Co.

Brebner, G. (1991). A CCS-based Investigation of Deadlock in a Multi-Process Electronic Mail System. Department of Computer Science, University of Edinburgh.

Cardelli, L., & Wegner, P. (1985). On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys, 17*(4), 471–522.

Cargill, T. (1992). *C++ Programming Style.* Addison-Wesley.

Chin, R., & Chanson, S. (1991). Distributed Object-Based Programming Systems.

ACM Computing Surveys, 23(1), 91–124.

Coad, P., & Yourdon, E. (1991). *Object-Oriented Design*. Yourdon Press.

Courtois, P., Heymans, F., & Parnas, D. (1971). Concurrent Control with 'Readers' and 'Writers'. *Communications of the ACM, 14*(10), 667–668.

Danforth, S., & Tomlinson, C. (1988). Type Theories and Object-Oriented Programming. *ACM Computing Surveys, 20*(1), 29–72.

Decouchant, D., Krakowiak, S., Meysembourg, M., Riveill, M., & de Pina, X. R. (1989). A Synchronization Mechanism for Typed Objects in a Distributed System. *SIGPLAN Notices, 24*(4), 105–107.

Detlefs, D., Herlihy, M., & Wing, J. (1988). Inheritance of Synchronization and Recovery Properties in Avalon/C++. *Computer, 21*(12), 57–69.

Dijkstra, E. (1965). Solution of a Problem in Concurrent Programming Control. *Communications of the ACM, 8*(9), 569.

Frølund, S. (1992). Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In Lehrmann Madsen, O. (Ed.), *Proc. ECOOP '92 (LNCS 615)*, pp. 185–196. Springer-Verlag.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

Harrison, W., Sweeney, P., & Shilling, J. (1989). Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm. In Meyrowitz, N. (Ed.), *Proc. OOPSLA '89*, pp. 85–94. ACM Press.

Hoare, C. (1974). Monitors: An Operating System Structuring Concept. *Communications of the ACM, 17*(10), 549–557.

Kafura, D., & Lee, K. (1989). Inheritance in Actor Based Concurrent Object-Oriented Languages. *Computer Journal, 32*(4), 297–304.

Kennedy, B. (1991). The Features of the Object-Oriented Abstract Type Hierarchy (OATH). In *Proc. 1991 USENIX C++ Conference*, pp. 41–50.

Kiczales, G., & Lamping, J. (1992). Issues and the Design and Documentation of Class Libraries. In Paepcke, A. (Ed.), *Proc. OOPSLA '92*, pp. 435–451. ACM Press.

Kittlitz, K. (1991). Verifying the Macintosh Threads and Futures System. Course Project Report. Department of Computer Science, University of Calgary.

Kozen, D. (1983). Results on the Propositional μ-Calculus. *Theoretical Computer*

Science, 27.

Löhr, K.-P. (1992). Concurrency Annotations. In Paepcke, A. (Ed.), *Proc. OOPSLA '92*, pp. 327–340. ACM Press.

McHale, C., Walsh, B., Baker, S., & Donnelly, A. (1992). Scheduling Predicates. In Tokoro, M., Nierstrasz, O., & Wegner, P. (Eds.), *Proc. ECOOP '91 Workshop on Object-Based Concurrent Computing (LNCS 612)*, pp. 177–193. Springer-Verlag.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall.

Meyers, S. (1992). *Effective C++*. Addison-Wesley.

Miller, G. (1956). The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review, 63*(2), 81–97.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Moller, F. (1991). The Edinburgh Concurrency Workbench (Version 6.0). Department of Computer Science, University of Edinburgh.

Moss, J., & Kohler, W. (1987). Concurrency Features for the Trellis/Owl Language. In *Proc. ECOOP '87 (LNCS 276)*, pp. 171–180. Springer-Verlag.

Nakajima, T., Yokote, Y., Tokoro, M., Ochiai, S., & Nagamatsu, T. (1989). DistributedConcurrentSmalltalk: A Language and System for the Interpersonal Environment. *SIGPLAN Notices, 24*(4), 43–45.

Nelson, M. (1991). Concurrency and Object-Oriented Programming. *ACM SIGPLAN Notices, 26*(10), 63–72.

Neusius, C. (1991). Synchronizing Actions. In America, P. (Ed.), *Proc. ECOOP '91 (LNCS 512)*, pp. 118–132. Springer-Verlag.

Nierstrasz, O. (1987). Active Objects in Hybrid. In Meyrowitz, N. (Ed.), *Proc. OOPSLA '87*, pp. 243–253. ACM Press.

Nierstrasz, O. (1992). Towards an Object Calculus. In Tokoro, M., Nierstrasz, O., & Wegner, P. (Eds.), *Proc. ECOOP '91 Workshop on Object-Based Concurrent Computing (LNCS 612)*, pp. 1–20. Springer-Verlag.

Nierstrasz, O., & Papathomas, M. (1990). Viewing Objects as Patterns of Communicating Agents. In Meyrowitz, N. (Ed.), *Proc. OOPSLA/ECOOP '90*, pp. 38–43. Addison-Wesley.

Parrington, G., & Shrivastava, S. (1988). Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems. In *Proc. ECOOP '88*, pp. 233–249.

Porter III, H. (1992). Separating the Subtype Hierarchy from the Inheritance of Implementation. *Journal of Object-Oriented Programming, 4*(9), 20–29.

Reghizzi, S., Paratesi, G., & Genolini, S. (1991). Definition of Reusable Concurrent Software Components. In America, P. (Ed.), *Proc. ECOOP '91 (LNCS 512)*, pp. 148–166. Springer-Verlag.

Saleh, H., & Gautron, P. (1992). A Concurrency Control Mechanism for C++. In Tokoro, M., Nierstrasz, O., & Wegner, P. (Eds.), *Proc. ECOOP '91 Workshop on Object-Based Concurrent Computing (LNCS 612)*, pp. 195–210. Springer-Verlag.

Stroustrup, B. (1991). *The C++ Programming Language (Second Edition)*. Addison-Wesley.

Tomlinson, C., & Singh, V. (1989). Inheritance and Synchronization with Enabled-Sets. In Meyrowitz, N. (Ed.), *Proc. OOPSLA '89*, pp. 103–112. ACM Press.

Uhl, J., & Schmid, H. (1990). *A Systematic Catalogue of Reusable Abstract Data Types (LNCS 460)*. Springer-Verlag.

Wirth, N. (1971). Program Development by Stepwise Refinement. *Communications of the ACM, 14*(4), 221–227.