Concurrent Data Structures and Actor Programming under the Matroshka Model

Lawrence A. Crowl
University of Rochester, Computer Science Department, Rochester NY 14627
crowl@cs.rochester.edu

In the late panel session, Maurice Herlihy questioned whether the Actor model [1] could implement a wait-free shared queue. Processes are guaranteed to complete queue operations in a finite amount of steps even if other processes may fail at arbitrary points. Consequently, the queue may not have an implementation based on mutual exclusion, but may be implemented with non-trivial atomic operations. Gul Agha did not provide a method for implementing such an abstraction, but argued that the techniques used to implement the wait-free queue would also apply to implementing the Actor model.

The Matroshka model (described in the position statement "A Uniform Object Model for Parallel Programming" within these proceedings and [2]) can implement both wait-free queues and the Actor model of computation.

A Concurrent Queue Implementation

The critical features needed to implement wait-free data structures are concurrent execution of operations and sufficiently powerful primitive operations. An example demonstrates that the Matroshka model can implement concurrent wait-free queues. This example uses the mechanism described by Herlihy [3]. The implementation atomically appends to a *history of operations* on the queue, and then reconstructs the state of the queue from the history. Herlihy also shows how to avoid saving the entire history. Since reconstructing the state from the history is straightforward, this example only provides the atomic addition to the history list. In the following singly-linked-list algorithm, additions occur at the head of the list and return the previous state of the history.

An object represents the head of the list and addition operations apply to the object. Operations reside on the history list as element records containing a next pointer. Clients have concurrent access to the list because operations under the Matroshka model proceed concurrently unless explicitly synchronized. This algorithm relies on synchronization by a wait-free atomic swap operation on primitive reference objects. This operation swaps the reference's value with another reference indicated by the parameter. This is a fair implementation since the literature [3] commonly assumes these primitive objects and operations.

```
object history
    head: reference element := nil

    operation add_to_history( new_elem: reference element ): reference element
        new_elem^.next := new_elem -- to link in self
        -- swap head and new_elem^.next
        head.atomic_swap( reference to new_elem^.next )
        -- now head = new_elem and new_elem^.next = old value of head
        return new_elem^.next -- return old history
    end
```

An Actor Implementation

The implementation of the Actor model under the Matroshka model is slightly more complicated. It is natural to represent each actor as an object and implement the sending of a message as invoking the message handling operation on the object. The steps in the handler are:

- Accept parameters.
- Execute an *early reply*, which allows the sender to proceed.
- Obtain exclusive access to the fields of the object. Use a test-and-set bit or a semaphore depending on fairness constraints. Some very simple wait-free operations may implement mutual exclusion [3, 4].
- Copy the actor state to a local variable (if necessary).
- Set the object state to the actor's replacement value (if necessary).
- Release exclusive access to the object.
- Continue independent processing.

The resulting code looks something like:

```
object actor
    actor state ...
    operation handler( parameters )
        local state variable
        reply -- allows sender to continue asynchronously
        exclusive.obtain( )
        -- now only one message may cause a replacement
        local state variable := actor state
        actor state := replacement for local state
        exclusive.release( )
        continue using local state variable only
    end
```

Much of the power of the Actor model derives from its unbounded message buffers. The Matroshka model provides no message buffers, but does allow an arbitrary number of operations to be active simultaneously. To achieve the effect of unbounded buffers under the Matroshka model, each message is handled by an activation that waits its turn.

References

[1] Gul A. Agha. An overview of Actor languages. *ACM SIGPLAN Notices*, 21(10):58–67, October 1986.

[2] Lawrence A. Crowl. A uniform object model for parallel programming. Distributed at the 1988 Workshop on Object Based Concurrent Programming and available from the University of Rochester Computer Science Department, September 1988.

[3] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

[4] Leslie Lamport. The mutual exclusion problem: Part II—statement and solutions. *Journal of the ACM*, 33(2):313–326, April 1986.