Linearizable Concurrent Objects

Maurice P. Herlihy and Jeannette M. Wing[1]
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Overview

Technological advances are making multiprocessors more readily available, but despite impressive progress at the hardware level, it is still difficult to realize these machines' potential for parallelism. In the sequential domain, "object-oriented" programming methodologies based on data abstraction are widely recognized as an effective means of enhancing modularity, expressibility, and correctness. Our paper describes the foundations for a new approach to extending object-oriented methodologies to highly-concurrent shared-memory multiprocessors.

The basic idea is the following: rather than communicating through low-level machine constructs such as bytes, words, registers, etc., processes communicate through abstract data structures called *concurrent objects*. For example, in a real-time system consisting of a pool of sensor and actuator processes, the processes might communicate via a first-in-first-out (FIFO) queue object in shared memory. When a sensor process detects a condition requiring a response, it records the condition by enqueuing a record in the queue. Whenever an actuator process becomes idle, it dequeues the next item from the queue and takes appropriate action. While the correct behavior of objects such as FIFO queues is well-understood in the sequential domain, it is not so obvious how a FIFO queue should behave when manipulated by concurrent processes. We advocate a new correctness condition, *linearizability*, for implementing and reasoning about concurrent objects.

We view a concurrent system as a collection of sequential processes that communicate through shared objects. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to create and manipulate that object. In the absence of concurrency, operations are executed one at a time, and their meanings can be captured by simple pre- and postconditions. In a concurrent program, however, operations can be executed concurrently, thus it is necessary to find a new meaning for operations that may overlap in time. Two requirements seem to make intuitive sense: First, each operation should appear to "take effect" instantaneously, and second, the order of non-concurrent operations should be preserved. We define a concurrent computation to be *linearizable*[2] if it is "equivalent" to a legal sequential computation that satisfies these two requirements.

Informally, linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response. This property implies that the effects of a concurrent object's operations can still be specified using pre- and post-conditions; however, we interpret a data type's sequential specification as permitting only linearizable interleavings. Thus, instead of treating data as a mass of hardware registers or single large database whose entities are uninterpreted, linearizability exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain.

Using axiomatic specifications and our notion of linearizability, we can reason about two kinds of problems: (1) the correctness of linearizable object implementations by using new techniques that generalize the sequential notions of representation invariant and abstraction function to the concurrent domain; and (2) the correctness of computations that use linearizable objects by transforming assertions about concurrent computations into simpler assertions about their sequential counterparts.

[1]This research was sponsored by IBM and the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864 (Amendment 20), under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories. Additional support for J. M. Wing was provided in part by the National Science Foundation under grant CCR-8620027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

[2]Formally defined in Herlihy and Wing, "Axioms for concurrent objects," *14th ACM Symposium on POPL*, January 1987, pp. 13-26.

Two Examples: Queues and Registers

Consider some informal examples to illustrate what we do and do not consider intuitively acceptable concurrent behavior. Our first set of examples employs a FIFO queue, a simple data type that provides two operations: *Enq* inserts an item in the queue, and *Deq* returns and removes the oldest item from the queue. Figure 1 shows four different ways in which a FIFO queue might behave when manipulated by concurrent processes. Here, a time axis runs from left to right, and each operation is associated with an interval. Overlapping intervals indicate concurrent operations. We use "q E(x) A" ("q D(x) A") to stand for the enqueue (dequeue) operation of item x by process A on queue object q.

The behavior shown in H_1 (Figure 1.a) corresponds to our intuitive notion of how a concurrent FIFO queue should behave. In this scenario, processes A and B concurrently enqueue x and y onto q. Later, B dequeues x, and then A dequeues y and begins enqueuing z. Since the dequeue for x precedes the dequeue for y, the FIFO property implies that their enqueues must have taken effect in the same order. In fact, their enqueues were concurrent, thus they could indeed have taken effect in that order. The uncompleted enqueue of z by A illustrates that we are interested in behaviors in which processes are continually executing operations, perhaps forever.

The behavior shown in H_2, however, is not intuitively acceptable. Here, it is clear to an external observer that x was enqueued before y, yet y is dequeued without x having been dequeued. To be consistent with our informal requirements, A should have dequeued x. We consider the behavior shown in H_3 to be acceptable, even though x is dequeued before its enqueuing operation has returned. Intuitively, the enqueue of x took effect before it completed. Finally, H_4 is clearly unacceptable because y is dequeued twice.

To decide whether a concurrent history is acceptable, it is necessary to take into account the object's intended semantics. For example, acceptable concurrent behaviors for FIFO queues would not be acceptable for stacks, sets, directories, etc. When restricted to register objects providing read and write operations, our intuitive notion of acceptability depends on an axiomatization of concurrent registers. For example, consider the register object x in Figures 1.e and 1.f. H_5 is acceptable, but H_6 is not. These two behaviors differ at one point: In H_5, B reads a 0, and in H_6, B reads a 1. The latter is intuitively unacceptable because A did a previous read of a 1, implying that B's write of 1 must have occurred before A's read. C's subsequent write of 0, though concurrent with B's write of 1, strictly follows A's read of 1.

Significance of Linearizability

The role of linearizability for concurrent objects is analogous to the role of serializability[3] for data base theory: it facilitates certain kinds of formal (and informal) reasoning by transforming assertions about complex concurrent behavior into assertions about simpler sequential behavior. Like serializability, linearizability is a safety property; it states that certain interleavings cannot occur, but makes no guarantees about what must occur. Other techniques, such as temporal logic must be used to reason about liveness properties like fairness or priority.

Unlike alternative correctness conditions such as sequential consistency[4] or serializability, linearizability is a local property. Locality enhances modularity and concurrency, since objects can be implemented and verified independently, and run-time scheduling can be completely decentralized. Linearizability is also a non-blocking property. Non-blocking enhances concurrency and implies that linearizability is an appropriate condition for systems for which real-time response is critical. Linearizability is a simple and intuitively appealing correctness condition that generalizes and unifies a number of correctness conditions both implicit and explicit in the literature.

Without linearizability, the meaning of an operation may depend on how it is interleaved with concurrent operations. Specifying such behavior would require a more complex specification language, as well as producing more complex

[3]Papadimitriou, "The serializability of concurrent database updates," *JACM*, 26:4, Oct. 1979, pp. 631-653.

[4]Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computers*, 28:9, Sept. 1979.

```
     q E(x) A                                        q D(y) A        q E(z) A   ...
    |-----------|                                   |----------|    |----------
      q E(y) B          q D(x) B
     |--------|        |----------|
```

a. H_1 (acceptable).

```
    q E(x) A                 q D(y) A
   |---------|              |----------|
                 q E(y) B
                |----------|
```

b. H_2 (not acceptable).

```
     q E(x) A   ...
    |---------
         q D(x) B
        |----------|
```

c. H_3 (acceptable).

```
     q E(x) A                  q D(y) A
    |----------|              |-----------|
          q E(y) B                   q D(y) C
         |-----------|              |----------|
```

d. H_4 (not acceptable).

```
  x W(0) A               x R(1) A          x W(0) C
 |-----------|          |-------------|   |------------|
                              x W(1) B                          x R(0) B
               |-------------------------------------|       |------------|
```

e. H_5 (acceptable).

```
  x W(0) A               x R(1) A          x W(0) C
 |-----------|          |-------------|   |------------|
                              x W(1) B                          x R(1) B
               |-------------------------------------|       |------------|
```

f. H_6 (not acceptable).

Figure 1: Queue (a-d) and Register (e-f) Histories

specifications. An implementation of a concurrent object need not realize all interleavings permitted by linearizability, but all interleavings it does realize must be linearizable. The actual set of interleavings permitted by a particular implementation may be quite difficult to specify at the abstract level, being the result of engineering trade-offs at lower levels. As long as the object's client relies only on linearizability to reason about safety properties, the object's implementor is free to support any level of concurrency that appears to be cost-effective.

Linearizability provides benefits for specifying, implementing, and verifying concurrent objects in multiprocessor systems. Rather than introducing complex new formalisms to reason directly about concurrent computations, we feel it is more effective to transform problems in the concurrent domain into simpler problems in the sequential domain.