NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Specifying Graceful Degradation in Distributed Systems

Maurice P. Herlihy, Jeannette M. Wing

May 18, 1987

CMU-CS-87-120

ABSTRACT

Distributed programs must often display graceful degradation, reacting adaptively to changes in the environment. Under ideal circumstances, the program's behavior satisfies a set of application-dependent constraints. In the presence of failures, timing anomalies, or synchronization conflicts, however, certain constraints may become difficult or impossible to satisfy, and the application designer may choose to relax them as long as the resulting behavior is sufficiently "close" to the preferred behavior. This paper describes the *relaxation lattice method*, a new approach to specifying graceful degradation for a large class of highly-concurrent fault-tolerant distributed programs. A relaxation lattice is a lattice of specifications parameterized by a set of constraints, where the stronger of set of constraints, the more restrictive the specification. While a program is able to satisfy its strongest set of constraints, it satisfies its preferred specification, but if changes to the environment force it to satisfy a weaker set, then it will permit additional "weakly consistent" computations which are undesired but tolerated. The use of relaxation lattices is illustrated by specifications for programs that tolerate (1) faults, such as site crashes and network partitions, (2) timing anomalies, such as attempting to read a value "too soon" after it was written, and (3) synchronization conflicts, such as choosing the oldest "unlocked" item from a queue.

Copyright©,1987 Maurice P. Herlihy and Jeannette M. Wing

This paper will appear in the Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, Aug 10-12 1987.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J. Wing was provided in part by the National Science Foundation under grant DMC-8519254.

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.

1. Overview

Distributed programs typically display more complex behavior than their single-site counterparts because they must perform efficiently and correctly in the presence of concurrency and failures. Often, such programs must display *graceful degradation*, reacting adaptively to changes in the environment. Under ideal circumstances, the program's behavior satisfies a set of application-dependent *preferred constraints*. Each constraint typically preserves a certain level of consistency, and each has an associated cost. In the presence of failures, timing anomalies, or synchronization conflicts, however, certain constraints may become difficult or impossible to satisfy, and the application designer may choose to relax them as long as the resulting behavior is sufficiently "close" to the preferred behavior.

Although numerous techniques have been proposed for implementing graceful degradation in the presence of concurrency and failures, the resulting behavior has proved difficult to specify using existing techniques. In this paper, we propose the *relaxation lattice method*, a new approach to specifying graceful degradation for a large class of highly-concurrent fault-tolerant distributed programs. This method incorporates sets of constraints into specifications. As with the usual correspondence between specifications and implementations (i.e., programs), the less constraining the specification, the greater the number of possible implementations.

Our specifications have the following advantages:

- They are *high-level* in that the user is not swamped by superfluous implementation details. Our axiomatic specifications require users only to describe desired behavior, not prescribe a model for achieving it.
- They capture *graceful degradation*, showing explicitly how changes in the environment correspond to changes in observable behavior.
- They are concerned only with functional behavior, yet they provide a natural interface to the probabilistic and queuing models commonly used to describe the occurrence of failures and synchronization conflicts.
- They serve as a guide to designers. Given an initial set of constraints, a designer need only decide which subsets represent acceptable and/or meaningful aberrant behaviors.

The relaxation lattice method is applicable to a variety of domains, such as replicated databases and transaction-based systems, each of which has bred its own set of specialized techniques and algorithms satisfying domain-specific properties. As we illustrate in several examples, our approach provides a unified and general framework for evaluating and comparing such techniques, specifying system behaviors, and characterizing the essential trade-offs between the costs of preserving consistency properties and the costs of relaxing them.

In Section 2, we introduce the basic specification method. We present examples illustrating how the method is used for replicated data in Section 3 and for atomic data in Section 4. In Section 5 we close with some remarks, and a discussion of related work.

2. Model

The basic containers for data are called *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 objects of that type. For example, a file might provide Read and Write operations, and a FIFO queue might provide Enq and Deq operations. A computation is modeled as a *history*, which is a finite sequence of executions

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY

of operations on objects; here, we focus on individual object subhistories of a computation.

For an operation (execution) in a history, we write *op(args*)/term(res*)*, where *op* is an operation name, *args** is a sequence of argument values, *term* is a termination condition name, and *res** is a sequence of result values. The operation name and argument values constitute the *invocation*, and the termination condition and result values constitute the *response*. We use "Ok" for normal termination and write "inv(*p*)" for the invocation of operation *p*.

We assume that operations on objects can be executed atomically; that is, an operation either takes place completely or not at all, and operations appear to take place instantaneously with respect to one another. Atomic operations can be implemented by a variety of well-known techniques, including the two-phase locking and two-phase commitment protocols [7, 11], or atomic broadcast protocols [2, 4].

2.1. Simple Object Automata

We model an object by a *simple object automaton*, an automaton that accepts certain sequences of operations. A simple object automaton is a four-tuple $\langle \text{STATE}, s_0, \text{OP}, \delta \rangle$, where STATE is the object's set of states, $s_0 \in \text{STATE}$ is its initial state, OP is a set of operations (the automaton's input alphabet), and δ: $\text{STATE} \times \text{OP} \rightarrow 2^{\text{STATE}}$ is a partial *transition function*.

The domain of the transition function can be extended to histories, $\delta^*: \text{STATE} \times \text{OP}^* \rightarrow 2^{\text{STATE}}$:

$$\delta^*(s, \Lambda) = s$$

$$\delta^*(s, H \cdot p) = \cup_{s' \in \delta^*(s,H)} \delta(s', p)$$

where "$\cdot$" denotes concatenation, and "Λ" denotes the empty history. We use $\delta^*(H)$ as shorthand for $\delta^*(s_0, H)$. A history H is *accepted* by an automaton if $\delta^*(H) \neq \varnothing$. We call $L(A)$, the language accepted by automaton A, the *behavior* of A.

2.2. Relaxation Lattices

Let $\boldsymbol{A}$ be a set of simple object automata having the same set of states, the same initial state, and the same operations, but (possibly) different transition functions. We say that $\boldsymbol{A}$ is a *lattice of automata* if the set $\{L(A) \mid A \in \boldsymbol{A}\}$ is a lattice under reverse inclusion (i.e., the smallest language is at the top). We call the language of the automaton at the top of the lattice the *preferred behavior* of the lattice.

A *relaxation lattice* is given by a set of constraints $\boldsymbol{C}$, a lattice of automata $\boldsymbol{A}$, and a lattice homomorphism,

$$\phi: 2^{\boldsymbol{C}} \rightarrow \boldsymbol{A}.$$

For now, we leave a relaxation lattice's set of constraints uninterpreted since the meaning of such constraints is domain-dependent. In later examples we will see that constraints for replicated objects are of a different nature from those for atomic objects. For now, it suffices to think of each constraint as an assertion to be satisfied. We orient the lattice $2^{\boldsymbol{C}}$ so that the largest (intuitively, the strongest) set of constraints lies at the top, and $\phi(\boldsymbol{C})$ is the preferred behavior of $\boldsymbol{A}$. In general, ϕ is defined over a sublattice of $2^{\boldsymbol{C}}$.

A relaxation lattice is thus a lattice of simple object automata parameterized by a set of constraints, where the stronger the set of constraints, the smaller the language accepted. Informally, a relaxation lattice describes an object's *conditional* behavior. If the environment is such that the object satisfies constraints

$C \subseteq \boldsymbol{C}$, then the object will behave like the simple object $\phi(C)$, accepting the language $\boldsymbol{L}(\phi(C))$. While an object is able to satisfy its strongest set of constraints, it will accept only histories from its preferred behavior. If changes to the environment force the object to satisfy a weaker set, then it will accept additional "weakly consistent" histories, which are undesired but tolerated.

The relaxation method is appropriate for modeling the behavior of objects for which there is a meaningful cost associated with moving up the relaxation lattice. The higher one goes in the lattice, the higher the price paid for the more preferred behavior. In the examples to follow, we use constraints to model the cost of tolerating (1) faults, such as site crashes and network partitions, (2) timing anomalies, such as attempting to read a value "too soon" after it was written, and (3) synchronization conflicts, such as choosing the oldest "unlocked" item from a queue.

2.3. The Environment

The environment determines which behavior, preferred or otherwise, an object exhibits. The environment itself can be represented by an automaton $<2^{\boldsymbol{C}}, c_0, \text{EVENT}, \delta_E>$, where input events in EVENT model changes in the current set of constraints (state), and δ_E: $2^{\boldsymbol{C}} \times \text{EVENT} \rightarrow 2^{\boldsymbol{C}}$ is the transition function (note that δ_E maps to a single state, not a set of states as for object automata). Let $\boldsymbol{A}$ be a lattice of automata, where each A in $\boldsymbol{A}$ is given by the tuple $<\text{STATE}, s_0, \text{OP}, \delta_A>$. The sets EVENT and OP may be disjoint, as in the replicated priority queue example of Section 3.3, or they may be overlapping, as in the bank account and atomic queue examples of Sections 3.4 and 4.2. Let ϕ: $2^{\boldsymbol{C}} \rightarrow \boldsymbol{A}$ be the lattice homomorphism.

The environment and the lattice can be combined into a single automaton that accepts interleaved events and operations:

$$<2^{\boldsymbol{C}} \times \text{STATE}, (c_0, s_0), \text{EVENT} \cup \text{OP}, \delta>$$

Let EVENTOP be EVENT $\cup$ OP. The transition function δ: $2^{\boldsymbol{C}} \times \text{STATE} \times \text{EVENTOP} \rightarrow 2^{\boldsymbol{C}} \times 2^{\text{STATE}}$ is defined by two components, δ_1: $2^{\boldsymbol{C}} \times \text{EVENTOP} \rightarrow 2^{\boldsymbol{C}}$, which defines the effects on the environment state, and δ_2: $2^{\boldsymbol{C}} \times \text{STATE} \times \text{EVENTOP} \rightarrow 2^{\text{STATE}}$, which defines the effects on the lattice state:

$$\delta_1(c, p) = \text{if } p \in \text{EVENT then } \delta_E(c, p) \text{ else } c$$

$$\delta_2(c, s, p) = \text{if } p \in \text{OP} \wedge A = \phi(\delta_1(c, p)) \text{ then } \delta_A(s, p) \text{ else } \{s\}$$

When the (combined) automaton accepts an event, it changes the environment state. When the automaton accepts an operation, it changes the object state, choosing the transition function indicated by the current environment. If the input is both an event and an operation, the environment changes before the transition function is selected. In this paper, we will focus our attention on the lattice $\boldsymbol{A}$, using informal descriptions to characterize the environment.

For many applications, an additional probabilistic model [14] would be used to characterize the likelihood that certain sets of constraints would be satisfied. Indeed, a strength of the relaxation method approach is that it can specify functional behavior independently of probabilistic behavior, while still providing a clean interface between the two domains.

2.4. Specification Language

In our examples, we will use the Larch Specification Language [12] to specify both STATE and δ of a simple object automaton. A state in STATE is a mapping between an object and its *value*, hence it is convenient to represent an object's possible states as a set of values. We use a Larch *trait*, which denotes a first-order theory, to specify an object's values. In a trait, the set of operators and their signatures following **introduces** defines a vocabulary of terms to denote values. For example, from the Bag trait of Figure 2-1, *emp* and *ins(emp, 5)* denote two different bag (multiset) values. The set of equational axioms following the **constrains** clause defines a meaning for the terms, more precisely, an equivalence relation on the terms, and hence on the values they denote. For example, from Bag, one could prove that *del(ins(ins(emp, 3), 3), 3) = ins(emp, 3)*. The **generated by** clause of Bag asserts that *emp* and *ins* are sufficient operators to generate all values of bags. Formally, it introduces an inductive rule of inference that allows one to prove properties of all terms of sort *B*.

```
Bag: trait
 introduces
  emp: → B
  ins: B, E → B
  del: B, E → B
  isEmp: B → Bool
  isIn: B, E → Bool
 constrains B so that for all [b: B, e, e1: E]
  B generated by [ emp, ins ]
    del(emp, e) = emp
    del(ins(b, e), e1) = if e = e1 then b else ins(del(b, e1), e)
    isEmp(emp) = true
    isEmp(ins(b, e)) = false
    isIn(emp, e) = false
    isIn(ins(b, e), e1) = (e = e1) ∨ isIn(b, e1)
```

Figure 2-1: Bag Trait

```
Enq(e)/Ok()
  ensures b' = ins(b, e)

Deq()/Ok(e)
  requires ¬ isEmp(b)
  ensures isIn(b, e) ∧ b' = del(b, e)
```

Figure 2-2: Bag Interfaces

Larch provides three ways of reusing traits: a trait T can **include**, **import**, or **assume** another trait T1. If T1 is included, then T extends the theory denoted by T1 by adding more operators and equations explicitly in T. For example, FifoQ of Figure 2-3 includes Bag and adds two operators, first and rest, and two equations to those of Bag. From FifoQ, one could show that *first(ins(ins(emp, 3), 3)) = 3*. If T1 is imported, then T must be a conservative extension of the theory of T1, i.e., T cannot place further constraints on the operators of T1. All traits implicitly import the Boolean trait, thereby giving meaning to "true" and "false" as they appear in the Bag trait. If T1 is assumed, then T may use T1's operators with their meaning as given in T1; a further use of T is required to discharge the assumption of T1's theory. For example, a trait for priority queues (q.v. Section 3.3) might assume the existence of a total ordering on the items inserted in the queue. With any of the three kinds of reuse, a **with** clause allows renaming of operator and sort identifiers.

```
FifoQ: trait
 includes Bag with [Q for B]
 introduces
    first: Q → E
    rest: Q → E
 constrains Q so that for all [q: Q, e: E]
    first(ins(q, e)) = if isEmp(q) then e else first(q)
    rest(ins(q, e)) = if isEmp(q) then emp else rest(q)
```

Figure 2-3: FIFO Queue Trait

```
Enq(e)/Ok()
   ensures q' = ins(q, e)

Deq()/Ok(e)
   requires ¬ isEmp(q)
   ensures q' = rest(q) ∧ e = first(q)
```

Figure 2-4: FIFO Queue Interfaces

We use Larch *interfaces* to describe transition functions for simple object automata. For example, interfaces for the Enq and Deq operations for FIFO queues are shown in Figure 2-4. The object's identifier, e.g., *q*, is an implicit argument and return formal of each operation. A **requires** clause states the precondition that must hold when an operation is invoked. An omitted **requires** clause is interpreted as equivalent to **requires** true. An **ensures** clause states the postcondition that the operation must establish upon termination. An unprimed argument formal, e.g., *q*, in a predicate stands for the value of the object when the operation begins. A return formal or a primed argument formal, e.g., *q'*, stands for the value of the object at the end of the operation. For an object *x*, the absence of the assertion $x' = x$ in the postcondition states that the object's value may change.

For an operation, *p*, of a simple object automaton, A, we write $p.pre_A$ and $p.post_A$ for the pre- and postconditions of *p*. The transition function δ for A is defined such that

$$(\forall\, s, s' \in \text{STATE})\ s' \in \delta(s, p) \text{ iff } p.pre_A(s) \wedge p.post_A(s, s').$$

We use the vocabulary of traits to write the assertions in the pre- and postconditions of an object's operations; we use the meaning of equality to reason about its values. Hence, the meaning of *ins* and = in Enq's postcondition is given by the FifoQ trait. Notice that the terms that denote values for FIFO queues and for bags are generated by the same trait operators, *emp* and *ins*, but their operations, Enq and Deq, differ. We will be revisiting these two specifications in later examples.

3. First Example Domain: Replicated Objects

A *replicated* object is one that is stored redundantly at multiple sites in a distributed system. Replication can enhance the availability, reliability, and accessibility of data. A *replication method* is a technique for managing replicated objects. A widely-accepted correctness criterion for replication methods is *one-copy serializability* [1], which states that the functional behavior of a replicated object should be identical to the functional behavior of an analogous single-site object. That is, except for availability, replication should be transparent. Although one-copy serializability is a natural and attractive correctness property, a

number of researchers [3, 8, 18] have investigated weaker notions of correctness. The motivation behind these efforts is the perception that strict one-copy serializability is sometimes too expensive in terms of *availability*, the likelihood the operation execution will succeed, and in terms of *latency*, the duration the caller must wait for the operation to complete.

In this section we outline how specifications based on relaxation lattices can express the behavior of a number of "weakly consistent" replication methods from the literature without sacrificing one-copy serializability as the basic correctness condition. Each of the weakly consistent methods is based on the observation that availability and latency costs can be reduced by performing updates at a small number of sites, relaying updates to be propagated asynchronously, perhaps as inaccessible sites rejoin the system. This technique gives rise to transient inconsistencies which are tolerated because the resulting behavior is considered sufficiently "close" to the preferred behavior.

3.1. Constraints on Replicated Objects

We begin with an informal review of quorum consensus to motivate the kinds of constraints that are meaningful for replicated objects. (A more complete discussion appears elsewhere [13].) A replicated object's state is represented as a *log*, which is a sequence of *entries*, where an entry is the timestamped record of an operation. Timestamps are generated by logical clocks [16]. For example, the following is a schematic representation of a queue replicated among three sites: S1, S2, and S3.

S1	S2	S3
1:01 Enq(x)/Ok()	1:01 Enq(x)/Ok()	
	1:03 Enq(y)/Ok()	1:03 Enq(y)/Ok()
2:02 Enq(z)/Ok()		2:02 Enq(z)/Ok()

A missing entry is denoted by an empty space. The queue's current value is *ins(ins(ins(emp,x),y),z)*, which can be reconstructed by merging the entries in timestamp order, discarding duplicates.

A client executes an operation in three steps:

1. The client merges the logs from an *initial quorum* of sites for the invocation to construct a *view* representing a subhistory of the object's current history.
2. The client chooses a response consistent with the view, and appends the new entry to the view.
3. The client sends the updated view to a *final quorum* of sites for the operation. Each site in the final quorum merges the view with its resident log.

A *quorum* for an operation is any set of sites that includes both an initial and a final quorum for that operation. A *quorum assignment* associates each operation with its initial and final quorums.

An object's quorum assignment determines the availability of its operations, and the constraints governing quorum assignment are the fundamental constraints governing the availability realizable by quorum consensus replication. These constraints take the form of requirements that certain initial and final quorums intersect. In the replicated queue example, a client executing a Deq can tell which item to dequeue only if it is able to observe the effects of earlier Enq and Deq operations, thus each initial quorum for Deq must intersect each final quorum for both Enq and Deq. In general, a replicated object's behavior is determined by its *quorum intersection relation* Q between invocations and operations: *inv(p)* Q *q* if each initial quorum for the invocation of the operation *p* has a non-empty intersection with each final

quorum for the operation *q*.

3.2. Quorum Consensus Automata

Given a simple object automaton A and a quorum intersection relation Q, the quorum consensus protocol implements the following *quorum consensus automaton* QCA(A,Q).

> **Definition 1:** G is a *Q-closed* subhistory of H if whenever it contains an operation *p* it also contains every earlier operation *q* of H such that *inv(p)* Q *q*.
>
> **Definition 2:** G is a *Q-view* of H for an operation *p* if (1) G includes every operation *q* such that *inv(p)* Q *q*, and (2) G is Q-closed.

The (quorum consensus) automaton's operations are identical to those of A, and the automaton's state is simply the history it has accepted so far. The transition function is defined in terms of Q and the pre- and postconditions of A's operations as follows: Let H be the automaton's current state. There exists G, a Q-view of H for *p*, *s* in $\delta^*(G)$, and *s'* in $\delta^*(G \cdot p)$ such that:

requires $p.pre_A(s)$
ensures $p.post_A(s, s') \wedge H' = H \cdot p$

Informally, G corresponds to the view constructed by merging the logs from an initial quorum for *p*. The view must satisfy the precondition for *p*, and the result of appending *p* to the view must satisfy the postcondition. If the pre- and postconditions are satisfied, the operation is recorded at a final quorum.

The standard notion of one-copy serializability is extended to typed objects as follows: QCA(A,Q) is *one-copy serializable* if L(QCA(A,Q)) = L(A). Quorum consensus replication guarantees one-copy serializability if and only if the quorum intersection relation Q satisfies the following condition:

> **Definition 3:** Q is a *serial dependency relation* for A if, for all histories G and H in L(A) such that G is a Q-view of H for *p*, $G \cdot p \in L(A) \Rightarrow H \cdot p \in L(A)$.

Let Q be a *minimal* serial dependency relation, meaning that no $R \subset Q$ guarantees one-copy serializability. $L(QCA(A,Q)) \subseteq L(QCA(A,R))$, since every history accepted by the former is accepted by the latter, thus the set $\{QCA(A,R) \mid R \subseteq Q\}$ is a lattice of automata, and the lattice homomorphism $\phi(R)$ = QCA(A,R) defines a relaxation lattice. As illustrated in the next two sections, these relaxed automata typically provide higher availability (because they impose fewer restrictions on quorums), at the cost of more complex behavior (because they accept histories not in L(A)).

Additional flexibility can be achieved by adding a third parameter to a quorum consensus automaton: an *evaluation* function η: $\text{STATE} \times \text{OP}^* \rightarrow 2^{\text{STATE}}$ that is required to agree with the transition function δ^* on histories in L(A). Informally, η is an extension of δ^* that allows us to assign an application-specific meaning to histories not in L(A). The automaton QCA(A,Q,η) is defined identically to QCA(A,Q) except that η replaces δ^* in the above **requires** and **ensures** clauses. If Q is a serial dependency relation for L(A), then $L(A) = L(QCA(A,Q)) = L(QCA(A,Q,\eta))$. The set $\{QCA(A,R,\eta) \mid R \subseteq Q\}$ is also a lattice of automata, although different choices of η may produce different lattices.

3.3. Example 1: A Real-Time Priority Queue

Consider an urban taxicab company, whose customers make telephone requests to dispatchers. The dispatchers assign priorities to requests and enqueue them in a priority queue. Whenever a taxicab is idle, the driver dequeues the highest priority pending request. Figures 3-1 and 3-2 describe the preferred

behavior of a priority queue automaton.

PQueue: **trait**
 assumes TotalOrder **with** [E **for** T] % > denotes the total order relation
 includes Bag **with** [PQ **for** B]
 introduces
  best: PQ → E
 constrains [best] **so that for all** [q: PQ, e: E]
  best(ins(q, e)) = **if** isEmp(q) **then** e
                else if e > best(q) **then** e **else** best(q)

Figure 3-1: Priority Queue Trait

Enq(e)/Ok()
  ensures q' = ins(q, e)

Deq()/Ok(e)
  requires ¬ isEmp(q)
  ensures e = best(q) ∧ q' = del(q, e)

Figure 3-2: Priority Queue Interfaces

Because the availability of the priority queue is critical, it is replicated at several sites throughout the city. We assume sites can crash, and that communication is unreliable (e.g., packet radio). Thus, the events in EVENT of the environment automaton (Section 2.3) include site crashes and communication failures, which can cause the priority queue to exhibit undesired behavior. Notice that these crash and failure events are disjoint from the Enq and Deq operations of the priority queue automaton.

The following set of constraints is necessary and sufficient for a one-copy serializable implementation of a replicated priority queue [13].

Q_1 Each initial Deq quorum intersects each final Enq quorum.

Q_2 Each initial Deq quorum intersects each final Deq quorum.

Constraint Q_1 implies that the availability of Enq and Deq can be traded off: if one operation's quorums are made smaller (rendering that operation more available), then the quorums for the other operation must be made larger to preserve the intersection property (rendering that operation less available). If quorums are established by voting [10], then Q_2 implies each Deq quorum must encompass a majority of votes.

Although such a replicated queue is more available than a single-site queue, it is still possible that a dispatcher or cab driver might be unable to locate a quorum for an operation. The taxicab application is subject to "soft" real-time constraints — customers are unlikely to wait until crashed sites recover or communication links are restored. Under such circumstances, it seems sensible to settle for behavior that is reasonably "close," for the purposes of the application, to the preferred behavior.

Since an operation's availability is determined by its set of quorums, and since those quorums are determined by the intersection constraints given above, it is natural to enquire how the queue would behave if we were to relax the constraints on quorum intersection, permitting the dispatchers and drivers to enqueue and dequeue requests from all available sites. This relaxed behavior can be specified as a relaxation lattice, $\{QCA(PQ,Q,\eta) \mid Q \subseteq \{Q_1, Q_2\}\}$ where η is the following evaluation function[1]:

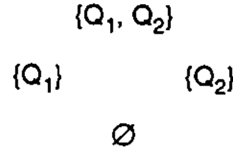
[1] $\eta(H)$ is shorthand for $\eta(s_0, H)$.

$\eta(\Lambda) = emp$

$\eta(H \cdot Enq(e)/Ok()) = ins(\eta(H),e)$

$\eta(H \cdot Deq()/Ok(e)) = del(\eta(H),e)$

Although η agrees with the priority queue's transition function on legal priority queue histories, it is defined for arbitrary sequences of Enq and Deq operations, not just for legal priority queue histories. This particular choice of η implies that each driver will dequeue the highest-priority request that appears not to have been served. Visually, the lattice of constraints looks like:



Henceforth, for notational convenience we write Q_1 (Q_2) for the set $\{Q_1\}$ ($\{Q_2\}$). We now discuss in turn each of the degraded behaviors corresponding to the three elements of the lattice: Q_1, Q_2, and $\varnothing$.

If we relax the constraint that Deq quorums must intersect, then requests may be serviced multiple times (i.e., by dispatching multiple taxicabs to the same customer), but customers are serviced in turn: no unserviced higher-priority request will ever be passed over in favor of an unserviced lower-priority request. More precisely, we claim the automaton $QCA(PQ,Q_1,\eta)$ is a one-copy-serializable implementation of the *multi-priority queue* automaton MPQ shown in Figure 3-3. This automaton's state is a two-component record: the *present* component is a bag of items (requests) that have been enqueued but not dequeued, and the *absent* component is a bag of previously enqueued items that have been dequeued. The MPQ automaton's transition function is as follows: Enq inserts an item in *present*, and Deq either transfers the best item from *present* to *absent* and returns it, or it returns an item from *absent* whose priority is greater than that of any item in *present*.

Theorem 4: $L(QCA(PQ,Q_1,\eta)) = L(MPQ)$.

Proof: We first show that $L(QCA(PQ,Q_1,\eta)) \subseteq L(MPQ)$. Q_1 is a serial dependency relation for MPQ (Definition 3), hence $L(QCA(MPQ,Q_1)) = L(MPQ)$, and so it suffices to show that $L(QCA(PQ,Q_1,\eta)) \subseteq L(QCA(MPQ,Q_1))$.

Let δ be the transition function for MPQ. The postconditions of multi-priority queue's interfaces completely determine the new value of the queue. Thus for all H in $L(MPQ)$, $\delta^*(H)$ is a singleton set, and we simplify our notation by treating δ^* as a function from histories to MPQ values, rather than sets of MPQ values. Define α: MPQ $\rightarrow$ PQ to be the (value) homomorphism defined by projecting on the first component of the MPQ value: $\alpha(m) = m.present$.

If p is Enq or Deq, it is easy to check that:

$$p.pre_{PQ}(\alpha(\delta^*(H))) \Rightarrow p.pre_{MPQ}(\delta^*(H))$$
$$\wedge$$
$$p.post_{PQ}(\alpha(\delta^*(H))) \Rightarrow p.post_{MPQ}(\delta^*(H))$$

We argue inductively that $\alpha(\delta^*(H)) = \eta(H)$ for all histories H in $L(MPQ)$. The base case is immediate:

$\alpha(\delta^*(\Lambda)) = \eta(\Lambda) = emp$.

Assume the result for all non-empty histories. Let $H' = H \cdot Enq(e)/Ok()$, $m = \delta^*(H)$, and $m' = \delta^*(H')$. By the Enq postcondition for MPQ, $m'.present = ins(m.present,e)$, hence $\alpha(\delta^*(H')) =$

ins($\alpha(\delta^*(H)),e$). By the induction hypothesis, $\eta(H) = \alpha(\delta^*(H))$, hence $\eta(H') = \alpha(\delta^*(H'))$. If $H' = H \cdot$ Deq()/Ok(e), the same argument holds with *del* replacing *ins*. Thus, by substitution:

$$p.pre_{PQ}(\eta(H)) \Rightarrow p.pre_{MPQ}(\delta^*(H))$$
$$\wedge$$
$$p.post_{PQ}(\eta(H)) \Rightarrow p.post_{MPQ}(\delta^*(H))$$

which is enough to show that $L(QCA(PQ,Q_1,\eta)) \subseteq L(MPQ)$. Note that the preconditions for both Enq's are true, and $Deq.pre_{MPQ}$ is true, thus making the first implication for p = Deq trivially true.

To show that $L(MPQ) \subseteq L(QCA(PQ,Q_1,\eta))$, we also argue by induction. Let H be a history in $L(MPQ)$ and $L(QCA(PQ,Q_1,\eta))$ such that $H \cdot p$ is in $L(MPQ)$. If p is Enq(x)/Ok() for some x, $H \cdot p$ is clearly in $L(QCA(PQ,Q_1,\eta))$. Suppose p is Deq()/Ok(x). If x is in *present*, choose a view that includes all Deq operations. If x is in *absent*, choose a view that includes all Deq operations except earlier Deq's for x.

If we relax the constraint that Enq and Deq quorums must intersect, then requests may be serviced out of order, but no request will be serviced more than once. More precisely, the automaton $QCA(PQ,Q_2,\eta)$ is a one-copy serializable implementation of the *out-of-order priority queue* automaton OPQ given in Figure 3-4. The behavior of an OPQ is just a bag (Figures 2-1 and 2-2). Enq inserts an item in the bag and Deq removes an item, although not necessarily the best one. The argument that $L(QCA(PQ,Q_2,\eta)) = L(OPQ)$ is similar to that given for Theorem 4, and is omitted.

Finally, if we relax both constraints Q_1 and Q_2, the result is a *degenerate priority queue* (Figure 3-5) which permits clients to be serviced multiple times and out of order. The automaton's set of states is given by the Bag trait of Figure 2-1, although its behavior is slightly different: Enq inserts an item in the bag, and Deq returns (but does not necessarily remove) some item in the bag.

When designing a relaxation lattice, the exact way in which the evaluation function η should extend the transition function δ^* is application dependent. For example, we might equally well have chosen an evaluation function η' that deletes higher-priority requests that had been skipped over in favor of lower-priority requests. The resulting lattice would produce a different set of relaxed behaviors: unlike $QCA(PQ,Q_2,\eta)$, $QCA(PQ,Q_2,\eta')$ never services requests out of order, but it could ignore certain requests.

Finally, we illustrate (informally) how a model of probabilistic behavior fits in our relaxation lattice method. The likelihood the queue will satisfy a particular quorum intersection relation would be given by an independent probabilistic model taking into account estimates of crashes and communication failures. As a simplistic example, suppose the environment is such that each queue operation satisfies Q_1 with independent probability 0.9, and Deq operations are certain to satisfy Q_2. The likelihood a Deq will fail to return an item whose priority is within the top n is $(0.1)^n$.

3.4. Example 2: A Replicated Bank Account

Constraints on quorum intersection can be used to model the effects of timing anomalies as well as faults. The cost incurred in attaining a more preferred behavior is the amount of time one is willing to wait for certain operations to complete. For example, consider a bank with a system of automatic teller machines (ATM). Customers' accounts are replicated at multiple branch offices. Each account provides Credit and Debit operations, where Debit returns an exception if the balance would become negative. The following

```
MPQueue: trait
  assumes TotalOrder with [E for T]   % > denotes the total order relation
  includes Bag with [Q for B],
    MPQ record of [present: Q, absent: Q]
  introduces
    best: Q → E
 constrains [best, isIn] so that for all [pq, PQ, q: Q, e: E]
   best(ins(q, e)) = if isEmp(q) then e
                                    else if e > best(q) then e else best(q)

Enq(e)/Ok()
  ensures q'.present = ins(q.present, e)

Deq()/Ok(e)
  ensures
   (isIn(q.absent, e) ∧ e > best(q.present)) ∨
   (e = best(q.present) ∧
    q'.absent = ins(q.absent, e) ∧
    q'.present = del(q.present, e))
```

Figure 3-3: Multi-Priority Queue

```
OPQ: trait
  includes Bag

Enq(e)/Ok()
  ensures q' = ins(q, e)

Deq()/Ok(e)
  requires ¬ isEmp(q)
  ensures isIn(q, e) ∧ q' = del(q, e)
```

Figure 3-4: Out-of-Order Priority Queue

```
DegenPQ: trait
  includes Bag

Enq(e)/Ok()
  ensures q' = ins(q, e)

Deq()/Ok(e)
  requires ¬ isEmp(q)
  ensures isIn(q, e)
```

Figure 3-5: Degenerate Priority Queue

is a necessary and sufficient set of constraints on quorum intersection for the account data type:

A_1 Every initial Debit quorum intersects every final Credit quorum.

A_2 Every initial Debit quorum intersects every final Debit quorum.

The larger an operation's quorums, the longer it takes to execute that operation. Rather than forcing customers to wait for all the updates to complete, the bank's ATM's might be reprogrammed to announce success as soon as any update is complete, assuming that the remaining updates can be performed in the background. This strategy is equivalent to allowing the operations' final quorums to grow asynchronously, and as long as updates to the same account do not occur too close together, the bank account will satisfy both constraints A_1 and A_2. A similar approach is taken in Locus [18] and Grapevine

[3].

Nevertheless, the bank naturally wishes to preserve the semantic consistency property that no account can be overdrawn, although it is not averse to bouncing checks spuriously. To preserve this property, the account object may relax constraint A_1, but not A_2— the relaxation lattice is defined over a sublattice of $2^{\{A_1, A_2\}}$. In other words, Debit operations must access a majority of sites, while Credit operations may be propagated when it is convenient to do so. Here, Credit quorums effectively grow in time. The environment events that cause constraint A_1 to be violated are "premature" debits executed before the effects of earlier credits have had time to propagate. The probability that an ATM performing a debit would fail to observe an earlier credit would diminish in time. Note that unlike the priority queue example, the object's set of operations and the environment's set of events are not disjoint.

4. Second Example Domain: Atomic Objects

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable and recoverable. *Serializability* means the execution of one transaction never appears to overlap (or contain) the execution of another, and *recoverability* means that a transaction either succeeds completely or has no effect. A transaction's effects become permanent when it *commits*, its effects are discarded if it *aborts*, and a transaction that has neither committed or aborted is *active*.

Atomicity is the basic correctness condition for objects accessed by multiple transactions. Although atomicity, like one-copy serializability, is a simple and appealing correctness condition, several researchers have suggested that weaker notions of correctness are necessary to support an adequate level of concurrency [9, 19]. In this section, we show how specifications based on relaxation lattices can capture the behavior of highly concurrent distributed applications without replacing atomicity with *ad hoc* notions of correctness. Our approach extends and formalizes that of Liskov and Weihl [17, 20], who have proposed that concurrency can be enhanced by introducing non-determinism into specifications of atomic objects. We believe that relaxation lattices are simpler and easier to use than techniques that require discarding atomicity, yet they have more expressive power than techniques that use non-determinism to mask anomalous behavior.

4.1. Atomic Object Automata

Let A be a simple object automaton. A *schedule* for A is a history of operations of the form <*p* P>, where *p* is either an operation of A, *commit*, or *abort*, and P is a *transaction identifier*. A schedule is *well-formed* if (1) no transaction has executed both a commit operation and an abort operation, and (2) no transaction executes any operation after a commit or abort operation.

Informally, a schedule for A is serializable if it is equivalent to a history for A in which transactions execute serially. More precisely, if H is a schedule for A, let H|P denote the history of operations of A executed by P in H.

> **Definition 5:** A schedule is *serializable* if there exists a total order < on transactions whose identifiers appear in H such that $H|P_1 \cdot \ldots \cdot H|P_n$ is in $L(A)$, where $P_1,\ldots,P_n$ are the transactions in H in the order <.

Let perm(H) be the subschedule of H consisting of operations of committed transactions.

> **Definition 6:** H is *atomic* if perm(H) is serializable.

Most techniques for implementing atomicity are *on-line*: the scheduler does not know in advance which transactions will commit and which will abort.

> **Definition 7:** A schedule H is *on-line atomic* if the result of appending commit operations for any subset of active transactions is atomic.

An *atomic object automaton* Atomic(A) is an automaton that accepts schedules of the simple object automaton A such that every schedule in L(Atomic(A)) is well-formed and on-line atomic.

All known techniques for implementing atomicity permit only a subset of the well-formed on-line atomic schedules. To make our examples as explicit as possible, we make the further assumption that all schedules in L(Atomic(A)) are *hybrid atomic* [21]: transactions are serializable in the order they commit. This property is guaranteed by a number of atomicity mechanisms in common use, including strict two-phase locking [7]. Our examples can easily be adapted to other atomicity properties.

4.2. Relaxing FIFO Queues

Consider a printing service in which a collection of clients spool files to be printed by a collection of printers. Client transactions spool their files on a single queue, and each printer controller executes transactions in which it dequeues the next file to be printed, prints it, and commits. Ideally, the spooling queue should be FIFO: files should be dequeued for printing in the order they were enqueued. Nevertheless, because the queue is shared among multiple clients and printer controllers, concurrency is important. Although clients can enqueue files without interference, the FIFO ordering cannot be guaranteed if two controllers are allowed to dequeue files concurrently, thus one dequeuing transaction must be delayed until the other commits or aborts. Such behavior is clearly ill-suited to the application; it is enough that the queue be "approximately" FIFO. In particular, the queue should be FIFO as long as transactions execute serially.

We can use relaxation lattices to formulate two alternative "gracefully degrading" queue specifications. In each case, the extent to which the queue departs from FIFO behavior depends on the level of concurrency. Suppose a transaction executing a Deq observes that a concurrent transaction has tentatively dequeued the item at the head of the queue. Instead of waiting for the concurrent dequeuer to commit or abort, an implementation might permit a dequeuing transaction to proceed in one of two ways:

- Optimistically assuming the earlier dequeuer will commit, the transaction skips the first item and returns the next undequeued item in the queue.
- Pessimistically assuming the earlier dequeuer will abort, the transaction ignores the pending dequeue and returns that same item.

As long as dequeuing transactions execute serially, each of these alternative implementations yields a FIFO queue. If dequeuing transactions overlap, however, the first implementation permits files to be printed out of order, but each file is printed only once, while the second permits files to be printed multiple times, but files are always printed in the order they were enqueued. Rather than viewing these implementations as "weakly consistent" FIFO queues, we view each as an atomic object automaton distinct from the FIFO queue.

For our examples, the constraints of interest are the number of Deq operations executed by active transactions. Let C_k denote the constraint that no more than k active transactions have executed Deq operations. The set of constraints $\mathbf{C}$ is $\{C_k \mid k > 0\}$. For each of the implementations sketched above, the lattice homomorphism ϕ assigns a behavior to each element in the lattice of constraints $2^{\mathbf{C}}$. As long as no

more than *k* dequeuing transactions attempt to access the queue concurrently, the object's behavior will be given by an atomic object automaton Atomic($\phi(C_k)$). While C_k is satisfied the behavior of the "optimistic" implementation is *L*(Atomic(Semiqueue$_k$)), and the behavior of the "pessimistic" implementation is *L*(Atomic(Stuttering$_j$ Queue)), where Semiqueue$_k$ and Stuttering$_j$ Queue are defined in the next two sections.

The events that affect the environment (Section 2.3) are the operations that affect the number of concurrent dequeuers: the Deq, *commit*, and *abort* operations. Like the bank account example, the object's set of operations and the environment's set of events are not disjoint. A probabilistic model of the environment could be expressed in terms of the distributions of transaction arrivals, durations, and success rates.

4.2.1. Semiqueues

A Semiqueue$_k$ object (Figure 4-1) is a sequence of items. The Enq operation inserts an item in the sequence, and the Deq deletes and returns one of the first *k* items in the queue. It is straightforward to show that if *k* is one, the object is a FIFO queue (Figures 2-3 and 2-4) and if *k* is *n*, the maximum number of items allowed in the queue, the object is a bag (Figures 2-1 and 2-2). Weihl and Liskov [20] give an example implementation of the semiqueue data type written in Argus.

```
SemiQ: trait
 imports Integer
 includes FifoQ, Set with [SetE for C]
 introduces
  prefix: Q, Int → SetE
 constrains [prefix] so that for all [q: Q, i: Int]
  prefix(q, i) = if (i = 0 ∨ isEmp(q)) then {}
                                   else prefix(rest(q), i-1) ∪ {first(q)}

Enq(e)/Ok()
  ensure q' = ins(q, e)

Deq()/Ok(e)
  requires ¬ isEmp(q)
  ensures q' = del(q, e) ∧ e ∈ prefix(q, k)
```

Figure 4-1: Semiqueue$_k$

The relaxation lattice is defined as follows. The set of constraints ***C*** is as defined above. The lattice homomorphism ϕ is defined over the sublattice of nonempty elements of ***C***: $\phi(B)$ = Semiqueue$_k$, where C_k is the element of B with the lowest index. For example, the constraint and behavior lattices for a three-item queue are shown in Figure 4-2. Notice that ϕ is a lattice homomorphism, not an isomorphism as in the replication example. Moreover, if the queue is unbounded, then the lattice of behaviors is infinite.

Constraints	Behavior
$\{C_1\}$, $\{C_1, C_2\}$, $\{C_1, C_2, C_3\}$	Semiqueue$_1$ (FIFO queue)
$\{C_2\}$, $\{C_2, C_3\}$	Semiqueue$_2$
$\{C_3\}$	Semiqueue$_3$ (bag)

Figure 4-2: Relaxation Lattice for a Three-Item Semiqueue

4.2.2. Stuttering Queues

A Stuttering_j Queue object (Figure 4-3) is like a FIFO queue except that the first item in the queue may be returned as many as *j* times. The relaxation lattice is similar to that for semiqueues: The lattice of automata is {Stuttering_j Queue | $j > 0$}, and the lattice homomorphism ϕ is defined over the sublattice of nonempty elements B of ***C***: $\phi(B) = \text{Stuttering}_j$ Queue, where C_j is the element of B with the lowest index.

The stuttering queue and semiqueue behaviors can be combined within a single lattice: the SSqueue_{jk} behavior would permit any of the first *k* items to be returned as many as *j* times. SSqueue_{11} is a FIFO queue.

```
StutQ: trait
  imports Integer
  includes FifoQ
  StQ record of [items: Q, count: Int]

Enq(e)/Ok()
  ensures q'.items = ins(q.items, e)

Deq()/Ok(e)
  requires ¬ isEmp(q.items)
  ensures q.count < j ⇒ [e = first(q.items) ∧
                          [[q'.count = q.count + 1 ∧ q'.items = q.items] ∨
                          [q'.count = 0 ∧ q'.items = rest(q.items)]]]
```

Figure 4-3: Stuttering_j Queue

5. Remarks and Related Work

In summary, our relaxation lattice method suggests the following design strategy:

- Identify a set of constraints ***C*** that characterizes the preferred behavior. This set induces a lattice 2^{C}.
- Not all elements in the lattice may correspond to an intuitively meaningful behavior, let alone an acceptable one. The homomorphism ϕ determines which elements in the lattice of automata represent acceptable behaviors.
- Given the lattices of constraints and automata, the cost function determines the price one must pay in moving up the lattice of automata toward the preferred behavior.

The relaxation lattice method is a natural way to capture graceful degradation in distributed systems. Moreover, the method is quite flexible. In this paper, we have reviewed three applications: a replicated priority queue, a replicated bank account, and an atomic queue. In each case, as summarized in Figure 5-1, the domain-specific correctness condition together with the preferred behavior impose a set of constraints on the implementation. These constraints impose a cost, which can be affected by environment events. These costs can be alleviated by relaxing the constraints, potentially producing "degraded" behavior. These trade-offs are captured naturally as a homomorphism between the lattice of constraints and the corresponding lattice of behaviors.

The lattice structure also permits us to compare specifications. As in Larch, if a specification is considered to denote a theory, i.e., set of formulae, then specifications may be compared by comparing the strengths of their theories [22], where it may be necessary to introduce *theory interpretations*, i.e.,

Correctness condition	*Preferred Behavior*	*Constraints*	*Cost*	*Events*
One-copy serializability	Priority Queue	Quorum intersection	Availability	Failures, crashes
One-copy serializability	Account	Quorum intersection	Latency	Premature Debits
Atomicity	FIFO Queue	Concurrent Deq's	Concurrency	Deq, commit, abort

Figure 5-1: Summary Chart

mappings between theories. Maibaum and others [15] use this notion of theory interpretation to define a database view as an interpretation between two different database specifications. Thus, instead of using a set of constraints to induce a lattice of behaviors under inclusion, a more general approach would have been to start with a lattice of predicates under implication or a lattice of theories under containment.

Nevertheless, we believe that sets of constraints are easier to work with than lattices of unstructured theories or specifications. Making constraints explicit forces the designer to compare the costs of satisfying the constraints with the complexity of the unconstrained behavior. As illustrated by the replication examples, generating the lattice of weaker quorum intersection relations effectively enumerates the possible trade-offs. Sometimes all trade-offs are acceptable, as in the taxi queue example, and sometimes certain trade-offs are unacceptable, as in the bank account example.

Our relaxation method captures precisely the intuition behind the informal specification method of Liskov and Weihl [20, 17]. Their method recognizes only two kinds of behavior: best case (normal) and worst case (abnormal). These behaviors are the informal counterparts of the specifications at the top and bottom of our relaxation lattice. The use of explicit constraints adds expressive power to our specifications by focusing attention on intermediate behaviors, providing a natural way to capture graceful degradation: the extent to which an object's behavior departs from its preferred specification is proportionate to the gravity of the faults that affect it. For example, while Liskov and Weihl's method might specify only that a printer spooler behaves either like a FIFO queue or like a bag, our method can make stronger statements, e.g., in a system where no more than k transactions concurrently access a semiqueue, no item will be dequeued out of order with respect to more than k items.

Furthermore, our method allows for a clean interface to probabilistic and queueing models typically used to describe a system's reliability. Separate functional and probabilistic models can be combined without compromising the expressive power of either. An alternative approach is to capture both kinds of properties in a single model, as in Durham and Shaw's analysis of a fault-tolerant parallel quicksort algorithm [6] or Cristian's [5] Markovian analysis of a two-disk stable storage implementation. Our approach, however, permits functional and probabilistic properties to be understood in isolation, which we believe makes our specifications easier to understand, more flexible, and more readily applicable to large and realistic problems.

Acknowledgments

We thank the anonymous referees for their suggestions and comments.

References

[1] P.A. Bernstein and N. Goodman.
The failure and recovery problem for replicated databases.
In *Proceedings, 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* 1983.
Montreal, Quebec.

[2] K.P. Birman.
Replication and fault-tolerance in the ISIS system.
In *Proc. 10th Symposium on Operating Systems Principles.* December, 1985.
Also TR 85-668, Cornell University Computer Science Dept.

[3] A.D. Birrell, R. Levin, R. Needham, and M. Schroeder.
Grapevine: an exercise in distributed computing.
Communications of the ACM 25(14):260-274, April, 1982.

[4] J. Chang and N.F. Maxemchuk.
Reliable broadcast protocols.
ACM Transactions on Computer Systems 2(3):251-273, August, 1984.

[5] F. Cristian.
A rigorous approach to fault-tolerant system development.
Technical Report RJ 4008, IBM Research Laboratory, September, 1983.

[6] I. Durham and M. Shaw.
Specifying reliability as a software attribute.
Technical Report CS-82-148, Carnegie-Mellon University, December, 1982.

[7] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The notion of consistency and predicate locks in a database system.
Communications of the ACM 19(11):624-633, November, 1976.

[8] M. Fischer and A. Michael.
Sacrificing serializabilithy to attain high availability of data in an unreliable network.
In *Proceedings, ACM SIGACT-SIGMOD Symp. on Principles of Database Systems.* March, 1982.

[9] H. Garcia-Molina.
Using semantic knowledge for transaction processing in a distributed database.
ACM Transactions on Database Systems 8(2):186-213, June, 1983.

[10] D.K. Gifford.
Weighted voting for replicated data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM SIGOPS, December, 1979.

[11] J. Gray.
Notes on database operating systems.
Lecture Notes in Computer Science 60.
Springer-Verlag, Berlin, 1978, pages 393-481.

[12] J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch family of specification languages.
IEEE Software 2(5):24-36, September, 1985.

[13] M.P. Herlihy.
A quorum-consensus replication method for abstract data types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[14] J.G. Kemeny, J.L. Snell, A.W. Knapp.
Graduate Texts in Mathematics. Volume 40: *Denumerable Markov Chains.*
Springer-Verlag, New York, 1976.

[15] S. Khosla, T.S.E. Maibaum, and M. Sadler.
Large database specifications from small views.
In *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science (LNCS 206)*, pages 246-271. Springer-Verlag, Berlin, 1985.

[16] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[17] B.H. Liskov and W.E. Weihl.
Specifications of distributed programs.
Distributed Computing 1(2):102-118, April, 1986.

[18] G.J. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel.
Locus: a network transparent high reliability distributed system.
In *Proceedings, Eighth Symposium on Operating Systems Principles.* December, 1981.

[19] P.M. Schwarz and A.Z. Spector.
Synchronizing shared abstract types.
ACM Transactions on Computer Systems 2(3):223-250, August, 1984.

[20] W.E. Weihl and B.H. Liskov.
Specification and implementation of resilient, atomic data types.
In *Proc. SIGPLAN Symposium on Programming Language Issues in Software Systems.* June, 1983.

[21] W.E. Weihl.
Specification and implementation of atomic data types.
Technical Report TR-314, Massachusetts Institute of Technology Laboratory for Computer Science, March, 1984.

[22] J.M. Wing.
A two-tiered approach to specifying programs.
Technical Report MIT-LCS-TR-299, MIT Laboratory for Computer Science, June, 1983.