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

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

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 auto-
maton that accepts certain histories. A simple object automaton
is a four-tuple $<\text{STATE}, s_0, op, \delta>$, where $\text{STATE}$ 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 $\delta : \text{STATE} \times op \rightarrow 2^{\text{STATE}}$ is
a partial transition function.

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

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

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

where “*” denotes concatenation, and $\Lambda$ 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 $A$ 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 $A$ is a lattice
of automata if the set $\{\mathcal{L}(A) | A \in A\}$ 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 $A$, and a lattice homomorphism,

$$ \phi : 2^C \rightarrow A $$

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
$2^C$, i.e., the domain of $\phi$, so that the largest (intuitively, the
strongest) set of constraints lies at the top, and $\phi(C)$ is the
preferred behavior of $A$. In general, $\phi$ is defined over a sublattice
of $2^C$.

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 $C \subseteq C$,
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 $\langle 2^C, c_0, \text{EVENT}, b_0 \rangle$.
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 $b : 2^C \times \text{EVENT} \rightarrow 2^C$.

Let $A$ be a lattice of automata, where each $A$ in $A$ is given by
the tuple $<\text{STATE}, s_0, op, \delta_A>$, and let a lattice homomorphism $\phi$
map each environment state to a simple object automaton.

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

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

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 EVENTTOP be EVENT \cup OP.

The transition function

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

is defined by two components

$$ b_1 : 2^C \times \text{EVENTTOP} \rightarrow 2^C $$

$$ b_2 : 2^C \times \text{STATE} \times \text{EVENTTOP} \rightarrow 2^{\text{STATE}} $$

Note that $\delta_C$ 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:

\[ \delta_t(c, e) = \text{if } e \in E \text{ EVENT then } \delta_e(c, e) \text{ else } c \]

\[ \delta_t(c, s, e) = \text{if } e \in \text{OP} \land A = \phi(\delta_t(c, e)) \text{ then } \delta_A(s, e) \text{ else } \{s\} \]

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 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, s) 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 \( \text{del} \text{ins} \text{ins} \text{emp}(4,3,3) = \text{ins} \text{ins} \text{emp}(4,3) \).

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 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 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(emp(4,3),3),3)) = 4. If T1 is assumed, then T may use T1’s operators with their meaning given in T1; a further use of T must discharge the assumption of T1’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., q, 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 Eqn’s postcondition is given by the FifoQ trait.

For an operation e of a simple object automaton A we write e.preA and e.postA for the pre- and postconditions of e. The transition function \( \delta \) for A is defined such that

\[ (\forall s, s’ \in \text{STATE}) s’ \in \delta(s, e) \iff e.preA(s) \land e.postA(s, s’) \]

For each automaton in a lattice 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 nat-
ural 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 in-
crease, and one-copy-serializability may be practically impossible
to maintain.

In this section, we outline how specifications based on relax-
ation lattices can express the behavior of a number of "weakly
consistent" replication methods from the literature without sacri-
ficing one-copy serializability as the basic correctness condition.
Each of the weakly consistent methods is based on the ob-
server 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 repli-
cated 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 curent
value is $inv\{ins\{ins\{emp, x, y\}, z\}\}$, which can be recon-
structed 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 real-
zizable 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: $inv(e) \cap Q e'$ if each initial quorum for the invocation
of the operation $e$ has a nonempty intersection with each final
quorum for the operation $e'$.

B. Quorum Consensus Automata

Given a simple object automaton $A$ and a quorum intersec-
tion 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 $e$ it also contains every earlier operation
$e'$ of $H$ such that $inv(e) \cap Q e'$.

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

\[
\begin{align*}
\text{requires } e.pre_{A}(s) \\
\text{ensures } e.post_{A}(s, s') \land H' = H \cdot e.
\end{align*}
\]

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: $QCA(A, Q)$ is one-copy serializable if
$L(QCA(A, Q)) \subseteq L(A)$. Quorum consensus replication guaran-
tees one-copy serializability if and only if the quorum intersec-
tion 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 $e$, $G \cdot e \in L(A) \Rightarrow H \cdot e \in L(A)$.

Let $Q$ be a minimal serial dependency relation, meaning that
no proper subset of $Q$ guarantees one-copy serializability. For all
$R \subseteq Q$, $L(QCA(A, Q)) \subseteq L(QCA(A, R))$, since every history
accepted by the former is accepted by the latter; thus the set
$\{QCA(A, R) | R \subseteq Q\}$ 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 param-
eter to a quorum consensus automaton: an evaluation function
$\eta : \text{STATS} \times \text{OP} \rightarrow \mathbb{2}^{\text{STATS}}$ that is required to agree with the
transition function $\delta^*$ on histories in $L(A)$. Informally, $\eta$ is
an extension of $\delta^*$ that allows us to assign an application-specific
meaning to histories not in \( L(A) \). Since a degraded behavior may contain such a history, \( \eta \) lets us give a meaningful value to an object after every state transition a "less-than-ideal" object takes. The automaton \( QCA(A, Q, \eta) \) is defined identically to \( QCA(A, Q) \) except that \( \eta \) replaces \( \delta^* \) 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 \( \{QCA(A, R, \eta) : R \subseteq Q\} \) is also a lattice of automata, although, as illustrated below, different choices of \( \eta \) 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 queue 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 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 \( \{QCA(PQ, Q, \eta) : Q \subseteq \{Q_1, Q_2\}\} \) where \( \eta \) is the following evaluation function:

\[
\begin{align*}
\eta(L) &= \text{emp} \\
\eta(H \cdot \text{Enq}(e) / \text{Ok}(e)) &= \text{ins}(\eta(H), e) \\
\eta(H \cdot \text{Deq}(e) / \text{Ok}(e)) &= \text{del}(\eta(H), e).
\end{align*}
\]

Although \( \eta \) 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 \( \eta \) 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 unserved higher priority request will ever be passed over in favor of an unserved lower priority request. More precisely, we claim the automaton \( QCA(PQ, Q_1, \eta) \) 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:** \( L(QCA(PQ, Q_1, \eta)) = L(MPQ) \).

**Proof:** We first show that \( L(QCA(PQ, Q_1, \eta)) \subseteq L(MPQ) \). If \( Q \) 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 \( \delta \) 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 \( L(MPQ) \), \( \delta^*(H) \) is a singleton set, and we simplify our notation by treating \( \delta^* \) as a function from histories to MPQ values, rather than sets of MPQ values. Define \( \alpha : MPQ \rightarrow PQ \) to be the (value) homomorphism defined by projecting on the first component of the MPQ value: \( \alpha(m) = m.\text{present} \).

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

\[
\begin{align*}
\text{e.pr} &\Rightarrow \alpha(\delta^*(H)) \Rightarrow \text{e.pr} &\Rightarrow \text{e.post} &\Rightarrow \alpha(\delta^*(H)) \Rightarrow \text{e.post}.
\end{align*}
\]

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

\[
\alpha(\delta^*(\Lambda)) = \eta(\Lambda) = \text{emp}.
\]
MPQueue: trait
assumes TotalOrder with [E for T] % i denote 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, 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(x)(Ok()) : P
ensures
[x ∈ best(q, present) ∧ x > best(q, present)] ∨
[x = best(q, present) ∧ x ∈ 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 · Enq(e)(Ok( )) , m = δ*(H), and m’ = δ*(H’). By the Enq postcondition for MPQ, m’ present = ins(m, present, x), hence
α(δ*(H’)) = ins(α(δ*(H)), x). By the induction hypothesis, α(δ*(H)) = α(δ*(H’)), hence α(δ*(H’)) = α(δ*(H’)). If H = H · Deq( )/Ok(x), the same argument holds with del replacing ins. Thus, by substitution:

where is enough to show that L(QCA(PQ, Q1, η)) ⊆ L(MPQ).
Note that the preconditions for both Enq’s are true, and Deq’s are true, thus making the first implication for e = Deq trivially true.

To show that L(MPQ) ⊆ L(QCA(PQ, Q1, η)), we argue by induction. Let H be a history in L(MPQ) and
L(QCA(PQ, Q1, η)) such that H · e is in L(MPQ).
If e is Enq(x)(Ok( )) for some x, H · e is clearly in
L(QCA(PQ, Q1, η)). Suppose e 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.
□

Q2: 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, Q2, η) is a one-copy serializable implementation of the out-of-order priority queue automaton OPO given in Fig. 7. The behavior of an OPO is just a bug (Fig. 1). Enq inserts an item in the bag and Deq removes an item, although not necessarily the best one. The argument that L(QCA(PQ, Q2, η)) = L(OPO) is similar to that given for Theorem 4, and is omitted.

□: Finally, if we relax both constraints Q1 and Q2, the result is a degenerate priority queue (Fig. 8) which permits clients to be

D. Example 2: A Replicated Bank Account

Constraints on quorums 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:

A1 Every initial Debit quorum intersects every final Credit quorum.

A2 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 A1 and A2. 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
ACount: trait

introduces
open: '→ A
bal: A → int
asserts
bal(open) = 0

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

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

a: Debit(t: amount)/Negative(): P
requires bal(a) < n
ensures a = a

Fig. 9. Bank account trait and interfaces.

preserve this property, the account object may relax constraint
A1, but not A3, thus the relaxation lattice is defined over
a sublattice of 2(A1, A3). 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 A1 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 trans-
actions, 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 | 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 | P = H’ | 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 | P1 · · · Hn | Pk is in L(A), where P1, · · · , Pk are the transactions in H in the order <.

Let permanent(H) be the subsequence 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 transac-
tions 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 in all schedules in 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 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


```

```

generally degrading" queue specifications. In each case, the extent to which the queue deparcs 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
deequeuing transactions overlap, however, the first implementa-
tion 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 $C$ is $\{C_k | k > 0\}$. For
each of the implementations sketched above, the lattice homo-

morphism $\phi$ assigns a behavior to each element in the lattice of
constraints $C_k$. 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 $\text{Atomic}(\phi(C_k))$. While
$C_k$ is satisfied the behavior of the "optimistic" implementa-
tion is $L(\text{Atomic}(\text{Semiquene})_{\phi})$ and the behavior of the "pes-

simistic" implementation is $L(\text{Atomic}(\text{Stuttering},_\phi))$, where
$\text{Semiquene}$ and $\text{Stuttering},_\phi$ 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.

Semiquene: A Semiquene 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). Weill and
Liskov [27] give an example implementation of the semiquene
data type written in Argus [22].

The relaxation lattice is defined as follows. The set of con-
straints $C$ is as defined above. The lattice homomorphism $\phi$
is defined over the sublatice of nonempty elements of $C : \phi(B) =
\text{Semiquene}_{\phi}$, 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 $\phi$ is many-to-one, and not one-to-one as in the
replication examples. Moreover, if the queue is unbounded, then
the lattice of behaviors is infinite.

$^5 \phi$ is also partial as in the bank account example.

2) Stuttering Queues: A Stuttering$_\phi$ 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 semiquenes: The lattice of automata is
$\{\text{Stuttering},_\phi, \text{Queue} | j > 0\}$, and the lattice homomor-
phism $\phi$ is defined over the sublatice of nonempty elements $B$ of
$C : \phi(B) = \text{Stuttering},_\phi, \text{Queue}$, where $C_j$ is the element of $B$ with
the lowest index.

The stuttering queue and semiquene behaviors can be com-
pared within a single lattice: the $\text{SSqueue}_{\phi,j}$ behavior would
permit any of the first $k$ items to be returned as many as $j$ times.
$\text{SSqueue}_{\phi,j}$ 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 $j$. 

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

$$f :: \text{Write}(v : \text{value}) :: P$$

requires $id(P) = \text{owner}(f)$

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 $E$ 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 “$A :: c : E^m$” in $H$ there exists some (pre, post) pair of predicates for operation $e$ in the $(P,A)$ entry of $E$ such that $e.\text{pre}(s,A)$ satisfies some (pre, post) pair of predicates for operation $e'$ in the $(P,A)$ entry of $E$ such that $e.\text{post}(s,A)$.

A secure object automaton $\text{Secure}(A)$ is an object automaton that accepts histories of the simple object automaton $A$ such that each history in $L(\text{Secure}(A))$ 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 tranmission of their own 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 $(id(U))$, 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.

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

\begin{align*}
S_1: & \text{Alice cannot dequeue messages.} \\
S_2: & \text{Bob cannot list Alice’s message.}
\end{align*}

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 dequeuing 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 alter-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 unexpected 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 behavior 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 $C$ 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 partial homomorphism $\phi$ determines which elements in the lattice of automata represent acceptable behaviors.

\[ \text{Fig. 15. Changes to interfaces if Alice can dequeue messages.} \]

\[ \text{Fig. 16. Changes to interfaces if Bob learns Alice’s password.} \]

\[ \text{Fig. 17. Summary chart.} \]

- 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 $\phi$ 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 quorums interaction 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. 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 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 semaphore, 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 Christian’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. In our experience, showing that one specification 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

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.