System Modelling and Design

Modelling a Queue
Towards Implementation

Revision: 1.6, April 28, 2010

Ken Robinson

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

October 6, 2010

© Ken Robinson 2005-2010

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

![Queue Diagram]

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.
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: \(\text{finite}(\text{TOKEN})\)
axm2: \(\text{finite}(\text{ITEM})\)
axm3: \(\text{MAXQUEUE} \in \mathbb{N}_1\)
axm4: \(\text{QUEUE} \subseteq 1 .. \text{MAXQUEUE} \Rightarrow \text{TOKEN}\)
axm5: \(\text{finite}(\text{QUEUE})\)
axm6: \(\text{QUEUE} = \{q | q \in 1 .. \text{card}(q) \Rightarrow \text{TOKEN}\}\)

thm1: \(\forall q \cdot q \in \text{QUEUE} \Rightarrow \text{dom}(q) = 1 .. \text{card}(q)\)

thm1: \(\emptyset \in \text{QUEUE}\)

END
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

\begin{align*}
inv1: & \quad \text{queuetokens} \subseteq \text{TOKEN} \\
inv2: & \quad \text{queue} \in \text{QUEUE} \\
inv3: & \quad \text{qsize} \in \mathbb{N} \\
inv4: & \quad \text{queue} \in 1 .. \text{qsize} \hookrightarrow \text{queuetokens} \\
inv5: & \quad \forall i, j \cdot i \in \text{dom} (\text{queue}) \land j \in \text{dom} (\text{queue}) \land i \neq j \\
& \quad \Rightarrow \quad \text{queue}(i) \neq \text{queue}(j) \\
inv6: & \quad \text{queuetokens} = \text{ran} (\text{queue}) \\
inv7: & \quad \text{queueitems} \in \text{queuetokens} \rightarrow \text{ITEM} \\
inv8: & \quad \text{card} (\text{queue}) = \text{qsize} \\
inv9: & \quad \text{queue}^{-1} \in \text{queuetokens} \rightarrow 1 .. \text{qsize} \\
\text{thm1}: & \quad \text{queuetokens} \neq \emptyset \iff \text{qsize} \neq 0
\end{align*}
EVENTS

Initialisation

begin

act1:  

act2:  

act3:  

act4:  

end
The Queue

A machine

Event

\( Enqueue \triangleq \)

any

\( item \)

\( qid \)

when

\( grd1: \) \( item \in ITEM \)

\( grd2: \) \( qid \in TOKEN \setminus \text{queuetokens} \)

then

\( act1: \) \( \text{queuetokens} := \text{queuetokens} \cup \{qid\} \)

\( act2: \) \( \text{queue}(qsize + 1) := qid \)

\( act3: \) \( \text{queueitems}(qid) := item \)

\( act4: \) \( qsize := qsize + 1 \)

end
The Queue A machine V

Event  $\text{Deque}ue \triangleq$

when

$\text{grd}1: \; \text{qsize} \neq 0$

then

$act1:$

$\begin{align*}
\text{queue} & : |\text{queue}' \in \text{QUEUE} \\
\land \text{queue}' & \in 1 .. \text{qsize} - 1 \Rightarrow \text{queuetokens} \setminus \{\text{queue}(1)\} \\
\land (\forall i \cdot i \in 1 .. \text{qsize} - 1 \Rightarrow \text{queue}'(i) = \text{queue}(i + 1))
\end{align*}$

$act2: \; \text{queueitems} := \{\text{queue}(1)\} \leftarrow \text{queueitems}$

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

$act4: \; \text{qsize} := \text{qsize} - 1$

end
The Queue A machine VI

Event  $Unqueue \triangleq$

any
$qid$
when
$grd1$: $qid \in \text{queuetokens}$
$grd2$: $qsize \neq 0$
The QueueA machine VII

then

act1:

\[\begin{align*}
\text{queue} & : |\text{queue}' \in 1..(\text{qsize} - 1) \mapsto \text{queuetokens} \setminus \{\text{qid}\} \\
\wedge (\text{qsize} = 1 \Rightarrow \text{queue}' = \emptyset) \\
\wedge (\text{qsize} > 1 \Rightarrow \\
(\forall i \cdot i \in 1..\text{queue}^{-1}(\text{qid}) - 1 \Rightarrow \text{queue}'(i) = \text{queue}(i)) \\
\wedge \\
(\forall j \cdot j \in \text{queue}^{-1}(\text{qid}) + 1..\text{qsize} \\
\Rightarrow \text{queue}'(j - 1) = \text{queue}(j)))
\end{align*}\]

act2: \(\text{queueitems} := \{\text{qid}\} \leftarrow \text{queueitems}\)
act3: \(\text{queuetokens} := \text{queuetokens} \setminus \{\text{qid}\}\)
act4: \(\text{qsize} := \text{qsize} - 1\)

end

END
Next we will define a QueueType with queue operations.

**CONTEXT** QueueType

**EXTENDS** QueueContext

**CONSTANTS**
- ENQUEUE
- DEQUEUE
- DELETE

**AXIOMS**

**axm1:** \( ENQUEUE \in \text{QUEUE} \times \text{TOKEN} \rightarrow \\
\text{QUEUE} \)

**axm2:** \( \forall q, t \cdot q \in \text{QUEUE} \land t \notin \text{ran}(q) \rightarrow \)
\( q \leftrightarrow t \in \text{dom}(\text{ENQUEUE}) \)

**axm3:** \( \forall q, t \cdot q \in \text{QUEUE} \land t \notin \text{ran}(q) \rightarrow \)
\( \text{ENQUEUE}(q \leftrightarrow t) = q \leftrightarrow \{ \text{card}(q) + 1 \leftrightarrow t \} \)
axm4: \[ \forall q, t \cdot q \in \text{QUEUE} \land t \in \text{TOKEN} \land t \notin \text{ran}(q) \Rightarrow \text{card(ENQUEUE}(q \mapsto t)) = \text{card}(q) + 1 \]

axm5: \[ \forall q, t \cdot q \in \text{QUEUE} \land t \in \text{TOKEN} \land t \notin \text{ran}(q) \Rightarrow \text{dom(ENQUEUE}(q \mapsto t)) = 1 \ldots \text{card}(q) + 1 \]

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

axm7: \[ \text{DEQUEUE} \in \text{QUEUE} \mapsto \text{QUEUE} \]

axm8: \[ \text{dom(DEQUEUE)} = \text{QUEUE \setminus \{\emptyset\}} \]

axm9: \[ \forall q \cdot q \in \text{QUEUE} \land q \neq \emptyset \Rightarrow \text{DEQUEUE}(q) \in 1 \ldots \text{card}(q) - 1 \Rightarrow \text{ran}(q) \setminus \{q(1)\} \]

axm10: \[ \forall q \cdot q \in \text{dom(DEQUEUE)} \Rightarrow \text{card(DEQUEUE}(q)) = \text{card}(q) - 1 \]
axm11: \( \forall q \cdot q \in \text{dom}(\text{DEQUEUE}) \Rightarrow \text{dom}(\text{DEQUEUE}(q)) = 1 \ldots \text{card}(q) - 1 \)

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

axm13: \( \text{DELETE} \in \text{QUEUE} \times \mathbb{N}_1 \mapsto \text{QUEUE} \)

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

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

axm16: 
\( \forall q, i \cdot q \mapsto i \in \text{dom}(\text{DELETE}) \Rightarrow \text{dom}(\text{DELETE}(q \mapsto i)) = 1 \ldots \text{card}(q) - 1 \)
axm17:

\[ \forall q, i, j. q \mapsto i \in \text{dom}(\text{DELETE}) \Rightarrow (j < i \land j \in \text{dom}(q) \Rightarrow \text{DELETE}(q \mapsto i)(j) = q(j)) \land (j \geq i \land j + 1 \in \text{dom}(q) \Rightarrow \text{DELETE}(q \mapsto i)(j) = q(j + 1)) \]
The QueueB machine I

Then refine QueueA to QueueB using QueueType

MACHINE QueueB
REFINES QueueA
SEES QueueType

VARIABLES
queue_tokens tokens for currently queued items
queue the queue of tokens
queue_items a function for fetching the item associated with a token
qsize current size of queue
The QueueB machine II

INVARINTS

inv1: queuetokens ⊆ TOKEN
inv2: queue ∈ QUEUE
inv3: qsize = card(queue)
inv4: queue ∈ 1 .. qsize ↦→ queuetokens
inv5: ∀ i, j · i ∈ dom(queue) ∧ j ∈ dom(queue) ∧ i ≠ j
⇒
    queue(i) ≠ queue(j)
inv6: queuetokens = ran(queue)
inv7: queueitems ∈ queuetokens → ITEM
inv8: queue⁻¹ ∈ queuetokens ↦→ 1 .. qsize
inv9:
(∀ qid · qid ∈ TOKEN \ queuetokens
⇒
    ENQUEUE(queue ↦→ qid) = queue ⇐ {qsize + 1 ↦→ qid})
The QueueB machine III

inv10: \( \forall \text{qid} \cdot \text{qid} \in \text{queuetokens} \Rightarrow \text{queue} \mapsto \text{queue}^{-1}(\text{qid}) \in \text{dom}(\text{DELETE}) \)

inv11:

\( qsize \neq 1 \Rightarrow \left( \forall \text{qid}, \ i \cdot \text{qid} \in \text{queuetokens} \land i \in 1 \ldots (\text{queue}^{-1}(\text{qid}) - 1) \Rightarrow \left( \text{DELETE}(\text{queue} \mapsto \text{queue}^{-1}(\text{qid}))) (i) = \text{queue}(i) \right) \)

inv12:

\( qsize \neq 1 \Rightarrow \left( \forall \text{qid}, \ i \cdot \text{qid} \in \text{queuetokens} \land i \in \text{queue}^{-1}(\text{qid}) + 1 \ldots \text{qsize} \Rightarrow \left( \text{DELETE}(\text{queue} \mapsto \text{queue}^{-1}(\text{qid}))) (i - 1) = \text{queue}(i) \right) \)
The QueueB machine IV

inv13: \( \forall qid \cdot qid \in queuetokens \Rightarrow queue^{-1}(qid) \leq qsize \)
The QueueB machine V

EVENTS
Initialisation
begin
act1:  queuetokens := ∅
act2:  queue := ∅
act3:  qsize := 0
act4:  queueitems := ∅
end
The QueueB machine VI

Event $Enqueue \triangleleft$
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 \( Dequeue \) \( \triangleleft \)
refines \( Dequeue \)
when
\[ \text{grd1: } \text{qsize} \neq 0 \]
then
\[ \text{act1: } \text{queue} := \text{DEQUEUE}(\text{queue}) \]
\[ \text{act2: } \text{queueitems} := \{\text{queue}(1)\} \triangleleft \text{queueitems} \]
\[ \text{act3: } \text{queuetokens} := \text{queuetokens} \setminus \{\text{queue}(1)\} \]
\[ \text{act4: } \text{qsize} := \text{qsize} - 1 \]
end
The QueueB machine VIII

Event  Unqueue ≡
refines  Unqueue

any
qid
when

grd1:  qid ∈ queuetokens

then

act1:  queue := DELETE(queue ↦ queue⁻¹(qid))
act2:  queueitems := {qid} ≪ queueitems
act3:  queuetokens := queuetokens \ {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 *queue_tokens* 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 \textit{queuetokens} are organised as a list using the following variables:

- \texttt{qfirst} the first element of the list;
- \texttt{qlast} the last element of the list;
- \texttt{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;
- \texttt{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

\[
\begin{align*}
q_{first} & \text{ gives the identity of the first item in the list} \\
qnext(q_{first}) & \text{ gives the identity of the second item in the list} \\
(qnext ; qnext)(q_{first}) & \text{ gives the identity of the third item in the list} \\
\ldots & \text{ etc}
\end{align*}
\]

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 describe 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

\[
\begin{align*}
q_{\text{first}} & \quad \text{gives the identity of the first item in the list} \\
qnext(q_{\text{first}}) & \quad \text{gives the identity of the second item in the list} \\
(qnext \circ qnext)(q_{\text{first}}) & \quad \text{gives the identity of the third item in the list} \\
\vdots & \quad \text{etc}
\end{align*}
\]

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

If \( r \in X \leftrightarrow X \), then \( r^0 = id(X) \) and \( r^{n+1} = r^n \circ r \).
Relational composition and iteration

Since we are modelling a list structure we will use relational composition on the \( \text{qnext} \) function to describes paths along the list, and we will use transitive closure of \( \text{qnext} \) to describe reachability.

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

\[
\begin{align*}
\text{qfirst} & \quad \text{gives the identity of the first item in the list} \\
\text{qnext(qfirst)} & \quad \text{gives the identity of the second item in the list} \\
(qnext \circ qnext)(qfirst) & \quad \text{gives the identity of the third item in the list} \\
\ldots & \quad \text{etc}
\end{align*}
\]

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

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

\[
\begin{align*}
q\text{first} & \quad \text{gives the identity of the first item in the list} \\
q\text{next}(q\text{first}) & \quad \text{gives the identity of the second item in the list} \\
(q\text{next} \circ q\text{next})(q\text{first}) & \quad \text{gives the identity of the third item in the list} \\
\vdots & \\
\end{align*}
\]

etc

Multiple composition is expressed by iteration: $q\text{next}^n$ (provided by the constant function $\text{iterate}(q\text{next} \mapsto n)$), is the result of composing $q\text{next}$ with itself $n$ times.

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

Since we are modelling a list structure we will use *relational composition* on the \( q\text{\textit{next}} \) function to describes paths along the list, and we will use transitive *closure of \( q\text{\textit{next}} \) to describe reachabilility.

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

\[
\begin{align*}
q\text{\textit{first}} & \quad \text{gives the identity of the first item in the list} \\
q\text{\textit{next}}(q\text{\textit{first}}) & \quad \text{gives the identity of the second item in the list} \\
(q\text{\textit{next}} ; q\text{\textit{next}})(q\text{\textit{first}}) & \quad \text{gives the identity of the third item in the list} \\
\ldots & \\
\text{etc}
\end{align*}
\]

Multiple composition is expressed by *iteration*: \( q\text{\textit{next}}^n \) (provided by the constant function \( \text{iterate}(q\text{\textit{next}} \mapsto n) \)), is the result of composing \( q\text{\textit{next}} \) 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 \( q_{next} \) function to describes paths along the list, and we will use transitive *closure* of \( q_{next} \) to describe reachability.

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

- \( q_{first} \) gives the identity of the first item in the list
- \( q_{next}(q_{first}) \) gives the identity of the second item in the list
- \( (q_{next} ; q_{next})(q_{first}) \) gives the identity of the third item in the list
- etc

Multiple composition is expressed by *iteration*: \( q_{next}^n \) (provided by the constant function \( iterate(q_{next} \mapsto n) \)), is the result of composing \( q_{next} \) 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 describe 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

\[
\begin{align*}
q_{first} & \quad \text{gives the identity of the first item in the list} \\
q_{next}(q_{first}) & \quad \text{gives the identity of the second item in the list} \\
(q_{next} ; q_{next})(q_{first}) & \quad \text{gives the identity of the third item in the list} \\
\vdots & \quad \text{etc}
\end{align*}
\]

Multiple composition is expressed by iteration: \( q_{next}^n \) (provided by the constant function \( \text{iterate}(q_{next} \mapsto n) \)), is the result of composing \( q_{next} \) with itself \( n \) times.

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

- \( q\text{first} \) gives the identity of the first item in the list
- \( q\text{next}(q\text{first}) \) gives the identity of the second item in the list
- \( (q\text{next} \circ q\text{next})(q\text{first}) \) gives the identity of the third item in the list
  
  etc

Multiple composition is expressed by *iteration*: \( q\text{next}^n \) (provided by the constant function \( \text{iterate}(q\text{next} \mapsto n) \)), is the result of composing \( q\text{next} \) with itself \( n \) times.

If \( r \in X \leftrightarrow X \), then \( r^0 = \text{id}(X) \) and \( r^{n+1} = r^n \circ r \).
Relational composition and iteration

Since we are modelling a list structure we will use relational composition on the \( q\text{next} \) function to describes paths along the list, and we will use transitive closure of \( q\text{next} \) to describe reachability.

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

\[
\begin{align*}
q\text{first} & \quad \text{gives the identity of the first item in the list} \\
q\text{next}(q\text{first}) & \quad \text{gives the identity of the second item in the list} \\
(q\text{next} ; q\text{next})(q\text{first}) & \quad \text{gives the identity of the third item in the list} \\
\ldots & \quad \text{etc}
\end{align*}
\]

Multiple composition is expressed by iteration: \( q\text{next}^n \) (provided by the constant function \( \text{iterate}(q\text{next} \mapsto n) \)), is the result of composing \( q\text{next} \) with itself \( n \) times.

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

\dagger 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

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

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 \textit{closure}; it has to be defined as a constant function.

\[\dagger\text{It should be clear that continuous composition of a relation with itself will eventually reach a stationary relation.}\]
Relational composition of functions

It should be clear that if $f$ is a function then $f \circ f$ 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, $q_{next}^n$ is an injective function that gives all paths of length $n$ within the list.

$q_{next}^+$ 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 $q_{next}^+\{q_{first}\}$, the image of the first node in the list under $q_{next}^+$, is the set of all nodes in the list.
It should be clear that if $f$ is a function then $f \circ f$ 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 $f \circ f$ 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, $q_{next}^n$ is an injective function that gives all paths of length $n$ within the list.

$q_{next}^+$ 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 $q_{next}^+[\{q_{first}\}]$, the image of the first node in the list under $q_{next}^+$, 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, $q_{next}^n$ is an injective function that gives all paths of length $n$ within the list.

$q_{next}^+$ 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 $q_{next}^+ [{q_{first}}]$, the image of the first node in the list under $q_{next}^+$, is the set of all nodes in the list.
Relational composition of functions

It should be clear that if $f$ is a function then $f \circ f$ 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:
\[ \text{iterate} \in (TOKEN \leftrightarrow TOKEN) \times \mathbb{N} \rightarrow (TOKEN \leftrightarrow TOKEN) \]

axm2:
\[ \forall r \cdot r \in TOKEN \leftrightarrow TOKEN \Rightarrow \text{iterate}(r \mapsto 0) = \text{dom}(r) \triangleleft \text{id} \]

axm3:
\[ \forall r, n \cdot r \in TOKEN \leftrightarrow TOKEN \land n \in \mathbb{N}_1 \]
\[ \Rightarrow \]
\[ \text{iterate}(r \mapsto n) = \text{iterate}(r \mapsto n - 1); r \]

axm4:
\[ \forall s \cdot s \subseteq \mathbb{N} \land 0 \in s \land (\forall n \cdot n \in s \Rightarrow n + 1 \in s) \Rightarrow \mathbb{N} \subseteq s \]
axm5: \( \forall r, n \cdot r \in TOKEN \leftrightarrow \text{TOKEN} \land n \in \mathbb{N}_1 \Rightarrow \)  
\[ \text{dom}(\text{iterate}(r \mapsto n)) \subseteq \text{dom}(r) \]

axm6: \( \forall r, n \cdot r \in TOKEN \leftrightarrow \text{TOKEN} \land n \in \mathbb{N}_1 \Rightarrow \)  
\[ \text{ran}(\text{iterate}(r \mapsto n)) \subseteq \text{ran}(r) \]
axm7: 
\[ iclosure \in \{ \text{TOKEN} \leftrightarrow \text{TOKEN} \} \rightarrow \{ \text{TOKEN} \leftrightarrow \text{TOKEN} \} \]

axm8: 
\[ \forall r. r \in \text{TOKEN} \leftrightarrow \text{TOKEN} \]
\[ \Rightarrow \]
\[ iclosure(r) = (\bigcup n. n \in \mathbb{N}_1 \mid iterate(r \mapsto n)) \]

axm9: 
\[ \forall r. r \in \text{TOKEN} \leftrightarrow \text{TOKEN} \]
\[ \Rightarrow \]
\[ \text{dom}(iclosure(r)) \subseteq \text{dom}(r) \]

END
The invariant of QueueR

The list consists of the elements of $\textit{queuetokens}$ hence

$$\textit{qsize} = \text{card}(\textit{queuetokens})$$

For non-empty lists, $\textit{qfirst}$ and $\textit{qlast}$ are elements of $\textit{queuetokens}$

$$\textit{queuetokens} \neq \emptyset \implies \textit{qfirst} \in \textit{queuetokens}$$
$$\textit{queuetokens} \neq \emptyset \implies \textit{qlast} \in \textit{queuetokens}$$

The list is linear and connected, hence $\textit{qnext}$ is injective, but it is also surjective and therefore bijective:

$$\textit{qnext} \in \textit{queuetokens} \setminus \{\textit{qlast}\} \mapsto \textit{queuetokens} \setminus \{\textit{qfirst}\}$$
The invariant of QueueR

The list consists of the elements of \( \text{queuetokens} \) hence

\[
qsize = \text{card}(\text{queuetokens})
\]

For non-empty lists, \( qfirst \) and \( qlast \) are elements of \( \text{queuetokens} \)

\[
\text{queuetokens} \neq \emptyset \implies qfirst \in \text{queuetokens} \\
\text{queuetokens} \neq \emptyset \implies qlast \in \text{queuetokens}
\]

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

\[
qnext \in \text{queuetokens} \setminus \{qlast\} \implies \text{queuetokens} \setminus \{qfirst\}
\]
The invariant of QueueR

The list consists of the elements of $\text{queuetokens}$ hence

$qsize = \text{card}(\text{queuetokens})$

For non-empty lists, $qfirst$ and $qlast$ are elements of $\text{queuetokens}$

\[
\begin{align*}
\text{queuetokens} \neq \emptyset & \implies qfirst \in \text{queuetokens} \\
\text{queuetokens} \neq \emptyset & \implies qlast \in \text{queuetokens}
\end{align*}
\]

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

$qnext \in \text{queuetokens} \setminus \{qlast\} \mapsto \text{queuetokens} \setminus \{qfirst\}$
The invariant of QueueR

The list consists of the elements of $\text{queuetokens}$ hence

$$qsize = \text{card}(\text{queuetokens})$$

For non-empty lists, $\text{qfirst}$ and $\text{qlast}$ are elements of $\text{queuetokens}$

$$\text{queuetokens} \neq \emptyset \implies \text{qfirst} \in \text{queuetokens}$$
$$\text{queuetokens} \neq \emptyset \implies \text{qlast} \in \text{queuetokens}$$

The list is linear and connected, hence $\text{qnext}$ is injective, but it is also surjective and therefore bijective:

$$\text{qnext} \in \text{queuetokens} \setminus \{\text{qlast}\} \mapsto \text{queuetokens} \setminus \{\text{qfirst}\}$$
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 \( \text{dom}(qnext) \)

   \[ \forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qlast \Rightarrow t \in \text{dom}(qnext) \]

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

   \[ \forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qfirst \Rightarrow t \in \text{ran}(qnext) \]

3. Following all sequences of \( qnext \) from \( qfirst \) should give all tokens in \( \text{queuetokens} \)

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

4. The following should also follow from the refinement relation:

   \( qsize \neq 0 \Rightarrow \text{queue}(1) = qfirst \)

   \( qsize \neq 0 \Rightarrow \text{queue}(qsize) = qlast \)

   \( qsize \neq 0 \Rightarrow \forall i \cdot i \in 1 .. qsize - 1 \Rightarrow \text{queue}(i + 1) = qnext(\text{queue}(i)) \)
QueueR Theorems

The following should follow from the invariant:

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

\[
\forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qlast \Rightarrow t \in \text{dom}(qnext)
\]

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

\[
\forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qfirst \Rightarrow t \in \text{ran}(qnext)
\]

3. Following all sequences of \( qnext \) from \( qfirst \) should give all tokens in \( \text{queuetokens} \)

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

4. The following should also follow from the refinement relation:

\[
\begin{align*}
qsize \neq 0 & \Rightarrow \text{queue}(1) = qfirst \\
qsize \neq 0 & \Rightarrow \text{queue}(qsize) = qlast \\
qsize \neq 0 & \Rightarrow \forall i \cdot i \in 1..qsize - 1 \Rightarrow \text{queue}(i + 1) = qnext(\text{queue}(i))
\end{align*}
\]
QueueR Theorems

The following should follow from the invariant:

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

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

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

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

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

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

4. The following should also follow from the refinement relation:

$$qsize \neq 0 \Rightarrow \text{queue}(1) = qfirst$$
$$qsize \neq 0 \Rightarrow \text{queue}(qsize) = qlast$$
$$qsize \neq 0 \Rightarrow \forall i \cdot i \in 1..qsize - 1 \Rightarrow \text{queue}(i + 1) = qnext(\text{queue}(i))$$
QueueR Theorems

The following should follow from the invariant:

1. Any element of the list that is not qfirst must be in dom(qnext)

\[ \forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qlast \Rightarrow t \in \text{dom}(qnext) \]

2. Any element of the list that is not qlast must be in ran(qnext)

\[ \forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qfirst \Rightarrow t \in \text{ran}(qnext) \]

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

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

4. The following should also follow from the refinement relation:

\[ qsize \neq 0 \Rightarrow \text{queue}(1) = qfirst \]
\[ qsize \neq 0 \Rightarrow \text{queue}(qsize) = qlast \]
\[ qsize \neq 0 \Rightarrow \forall i \cdot i \in 1 .. qsize - 1 \Rightarrow \text{queue}(i + 1) = qnext(\text{queue}(i)) \]
QueueR Theorems

The following should follow from the invariant:

1. Any element of the list that is not \( qfirst \) must be in \( \text{dom}(qnext) \)
   \[
   \forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qlast \Rightarrow t \in \text{dom}(qnext)
   \]

2. Any element of the list that is not \( qlast \) must be in \( \text{ran}(qnext) \)
   \[
   \forall t \cdot t \in \text{queuetokens} \land qsize > 1 \land t \neq qfirst \Rightarrow t \in \text{ran}(qnext)
   \]

3. Following all sequences of \( qnext \) from \( qfirst \) should give all tokens in \( \text{queuetokens} \)
   \[
   \text{closure1}(qnext)[\{qfirst\}] = \text{queuetokens}
   \]

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

\[
qnext^+ \cap id(queuetokens) = \emptyset
\]

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:

$$qnext^+ \cap id(queuetokens) = \emptyset$$

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

INVARINTS
inv1:       \( qfirst \in TOKEN \)
inv2:       \( qlast \in TOKEN \)
inv3:       \( qsize \neq 0 \Rightarrow qfirst = \text{queue}(1) \)
The QueueR machine II

inv4: \( qsize \neq 0 \Rightarrow qlast = \text{queue}(qsize) \)

inv5: \( qnext \in \text{queuetokens} \Rightarrow \text{queuetokens} \)

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

inv7: \( qnext \cap id = \emptyset \)

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

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

inv12: \( qsize \geq 1 \Rightarrow (\forall n \cdot n \in 1..qsize - 1 \land \text{iterate}(qnext \mapsto n - 1)[\{qfirst\}] = \{\text{queue}(n)\} \Rightarrow \text{iterate}(qnext \mapsto n)[\{qfirst\}] = \{\text{queue}(n + 1)\}) \)
The QueueR machine III

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

**inv14:** \( qsize \geq 1 \Rightarrow iclosure(qnext)[\{qfirst\}] = \text{queuetokens} \)
The QueueR machine IV

EVENTS

Initialisation

begin
  \textit{act1:} \quad \textit{queuetokens} := \emptyset
  \textit{act2:} \quad \textit{qsize} := 0
  \textit{act3:} \quad \textit{queueitems} := \emptyset
  \textit{act4:} \quad \textit{qfirst} \in \textit{TOKEN}
  \textit{act5:} \quad \textit{qlast} \in \textit{TOKEN}
  \textit{act6:} \quad \textit{qnext} := \emptyset
end
The QueueR machine V

Event $Enqueue_0 \triangleq$
refines $Enqueue$

any
item
qid
when

$grd1$: \hspace{1em} item \in ITEM
$grd2$: \hspace{1em} qid \in TOKEN \setminus \text{queuetokens}
$grd3$: \hspace{1em} qsize = 0

then

$act1$: \hspace{1em} \text{queuetokens} := \text{queuetokens} \cup \{qid\}
$act2$: \hspace{1em} \text{queueitems}(qid) := item
$act3$: \hspace{1em} qsize := qsize + 1
$act4$: \hspace{1em} qfirst := qid
$act5$: \hspace{1em} qlast := qid

end
The QueueR machine VI

Event $\text{Enqueue1} \sqsupset$

refines $\text{Enqueue}$

any

item

$qid$

when

$\text{grd1}$: $\text{item} \in \text{ITEM}$

$\text{grd2}$: $\text{qid} \in \text{TOKEN} \setminus \text{queuetokens}$

$\text{grd3}$: $\text{qsize} \neq 0$

then

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

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

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

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

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

end
The QueueR machine VII

Event \( Dequeue0 \cong \)
refines \( Dequeue \)
when
\( grd1: \ qsize = 1 \)
then
\( act1: \ qsize := qsize - 1 \)
\( act2: \ queuetokens := queuetokens \setminus \{qfirst\} \)
\( act3: \ queueitems := \{qfirst\} \triangleleft queueitems \)
\( act4: \ qnext := \{qfirst\} \triangleleft qnext \)
end
Event \texttt{Dequeuel} \cong

refines \texttt{Dequeuel}

when

\texttt{grd1}: \text{qsize} > 1

then

\texttt{act1}: \text{qsize} := \text{qsize} - 1

\texttt{act2}: \text{queuetokens} := \text{queuetokens} \setminus \{\text{qfirst}\}

\texttt{act3}: \text{queueitems} := \{\text{qfirst}\} \cup \text{queueitems}

\texttt{act4}: \text{qfirst} := \text{qnext}(\text{qfirst})

\texttt{act5}: \text{qnext} := \{\text{qfirst}\} \cup \text{qnext}

end
Event $Unqueue0 \equiv$
refines $Unqueue$
a
ty $qid$
when
$grd1$: $qid \in queuetokens$
$grd2$: $qsize = 1$
then
$act1$: $queueitems := \{qid\} \triangleleft queueitems$
$act2$: $queuetokens := queuetokens \setminus \{qid\}$
$act3$: $qsize := qsize - 1$
end
The QueueR machine X

Event Unqueue1 ≜
refines Unqueue

any

qid

when

grd1: qid ∈ queuetokens
grd2: qsize > 1
grd3: qid = qfirst

then

act1: queueitems := \{qid\} \cup queueitems
act2: queuetokens := queuetokens \ \{qid\}
act3: qsize := qsize − 1
act4: qfirst := qnext(qid)
act5: qnext := \{qid\} \cup qnext

end
The QueueR machine XI

Event $Unqueue_2 \equiv$

refines $Unqueue$

any $qid$

when

$grd_1$: $qid \in queuetokens$
$grd_2$: $qsize > 1$
$grd_3$: $qlast = qid$

then

$act_1$: $queueitems := \{qid\} \leftarrow queueitems$
$act_2$: $queuetokens := queuetokens \setminus \{qid\}$
$act_3$: $qsize := qsize - 1$
$act_4$: $qlast := qnext^{-1}(qid)$
$act_5$: $qnext := qnext \triangleright \{qid\}$

end
The QueueR machine XII

Event $Unqueue3 \equiv$

refines $Unqueue$

any $qid$

when

$grd1$: $qid \in queuetokens$

$grd2$: $qsize > 1$

$grd3$: $qfirst \neq qid$

$grd4$: $qlast \neq qid$

then

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

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

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

$act4$: $qnext(qnext^{-1}(qid)) := qnext(qid)$

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, i.e implement qprev.
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.
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 Unqueue3 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 $q_{prev}$. This event sets $q_{prev}$ to $q_{first}$ and sets a flag, $deleting$, to $TRUE$.

- **Unqueue3M**: an event that represents *still searching*. It advances $q_{prev}$ to $q_{next}(q_{prev})$.

- **Unqueue3F**: the final step. The item to be deleted has been found, so the current value of $q_{prev}$ 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 *Unqueue3* 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 := ∅
act2: qsize := 0
act3: queueitems := ∅
act4: qfirst ∈ TOKEN
act5: qlast ∈ TOKEN
act6: qnext := ∅

act7: deleting := FALSE
act8: qprev ∈ TOKEN
act9: qidv ∈ TOKEN
QueueRR IV

end

Event \( Enqueue0 \triangleq \)
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

\begin{itemize}
  \item \texttt{act2} : \texttt{queueitems}(qid) := item
  \item \texttt{act3} : \texttt{qsize} := \texttt{qsize} + 1
  \item \texttt{act4} : \texttt{qfirst} := qid
  \item \texttt{act5} : \texttt{qlast} := qid
\end{itemize}

end
QueueRR VI

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

act3: \( qsize := qsize + 1 \)
act4: \( qnext(qlast) := qid \)
act5: \( qlast := qid \)

end
Event $Deque0$ extends $Deque0$

when

$grd1$ : $qsize = 1$
$grd2$ : $deleting = FALSE$

then

$act1$ : $qsize := qsize - 1$
$act2$ : $queuetokens := queuetokens \ \{qfirst\}$
$act3$ : $queueitems := \{qfirst\} \leftarrow queueitems$
$act4$ : $qnext := \{qfirst\} \leftarrow qnext$

end
Dequeue1 \equiv
extends Dequeue1

when
grd1: qsize > 1
grd2: deleting = FALSE

then

act1: qsize := qsize − 1
act2: queuetokens := queuetokens \ {qfirst}
act3: queueitems := {qfirst} \ queueitems
act4: qfirst := qnext(qfirst)
act5: qnext := {qfirst} \ qnext

end
QueueRR X

Event  $Unqueue0 \triangleq$

extends  $Unqueue0$

any

$qid$

when

$grd1: qid \in queuetokens$

$grd2: qsize = 1$

$grd3: deleting = FALSE$

then

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

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

$act3: qsize := qsize - 1$

end
QueueRR XI

Event $Unqueue1 \equiv$

extends $Unqueue1$

any

$qid$

when

grd1: $qid \in queuetokens$

grd2: $qsize > 1$

grd3: $qid = qfirst$

grd4: $deleting = FALSE$

then

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

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

act3: $qsize := qsize - 1$
QueueRR XII

act4: \( qfirst := qnext(qid) \)

act5: \( qnext := \{qid\} \leftarrow qnext \)

end
QueueRR XIII

Event \textit{Unqueue2} \supseteq \textit{Unqueue2}

refines \textit{Unqueue2}

when

\textit{grd1}: deleting = \textit{TRUE}

\textit{grd2}: qnext(qprev) = qidv

\textit{grd3}: qlast = qidv

with

\textit{qid}: qid = qidv

then

\textit{act1}: queueitems := \{qidv\} \leftarrow queueitems

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

\textit{act3}: qsize := qsize - 1

\textit{act4}: qlast := qprev
QueueRR XIV

act5: \[ q_{next} := q_{next} \triangleright \{ q_{idv} \} \]

act6: \[ deleting := \text{FALSE} \]

end
QueueRR XV

Event \( Unqueue3 \equiv \)

refines \( Unqueue3 \)

when

\( grd1: \) deleting = TRUE
\( grd2: \) qnext(qprev) = qidv
\( grd3: \) qidv \( \neq \) qlast

with

\( qid: \) qid = qidv

then

\( act1: \) queueitems := \{qidv\} \( \triangleleft \) queueitems
\( act2: \) queuetokens := queuetokens \( \setminus \) \{qidv\}
\( act3: \) qsize := qsize – 1
\( act4: \) qnext(qprev) := qnext(qidv)
\( act5: \) deleting := FALSE

end
QueueRR XVI

Event  $UnqueueI \equiv$
  any $qid$
when
  $grd1$:  $qid \in queuetokens$
  $grd2$:  $qsize > 1$
  $grd3$:  $qfirst \neq qid$
  $grd4$:  $deleting = FALSE$
  then
  $act1$:  $qprev := qfirst$
  $act2$:  $qidv := qid$
  $act3$:  $deleting := TRUE$
  end
Event \( UnqueueS \triangleq \)

Status convergent

when

\( grd1: \) deleting = TRUE

\( grd2: \) \( qnext(qprev) \neq qidv \)

then

\( act1: \) \( qprev := qnext(qprev) \)

end
QueueRR XVIII

VARIANT

\[ \text{closure}(q_{\text{next}})[\{q_{\text{prev}}\}] \]

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 \textit{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 \textit{closure(qnext)} to \textit{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 $prev$. 