Specifying Graceful Degradation

Maurice P. Herlihy, *Member, IEEE,* and Jeannette M. Wing, *Member, IEEE*

Abstract— **Complex programs are often required to 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 events such as failures, timing anomalies, synchronization conflicts, or security breaches, 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 programs. A relaxation lattice is a lattice of specifications parameterized by a set of constraints, where the stronger the 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, 3) synchronization conflicts, such as choosing the oldest "unlocked" item from a queue, and 4) security breaches, such as acquiring unauthorized capabilities. A preliminary version of this paper appeared in the proceedings of the Sixth ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing [17].**

Index Terms— **Concurrency, distributed computing, fault tolerance, formal specification, Larch.**

I. Overview

COMPLEX programs are often required to 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, synchronization conflicts, or security violations, 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 a variety of domains, 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 programs, including sequential, concurrent, and 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 queueing 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, transactional systems, and secure operating systems, each of which has bred its own set of specialized techniques and algorithms satisfying domain-specific correctness 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 tradeoffs between the costs of preserving correctness properties and the costs of relaxing them.

An important aspect of our method is to make explicit the role of the environment in its effect on the observable behavior of the rest of the system. In particular, we hold the environment responsible for catastrophic, unpredictable and/or anomalous events. Our method provides a way to explain the behavior of the rest of the system when such failures occur. Though appropriate for the simpler contexts of sequential programs and abstract data types, our method is especially appropriate for describing the behavior of concurrent and distributed systems in which a wider range of failure modes can arise. Hence, we focus on examples that occur naturally in the context of concurrent and distributed systems.

In Section II, we introduce the basic specification method. We present examples illustrating how the method is used for replicated data in Section III, for atomic data in Section IV, and for secure systems in Section V. In Section VI, we close with some remarks and a discussion of related work.

II. Model

Informally, a system consists of a collection of sequential threads of control called *processes* that access data structures 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

Manuscript received September 15, 1989; revised August 24, 1990. This work was supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order 4864 (Amendment 20), monitored by the Air Force Avionics Laboratory under Contract F33615-87-C-1499. Additional support for J. Wing was provided in part by the National Science Foundation under Grant CCR-8620027. The views and conclusions contained in this paper 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 U.S. Government.

M. P. Herlihy is with Digital Equipment Corporation, Cambridge Research Laboratory, Cambridge, MA 02139.

J. M. Wing is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 9040812.

example, a file might provide read and write operations, and a FIFO queue might provide Enq and Deq operations.

We model a system computation as a *history,* which is a finite sequence of interleaved executions of operations on objects by processes. Each process can execute operations on more than one object and each object can be accessed by more than one process. To denote an operation of object x executed by process P, we write

$$x :: op(args^*)/term(res^*) :: P$$

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*(*e*)" for the invocation of operation e.

We assume that operations on objects are 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 [9], [13], or atomic broadcast protocols [4], [6]. Finally, for a distributed system, we assume a standard client–server configuration where a *server* encapsulates a set of objects and a *client* accesses a server's objects through remote procedure calls on the objects' operations.

A. Simple Object Automata

We model an object by a *simple object automaton,* an automaton that accepts certain histories. A simple object automaton is a four-tuple <STATE, s_0, OP, δ>, where STATE is the object's set of states, $s_0 \in$ STATE is its initial state, OP is a set of operations (the automaton's input alphabet), and δ : STATE $\times$ OP $\rightarrow 2^{\text{STATE}}$ is a partial *transition function.*[1]

The domain of the transition function can be extended to histories, δ^* : STATE $\times$ 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 $\delta^*(H) \neq \emptyset$. We call $\mathcal{L}(A)$, the language accepted by automaton A, the *behavior* of A.

B. Relaxation Lattices

Let $\mathcal{A}$ 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 $\mathcal{A}$ is a *lattice of automata* if the set $\{\mathcal{L}(A) \mid A \in \mathcal{A}\}$ 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 C, a lattice of automata $\mathcal{A}$, and a lattice homomorphism,

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

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

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 $C \subseteq \mathcal{C}$, then the object will behave like the simple object $\phi(C)$, accepting the language $\mathcal{L}(\phi(C))$. 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, e.g., site crashes or security violations, force the object to satisfy a weaker set, then it will accept additional "weakly consistent" histories, which are undesired but tolerated. Similarly, if changes to the environment, e.g., repairs or compensating actions, later occur, then the object will resume a more desired behavior.

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, 3) synchronization conflicts, such as choosing the oldest "unlocked" item from a queue, and 4) security breaches, such as illegally accessing a file.

C. The Environment

The current state of the environment determines which behavior, preferred or otherwise, an object exhibits. Formally, the environment is represented by an automaton <$2^{\mathcal{C}}$, c_0, EVENT, δ_E>. The environment's state is just the set of constraints it currently satisfies. Each input event in EVENT changes the environment state according to the transition function[2] $\delta_E : 2^{\mathcal{C}} \times$ EVENT $\rightarrow 2^{\mathcal{C}}$. Let $\mathcal{A}$ be a lattice of automata, where each A in $\mathcal{A}$ is given by the tuple <STATE, s_0, OP, δ_A>, and let a lattice homomorphism ϕ map each environment state to a simple object automaton.

The environment and the lattice can be combined into a single automaton:

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

The combined automaton accepts interleaved sequences of events and operations. Events change the environment state, and operations change the object state, but, as illustrated below, the sets EVENT and OP need not be disjoint. Let EVENTOP be EVENT $\cup$ OP. The transition function

$$\delta : 2^{\mathcal{C}} \times \text{STATE} \times \text{EVENTOP} \rightarrow 2^{\mathcal{C}} \times 2^{\text{STATE}}$$

is defined by two components

$$\delta_1 : 2^{\mathcal{C}} \times \text{EVENTOP} \rightarrow 2^{\mathcal{C}}$$

which defines the effects on the environment state, and

$$\delta_2 : 2^{\mathcal{C}} \times \text{STATE} \times \text{EVENTOP} \rightarrow 2^{\text{STATE}}$$

[1] We use 2^X to denote the powerset of the set X.

[2] Note that δ_E maps to a single state, not a set of states as for object automata. Modeling the environment as a nondeterministic automaton would add complexity to our definitions without providing any needed generality.

which defines the effects on the object's state:

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

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

When the combined automaton accepts an event, it changes the environment state. When it 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.

D. Specification Language

We use the Larch Specification Language [14], [15] to specify formally the states and transition functions for automata. In this paper, we give complete formal specifications for the simple object automata that comprise a lattice of automata. We give only informal descriptions to characterize environment automata, though it would be straightforward to give complete Larch specifications.

We represent an automaton's possible states as a set of *values,* as specified by a Larch *trait.* In a trait, the set of operators and their signatures, shown following the keyword **introduces**, defines a vocabulary of terms to denote values. For example, from the Bag trait of Fig. 1, *emp* and *ins*(*emp*, 5) denote two different bag (multiset) values. The set of equational axioms following the **asserts** 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*(*ins*(*emp*, 4), 3), 3), 3) = *ins*(*ins*(*emp*, 4), 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.

Larch provides two ways of reusing traits: a trait T can **include** or **assume** another trait T_1. If T_1 is included, then T extends the theory denoted by T_1 by adding more operators and equations explicitly in T. For example, FifoQ of Fig. 2 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*(*ins*(*emp*, 4), 3), 3)) = 4. If T_1 is assumed, then T may use T_1's operators with their meaning as given in T_1; a further use of T must discharge the assumption of T_1's theory. For example, a trait for priority queues (cf., Section III-C) might assume the existence of a total ordering on the items inserted in the queue. With either kind of reuse, a **with** clause allows renaming of operator and sort identifiers.

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 Fig. 2. The object's identifier, e.g., q, is an implicit argument and return formal (parameter) of each operation; the process's identifier, e.g., P, is an implicit argument, useful for applications for which the identity of an operation's caller is needed (see Section V). 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. 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.

```
Bag: trait
  introduces
    emp: → B
    ins: B, E → B
    del: B, E → B
    isEmp: B → Bool
    isIn: B, E → Bool
  asserts
    B generated by [ emp, ins ]
    for all [b: B, e, e1: E]
      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)

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

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

Fig. 1. Bag trait and interfaces.

```
FifoQ: trait
  includes Bag with [Q for B]
  introduces
    first: Q → E
    rest: Q → Q
  asserts 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 ins(rest(q), e)

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

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

Fig. 2. FIFO queue trait and interfaces.

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

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

For each automaton in a lattice $\mathcal{A}$, the sets of states (values) are the same, but their transition functions differ. Thus, their specifications will all use the same trait, but will have different interfaces. For example, 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.

III. First Example Domain: Replicated Objects

In a large distributed system, replication is often used to enhance the availability, reliability, and accessibility of data. For example, an airline's database might be replicated at different travel agencies to ensure that an airlines reservations system can service travel agents and their customers in a timely manner. Logically, there is only one database, but physically there are several replicas.

A *replicated* object is one that is stored redundantly at multiple sites in a distributed system. A *replication method* is a technique

for managing replicated objects. A widely-accepted correctness criterion for replication methods is *one-copy serializability* [3], 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 [5], [10], [24] 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. As networks grow in number of sites, chances of network partition and disconnected operations increase, and one-copy-serializability may be practically impossible to maintain.

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.

A. 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 [16].) A replicated object's state is represented as a *log,* which is a sequence of *entries,* where an entry is the time-stamped record of an operation. Time-stamps are generated by logical clocks [19]. For example, Fig. 3 shows a schematic representation of a queue replicated among three sites: S_1, S_2, and S_3.

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 time-stamp 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: $inv(e)\ Q\ e'$ if each initial quorum for the invocation of the operation e has a nonempty intersection with each final quorum for the operation e'.

S_1	S_2	S_3
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()

Fig. 3. A queue replicated among three sites.

B. 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* $\mathrm{QCA}(A, Q)$.

Definition 1: G is a Q-closed subhistory of H if whenever it contains an operation e it also contains every earlier operation e' of H such that $inv(e)\ Q\ e'$.

Definition 2: G is a Q-view of H for an operation e if 1) G includes every operation e' such that $inv(e)\ Q\ e'$, 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 e, $s \in \delta^*(G)$, and $s' \in \delta^*(G \cdot e)$ such that

$$\textbf{requires}\ e.pre_A(s)$$
$$\textbf{ensures}\ e.post_A(s, s') \wedge H' = H \cdot e.$$

Informally, G corresponds to the view constructed by merging the logs from an initial quorum for e. The view must satisfy the precondition for e, and the result of appending e 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: $\mathrm{QCA}(A, Q)$ is *one-copy serializable* if $\mathcal{L}(\mathrm{QCA}(A, Q)) = \mathcal{L}(A)$. 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 $\mathcal{L}(A)$ such that G is a Q-view of H for e, $G \cdot e \in \mathcal{L}(A) \Rightarrow H \cdot e \in \mathcal{L}(A)$.

Let Q be a *minimal* serial dependency relation, meaning that no proper subset of Q guarantees one-copy serializability. For all $R \subset Q$, $\mathcal{L}(\mathrm{QCA}(A, Q)) \subseteq \mathcal{L}(\mathrm{QCA}(A, R))$, since every history accepted by the former is accepted by the latter; thus the set $\{\mathrm{QCA}(A, R) \mid R \subseteq Q\}$ is a lattice of automata, and the lattice homomorphism $\phi(R) = \mathrm{QCA}(A, R)$ 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 $\mathcal{L}(A)$].

Additional flexibility can be achieved by adding a third parameter to a quorum consensus automaton: an *evaluation* function $\eta : \mathrm{STATE} \times \mathrm{OP}^* \rightarrow 2^{\mathrm{STATE}}$ that is required to agree with the transition function δ^* on histories in $\mathcal{L}(A)$. Informally, η is an extension of δ^* that allows us to assign an application-specific

meaning to histories not in $\mathcal{L}(A)$. Since a degraded behavior may contain such a history, η lets us give a meaningful value to an object after every state transition a "less-than-ideal" object takes.[3] The automaton $\mathrm{QCA}(A, Q, \eta)$ is defined identically to $\mathrm{QCA}(A, Q)$ except that η replaces δ^* in the above **requires** and **ensures** clauses. If Q is a serial dependency relation for $\mathcal{L}(A)$, then $\mathcal{L}(A) = \mathcal{L}(\mathrm{QCA}(A, Q)) = \mathcal{L}(\mathrm{QCA}(A, Q, \eta))$. The set $\{\mathrm{QCA}(A, R, \eta) \mid R \subseteq Q\}$ is also a lattice of automata, although, as illustrated below, different choices of η may produce different lattices.

C. 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. Fig. 4 describes the preferred behavior of a priority queue automaton.

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 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 [16].

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 weighted voting [12], 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 inquire 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 $\{\mathrm{QCA}(PQ, Q, \eta) \mid Q \subseteq \{Q_1, Q_2\}\}$ where η is the following evaluation function:[4]

$$\eta(\Lambda) = emp$$
$$\eta(H \cdot Enq(e)/Ok(\,)) = ins(\eta(H), e)$$
$$\eta(H \cdot Deq(\,)/Ok(e)) = del(\eta H, e).$$

[3] This property of η is especially useful when doing inductive arguments to prove the equivalence of automata (cf. Theorem 4).

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

```
PQueue: trait
  assumes TotalOrder with [E for T] % ¿ denotes the total order relation
  includes Bag with [PQ for B]
  introduces
    best: PQ → E
  asserts 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)

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

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

Fig. 4. Priority queue trait and interfaces.

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. A visual representation of the lattice of constraints appears in Fig. 5.

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 $\emptyset$.

Q_1: 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 $\mathrm{QCA}(\mathrm{PQ}, Q_1, \eta)$ is a one-copy serializable implementation of the *multipriority queue* automaton MPQ shown in Fig. 6. 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: $\mathcal{L}(\mathrm{QCA}(\mathrm{PQ}, Q_1, \eta)) = \mathcal{L}(\mathrm{MPQ})$.

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

Let δ be the transition function for MPQ. The postconditions of multipriority queue's interfaces completely determine the new value of the queue. Thus, for all H in $\mathcal{L}(\mathrm{MPQ})$, $\delta^*(H)$ 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 $\alpha : \mathrm{MPQ} \to \mathrm{PQ}$ to be the (value) homomorphism defined by projecting on the first component of the MPQ value: $\alpha(m) = m.present$.

If e is Enq or Deq, it is easy to check that

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

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

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

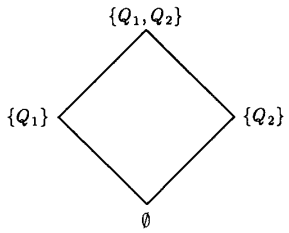

Fig. 5. Lattice of constraints.

```
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
  asserts 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)

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

q:: Deq()/Ok(e) ::P
  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))
```

Fig. 6. Multipriority queue.

Assume the result for all nonempty histories. Let $H' = H \cdot \mathrm{Enq}(e)/\mathrm{Ok}()$, $m = \delta^*(H)$, and $m' = \delta^*(H')$. By the Enq postcondition for MPQ, $m'.present = ins(m.present, x)$, hence $\alpha(\delta^*(H')) = ins(\alpha(\delta^*(H)), x)$. By the induction hypothesis, $\eta(H) = \alpha(\delta^*(H))$, hence $\eta(H') = \alpha(\delta^*(H'))$. If $H' = H \cdot \mathrm{Deq}()/\mathrm{Ok}(x)$, the same argument holds with *del* replacing *ins*. Thus, by substitution:

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

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

To show that $\mathcal{L}(\mathrm{MPQ}) \subseteq \mathcal{L}(\mathrm{QCA}(\mathrm{PQ}, Q_1, \eta))$, we also argue by induction. Let H be a history in $\mathcal{L}(\mathrm{MPQ})$ and $\mathcal{L}(\mathrm{QCA}(\mathrm{PQ}, Q_1, \eta))$ such that $H \cdot e$ is in $\mathcal{L}(\mathrm{MPQ})$. If e is $\mathrm{Enq}(x)/\mathrm{Ok}()$ for some x, $H \cdot e$ is clearly in $\mathcal{L}(\mathrm{QCA}(\mathrm{PQ}, Q_1, \eta))$. Suppose e is $\mathrm{Deq}()/\mathrm{Ok}(x)$. 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. □

Q_2: 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 $\mathrm{QCA}(\mathrm{PQ}, Q_2, \eta)$ is a one-copy serializable implementation of the *out-of-order priority queue* automaton OPQ given in Fig. 7. The behavior of an OPQ is just a bag (Fig. 1). Enq inserts an item in the bag and Deq removes an item, although not necessarily the best one. The argument that $\mathcal{L}(\mathrm{QCA}(\mathrm{PQ}, Q_2, \eta)) = \mathcal{L}(\mathrm{OPQ})$ is similar to that given for Theorem 4, and is omitted.

$\emptyset$: Finally, if we relax both constraints Q_1 and Q_2, the result is a *degenerate priority queue* (Fig. 8) 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 Fig. 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.

```
OPQ: trait
  includes Bag with [Q for B]

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

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

Fig. 7. Out-of-order priority queue.

```
DegenPQ: trait
  includes Bag with [Q for B]

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

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

Fig. 8. Degenerate priority queue.

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 $\mathrm{QCA}(\mathrm{PQ}, Q_2, \eta)$, $\mathrm{QCA}(\mathrm{PQ}, Q_2, \eta')$ never services requests out of order, but it could ignore certain requests.

D. 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 (Fig. 9) 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 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 [24] and Grapevine [5].

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

```
Account: trait
  introduces
    open: → A
    bal: A → Int
  asserts
    bal(open) = 0

a:: Credit(n: amount)/Ok() ::P
  ensures bal(a') = bal(a) + n

a:: Debit(n: amount)/Ok() ::P
  requires bal(a) - n ≥ 0
  ensures bal(a') = bal(a) - n

a:: Debit(n: amount)/Negative() ::P
  requires bal(a) - n < 0
  ensures a' = a
```

Fig. 9. Bank account trait and interfaces.

preserve this property, the account object may relax constraint A_1, but not A_2, thus the relaxation lattice is defined over a sublattice of $2^{\{A_1,A_2\}}$. These relaxed constraints imply that 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. Unlike the priority queue example, the object's set of operations and the environment's set of events are not disjoint.

IV. 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.* Here, failure events such as site crashes or network partitions can be masked as transaction aborts.

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 [11], [25]. 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. The cost paid for attaining the ideal ("correct") behavior is the degree of concurrency permitted: concurrency is sacrificed for more desired behavior. Our approach extends and formalizes that of Liskov and Weihl [23], [27], who have proposed that concurrency can be enhanced by introducing nondeterminism 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 nondeterminism to mask anomalous behavior.

A. Atomic Object Automata

To model atomic objects, we identify processes with transactions, and we augment each object's set of operations with special *commit* and *abort* operations. The state of the environment corresponds to the number and state of the active transactions; changes to the environment occur when a new transaction starts, or when one commits or aborts.

Formally, let A be a simple object automaton. A *schedule* for A is a history of operation executions where each operation is either an operation of A, *commit,* or *abort,* and each process name 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.

A *process subhistory,* $H \mid P$ (H at P), of a history H is the subsequence of operations in H whose process names are P. Two histories H and H' are *equivalent* if for every process P, $H \mid P = H' \mid P$. 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,

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

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

Definition 6: H is *atomic* if permanent(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 $\mathcal{L}(Atomic(A))$ 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 in all schedules in $\mathcal{L}(Atomic(A))$ transactions are serializable in the order they commit, a property known as *hybrid atomicity* [26]. This property is guaranteed by a number of atomicity mechanisms in common use, including strict two-phase locking [9]. Our examples can easily be adapted to other atomicity properties.

B. 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 dequeueing 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 dequeueing 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 dequeueing transactions execute serially, each of these alternative implementations yields a FIFO queue. If dequeueing 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 $\mathcal{C}$ 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 $2^{\mathcal{C}}$. As long as no more than k dequeueing 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 $\mathcal{L}(Atomic(Semiqueue_k))$ and the behavior of the "pessimistic" implementation is $\mathcal{L}(Atomic(Stuttering_j\text{-}Queue))$, where $Semiqueue_k$ and $Stuttering_j\text{-}Queue$ are defined in the next two sections.

The events that affect the environment 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.

1) Semiqueues: A $Semiqueue_k$ object (Fig. 10) 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 (Fig. 2) and if k is n, the maximum number of items allowed in the queue, the object is a bag (Fig. 1). Weihl and Liskov [27] give an example implementation of the semiqueue data type written in Argus [22].

The relaxation lattice is defined as follows. The set of constraints $\mathcal{C}$ is as defined above. The lattice homomorphism ϕ is defined over the sublattice of nonempty elements of $\mathcal{C}$: $\phi(B) = Semiqueue_k$, where c_k is the element of B with the lowest index. For example, the constraint and behavior lattices for a three-item queue is shown in Fig. 11.

Notice that ϕ is many-to-one, and not one-to-one as in the replication examples.[5] Moreover, if the queue is unbounded, then the lattice of behaviors is infinite.

[5] ϕ is also partial as in the bank account example.

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

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

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

Fig. 10. $Semiqueue_k$.

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)

Fig. 11. Relaxation lattice for a three-item semiqueue.

2) Stuttering Queues: A $Stuttering_j\text{-}Queue$ object (Fig. 12) 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 $\{Stuttering_j\text{-}Queue \mid j > 0\}$, and the lattice homomorphism ϕ is defined over the sublattice of nonempty elements B of $\mathcal{C}$: $\phi(B) = Stuttering_j\text{-}Queue$, 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.

V. Third Example Domain: Security

Relaxation lattices can be used to specify certain kinds of security properties and to characterize the expense incurred to ensure security. The cost of preserving security is the cost in maintaining passwords, implementing a secure encryption algorithm, or hiring personnel to guard a locked room. Here, we consider the two security properties, secrecy and integrity. To preserve *secrecy,* we must ensure that unauthorized users are prevented from executing operations that return information about the object's state, and to preserve *integrity,* we must ensure that unauthorized users are prevented from executing operations that modify the object's state. Although secrecy and integrity are often treated as monolithic properties, they too can be viewed as subject to graceful degradation.

A. Secure Object Automata

Let S and O be the sets of subjects and objects. Intuitively, S consists of all system users and programs, i.e., processes in our general model; O consists of all the resources to be protected, e.g., files, directories, and laserwriters. Let M be an *access-rights matrix* [21] where the (i, j)th entry in M is a set of rights that subject $i \in S$ has for object $j \in O$.

Unlike standard models of security (e.g., Bell and LaPadula's [2] or Lampson's [21]) in our model, a *right* of a subject i is not just the name of an access mode (e.g., read, modify, execute) or operation (e.g., Enq, Deq), but is a pair of predicates (i.e., pre- and postconditions) on the name of each operation of object j.

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

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

q:: Deq()/Ok(e) ::P
  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)]]]
```

Fig. 12. *Stuttering*$_j$*_Queue.*

For example, an entry for a subject P on a file f might contain the following pair of predicates for a write operation:

$$f :: \text{Write}(v : \text{value}) :: P$$
$$\textbf{requires } \text{id}(P) = \text{owner}(f)$$
$$\textbf{ensures } \text{val}(f') = v$$

where "id," "owner," and "val" would be defined in the appropriate traits. This element of the (P, f) entry in M restricts the process P invoking the write operation to be the owner of the file f.

Definition 8: A history H is *secure* if for each operation "$A :: e :: P$" in H there exists some (pre, post) pair of predicates for operation e in the (P, A)th entry of M such that $e.pre(s) \wedge e.post(s, s')$, where $s, s' \in \text{STATE}_A$ and s is the state of A upon invoking e and s' is the state upon return.

A *secure object automaton* $Secure(A)$ is an object automaton that accepts histories of the simple object automaton A such that each history in $\mathcal{L}(Secure(A))$ is secure.

Since an access-rights matrix M can be viewed as a set of permissions, we identify an environment's set of constraints (its "state") to be the complement of the set of permissions. Intuitively, constraints are prohibitions of the form "User X does not have the capability for operation Y on object Z," that are formally derivable from the sets of pairs of predicates of M.

B. Secure Mail Queue

As an example, consider a *mail queue* used as a temporary repository for mail intended for other sites. The mail queue provides four operations: a user can enqueue a message, dequeue the oldest message from the queue, cancel an unsent message, and list unsent messages. Clearly, not everyone should be allowed to execute every operation. For this example, we recognize four disjoint classes of users (see Fig. 13): 1) *operators* may execute any operation, 2) *faculty* members may enqueue messages and list or cancel transmission of their own messages, 3) *mailer* processes may dequeue messages for transmission, and 4) *students,* naturally, have no privileges at all. The specification of the operations is given formally in Fig. 14. In this specification, we assume that each user U invoking an operation on the queue q has an unforgeable name ($\text{id}(U)$), and that any attempt to execute an unauthorized operation signals an *Unauthorized* exception.

We can use relaxation lattices to formulate a variety of alternative "less secure" mail queue specifications. Under ideal circumstances, each user has the set of capabilities appropriate to his or her class. The cost of preserving these constraints is the cost of keeping passwords secret, using secure encryption protocols, etc. An event that affects the environment occurs when a user acquires capabilities to which he is not entitled, increasing the set of possible behaviors. Since a user who has discovered a new security breach is always free to refrain from exploiting it, each such breach introduces the possibility of new behaviors without excluding the possibility of older behaviors, thus the resulting set of behaviors clearly forms a relaxation lattice. The relaxation lattice characterizes the extent to which the resulting insecure behavior is close to the preferred secure behavior.

```
Class: trait
  C enumeration of [operator, faculty, mailer, student]

Users: trait
  includes Class, Set[USet, User]
    User record of [id: Id, class: C]
  introduces
    ops: USet → USet
    fac: USet → USet
    mlr: USet → USet
    stu: USet → USet
  asserts for all [u: U, us: US]
    u ∈ ops(us) = [class(u) = operator]
    u ∈ fac(us) = [class(u) = faculty]
    u ∈ mlr(us) = [class(u) = mailer]
    u ∈ stu(us) = [class(u) = student]
```

Fig. 13. Traits for users of the secure mail queue.

```
MailQ: trait
  includes Bag with [Q for B]

q:: Enq(m: message) / Ok() ::U
  requires class(U) = operator ∨ class(U) = faculty
  ensures q' = ins(q, m)
q:: Enq(m: message) / Unauthorized() ::U
  requires ¬ (class(U) = operator ∨ class(U) = faculty)
  ensures q' = q
q:: Deq() / Ok(m: message) ::U
  requires class(U) = mailer ∨ class(U) = operator
  ensures q' = del(q, m)
q:: Deq() / Unauthorized() ::U
  requires ¬ (class(U) = mailer ∨ class(U) = operator)
  ensures q' = q
q:: Cancel(m: message) / Ok() ::U
  requires [class(U) = operator ∧ isIn(q, m)] ∨
    [class(U) = faculty ∧ sender(m) = id(U) ∧ isIn(q,m)]
  ensures q' = del(q, m)
q:: Cancel(m: message) / Unauthorized() ::U
  requires [class(u) = student] ∨
    [class(u) = faculty ∧ sender(m) ≠ id(u)]
  ensures q' = q
q:: Cancel(m: message) / Absent() ::U
  requires [class(U) = operator ∧ ¬ isIn(q, m)] ∨
    [class(U) = faculty ∧ ¬ isIn(q, m)]
  ensures q' = del(q, m)
q:: List() / Ok(p: queue) ::U
  requires class(U) = operator ∨ class(U) = faculty
  ensures q = q' ∧
    class(U) = operator ⇒ ∀ m. (isIn(p', m) ⇔ isIn(q, m)) ∧
    class(U) = faculty ⇒ ∀ m.(isIn(p', m) ⇔ (isIn(q, m) ∧ sender(m) = id(U)))
q:: List() / Unauthorized() ::U
  requires ¬ (class(U) = operator ∨ class(U) = faculty)
  ensures q = q'
```

Fig. 14. Secure mail queue trait and interfaces.

Suppose Alice is a faculty member and Bob a student. Ideally, the following constraints hold (among many others):

S_1 Alice cannot dequeue messages.

S_2 Bob cannot list Alice's message.

If Alice discovers a way to relax constraint S_1, then the specification for Deq would change as shown in Fig. 15. Note that because Alice is always free to refrain from exploiting her knowledge, the precondition for the *Unauthorized* signal remains unchanged, and the corresponding automaton accepts a strictly larger language. Note also that the specification is independent of *how* Alice manages to circumvent the prohibition against dequeueing messages.

```
q:: Deq() / Ok(m: message) ::U
  requires class(U) = mailer ∨ class(U) = operator ∨ id(U) = Alice
  ensures q' = del(q, m)
q:: Deq() / Unauthorized() ::U
  requires ¬ (class(U) = mailer ∨ class(U) = operator)
  ensures q' = q
```

Fig. 15. Changes to interfaces if Alice can dequeue messages.

Similarly, Bob might relax constraint S_2 if he learns Alice's password. The resulting specification for List is shown in Fig. 16. Here too, since Bob can always refrain from using Alice's password, the precondition for the *Unauthorized* execution is unchanged, and the set of possible behaviors is strictly larger. Finally, an even larger set of behaviors is possible if both constraints are relaxed.

The application of relaxation lattices to security illustrates an alternative use of our method: as an after-the-fact investigative tool instead of a before-the-fact design tool. For security, a system should ideally never stray from the preferred behavior; system designers build in clever encryption and file protection schemes to ensure that "at all costs" secrecy and integrity of data are preserved. Thus, the designer would not explicitly handle cases of unpreferred behaviors because they should never occur. In practice, however, no matter how much foresight designers have put into systems, most do have "holes;" people find ways to compromise software, tamper with hardware, or deceive personnel. Some of these breaches are more severe than others in terms of the damage done and the cost of repair. For example, if Bob finds out Alice's password because Alice was careless, then that is not as bad as if he found out her password by having gained access to a password file, thereby gaining access to everyone's directories. At practically zero cost, Alice can simply change her password, whereas having everyone change their passwords would impose a nontrivial (and aggravating) cost. Thus, given some security breach, the designer can traverse a relaxation lattice to identify how much damage to the system was incurred and its cost of repair. Future systems can of course benefit from the identification of the unforeseen holes and the resulting degraded behaviors by designing ways to avoid them.

VI. Conclusions

The relaxation lattice method is quite flexible. It is applicable to a wide range of different domains for which very specialized formal techniques have been devised. In this paper, we have reviewed four applications: a replicated priority queue, a replicated bank account, an atomic FIFO queue, and a secure mail queue. In each case, as summarized in Fig. 17, 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 tradeoffs are captured naturally as a homomorphism between the lattice of constraints and the corresponding lattice of behaviors.

Our specification method suggests the following design strategy.

- Identify a set of constraints $\mathcal{C}$ that characterizes the preferred behavior. This set induces a lattice $2^{\mathcal{C}}$.
- Not all elements in the lattice may correspond to an intuitively meaningful behavior, let alone an acceptable one. The partial homomorphism ϕ determines which elements in the lattice of automata represent acceptable behaviors.

```
q:: List() / Ok(p: queue) ::U
  requires class(U) = operator ∨ class(U) = faculty ∨ id(U) = Bob
  ensures q = q' ∧
    class(U) = operator ⇒ ∀ m.(isIn(p', m) ⇔ isIn(q, m)) ∧
    class(U) = faculty ⇒ ∀ m.(isIn(p', m) ⇔ (isIn(q, m) ∧ sender(m) = id(U))) ∧
    id(U) = Bob ⇒ ∀ m.(isIn(p', m) ⇔ (isIn(q, m) ∧ sender(m) = Alice))
q:: List() / Unauthorized() ::U
  requires ¬ (class(U) = operator ∨ class(U) = faculty)
  ensures q = q'
```

Fig. 16. Changes to interfaces if Bob learns Alice's password.

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
Secrecy and integrity	Mail Queue	Access Rights	Protecting passwords	Obtaining passwords

Fig. 17. Summary chart.

- 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.

An alternative view of the relaxation lattice method is that a specification is a single, all encompassing, logical predicate on both the state of an object and the state of its environment. This predicate would be a conjunction of implications of the form "$E \Rightarrow A$" read as "if the predicate E holds of the environment, then A holds of the object." Each "$E \Rightarrow A$" represents the conditional behavior of an object. The entire conjunction represents the behavior of the object under different environmental conditions.

Instead of one big predicate, however, we separate the specification into two pieces (the object and environment automata) and moreover, impose a lattice structure on each piece. By using two pieces and the homomorphism ϕ between the two, we cleanly factor out that part of the environment that affects the behavior of an object. Making the environment explicit forces the specifier to state assumptions that are at best implicit in, but more often omitted from, a typical system specification [20], [29]. By additionally imposing a lattice structure, we gain a methodological advantage: we can systematically consider which of the constraints we are willing to sacrifice, and what object behavior is acceptable for each relaxed constraint. Making constraints explicit forces the designer to compare the costs of satisfying the constraints to the complexity of the unconstrained behavior. As illustrated by the replication examples, generating the lattice of weaker quorum intersection relations effectively enumerates the possible tradeoffs. Sometimes all tradeoffs are acceptable, as in the taxi queue example, and sometimes certain tradeoffs are unacceptable, as in the bank account example.

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. As in Larch, if a specification is considered to denote a theory, i.e., set of formulas, then specifications may be compared by comparing the strengths of their theories [28], where it may be necessary to introduce *theory interpretations,* i.e., mappings between theories. Maibaum and others [18] use this notion of theory interpretation to define a database view as an interpretation between two different database specifications. With our method, since the elements of the lattices

share the same vocabulary (i.e., based on the same trait), but just differ in the accepted strings (i.e., have different interfaces), we do not need to go through the complexity of defining theory interpretations. In our experience, showing that one specification is weaker than another, and thus, that one behavior includes another, is typically a simple exercise in logic (as in the proof of Theorem 4). Thus, we believe that sets of constraints are easier to work with than lattices of unstructured theories or specifications.

Our relaxation method captures precisely the intuition behind the informal specification method of Liskov and Weihl [27], [23]. 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.

The specifications given in this paper are functional in nature, requiring that an object display certain behavior when the environment satisfies certain constraints. By imposing an additional probabilistic structure on the environment, one could independently characterize the *likelihood* that certain constraints would be satisfied, and hence the likelihood the object would display certain behavior. In the replicated queue example, a probabilistic model of crashes and communication failures could be used to derive the likelihood that particular quorums will be available (e.g., as in Ahamad and Ammar's analysis [1]). We believe that a strength of the relaxation lattice method is that it preserves a clean interface between the functional and probabilistic domains, permitting the two kinds of specifications to be given independently and understood in isolation. 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 [8] or Cristian's [7] Markovian analysis of a two-disk stable storage implementation. We believe that separating the two domains makes our specifications easier to understand, more flexible, and more readily applicable to large and realistic problems.

The relaxation lattice method is a natural way to capture graceful degradation of large, complex systems. It can be used for two purposes: 1) as a descriptive technique for specifying the behavior of systems in existence and 2) as a prescriptive technique for specifying a range of acceptable behaviors of systems to be implemented. Our method can help identify those classes of faults that are either not explicitly or inappropriately handled. It can also help identify the costs associated with patching an existing system to handle previously undetected fault classes or making a new system more robust in the anticipation of failures.

Acknowledgment

We thank the anonymous referees for their suggestions for improving the presentation of our ideas.

References

[1] M. Ahamad and M. H. Ammar, "Performance characterization of quorum-consensus algorithms for replicated data," Tech. Rep. GIT-ICS-86/23, School of Inform. and Comput. Sci., Georgia Institute of Technol., Sept. 1986.

[2] D. E. Bell and L. J. LaPadula, "Secure computer systems: Unified exposition and multics interpretation," Tech. Rep. ESD-TR-75-306, The MITRE Corp., Bedford, MA, Mar. 1976.

[3] P. A. Bernstein and N. Goodman, "The failure and recovery problem for replicated databases," in *Proc. 2nd ACM SIGACT–SIGOPS Symp. Principles Distributed Comput.,* Montreal, P.Q., Canada, 1983.

[4] K. P. Birman, "Replication and fault-tolerance in the isis system," in *Proc. 10th Symp. Oper. Syst. Principles,* Dec. 1985. Also TR 85-668, Cornell Univ. Comput. Sci. Dep.

[5] A. D. Birrell, R. Levin, R. Needham, and M. Schroeder, "Grapevine: An exercise in distributed computing," *Commun. ACM,* vol. 25, no. 14, pp. 260–274, Apr. 1982.

[6] J. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. Comput. Syst.,* vol. 2, no. 3, pp. 251–273, Aug. 1984.

[7] F. Cristian, "A rigorous approach to fault-tolerant system development," Tech. Rep. RJ 4008, IBM Res. Lab., Sept. 1983.

[8] I. Durham and M. Shaw, "Specifying reliability as a software attribute," Tech. Rep. CS-82-148, Carnegie-Mellon Univ., Dec. 1982.

[9] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notion of consistency and predicate locks in a database system," *Commun. ACM,* vol. 19, no. 11, pp. 624–633, Nov. 1976.

[10] M. Fischer and A. Michael, "Sacrificing serializability to attain high availability of data in an unreliable network," in *Proc. ACM SIGACT–SIGMOD Symp. Principles Database Syst.,* Mar. 1982.

[11] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," *ACM Trans. Database Syst.,* vol. 8, no. 2, pp. 186–213, June 1983.

[12] D. K. Gifford, "Weighted voting for replicated data," in *Proc. Seventh Symp. Oper. Syst. Principles,* ACM SIGOPS, Dec. 1979.

[13] J. Gray, *Notes on Database Operating Systems.* Berlin, Germany: Springer-Verlag, 1978, pp. 393–481.

[14] J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch family of specification languages," *IEEE Software,* vol. 2, no. 5, pp. 24–36, Sept. 1985.

[15] J. V. Guttag, J. J. Horning, and J. M. Wing, "Larch in five easy pieces," Tech. Rep. 5, DEC Systems Research Center, July 1985.

[16] M. P. Herlihy, "A quorum-consensus replication method for abstract data types," *ACM Trans. Comput. Syst.,* vol. 4, no. 1, Feb. 1986.

[17] M. P. Herlihy and J. M. Wing, "Specifying graceful degradation in distributed systems," in *Proc. Sixth ACM SIGACT–SIGOPS Symp. Principles Distributed Comput. (PODC),* Aug. 1987. Also CMU-CS-87-120.

[18] S. Khosla, T. S. E. Maibaum, and M. Sadler, "Large database specifications from small views," in *Proc. Fifth Conf. Foundations Software Technol. Theoret. Comput. Science (LNCS 206).* Berlin, Germany: Springer-Verlag, 1985, pp. 246–271.

[19] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM,* vol. 21, no. 7, pp. 558–565, July 1978.

[20] ——, "A simple approach to specifying concurrent systems," *Commun. ACM,* vol. 32, no. 1, pp. 32–45, Jan. 1989.

[21] B. W. Lampson, "Protection," *ACM Oper. Syst. Rev.,* vol. 19, no. 5, pp. 13–24, Dec. 1985.

[22] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *Trans. Programming Languages Syst.,* vol. 5, no. 3, pp. 381–404, July 1983.

[23] B. H. Liskov and W. E. Weihl, "Specifications of distributed programs," *Distributed Comput.,* vol. 1, no. 2, pp. 102–118, Apr. 1986.

[24] 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 *Proc. Eighth Symp. Oper. Syst. Principles,* Dec. 1981.

[25] P. M. Schwarz and A. Z. Spector, "Synchronizing shared abstract types," *ACM Trans. Comput. Syst.,* vol. 2, no. 3, pp. 223–250, Aug. 1984.

[26] W. E. Weihl, "Specification and implementation of atomic data types," Tech. Rep. TR-314, MIT Lab. for Comput. Sci., Mar. 1984.

[27] W. E. Weihl and B. H. Liskov, "Specification and implementation of resilient, atomic data types," in *Proc. SIGPLAN Symp. Programming Language Issues in Software Syst.,* June 1983.

[28] J. M. Wing, "A two-tiered approach to specifying programs," Tech. Rep. MIT-LCS-TR-299, MIT Lab. for Comput. Sci., June 1983.

[29] ——, "A specifier's introduction to formal methods," *IEEE Comput. Mag.,* Sept. 1990.

Maurice P. Herlihy (S'80–M'84) received the A.M. degree in mathematics from Harvard University, and the M.S. and the Ph.D. degrees in computer science from M.I.T.

From 1984 to 1989 he was a faculty member in the Computer Science Department at Carnegie Mellon University, Pittsburgh, PA. In 1989 he joined the Research Staff at the Digital Equipment Corporation's Cambridge Research Laboratory, Cambridge, MA. His research interests include algorithms for replication and concurrency control, as well as formal and informal aspects of programming language support for reliable distributed computation.

Jeannette M. Wing (S'76–M'83) received the S.B., S.M., and Ph.D. degrees in computer science from the Massachusetts Institute of Technology.

She is an Associate Professor of Computer Science at Carnegie Mellon University. Her research interests include formal specifications, programming languages, concurrent and distributed systems, visual languages, and object management. At CMU she directs the Venari Project on semantics-based object search and retrieval. She also continues to work on the Avalon, Miro, and Larch Projects with colleagues at CMU, MIT, and DEC.

Dr. Wing is a member of the Association for Computing Machinery.