The feasibility of using standard Z notation in the design of complex systems

Thesis

How to cite:


© 1993 The Author

Version: Version of Record

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online’s data policy on reuse of materials please consult the policies page.
The Feasibility of using Standard Z Notation in the Design of Complex Systems

Volume 2 of 2

by

David John Reed
BSc BA MSc MIEEE CEng

4 January 1993

Thesis submitted for the Degree of Doctor of Philosophy
Chapter 6
Graceful Degradation of Queues

This chapter contains a study of applying the Z notation in the specification of the graceful degradation of replicated queues. The replicated queues adopt a concurrency control method based on a quorum consensus algorithm [Herl86].

The Z notation is applied in this chapter to specify a one copy queue whose behaviour is shown to be equivalent to the behaviour of an implementation of a replicated queue similarly described in the Z notation. That is, the replicated queue is shown to be a correct implementation of a one queue specification. The activities that occur at the design stage presented in this chapter take the form of constructing two models of a system:

1. one model represents the specification of the system as a one copy queue that has the required properties, and
2. the other model represents an implementation of the system as a network of replicated priority queues that follow a quorum consensus algorithm for concurrency control.

The implementation is then proved using two different approaches, with varying degrees of success, to meet the specification. The first approach applies data refinement conditions for the Z notation [Spiv89A, Dillr90, Pottr91]. The second approach uses informal proof sketches to prove that the behaviours implied by the specification and implementation are of the same type.
6.1 Introduction

The replicated queues are examples of simple replicated database systems where the only operations are enqueue and dequeue. The operations form complete transactions, hence there is no need to have either commit or abort operations.

The one copy serialization property is too strict for some applications and a more relaxed requirement is useful. The work by Herlihy and Wing [Her91] describes how the one copy serialization property can be relaxed in a controlled manner, such that the requirements form a lattice of acceptable behaviours. Each node in the lattice represents the behaviour capable of a replicated queue with a different quorum intersection constraint. The lattice describes how the behaviour of a priority queue can degrade gracefully to that of a degenerate priority queue. The importance of a lattice structure is that a lattice reflects the relationships between its members, showing how members of the lattice can be compared meaningfully. In the case where the members in a lattice are specifications, it is important to know that changes in constraints do not imply a completely different specification.

The basic characteristics of a priority queue are that it always returns the element with the highest priority whenever requested and it only returns an element once. In some circumstances such strict requirements are not necessary and it is acceptable to reduce the constraints on the behaviour of a queue. If the behaviour of a priority queue is a subset of the behaviour of the less strict queue, then the behaviour of the former is said to gracefully degrade to the latter. This is interpreted to mean that the behaviour of a priority queue is still possible by the less strict queue, but the less strict queue is also capable of other types of behaviour.

The concept of graceful degradation can be seen from a different perspective by considering a replicated queue implemented by several sites. When a sufficient number of sites are in agreement with each other, the replicated queues exhibit the behaviour of a single priority queue. Under fault conditions, when one or more sites or communication links fail, the behaviour of a replicated queue is shown to degrade gracefully to a set of less strict requirements.
This chapter is based on the real time priority queue example given by Herlihy and Wing [Herl91]. The behaviour of a priority queue can degrade to a multiple priority queue, an out of order priority queue, or a degenerate priority queue. The possible behaviours of these four queues form a relaxation lattice. The two main differences between the method presented here and that advocated by Herlihy and Wing are that the specifications are written in the Z notation instead of Larch, and that the verification is achieved directly by informal proof sketches instead of proof sketches based on the concept of quorum consensus automata.

In the four implementations of replicated queues, the descriptions of the queue operations are not changed. The only changes are to the invariants that describe the relationships between the sites that are accessed for the queue operations. The four specifications of the replicated queues correspond to the four different one copy queues that define the required behaviour of the replicated queues.

The network of sites is represented as a function from sites to queues. The behaviour of a replicated queue is modelled as though the queue operations are received by a transaction manager that distributes queue operations to individual queues located at the sites. In an actual replicated queue, it is the individual sites that receive the operations and the control of the network is distributed across the network, with each site acting independently of the other sites, but in a cooperative manner.

The complete descriptions of the queues are given in the Z notation such that the data that determines the behaviour of the system are made explicit. The behaviour of each queue is described in terms of operations that change the state of the queue and the schemas that represent these operations are connected together to form an overall description of the queue.

As with the two other studies, all the schemas for this study are included in this chapter and thereby represent worthwhile unabridged examples of implementations of complex systems described in the Z notation.

Table 6.1 indicates the principal schemas in the specification and implementation of each of the four types of queues described in Section 6.2 to 6.6.
### Table 6.1 Summary of Schemas in Chapter 6

<table>
<thead>
<tr>
<th>Queue Type</th>
<th>Specification</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Priority</strong></td>
<td>Queue_Spec = Enqueue \lor Dequeue</td>
<td>Queue_Imp = Intersection \land (Site_Enq \lor Site_Deq)</td>
</tr>
<tr>
<td>Returns an element</td>
<td>with the highest priority from those stored in the queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page 231</td>
<td>Page 244</td>
</tr>
<tr>
<td></td>
<td>Demonstrate Queue_Spec and Queue_Imp are equivalent in Sections 6.2.3 and 6.2.4</td>
<td></td>
</tr>
<tr>
<td><strong>Multiple Priority</strong></td>
<td>MP_Queue_Spec = MP_Enqueue \lor MP_Dequeue</td>
<td>MP_Queue_Imp = MP_Intersection \land (Site_Enq \lor Site_Deq)</td>
</tr>
<tr>
<td>Returns either an</td>
<td>element that has the highest priority or one that has been returned before and</td>
<td></td>
</tr>
<tr>
<td>element</td>
<td>that has a priority greater than any element in the queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page 266</td>
<td>Page 269</td>
</tr>
<tr>
<td></td>
<td>Demonstrate MP_Queue_Spec and MP_Queue_Imp are equivalent in Sections 6.3.3 and 6.3.4</td>
<td></td>
</tr>
<tr>
<td><strong>Out of Order</strong></td>
<td>OO_Queue_Spec = OO_Enqueue \lor OO_Dequeue</td>
<td>OO_Queue_Imp = OO_Intersection \land (Site_Enq \lor Site_Deq)</td>
</tr>
<tr>
<td>Returns an element</td>
<td>in any order from those elements stored in the queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page 282</td>
<td>Page 285</td>
</tr>
<tr>
<td></td>
<td>Demonstrate OO_Queue_Spec and OO_Queue_Imp are equivalent in Sections 6.4.3 and 6.4.4</td>
<td></td>
</tr>
<tr>
<td><strong>Degenerate Priority</strong></td>
<td>DP_Queue_Spec = DP_Enqueue \lor DP_Dequeue</td>
<td>DP_Queue_Imp = Site_Enq \lor Site_Deq</td>
</tr>
<tr>
<td>Returns an element</td>
<td>in any order and any number of times from those elements that have been</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stored in the queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Page 298</td>
<td>Page 300</td>
</tr>
<tr>
<td></td>
<td>Demonstrate DP_Queue_Spec and DP_Queue_Imp are equivalent in Sections 6.5.3 and 6.5.4</td>
<td></td>
</tr>
</tbody>
</table>

The lattice structure of the behaviours of the four types of queues is discussed in Section 6.6 and Section 6.7 contains a summary of the main results of this study.
6.2 Priority Queue

The behaviour of a priority queue is such that the elements are removed from the queue once and in strict priority order. The study of a priority is split into the following sections:

6.2.1 Specification of a One Copy Priority Queue

6.2.2 Implementation of a Replicated Priority Queue

6.2.3 Verification using Refinement Conditions of a Priority Queue

6.2.4 Verification using a Proof Sketch of Equivalent Behaviour of a Priority Queue

6.2.1 Specification of a One Copy Priority Queue

The specification in this section represents the storage of data in queues. Figure 6.1 illustrates the interactions between the operation schemas and state schemas in the specification of a one copy priority queue.

*Figure 6.1 Interactions in the Specification of a Priority Queue*

Include the diagram here.

Import toolkit

[ELEMENT, LETTER, TIME]

The three given sets for the schemas in this chapter are:

ELEMENT

LETTER
TIME

The set ELEMENT represents the type of data that is stored in the queue. The given set ELEMENT is assumed to contain other information such as the priority. The hidden components of the type ELEMENT are accessed by projection functions.

The given set LETTER is used in the implementation schemas to identify the sites that contain a copy of the queue.

The given set TIME is included as the means of determining the order in which operations are performed.

Proof Obligation for the Given Sets

The members of the given sets can be replaced by standard sets or tuples of standard sets to illustrate the consistency of the models of the specifications used in this chapter.

Bags are used to represent the method of storing data in queues. The next schema specifies two operators for bags.

\[
\begin{align*}
\text{Add} & \colon (\text{bag } X \times X) \rightarrow \text{bag } X \\
\text{Delete} & \colon (\text{bag } X \times X) \rightarrow \text{bag } X
\end{align*}
\]

\[
\forall a : \text{bag } X; b : X \cdot \\
\text{Add} (a, b) = a \oplus \{b \mapsto \text{count } a \cdot b + 1\} \land \\
\text{Delete} (a, b) = a \oplus \{b \mapsto \text{max } \{0, \text{count } a \cdot b - 1\}\}
\]

The bag operator Add updates the count of an element in a bag for the addition of a new member and the bag operator Delete updates the count of an element in a bag for the deletion of an element. The set operator max ensures that the count of an element is
always positive.

The generic schema below defines three projection functions to retrieve each of the elements of tuples with three components.

\[ [X, Y, Z] \]

First : \((X \times Y \times Z) \rightarrow X\)
Second : \((X \times Y \times Z) \rightarrow Y\)
Third : \((X \times Y \times Z) \rightarrow Z\)

\(\forall x : X; y : Y; z : Z \cdot\)
First \((x, y, z) = x\) \land Second \((x, y, z) = y\) \land Third \((x, y, z) = z\)

Operator ::= \(\text{enq} \mid \text{deq}\)

The above free data declaration specifies the two operators, \(\text{enq}\) and \(\text{deq}\), for the queues in this chapter.

Status ::= \(\text{okay} \mid \text{error}\)

The operations that are performed are given the status of \text{okay}, the ones that are not are given the status of \text{error}.

---

\textit{Proof Obligation for the Free Type Definitions}

Both free type definitions for Operator and Status do not contain recursive branches, hence their definitions are consistent.
Operation == (Operator × ELEMENT × TIME)

The operations are represented by the global syntactic type Operation.

priority : ELEMENT → \mathbb{N}

The function priority maps each member of the given set ELEMENT to a natural number that represents its priority. The details of how this is achieved are not specified here.

Response == (Status × ELEMENT)

The queues respond to each operation with an output of the type given by Response.

null : ELEMENT

The identifier null is a member of the given set ELEMENT and represents the cases where no useful data are assigned to a component of an operation or a response.

History == \mathcal{P} (Operation × Response)

The record of operations performed and responses of the queues are represented by the data type History. The given set TIME is included in the type Operation to allow the order of operations to be identified. Other possible means for deriving the order include using a partial function from TIME to sets of operations (without a time component) and responses.
In the specification of priority queues it is necessary to be able to construct a queue based on a history of queue operations. The function below provides a means of constructing such queues.

\[
\text{Create\_Bag : History} \rightarrow \text{bag ELEMENT}
\]

The function \text{Create\_Bag} transforms a history of operations into a bag of elements, which represents a queue. The function is partial over all possible histories because some histories cannot be produced by using the operations defined here. An implementation of this function could be one that returns the final state of a queue by simulating the operation of a queue based on the set of operation and response pairs.

The schema \text{Queue} defines the data type that represents priority queues in the specification.

\[
\text{Queue} \\
\begin{align*}
q & : \text{bag ELEMENT} \\
h & : \text{History} \\
q &= \text{Create\_Bag } h
\end{align*}
\]

The bag \(q\) in the schema \text{Queue} represents the data held in the queue and the set \(h\) records all the successful operations performed on that queue. A sequence is not needed to record the order of occurrence of operations since the type \text{Operation} contains a time component. The history component is included to facilitate the comparison of the behaviours implied by the specification and implementation of the queues by identifying the state of the queue.

The component \(q\) is not strictly necessary as it can always be derived from the \(h\) component using the function \text{Create\_Bag}. The invariant of the schema states that the contents
of the component $q$ must always be consistent with the recorded history component.

The schema $\text{Enqueue}$ below specifies the enqueue operation performed on a queue.

$$
\begin{align*}
\text{Enqueue} \\
\Delta \text{Queue} \\
\text{Op?} : \text{Operation} \\
\text{Res!} : \text{Response} \\
\text{First Op?} = \text{enq} \\
q' = \text{Add_Bag} (q, \text{Second Op?}) \\
\text{Res!} = (\text{okay, null}) \\
h' = h \cup \{(\text{Op?}, \text{Res!})\}
\end{align*}
$$

The input to the schema $\text{Enqueue}$ is $\text{Op?}$ which is a tuple that contains the identifier for the enqueue operator as the first component and the element value as the second component.

The output $\text{Res!}$ is a tuple with the okay status as the first component and the element null value as the second component.

---

**Proof Obligation for the Schema Enqueue**

Adherence to the type rules of the Z notation is enforced by CADi2.

The components $q$ and $h$ are updated by the schema $\text{Enqueue}$.

The invariant of the schema $\text{Queue}$ is that

$$q = \text{Create_Bag} \ h$$

hence, there is a proof obligation to ensure that $q'$ and $h'$ are related by the partial function $\text{Create_Bag}$, i.e.

$$q = \text{Create_Bag} \ h \Rightarrow q' = \text{Create_Bag} \ h'$$
However, the function `Create_Bag` is not given a formal specification, but is given an informal definition based on a simulation of the specification. Therefore, a requirement for any implementation of `Create_Bag` is that all valid histories of operations must result in the queue value given by the specification. Namely, if $q'$ results from a particular enqueue operation, then `Create_Bag` must be able to re-construct $q'$ from the historical records. Therefore, by definition, satisfies the invariant.

The schema `Dequeue` below specifies the behaviour of the queue in response to a dequeue operation.

```
Dequeue

\Delta Queue
Op?: Operation
Res!: Response

First Op? = deq
$q \neq [] \land
(\exists eout : \text{ELEMENT} \cdot
  eout in q \land
  (\forall e : \text{ELEMENT} \mid e in q \cdot \text{priority eout} \geq \text{priority e}) \land
  q' = \text{Delete_Bag} (q, eout) \land
  \text{Res!} = (\text{okay, eout}) \land
  h' = h \cup \{(\text{Op?, Res!})\})
\lor
$q = [] \land \theta_{\text{Queue'}} = \theta_{\text{Queue}} \land
\text{Res!} = (\text{error, null})
```

The type of operation is identified by the first component of the input tuple `Op?` in the schema `Dequeue`. The second component of the input is not used. The output tuple has a first component of the status value `okay` and a second component of the element with the highest priority value of the members of the queue.
A dequeue operation performed on an empty queue results in no change of state.

---

**Proof Obligation for the Schema Dequeue**

Adherence to the type rules of the notation is enforced by CAi2.

The invariant of the schema Queue is based on the function Create_Bag being able to re-construct $q'$ from the updated history component $h'$. This is a basic requirement of Create_Bag, hence the invariant must be satisfied.

---

The schema Queue_Spec specifies the queue as the disjunction of the schemas Enqueue and Dequeue.

$$\text{Queue}_\text{Spec} \equiv \text{Enqueue} \lor \text{Dequeue}$$

The invariants for the schema Queue are obviously satisfied by this operation because the component schemas maintain the invariants.

**Preconditions of the schema Queue_Spec**

The simplified preconditions of the schema Queue_Spec are defined in the schema Pre_Queue_Spec_Simple below.

```
  Pre_Queue_Spec_Simple
  Queue
  Op? : Operation

  First Op? = enq
  \lor
  First Op? = deq
```

The simplified preconditions are checked by verifying the following schema.
Simplified_6_1 \equiv \text{pre Queue\_Spec} \iff \text{Pre\_Queue\_Spec\_Simple}

The initial state is represented by the schema below.

\[
\begin{align*}
\text{Initial\_Queue\_Spec} & \\
\text{Queue}' & \\
q' & = [] \\
h' & = \emptyset
\end{align*}
\]

---

**Proof Obligation for the Initial State**

The schema Initial\_Queue\_Spec is obviously a valid element of the state described by the schema Queue.

The schema Queue\_Spec is total over all possible values of the component \( q \), but not all values of the component \( h \) because some histories cannot be achieved by using the operations defined by the schemas. For instance, the history may contain operation and response pairs of dequeue operations that remove elements that have not been enqueued. It is possible to constrain the value of the history component to those events that are possible by a single priority queue, but this may clash with that specified indirectly by the schema Queue\_Spec.

### 6.2.2 Implementation of a Replicated Priority Queue

The implementation of a priority queue is given in terms of three schemas. One schema specifies the invariants that apply to the initial and final quorums, and the other two schemas specify the changes of state in response to the enqueue and dequeue operations. Figure 6.2 illustrates the interactions between the schemas used in the implementation of a priority...
Some preliminary data declarations have to be made before the schema definitions.

\[ \text{Site} \equiv \text{LETTER} \]

The queues are considered to be replicated at sites and each site is identified by a member of the given set \text{LETTER}. The set \text{Site} is defined above to be syntactically equivalent to the set of \text{LETTER}.

\[ \text{Quorum} \equiv P \text{ Site} \]

A quorum is a set of sites and represents either the sites accessed before an operation is
performed or the sites communicated with after an operation has been performed. The quorums of the sites accessed before operations are performed are called *initial quorums*, and those containing the sites that are communicated with after operations are performed are called *final quorums*. Different quorums can apply for different operations and sites.

The schema Quorum_Set below declares the two sets that represent the initial and final quorums.

```
Quorum_Set

Final : \( P (\text{Operator} \times \text{Quorum}) \)
Initial : \( P (\text{Operator} \times \text{Quorum}) \)

\( \forall fq : \text{Quorum}; \text{op} : \text{Operator} \mid (\text{op},fq) \in \text{Final} \wedge fq \neq \emptyset \)
\( \forall iq : \text{Quorum}; \text{op} : \text{Operator} \mid (\text{op},iq) \in \text{Initial} \wedge iq \neq \emptyset \)
\( \forall \text{op} : \text{Operator} \cdot \)

Site =
\{ q1 : \text{Quorum}; s1 : \text{Site} \mid (\text{op},q1) \in \text{Final} \wedge s1 \in q1 \cdot s1 \} \wedge

Site =
\{ q1 : \text{Quorum}; s1 : \text{Site} \mid (\text{op},q1) \in \text{Initial} \wedge s1 \in q1 \cdot s1 \}

Final \neq \emptyset
Initial \neq \emptyset
```

Members of the sets *Initial* and *Final* are tuples of pairs of operator and quorum.

The predicate part of the schema Quorum_Set is not strictly necessary for the operation of the quorum consensus algorithm but, as a practical consideration, it is sensible to have each site included in at least one of the quorums, for there not to be any empty quorums, and the two sets of quorums not to be empty.

The relationships between the quorums are specified in the schema Intersection below.
The first predicate, $\{Q1\}$, states that all the initial quorums for the dequeue operator intersect all the final quorums for the enqueue operator.

The second predicate, $\{Q2\}$, states that all the initial quorums for the dequeue operator intersect all the final quorums for the dequeue operator.

No information is given about what the sets Final and Initial actually contain. The only restrictions are that the members must satisfy the properties given in the schemas Quorum_Set and Intersection. The actual replicated queue system would have the quorums set up for each site and these would be fixed (assuming no faults occur). However, at this level of abstraction, it is sufficient to assume that they exist and the one chosen on each occasion is selected by the omnipresent controller instead of being controlled by which site receives the request to perform each queue operation.

A single queue is represented by the schema Single_Q below.

A schema represents a single queue instead of a variable of the type bag of ELEMENT.
which provides another example of using Z schemas to represent multiple objects. Using schemas instead of other types also means that the characteristics of the queues can be changed easily.

The initial state of the single queue is an empty bag for the component $Q$.

\[
\text{Initial\_Single\_Q}
\]

\[
\text{Single\_Q'}
\]

\[
Q' = []
\]

\[\text{Proof Obligation for the Initial State}\]

The empty bag is always a valid bag, hence the initial state given above must exist.

It is necessary to define some operation to modify the bindings to the schema Single\_Q so that new elements can be added or old elements deleted.

\[
\text{Add\_Q}
\]

\[
\Delta\text{Single\_Q}
\]

\[
\text{new? : ELEMENT}
\]

\[
Q' = \text{Add\_Bag} (Q, \text{new?})
\]

Note that, from the definition of the bag operation \text{Delete\_Bag} given previously, deleting an element that is not in the bag results in no change to the object bound to the component $Q$. 

Delete_Q

\Delta \text{Single}_Q

\text{old?} : \text{ELEMENT}

Q' = \text{Delete_Bag}(Q, \text{old?})

Instead of schemas, functions for modifying the bindings to Single_Q could had been defined, but using schemas to represent operations ties up with the work in Chapter 5.

Three further operators are required to test the conditions of the current binding to the schema type Single_Q.

\begin{align*}
\text{Not Empty}_Q & : \mathbb{P} \text{ Single}_Q \\
\forall q : \text{Single}_Q \cdot \text{Not Empty}_Q q & \iff q.Q \neq []
\end{align*}

The prefix operator \text{Not Empty}_Q returns a true value if and only if the component Q is not an empty bag.

\begin{align*}
\text{Empty}_Q & : \mathbb{P} \text{ Single}_Q \\
\forall q : \text{Single}_Q \cdot \text{Empty}_Q q & \iff q.Q = []
\end{align*}

The prefix operator \text{Empty}_Q returns a true value if and only if the component Q is an empty bag.
The infix operator \textit{In\_Q} is true if and only if the given element is in the given queue value. The two functions are declared globally below and are defined for histories of operations.

\begin{center}
\texttt{Create\_Q : History \rightarrow Single\_Q} \\
\texttt{Merge : \mathcal{P} History \rightarrow History}
\end{center}

The function \texttt{Create\_Q} transforms a history of operations into a binding of the schema type \texttt{Single\_Q} which represents a particular queue state. An implementation of this function can be one that simulates the operation of a queue based on the set of operation and response pairs, and returns the final state of the queue. The primary purpose of the function \texttt{Create\_Q} is to link the histories that are used in later proof sketches with the values of queues and not to form part of any actual implementation.

The function \texttt{Merge} reconciles a set of records of operations into one consistent set of records. The function ensures that all operations mentioned in any history are included once only and that there are no duplicates nor missing operations.

Functions are used because each element in the domains is mapped to only one element in the ranges. Both the functions \texttt{Create\_Q} and \texttt{Merge} are many to one functions as several histories can result in the same queue and the same merged history can result from different combinations of distributed histories. Partial functions are used because some elements of the type \texttt{History} are not be valid records of operations capable of priority queues, and some elements of the type \texttt{\mathcal{P} History} can contain inconsistent information.
The variables declared in the schema Replication below are referred to by the site schemas defined later.

```
Replication

R_Q : Site → Single_Q
H_R : Site → History

∀ s : Site • R_Q s = Create_Q (H_R s)
```

The function $R_Q$ represents the whole network of queues and provides the mapping from a site to the queue located at that site. Each queue is represented as a binding with the schema type $\text{Single}_Q$.

The function $H_R$ gives the mapping from each site to the record of operations seen by that site.

The invariant in the schema Replication states that each single queue must be consistent with the historical record maintained for that queue.

The schema $\text{Site}_\text{Enq}$ specifies the change of state and preconditions for the enqueue operation.
Site_Enq

Quorum_Set
\Delta Replication
Op? : Operation
Res? : Response

First Op? = enq
\exists q^' : Single_Q; h : History \cdot
(\exists_{iq} : \text{Quorum} \mid (\text{enq},iq) \in \text{Initial}) \cdot
h = \text{Merge} \{s : \text{Site} \mid s \in iq \cdot H_R s\} \land
q^' =
(\mu Add_Q)
\theta \text{Single}_Q = \text{Create}_Q \ h \land \text{new?} = \text{Second Op?} \cdot
\theta (\text{Single}_Q')

Postconditions description of new state

\land
(\exists_{fq} : \text{Quorum} \mid (\text{enq},fq) \in \text{Final}) \cdot
Res? = (\text{okay},\text{null}) \land
H_{R^'} =
H_R \oplus
\{s : \text{Site} \mid s \in fq \cdot
s \mapsto H_R s \cup h \cup \{(\text{Op?},\text{Res?})\}\} \land
R_{Q^'} = R_Q' \oplus \{s : \text{Site} \mid s \in fq \cdot s \mapsto q^'\})

The schema Site_Enq does not model the behaviour of any particular site, but gives the
specification of all the sites in terms of the functions $H_R$ and $R_Q$.

The schema Site_Enq states that an initial quorum for the enqueue operation is selected
from the set of initial quorums such that the composite (merged) history of all operations
creates the current state of an hypothetical queue. This hypothetical queue is updated by adding the new element given in the input \( Op? \).

The postconditions of the schema Site_Enq are based on a final quorum being selected for the enqueue operation from the set of final quorums. All sites in that final quorum have their historical records and queues updated to reflect the changes.

The constructed queue, \( q' \), is existentially quantified to reduce the size of the predicates; only the value it represents is used in the new state of the system.

---

**Proof Obligation for the Schema Site_Enq**

Adherence to the type rules for the Z notation has been enforced by CADIz².

The functions \( H_R \) and \( R_Q \) are updated using functional override such that a set elements are modified, hence retaining their functional properties.

The invariant of the schema Replication is that all historical records for the sites given by the function \( H_R \) are linked to their corresponding queue value given by the function \( R_Q \). The linkage between histories and single queues is given by the function Create_\( Q \). The function Create_\( Q \) is not defined formally, but a basic requirement of the function is that all valid histories give rise to the same queue values as would a single queue described in the implementation. Therefore the invariants are satisfied by definition.

---

The schema Site_Deq below specifies the change of state and preconditions for the dequeue operation.
Site_Deq

Quorum_Set

\( \Delta \)Replication

Op?: Operation

Res!: Response

First Op? = deq

\( \exists q, q': \text{Single}_Q; h : \text{History} \cdot \)

\( \exists_1 iq : \text{Quorum} \mid (\text{deq}, iq) \in \text{Initial} \cdot \)

\( h = \text{Merge} \{ s : \text{Site} \mid s \in iq \cdot H_R s \} \land \)

\( q = \text{Create}_Q h \land \)

\( \text{Not_Empty}_Q q \land \)

\( \exists_1 eout : \text{ELEMENT} \mid eout \in Q q \cdot \)

\( (\forall e : \text{ELEMENT} \mid e \in Q q \cdot \)

\( \text{priority} eout > \text{priority} e) \)

Postconditions for non empty queue

\( \land \)

\( q' = \)

\( (\mu \text{Delete}_Q \mid \theta \text{Single}_Q = q \land \text{old}? = eout \cdot \)

\( \theta \text{Single}_Q') \land \)

\( \exists_1 fq : \text{Quorum} \mid (\text{deq}, fq) \in \text{Final} \cdot \)

\( \text{Res!} = (\text{okay}, eout) \land \)

\( H_R' = \)

\( H_R \oplus \)

\( \{ s : \text{Site} \mid s \in fq \cdot \)

\( s \rightarrow H_R s \cup h \cup \{(\text{Op?}, \text{Res!})\} \} \land \)

\( \text{merge} \{ s : \text{Site} \mid s \in fq \cdot \}

\( s \rightarrow H_R s \cup h \cup \{(\text{Op?}, \text{Res!})\} \} \land \)
An initial quorum for the dequeue operator is selected from the set of initial quorums and a composite history is formed from all the sites in that initial quorum. A queue is then constructed based on this history. The constructed queue is existentially quantified to reduce the complexity of the schema by reducing the number of times the term Create_Q h is repeated. An element value, eout, is chosen such that it is a member of the constructed queue and it has the highest priority. This element is deleted from the composite queue. A final quorum for the dequeue operator is then selected from the set of final quorums, and the historical records and queues for all the sites in the selected final quorum are updated.

Proof Obligation for the Schema Site_Deq

Adherence to the type rules for the Z notation has been enforced by CADi2.

The functions H_R and R_Q are updated using functional override such that a set elements are modified, hence retaining their functional properties.

The invariant of the schema Replication is that all historical records for the sites given by the function H_R are linked to their corresponding queue value given by the function R_Q is satisfied by definition of the function H_R.

The schema Queue_Imp specifies the replicated queue implementation of a priority queue.
Queue_Imp Def Intersection \land (Site_{Enq} \lor Site_{Deq})

Preconditions for the Schema Site_{Deq}

The schema Pre.Queue_Imp.Simple below specifies the simplified preconditions of the schema Queue_Imp.

\[
\begin{align*}
\text{Pre.Queue_Imp.Simple} & \\
\text{Intersection} & \\
\text{Replication} & \\
\text{Op? : Operation} & \\
\text{First Op? = enq} & \\
\lor & \\
\text{First Op? = deq} & \\
\end{align*}
\]

The correctness of these preconditions is expressed in the schema below.

Simplified_6_2 \equiv \text{pre Queue_Imp} \iff \text{Pre.Queue_Imp.Simple}

The initial state is defined by the following schema.
The schema `Initial_Queue_Imp` describes the state in which all the bindings in the range of the function `R_Q` are equal to the initial state of the schema `Single_Q`. Also, the function `H_R` maps all the elements in its domain to the empty set.

Proof Obligation for the Initial State

The definitions of the function `R_Q` and `H_R` have the correct functional types. The correspondence between no operations in the historical records for the sites and the initial queue value conforms to the requirement for the function `Create_Q` used in the schema `Replication`, hence an initial state given by the schema `Initial_Queue_Imp` must be possible.

### 6.2.3 Verification using Refinement Conditions of a Priority Queue

The schema `Verify` below expresses the correctness criterion for the implementation to be equivalent to the specification.

\[
\text{Verify} \triangleq \text{Queue}_\text{Imp} \iff \text{Queue}_\text{Spec}
\]

An alternative approach to the verification of the implementation is to prove that the implementation is a refinement of the specification by the three refinement conditions for the Z notation [Spiv89A, Dillr90, Pottr91]. It is the latter approach taken in this section and, for
comparison, the next section contains a proof sketch of the schema Verify. Appendix D.1 contains a simple animation of a priority queue which may help to understand the proof sketch in this section.

Abstraction Schema

Before the data refinement conditions are expressed it is necessary to define a schema linking the abstract (specification) and the concrete (implementation) states. The schema Abstract Queue below does this for the priority queue example.

```
Abstract Queue

Replication
Queue

q = (Create_Q (Merge (ran H_R))).Q
h = Merge (ran H_R)
```

The schema Abstract Queue uses the partial functions Create_Q and Merge to construct the components q and h based on the histories of operations.

The reason for adopting the above form for the abstraction schema is to be compatible with the requirement for the responses of replicated queue to be equivalent to the responses of a one copy queue. That is, the history of operations performed on the one copy queue is consistent with the history of operations performed on the collection of replicated queues.

Figure 6.3 illustrates the relationship between the concrete and abstract states that is provided by the schema Abstract Queue.
Figure 6.3 Schema Abstract Queue Mappings

Each abstract state in the specification is related by the abstraction schema to at least one concrete state in an implementation. This can be seen by considering any valid abstract history component, $h$, and queue component, $q$. A possible concrete state given by the schema Abstract Queue is a single site mapped to the same value as the history component $h$ by the function $H_R$, and the same site mapped to an equivalent queue component to $q$ by the function $R_Q$. This does not guarantee that the history $h$ is possible by the concrete queue operations in the implementation, although there is strong intuitive evidence to suggest that this is the case. Other solutions that involve several sites are also possible.

Because the functions Create Q and Merge are not specified formally it is not possible to prove the three refinement conditions formally. However, the conditions do provide the basis for informal arguments about the validity of the refinement process.

Initialization Condition

The initialization condition for this example is expressed in the schema equation Initial_Rule_Q below.
Initial_Rule_Q ≡

Initial_Queue_Impl ⇒

(∃ Queue' • Initial_Queue_Spec ∧ Abstract_Queue')

The schema Initial_Queue_Impl maps all sites in the domain of the function $H_R$ to empty sets. From the schema Abstract_Queue, the history used by the function Create_Q must be the empty set, which is the initial binding for the component $h$. A reasonable definition for the function Create_Q is that the empty set maps to the empty bag, which is the initial binding of the component $q$. This gives an initial state for the Initial_Queue_Spec schema.

Applicability Condition

The applicability condition is stated for the implementation of priority queue in the schema below.

Applicability_Rule_Q ≡

pre Queue_Spec ∧ Abstract_Queue ⇒ pre Queue_Impl

The preconditions for the schema Queue_Spec are simplified to those in the schema Pre_Queue_Spec_Simple. The preconditions for the schema Queue_Impl are simplified to those in the schema Pre_Queue_Impl_Simple.

Both preconditions rely solely on the input operation being either an enqueue or a dequeue operation, and assuming that all abstract states can be represented by a concrete state, the refinement activity satisfies the applicability condition.

Correctness Condition

The correctness condition is expressed in the schema Correctness_Rule_Q below for the implementation of the priority queue.
The proof sketch of the correctness of the data refinement uses the Lemma 6.1 below, but before the lemma can be stated an additional function must be defined.

\[
\begin{align*}
[X, Y] & \\
\text{apply } : (X \rightarrow Y \times \mathcal{P} X) \rightarrow \mathcal{P} Y & \\
\forall f : X \rightarrow Y; \text{ setX} : \mathcal{P} X \quad & \\
\text{apply } (f, \text{ setX}) = \{x : X \mid x \in \text{ setX} \cdot f x\}
\end{align*}
\]

The function apply applies a function to all the elements in a set to create another set of elements of the type given by the target type of the function.

**Lemma 6.1 \{Q1\} and \{Q2\}**

The proof sketch uses the following lemma about the characteristics of the replicated priority queues imposed by the intersection invariants \{Q1\} and \{Q2\} for the initial and final quorums.

\[
\begin{align*}
\text{Queue\_Imp } \vdash \\
\forall iq : \text{ Quorum } \mid (\text{ deq, iq} ) \in \text{ Initial } \quad & \\
\text{ Merge } (\text{ ran H\_R}) = \text{ Merge } (\text{ apply } (H\_R, iq))
\end{align*}
\]

Informally this lemma states that the composite history based on any initial quorum for the \text{ deq} operator is equal to the composite history based on the histories of all the sites.
Proof of Lemma 6.1

From the intersection invariant \( \{Q1\} \), each initial quorum for the \texttt{deq} operator intersects each final quorum for the \texttt{enq} operator. The schema \texttt{Site\_Enq} specifies that each operation appears in the historical records for all the sites in exactly one final quorum. Because there is always at least one site that contains a historical record of each enqueue operation, the composite history based on the historical record from the sites in any initial quorum for the \texttt{deq} operator must contain all the enqueue operations.

Similarly, from the intersection invariant \( \{Q2\} \), each initial quorum for the \texttt{deq} operator intersects each final quorum for the \texttt{deq} operator. The schema \texttt{Site\_Deq} specifies that each dequeue operation appears in the historical records for all the sites in exactly one final quorum. Because there is always at least one site that contains a historical record of each dequeue operation, the composite history based on the historical record from the sites in any initial quorum for the \texttt{deq} operator must contain all the dequeue operations.

Therefore, since the composite history based on any initial quorum for the \texttt{deq} operator contains all the enqueue and dequeue operations performed on any site, it must be equal to the composite history based on the historical records of all the sites.

Correctness Condition for the Priority Queue

The correctness proof required for the implementation of replicated priority queues meeting the specification of a one copy priority queue is represented diagrammatically by Figure 6.4. The correctness condition states, in effect, that there is an homomorphism between the abstract and concrete states using the abstraction schema \texttt{Abstract\_Queue}.
Let $A$ be any abstract state meeting the preconditions of the abstract operation $Queue\_Spec$ which represents a valid history of abstract operations.

Let $C$ be any concrete state from the set $\{C\}$ that is related to the abstract state, $A$, by the abstraction schema $Abstract\_Queue$.

Let $C'$ be an after concrete state that results from the concrete operation $Queue\_Imp$ in the concrete state $C$.

Let $A'$ be the abstract state that results from the abstract operation $Queue\_Spec$ in the abstract state $A$.

Note that the abstract operation $Queue\_Spec$ is deterministic for the ordering ‘$>$’, where each new state is determined uniquely by the schema.

The two after states $A'$ and $C'$ must be related by the abstraction schema $Abstract\_Queue'$ for the concrete operations to implement correctly the abstract operation.

One of the difficulties of the specification of the data refinement process at this level of abstraction is devising a suitable abstraction schema such that it is total over all the possible abstract states. Another difficulty is to ensure that the concrete states that correspond to any abstract state can be reached by using the concrete operations, otherwise the concrete state
is not valid. It cannot be assumed that concrete states given by the abstraction schema are always possible in this example, since that would assume that the historical records from the replicated queues is one copy serializable, which is the aim of the proof. The method used to overcome this difficulty is to apply an induction argument on the validity of the historical records. Assuming that for each valid state of the component q it is possible to construct a number of equivalent states of the component H_R, then the operations performed on these concrete states result in new valid concrete states that relate to new abstract states.

Consider an arbitrary abstract state A of the schema type Queue. From the schema Abstract_Q a number of valid concrete states are possible of the type Replication. Let one of these concrete states be C.

From the schema Abstract_Q:

\[ A.q = (Create_Q (Merge (ran C.H_R))).Q \]

\[ A.h = Merge (ran C.H_R) \]

where the notation A.q represents the q component of the binding of the schema type identified by A, similarly for A.h and C.H_R.

Both the schemas Site_Enq and Site_Deq update the component H_R for successful operations such that:

\[ C.H_R' = C.H_R \oplus \{ s : Site \mid s \in fq \cdot s \mapsto C.H_R \cdot s \cup C.h \cup \{(C.Op?, C.Res!)} \} \]

where

\[ C.h = Merge \{ s : Site \mid s \in iq \cdot C.H_R \cdot s \} \]

One of the laws for the override operator is the following [Spiv89A]:

\[ (ran f \oplus g) = (ran ((dom g) \triangleleft f)) \cup (ran g) \]

where the operator \( \triangleleft \) is domain subtraction.

Substituting
\[ f = C. H_R \]

and

\[ g = \{ s : \text{Site} \mid s \in f q \cdot s \mapsto C. H_R s \cup C.h \cup \{(C.Op?, C.Res!\}) \} \]

and the fact that:

\[(\text{dom} \{ s : \text{Site} \mid s \in f q \cdot s \mapsto C. H_R s \cup C.h \cup \{(C.Op?, C.Res!\}) \}) = f q\]

results in:

\[(\text{ran} C. H_R') = (\text{ran} (f q \triangleleft C. H_R)) \cup (\text{ran} \{ s : \text{Site} \mid s \in f q \cdot s \mapsto C. H_R s \cup C.h \cup \{(C.Op?, C.Res!\}) \})\]

This is equivalent to:

\[(\text{ran} C. H_R') = (\text{ran} (f q \triangleleft C. H_R)) \cup (\text{ran} \{ s : \text{Site} \mid s \in f q \cdot s \mapsto C. H_R s \}) \cup C.h \cup \{(C.Op?, C.Res!\})\]

The term

\[(\text{ran} \{ s : \text{Site} \mid s \in f q \cdot s \mapsto C. H_R s \})\]

is equal to

\[(f q \triangleleft C. H_R)\]

where the operator \(\triangleleft\) is domain restriction. The above term fills in those elements missed by the \(\triangleleft\) operator, and the equation simplifies to:

\[(\text{ran} C. H_R') = (\text{ran} C. H_R) \cup C.h \cup \{(C.Op?, C.Res!\})\]

and merging the historical records results in:

\[\text{Merge} (\text{ran} C. H_R') = \text{Merge} (\text{ran} C. H_R) \cup C.h \cup \{(C.Op?, C.Res!\})\]

Since for consistent histories:
**Merge** ((ran C.H.R) ∪ C.h) = **Merge** (ran C.H.R) ∪ C.h

From Lemma 6.1:

C.h = **Merge** (ran C.H.R)

hence

**Merge** (ran C.H.R') = **Merge** (ran C.H.R) ∪ {(C.Op?, C.Res!)}

The new state is now determined by a case analysis of the value of Op?:

1 Assume that the operation Op? has the enq operator

From schema Site_Enq it follows that:

C.Res! = (okay, null)

From schema Enqueue it follows that:

A.Res! = (okay, null)

and

A.h' = h ∪ {(A.Op?, A.Res!)}

The new abstract state component A.h' is related to the concrete state component C.H.R by the abstraction schema Abstract_Queue' with:

A.h = **Merge** (ran C.H.R)

and


Therefore proving the correctness of the implementation of the enqueue operation.

2 Assume that the operation Op? has the deq operator

From the schema Site_Deq:
\( C.q = Create_Q \ C.h \)

The history component \( C.h \) is in the domain of the function \( Create_Q \) because it is a valid history of operations.

From Lemma 6.1:

\[ C.q = Create_Q \ (Merge \ (ran \ C.H_R)) \]

and from \textit{Abstract\_Queue}:

\[ A.q = C.q.Q \]

There are two cases to consider.

2.1 \textit{Empty} \( C.q \)

From schema \textit{Site\_Deq}:

\[ C.Res \neq (error, null) \]

and

\[ C.H_R' = C.H_R \]

From schema \textit{Dequeue} with \( A.q = \emptyset \):

\[ A.Res \neq (error, null) \]

and

\[ A.h' = A.h \]

The schema \textit{Abstract\_Queue}' provides the relationship between the new abstract state and the new concrete state.

2.2 \textit{Not\_Empty\_Q} \( C.q \)

From schema \textit{Site\_Deq}:

\[ C.Res \neq (okay, C.eout) \]
and

\[ C.H_R' = C.H_R \oplus \{ s : Site \mid s \in f q \Rightarrow C.H_R \cup C.h \cup \}
\{(C.Op?, C.Res!\})\}

where

\( (\text{priority } C.eout) \triangleright (\text{priority } e) \)

for all \( e \) in \( C.q \).

From schema Dequeue with \( A.q \neq \{\} \):

\[ A.Res! = (\text{okay}, A.eout) \]

and

\[ A.h' = A.h \cup \{(A.Op?, A.Res!\})\]

where

\( (\text{priority } A.eout) \triangleright (\text{priority } e) \)

for all \( e \) in \( A.q \).

Since \( A.q = C.q \cup A.eout = C.eout \) (for order relation '>', or have same priority for order relation '≥'). The concrete state and abstract state are related by the schema Abstract Queue'.

If the original concrete and abstract states were valid (in the sense that they could be reached by using concrete and abstract operations) the new concrete and abstract states are also valid as they derived their values by using the operations.

This completes the proof that for equivalent histories, the replicated priority queues implementation gives identical behaviour as the specification of a one copy priority queue.
6.2.4 Verification using Proof Sketch of Equivalence Behaviour of a Priority Queue

This section is based on an induction on the stage of the history as seen by the equivalent one copy queue and any of the sites in the replicated system within the initial quorum of a deq operation.

The proof sketch is presented here for comparison with the proof sketches given in the previous section as part of the refinement process.

The aim of the proof is to show that the behaviour of the replicated queue is equivalent to that of a one copy queue. This is interpreted as a serializability property, with the enq and deq operations forming complete committed transactions.

The verification takes the form of:

 Specification ⇔ Implementation

The important property for replicated objects is one copy serialization, which is interpreted here that the replicated queues will always give the same deq response as a hypothetical one copy queue.

First consider the one copy producing the response Res! to a deq operation:

 Specification ⇒ Implementation

1 Base case

Consider the first deq operation seen by the one copy queue.

The one copy queue must have \( k \) enq operations, where \( k \geq 1 \).

The case of \( k = 0 \) is not allowed as the queue would be empty, resulting in an error response which does not appear in the history.

The initial quorum for the deq operation in the replicated system must contain at least one of the sites in each of the final quorums of the enq operations performed on the one copy queue. Therefore, the composite queue is also of length \( k \) and, in fact, contains identical elements as the one copy queue history.

As the element selection procedure in the deq operation is identical in both the one
copy queue and the replicated queue schemas, and the queues in both cases are identical, then the response from the replicated queue must also be \( \text{Res!} \) (assuming a total ordering relation using ‘\( > \)’, not ‘\( \geq \)’ where more than one element can have the same priority). If ‘\( \geq \)’ order relation is used, then the element chosen has the same priority as \( \text{Res!} \) and hence is equivalent in the context of this proof.

2 Induction step

Consider the case after \( m \) \texttt{deq} operations have taken place in the one copy queue.

Both the one copy queue and replicated queue systems have seen \( m \) \texttt{deq} operations and agree on the contents of the one copy queue and composite queue up to that point.

Between the \( m \text{th} \) and \( (m+1)\text{th} \) \texttt{deq} operation, the one copy queue could had seen \( k \) \texttt{enq} operations, where \( k \geq 0 \). The initial quorum for the \texttt{deq} operation in the replicated system must contain at least one of the sites in each final quorum of each \texttt{enq} operation performed on the one copy queue. If any \texttt{enq} operations occurred, they would had been recorded in the composite history, which would be identical to the history of the one copy queue. The rest of the argument follows as for the base case.

Now consider the replicated queue producing the value \( \text{Res!} \) for a \texttt{deq} operation.

\textbf{Implementation } \Rightarrow \textbf{Specification}

1 The base case

Consider the first \texttt{deq} operation seen by a site in the initial quorum of the \texttt{deq} operation.

The history of that site had seen \( k \) \texttt{enq} operations, where \( k \geq 1 \).

The one copy queue history must also had seen at least \( k \) \texttt{enq} operations because it sees all the operations. But it could possibly had seen more. However, if there were any such \texttt{enq} operations, then their final quorums did not intersect the initial quorum of the current \texttt{deq} operation, which is a contradiction of the invariant \((\text{Q1})\).

If any \texttt{deq} operation occurred at any other copy of the queue such that another copy of the queue is unaware of it, then there must be a final quorum for the \texttt{deq} operation
that does not intersect an initial quorum for the \texttt{deq} operation, which contradicts the invariant \(Q2\).

It follows that the histories must be identical and the result from the \texttt{deq} operation on the one copy queue is also \texttt{Res!}.

2 The induction step

Assume that the composite history seen by the sites in the initial quorum of a \texttt{deq} operation is identical to the history seen by the hypothetical one copy queue after the \textit{mth} \texttt{deq} operation.

The composite queue formed for the \((m+1)th\) \texttt{deq} operation may differ by \(k\) \texttt{enq} operations from the queue after the \textit{mth} \texttt{deq} operation, where \(k \geq 0\). A similar argument can be presented here to the base case for the one copy queue producing the same result of \texttt{Res!} due to the constraints \(Q1\) and \(Q2\).

Therefore, the replicated queue and the one copy queue will produce identical results for each \texttt{deq} operation.

### 6.3 Multiple Priority Queue

A multiple priority queue will always dequeue the element with the highest priority, but can dequeue that element more than once. The study of a multiple priority is split into the following sections:

- 6.3.1 Specification of a One Copy Multiple Priority Queue
- 6.3.2 Implementation of a Replicated Multiple Priority Queue
- 6.3.3 Verification using Refinement Conditions of a Multiple Priority Queue
- 6.3.4 Verification using a Proof Sketch of Equivalent Behaviour of a Multiple Priority Queue
6.3.1 Specification of a One Copy Multiple Priority Queue

Figure 6.5 illustrates the basic interactions between the schemas in the specification of the multiple priority queue.

Figure 6.5  Interactions in the Specification of a Multiple Priority Queue

The specification of the one copy queue requires a function to compile all the elements that have been dequeued from any of the replicated queues. This function is specified below.

\[
\text{Filter\_Deq : History} \rightarrow \mathbb{P} \text{ ELEMENT}
\]

\[
\forall h : \text{History} \cdot \\
\text{Filter\_Deq } h = \\
\{ e, eout : \text{ELEMENT}; t : \text{TIME}; s : \text{Status} \mid \\
((\text{deq}, e, t), (s, eout)) \in h \in eout \}
\]

A multiple priority queue is specified using three components as shown in the schema MP\_Queue below.
The bag \textit{present\_q} represents the elements enqueued and waiting to be dequeued, and the set \textit{absent\_q} represents the elements that have been dequeued. Elements are never removed from the latter set. The specification by Herlihy and Wing [Herl91] uses bags for both the components \textit{present\_q} and \textit{absent\_q}. Bags are not used here for the component \textit{absent\_q} because their properties are not required in the specification. The component \textit{h} represents the history of all successful operations performed on the queue.

The schema \texttt{MP\_Enqueue} adds the new element to the bag \textit{present\_q}. 

\begin{verbatim}
MP_Queue

present_q : bag ELEMENT
absent_q : P ELEMENT
h : History

present_q = Create_Bag h
absent_q = Filter_Deq h

\end{verbatim}

\begin{verbatim}
MP_Enqueue

\Delta MP_Queue

Op? : Operation
Res! : Response

First Op? = enq
present_q' = Add_Bag (present_q, Second Op?)
Res! = (okay, null)
h' = h \cup \{(Op?, Res!)}

No change to the absent_q component
absent_q' = absent_q

\end{verbatim}

The schema \texttt{MP\_Enqueue} adds the new element to the bag \textit{present\_q}. 

Proof Obligation for the Schema MP_Enqueue

Adherence to the type rules of the Z notation is enforced by CADiZ.

The link between the updated components present_q and h is based on the function Create_Bag, which is defined to do this for all enqueue operations.

The component absent_q is not changed by the schema MP_Enqueue and since the enq operator is added to the history

\[ \text{Filter\_Deq} \ h' = \text{Filter\_Deq} \ h = \text{absent}_q = \text{absent}_q' \]

Therefore, the invariants for the schema MP_Queue are maintained.
MP_Dequeue

ΔMP_Queue

Op? : Operation
Res! : Response

First Op? = deq

Preconditions for performing the operations

\[(\text{absent}_q \neq \emptyset \lor \text{present}_q \neq []) \land (\exists \ eout : \text{ELEMENT} \cdot \text{eout} \in \text{absent}_q \land (\forall e : \text{ELEMENT} \mid e \in \text{present}_q \cdot \text{priority eout} \geq \text{priority e}))\]

*Condition 1 for dequeue operation*

\[\land \ \text{Res!} = (\text{okay, eout}) \land h' = h \cup \{(\text{Op?}, \text{Res!})\} \land \text{absent}_q' = \text{absent}_q \land \text{present}_q' = \text{present}_q \lor \text{eout in present}_q \land (\forall e : \text{ELEMENT} \mid e \in \text{present}_q \cdot \text{priority eout} \geq \text{priority e})\]

*Condition 2 for dequeue operation*

\[\land \ \text{Res!} = (\text{okay, eout}) \land \text{absent}_q' = \text{absent}_q \cup \{\text{eout}\} \land \text{present}_q' = \text{Delete_Bag} (\text{present}_q, \text{eout}) \land h' = h \cup \{(\text{Op?}, \text{Res!})\}\]

\[\lor \]

\[\text{present}_q = []\]

Preconditions for not performing the operation
An element is dequeued as specified in the schema MP_Dequeue if either of the following two conditions are satisfied:

1. the element has already been dequeued at least once and its priority is greater than or equal to any element waiting to be dequeued, or
2. the element is waiting to be dequeued and it has the highest priority of all elements waiting to be dequeued.

The state of the queue is changed only as a consequence of the second condition. In the context of replicated queues, the first condition can occur when each queue has a record of all elements enqueued, but some copies can be out of date about which elements have been dequeued.

---

Proof Obligation for the Schema MP_Dequeue

Adherence to the type rules of the Z notation is enforced by CADiZ.

The link between the updated components present_q and h is based on the function Create_Bag, which is defined to do this for all enqueue operations.

For successful operations, the history component h is changed so that

$$h' = h \cup \{(\text{Op}, \text{Res})\}$$

where Res = (okay, eout)

From the definition of Filter_Deq

$$\text{Filter}_\text{Deq}(h \cup \{((\text{deq}, e, t), (s, eout))\}) = \text{Filter}_\text{Deq}(h)$$

$$\cup \text{Filter}_\text{Deq}(\{((\text{deq}, e, t), (s, eout))\})$$
which simplifies to

\[ \text{absent}_q \cup \{\text{eout}\} \]

For condition (1) in schema MP_Dequeue

\[ \text{eout} \in \text{absent}_q \]

hence

\[ \text{absent}_q \cup \{\text{eout}\} = \text{absent}_q \]

as is required.

For condition (2) in the schema MP_Dequeue the change to the component \( \text{absent}_q \) is independent of whether \( \text{eout} \) is a member or not, and is given by

\[ \text{absent}_q \cup \{\text{eout}\} \]

and is therefore related to the component \( h' \) by the function \( \text{Filter}_{\text{Deq}} \).

Inputs that give rise to the error status do not change the components of the schema MP_Queue hence must satisfy the invariants.

The paper by Herlihy and Wing [Herl91] specifies partially the behaviour of the multiple priority dequeue operation. The behaviour when both bags representing the present queue elements and past dequeue elements are empty is not specified in the paper, i.e. this can arise if the first dequeue operation occurs before any enqueue operations. The approach taken in the schema MP_Dequeue is to increase the non determinism when the bag \( \text{present}_q \) is empty. The response of the queue to a dequeue operation when \( \text{present}_q \) is empty is either an error message or a dequeue element from the set \( \text{absent}_q \) if that set is not empty. The choice is non deterministic.

A different approach (probably that assumed by Herlihy and Wing) is to return an error status only if both the bag \( \text{present}_q \) and set \( \text{absent}_q \) are empty.

The schema MP_Queue_Spec gives the specification of the required behaviour of a one
copy multiple priority queue.

\[ \text{MP\_Queue\_Spec} \triangleq \text{MP\_Enqueue} \lor \text{MP\_Dequeue} \]

**Preconditions for the Schema MP\_Queue\_Spec**

The schema Pre\_MP\_Queue\_Spec\_Simple below specifies the simplified preconditions of the schema MP\_Queue\_Spec.

\[
\begin{align*}
\text{Pre\_MP\_Queue\_Spec\_Simple} & \\
\text{MP\_Queue} & \\
\text{Op?} & : \text{Operation} \\
\text{First Op?} & = \text{enq} \\
\land \\
\text{First Op?} & = \text{deq}
\end{align*}
\]

The correctness of this schema is expressed as the schema below.

\[ \text{Simplified\_6\_3} \triangleq \text{pre MP\_Queue\_Spec} \iff \text{Pre\_MP\_Queue\_Spec\_Simple} \]

An initial state for the multiple priority queue is given in the schema Initial\_MP\_Spec below.
Proof Obligation for the Initial State

The above initial state is obviously a possible value of the schema type MP_Queue.

6.3.2 Implementation of a Replicated Multiple Priority Queue

The only change to the implementation of the multiple priority queue from that of the priority queue is to the schema Intersection. Figure 6.6 illustrates the interactions between the operation schemas and state schemas in the implementation of a multiple priority queue. This figure is very similar to Figure 6.2 and is included here to ease references to the schemas.
The change reflects a different quorum consensus relation as specified in the schema MP_Intersection below.

\[ (Q1) \]
\[ \forall iq : Quorum \mid (deq, iq) \in \text{Initial} \cdot \forall fq : Quorum \mid (enq, fq) \in \text{Final} \cdot iq \cap fq \neq \emptyset \]

The predicate \((Q1)\) specifies that each initial quorum for the dequeue operator intersects
each final quorum for the enqueue operator.

The schema MP_Queue_Imp below incorporates the disjunction of the site schemas Site_Enq and Site_Deq in conjunction with the modified invariant schema MP_Intersection.

\[
MP_{\text{Queue}_\text{Imp}} \equiv MP_{\text{Intersection}} \land (\text{Site}_\text{Enq} \lor \text{Site}_\text{Deq})
\]

There are no proof obligations for the schema MP_Queue_Imp in addition to those presented before for the schemas Site_Enq and Site_Deq because there are no additional changes introduced by this new schema.

**Preconditions for the Schema MP_Queue_Imp**

The predicate part of the preconditions in the schema Pre_MP_Queue_Imp_Simple below is the same as that of the schema Pre_Queue_Imp_Simple.

\[
\begin{align*}
\text{Pre}_{\text{MP}_\text{Queue}_\text{Imp}_\text{Simple}} & \equiv & \text{MP}_\text{Intersection} \\ & \land & \text{Replication} \\ & \land & \text{Op}?: \text{Operation} \\
& \rightarrow & \begin{cases} 
\text{First Op}?: \text{enq} \\
\lor \\
\text{First Op}?: \text{deq}
\end{cases}
\end{align*}
\]

The correctness of the simplified preconditions is expressed in the schema below.

\[
\text{Simplified}_6.4 \equiv \pre_{\text{MP}_\text{Queue}_\text{Imp}} \iff \text{Pre}_{\text{MP}_\text{Queue}_\text{Imp}_\text{Simple}}
\]

The initial state is given by the schema Initial_Queue_Imp introduced in Section 6.2.2.
6.3.3 Verification using Refinement Conditions of a Multiple Priority Queue

The formal statement of the verification condition of the equivalence between the implementation and the specification is represented by the schema MP_Verify below.

\[ \text{MP}_\text{Verify} \equiv \text{MP}_\text{Queue}_\text{Imp} \iff \text{MP}_\text{Queue}_\text{Spec} \]

This section contains a proof sketch of the three data refinement conditions.

**Abstraction Schema**

The abstraction schema for the multiple priority queue implementation is the schema Abstract_MP below.

\[
\begin{align*}
\text{Abstract_MP} \\
\text{MP}_\text{Queue} \\
\text{Replication} \\
\text{present}_q &= (\text{Create}_Q (\text{Merge} (\text{ran} H_R))). Q \\
\text{absent}_q &= \text{Filter}_\text{Deq} (\text{Merge} (\text{ran} H_R)) \\
\text{h} &= \text{Merge} (\text{ran} H_R)
\end{align*}
\]

It is possible to construct concrete history records represented by \( H_R \) to be related by the schema Abstract_MP to the abstract components \( \text{present}_q, \text{abstract}_q \) and \( h \). The difficulty is proving that the function \( H_R \) is valid for the implementation without making assumptions about the behaviour of the implementation. The relationship between the operations possible by the specification and implementation is postponed until the analysis of the correctness condition below.

**Initialization Condition**

The initialization condition for the multiple priority queue implementation is specified in the schema Initial_Rule_MEP below.
Initial_Rule_MP ≜
Initial_Queue_Imp ⇒
(∃ MP_Queue' • Initial_MP_Spec ∧ Abstract_MP')

The schema Initial_Queue_Imp specifies that the function H_R maps each site to the empty set. From the schema Abstract_MP, the corresponding values for the components of the schema MP_Queue are present_q = [], absent_q = ∅ and h = ∅ which are the values given by the schema Initial_MP_Spec.

**Applicability Condition**

The applicability condition for the multiple priority queue implementation is specified by the schema Applicability_Rule_MP below.

**Applicability_Rule_MP**

pre MP_Queue_Spec ∧ Abstract_MP ⇒ pre MP_Queue_Imp

As both schemas are total over all values of inputs when in valid states, it follows that this condition is true. However, it is possible for there not to be a valid concrete state to correspond to a valid abstract state, this possibility is eliminated by the correctness condition.

**Correctness Condition**

The correctness condition for the data refinement of the multiple priority queue implementation is given in the schema Correctness_Rule_MP below.

**Correctness_Rule_MP**

pre MP_Queue_Spec ∧ Abstract_MP ∧ Queue_Imp ⇒
(∃ MP_Queue' • Abstract_MP' ∧ MP_Queue_Spec)

The proof sketch of the correctness condition uses Lemma 6.2 stated below.

Before reading the proof below it may be helpful to look at the manually produced
animation of the schemas for the priority queues contained in Appendix D.2.

The statement of Lemma 6.2 uses the predicate bag_subset that is true if the left hand bag is contained within the right hand bag.

\[
\begin{align*}
\text{bag_subset} : & \text{ bag } X \leftrightarrow \text{ bag } X \\
\forall b_1, b_2 : & \text{ bag } X \quad \text{such that} \\
& b_1 \text{ bag_subset } b_2 \iff (\forall x : X \cdot \text{ count } b_2 x \geq \text{ count } b_1 x)
\end{align*}
\]

Lemma 6.2 \([Q_I]\)

\[
\begin{align*}
\text{MP} \_	ext{Queue} \_\text{Imp} \downarrow \\
\forall \text{ iq : Quorum } \mid (\text{ deq, iq } \in \text{ Initial} \quad \rightarrow \\
\quad (\text{ Create } Q \ (\text{ Merge } (\text{ ran } H \_R))) \ . Q \ \text{ bag_subset} \\
\quad (\text{ Create } Q \ (\text{ Merge } (\text{ apply } (H \_R, \text{ iq})))) \ . Q
\end{align*}
\]

Lemma 6.2 states that from the intersection invariant \([Q_I]\), the component \(Q\) created from the composite historical records of an initial quorum for the deq operator contains at least all those elements in any component \(Q\) based on the composite historical records from all the sites.

However, the component \(Q\) from the whole range of sites can contain fewer elements because some dequeue operations are missing from the composite historical records from the initial quorum of sites.

Proof of Lemma 6.2

From the intersection invariant \([Q_I]\), each initial quorum for the deq operator intersects each final quorum for the enq operator. The schema Site\_Enq specifies that each enqueue operation is inserted in the historical records of all the sites in a final
quorum for the enq operator. The schema Site_Deq specifies that a composite history is formed from all the historical records from the sites in an initial quorum for the deq operator. Therefore, the composite history must contain all the enqueue operations performed on any site in the network.

**Correctness Condition for the Multiple Priority Queue**

The proof of the correctness of the implementation of a multiple priority queue is very similar to that of the priority queue. The approach is depicted by Figure 6.7.

**Figure 6.7 Data Refinement for Multiple Priority Queues**

Note that for some abstract states the application of the abstract operation can result in a number of new abstracts and the one selected for any particular application of the operation cannot be determined before the operation takes place.

From the schema Abstract_MP:

A.present \_q = (Create\_Q (Merge (ran C.H.R))).Q

A.absent \_q = Filter\_Deq (Merge (ran C.H.R))

A.h = Merge (ran C.H.R)

The following analysis assumes that the before states for both the concrete and abstract
operations are valid, and proves that the after states for the concrete and abstract operations are related by the schema Abstract_MP'.

Consider the case analysis of possible values for Op?:

1. First Op? = enq

   From schema Site_Enq:

   \[ C.H_R' = C.H_R \oplus \{ s : Site \mid s \in fq \cdot s \mapsto C.H_R s \cup C.h \cup \{(C.Op?, C.Res!}\} \]

   where

   \[ C.h = \text{Merge} \{ s : Site \mid s \in iq \cdot C.H_R s \} \]

   \[ \text{Merge} (\text{ran } C.H_R') = \text{Merge} (\text{ran } C.H_R) \cup C.h \cup \{(C.Op?, C.Res!}\} \]

   Since C.h must represent a sub history of all the sites:

   \[ C.h \subseteq \text{Merge} (\text{ran } C.H_R) \]

   \[ \text{Merge} (\text{ran } C.H_R') = \text{Merge} (\text{ran } C.H_R) \cup \{(C.Op?, C.Res!}\} \]

   From schema Site_Enq it follows that:

   \[ C.Res! = (\text{okay, null}) \]

   From schema MP_Enqueue it follows that:

   \[ A.Res! = (\text{okay, null}) \]

   and

   \[ A.h' = h \cup \{(A.Op?, A.Res!}\} \]

   The after abstract state component A.h' is related to the after concrete state component C.H_R by the schema Abstract_MP', proving the correctness of the implementation of the enqueue operation. Note that for the enqueue abstract operation the next abstract state is deterministic.
First Op? = deq

From schema Site_Deq:

\[ C.q = \text{Create}_Q \ C.h \]

There are two possible cases for \( C.q \) to be considered.

2.1 \textit{Empty}_Q \ C.q

From schema Site_Deq:

\[ C.\text{Res!} = (\text{error}, \text{null}) \text{ and } C.\text{H}_R' = C.\text{H}_R \]

From Lemma 6.2, if \((\text{Empty}_Q \ C.q)\), then

\((\text{Empty}_Q \ \text{Create}_Q (\text{Merge} (\text{ran} \ H_R)))\)

and, from the schema Abstract_MP, it follows that \( A.\text{present}_q = \emptyset \)

Schema Dequeue with \( A.\text{present}_q = \emptyset \) results in:

\[ A.\text{Res!} = (\text{error}, \text{null}) \text{ and } A.\text{h}' = A.h \]

The new abstract state is related to the new concrete state by the schema Abstract_MP'.

However, the schema MP_Dequeue can also dequeue an element from the set \( A.\text{absent}_q \) if the set is not empty. Moreover, if a different quorum is used then it is possible for \((\text{Not Empty}_Q \ C.q)\) to be true. Also, by suitable manipulation of the choice of quorums, the same element selected from \( A.\text{absent}_q \) can be selected from the newly constructed \( C.q \). This highlights the non determinism of both the specification and implementation of the dequeue operations under some conditions.

2.2 \textit{Not Empty}_Q \ C.q

From schema Site_Deq:

\[ C.\text{Res!} = (\text{okay}, C.eout) \]

and
\[ C.H_R' = C.H_R \oplus \{ s : Site \mid s \in \mathcal{F}_q \land s \mapsto C.H_R \cup C.h \cup ((C.Op?, C.Res!)) \} \]

where

\((\text{priority } C.eout) \geq (\text{priority } e)\)

for all \(e\) in \(C.q\).

From schema Dequeue with \(A.\text{present}_q = \emptyset\):

\[ A.\text{Res}! = (\text{okay}, A.\text{eout}) \]

and

\[ A.h' = A.h \cup \{(A.Op?, A.\text{Res}!)\} \]

where

\((\text{priority } A.eout) \geq (\text{priority } e)\)

for all \(e\) in \(A.\text{present}_q\).

Since \(A.\text{present}_q \subseteq C.q.Q\) and \(A.\text{eout} \leq C.eout\), if the priority of \(C.eout\) is the same as the priority of \(A.eout\), then the next abstract state of the schema MP\_Queue\_Spec is related to the concrete state by the schema Abstract\_MP'.

However, this is not the only response possible by the schema MP\_Dequeue:

1. The value of the priority of \(A.eout\) can be less than that of \(C.eout\). This can occur if the abstract element with the same priority as \(C.eout\) has been removed by a dequeue operation on the abstract queue and the equivalent concrete operation is not recorded by any site in the initial quorum used for this particular dequeue operation.

2. If the abstract element with the priority of \(C.eout\) has been dequeued, then the element is a member of the set \(A.absent_q\) and is still a possible result of the schema MP\_Dequeue. This results in the composite history of the same
element being dequeued more than once.

3 It is also possible for $A.present_q = \emptyset$, which results if addition dequeue operations have occurred at sites outside the initial quorum for the $\text{deq}$ operator chosen for this particular dequeue operation. The response in this case is an error message.

All three responses can also be produced by the concrete operation by a different selection of initial and final quorums. Moreover, there will be limits on how well the implementation can reproduce identical behaviour of the specification. For example, from the specification of the dequeue operation, an element can be dequeued an infinite number of times provided its priority is greater than or equal to any element in the $present_q$ component of the queue. However, the implementation can only dequeue the same element once from each copy of the queue, hence to dequeue the same element an infinite number of times requires an infinite number of sites.

The possible behaviours represented by the schemas $\text{MP\_Queue\_Spec}$ and $\text{MP\_Queue\_Imp}$ are not deterministic for the dequeue operation. The response taken by the schema $\text{MP\_Queue\_Imp}$ depends on the initial quorum chosen for the dequeue operation, and that taken by the schema $\text{MP\_Queue\_Imp}$ depends on whether the element dequeued comes for the bag $present_q$ or the set $absent_q$.

It has been shown that both the specification and implementation can produce the same type of responses when exposed to the same set of historical events. However, there are some differences between the exact behaviours of the specification and implementation.

Note that, both the concrete and abstract after states are valid states because they can be reached using concrete and abstraction operations respectively.

6.3.4 Verification using a Proof Sketch of Equivalent Behaviour of a Multiple Priority Queue

The aim of the proof is to demonstrate that the behaviour of the replicated queue is equivalent to the behaviour of a one copy queue. The constraints of a one copy queue have been relaxed to allow a greater range of behaviours than for priority queues.
The proof sketch of equivalence is done by induction on the number of deq operations seen by the one copy queue and the replicated queue. What is proved is that any behaviour of one can also be the behaviour of the other. As the queues are not deterministic, it is not necessary for the queue to produce identical behaviour, only that each is capable of identical behaviour.

The first step is to prove that any behaviour of the one copy queue is also capable of the replicated queue.

1 Base case

The first deq operation seen by the one copy queue specification and the element eout dequeued.

From the invariant \( \{Q1\} \), all previous enq operations are recorded in the composite queue formed from the initial quorum of sites for the deq operation. Therefore, the present_\(q\) component of the one copy queue and composite queue agree on entries at this point. Also, the absent_\(q\) component is empty.

The one copy queue schema selects the highest priority element in the present_\(q\) component of the queue, which is identical to the composite queue.

2 Induction step

Assume equivalent behaviour up to the \(m\)th deq operation.

Let the eout element be dequeued by the \((m+1)\)th deq operation seen by the one copy queue. There are two cases possible:

2.1 \(eout \in \text{absent}_q\)

This occurs if the element was previously dequeued.

In the replicated queue the same element can be dequeued more than once if two sites do not have their initial and final quorums intersecting. This is expressed for any two sites \(i\) and \(j\) that dequeue the same element eout as:

\[
\exists iqi : \text{Quorum} \cdot (\text{deq}, iqi) \in \text{Initial} \land \\
\exists fqj : \text{Quorum} \cdot (\text{deq}, fqj) \in \text{Final} \land iqi \cap fqj = \emptyset
\]
This does not conflict with the invariant \( \{Q1\} \), hence it is possible to construct the quorums to meet these conditions. Therefore such a behaviour is possible for replicated queues provided that the corresponding \texttt{enq} operation is in the composite history of all sites, which it is from the invariant \( \{Q1\} \).

2.2 \( eout \in \text{present}_q \)

This corresponds to the case of the replicated priority queue, hence is possible behaviour of the replicated multiple priority queue as the schemas are less restrictive.

Now it is necessary to prove that all the possible behaviours of the replicated queue are possible by the one copy queue. The proof sketch is based on induction on the number of \texttt{deq} operations.

1. The base step of the induction

The first \texttt{deq} operation seen by the composition of all the sites in an initial quorum for a \texttt{deq} operation.

It is possible for the one copy queue to have seen previous \texttt{deq} operations as the final quorums for these \texttt{deq} operations need not intersect the initial quorum for this \texttt{deq} operation. However, the one copy queue can dequeue the same element more than once, so such behaviour is consistent with the one copy queue specification.

2. Induction step

The behaviours are equally possible up to the \textit{mth} step seen by the composition of all the sites in an initial quorum for a \texttt{deq} operation.

The reasoning for an equivalent behaviour being possible for the one copy queue follows in a similar manner to the base case.

Therefore the behaviour of one queue is also possible by the other queue.
6.4 Out of Order Priority Queue

An out of order priority queue can dequeue an element once only, but the order elements dequeued are not in priority order. The study of a priority is split into the following sections:

6.4.1 Specification of a One Copy Out of Order Priority Queue

6.4.2 Implementation of a Replicated Out of Order Priority Queue

6.4.3 Verification using Refinement Conditions of an Out of Order Priority Queue

6.4.4 Verification using a Proof Sketch of Equivalent Behaviour of an Out of Order Priority Queue

6.4.1 Specification of a One Copy Out of Order Priority Queue

Figure 6.8 illustrates the interactions between the operation schemas and state schema in the specification of an out of order priority queue.

Figure 6.8 Interactions in the Specification of an Out of Order Priority Queue

The schema OO_Queue has components for the bag of elements and a history of operations.
The schema **OO_Enqueue** below specifies the conditions for adding new elements to the queue.

\[
\begin{align*}
\text{OO\_Queue} \quad & \quad q : \text{bag ELEMENT} \\
& \quad h : \text{History} \\
q &= \text{Create\_Bag } h
\end{align*}
\]

The schema **OO_Enqueue** below specifies the conditions for adding new elements to the queue.

\[
\begin{align*}
\text{OO\_Enqueue} \quad & \quad \Delta \text{OO\_Queue} \\
& \quad \text{Op?} : \text{Operation} \\
& \quad \text{Res!} : \text{Response} \\
\text{First Op?} &= \text{enq} \\
q' &= \text{Add\_Bag } (q, \text{Second Op?}) \\
\text{Res!} &= (\text{okay}, \text{null}) \\
h' &= h \cup \{(\text{Op?}, \text{Res!})\}
\end{align*}
\]

**Proof Obligation for the Schema OO_Enqueue**

Adherence to the type rules of the Z notation is enforced by CADiZ.

The invariant of the schema **Queue** is based on the function **Create\_Bag** being able to re-construct \( q' \) from the updated history component \( h' \). This is a basic requirement of **Create\_Bag**, hence the invariant must be satisfied.

The schema **OO\_Dequeue** specifies the selection of any element to be removed from the
queue in response to a deq operation.

\[
\begin{align*}
\text{OO_Dequeue} & \quad \Delta \text{OO_Queue} \\
\text{Op?} : \text{Operation} & \\
\text{Res!} : \text{Response} & \\
\text{First Op?} = \text{deq} & \\
q \neq \emptyset & \\
(\exists \, \text{eout}: \text{ELEMENT} \cdot \\
\text{eout} \in q \land q' = \text{Delete_Bag} (q, \text{eout}) & \\
\text{Res!} = (\text{okay, eout}) \land h' = h \cup \{(\text{Op?, Res!})\}) \\
\lor & \\
q = \emptyset \land \emptyset \text{OO_Queue'} = \emptyset \text{OO_Queue} \land \text{Res!} = (\text{error, null})
\end{align*}
\]

Proof Obligation for the Schema OO_Dequeue

Adherence to the type rules of the Z notation is enforced by CADiZ.

The invariant of the schema Queue is based on the function Create_Bag being able to re-construct \( q' \) from the updated history component \( h' \). This is a basic requirement of Create_Bag, hence the invariant must be satisfied.

The schema OO_Queue_Spec below combines the schemas for the enqueue and dequeue operations.

\[
\text{OO_Queue_Spec} \equiv \text{OO_Enqueue} \lor \text{OO_Dequeue}
\]
Preconditions for the Schema OO_Queue_Spec

The schema Pre ОО_Queue_Spec_Simple specifies the simplified preconditions of the schema ОО_Queue_Spec.

The initial state for the out of order queue is specified in the schema below.

Simplified_6_5 \triangleq pre OO_Queue_Spec \iff Pre ОО_Queue_Spec_Simple

The initial state for the out of order queue is specified in the schema below.
Proof Obligation for the Initial State

The schema Initial ОО.Queue_Spec obviously describes a subset of the possible bindings of the schema ОО.Queue.

6.4.2 Implementation of a Replicated Out of Order Priority Queue

The only schema changed is the schema Intersection, see Figure 6.9.

Figure 6.9 Interactions in the Implementation of an Out of Order Priority Queue

The invariants are modified to give the schema ОО.Intersection below.
The schema $\text{OO\_Intersection}$ specifies the predicate $\{Q2\}$ only. This predicate states that each initial quorum for the dequeue operator must intersect each final quorum for the dequeue operator.

$\text{OO\_Queue\_Imp} \equiv \text{OO\_Intersection} \land (\text{Site\_Enq} \lor \text{Site\_Deq})$

The schema $\text{OO\_Queue\_Imp}$ forms the conjunction between the previous site implementation schemas and the new intersection invariant schema.

*Preconditions for the Schema $\text{OO\_Queue\_Imp}$*

The schema $\text{Pre\_OO\_Queue\_Imp\_Simple}$ below specifies the simplified preconditions of the implementation of the replicated out of order priority queue.
Note that the predicate part of the schema \( \text{Pre}_\text{OO}_\text{Queue}_\text{Imp}_\text{Simple} \) is the same as that in the two previous implementations of replicated queues, but different schemas are included in the declaration part which impose different constraints on the behaviour of the system.

The correctness of the preconditions in the schema \( \text{Pre}_\text{OO}_\text{Queue}_\text{Imp}_\text{Simple} \) is expressed in the schema below.

\[
\text{Simplified}_6 \equiv \quad \pre \text{OO}_\text{Queue}_\text{Imp} \iff \text{Pre}_\text{OO}_\text{Queue}_\text{Imp}_\text{Simple}
\]

### 6.4.3 Verification using Refinement Conditions of an Out of Order Priority Queue

The schema below gives a formal statement of the correctness criterion of the equivalence between the implementation and specification of out of order priority queues.

\[
\text{OO}_\text{Verify} \equiv \text{OO}_\text{Queue}_\text{Imp} \iff \text{OO}_\text{Queue}_\text{Spec}
\]

This section contains a proof sketch of the data refinement conditions for the out of order priority queue implementation and the next section contains a proof sketch for the
equivalent behaviours implied by the specification and implementation schemas.

*Abstraction Schema*

The abstraction schema for the out of order priority queue implementation is given by the schema Abstract OO below.

```
Abstract OO

Replication
OO_Queue

q = (Create_Q (Merge (ran H_R))). Q
h = Merge (ran H_R)
```

Except for the invariants contained in the schemas included in the data declaration part of the schema Abstract OO, it is identical to the abstraction schema Abstract Queue.

The relationship between the abstract and concrete history records is discussed in the analysis of the correctness condition.

*Initialization Condition*

The initialization condition for the out of order priority queue implementation is expressed in the schema Initial_Rule OO below.

```
Initial_Rule OO

Initial_Queue_Imp ⇒
(∃ OO_Queue' • Initial OO_Queue_Spec ∧ Abstract OO')
```

The schema Initial_Queue_Imp specifies the binding of each site in the domain of the function $H_R$ is an empty set. The schema Abstract OO' gives the values of $q = []$ and $h = ∅$, which correspond to the values given by the schema Initial OO_Queue_Spec.
Applicability Condition

The applicability condition of the data refinement process for the out of order priority queue is expressed in the schema Applicability_Rule_OO below.

\[
\text{Applicability\_Rule\_OO} \equiv \\
\text{pre } \text{OO\_Queue\_Spec} \land \text{Abstract\_OO} \Rightarrow \text{pre } \text{OO\_Queue\_Imp}
\]

Both specification and implementation schemas are total over all the input values in all valid states, hence the proof is trivial, assuming that valid states are related by the schema Abstract\_OO'.

Correctness Condition

The correctness condition for the data refinement of the out of order priority queue is expressed in the schema Correctness_Rule_OO below.

\[
\text{Correctness\_Rule\_OO} \equiv \\
\text{pre } \text{OO\_Queue\_Spec} \land \text{Abstract\_OO} \land \text{Queue\_Imp} \Rightarrow \\
(\exists \text{OO\_Queue'} \cdot \text{Abstract\_OO'} \land \text{OO\_Queue\_Spec})
\]

Before reading the proof sketch below it may be helpful to look at the simple animation of the schemas for the out of order priority queue in Appendix D.3.

The proof sketch of the correct implementation of an out of order priority queue by a network of replicated priority queues uses Lemma 6.3 below.
Lemma 6.3 \((Q2)\)

\[
\text{OO\_Queue\_Imp } \vdash \\
\forall iq : \text{Quorum } | \ (\text{deq}, iq) \in \text{Initial} \cdot \\
(\text{Create\_Q} (\text{Merge} (\text{apply} (\text{H}_R, iq)))) . Q \ \text{bag\_subset} \\
(\text{Create\_Q} (\text{Merge} (\text{ran} H_R))) . Q
\]

Lemma 6.6 states that, from the intersection invariant \((Q2)\), whichever initial quorum is selected from \text{Initial} for the \text{deq} operator, the composite history created from the sites in that quorum will contain all the dequeue operations in the composite history created from all the sites in the network.

Proof of Lemma 6.3

There must be at least one site in common between the initial and final quorums because, whichever final quorum is used by a dequeue operation, the initial quorums for the \text{deq} operator intersect the final quorums for the \text{deq} operator. It follows that any subsequent \text{deq} operation detects the occurrences of all previous dequeue operations from the historical records of the sites in an initial quorum.

Therefore, any queue created from an initial quorum contains either the same elements or fewer elements than a queue created from the whole network. Fewer elements are possible if enqueue operations occur at sites that do not have final enqueue quorums that intersect the initial quorum for the dequeue operation.

Note that the combination of Lemmas 6.2 and 6.3 gives an equivalent result to Lemma 6.1.

Correctness Condition for the Out of Order Priority Queue

The basis of the correctness proof is represented by Figure 6.10.
The abstract state $A$ is related to a number of possible concrete states by the schema Abstract ОО.

As the result of the concrete operation described by the schema OO_Queue_Imp, a number of new concrete states are possible; the actual one depends on the initial quorum used for the operation.

As the result of the abstract operation described by the schema OO_Queue_Spec, a number of new abstract states are possible; the actual one chosen depends on the element selected from the queue.

Whichever concrete state is chosen, all the new concrete states can be related to a new abstract state, however, some of these abstract states cannot be reached from the abstract state $A$ using the operation OO_Queue_Spec, hence invalidating the method of proving the correctness of the implementation with the chosen abstraction schema Abstract ОО. It may be possible to use a more complex abstraction schema to overcome this problem, but the difficulty stems from the freedom of the initial and final quorums; any restrictions imposed on these would impact the description of the implementation. This results in verifying a different implementation to the one required.
Continuing the analysis of the implementation through the correctness condition is a useful exercise because it reveals similarities between the specification and implementation.

From the schema Abstract ОО:

\[ A.q = (Create_Q (Merge (ran C.H_R))).Q \]

\[ A.h = Merge (ran C.H_R) \]

The new states are determined by case analysis of the values of Op?:

1. **First Op? = enq**

   From the schema Site_Enq:

   \[ C.H_R' = C.H_R \oplus \{ s : Site \mid s \in fq \cdot s \rightarrow C.H_R s \cup C.h \cup \{(C.Op?, C.Res!)\} \]  

   where

   \[ C.h = Merge \{ s : Site \mid s \in iq \cdot C.H_R s \} \]

   Since \( C.h \) is a sub history of the complete network, it is also a subset of the history recorded by all the sites:

   \[ C.h \subseteq Merge (ran C.H_R) \]

   \[ Merge (ran C.H_R') = Merge (ran C.H_R) \cup \{(C.Op?, C.Res!)\} \]

   From schema Site_Enq it follows that:

   \[ C.Res! = (okay, null) \]

   From schema ОО_Enqueue it follows that:

   \[ A.Res! = (okay, null) \]

   and

   \[ A.h' = h \cup \{(A.Op?, A.Res!)\} \]
The after abstract state component $A.h'$ is related to the after concrete state component $C.H'_R$ by the schema Abstract_OO', proving the correctness of the implementation of the enqueue operation. Note that the abstract enqueue operation is deterministic and the concrete enqueue operation is similarly deterministic for each initial quorum, also the sets of new concrete states are homomorphic with the new abstract state using the abstraction schema.

2 First $Op? = deq$

From schema Site_Deq:

$$C.h = \text{Merge} \{ s : \text{Site} \mid s \in iq \rightarrow C.H' R s \}$$

$$C.q = Create_Q C.h$$

There are two cases for $C.q$ to be considered.

2.1 Empty $Q C.q$

From schema Site_Deq:

$$C.Res! = (error, null) \text{ and } C.H'_R = C.H_R$$

Schema OO_Dequeue with $A.q = \emptyset$ results in:

$$A.Res! = (error, null) \text{ and } A.h' = A.h$$

The new abstract state is related to the new concrete state by the schema Abstract_OO'.

However, it is possible for $A.q \neq \emptyset$, in which case such response is not possible. This is an example of a new concrete state being related to new abstract states that cannot be reached by an application of the abstract operation from the previous abstract state. That is, if the original abstract state included a non empty queue then it is not possible for the new abstract state to give the response status of error. However, it is possible for this response to be given by a new concrete state as a result of the concrete operation being performed on a concrete state related to the initial abstract state; see animation in Appendix D.3. Of course, given suitable choice of initial quorum, a non empty queue response is still
possible by the concrete operation.

As was mentioned earlier, a method of avoiding this specification difficulty is to change the abstraction schema such that it ensures that only suitable initial quorums are chosen, but this is imposing additional constraints on the implementation that are not present in its description.

2.2 Not_Empty_Q C.q

From schema Site_Deq:

\[ C.Res! = (okay, C.eout) \]

and

\[ C.H_R' = C.H_R \oplus \{ s : Site \mid s \in fq \cdot s \rightarrow C.H_R s \cup C.h \cup \{(C.Op?, C.Res!\}) \} \]

where

\[(priority C.eout) \geq (priority e)\]

for all \( e \) in \( C.q \).

If (Not_Empty_Q C.q), then A.q must also be not empty from Lemma 6.3.

From schema OO_Dequeue with Aq \#= []:

\[ A.Res! = (okay, A.eout) \]

and

\[ A.h' = A.h \cup \{(A.Op?, A.Res!\}) \]

where A.eout is in A.q.

A possible new concrete state can be related to the concrete state by the schema Abstract_OO'. Other abstract states are also possible (using different values of A.eout) that are related to different concrete states if either a different initial
quorum is used or a different original concrete state is used.

The above analysis brings out a difficulty of using an homomorphic type of approach when there is a conflict between the needs of modelling and the needs of verification. A historical record is an obvious choice of method for comparing the type of behaviour implicit in a specification and that in an implementation. However, in the case of an out of order priority queue presented here, the strict equivalence between historical records is not required. What is required is the same type of behaviour; elements can be dequeued from queues out of priority order, but can only be dequeued once. This can be expressed as a theorem and then proved that the implementation must produce histories that meet this theorem, but this approach is not pursued here.

6.4.4 Verification using a Proof Sketch of Equivalent Behaviour of an Out of Order Priority Queue

The aim of the proof sketch is to demonstrate that the replicated queue behaves equivalently to an one copy priority queue with regard to certain properties.

The proof is performed by constructing arbitrary queues. First an arbitrary deq element for the one copy queue is considered and it is demonstrated how an equivalent behaviour is possible by the replicated queues.

Let \( q \) be an arbitrary state of the one copy queue such that \( e_{out} \) is in the queue with a priority in the range:

\[
\text{lowest} \prec (\text{priority } e_{out}) \prec \text{highest}
\]

The next step is to construct a state for the replicated queue that can dequeue the same arbitrary element. This can be done if the initial quorum for the deq operation intersects the final quorum of the enq operation for the element \( e_{out} \), but does not intersect any of the final quorums of the enq operation where the priorities of the elements are higher than that of \( e_{out} \). Such a construction is possible because it does not violate any invariant.

The next stage in the proof is to demonstrate that, given an arbitrary state of the replicated queue, it is possible to construct a state of the one copy queue such that the same element is
dequeued.

Let $q$ be an arbitrary state of one of the queues in the system of replicated queues such that $e_{out}$ is in that queue.

It is now necessary to construct a state for the one copy queue that can dequeue the same element.

The one copy queue can dequeue an element in any order and the one copy queue contains all the elements that were enqueued (including $e_{out}$). Therefore, the dequeue operation on the one copy queue can dequeue any element in the queue, hence can dequeue $e_{out}$.

Multiple `deq` operations are not possible by the replicated queue because of the invariant $(Q2)$, which means that the composite history formed before every dequeue operation contains a record of all previous dequeue operations.

### 6.5 Degenerate Priority Queue

A degenerate priority queue is one that can dequeue elements in any order and any number of times. The only restriction is that the elements must have been enqueued before being dequeued. The study of a priority is split into the following sections:

- **6.5.1 Specification of a One Copy Priority Queue**
- **6.5.2 Implementation of a Replicated Degenerate Priority Queue**
- **6.5.3 Verification using Refinement Conditions of a Degenerate Priority Queue**
- **6.5.4 Verification using a Proof Sketch of Equivalent Behaviour of a Degenerate Priority Queue**

#### 6.5.1 Specification of a One Copy Degenerate Priority Queue

The main components of the specification of a one degenerate priority queue are shown in Figure 6.11.
The basic queue is once again represented as a bag in the schema DP_QUEUE below.

\[
\text{DP_QUEUE}
\]

\[
\begin{align*}
q & : \text{bag ELEMENT} \\
h & : \text{History} \\
q & = \text{Create_Bag } h
\end{align*}
\]

The schema DP.Enqueue below specifies the enqueue operation for a one copy degenerate priority queue.
The schema **DP_Enqueue** adds new elements contained in enqueue operations to the bag $q$.

---

**Proof Obligation for the Schema DP_Enqueue**

Adherence to the type rules of the Z notation is enforced by CADiZ.

The invariant of the schema **Queue** is based on the function *Create_Bag* being able to re-construct $q'$ from the updated history component $h'$. This is a basic requirement of *Create_Bag*, hence the invariant must be satisfied.

---

The schema **DP_Dequeue** below specifies the dequeue operation, in which an element can be removed from the queue without changing the queue.
Proof Obligation for the Schema DP_Dequeue

Adherence to the type rules of the Z notation is enforced by CADiZ.

The invariant of the schema Queue is based on the function Create_Bag being able to re-construct q' from the updated history component h'. This is a basic requirement of Create_Bag, hence the invariant must be satisfied.

The schema DP_Queue_Spec gives the specification of the required degenerate priority queue.

\[ \text{DP}_\text{Queue}_\text{Spec} \triangleq \text{DP}_\text{Enqueue} \lor \text{DP}_\text{Dequeue} \]

Precondition for the Schema DP_Queue_Spec

The schema Pre_DP_Queue_Spec_Simple below gives the simplified preconditions for...
the specification of the degenerate priority queue.

\[
\text{Pre\_DP\_Queue\_Spec\_Simple}
\]

\[
\begin{align*}
\text{DP\_Queue} \\
\text{Op? : Operation} \\
\text{First Op?} & = \text{enq} \\
\lor & \\
\text{First Op?} & = \text{deq}
\end{align*}
\]

The schema below gives the relationship between the simplified preconditions and the rudimentary preconditions.

Simplified_6_7 \equiv

\[\text{pre DP\_Queue\_Spec} \iff \text{Pre\_DP\_Queue\_Spec\_Simple}\]

The initial state for the degenerate priority queue specification is given by the schema below.

\[
\text{Initial\_DP\_Queue\_Spec}
\]

\[
\begin{align*}
\text{DP\_Queue'} \\
q' & = [] \\
h' & = \emptyset
\end{align*}
\]

\textit{Proof Obligation for the Initial State}

The values of the schema Initial\_DP\_Queue\_Spec are valid bindings of the schema
DPQueue because an empty bag and an empty set are possible values of the components $q'$ and $h'$.

6.5.2 Implementation of a Replicated Degenerate Priority Queue

The intersection invariants are not used in the implementation of a degenerate priority queue, see Figure 6.12.

*Figure 6.12 Interactions in the Implementation of a Degenerate Priority Queue*

\[
\text{DPQueue}\Imp \equiv \text{Site}_\text{Enq} \lor \text{Site}_\text{Deq}
\]

The schema $\text{DPQueue}\Imp$ specifies the implementation as the disjunction of the replicated site schemas used before, hence do not incur any additional proof obligations.
Preconditions for the Schema DP_Queue_Imp

The schema Pre_DP_Queue_Imp_Simple specifies the simplified preconditions for the implementation of a degenerate priority queue.

\[
\begin{align*}
\text{Pre_DP_Queue_Imp_Simple} \\
\text{Quorum_Set} \\
\text{Replication} \\
\text{Op? : Operation} \\
\text{First Op? = enq} \\
\lor \\
\text{First Op? = deq}
\end{align*}
\]

The predicate part in the simplified preconditions is the same as before, as is the declaration part except for not containing any intersection invariant. Instead the declaration part includes the schema Quorum_Set to provide the invariants for the sets Initial and Final.

\[
\text{Simplified}_6_8 \iff
\text{pre DP_Queue_Imp} \iff \text{Pre_DP_Queue_Imp}_\text{Simple}
\]

The above schema expresses the proof obligation necessary to ensure that the simplified preconditions are correct.

6.5.3 Verification using Refinement Conditions of a Degenerate Priority Queue

The schema DP_Verify gives the formal criterion for verifying that the implementation is equivalent to the specification.
DP_Verify \iff DP_{Queue\_Imp} \iff DP_{Queue\_Spec}

This section contains a proof sketch of the three data refinement conditions for this implementation of the degenerate priority queue and the next section contains a very brief sketch indicating the equivalent behaviour represented by the specification and implementation schemas.

**Abstraction Schema**

The abstraction schema for the refinement from the specification to the implementation of the degenerate priority queue is expressed as the schema \texttt{Abstract\_DP} below.

```
Abstract\_DP
  Replication
  DP\_Queue

q = \text{Create\_Q} \,(\text{Merge}\,(\text{ran}\,H\_R))\,\cdot\,Q
h = \text{Merge}\,(\text{ran}\,H\_R)
```

The schema takes the same form to that used before and suffers the same problems of ensuring all states are valid.

**Initialization Condition**

The initialization condition for the refinement process of the implementation of the degenerate priority queue is expressed as the schema \texttt{Initial\_Rule\_DP} below.

```
Initial\_Rule\_DP \equiv
Initial\_Queue\_Imp \Rightarrow
(\exists \, DP\_Queue' \cdot Initial\_DP\_Queue\_Spec \land Abstract\_DP')
```

The direct comparison of the initial states is obvious.
Applicability Condition

The applicability condition for the refinement of the implementation of the degenerate priority queue is expressed in the schema Applicability_Rule_DP below.

\[ \text{Applicability\_Rule\_DP} \equiv \]
\[ \text{pre DP\_Queue\_Spec} \land \text{Abstract\_DP} \Rightarrow \text{pre DP\_Queue\_Imp} \]

Since both specification and implementation schemas are total over all possible input values, and valid states are assumed, the proof is trivial.

Correctness Condition

The correctness condition for the refinement of the implementation of the degenerate priority queue is expressed by the schema Correctness_Rule_DP below.

\[ \text{Correctness\_Rule\_DP} \equiv \]
\[ \text{pre DP\_Queue\_Spec} \land \text{Abstract\_DP} \land \text{DP\_Queue\_Imp} \Rightarrow (\exists \text{DP\_Queue}' \land \text{Abstract\_DP}' \land \text{DP\_Queue\_Spec}) \]

Before reading the proof sketch below it may be helpful looking at the animation for the specification and implementation schemas for the degenerate priority queue contained in Appendix D.4.

The same style of correctness proof of the implementation is used for a degenerate priority queue as that for the other queues and is represented by Figure 6.13. Note that different new abstract states are possible using the concrete operation and abstraction schema to those possible from the abstract operation alone. This again invalidates the formal application of the verification process, but the process is still useful as a framework for arguing about the equivalent characteristics of the behaviours implied by the specification and implementation.
Figure 6.13 Data Refinement for Degenerate Priority Queues

From the schema Abstract_DP:

\[ A.q = (\text{Create}_Q (\text{Merge} (\text{ran} \ C.H_R))).Q \]

\[ A.h = \text{Merge} (\text{ran} \ C.H_R) \]

The new states are determined by case analysis of the values of \( \text{Op}^? \):

1. \( \text{First Op}^? = \text{enq} \)

From the schema Site_Enq:

\[ C.H_R' = C.H_R \uplus \{ s : \text{Site} \mid s \in f.q \ast \rightarrow C.H_R s \cup C.h \cup \{(C.Op^?, C.Res^!))\} \]

where

\[ C.h = \text{Merge} \{ s : \text{Site} \mid s \in i.q \ast \ C.H_R s \} \]

Since \( C.h \) is a sub history, it is also a subset of the history recorded by all the sites:

\[ C.h \subseteq \text{Merge} (\text{ran} \ C.H_R) \]
Merge (ran C.H_R') = Merge (ran C.H_R) \cup \{ (C.Op?, C.Res!) \}

From schema Site_Enq it follows that:

C.Res! = (okay, null)

From schema DP_Enqueue it follows that:

A.Res! = (okay, null)

and

A.h' = h \cup \{ (A.Op?, A.Res!) \}

The after abstract state component A.h' is related to the after concrete state component C.H_R' by the schema Abstract_DP'. Therefore proving the correctness of the implementation of the enqueue operation.

2 First Op? = deq

From schema Site_Deq:

\[ C.h = \text{Merge}\ \{ s : \text{Site} \mid s \in iq \cdot C.H_R\ s\} \]

\[ C.q = \text{Create}_Q \ C.h \]

There are two cases of C.h to consider.

2.1 Empty_Q C.q

From schema Site_Deq:

C.Res! = (error, null) and C.H_R' = C.H_R

Schema DP_Dequeue with A.q = \emptyset results in:

A.Res! = (error, null) and A.h' = A.h

The new abstract state is related to the new concrete state by the schema Abstract_DP'.

However, other new concrete states are possible for A.q \neq \emptyset, these states can be related to other concrete states by selecting different initial quorums.
2.2 Not_empty_Q C.q

From schema Site_Deq:

\[ C.\text{Res}\! = (\text{okay}, C.e\text{out}) \]

and

\[ C.H.\_R\! = C.H.\_R \oplus \{s : \text{Site } | s \in f_q \cdot s \mapsto C.H.\_R \cup C.h \cup \{(C.\text{Op}\?, C.\text{Res}\! )\}\} \]

where

\[ (\text{priority } C.e\text{out}) > (\text{priority } e) \]

for all e in C.q.

From schema Dequeue with A.q ≠ []:

\[ A.\text{Res}\! = (\text{okay}, A.e\text{out}) \]

and

\[ A.h\! = A.h \cup \{(A.\text{Op}\?, A.\text{Res}\! )\} \]

where e is in A.q.

Therefore, a possible new concrete state can be related to the concrete state by the schema Abstract_DP' provided A.e\text{out} is in A.q.

It is possible for e\text{out} to have been dequeued previously from both the abstract one copy queue and a subset of the concrete replicated queues. Such multiple dequeue operations are possible because the abstract dequeue operation does not necessarily update the q component.
6.5.4 Verification using a Proof Sketch of Equivalent Behaviour of a Degenerate Priority Queue

The proof follows the same lines as for the proof of out of order priority queue by constructing arbitrary queues for both the replicated and one copy queues. The behaviour of the one copy queue being able to dequeue an element more than once can also be the behaviour of the replicated queue. In the case of the replicated queues, this ability is a consequence of the initial and final quorums for the deq operation not intersecting. By using a totally partitioned network, any behaviour of the one copy degenerate queue is possible of the replicated queue by using the appropriate sites for the deq operation.

A similar argument can be presented for any behaviour of the replicated queue being possible by the one copy queue.
6.6 Lattice Structure of Behaviours

The behaviours possible by the four queues form a lattice structure as shown in Figure 6.14.

Figure 6.14 Lattice Structure for Priority Queues

The lattice structure can be seen by considering the following sub history of the single queue:

\[ \text{enq}(x) \text{ enq}(y) \text{ deq}(a) \text{ deq}(b) \]

Each of the four queues have the restrictions:

**Degenerate**

The elements \( a \) and \( b \) must be chosen from the elements \( x \) and \( y \). That is \( a \in \{ x, y \} \) and \( b \in \{ x, y \} \). Note that \( b \) can be the same as \( a \).

**Multiple priority**

The dequeue operations must be in priority order. That is \( a \in \{ x, y \} \), \( b \in \{ x, y \} \), and \( (\text{priority } a) = (\max (\text{priority } x), (\text{priority } y)) \), assuming that the queue is empty at the start of the history. Note that \( b \) can be the same as \( a \).
Out of order

The dequeue operations must dequeue different elements. That is, \( a \neq b, a \in \{x, y\} \), and \( b \in \{x, y\} \).

Priority

The dequeue operations must dequeue different elements and be in priority order. That is \( a \neq b, a \in \{x, y\}, b \in \{x, y\}, (\text{priority } a) = (\max (\text{priority } x), (\text{priority } y)), \) and \( (\text{priority } b) = (\min (\text{priority } x), (\text{priority } y)) \), assuming that the queue is empty at the start of the history.

A partially ordered set must have the following three properties:

Reflexitivity \( \alpha \leq \alpha \)

Antisymmetry \( \alpha \leq \beta \) and \( \beta \leq \alpha \) imply that \( \alpha = \beta \)

Transitivity \( \alpha \leq \beta \) and \( \beta \leq \gamma \) imply that \( \alpha \leq \gamma \)

A lattice has the above three properties, plus the property that a least upper bound (supremum) and a greatest lower bound (infimum) exists for all \( a \) and \( b \) in the lattice [Grät71].

In this case the members of the lattice are members of queues and the ordering relation is contains (or subset). It is straight forward to see that:

\[
\text{history of priority queue} \subset \text{history of multiple priority queue} \subset \text{history of degenerate queue}
\]

and

\[
\text{history of priority queue} \subset \text{history of out of order queue} \subset \text{history of degenerate queue}
\]

The histories have all the properties of a lattice which has the structure shown in Figure 6.14, where the members of the out of order queue and the multiple priority queue are incomparable because the behaviour of one does not contain all the behaviours of the other.
6.7 Summary

The four specifications and four implementations in this chapter are all complete descriptions in the sense that all the information required to analyse their behaviour is made explicit. The ability to analyse the descriptions in terms of input values is used in the proof sketches of the verifications in this chapter.

The three data refinement conditions are applied to each of the four types of queues contained in this chapter. Some difficulties are encountered with the correctness condition of the dequeue operation in all the queues except for the priority queue. However, the correctness condition is verified for the enqueue operation for all the four queues.

Applying the correctness condition for data refinement did uncover some of the difficulties with using the refinement conditions when a strict mathematical relationship is not required between the specification and the implementation. One of the difficulties in performing the data refinement proofs is deciding on a suitable abstraction schema. The approach adopted in this chapter is to specify the minimum amount of detail in the abstraction schema so not to impose any constraints on the implementation that are not contained in its description.

Abstraction (or retrieve) schemas are described as the relationships between concrete and abstract states [Spiv89A, Dilh90, Pott91]. The references only provide very simple examples. In the context of VDM [Jones86], the retrieve function must be total over all abstract states and an adequacy proof obligation of the following form is used:

\[ \forall a \in \text{Abstract State} \cdot \exists c \in \text{Concrete State} \cdot (\text{retr } c) = a \]

\[ \text{retr : Concrete State } \rightarrow \text{Abstract State} \]

Simple statements about which concrete and abstract states are valid out of all possible state spaces for the data types are not be feasible for schemas written at the level of abstraction addressed in this chapter. The operations are defined to be total over all possible input values, but it is assumed that the values of the data components are consistent with each other, i.e. the only state changes are brought about by using the operations defined by the Z schemas.

The behaviours implicit in the specification and implementation of each of the queues for the dequeue operation are summarised in Table 6.2. From this table it is clear that the imple-
Implementations describe behaviours that have the same basic characteristics of each of the behaviours implicit in the specifications, but there are some differences in the degrees of freedom.

**Table 6.2 Analysis of Queue Implementations**

<table>
<thead>
<tr>
<th>Type of Queue</th>
<th>Specification</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>Always returns the element with the highest priority in the one copy queue. The element dequeued is determined completely by the current state of the queue for a total ordering relation ‘&gt;’.</td>
<td>Always returns the element with the highest priority that is enqueued at any copy of a replicated queue. The element dequeued is determined completely by the current state of the replicated queues for a total ordering relation ‘&gt;’.</td>
</tr>
<tr>
<td>Multiple Priority</td>
<td>Always dequeues an element that has either the highest priority of any element waiting to be dequeued or an element that has been dequeued before and that has a priority greater than any element waiting to be dequeued. The element dequeued is not determined completely by the current state of the queue.</td>
<td>Always dequeues the element with the highest priority that is held by any copy of a queue, however as not all copies are necessary updated, the same element can be dequeued more than once. The number of times an element can be dequeued is related to the final quorums used in the implementation. The element dequeued is determined by the current state of the replicated queues and the initial quorum used for the operation.</td>
</tr>
<tr>
<td>Priority Type</td>
<td>Behavior</td>
<td>Elements are not necessarily dequeued in priority order. However, elements cannot be dequeued more than once because all initial quorums are notified of any dequeue operations. A dequeue operation can fail even when there are elements waiting to be dequeued at sites outside the initial quorum used by the implementation. The element dequeued is determined by the current state of the replicated queues and the initial quorum used for the operation.</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Out of Order Priority</td>
<td>Always dequeues an element if one is waiting to be dequeued, but elements are dequeued in any order. The element dequeued is not determined completely by the current state of the queue.</td>
<td></td>
</tr>
<tr>
<td>Degenerate Priority</td>
<td>Always dequeues an element if one is waiting to be dequeued, but elements are not dequeued in priority order and elements can be dequeued any number of times. The element dequeued is not determined completely by the current state of the queue.</td>
<td>Elements are not necessarily dequeued in priority order and elements can be dequeued more than once. The number of times an element can be dequeued is related to the number of sites and the final quorum used in the implementation. A dequeue operation can fail even when there are elements waiting to be dequeued at sites outside the initial quorum used by the implementation. The element dequeued is determined by the current state of the replicated queues and the initial quorum used for the operation.</td>
</tr>
</tbody>
</table>
Applying the three data refinement conditions for implementations and specifications has the advantage of standardising the verification procedure. Without a framework for the verification, it is very easy to miss important checks, such as ensuring that there exist states that satisfy the predicates in the schemas.

A more flexible approach is to adopt proof sketches to verify that the dequeue behaviour defined by the specification is also defined by the implementation, and vice versa. A brief and informal proof sketch is given for each of the four queues analysed in this chapter. A disadvantage of the proof sketches of the behaviour of the implementation with respect to the specification is that they encompass only the dynamic aspect of the behaviour. They do not address the relationship between the abstract and concrete states for the operations to take place.

In addition to verifying implementations correct with specification, proof sketches are used in this chapter to discharge proof obligations in a similar manner to that described in Section 3.8.2.
Chapter 7
Conclusions

This thesis has shown how proof sketches can be usefully applied in the design of complex systems. The systems are described at a high level of abstraction using the Z notation, and proof sketches are used to verify complete descriptions of the systems.

7.1 Complete Descriptions

Complete descriptions in the Z notation contain all the information necessary to determine the behaviour of a system. The response of the system to any condition is determined without recourse to information outside the formal descriptions. In all three studies in this thesis, the information used by the operation schemas identifies the operation required. This means that when several operation schemas are connected and simplified, the resultant predicate is still separated into segments that refer to each type of operation. Identifiable segments in schema expansions assist in the formulation of proof sketches and reduce the reliance on either informal text or assumed knowledge.

Using a unified approach with all operation schemas in a system permits a description of the system to be represented as single schema equation that is a comprehensive model of the system. This approach is taken with all the implementations in this thesis, for example Network_Imp (Section 3.3.7), DBS_Imp (Section 5.5) and Queue_Imp (Section 6.2.2). Proof sketches based on the complete descriptions of implementations are able to identify the
relevant parts of the implementation by using the operations associated with the input information. For instance, the proof sketch for Theorem 3.5 in Section 3.4, and the proof sketches in Sections 5.6.2 and 6.2.3 illustrate the identification of predicates using the name of operations.

Preconditions determine the conditions under which operations are performed. The preconditions of a system are calculated by connecting the preconditions of the operations together, thereby forming a complete statement of the conditions in which the system will respond. If each operation schema includes a means of identifying the operation, then this information is retained by the preconditions and is available in the complete statement of preconditions for the system.

Simulation is an important technique in a design process. A single complete description of a system means that simulations can be performed based on the input information, without requiring additional information to determine which operation is to be performed by the system.

This thesis has demonstrated the length of descriptions in the Z notation required to express specifications and implementations of moderately complex systems. To be meaningful, the descriptions must include both formal components expressed in the Z notation and informal components to give the intended interpretations of the schemas. This inevitably results in descriptions occupying many pages of text and gives rise to problems of organising the information.

7.2 Proof Sketches

A proof sketch is an outline of a proof demonstration. There is a wide variation in the amount of detail in a proof sketch. For example, a single sentence for proof obligations for free data types and several pages for the representation of serialization graphs in Chapter 5. The proof sketches in this thesis range from a few lines to several pages, but they are all intentionally liberal in their approach to what constitutes a proof. The emphasis is on usefulness rather than formality.

One of the problems with proof sketches is that they can contain mistakes. Even well established proofs have been found to contain errors after being accepted for many years.
Proof demonstrations are less likely to contain errors because the steps can be verified formally, however, proof demonstrations are still subject to the limitation of the specification being correct, see Section 1.1.3.

7.2.1 Success of Proof Sketches

This thesis has shown that proof sketches can fulfil the role of discharging basic proof obligations. Examples of proof obligations are given in Chapters 3, 5 and 6 for given sets, free data types, initial states and the maintenance of invariants for operation schemas. Discharging proof obligations give a high level of confidence about the validity of descriptions in the Z notation. Proof demonstration of the basic proof obligations in this thesis would be difficult, even with automated tools.

Difficulties in constructing proof sketches are indications of potential problems in verification. Any difficulty in constructing proof sketches for any aspect of a description will also be present in a much magnified form in proof demonstrations.

Proof sketches have also been demonstrated in this thesis to be successful at verifying implementations with respect to specifications. The specifications are given in several different forms: in terms of simple predicate logic terms representing properties (Chapter 3); complex schema representation of a mathematical property (Chapter 5); and models of required behaviour (Chapter 6).

Mathematical theories can be introduced into proof sketches to assist in the completion of their tasks. Chapter 5 incorporates a mathematical theory in a verification process by expressing the theory in terms of the Z notation. This helps understanding both the theory and the actual verification. Section 5.6.2 uses a proof sketch that refers to a representation in the Z notation of a mathematical theory based on replicated data serialization graphs. Section 5.6.1 uses a proof sketch that does not refer to a formal expression of the same mathematical theory, but the understanding of the theory derived from the expression in the Z notation helped to construct the proof sketch.

Proof sketches in this thesis, in addition to discovering a few slips, revealed assumptions about the size of given sets (Theorem 3.3 in Section 3.4), differences in an assumption in a specification and an implementation (type of precedence relations in Section 5.6.2), and dif-
ferences in behaviour indicated by specifications and implementations (similar behaviour between queues revealed by proof sketches is summarised in Table 6.2).

Proof sketches provide an opportunity of questioning the validity of implementation, thereby providing a good review process. Diagrams also provide a good opportunity of examining the descriptions written in the Z notation. In particular, the possible conflicts between operation schemas that use the same state components are not obvious from the descriptions in the Z notation. Showing interactions in the form of diagrams clearly reveal possible problems. Most of problems can be resolved by revising the preconditions and schema definitions.

In addition to helping to verify descriptions, diagrams are used to enhance the descriptions in this thesis by:

1. cross referencing schemas by showing the page numbers of their definitions, this is useful in the large descriptions in all three studies
2. indicating the structure of schemas that include other schemas, e.g. Figure 5.1
3. indicating the interaction between schemas, e.g. Figure 3.1 and 3.2.

7.2.2 Proof Framework

The approach taken in this thesis has been to construct complete descriptions of systems in which the input information is based on the form of information that is expected to be used in the final systems.

The following list of five areas describes how proof sketches are used in this thesis and the list provides a good framework for using the Z notation in a design process:

1. Check consistency of all given sets and free type definitions. Confirm that standard sets can replace the given sets and lead to a consistent model, hence verifying that there is at least one feasible solution. Ensure that any recursive branches in free type definitions do not cause problems.
2. Check data invariants for each operation schema. Breaking type rules is common and is difficult to check manually. CADiZ is used in this thesis to type check all schemas. Functional properties can be violated by some operations that still obey the
basic type rules of the Z notation and these properties require additional checks. The functional override operator used in this thesis reduces the checks to ensuring that an overriding function is a partial function.

3 Check that the system has at least one valid state. Each component of the state of the systems has an initial state schema that defines a valid initial state.

4 Calculate and simplify the preconditions for operation schemas. Preconditions are important for checking that: the operations are performed when they are intended to be performed; the operations are mutually exclusive; the complete state space is covered; the preconditions are necessary for the refinement conditions. Appendices B.2, C.6 and D.6 contain examples of simplifying preconditions.

5 Verify that the implementation is correct with respect to a specification. Refinement conditions give a good starting point for verification, but the requirements can be too strict at high levels of abstraction. However, the three refinement conditions give a good indication of what areas to examine, i.e. the applicability, the correctness and the initialization of schemas.

Proof sketches in this thesis were supported by computer tools using by:

1 Type checking by CADiZ. Enforcing the type rules for the Z notation under covered several simple errors.

2 Type setting by CADiZ. A clear layout helped manual checking of schema definitions.

3 Expansion of schemas provided by CADiZ. This was of limited use because CADiZ does not expand schema definition that used the $\Xi$ operator, but does expand all predicates included in schema definitions which is sometimes unnecessary.

4 Simplification of preconditions. This was facilitated by using cut and paste type of commands on word processor.

7.2.3 Difficulties with Proof Sketches

Complexity of descriptions makes proof sketches strictly based on the information contained in the schemas difficult. However, convincing proof sketches can be constructed using an
interpretation of the information in schemas, such that much of the detail is hidden, see Section 5.6 for two proof sketches that ignore large parts of the details in the formal descriptions.

If part of the descriptions are defined weakly, then verification of proof obligation will also appear weak. Chapter 6 includes functions (Create_Bag and Create_Q) that linked queues and histories of operations. The two functions are primarily for reasons of verification, hence are not expected to be implemented and do not form an important part of the specifications and implementations when viewed in isolation of each other. The connection between histories and queues only become important when the implementations are verified with respect to the specifications.

The Z notation does not have any formal mechanism for separating preconditions and postconditions in the predicate part of schemas. In this thesis the majority of the schemas are written so that the preconditions, involving the before variables, are separate from the postconditions, involving after variables. This separation helps to simplify preconditions, but does sometimes result in extra variables being used therefore the style is not always followed, for instance see Section 5.4.3 for the schema Trans_Man_Commit.

7.3 Future Work

This thesis outlines five areas in which proof sketches can play useful roles in the verification of implementations at an early design stage. These five areas provide some guidance about the proof obligations that should be discharged. Additional work is required to create a body of experience that will provide a range of techniques of using proof sketches with the Z notation for different applications.

More flexible computer tools are required to assist the construction of proof sketches. For instance, the ability to expand schema definitions partially, in a manner similar to that given in the simplifications in Appendix C.6, would be useful. The expansions produced by computer tools should also be in a form that permitted further manual modification.

Automated methods to assist proof such as B-Tool [Btool91] are not useful because of the lack of built in knowledge. Automated methods will be more productive if they are aimed at routine tasks such as simplification of preconditions.
Abstraction schemas are required to link the data types at one level of abstraction to the data types used at a lower level of abstraction. One of the difficulties in creating abstraction schemas is devising the relation between the two data types without imposing restrictions that are not defined in the descriptions, see Section 6.7. More research is needed into the requirements of abstraction schemas and the checks that must be applied to verify that the abstraction schemas are well defined.

A large number of schemas are required to describe complex systems which makes cross referencing awkward because of the number of pages that have to be reference. Diagrams are shown in this thesis to help by providing summaries of the relationships between schemas. Greater help may be possible by computer tools, for instance creating a window environment where several schemas can be in view simultaneously.
Glossary

\( \mathbb{N} \) natural number
\( \mathbb{N}^* \) positive natural number
\( \mathbb{Z} \) integers
\( \mathbb{R} \) real numbers
\( \wedge \) conjunction
\( \vee \) disjunction
\( \Rightarrow \) implies
\( \Leftarrow \) implied by
\( \iff \) if and only if (equivalence)
\( \neg \) negation
\( \equiv \) equivalence
\( \forall \) for all (universal quantification)
\( \exists \) there exits (existential quantification)
\( \cdot \) dot (such that)
\( \mid \) vertical bar (such that)
\( \times \) cartesian product
\( \mapsto \) maplet
\( \leftrightarrow \) relation
\( \rightarrow \) total function
partial function

partial finite function

injection

partial injection

partial finite injection

surjection

partial surjection

partial finite surjection

bijection

power set

domain restriction

domain subtraction

range restriction

range subtraction

function overriding

forward relational composition

first element in a sequence

last element in a sequence

all the elements in a sequence except the last

all the elements in a sequence except the first

the domain of a function

the range of a function

sequence

Set of finite non-empty sequences

sequence length

open sequence bracket

close sequence bracket

sequence restriction

sequence range restriction
\emptyset \quad \text{empty set}
\in \quad \text{set membership}
\not\in \quad \text{not set membership}
\subset \quad \text{strict set inclusion}
\subseteq \quad \text{set inclusion}
\cap \quad \text{set intersection (set and)}
\cup \quad \text{set union (set or)}
\bigcap \quad \text{distributed set intersection (distributed and)}
\bigcup \quad \text{distributed set union (distributed or)}
\supset \quad \text{strict super set}
\supseteq \quad \text{super set}
\not\subset \quad \text{not set inclusion}
\not\supset \quad \text{not strict set inclusion}
\cup \quad \text{union bag}
\llbracket \quad \text{open bag bracket}
\rrbracket \quad \text{close bag bracket}
\neq \quad \text{not equals}
\leq \quad \text{less than or equal to}
\geq \quad \text{greater than or equal to}
= \quad \text{equal to}
\hat{\text{prime}}
\mid \quad \text{pipe line}
\vdash \quad \text{theorem}
\equiv \quad \text{syntactically equivalent to}
\langle \quad \text{open binding bracket}
\rangle \quad \text{close binding bracket}
\setminus \quad \text{Set difference}
\sim \quad \text{Inverse}
Bibliography

Abbad89  Maintaining Availability in Partitioned Replicated Databases
Abbadi A. el and S. Toueg

Agu87   A Plausibility-Driven Approach to Computer Architecture Design
Agüero U. and S. Dasgupta

Arth92  On Free Type Definitions in Z
Arthan R.
Edited by J. E. Nicholls
Aylor86 VHDL – Feature Description and Analysis

Aylor J. H.

IEEE Design and Test, April, 1986, Pages 17 - 27.

Ben90 A Message Passing System: An Example of Combining CSP and Z

Benjamin M.


Edited by J. E. Nicholls

Springer-Verlag, 1990, Pages 221 - 228.

Bern81 Concurrency Control in Distributed Database Systems

Bernstein P. A. and N. Goodman

Computer Surveys, Volume 13, Number 2, June 1981, Pages 185 - 221.

Bern83 The Failure and Recovery Problem for Replicated Databases

Bernstein P. A. and N. Goodman


Bern87 Concurrency Control and Recovery in Database Systems

Bernstein P. A., V. Hadzilcos, N. Goodman


Brien91 A Semantics for Z

Brien S. M., Gardiner P. H. B., Lupton P. J., Woodcock J. C. P.

Btool91  B-Tool Reference Manual
Abral J. R.
Version 1.1, 1991
BP International Limited

CADiZ91  The CADiZ Tutorial

Cam88  Formal Verification of Hardware Correctness: Introduction and Survey of Current Research
Camurati P. and P. Prinetto

Carr92  ZOOM Workshop Report
Carrington D.
Edited by J. E. Nicholls

Cohen86  The Specification of Complex Systems
Cohen B, W. T. Harwood, M. I. Jackson
Addison-Wesley, 1986

Cohn89  Correctness Properties of the Viper Block Model: The Second Level
Cohn A.
Current Trends in Hardware Verification and Automated Theorem Proving
G. Birtwistle and P. A. Subrahmanyam (editors)
Springer-Verlag, 1989.
Cra91  Formal Specification of Advanced AI Architecture
Craig I. D.

Dan87  Design Criterias and Formal Description Techniques
Daniels M.

Dasg87 On the Plausibility of Architecture Designs
Dasgupta S. and U. Agüero

Dillr90 Z: An Introduction to Formal Methods
Diller A.

Dud83 Types of Hardware Description
Dudani S. and E. Stabler

Duke R., I. Hayes, P. King, G. Rose
Duke89  Temporal Logic and Z Specifications
Duke R. and G. Smith

Duke90  Towards a Semantics for Object-Z
Duke D. and R. Duke
VDM'90 VDM and Z Formal Methods in Software Development
Edited by D. Bjørner, C. A. R. Hoare and H. Langmaack

Duke92  Enhancing the Structure of Z Specifications
Edited by J. E. Nicholls

Fetz88  Program Verification: The Very Idea
Fetzer J. M.
Communications of the ACM, September 1988, Volume 31, Number 9, Pages 1048 - 1063.

Gibb88  What are Formal Methods?
Gibbons P. F.
Information and Software Engineering, Volume 30, Number 3, April 1988, Pages 131 - 137.
Gord85  HOL: A Machine Oriented Formulation of Higher-Order Logic
Gordon M.
University of Cambridge Computer Laboratory Technical Report
Number 68, 1985.

Grät71  Lattice Theory: First Concepts and Distributed Lattices
Grätzer G.

Hall90  Using Z as a Specification Calculus for Object Oriented Systems
Hall A.
VDM'90 VDM and Z Formal Methods in Software Development
Edited by D. Bjørner, C. A. R. Hoare and H. Langmaack
Lecture Notes in Computer Science, Volume 428, Springer-Verlag,

Hayes87  Specification Case Studies

Hayes91  Interpretation of Z Schema Operators
Hayes I.
Z User Workshop, Proceedings of the Fifth Annual Z User Meeting
Edited by J. E. Nicholls.

Herl86  A Quorum-Consensus Replication Method for Abstract Data Types
Herlihy M.
ACM Transactions on Computer Systems, Volume 4, Number 1,
February 1986, Pages 32 - 53.
Herlihy M. P. and J. M. Wing

IEEE Transactions on Parallel and Distributed Systems, Volume 2, Number 1, January 1991, Pages 93 - 104.

Jones C. B


Jordan D., J. McDermid and I. Toyn


J. E. Nicholls (editor)


King S.

VDM'90 VDM and Z Formal Methods in Software Development

Edited by D. Bjørner, C. A. R. Hoare and H. Langmaack


Marsh J. D., L. F. Saunders

IEEE Design and Test, April, 1986, Pages 54 - 65.
Millo79  Social Processes and Proofs of Theorems and Programs
Millo De R. A., R. J. Lipton, A. Perlis

Milne86  Towards Verifiably Correct VLSI Design
Milne G. J.

Milne88  The Fusion of Hardware Design and Verification
Milne G. J. (editor)

Morg83  Specification of a Communication System
Morgan C.

Mori85  The Design Rationale of Ella, a Hardware Design and Description Language
Morison J. D., N. E. Peeling, T. L. Thorp
Nich92  Domains of Application for Formal Methods
Nicholls J. E.
Edited by J. E. Nicholls

Polac92 Structured Analysis - A Draft Method for Writing Z Specifications
Polack F., M. Whiston, P. Hitchcock
Edited by J. E. Nicholls

Pottr91 An Introduction to Formal Specification and Z
Potter B., J. Sinclair, D. Till

Pyle89 Formality does not ensure Reliability
Pyle I.C.
Proceedings of Reliability 89, Volume 1, Pages 3A/6/1 to 3A/6/10.

ReedD91 Z in Database Systems
Reed D.
Saal92 Z and Eves
Saaltink M.


Edited by J. E. Nicholls


Sch90 Objected-Oriented Process Specification
Schuman S. A., D. H. Pitt and P. J. Byers


Semm91 Using Yourdon and Z: An Approach to Formal Specification
Semmens L. and P. Allen


Edited by J. E. Nicholls.


Smit87 The MoDL Hardware Design System
Smit J., Beijnum B. J. F. van, Gerez S. H., Mulder R. J., Spaanenburg L.


Spivey J. M.

Spiv89A  The Z Notation: A Reference Manual
         Spivey J. M.

Spiv89B  An Introduction to Z and Formal Specifications
         Spivey J. M.

Swat92  Extending the Useful Application Domain for Formal Methods
         Swatman P. A., D. Fowler, C. Y. M. Gan
         Edited by J. E. Nicholls

Wing90  A Specifier's Introduction to Formal Methods
         Wing J. M.

         Woodcock J. and M. Loomes.

Wood89A  Calculating Properties of Z Specifications
         Woodcock J. C. P.
Wood89B  Structuring Specifications in Z
Woodcock J. C. P.

Zave92  Techniques for Partial Specification and Specification of Switching Systems
Zave P., M. Jackson
Edited by J. E. Nicholls

ZipBS91  Z Base Standard
Brien S. M., P. J. Lupton, J. E. Nicholls, I. H. Sørensen

ZipCt91  Z Tools Catalogue
Parker C. E.

ZipOb91  Comparative Study of Object Orientation in Z
Stepney S.