System Modelling and Design

Modelling a Queue

Towards Implementation

Revision: March 28 2011

Ken Robinson

School of Computer Science & Engineering
The University of New South Wales, Sydney Australia

March 28, 2011

© Ken Robinson 2005-2010

mailto::k.robinson@unsw.edu.au

A Queue Development

This model will involve data refinement towards what could be called a *pointer implementation*.

We will specify a simple *Queue* machine that models a queue manager. A *queue*, of course, is a *first in first out* structure.

The items in the queue are represented by the set *ITEM* and it should be noted that we allow the same item to appear more than once in the queue. We are never concerned about the identity of the items, we are only concerned with the queue tokens that are taken from the set *QUEUE*. The queue tokens are unique.

A Queue Development

This model will involve data refinement towards what could be called a *pointer implementation*.

We will specify a simple *Queue* machine that models a queue manager. A *queue*, of course, is a *first in first out* structure.

The items in the queue are represented by the set *ITEM* and it should be noted that we allow the same item to appear more than once in the queue. We are never concerned about the identity of the items, we are only concerned with the queue tokens that are taken from the set *QUEUE*. The queue tokens are unique.

A Queue Development

This model will involve data refinement towards what could be called a *pointer implementation*.

We will specify a simple *Queue* machine that models a queue manager. A *queue*, of course, is a *first in first out* structure.

The items in the queue are represented by the set *ITEM* and it should be noted that we allow the same item to appear more than once in the queue. We are never concerned about the identity of the items, we are only concerned with the queue tokens that are taken from the set *QUEUE*. The queue tokens are unique.

Queue Events

The machine has the following events:

Enqueue(*item*) an event that places an item on the end of the queue. The event creates a unique queue identifier for this item. A unique item identifier is also generated for the item that is queued. A queue can contain multiple instances of the same item value.

Dequeue an event that removes the item that is at the head of the queue.

Unqueue(*qid*) removes the item from the queue identified by the queue identifier, *qid*. This is not a strict queue event; it is used to remove an item outside the queue discipline.

Queue Events

The machine has the following events:

Enqueue(item) an event that places an item on the end of the queue. The event creates a unique queue identifier for this item. A unique item identifier is also generated for the item that is queued. A queue can contain multiple instances of the same item value.

Dequeue an event that removes the item that is at the head of the queue.

Unqueue(qid) removes the item from the queue identified by the queue identifier, *qid*. This is not a strict queue event; it is used to remove an item outside the queue discipline.

Queue Events

The machine has the following events:

Enqueue(*item*) an event that places an item on the end of the queue. The event creates a unique queue identifier for this item. A unique item identifier is also generated for the item that is queued. A queue can contain multiple instances of the same item value.

Dequeue an event that removes the item that is at the head of the queue.

Unqueue(*qid*) removes the item from the queue identified by the queue identifier, *qid*. This is not a strict queue event; it is used to remove an item outside the queue discipline.

Queue Events

The machine has the following events:

Enqueue(*item*) an event that places an item on the end of the queue. The event creates a unique queue identifier for this item. A unique item identifier is also generated for the item that is queued. A queue can contain multiple instances of the same item value.

Dequeue an event that removes the item that is at the head of the queue.

Unqueue(*qid*) removes the item from the queue identified by the queue identifier, *qid*. This is not a strict queue event; it is used to remove an item outside the queue discipline.

Modelling of queue

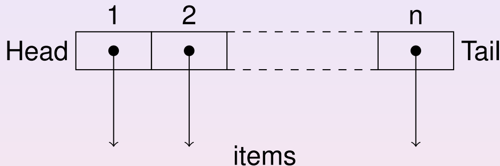
The queue is first modelled by a sequence with the head of the queue being the first element of the sequence; the end of the queue is the last element of the sequence.

Because a sequence is a monolithic structure the coherence of the queue structure is trivially guaranteed.

The Unqueue event requires unique identification of items in the queue. Since the position of an item in the queue changes as the queue changes, the initial position of an item in the queue cannot be used to uniquely identify the item. For that reason the elements of the queue will be unique identifiers, *queuetokens*. A function, *queueitem*, maps from *queuetokens* to the actual items.

Modelling of queue

The queue is first modelled by a sequence with the head of the queue being the first element of the sequence; the end of the queue is the last element of the sequence.

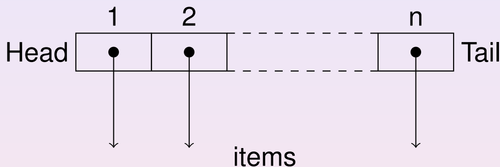


Because a sequence is a monolithic structure the coherence of the queue structure is trivially guaranteed.

The Unqueue event requires unique identification of items in the queue. Since the position of an item in the queue changes as the queue changes, the initial position of an item in the queue cannot be used to uniquely identify the item. For that reason the elements of the queue will be unique identifiers, *queuetokens*. A function, *queueitem*, maps from *queuetokens* to the actual items.

Modelling of queue

The queue is first modelled by a sequence with the head of the queue being the first element of the sequence; the end of the queue is the last element of the sequence.



Because a sequence is a monolithic structure the coherence of the queue structure is trivially guaranteed.

The Unqueue event requires unique identification of items in the queue. Since the position of an item in the queue changes as the queue changes, the initial position of an item in the queue cannot be used to uniquely identify the item. For that reason the elements of the queue will be unique identifiers, *queuetokens*. A function, *queueitem*, maps from *queuetokens* to the actual items.

Modelling of queue

The queue is first modelled by a sequence with the head of the queue being the first element of the sequence; the end of the queue is the last element of the sequence.

Because a sequence is a monolithic structure the coherence of the queue structure is trivially guaranteed.

The Unqueue event requires unique identification of items in the queue. Since the position of an item in the queue changes as the queue changes, the initial position of an item in the queue cannot be used to uniquely identify the item. For that reason the elements of the queue will be unique identifiers, *queuetokens*. A function, *queueitem*, maps from *queuetokens* to the actual items.

Context Machines

Since EventB does not have a sequence type we need to define our own sequence type and accompanying functions for managing queues represented as sequences.

QueueContext contains *QUEUE*, which models the set of all injective queues of *TOKEN*.

QueueContext also contains the carrier set *ITEM*, to represent the items that are contained in a queue.

CONTEXT QueueContext

SETS

TOKEN

ITEM

CONSTANTS

MAXQUEUE

QUEUE

AXIOMS

axm1: $finite(TOKEN)$

axm2: $finite(ITEM)$

axm3: $MAXQUEUE \in \mathbb{N}_1$

axm4: $QUEUE \subseteq 1 \,..\, MAXQUEUE \rightarrowtail\!\!\!\!\!\!\!\nrightarrow TOKEN$

axm5: $finite(QUEUE)$

axm6: $QUEUE = \{q | q \in 1 \,..\, card(q) \rightarrowtail TOKEN\}$

thm1: $\forall q \cdot q \in QUEUE \Rightarrow dom(q) = 1 \,..\, card(q)$

thm1: $\varnothing \in QUEUE$

END

The QueueA machine I

The first model we build uses rather ad-hoc queue operations.

MACHINE QueueA

SEES QueueContext

VARIABLES

queuetokens
: tokens for currently queued items

queue
: the queue of tokens

queueitems
: a function that binds the item associated with a token

qsize
: current size of queue

The QueueA machine II

INVARIANTS

$inv1$: $queuetokens \subseteq TOKEN$

$inv2$: $queue \in QUEUE$

$inv3$: $qsize \in \mathbb{N}$

$inv4$: $queue \in 1 \,..\, qsize \rightarrowtail\!\!\!\!\rightarrow queuetokens$

$inv5$: $\forall i, j \cdot i \in dom(queue) \land j \in dom(queue) \land i \neq j \Rightarrow queue(i) \neq queue(j)$

$inv6$: $queuetokens = ran(queue)$

$inv7$: $queueitems \in queuetokens \rightarrow ITEM$

$inv8$: $card(queue) = qsize$

$inv9$: $queue^{-1} \in queuetokens \rightarrowtail\!\!\!\!\rightarrow 1 \,..\, qsize$

$thm1$: $queuetokens \neq \varnothing \Leftrightarrow qsize \neq 0$

The QueueA machine III

EVENTS

Initialisation

begin

act1: $queuetokens := \varnothing$

act2: $queue := \varnothing$

act3: $qsize := 0$

act4: $queueitems := \varnothing$

end

The QueueA machine IV

Event $Enqueue \mathrel{\widehat{=}}$

any

$item$

qid

when

$grd1$: $item \in ITEM$

$grd2$: $qid \in TOKEN \setminus queuetokens$

then

$act1$: $queuetokens := queuetokens \cup \{qid\}$

$act2$: $queue(qsize + 1) := qid$

$act3$: $queueitems(qid) := item$

$act4$: $qsize := qsize + 1$

end

The QueueA machine V

Event $Dequeue \mathrel{\widehat{=}}$

when

$grd1$: $qsize \neq 0$

then

$act1$:

$$queue :| queue' \in QUEUE \wedge queue' \in 1\,..\,qsize-1 \rightarrowtail\!\!\!\!\rightarrow queuetokens \setminus \{queue(1)\} \wedge (\forall i \cdot i \in 1\,..\,qsize-1 \Rightarrow queue'(i) = queue(i+1))$$

$act2$: $queueitems := \{queue(1)\} \lhd queueitems$

$act3$: $queuetokens := queuetokens \setminus \{queue(1)\}$

$act4$: $qsize := qsize - 1$

end

The QueueA machine VI

Event $Unqueue \mathrel{\widehat{=}}$

any

qid

when

$grd1$: $qid \in queuetokens$

$grd2$: $qsize \neq 0$

The QueueA machine VII

then

*act*1:

$$queue :| queue' \in 1 \,..\, (qsize - 1) \twoheadrightarrowtail queuetokens \setminus \{qid\}$$
$$\land (qsize = 1 \Rightarrow queue' = \varnothing)$$
$$\land (qsize > 1 \Rightarrow$$
$$(\forall i \cdot i \in 1 \,..\, queue^{-1}(qid) - 1 \Rightarrow queue'(i) = queue(i))$$
$$\land$$
$$(\forall j \cdot j \in queue^{-1}(qid) + 1 \,..\, qsize$$
$$\Rightarrow queue'(j - 1) = queue(j)))$$

*act*2: $queueitems := \{qid\} \lhd queueitems$

*act*3: $queuetokens := queuetokens \setminus \{qid\}$

*act*4: $qsize := qsize - 1$

end

END

Next we will define a QueueType with queue operations.

CONTEXT QueueType
EXTENDS QueueContext
CONSTANTS

ENQUEUE

DEQUEUE

DELETE

AXIOMS

axm1: $ENQUEUE \in QUEUE \times TOKEN \rightarrowtail QUEUE$

axm2: $\forall q, t \cdot q \in QUEUE \land t \notin ran(q) \Rightarrow q \mapsto t \in dom(ENQUEUE)$

axm3: $\forall q, t \cdot q \in QUEUE \land t \notin ran(q) \Rightarrow ENQUEUE(q \mapsto t) = q \lhd\!\!\!- \{card(q) + 1 \mapsto t\}$

axm4: $\forall q, t \cdot q \in QUEUE \land t \in TOKEN \land t \notin ran(q)$
$\Rightarrow card(ENQUEUE(q \mapsto t)) = card(q) + 1$

axm5: $\forall q, t \cdot q \in QUEUE \land t \in TOKEN \land t \notin ran(q)$
$\Rightarrow dom(ENQUEUE(q \mapsto t)) = 1 \,..\, card(q) + 1$

axm6: $\forall q, t, i \cdot q \in QUEUE \land t \notin ran(q) \Rightarrow$
$(i \in dom(q) \Rightarrow ENQUEUE(q \mapsto t)(i) = q(i))$
$\land$
$(i = card(q) + 1 \Rightarrow ENQUEUE(q \mapsto t)(i) = t)$

axm7: $DEQUEUE \in QUEUE \nrightarrow QUEUE$

axm8: $dom(DEQUEUE) = QUEUE \setminus \{\varnothing\}$

axm9: $\forall q \cdot q \in QUEUE \land q \neq \varnothing$
$\Rightarrow$
$DEQUEUE(q) \in 1 \,..\, card(q) - 1 \twoheadrightarrowtail ran(q) \setminus \{q(1)\}$

axm10: $\forall q \cdot q \in dom(DEQUEUE)$
$\Rightarrow$
$card(DEQUEUE(q)) = card(q) - 1$

axm11: $\forall q \cdot q \in dom(DEQUEUE) \Rightarrow dom(DEQUEUE(q)) = 1 \,..\, card(q) - 1$

axm12: $\forall q, i \cdot q \in dom(DEQUEUE) \land i \in dom(DEQUEUE(q)) \Rightarrow DEQUEUE(q)(i) = q(i + 1)$

axm13: $DELETE \in QUEUE \times \mathbb{N}_1 \nrightarrow QUEUE$

axm14: $\forall q, i \cdot q \in QUEUE \land i \in dom(q) \Leftrightarrow q \mapsto i \in dom(DELETE)$

axm15: $\forall q, i \cdot q \mapsto i \in dom(DELETE) \Rightarrow card(DELETE(q \mapsto i)) = card(q) - 1$

axm16: $\forall q, i \cdot q \mapsto i \in dom(DELETE) \Rightarrow dom(DELETE(q \mapsto i)) = 1 \,..\, card(q) - 1$

*axm*17:

$$\forall q, i, j \cdot q \mapsto i \in dom(DELETE)$$
$$\Rightarrow$$
$$(j < i \land j \in dom(q) \Rightarrow DELETE(q \mapsto i)(j) = q(j))$$
$$\land$$
$$(j \geq i \land j + 1 \in dom(q) \Rightarrow DELETE(q \mapsto i)(j) = q(j + 1))$$

END

The QueueB machine I

Then refine QueueA to QueueB using QueueType

MACHINE QueueB

REFINES QueueA

SEES QueueType

VARIABLES

queuetokens tokens for currently queued items

queue the queue of tokens

queueitems a function for fetching the item associated with a token

qsize current size of queue

The QueueB machine II

INVARIANTS

*inv*1: $queuetokens \subseteq TOKEN$

*inv*2: $queue \in QUEUE$

*inv*3: $qsize = card(queue)$

*inv*4: $queue \in 1 \,..\, qsize \rightarrowtail\!\!\!\!\twoheadrightarrow queuetokens$

*inv*5: $\forall i, j \cdot i \in dom(queue) \land j \in dom(queue) \land i \neq j$
$\Rightarrow$
$queue(i) \neq queue(j)$

*inv*6: $queuetokens = ran(queue)$

*inv*7: $queueitems \in queuetokens \rightarrow ITEM$

*inv*8: $queue^{-1} \in queuetokens \rightarrowtail\!\!\!\!\twoheadrightarrow 1 \,..\, qsize$

*inv*9:
$(\forall qid \cdot qid \in TOKEN \setminus queuetokens$
$\Rightarrow$
$ENQUEUE(queue \mapsto qid) = queue \lhd\!\!- \{qsize + 1 \mapsto qid\})$

The QueueB machine III

*inv*10: $\forall qid \cdot qid \in queuetokens$
$\Rightarrow$
$queue \mapsto queue^{-1}(qid) \in dom(DELETE)$

*inv*11:
$qsize \neq 1$
$\Rightarrow$
$(\forall qid, i \cdot qid \in queuetokens \land i \in 1 \,..\, (queue^{-1}(qid) - 1)$
$\Rightarrow$
$(DELETE(queue \mapsto queue^{-1}(qid)))(i) = queue(i))$

*inv*12:
$qsize \neq 1$
$\Rightarrow$
$(\forall qid, i \cdot qid \in queuetokens \land i \in queue^{-1}(qid) + 1 \,..\, qsize$
$\Rightarrow$
$(DELETE(queue \mapsto queue^{-1}(qid)))(i - 1) = queue(i))$

The QueueB machine IV

$$inv13: \quad \forall qid \cdot qid \in queuetokens \Rightarrow queue^{-1}(qid) \leq qsize$$

The QueueB machine V

EVENTS

Initialisation

begin

act1: $queuetokens := \varnothing$

act2: $queue := \varnothing$

act3: $qsize := 0$

act4: $queueitems := \varnothing$

end

The QueueB machine VI

Event *Enqueue* $\widehat{=}$
refines *Enqueue*
any
item
qid
when
grd1: $item \in ITEM$
grd2: $qid \in TOKEN \setminus queuetokens$
then
act1: $queuetokens := queuetokens \cup \{qid\}$
act2: $queue := ENQUEUE(queue \mapsto qid)$
act3: $queueitems(qid) := item$
act4: $qsize := qsize + 1$
end

The QueueB machine VII

Event *Dequeue* $\widehat{=}$
refines *Dequeue*
when
grd1: $qsize \neq 0$
then
act1: $queue := DEQUEUE(queue)$
act2: $queueitems := \{queue(1)\} \lhd queueitems$
act3: $queuetokens := queuetokens \setminus \{queue(1)\}$
act4: $qsize := qsize - 1$
end

The QueueB machine VIII

Event $Unqueue \mathrel{\widehat{=}}$
refines $Unqueue$
any
qid
when
$grd1$: $qid \in queuetokens$
then
$act1$: $queue := DELETE(queue \mapsto queue^{-1}(qid))$
$act2$: $queueitems := \{qid\} \lhd queueitems$
$act3$: $queuetokens := queuetokens \setminus \{qid\}$
$act4$: $qsize := qsize - 1$
end
END

Refining the Queue machine

The refinement replaces the monolithic sequence model by a list model, in which the discrete elements of the set *queuetokens* are organised as a list using the following variables:

- qfirst the first element of the list;
- qlast the last element of the list;
- qnext a function that links an element of the list to the next element in the list —relevant only to lists with more than one item;
- qsize the size of the list.

picture required

Refining the Queue machine

The refinement replaces the monolithic sequence model by a list model, in which the discrete elements of the set *queuetokens* are organised as a list using the following variables:

- qfirst the first element of the list;
- qlast the last element of the list;
- qnext a function that links an element of the list to the next element in the list —relevant only to lists with more than one item;
- qsize the size of the list.

picture required

Refining the Queue machine

The refinement replaces the monolithic sequence model by a list model, in which the discrete elements of the set *queuetokens* are organised as a list using the following variables:

qfirst the first element of the list;

qlast the last element of the list;

qnext a function that links an element of the list to the next element in the list —relevant only to lists with more than one item;

qsize the size of the list.

picture required

Refining the Queue machine

The refinement replaces the monolithic sequence model by a list model, in which the discrete elements of the set *queuetokens* are organised as a list using the following variables:

- **qfirst** the first element of the list;
- **qlast** the last element of the list;
- **qnext** a function that links an element of the list to the next element in the list —relevant only to lists with more than one item;
- **qsize** the size of the list.

picture required

Refining the Queue machine

The refinement replaces the monolithic sequence model by a list model, in which the discrete elements of the set *queuetokens* are organised as a list using the following variables:

qfirst	the first element of the list;
qlast	the last element of the list;
qnext	a function that links an element of the list to the next element in the list —relevant only to lists with more than one item;
qsize	the size of the list.

picture required

Additionally, the refinement uses the variable *queueitem* in the same role as in the *Queue* machine. Although this variables has the same name it is a new variable that is related by equivalence to the variable in the refined machine.

A refinement relation relates the list model to the queue model.

Data refinements may not use variables of the refined machine except in invariants. *Complete hiding* is enforced.

Additionally, the refinement uses the variable *queueitem* in the same role as in the *Queue* machine. Although this variables has the same name it is a new variable that is related by equivalence to the variable in the refined machine.

A refinement relation relates the list model to the queue model.

Data refinements may not use variables of the refined machine except in invariants. *Complete hiding* is enforced.

Additionally, the refinement uses the variable *queueitem* in the same role as in the *Queue* machine. Although this variables has the same name it is a new variable that is related by equivalence to the variable in the refined machine.

A refinement relation relates the list model to the queue model.

Data refinements may not use variables of the refined machine except in invariants. *Complete hiding* is enforced.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext \,;\, qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself n times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \,;\, r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext \,;\, qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself n times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \,;\, r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext \; ; \; qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself *n* times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \; ; \; r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext\ ;\ qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself n times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n\ ;\ r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext\ ;\ qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself *n* times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n\ ;\ r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext \mathbin{;} qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself *n* times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \mathbin{;} r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext \mathbin{;} qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself *n* times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \mathbin{;} r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext \; ; \; qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself *n* times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \; ; \; r$.

Relational composition and iteration

Since we are modelling a list structure we will use *relational composition* on the *qnext* function to describes paths along the list, and we will use transitive *closure* of *qnext* to describe reachability.

Suppose we have a list with at least 2 elements, then

$qfirst$	gives the identity of the first item in the list
$qnext(qfirst)$	gives the identity of the second item in the list
$(qnext \mathbin{;} qnext)(qfirst)$	gives the identity of the third item in the list
$\ldots$	etc

Multiple composition is expressed by *iteration*: $qnext^n$ (provided by the constant function $iterate(qnext \mapsto n)$), is the result of composing *qnext* with itself *n* times.

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \mathbin{;} r$.

Closure

Reflexive transitive closure of a relation r, written r^*, is the union of all iterations of r, that is

$$r^* = \bigcup n.(n \in \mathbb{N} \mid r^n)^\dagger$$

Irreflexive transitive closure of a relation, written r^+, does not explicitly include r^0 from the union

$$r^+ = \bigcup n.(n \in \mathbb{N}_1 \mid r^n),$$

but it may be present, depending on r. EventB (RODIN) does not supply *closure*; it has to be defined as a constant function.

† It should be clear that continuous composition of a relation with itself will eventually reach a stationary relation.

Closure

Reflexive transitive closure of a relation r, written r^*, is the union of all iterations of r, that is

$$r^* = \bigcup n.(n \in \mathbb{N} \mid r^n)^\dagger$$

Irreflexive transitive closure of a relation, written r^+, does not explicitly include r^0 from the union

$$r^+ = \bigcup n.(n \in \mathbb{N}_1 \mid r^n),$$

but it may be present, depending on r. EventB (RODIN) does not supply *closure*; it has to be defined as a constant function.

† It should be clear that continuous composition of a relation with itself will eventually reach a stationary relation.

Relational composition of functions

It should be clear that if f is a function then $f \,;\, f$ is also a function and by extrapolation f^n is a function.

Further, if f is an injective function then f^n is also an injective function.

Thus, $qnext^n$ is an injective function that gives all paths of length n within the list.

$qnext^+$ is a set of injective functions representing all paths, of all lengths from 0 to the length of the list, within the list.

It follows that $qnext^+[\{qfirst\}]$, the image of the first node in the list under $qnext^+$, is the set of all nodes in the list.

Relational composition of functions

It should be clear that if f is a function then $f \, ; f$ is also a function and by extrapolation f^n is a function.

Further, if f is an injective function then f^n is also an injective function.

Thus, $qnext^n$ is an injective function that gives all paths of length n within the list.

$qnext^+$ is a set of injective functions representing all paths, of all lengths from 0 to the length of the list, within the list.

It follows that $qnext^+[\{qfirst\}]$, the image of the first node in the list under $qnext^+$, is the set of all nodes in the list.

Relational composition of functions

It should be clear that if f is a function then $f\,;f$ is also a function and by extrapolation f^n is a function.

Further, if f is an injective function then f^n is also an injective function.

Thus, $qnext^n$ is an injective function that gives all paths of length n within the list.

$qnext^+$ is a set of injective functions representing all paths, of all lengths from 0 to the length of the list, within the list.

It follows that $qnext^+[\{qfirst\}]$, the image of the first node in the list under $qnext^+$, is the set of all nodes in the list.

Relational composition of functions

It should be clear that if f is a function then $f\,;f$ is also a function and by extrapolation f^n is a function.

Further, if f is an injective function then f^n is also an injective function.

Thus, $qnext^n$ is an injective function that gives all paths of length n within the list.

$qnext^+$ is a set of injective functions representing all paths, of all lengths from 0 to the length of the list, within the list.

It follows that $qnext^+[\{qfirst\}]$, the image of the first node in the list under $qnext^+$, is the set of all nodes in the list.

Relational composition of functions

It should be clear that if f is a function then $f \,;\, f$ is also a function and by extrapolation f^n is a function.

Further, if f is an injective function then f^n is also an injective function.

Thus, $qnext^n$ is an injective function that gives all paths of length n within the list.

$qnext^+$ is a set of injective functions representing all paths, of all lengths from 0 to the length of the list, within the list.

It follows that $qnext^+[\{qfirst\}]$, the image of the first node in the list under $qnext^+$, is the set of all nodes in the list.

Relational composition of functions

It should be clear that if f is a function then $f \,;\, f$ is also a function and by extrapolation f^n is a function.

Further, if f is an injective function then f^n is also an injective function.

Thus, $qnext^n$ is an injective function that gives all paths of length n within the list.

$qnext^+$ is a set of injective functions representing all paths, of all lengths from 0 to the length of the list, within the list.

It follows that $qnext^+[\{qfirst\}]$, the image of the first node in the list under $qnext^+$, is the set of all nodes in the list.

CONTEXT Iteration

EXTENDS QueueType

CONSTANTS

`iterate`

`iclosure`

AXIOMS

axm1:

$iterate \in (TOKEN \leftrightarrow TOKEN) \times \mathbb{N} \rightarrow (TOKEN \leftrightarrow TOKEN)$

axm2:

$\forall r \cdot r \in TOKEN \leftrightarrow TOKEN \Rightarrow iterate(r \mapsto 0) = dom(r) \lhd id$

axm3: $\forall r, n \cdot r \in TOKEN \leftrightarrow TOKEN \land n \in \mathbb{N}_1$

$\Rightarrow$

$iterate(r \mapsto n) = iterate(r \mapsto n - 1); r$

thm1: $\forall s \cdot s \subseteq \mathbb{N} \land 0 \in s \land (\forall n \cdot n \in s \Rightarrow n + 1 \in s) \Rightarrow \mathbb{N} \subseteq s$

$thm2$: $\forall r, n \cdot r \in TOKEN \leftrightarrow TOKEN \land n \in \mathbb{N}_1$
$\Rightarrow$
$dom(iterate(r \mapsto n)) \subseteq dom(r)$

$thm3$: $\forall r, n \cdot r \in TOKEN \leftrightarrow TOKEN \land n \in \mathbb{N}_1$
$\Rightarrow$
$ran(iterate(r \mapsto n)) \subseteq ran(r)$

thm4:
$$iclosure \in (TOKEN \leftrightarrow TOKEN) \rightarrow (TOKEN \leftrightarrow TOKEN)$$

axm5: $\forall r \cdot r \in TOKEN \leftrightarrow TOKEN$
$$\Rightarrow$$
$$iclosure(r) = (\bigcup n \cdot n \in \mathbb{N}_1 | iterate(r \mapsto n))$$

thm5: $\forall r \cdot r \in TOKEN \leftrightarrow TOKEN$
$$\Rightarrow$$
$$dom(iclosure(r)) \subseteq dom(r)$$

END

The invariant of QueueR

The list consists of the elements of *queuetokens* hence

$$qsize = card(queuetokens)$$

For non-empty lists, *qfirst* and *qlast* are elements of *queuetokens*

$$queuetokens \neq \varnothing \implies qfirst \in queuetokens$$
$$queuetokens \neq \varnothing \implies qlast \in queuetokens$$

The list is linear and connected, hence qnext is injective, but it is also surjective and therefore bijective:

$$qnext \in queuetokens \setminus \{qlast\} \rightarrowtail queuetokens \setminus \{qfirst\}$$

The invariant of QueueR

The list consists of the elements of *queuetokens* hence

$$qsize = card(queuetokens)$$

For non-empty lists, *qfirst* and *qlast* are elements of *queuetokens*

$$queuetokens \neq \varnothing \implies qfirst \in queuetokens$$
$$queuetokens \neq \varnothing \implies qlast \in queuetokens$$

The list is linear and connected, hence qnext is injective, but it is also surjective and therefore bijective:

$$qnext \in queuetokens \setminus \{qlast\} \rightarrowtail\!\!\!\!\twoheadrightarrow queuetokens \setminus \{qfirst\}$$

The invariant of QueueR

The list consists of the elements of *queuetokens* hence

$$qsize = card(queuetokens)$$

For non-empty lists, *qfirst* and *qlast* are elements of *queuetokens*

$$queuetokens \neq \varnothing \implies qfirst \in queuetokens$$
$$queuetokens \neq \varnothing \implies qlast \in queuetokens$$

The list is linear and connected, hence qnext is injective, but it is also surjective and therefore bijective:

$$qnext \in queuetokens \setminus \{qlast\} \rightarrowtail\!\!\!\!\twoheadrightarrow queuetokens \setminus \{qfirst\}$$

The invariant of QueueR

The list consists of the elements of *queuetokens* hence

$$qsize = card(queuetokens)$$

For non-empty lists, *qfirst* and *qlast* are elements of *queuetokens*

$$\begin{aligned} queuetokens \neq \varnothing &\implies qfirst \in queuetokens \\ queuetokens \neq \varnothing &\implies qlast \in queuetokens \end{aligned}$$

The list is linear and connected, hence qnext is injective, but it is also surjective and therefore bijective:

$$qnext \in queuetokens \setminus \{qlast\} \rightarrowtail\!\!\!\!\!\twoheadrightarrow queuetokens \setminus \{qfirst\}$$

The Refinement relation

Each element of the queue model can be retrieved from the list model

$$\forall i \cdot i \in 1 \,..\, qsize \implies queue(i) = qnext^{i-1}(qfirst)$$

QueueR Theorems

The following should follow from the invariant:

1. Any element of the list that is not *qfirst* must be in $\mathrm{dom}(qnext)$

$$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qlast \Rightarrow t \in dom(qnext)$$

2. Any element of the list that is not *qlast* must be in $\mathrm{ran}(qnext)$

$$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qfirst \Rightarrow t \in ran(qnext)$$

3. Following all sequences of *qnext* from *qfirst* should give all tokens in *queuetokens*

$$closure1(qnext)[\{qfirst\}] = queuetokens$$

4. The following should also follow from the refinement relation:
   $qsize \neq 0 \Rightarrow queue(1) = qfirst$
   $qsize \neq 0 \Rightarrow queue(qsize) = qlast$
   $qsize \neq 0 \Rightarrow \forall i \cdot i \in 1 \,..\, qsize - 1 \Rightarrow queue(i + 1) = qnext(queue(i))$

QueueR Theorems

The following should follow from the invariant:

1. Any element of the list that is not *qfirst* must be in $\mathrm{dom}(qnext)$

$$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qlast \Rightarrow t \in dom(qnext)$$

2. Any element of the list that is not *qlast* must be in $\mathrm{ran}(qnext)$

$$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qfirst \Rightarrow t \in ran(qnext)$$

3. Following all sequences of *qnext* from *qfirst* should give all tokens in *queuetokens*

$$closure1(qnext)[\{qfirst\}] = queuetokens$$

4. The following should also follow from the refinement relation:
   $qsize \neq 0 \Rightarrow queue(1) = qfirst$
   $qsize \neq 0 \Rightarrow queue(qsize) = qlast$
   $qsize \neq 0 \Rightarrow \forall i \cdot i \in 1 \,..\, qsize - 1 \Rightarrow queue(i+1) = qnext(queue(i))$

QueueR Theorems

The following should follow from the invariant:

1. Any element of the list that is not *qfirst* must be in $\mathrm{dom}(qnext)$

   $$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qlast \Rightarrow t \in dom(qnext)$$

2. Any element of the list that is not *qlast* must be in $\mathrm{ran}(qnext)$

   $$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qfirst \Rightarrow t \in ran(qnext)$$

3. Following all sequences of *qnext* from *qfirst* should give all tokens in *queuetokens*

   $$closure1(qnext)[\{qfirst\}] = queuetokens$$

4. The following should also follow from the refinement relation:
   $qsize \neq 0 \Rightarrow queue(1) = qfirst$
   $qsize \neq 0 \Rightarrow queue(qsize) = qlast$
   $qsize \neq 0 \Rightarrow \forall i \cdot i \in 1 \,..\, qsize - 1 \Rightarrow queue(i+1) = qnext(queue(i))$

QueueR Theorems

The following should follow from the invariant:

1. Any element of the list that is not *qfirst* must be in $\mathrm{dom}(qnext)$

   $$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qlast \Rightarrow t \in dom(qnext)$$

2. Any element of the list that is not *qlast* must be in $\mathrm{ran}(qnext)$

   $$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qfirst \Rightarrow t \in ran(qnext)$$

3. Following all sequences of *qnext* from *qfirst* should give all tokens in *queuetokens*

   $$closure1(qnext)[\{qfirst\}] = queuetokens$$

4. The following should also follow from the refinement relation:
   $qsize \neq 0 \Rightarrow queue(1) = qfirst$
   $qsize \neq 0 \Rightarrow queue(qsize) = qlast$
   $qsize \neq 0 \Rightarrow \forall i \cdot i \in 1 \,..\, qsize - 1 \Rightarrow queue(i+1) = qnext(queue(i))$

QueueR Theorems

The following should follow from the invariant:

1. Any element of the list that is not *qfirst* must be in $\mathrm{dom}(qnext)$

$$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qlast \Rightarrow t \in dom(qnext)$$

2. Any element of the list that is not *qlast* must be in $\mathrm{ran}(qnext)$

$$\forall t \cdot t \in queuetokens \land qsize > 1 \land t \neq qfirst \Rightarrow t \in ran(qnext)$$

3. Following all sequences of *qnext* from *qfirst* should give all tokens in *queuetokens*

$$closure1(qnext)[\{qfirst\}] = queuetokens$$

4. The following should also follow from the refinement relation:
   $qsize \neq 0 \Rightarrow queue(1) = qfirst$
   $qsize \neq 0 \Rightarrow queue(qsize) = qlast$
   $qsize \neq 0 \Rightarrow \forall i \cdot i \in 1 \,..\, qsize - 1 \Rightarrow queue(i+1) = qnext(queue(i))$

Loops

There must be no loops. When moving from a monolithic structure to a list it is clear that loops are possible. It is easy to see by informal induction on the way the list is built that there will be no loops, but it follows from the type of *qnext*, so the following should be a theorem:

$$qnext^{+} \cap id(queuetokens) = \varnothing$$

Traversing the list from *qfirst* should cover all the elements of *queuetokens*

$$qnext^{+}[\{qfirst\}] = queuetokens$$

Loops

There must be no loops. When moving from a monolithic structure to a list it is clear that loops are possible. It is easy to see by informal induction on the way the list is built that there will be no loops, but it follows from the type of *qnext*, so the following should be a theorem:

$$qnext^{+} \cap id(queuetokens) = \varnothing$$

Traversing the list from *qfirst* should cover all the elements of *queuetokens*

$$qnext^{+}[\{qfirst\}] = queuetokens$$

The QueueR machine I

MACHINE QueueR

REFINES QueueB

SEES Iteration

VARIABLES

`queuetokens`	tokens currently in queue
`queueitems`	a function for fetching the item associated with a token
`qsize`	current size of queue
`qfirst`	first item, if any, in queue
`qlast`	last item, if any, in queue
`qnext`	link to next item, if any, in queue

INVARIANTS

*inv*1: $qfirst \in TOKEN$

*inv*2: $qlast \in TOKEN$

*inv*3: $qsize \neq 0 \Rightarrow qfirst = queue(1)$

The QueueR machine II

$inv4$: $qsize \neq 0 \Rightarrow qlast = queue(qsize)$

$inv5$: $qnext \in queuetokens \rightarrowtail\!\!\!\!\!\!\!\!\twoheadrightarrow queuetokens$

$inv6$: $dom(qnext) = queuetokens \setminus \{qlast\}$

$inv7$: $qnext \cap id = \varnothing$

$inv8$: $ran(qnext) = queuetokens \setminus \{qfirst\}$

$inv9$: $qsize = 1 \Rightarrow qfirst = qlast$

$inv10$: $\forall i \cdot i \in 1 \,..\, qsize \land i < qsize \Rightarrow qnext(queue(i)) = queue(i+1)$

$inv11$: $qsize \geq 1 \Rightarrow iterate(qnext \mapsto 0)[\{qfirst\}] = \{queue(1)\}$

$inv12$: $qsize \geq 1 \Rightarrow (\forall n \cdot n \in 1 \,..\, qsize - 1 \land iterate(qnext \mapsto n-1)[\{qfirst\}] = \{queue(n)\} \Rightarrow iterate(qnext \mapsto n)[\{qfirst\}] = \{queue(n+1)\})$

The QueueR machine III

$inv13$: $qsize \geq 1$
$\Rightarrow (\forall n \cdot n \in 1 \,..\, qsize - 1$
$\Rightarrow iterate(qnext \mapsto n - 1)[\{qfirst\}] = \{queue(n)\})$

$inv14$: $qsize \geq 1 \Rightarrow iclosure(qnext)[\{qfirst\}] = queuetokens$

The QueueR machine IV

EVENTS

Initialisation

begin

$act1$: $queuetokens := \varnothing$

$act2$: $qsize := 0$

$act3$: $queueitems := \varnothing$

$act4$: $qfirst :\in TOKEN$

$act5$: $qlast :\in TOKEN$

$act6$: $qnext := \varnothing$

end

The QueueR machine V

Event $Enqueue0 \mathrel{\widehat{=}}$
refines $Enqueue$
any
$item$
qid
when
grd1: $item \in ITEM$
grd2: $qid \in TOKEN \setminus queuetokens$
grd3: $qsize = 0$
then
act1: $queuetokens := queuetokens \cup \{qid\}$
act2: $queueitems(qid) := item$
act3: $qsize := qsize + 1$
act4: $qfirst := qid$
act5: $qlast := qid$
end

The QueueR machine VI

Event	*Enqueue1* $\widehat{=}$
refines	*Enqueue*
any	
item	
qid	
when	
grd1:	$item \in ITEM$
grd2:	$qid \in TOKEN \setminus queuetokens$
grd3:	$qsize \neq 0$
then	
act1:	$queuetokens := queuetokens \cup \{qid\}$
act2:	$queueitems(qid) := item$
act3:	$qsize := qsize + 1$
act4:	$qnext(qlast) := qid$
act5:	$qlast := qid$
end	

The QueueR machine VII

Event	*Dequeue0* $\widehat{=}$
refines	*Dequeue*
when	
grd1:	$qsize = 1$
then	
act1:	$qsize := qsize - 1$
act2:	$queuetokens := queuetokens \setminus \{qfirst\}$
act3:	$queueitems := \{qfirst\} \lhd queueitems$
act4:	$qnext := \{qfirst\} \lhd qnext$
end	

The QueueR machine VIII

Event $\mathit{Dequeue1} \mathrel{\widehat{=}}$

refines $\mathit{Dequeue}$

when

*grd*1: $\mathit{qsize} > 1$

then

*act*1: $\mathit{qsize} := \mathit{qsize} - 1$

*act*2: $\mathit{queuetokens} := \mathit{queuetokens} \setminus \{\mathit{qfirst}\}$

*act*3: $\mathit{queueitems} := \{\mathit{qfirst}\} \lhd\!\!\!- \mathit{queueitems}$

*act*4: $\mathit{qfirst} := \mathit{qnext}(\mathit{qfirst})$

*act*5: $\mathit{qnext} := \{\mathit{qfirst}\} \lhd\!\!\!- \mathit{qnext}$

end

The QueueR machine IX

Event $Unqueue0 \mathrel{\widehat{=}}$

refines $Unqueue$

any

qid

when

$grd1$: $qid \in queuetokens$

$grd2$: $qsize = 1$

then

$act1$: $queueitems := \{qid\} \lhd queueitems$

$act2$: $queuetokens := queuetokens \setminus \{qid\}$

$act3$: $qsize := qsize - 1$

end

The QueueR machine X

Event	*Unqueue1* $\widehat{=}$
refines	*Unqueue*
any	
qid	
when	
*grd*1:	$qid \in queuetokens$
*grd*2:	$qsize > 1$
*grd*3:	$qid = qfirst$
then	
*act*1:	$queueitems := \{qid\} \lhd queueitems$
*act*2:	$queuetokens := queuetokens \setminus \{qid\}$
*act*3:	$qsize := qsize - 1$
*act*4:	$qfirst := qnext(qid)$
*act*5:	$qnext := \{qid\} \lhd qnext$
end	

The QueueR machine XI

Event $Unqueue2 \mathrel{\widehat{=}}$
refines $Unqueue$
any
qid
when
$grd1$: $qid \in queuetokens$
$grd2$: $qsize > 1$
$grd3$: $qlast = qid$
then
$act1$: $queueitems := \{qid\} \lhd queueitems$
$act2$: $queuetokens := queuetokens \setminus \{qid\}$
$act3$: $qsize := qsize - 1$
$act4$: $qlast := qnext^{-1}(qid)$
$act5$: $qnext := qnext \rhd \{qid\}$
end

The QueueR machine XII

Event $\mathit{Unqueue3} \widehat{=}$

refines $\mathit{Unqueue}$

any

qid

when

$\mathit{grd}1$: $\mathit{qid} \in \mathit{queuetokens}$

$\mathit{grd}2$: $\mathit{qsize} > 1$

$\mathit{grd}3$: $\mathit{qfirst} \neq \mathit{qid}$

$\mathit{grd}4$: $\mathit{qlast} \neq \mathit{qid}$

then

$\mathit{act}1$: $\mathit{queueitems} := \{\mathit{qid}\} \lhd\!\!\!- \mathit{queueitems}$

$\mathit{act}2$: $\mathit{queuetokens} := \mathit{queuetokens} \setminus \{\mathit{qid}\}$

$\mathit{act}3$: $\mathit{qsize} := \mathit{qsize} - 1$

$\mathit{act}4$: $\mathit{qnext}(\mathit{qnext}^{-1}(\mathit{qid})) := \mathit{qnext}(\mathit{qid})$

end

END

Refinement of Unqueue3

The event Unqueue3 deletes an item from within the queue, that is neither the first or last items on the queue.

Implementing prev

Until now we got *prev* for free because *qnext* is an injective function, so *prev* has been obtained by simply inverting *qnext*. In an implementation we have no such luxury. In the refinement of Unqueue3 we implement *prev* by using a loop to search from the beginning of the queue (list) for the predecessor of the item to be deleted. This, of course, is inefficient. If efficiency is important, we could implement a doubly linked list, ie implement *qprev*.

Refinement of Unqueue3

The event Unqueue3 deletes an item from within the queue, that is neither the first or last items on the queue.

Implementing prev

Until now we got *prev* for free because *qnext* is an injective function, so *prev* has been obtained by simply inverting *qnext*. In an implementation we have no such luxury. In the refinement of Unqueue3 we implement *prev* by using a loop to search from the beginning of the queue (list) for the predecessor of the item to be deleted. This, of course, is inefficient. If efficiency is important, we could implement a doubly linked list, ie implement *qprev*.

Refinement of Unqueue3

The event Unqueue3 deletes an item from within the queue, that is neither the first or last items on the queue.

Implementing prev

Until now we got *prev* for free because *qnext* is an injective function, so *prev* has been obtained by simply inverting *qnext*. In an implementation we have no such luxury. In the refinement of Unqueue3 we implement *prev* by using a loop to search from the beginning of the queue (list) for the predecessor of the item to be deleted. This, of course, is inefficient. If efficiency is important, we could implement a doubly linked list, ie implement *qprev*.

Preventing Interference

The refinement of Unqueue3 consists of three events:

Unqueue3I: initiates the computation of *qprev*. This event sets *qprev* to *qfirst* and sets a flag, *deleting*, to *TRUE*.

Unqueue3M: an event that represents *still searching*. It advances *qprev* to *qnext*(*qprev*).

Unqueue3F: the final step. The item to be deleted has been found, so the current value of *qprev* is the value we want. This event does the deletion and sets *deleting* to *FALSE*.

The purpose of *deleting*

Until the deletion is complete the other queue events must not run as the state of the queue is not yet correct. Until now *Unqueue*3 was an atomic event; in this refinement the actions of that event are spread across three events.

Preventing Interference

The refinement of Unqueue3 consists of three events:

Unqueue3I: initiates the computation of *qprev*. This event sets *qprev* to *qfirst* and sets a flag, *deleting*, to *TRUE*.

Unqueue3M: an event that represents *still searching*. It advances *qprev* to *qnext*(*qprev*).

Unqueue3F: the final step. The item to be deleted has been found, so the current value of *qprev* is the value we want. This event does the deletion and sets *deleting* to *FALSE*.

The purpose of *deleting*

Until the deletion is complete the other queue events must not run as the state of the queue is not yet correct. Until now *Unqueue*3 was an atomic event; in this refinement the actions of that event are spread across three events.

QueueRR I

MACHINE QueueRR

REFINES QueueR

SEES Iteration

VARIABLES

`queuetokens`	tokens currently in queue
`queueitems`	a function for fetching the item associated with a token
`qsize`	current size of queue
`qfirst`	first item, if any, in queue
`qlast`	last item, if any, in queue
`qnext`	link to next item, if any, in queue
`deleting`	Unqueue deletion in progress
`qprev`	concrete version of queue
`qidv`	copy of qid

QueueRR II

INVARIANTS

$inv1$: $deleting \in BOOL$

$inv2$: $qprev \in TOKEN$

$inv3$: $qidv \in TOKEN$

$inv4$: $deleting = TRUE \Rightarrow qidv \in queuetokens$

$inv5$: $deleting = TRUE \Rightarrow qidv \neq qfirst$

$inv6$: $deleting = TRUE \Rightarrow qsize > 1$

$inv7$: $deleting = TRUE \Rightarrow qprev \in dom(qnext)$

$inv8$: $deleting = TRUE \Rightarrow qidv \in iclosure(qnext)[\{qprev\}]$

QueueRR III

EVENTS

Initialisation

extended

begin

act1 : $queuetokens := \varnothing$

act2 : $qsize := 0$

act3 : $queueitems := \varnothing$

act4 : $qfirst :\in TOKEN$

act5 : $qlast :\in TOKEN$

act6 : $qnext := \varnothing$

act7: $deleting := FALSE$

act8: $qprev :\in TOKEN$

act9: $qidv :\in TOKEN$

QueueRR IV

end
Event *Enqueue0* $\widehat{=}$
extends *Enqueue0*
any
`item`
`qid`
when
`grd1` : $item \in ITEM$
`grd2` : $qid \in TOKEN \setminus queuetokens$
`grd3` : $qsize = 0$
grd4: $deleting = FALSE$
then
`act1` : $queuetokens := queuetokens \cup \{qid\}$

QueueRR V

$\texttt{act2}: \mathit{queueitems}(\mathit{qid}) := \mathit{item}$

$\texttt{act3}: \mathit{qsize} := \mathit{qsize} + 1$

$\texttt{act4}: \mathit{qfirst} := \mathit{qid}$

$\texttt{act5}: \mathit{qlast} := \mathit{qid}$

end

QueueRR VI

Event *Enqueue1* $\widehat{=}$
extends *Enqueue1*
any
`item`
`qid`
when
`grd1` : $item \in ITEM$
`grd2` : $qid \in TOKEN \setminus queuetokens$
`grd3` : $qsize \neq 0$
grd4: $deleting = FALSE$
then
`act1` : $queuetokens := queuetokens \cup \{qid\}$
`act2` : $queueitems(qid) := item$

QueueRR VII

$\texttt{act3}: \ qsize := qsize + 1$

$\texttt{act4}: \ qnext(qlast) := qid$

$\texttt{act5}: \ qlast := qid$

end

QueueRR VIII

Event $\mathit{Dequeue0} \mathrel{\widehat{=}}$

extends $\mathit{Dequeue0}$

when

grd1 : $\mathit{qsize} = 1$

grd2: $\mathit{deleting} = \mathit{FALSE}$

then

act1 : $\mathit{qsize} := \mathit{qsize} - 1$

act2 : $\mathit{queuetokens} := \mathit{queuetokens} \setminus \{\mathit{qfirst}\}$

act3 : $\mathit{queueitems} := \{\mathit{qfirst}\} \lhd\!\!\!- \mathit{queueitems}$

act4 : $\mathit{qnext} := \{\mathit{qfirst}\} \lhd\!\!\!- \mathit{qnext}$

end

QueueRR IX

Event *Deque1* $\widehat{=}$
extends *Deque1*
when
grd1 : $qsize > 1$
grd2: $deleting = FALSE$
then
act1 : $qsize := qsize - 1$
act2 : $queuetokens := queuetokens \setminus \{qfirst\}$
act3 : $queueitems := \{qfirst\} \lhd queueitems$
act4 : $qfirst := qnext(qfirst)$
act5 : $qnext := \{qfirst\} \lhd qnext$
end

QueueRR X

Event	$Unqueue0 \mathrel{\widehat{=}}$
extends	$Unqueue0$
any	
qid	
when	
grd1 :	$qid \in queuetokens$
grd2 :	$qsize = 1$
grd3:	$deleting = FALSE$
then	
act1 :	$queueitems := \{qid\} \lhd queueitems$
act2 :	$queuetokens := queuetokens \setminus \{qid\}$
act3 :	$qsize := qsize - 1$
end	

QueueRR XI

Event $\mathit{Unqueue1} \mathrel{\widehat{=}}$
extends $\mathit{Unqueue1}$
any
qid
when
grd1 : $qid \in queuetokens$
grd2 : $qsize > 1$
grd3 : $qid = qfirst$
grd4: $deleting = FALSE$
then
act1 : $queueitems := \{qid\} \lhd queueitems$
act2 : $queuetokens := queuetokens \setminus \{qid\}$
act3 : $qsize := qsize - 1$

QueueRR XII

$\texttt{act4}: \; qfirst := qnext(qid)$

$\texttt{act5}: \; qnext := \{qid\} \lhd qnext$

end

QueueRR XIII

Event $Unqueue2 \mathrel{\widehat{=}}$

refines $Unqueue2$

when

$grd1$: $deleting = TRUE$

$grd2$: $qnext(qprev) = qidv$

$grd3$: $qlast = qidv$

with

$\mathtt{qid}$: $qid = qidv$

then

$act1$: $queueitems := \{qidv\} \lhd queueitems$

$act2$: $queuetokens := queuetokens \setminus \{qidv\}$

$act3$: $qsize := qsize - 1$

$act4$: $qlast := qprev$

QueueRR XIV

$act5$: $qnext := qnext \rhd \{qidv\}$

$act6$: $deleting := FALSE$

end

QueueRR XV

Event $Unqueue3 \widehat{=}$

refines $Unqueue3$

when

grd1: $deleting = TRUE$

grd2: $qnext(qprev) = qidv$

grd3: $qidv \neq qlast$

with

qid : $qid = qidv$

then

act1: $queueitems := \{qidv\} \lhd queueitems$

act2: $queuetokens := queuetokens \setminus \{qidv\}$

act3: $qsize := qsize - 1$

act4: $qnext(qprev) := qnext(qidv)$

act5: $deleting := FALSE$

end

QueueRR XVI

Event $Unqueuel \mathrel{\widehat{=}}$

any

qid

when

$grd1: \quad qid \in queuetokens$

$grd2: \quad qsize > 1$

$grd3: \quad qfirst \neq qid$

$grd4: \quad deleting = FALSE$

then

$act1: \quad qprev := qfirst$

$act2: \quad qidv := qid$

$act3: \quad deleting := TRUE$

end

QueueRR XVII

Event	$UnqueueS \triangleq$
Status	convergent
when	
$grd1$:	$deleting = TRUE$
$grd2$:	$qnext(qprev) \neq qidv$
then	
$act1$:	$qprev := qnext(qprev)$
end	

QueueRR XVIII

VARIANT

```
iclosure(qnext)[{qprev}]
```

END

Notes on the Variant

The variant for the search event is the set of remaining items in the queue from the current item pointed to by *prev*. Clearly we expect that the number of remaining items in that set is finite and decreasing. The set of items is obtained by applying *closure(qnext)* to *prev*.

Notes on the Variant

The variant for the search event is the set of remaining items in the queue from the current item pointed to by *prev*. Clearly we expect that the number of remaining items in that set is finite and decreasing. The set of items is obtained by applying *closure(qnext)* to *prev*.