an example of an abstract data type for which we can improve the time bounds for the other operations.

Consider the TWOARRAY abstract data type (Corollary 8.4). The operations that update the array objects are $r1w2$ and $w1$. The updates performed by these two update operations modify different components of the arrays.

Suppose we want a linearizable implementation of TWOARRAY objects that optimizes $|R1W2|$. Since $R1W2$ is self-oblivious, $|R1W2|$ can be optimized. If $|R1W2| = 0$, then $|W1| \geq d$ and $|R2| \geq d$ by Theorem 8.2. Thus, the best we can hope for is a linearizable implementation with $|R1W2| = 0$ and $|W1| = |R2| = d$. We now describe such an implementation.

Theorem 11.1 *There exists a linearizable implementation of TWOARRAY objects such that* $|R1W2| = 0$ *and* $|W1| = |R2| = d$.

Proof We exhibit our implementation in Figure 11.2. If a process receives several messages at the same time, it handles $r1w2$ messages before $w1$ messages, using process identifiers to break any other ties.

We now prove that our implementation produces linearizable executions. Let ρ be an admissible execution. We show how to produce τ.

Since we are in a system with perfectly synchronized clocks and constant message delays, there is a total order on the messages delivered. This seems like a promising foundation for a method to determine τ. However, we have a problem if an $r1w2$ operation overlaps a $w1$ operation for the same object. Thus, we need a different placement method to construct τ.

We first place $w1$ operations in the order in which their messages are received. The $w1$ operations are all legal (they do not return any values), and they satisfy the ordering constraints required by linearizability.

Next we place the $r1w2$ operations. We consider them in order of their invocation times, breaking ties with process identifiers. We place an $r1w2$ operation immediately after the latest of

1. its last preceding operation placed so far,

2. the $w1$ operation from which it returns its value, and
3. the last $r1w2$ operation for the same object occurring at the same time and preceding its message in the message delivery order.

An $r1w2$ operation will not be placed before any subsequent operations. Now we must verify the legality of $r1w2$ operations.

If an $r1w2$ operation is the first operation in the current version of τ, then it is legal because it returns the initial value of the first component of its object. If an $r1w2$ operation is placed before any $w1$ operations for the same object, then it is legal because it returns the initial value of the first component of its object.

Now consider an $r1w2$ operation which does not satisfy either of the previous two conditions. If in the placement rule, the $r1w2$'s last preceding operation placed so far is the $w1$ operation from which the $r1w2$ returns its value, then the $r1w2$ operation is legal.

The $w1$ operation from which an $r1w2$ operation returns must start at least time d before the $r1w2$ because otherwise the process invoking the $r1w2$ operation would not yet have received the message about the $w1$ operation.

Thus, if in applying the placement rule, the $r1w2$ operation's last preceding operation comes before the $w1$ operation from which it returns its value in the current version of τ, the $w1$ operation starts exactly d before it. The $r1w2$ operation is legal.

The final case to consider is the case in which the $w1$ operation from which the $r1w2$ operation returns comes before the $r1w2$ operation's last preceding operation in the current version of τ. We must show that the last preceding operation is not a $w1$ operation on $r1w2$'s object, and we must show that no $w1$ operation on $r1w2$'s object occurs between $r1w2$'s source operation (the $w1$ from which it returns its value) and its last preceding operation. Suppose in contradiction that there is an interfering $w1$ operation for $r1w2$'s object. Then this operation also precedes the $r1w2$ operation. By the algorithm, the process invoking the $r1w2$ would have applied the update for the $w1$ operation before the $r1w2$ operation was started, contradicting the choice of return value and responsible $w1$ operation.

Thus, $r1w2$ operations are legal.

Finally, we must place $r2$ operations in τ. We consider them in order of their invocation times, breaking ties with process identifiers. We place an $r2$ operation immediately after the later of

1. its last preceding operation placed so far and

2. the $r1w2$ operation from which it returns its value.

An $r2$ operation will not be placed before any subsequent operations. Now we must verify the legality of $r2$ operations.

If an $r2$ operation is the first operation in τ, then it is legal because it returns the initial value of the second component of its object. If an $r2$ operation is placed before any $r1w2$ operation for the same object, then it is legal because it returns the initial value of the second component of its object.

Now consider an $r2$ operation which does not satisfy either of the previous two conditions. If in the placement rule, the $r2$'s last preceding operation placed so far is the $r1w2$ operation from which $r2$ returns its value, then the $r2$ operation is legal.

The $r1w2$ from which an $r2$ operation returns its value must not start later than the $r2$ because otherwise the process invoking the $r2$ operation would not yet have received the message about the $r1w2$ operation.

If $r2$'s last preceding operation comes before the $r1w2$ from which it returns in τ, then $r2$ is legal by construction.

The final case to consider is the case in which the $r1w2$ from which the $r2$ returns comes before the $r2$'s last preceding operation in τ. We must show that the last preceding operation is not an $r1w2$ operation on $r2$'s object, and we must show that no $r1w2$ operation on $r2$'s object occurs between $r2$'s source operation (the $r1w2$ operation from which it returns its value) and its last preceding operation. Suppose in contradiction that there is an interfering $r1w2$ operation on $r2$'s object. Then this operation also precedes the $r2$ operation. By the algorithm, the process invoking $r2$ would have applied the update for the $r1w2$ operation before the $r2$

```
$r1w2(v)[X,p]$:
        broadcast($DOr1w2(v,X)$)
        $r1w2return(X[1])[X,p]$

$DOr1w2_p(v,X)$: /* executed when message is received */
        $X[2] := v$

$w1(v)[X,p]$:
        timerset($p, d$)
        broadcast($DOw1(v,X)$)
        when timer expires:
            $w1ack(Ok)[X,p]$

$DOw1_p(v,X)$: /* executed when message is received */
        $X[1] := v$

$r2[X,p]$:
        timerset($p, d$)
        when timer expires:
            $r2return(X[2])[X,p]$
```

Figure 11.2: Algorithm for TWOARRAY - Code for Process p

operation was started, contradicting our choice of return value and responsible $r1w2$ operation.

Thus, $r2$ operations are legal.

We have exhibited a legal linearization τ for our admissible execution ρ. ■

Can we use the same approach as in TWOARRAY in order to optimize a general self-oblivious operation without cyclic dependences with other operations? The answer is "No", and we will answer this question with a type called THREEARRAY. The objects are three-element arrays. The operations are $r2i1$ (which reads the second component and increments the first component of the array on which it is invoked), $r3d1i2$ (which reads the third component, doubles the first component, and increments the second component of the array on which it is invoked), and $r1i3$ (which reads the first component and increments the third component of the array on which it is invoked). Each operation commutes with itself.

Suppose we want a linearizable implementation of THREEARRAY objects that optimizes $|R2I1|$. If $|R2I1| = 0$, then $|R3D1I2| \geq d$ and $|R1I3| \geq d$ by Theorem 8.2.

Let us adapt the implementation in Figure 11.2 for THREEARRAY objects as a first attempt. In this, $|R2I1| = 0$ and $|R3D1I2| = |R1I3| = d$. We assume that $X[1]$ has been initialized to 1 and that $X[2]$ and $X[3]$ have been initialized to 0. The following execution ρ is a possible execution:

- At time 0, process 4 starts an $r1i3$ operation on object X. It completes at time d.
- At time $d/4$, process 2 starts an $r3d1i2$ operation on object X. It completes at time $5d/4$.
- At time $d/2$, process 3 starts an $r1i3$ operation on object X. It completes at time $3d/2$.
- At time $d/2$, process 1 executes an $r2i1$ operation on object X, returning 0.

Process 4 returns 1 for its $r1i3$ operation when it completes. Process 2 returns 1 for its $r3d1i2$ operation when it completes. Process 3 returns 3 for its $r1i3$ operation when it completes.

In the total ordering of all operations in ρ, the $r2i1$ operation must come before the $r3d1i2$ operation in order to be legal. However, the $r3d1i2$ operation must come before the $r2i1$ operation in order for process 3's $r1i3$ operation to be legal. This implies a cycle in the total ordering, and therefore a contradiction. However, all is not lost because we can instantiate the implementation described in Theorem 8.7 to optimize $|R2I1|$ because $R2I1$ is self-oblivious.

11.3 Optimizing Multiple Operations

The examples from the previous sections show that optimizing just a single operation from an abstract data type is a nontrivial task. What can we say about optimizing *multiple* operations, besides all accessors and all pure modifiers? In our attempt to determine exactly when a single operation can be optimized, we found a result which could have a negative impact on optimizing multiple operations, as discussed after the result.

Theorem 11.2 *Given abstract data type T with generic operations OP_1 and OP_2, an operation sequence ρ, and operation instances op_1^1, op_1^2, and op_2 such that*

- *$\rho \cdot op_1^1$ is legal,*
- *$\rho \cdot op_2 \cdot op_1^2$ is legal,*
- *$\rho \cdot op_2 \cdot op_1^1$ is not legal,*
- *and $\rho \cdot op_1^1 \cdot op_2 \cdot op_1^2$ is not legal,*

then $|OP_1| > 0$ or $|OP_1| + |OP_2| \geq 2d$ in any linearizable implementation of objects of type T.

Intuitively, this means that OP_1 is not self-oblivious.

Proof Let A be an object of type T. We consider A_ρ, the ρ-initialized version of A.

Let processes 1 and 2 use A. Suppose in contradiction that there is a linearizable implementation of A such that $|OP_1| = 0$ and $|OP_1| + |OP_2| < 2d$. Thus, $|OP_2| = 2d - \epsilon$, for some $\epsilon > 0$.

By the sequential specification for A_ρ, there is some admissible execution α_1 such that $ops(\alpha_1)$ is $op_1^1[A_\rho, 1]$. Assume that the op_1 operation starts at time $d - \epsilon/2$. Any message sent by process 1 would not arrive until at least time $2d - \epsilon/2$.

By the sequential specification for A_ρ, there is some admissible execution α_2 such that $ops(\alpha_2)$ is $op_2[A_\rho, 2] \cdot op_1^2[A_\rho, 2]$. Assume that the op_2 operation starts at time 0 and the op_1 operation starts at time $2d - \epsilon$. Any message sent by process 2 would not be delivered until at least time d.

Since no messages are received in α_1 and α_2 before time d, replacing process 1's history in α_2 with its history in α_1 results in another admissible execution, α. By assumption, α is linearizable. Thus, we can construct a linearization of $ops(\alpha)$ which is legal for A_ρ. In this linearization, op_2 must precede op_1^2, and op_1^1 must precede op_1^2. Also, op_1^1 must precede op_2. This implies the ordering $op_1^1 \cdot op_2 \cdot op_1^2$, which is not legal for A_ρ. ■

As a result of this, suppose OP_2 is self-oblivious. We obtain the following corollary. In any linearizable implementation where $|OP_2| = 0$, $|OP_1| > 0$.

Determining whether a given operation or group of operations can be optimized is a very interesting problem. Optimizing a given operation or group of operations is useful when they are frequently used. Although we do not have a complete characterization of when a given operation or group of operations cannot be optimized, we have made reasonable progress on this problem, and we hope that our work can be used as a foundation for determining this elusive complete characterization.

Chapter 12

Future Work

In this thesis, we have investigated the costs of implementing concurrent shared objects. Studying these costs is important because using shared objects is a form of interprocess communication. Using shared objects enables computer programs to be developed with software engineering principles.

We have investigated both physical and virtual implementations of concurrent shared objects.

With respect to physical implementations of concurrent shared objects, we studied wait-free implementations of k-valued single-writer regular (respectively, atomic) registers from regular (respectively, atomic) bits. We sought to optimize (and did!) the protocol for the writer of the implemented k-valued register and studied the impact of the optimization on the space requirement and the work requirement for the readers of the register. We improved previous lower bounds on space for two classes of regular one-write algorithms, toggle and symmetric, yielding a tight bound for toggle algorithms. There is still room for improvement in the space bounds for symmetric algorithms and more general one-write algorithms. In the case of safe one-write algorithms, [CW90] found a connection with coding theory in order to determine a tight space bound. Perhaps there is some connection with coding theory that we can exploit in order to tighten our space bounds for regular and atomic one-write algorithms.

Other areas for future work include studying the (inherent) costs of implementations of multiwriter and atomic registers and investigating the costs of shared register implementations when readers and writers are not totally asynchronous.

With respect to virtual implementations of concurrent shared objects, we studied implementations of objects from general abstract data types. We evaluated the impact of the consistency guarantee to be provided (sequential consistency, linearizability, and hybrid consistency), the amount of system synchrony, and algebraic properties of the operations of the type on the worst-case time complexity of operations of the type.

We have identified several algebraic properties that cause operations (or pairs of operations) to be "slow" (have worst-case execution times which are $\Omega(d)$, where d is the message delay of the network) in sequentially consistent (and linearizable) implementations of abstract data types. Some of these properties can be used to show that concurrent abstract data types with weaker consistency conditions are not necessarily asymptotically cheaper to implement with respect to time than concurrent abstract data types with stronger consistency conditions.

In perfectly synchronized systems, we have shown that for a large class of data types, we can choose one operation and optimize its worst-case execution time (make it be effectively 0) in a linearizable implementation of its abstract data type. We would like to determine if it is **always** possible to optimize the worst-case execution time for an operation, given that it immediately self-commutes. An interesting avenue to pursue further is optimization of the worst-case execution time of **all** operations of an abstract data type. This could involve finding new algebraic properties of operations which cause them to be "slow". Ideally we envision a "compiler" which takes a representation of the algebraic structure of an abstract data type (i.e., commutativity graph or something similar) as input and produces an optimized linearizable implementation as output. Optimization of the worst-case execution time of all operations of an abstract data type is useful when all operations are invoked with approximately the same frequency.

In systems with approximately synchronized clocks, we have identified some algebraic properties that "slow down" operations, or make their worst-case execution times be $\Omega(u)$, where u is the uncertainty in the network message delay, in linearizable implementations of abstract data types. We have shown that for a large class of data types, atomic broadcast can be used to develop sequentially consistent implementations in which certain operations can be optimized with respect to worst-case completion time. It would be very interesting to develop linearizable implementations of abstract data types in which the slowed down operations are as fast as they can possibly be. This may lead to higher lower bounds on worst-case time complexity for the slowed down operations. Also, we could study atomic broadcast further in order to improve its time complexity, yielding faster sequentially consistent implementations for free. Alternatively, we could investigate lower bounds on the worst-case time complexity for sequentially consistent (or providing a related consistency guarantee) implementations of abstract data types.

In systems with approximately synchronized clocks, lower bounds on the worst-case time complexity for sequentially consistent operations are at least as high as in systems with perfectly synchronized clocks. Intuition tells us that they may possibly be higher. [MR92] claimed that for sequentially consistent read/write objects, $|READ| + |WRITE| \geq d + u/2$. However, their proof relies wrongly on not being able to shift an operation too far relative to its original position. This observation suggests defining a class of consistency guarantees intermediate between sequential consistency and linearizability, parameterized by some ϵ. The proof techniques used in [MR92] could perhaps be used to show similar lower bounds for general types for this family of conditions.

We now propose a class of intermediate consistency guarantees.

Definition 6 *An execution ρ is* ϵ-**sublinearizable** *if there exists a legal sequence τ of operations such that τ is a permutation of $ops(\rho)$, $ops(\rho)|p = \tau|p$ for each process p, and op_2 follows op_1 in τ if the call for op_2 happens at least ϵ time after the response for op_1.*

When $\epsilon = 0$, we have plain linearizability. ϵ-sublinearizability implies sequential consistency, but not vice versa.

The following is a very simple observation about ϵ-sublinearizability.

Theorem 12.1 *In any ϵ-sublinearizable implementation of read/write objects where $\epsilon \leq d - u$, $|READ| + |WRITE| \geq \epsilon + u$.*

Proof Since ϵ-sublinearizability implies sequential consistency, $|READ|+|WRITE| \geq d$ by Corollary 8.2. Because $\epsilon + u \leq d$, $|READ| + |WRITE| \geq \epsilon + u$. ■

It would be interesting to see what happens to the lower bound on the worst-case time complexity for operations in ϵ-sublinearizable implementations of concurrent shared objects in systems with approximately synchronized clocks.

In this thesis, we were concerned with lower bounds on worst-case time complexity for hybrid consistent implementations of abstract data types. Upper bounds, or actual implementations, are very important as well. Attiya and Friedman [AF92] gave a hybrid consistent implementation of read/write objects in which weak operations are optimized. We would like to find more abstract data types, or better yet, classes of abstract data types, for which weak operations can be optimized.

We have chosen to focus on worst-case **time** complexity as our cost measure in our study of virtual implementations of concurrent shared objects. Space complexity is another very important cost measure. We can study this cost measure from two points of view: the amount of local space required and the capacity required in the network (the number and size of messages that are sent). We can also study the impact of the network topology on the costs of virtual implementations of concurrent shared objects. We had assumed a complete communication network that was powerful enough to handle all our message transmission needs. Our lower bounds are still true for systems without a complete communication network. In systems with perfectly synchronized clocks, our lower bounds can be multiplied by the network diameter to yield improved lower bounds. Studying the impact of the network topology and capacity should prove to be very interesting because it will involve issues about routing algorithms and fault tolerance.

In this thesis, we have uncovered some interesting results about the costs of implementing concurrent shared objects, for both physical and virtual implementations. We have built k-valued shared registers (regular and atomic, respectively) from binary registers (regular and atomic, respectively) in which the writer's protocol is optimal, and we have rigorously proved the correctness of the respective implementations. We have also proved lower bounds on space requirements for classes of one-write algorithms. As a result of this work, we have shown again how combinatorial mathematics and graph-theoretic concepts can be applied to problems in distributed computing. In our study of virtual implementations of concurrent shared objects, we have used algebraic properties of operations to prove that strong consistency is expensive to provide. We have also shown that weaker correctness guarantees are not necessarily cheaper to provide than stronger guarantees. We have shown that optimizing the worst-case time complexity of even a single operation of an abstract data type is a nontrivial task. We hope our work gives users of virtual implementations of concurrent shared objects clues about what they can and cannot expect from such implementations. We have listed some new areas for further investigation. We hope that our work can provide a foundation for gaining new understanding about implementations of concurrent shared objects.

Bibliography

[ABM93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. A Lazy Cache Algorithm. *ACM Transactions on Programming Languages and Systems*, 13(1), 1993.

[ACFW93] Hagit Attiya, Soma Chaudhuri, Roy Friedman, and Jennifer Welch. Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[AF92] Hagit Attiya and Roy Friedman. A Hybrid Condition for Shared Memory Consistency. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, May 1992.

[AG91] James H. Anderson and Bojan Grošelj. Beyond Atomic Registers: Bounded Wait-Free Implementations of Nontrivial Objects. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, October 1991.

[AH90] Sarita Adve and Mark Hill. Weak Ordering - A New Definition. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 2–14, 1990.

[Att91] Hagit Attiya. Implementing FIFO Queues and Stacks. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, pages 80–94, October 1991.

[AW91] Hagit Attiya and Jennifer L. Welch. Sequential Consistency versus Linearizability. In *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*, pages 304–315, 1991.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 1994. To appear. Also available as Texas A&M University Department of Computer Science Technical Report 93-016.

[Blo87] Bard Bloom. Constructing Two-Writer Atomic Registers. In *Proceedings of the Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 249–259, August 1987.

[BP87] James E. Burns and Gary L. Peterson. Constructing Multi-Reader Atomic Values from Non-Atomic Values. In *Proceedings of the Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 222–231, August 1987.

[CW90] Soma Chaudhuri and Jennifer L. Welch. Bounds on the Costs of Register Implementations. In *Proceedings of the Fourth International Workshop on Distributed Algorithms*, September 1990. Also available as TR90-025 from the University of North Carolina at Chapel Hill. To appear in *SIAM Journal of Computing.*

[DSB88] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *IEEE Computer*, 21(2):9–22, February 1988.

[Fri93] Roy Friedman. Implementing Hybrid Consistency with High-Level Synchronization Operations. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, 1993.

[GLL$^+$90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessey. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, 1990.

[GMG91] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proceedings of the Third ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[HW90] Maurice Herlihy and Jeannette Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 10(3):463–492, 1990.

[JSL91] Prasad Jayanti, Adarshpal Sethi, and Errol L. Lloyd. Minimal Shared Information for Concurrent Reading and Writing. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, October 1991.

[Lam86] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(1):86–101, 1986.

[LL84] Jennifer Lundelius and Nancy Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62(2/3):190–204, 1984.

[LS88] Richard A. Lipton and Jonathan S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 137–151, August 1987.

[LTV89] Ming Li, John Tromp, and Paul M. B. Vitányi. How to Share Concurrent Wait-Free Variables. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pages 488–505, 1989.

[MR92] Marios Mavronicolas and Dan Roth. Efficient, Strongly Consistent Implementations of Shared Memory. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, pages 346–361, November 1992.

[NW87] Richard Newman-Wolfe. A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables. In *Proceedings of the Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 232–248, August 1987.

[Pet83] Gary Peterson. Concurrent Reading While Writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[SAG87] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The Elusive Atomic Register Revisited. In *Proceedings of the Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 206–221, August 1987.

[Tro89] J. T. Tromp. How to Construct an Atomic Variable. Technical Report CS-R8939, Centre for Mathematics and Computer Science, Amsterdam, October 1989.

[VA86] Paul M. B. Vitányi and Baruch Awerbuch. Atomic Shared Register Access by Asynchronous Hardware. In *Proceedings of the Twenty-seventh Annual IEEE Symposium on Foundations of Computer Science*, pages 233–243, October 1986.

[Vid88] K. Vidyasankar. Converting Lamport's Regular Register to Atomic Register. *Information Processing Letters*, 28:287–290, 1988.

[Wei93] William Weihl. The Impact of Recovery on Concurrency Control. *Journal of Computer and System Sciences*, August 1993.

Appendix A

Examples of Abstract Data Types

We now give some examples of abstract data types with their commutativity properties. We will refer to these examples throughout this work.

Let us examine the augmented queue abstract data type. In the augmented queue abstract data type, there are three operations: enqueue, dequeue, and peek. We assume that queues can be of arbitrary length. $\perp$ is returned by a dequeue or peek operation that is invoked on an empty queue. Table A.1 gives the commutativity table for the queue abstract data type. An $\times$ in entry (i, j) means that operation i and operation j do not commute (either eventually or immediately). This holds for all tables. In the augmented queue abstract data type, all pairs of operations which do not commute, immediately do not commute, with the exception of the entry (enq, enq), for which the operations in the pair eventually do not commute.

Next we examine the augmented stack abstract data type. In the augmented stack abstract data type, there are three operations: push, pop, and peek. We assume that stacks can be of arbitrary height. $\perp$ is returned by a pop or peek operation that is invoked on an empty stack. Table A.2 gives the commutativity table for the stack abstract data type. An $\times$ in entry (i, j) means that operation i and operation j do not commute. This holds for all tables. In the augmented stack abstract data type, all pairs of operations which do not commute, immediately do not commute, with the exception of the entry $(push, push)$, for which the operations in the pair eventually do not commute.

Queue Operation	$enq(x)\cdot$ $ok()$	$deq()\cdot$ $ret(x)$	$deq()\cdot$ $ret(\perp)$	$peek()\cdot$ $ret(x)$	$peek()\cdot$ $ret(\perp)$
$enq(y)\cdot ok()$	$\times$		$\times$		$\times$
$deq()\cdot ret(y)$		$\times$	$\times$	$\times$	$\times$
$deq()\cdot ret(\perp)$	$\times$	$\times$		$\times$	
$peek()\cdot ret(y)$		$\times$	$\times$		$\times$
$peek()\cdot ret(\perp)$	$\times$	$\times$		$\times$	

Table A.1: Commutativity Table for the Augmented Queue Abstract Data Type

Stack Operation	$push(x)\cdot$ $ok()$	$pop()\cdot$ $ret(x)$	$pop()\cdot$ $ret(\perp)$	$peek()\cdot$ $ret(x)$	$peek()\cdot$ $ret(\perp)$
$push(y)\cdot ok()$	$\times$		$\times$		$\times$
$pop()\cdot ret(y)$		$\times$	$\times$	$\times$	$\times$
$pop()\cdot ret(\perp)$	$\times$	$\times$		$\times$	
$peek()\cdot ret(y)$		$\times$	$\times$		$\times$
$peek()\cdot ret(\perp)$	$\times$	$\times$		$\times$	

Table A.2: Commutativity Table for the Augmented Stack Abstract Data Type

Set Operation	$ins(x,v)\cdot$ $ok()$	$del(x)\cdot$ $ack(ok)$	$del(x)\cdot$ $ack(\perp)$	$search(x)\cdot$ $ret(v)$	$search(x)\cdot$ $ret(\perp)$
$ins(y,w)\cdot ok()$		$\times$	$\times$		$\times$
$del(y)\cdot ack(ok)$	$\times$	$\times$	$\times$	$\times$	
$del(y)\cdot ack(\perp)$	$\times$	$\times$			
$search(y)\cdot ret(w)$		$\times$			
$search(y)\cdot ret(\perp)$	$\times$				

Table A.3: Commutativity Table for the Dictionary Set Abstract Data Type

Let us now examine the dictionary set abstract data type. We assume that a set can have an arbitrary number of (element,key) pairs. The operations we consider are insert (an element and its key), delete (an element and its key), and search (for an element and return its key). $\perp$ is returned by a delete or search operation that is performed when the input argument is not an element in the set. Table A.3 gives the commutativity table for the set abstract data type. In the set abstract data type, all pairs of operations which do not commute, immediately do not commute.

We now examine the bank account abstract data type. This example comes from [Wei93]. The operations we consider are deposit, withdraw, and balance. $\perp$ is returned by a withdraw operation that is performed when the account contains less money than the amount specified in the input argument. Table A.4 gives the commutativity table for the bank account abstract data type. In the bank account abstract data type, all pairs of operations which do not commute, immediately do not commute.

We now examine a different form of set, a reference-count set. We assume that a reference-count set can have an arbitrary number of elements. The operations we consider are insert, update, and find (search). When an item is inserted into this set, it has a data field which is initialized to 1. All inserts are normal (if we are assuming that sets can be of arbitrary size). When an item is updated, its data field is incremented by 1 (if the item is present). When an item is searched, its data field is returned (if the item is present). $\perp$ is returned by an update or search operation that is performed when the input argument is not in the set. An

Bank Account Operation	$deposit(j) \cdot ok()$	$withdraw(j) \cdot ack(ok)$	$withdraw(j) \cdot ack(\perp)$	$balance() \cdot ret(j)$
$deposit(i) \cdot ok()$			$\times$	$\times$
$withdraw(i) \cdot ack(ok)$		$\times$		$\times$
$withdraw(i) \cdot ack(\perp)$	$\times$			
$balance() \cdot ret(i)$	$\times$	$\times$		

Table A.4: Commutativity Table for the Bank Account Abstract Data Type

Reference Set Operation	$ins(x) \cdot ok()$	$up(x) \cdot ack(ok)$	$up(x) \cdot ack(\perp)$	$find(x) \cdot ret(v)$	$find(x) \cdot ret(\perp)$
$ins(y) \cdot ok()$			$\times$		$\times$
$up(y) \cdot ack(ok)$			$\times$	$\times$	
$up(y) \cdot ack(\perp)$	$\times$	$\times$			
$find(y) \cdot ret(w)$		$\times$			
$find(y) \cdot ret(\perp)$	$\times$				

Table A.5: Commutativity Table for the Reference-Count Set

operation that returns $\perp$ is abnormal. Table A.5 gives the commutativity table for the reference-count set abstract data type. In the reference-count set abstract data type, all pairs of operations which do not commute, immediately do not commute.

We finally examine an abstract data type where the values are real numbers and the operations are read, increment, and half. Table A.6 gives the commutativity table for this abstract data type. Increment and half eventually do not commute.

INC-HALF	$read() \cdot ret(v)$	$inc(x) \cdot ok()$	$half() \cdot ok()$
$read() \cdot ret(v)$		$\times$	$\times$
$inc() \cdot ok()$	$\times$		$\times$
$half() \cdot ok()$	$\times$	$\times$	

Table A.6: Commutativity Table for the Increment-Half Abstract Data Type

Appendix B

Glossary of Terms

$C(k,2)$ the number of combinations of k elements taken 2 at a time.

d the message delay of the network.

OP denotes a generic operation with op as the name of its call event.

$|OP|$ the worst-case completion time for operation OP.

op denotes an operation instance.

op^1, op^2 denoting two separate instances of the same generic operation OP.

$s \cdot t$ sequence s immediately followed by sequence t.

u the uncertainty in the network message delay.

accessor an operation of an abstract data type which **never** changes an object.

action of a register implementation a logical READ, WRITE, RETURN, or ACK, or a physical read, write, return, or ack.

admissible An execution is admissible if all messages sent in it have delays in the range $[d-u, d]$ and for any time t, each process has at most one operation that has not yet finished.

approximately synchronized A system of processes is only approximately synchronized if their clocks are not initialized to the same value and the uncertainty in the message delay between processes is a positive number.

atomicity a property of read/write registers which requires that all operations must be totally ordered in a sequence with the following two restrictions. If operation i on the register completes before operation j on the register begins, operation i precedes operation j in the sequence. Each read operation in the sequence returns the value written by the last write operation placed before it in the sequence.

augmented queue a queue with enqueue, dequeue, and peek (returning the front element of the queue without modifying the queue) operations.

augmented stack a stack with push, pop, and peek (returning the top element of the stack without modifying the stack) operations.

call event the invocation of an operation. This includes the name of the operation, the name of the object on which the operation is invoked, and any arguments of the operation.

clock a monotonically increasing function which maps global (real) times, expressed as real numbers, to local times, expressed as real numbers.

commutativity graph an undirected graph representing the commutativity properties of an abstract data type. The nodes are labeled with the names of the generic operations of the types. There is an edge between two distinct nodes if their corresponding generic operations do not immediately commute. There is a loop from a node to itself if its corresponding generic operation does not immediately commute with itself.

commute Two operations OP_1 and OP_2 commute if for all instances op_1 and op_2 and all operation sequences s, $s \cdot op_1 \cdot op_2$ looks like $s \cdot op_2 \cdot op_1$.

cyclically dependent Two operations OP_1 and OP_2 are cyclically dependent there are instances op_1 and op_2 and a legal operation sequence s such that $s \cdot op_1$ and $s \cdot op_2$ are legal, but $s \cdot op_1 \cdot op_2$ and $s \cdot op_2 \cdot op_1$ are not legal.

doubly noninterleavable with respect to a pair of operations An operation OP is doubly noninterleavable with respect to a pair of operations OP_1 and OP_2 if there exist legal operation sequence s and instances op^1, op^2, op_1, op_2 such that $s \cdot op_1 \cdot op^1$ and $s \cdot op_2 \cdot op^2$ are legal, but each operation sequence formed by the concatenation of s with a permutation of the four operation instances is illegal.

ϵ-sublinearizable An execution is ϵ-sublinearizable if it is sequentially consistent and op_2 follows op_1 in the serialization if the call for op_2 happens at least ϵ time after the response for op_1.

equivalent (with respect to histories) Two histories are equivalent if the views of all processes in both histories are the same.

equivalent (with respect to operations) Two operation sequences are equivalent if operations legally executed after the first sequence are also legal after the second sequence, and vice versa.

eventually do not commute Two operations OP_1 and OP_2 eventually do not commute if there exist a legal operation sequence s and instances op_1 and op_2 such that $s \cdot op_1 \cdot op_2$ and $s \cdot op_2 \cdot op_1$ are legal, but $s \cdot op_1 \cdot op_2$ does not look like $s \cdot op_2 \cdot op_1$.

execution of an mcs An execution of an mcs is a set of histories, one for each process maintaining the shared objects, in which all messages sent are received by the processes to which they were sent and all messages received were actually sent by processes.

execution of a register implementation An execution of a register implementation is a sequence of alternating state tuples (an entry for each reader, each writer, and each physical register) and actions.

fast describing an operation with worst-case completion time of 0. This means that the operation can be completed with only local computation.

generic operation A generic operation encompasses all operations with the same name for their call events. For example, enqueue is a generic operation for queues.

history A history of a process with a clock is a record of everything that occurs at the process (updates to the state of the process, the sending and receipt of messages, interrupts, timers being set and expiring), ordered by clock time (with some tie-breaking).

hybrid consistency a property of shared objects which requires the relative ordering of (weak, strong), (strong,weak), and (strong,strong) operation pairs that are executed by the same process to be preserved in some legal serialization of every execution. In addition, for every process p, there exists a legal serialization of the execution in which response i by p precedes call j by p in the serialization if response i by p precedes call j by p in the execution.

immediately do not commute Two operations op_1 and op_2 immediately do not commute if there exists some sequence of operations such that op_1 and op_2 can individually legally (immediately) follow the sequence, but there is some permutation of op_1 and op_2 which cannot legally immediately follow the sequence.

interrupt event a call event, a message receive event, or a timer event.

legal A sequence of operation instances is legal if it conforms to the semantics of the abstract data types of the objects on which the operations are performed.

linearizability a property of shared objects which requires a legal linearization of every execution.

linearization a serialization with the additional property that if response i precedes call j in the execution, then (call i, response i, call j, response j) is a subsequence of the serialization.

logical register a register which is to be simulated by using a set of (usually) weaker registers and a protocol (a register implementation).

looks like An operation sequence s_1 looks like operation sequence s_2 if any operation sequence that can immediately legally follow s_2 can also immediately legally follow s_1 and any operation sequence that cannot immediately legally follow s_2 also cannot immediately legally follow s_1.

memory consistency system (mcs) A memory consistency system (mcs) is a set of processes and a set of clocks, one for each process. The processes use the clocks and run a protocol to maintain a set of virtual shared objects according to some consistency guarantee.

message delay the amount of real time between the sending of a message and its receipt.

message receive event an indication of the receipt of a message. This includes an identification of the sender of the message, an identification of the receiver of the message, and the contents of the message.

message send event an indication of the sending of a message. This includes an identification of the sender of the message, an identification of the receiver of the message, and the contents of the message.

modifier an operation of an abstract data type which changes an object.

noninterleavable with respect to an ordered pair of operations An operation OP is noninterleavable with respect to an ordered pair of operations OP_1 and OP_2 if there exists a legal operation sequence that can be legally immediately followed by an instance of OP or by an instance of OP_1 immediately followed by an instance of OP_2, but not by all three instances, where the ordering of the three instances which preserves the relative ordering of the instances of OP_1 and OP_2.

operation instance an indication that an operation has performed on an object. It is of the form $op(args, rets)$, where op represents the name of the operation, $args$ is the list of input arguments for the operation, and $rets$ is the list of values returned by the operation.

perfectly synchronized A system of processes is perfectly synchronized if their clocks are initialized to the same value and run at the same rate.

physical register a register which can be used as a piece of "hardware" in a logical register implementation. The register is assumed to provide a certain consistency guarantee.

(mcs) process informally, a program which maintains virtual shared objects.

pure modifier an operation of an abstract data type which changes its object without returning any information about the object.

pure modify/read (PMR) object an object with pure modifier operations and a read operation that returns its current value.

regularity a property of read/write registers which requires the value returned by each read operation to be the value written by the last write operation completed before the read began or by some write overlapping the read.

response event an indication that an operation has completed. This includes an indication of which operation has completed, the name of the object on which the operation was performed, and any values returned as a result of performing the operation.

ρ-initialized version describing an object which has had the sequence ρ of operations performed on it starting from its initial state.

safety a property of read/write registers which requires the value returned by each read operation not overlapping a write operation to be the value written by the last write operation completed before the read began. The only restriction on the values returned by other read operations is that they must be values which could be written to the register.

schedule the sequence of actions in an execution of a register implementation, in the order in which they occurred in the execution.

self-oblivious A generic operation OP is self-oblivious if there exist instances op^1 and op^2 and sequences s_1 and s_2, where s_1 is a prefix of s_2, such that whenever $s_1 \cdot op^1$ and $s_2 \cdot op^2$ are legal, there exists an instantiation of s_2 (only possibly changing return values of operations, implying an instantiation of s_1) such that $s_1^{instantiated} \cdot op^1 \cdot (s_2 - s_1)^{instantiated} \cdot op^2$ is legal, where $s_1 \cdot (s_2 - s_1) = s_2$.

sequential consistency a property of shared objects which requires a legal serialization of every execution such that for every process, if operation i and operation j are executed by that process in the order operation i before operation j, then operation i precedes operation j in the serialization.

sequential specification A sequential specification for an abstract data type describes the operations that can be performed on the objects of the type, along with all possible legal orderings of the operations. These legal orderings are given as sequences.

serialization a sequence formed from the call events and response events of an execution in which each response event occurs immediately after its matching call event.

shift to change the times at which all events in an execution occur by the same amount, or to change the clock function of a process by adding a certain amount to it.

slow describing an operation with worst-case completion time of $\Omega(d)$.

step A step of a process is a tuple containing information about what the process is doing. This information consists of the process' current state and next state, the current clock time at the process, the current interrupt event, the next set of response events, the next set of message send events, the next set of timer set events.

timer event indicating that a timer has gone off at a process at a given time. It is of the form timer(p, T), where p is the process at which the timer went off, and T is the time (according to p's clock) at which the timer went off.

timer set event indicating that a process needs a timer to go off at some point in the future. It is of the form timerset(p,T), where p is the process setting the time, and T is the time (according to p's clock) at which the timer should go off.

view The view of a process in its history is the sequence of steps that it takes in its history, arranged in ascending real-time order. The view does not contain the times when the steps occurred.