Outline

Outline I

1. What is Refinement?
   - What does refinement guarantee?
   - Some things you can do in refinement
   - Reducing nondeterminism: examples
   - Refinement is not equivalence
   - Refining the state
   - A function machine
   - A refinement of the function machine
Objectives of this Lecture

- to introduce the concept of refinement, both algorithmic and data refinement;
- to understand the concept of refinement both informally and formally;
- to explore a number of examples of refinement;
Objectives of this Lecture

- to introduce the concept of refinement, both algorithmic and data refinement;
- to understand the concept of refinement both informally and formally;
- to explore a number of examples of refinement;
Objectives of this Lecture

- to introduce the concept of refinement, both algorithmic and data refinement;
- to understand the concept of refinement both informally and formally;
- to explore a number of examples of refinement;
Objectives of this Lecture

- to introduce the concept of refinement, both algorithmic and data refinement;
- to understand the concept of refinement both informally and formally;
- to explore a number of examples of refinement;
**What is Refinement?**

*Refinement* is the name given to the process of transforming an *abstract* specification into a *concrete* implementation.

There are two aspects of refinement:

- **Algorithmic refinement**, in which an algorithm is transformed, and
- **Data refinement**, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term *refinement* to cover either or both.
Refinement is the name given to the process of transforming an abstract specification into a concrete implementation.

There are two aspects of refinement:

Algorithmic refinement, in which an algorithm is transformed, and
Data refinement, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term refinement to cover either or both.
Refinement is the name given to the process of transforming an abstract specification into a concrete implementation.

There are two aspects of refinement:

Algorithmic refinement, in which an algorithm is transformed, and Data refinement, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term refinement to cover either or both.
Refinement is the name given to the process of transforming an abstract specification into a concrete implementation.

There are two aspects of refinement:

Algorithmic refinement, in which an algorithm is transformed, and Data refinement, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term refinement to cover either or both.
What is Refinement?

Refinement is the name given to the process of transforming an abstract specification into a concrete implementation.

There are two aspects of refinement:

Algorithmic refinement, in which an algorithm is transformed, and Data refinement, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term refinement to cover either or both.
What is Refinement?

Refinement is the name given to the process of transforming an abstract specification into a concrete implementation.

There are two aspects of refinement:

Algorithmic refinement, in which an algorithm is transformed, and Data refinement, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term refinement to cover either or both.
What is Refinement?

*Refinement* is the name given to the process of transforming an abstract specification into a concrete implementation.

There are two aspects of refinement:

**Algorithmic refinement**, in which an algorithm is transformed, and **Data refinement**, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term *refinement* to cover either or both.
The refinement of a construct —for example an event— promises behaviour that is consistent with the behaviour of the construct being refined.

That is, *in the context*, the behaviour offered by the *refining* construct could have been offered by the *refined* construct.

Consistency must take nondeterminism into account.
The refinement of a construct —for example an event— promises behaviour that is consistent with the behaviour of the construct being refined.

That is, in the context, the behaviour offered by the refining construct could have been offered by the refined construct.

Consistency must take nondeterminism into account.
What does refinement guarantee?

The refinement of a construct —for example an event— promises behaviour that is consistent with the behaviour of the construct being refined.

That is, *in the context*, the behaviour offered by the *refining* construct could have been offered by the *refined* construct.

Consistency must take nondeterminism into account.
What does refinement guarantee?

The refinement of a construct —for example an event— promises behaviour that is consistent with the behaviour of the construct being refined.

That is, \textit{in the context}, the behaviour offered by the \textit{refining} construct could have been offered by the \textit{refined} construct.

Consistency must take nondeterminism into account.
Some things you can do in refinement

- Reduce nondeterminism: nondeterminism in a construct is interpreted as a choice in which any of the outcomes are satisfactory, so the refiner can choose between those options.
- Strengthening guards: individual guards of an event may be strengthened subject to the disjunction of all the guards remaining equivalent.
Some things you can do in refinement

- Reduce nondeterminism: nondeterminism in a construct is interpreted as a choice in which any of the outcomes are satisfactory, so the refiner can choose between those options.

- Strengthening guards: individual guards of an event may be strengthened subject to the disjunction of all the guards remaining equivalent.
Some things you can do in refinement

- Reduce nondeterminism: nondeterminism in a construct is interpreted as a choice in which any of the outcomes are satisfactory, so the refiner can choose between those options.

- Strengthening guards: individual guards of an event may be strengthened subject to the disjunction of all the guards remaining equivalent.
Reducing nondeterminism: examples

**Specification**

```
SimpleChoice
result ∈ { 3 , 6 , 9 , 12 } ;
```

**Refinement**

```
SimpleChoice
result := 6 ;
```
Reducing nondeterminism: examples

**Specification**

```
SimpleChoice

result ∈ { 3, 6, 9, 12 } ;
```

**Refinement**

```
SimpleChoice

result := 6 ;
```
Reducing nondeterminism: examples

<table>
<thead>
<tr>
<th>Specification</th>
<th>Refinement</th>
</tr>
</thead>
</table>
| **SimpleChoice**
  \[ result \in \{3, 6, 9, 12\} \] | **SimpleChoice**
  \[ result := 6 \] |
Refinement is not equivalence

It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set

\[ \text{COIN} = \{ \text{Head}, \text{Tail} \} \]

Specification

\[ \text{Flip coin} : \in \text{COIN} \]

Refinement

\[ \text{Flip coin} : \in \text{COIN} \]

It should be clear that it is reasonable for an event to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.
It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set

$$COIN = \{ \text{Head}, \text{Tail} \}$$

Specification

Flip

$$coin : \in COIN$$

Refinement

Flip

$$coin : \in COIN$$

It should be clear that it is reasonable for an event to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.
Refinement is not equivalence

It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set

\[ COIN = \{ \text{Head} , \text{Tail} \} \]

<table>
<thead>
<tr>
<th>Specification</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flip</td>
<td>Flip</td>
</tr>
<tr>
<td>( coin \in COIN )</td>
<td>( coin \in COIN )</td>
</tr>
</tbody>
</table>

It should be clear that it is reasonable for an event to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.
Refinement is not equivalence

It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set

\[
COIN = \{ \text{Head, Tail} \}
\]

<table>
<thead>
<tr>
<th>Specification</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flip</td>
<td>Flip</td>
</tr>
<tr>
<td>( coin \in COIN )</td>
<td>( coin \in COIN )</td>
</tr>
</tbody>
</table>

It should be clear that it is reasonable for an event to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.
Refinement is not equivalence

It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set

\[ \text{COIN} = \{ \text{Head} , \text{Tail} \} \]

Specification

\[
\text{Flip} \\
\text{coin} : \in \text{COIN}
\]

Refinement

\[
\text{Flip} \\
\text{coin} : \in \text{COIN}
\]

It should be clear that it is reasonable for an event to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.
Refinement is not equivalence

It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set

\[ \text{COIN} = \{ \text{Head, Tail} \} \]

Specification

\begin{align*}
\text{Flip} \\
\quad \text{coin} & \in \text{COIN}
\end{align*}

Refinement

\begin{align*}
\text{Flip} \\
\quad \text{coin} & \in \text{COIN}
\end{align*}

It should be clear that it is reasonable for an event to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.
Refinement is not equivalence

It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set

\[ \text{COIN} = \{ \text{Head}, \text{Tail} \} \]

Specification

\[ \text{Flip} \quad \text{coin :} \in \text{COIN} \]

Refinement

\[ \text{Flip} \quad \text{coin :} \in \text{COIN} \]

It should be clear that it is reasonable for an event to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.
Refining the state

The refining machine can have its own state, which in some way simulates the state of the refined machine.

The invariant of the refining machine has two components:

1. The constraints on its own state variables as for any other machine;
2. A refinement relation that describes how the refining machine’s state simulates the state of the refined machine.
Refining the state

The refining machine can have its own state, which in some way simulates the state of the refined machine.

The invariant of the refining machine has two components:

1. the constraints on its own state variables as for any other machine;
2. a refinement relation that describes how the refining machine’s state simulates the state of the refined machine.
Refining the state

The refining machine can have its own state, which in some way simulates the state of the refined machine.

The invariant of the refining machine has two components:

1. the constraints on its own state variables as for any other machine;

2. a refinement relation that describes how the refining machine’s state simulates the state of the refined machine.
The refining machine can have its own state, which in some way simulates the state of the refined machine.

The invariant of the refining machine has two components:

1. the constraints on its own state variables as for any other machine;
2. a refinement relation that describes how the refining machine’s state simulates the state of the refined machine.
The refining machine can have its own state, which in some way simulates the state of the refined machine.

The invariant of the refining machine has two components:

1. the constraints on its own state variables as for any other machine;
2. a refinement relation that describes how the refining machine’s state simulates the state of the refined machine.
A function machine I

MACHINE Function
SEES Function_ctx
VARIABLES

\[ \textit{fun}, \textit{val} \]

ININVARIANTS

\[ \textit{inv1}: \textit{fun} \in \text{DOM} \rightarrow \text{RAN} \]
\[ \textit{inv2}: \textit{val} \in \text{RAN} \]

EVENTS

Initialisation

begin

\[ \textit{act1}: \textit{fun} := \emptyset \]
\[ \textit{act2}: \textit{val} \in \text{RAN} \]

end
A function machine II

Event \( \text{Update} \triangleq \)

\[
\text{any } dval
\]

\[
\text{rval}
\]

when

\[
grd1 : \text{dval} \in \text{DOM}
\]

\[
grd2 : \text{rval} \in \text{RAN}
\]

then

\[
\text{act1} : \text{fun}(\text{dval}) := \text{rval}
\]

end
A function machine III

Event \( Fetch \triangleq \)

any \( dval \)

when

grd1 : \( dval \in \text{dom}(\text{fun}) \)

then

act1 : \( val := \text{fun}(dval) \)

end

END
A refinement of the function machine

Functions are commonly used concepts and there are many algorithms, that are, essentially, concerned with implementing function application.

Although arrays can be viewed as functions, the important property of an array is that it has a coherent domain of natural numbers. Generally, the domain of a function will not be coherent and in many cases consists of values from some opaque set. Thus, while an array can be simply mapped onto computer storage, a function generally cannot.

The strategy we adopt here is to store the domain of the function in an injective sequence and the range in a parallel sequence as shown in FunctionR.
MACHINE FunctionR
REFINES Function
SEES Function_ctx
VARIABLES
  fundom
  funran
  fun
  val

INVARIANTS
inv1: fundom ∈ 1 .. maxdom ↦ DOM  
     Injelive sequence
inv2: dom(fundom) = 1 .. card(fundom)
inv3: funran ∈ 1 .. maxdom ↦ RAN
     Sequence
inv4: dom(funran) = dom(fundom)
inv5: fun = fundom⁻¹; funran
     Refinement relation
THEOREMS

thm1: \( \text{dom}(\text{fundom}) = \text{dom}(\text{funran}) \)

thm2: \( \text{dom}(\text{fun}) = \text{ran}(\text{fundom}) \)

thm3: \( \text{ran}(\text{fun}) = \text{ran}(\text{funran}) \)

thm4: \( \text{dom}(\text{funran}) = \text{ran}(\text{fundom}^{-1}) \)
EVENTS

Initialisation

begin

act1: fundom := ∅
act2: funran := ∅
act3: fun := ∅
act4: val ∈ RAN

end
Event  \( \text{Update1} \equiv \)
refines  \( \text{Update} \)

any  \( dval \)
\( rval \)

when

\( \text{grd1} : \ dval \notin \text{ran}(\text{fundom}) \)
\( \text{grd2} : \ rval \in \text{RAN} \)

then

\( \text{act1} : \ \text{fundom}(\text{card}(\text{fundom}) + 1) := dval \)
\( \text{act2} : \ \text{funran}(\text{card}(\text{fundom}) + 1) := rval \)

end
Event $Update2 \triangleq$
refines $Update$
any $dval$

$rval$

when

$grd1 : dval \in ran(fundom)$

$grd2 : rval \in RAN$

then

$act1 : funran(fundom^{-1}(dval)) := rval$

end
Event \( Fetch \uparrow \)
refines \( Fetch \)
any \( dval \)
when
\( grd1 : \) \( dval \in \text{ran}(\text{fundom}) \)
then
\( act1 : \) \( val := \text{funran}(\text{fundom}^{-1}(dval)) \)
end
END
Towards a formal understanding of refinement

- Coin flip machine and refinement
- Event refinement
- Simple refinement intuition
- Checking our intuition
- The effect of nondeterminism
- A revised formalisation of refinement
- Isolating the problem
- Final formulation
Towards a formal understanding of refinement

Outline II

- Validating Flip under the new formulation
- Notes on the conjugate weakest precondition
- Refinement and feasibility
- Avoiding the infeasible
- Formality does not guarantee feasibility
- Proving feasibility
- The Infeasible cannot be made Feasible
Towards a formal understanding of refinement

In this section we are going to explore the formalisation of the notion of refinement that has been described in section 2.

To start our exploration we will use the simple coin flip event recast as an event that changes the state.
In this section we are going to explore the formalisation of the notion of refinement that has been described in section 2.

To start our exploration we will use the simple coin flip event recast as an event that changes the state.
In this section we are going to explore the formalisation of the notion of refinement that has been described in section 2.

To start our exploration we will use the simple coin flip event recast as an event that changes the state.
### Coin flip machine and refinement

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>CoinFlip</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEES</td>
<td>Coin_ctx</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>coin</td>
</tr>
<tr>
<td>INVARIANTS</td>
<td>inv1 : ( coin \in COIN )</td>
</tr>
<tr>
<td>EVENTS</td>
<td>Event ( Flip ) ( \cong )</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>act1 : ( coin :\in COIN )</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>CoinFlipR</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFINES</td>
<td>CoinFlip</td>
</tr>
<tr>
<td>SEES</td>
<td>Coin_ctx</td>
</tr>
<tr>
<td>VARIABLES</td>
<td>coin</td>
</tr>
<tr>
<td>EVENTS</td>
<td>Event ( Flip ) ( \cong )</td>
</tr>
<tr>
<td></td>
<td>refines ( Flip )</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>act1 : ( coin :\in COIN )</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
The refined event will be referred to as the *abstract* event and the refining event will be referred to as the *concrete* event.

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>$v_A$</td>
</tr>
<tr>
<td>Invariant</td>
<td>$I_A$</td>
</tr>
<tr>
<td>Refinement relation</td>
<td>$R$</td>
</tr>
<tr>
<td>Event</td>
<td>$Op_A$</td>
</tr>
<tr>
<td>Guard</td>
<td>$G$</td>
</tr>
<tr>
<td>Event body</td>
<td>$B_A$</td>
</tr>
</tbody>
</table>

Without any loss of generality we will assume that the states $v_A$ and $v_C$ are disjoint.
Towards a formal understanding of refinement

Simple refinement intuition

A simple intuition about refinement is shown above.

This leads to the following mathematical formulation of refinement

\[ I_A \land I_C \land R \land G \implies [B_C ; B_A](I_A \land I_C \land R) \]  \hfill (1)
Checking our intuition

We will check our current intuition for the Flip events as presented in Flip and FlipR.

We have to prove:

\[ \text{coinA} \in \text{COIN} \land \text{coinC} \in \text{COIN} \land \text{coinC} = \text{coinA} \Rightarrow [\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}] \\
(\text{coinA} \in \text{COIN} \land \text{coinC} \in \text{COIN} \land \text{coinC} = \text{coinA}) \]

To simplify the proof we will omit \( \text{coinA} \in \text{COIN} \) and \( \text{coinC} \in \text{COIN} \)
Checking our intuition

We will check our current intuition for the Flip events as presented in Flip and FlipR.

We have to prove:

\[\text{coinA} \in \text{COIN} \land \text{coinC} \in \text{COIN} \land \text{coinC} = \text{coinA}\]

\[\implies [\text{coinC} \in \text{COIN} \land \text{coinA} \in \text{COIN}]\]

\[(\text{coinA} \in \text{COIN} \land \text{coinC} \in \text{COIN} \land \text{coinC} = \text{coinA})\]

To simplify the proof we will omit \(\text{coinA} \in \text{COIN}\) and \(\text{coinC} \in \text{COIN}\).
Towards a formal understanding of refinement

Coin flip machine and refinement Event refinement Simple refinement intuition Checking our intuition The effect of refinement on feasibility Formality does not guarantee feasibility Proving feasibility The infeasible cannot be made feasible

Thus, it appears that our initial intuition doesn’t work.

What went wrong?

\[
\begin{align*}
\text{coinC} &= \text{coinA} \\
\iff& [\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \\
\iff& [\text{coinC} \in \text{COIN}] [\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \\
\iff& [\text{coinC} \in \text{COIN}][\text{coinA} := \text{Head}](\text{coinC} = \text{coinA}) \land \\
& [\text{coinA} := \text{Tail}](\text{coinC} = \text{coinA}) \\
\iff& [\text{coinC} \in \text{COIN}]((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \\
\iff& [\text{coinC} := \text{Head}]((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \land \\
& [\text{coinC} := \text{Tail}]((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \\
\iff& (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land \\
& (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \\
\iff& (\text{Head} = \text{Tail}) \\
\iff& \text{false}
\end{align*}
\]
Towards a formal understanding of refinement

Coin flip machine and refinement  Event refinement  Simple refinement intuition  Checking our intuition  The effect of refinement  Formality does not guarantee feasibility  Proving feasibility  The Infeasible cannot be made Feasible

\[ \text{coinC} = \text{coinA} \]

\[ \iff \quad \left[ \text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN} \right] (\text{coinC} = \text{coinA}) \]

\[ \iff \quad \left[ \text{coinC} \in \text{COIN} \right] [\text{coinA} \in \text{COIN}] (\text{coinC} = \text{coinA}) \]

\[ \iff \quad \left[ \text{coinC} \in \text{COIN} \right] ([\text{coinA} := \text{Head}] (\text{coinC} = \text{coinA}) \land \right] \]

\[ \iff \quad \left[ \text{coinA} := \text{Tail} \right] (\text{coinC} = \text{coinA}) \right] \]

\[ \iff \quad [\text{coinC} \in \text{COIN}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]

\[ \iff \quad [\text{coinC} := \text{Head}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \land \]

\[ \iff \quad [\text{coinC} := \text{Tail}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]

\[ \iff \quad (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land \]

\[ (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \]

\[ \iff \quad (\text{Head} = \text{Tail}) \]

\[ \iff \quad \text{false} \]

Thus, it appears that our initial intuition doesn’t work.

What went wrong?
Towards a formal understanding of refinement

Coin flip machine and refinement
Event refinement
Simple refinement intuition
Checking our intuition

The effect of ... the infeasible

Formality does not guarantee feasibility
Proving feasibility
The Infeasible cannot be made Feasible

\[
\text{coinC} = \text{coinA} \\
\implies \quad [\text{coinC} \in \text{COIN}; \text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \\
\implies \quad [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \\
\implies \quad [\text{coinC} \in \text{COIN}][\text{coinA} := \text{Head}](\text{coinC} = \text{coinA}) \land \\
\qquad [\text{coinA} := \text{Tail}](\text{coinC} = \text{coinA}) \\
\implies \quad [\text{coinC} \in \text{COIN}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \\
\implies \quad [\text{coinC} := \text{Head}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \land \\
\qquad [\text{coinC} := \text{Tail}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \\
\implies \quad (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land \\
\quad (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \\
\implies \quad (\text{Head} = \text{Tail}) \\
\implies \quad \text{false}
\]

Thus, it appears that our initial intuition doesn’t work.

What went wrong?
Towards a formal understanding of refinement

Coin flip machine and refinement

Event refinement

Simple refinement intuition

Checking our intuition

The effect of refinement

The infeasible

Formality does not guarantee feasibility

Proving feasibility

The Infeasible cannot be made Feasible

\[ coinC = coinA \]

\[ \Rightarrow \quad [\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}] (\text{coinC} = \text{coinA}) \]

\[ \Rightarrow \quad [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}] (\text{coinC} = \text{coinA}) \]

\[ \Rightarrow \quad [\text{coinC} \in \text{COIN}] ([\text{coinA} : = \text{Head}] (\text{coinC} = \text{coinA}) \land [\text{coinA} : = \text{Tail}] (\text{coinC} = \text{coinA})) \]

\[ \Rightarrow \quad [\text{coinC} \in \text{COIN}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]

\[ \Rightarrow \quad [\text{coinC} : = \text{Head}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \land [\text{coinC} : = \text{Tail}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]

\[ \Rightarrow \quad (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \]

\[ \Rightarrow \quad (\text{Head} = \text{Tail}) \]

\[ \Rightarrow \quad \text{false} \]

Thus, it appears that our initial intuition doesn’t work.

What went wrong?
Towards a formal understanding of refinement

Coin flip machine and refinement

Event refinement

Simple refinement intuition

Checking our intuition

The effect of refinement

The infeasible

Formality does not guarantee feasibility

Proving feasibility

The Infeasible cannot be made Feasible

\[ coinC = coinA \]

\[ \implies [coinC \in \text{COIN} ; coinA \in \text{COIN}](coinC = coinA) \]

\[ \implies [coinC \in \text{COIN}][coinA \in \text{COIN}](coinC = coinA) \]

\[ \implies [coinC \in \text{COIN}][(coinA := \text{Head})(coinC = coinA) \land \]

\[ (coinA := \text{Tail})(coinC = coinA)) \]

\[ \implies [coinC \in \text{COIN}][(coinC = \text{Head}) \land (coinC = \text{Tail})] \]

\[ \implies [coinC := \text{Head}]((coinC = \text{Head}) \land (coinC = \text{Tail})) \land \]

\[ [coinC := \text{Tail}]((coinC = \text{Head}) \land (coinC = \text{Tail})) \]

\[ \implies (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land \]

\[ (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \]

\[ \implies (\text{Head} = \text{Tail}) \]

\[ \implies \text{false} \]

Thus, it appears that our initial intuition doesn’t work.

What went wrong?
$\text{coinC} = \text{coinA}$

\[
\implies [\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA})
\implies [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA})
\implies [\text{coinC} \in \text{COIN}][(\text{coinA} := \text{Head})(\text{coinC} = \text{coinA}) \land
[\text{coinA} := \text{Tail}](\text{coinC} = \text{coinA}))
\implies [\text{coinC} \in \text{COIN}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail}))
\implies [\text{coinC} := \text{Head}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \land
[\text{coinC} := \text{Tail}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail}))
\implies (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land
(\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail})
\implies (\text{Head} = \text{Tail})
\implies \text{false}
\]

Thus, it appears that our initial intuition doesn’t work.

What went wrong?
Towards a formal understanding of refinement

Coin flip machine and refinement
Event refinement
Simple refinement intuition
Checking our intuition

The effect of refinement

Formality does not guarantee feasibility
Proving feasibility
The infeasible
The Infeasible cannot be made Feasible

\[
coinC = coinA \\
\implies [coinC \in \text{COIN} ; coinA \in \text{COIN}](coinC = coinA) \\
\implies [coinC \in \text{COIN}][coinA \in \text{COIN}](coinC = coinA) \\
\implies [coinC \in \text{COIN}][[coinA := \text{Head}](coinC = coinA) \land \\
[coinA := \text{Tail}](coinC = coinA)) \\
\implies [coinC \in \text{COIN}][((coinC = \text{Head}) \land (coinC = \text{Tail})) \\
\implies [coinC := \text{Head}][((coinC = \text{Head}) \land (coinC = \text{Tail})) \land \\
[coinC := \text{Tail}][((coinC = \text{Head}) \land (coinC = \text{Tail})) \\
\implies (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land \\
(\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \\
\implies (\text{Head} = \text{Tail}) \\
\implies false
\]

Thus, it appears that our initial intuition doesn’t work.

What went wrong?
Thus, it appears that our initial intuition doesn’t work.

What went wrong?


\[ \text{coinC} = \text{coinA} \]
\[ \implies [\text{coinC} \in \text{COIN} \land \text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \]
\[ \implies [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \]
\[ \implies [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \land\]
\[ \text{[coinA} := \text{Head}](\text{coinC} = \text{coinA}) \land\]
\[ \text{[coinA} := \text{Tail}](\text{coinC} = \text{coinA}) \]
\[ \implies [\text{coinC} \in \text{COIN}](\text{coinC} = \text{Head} \land \text{coinC} = \text{Tail}) \]
\[ \implies [\text{coinC} := \text{Head}](\text{coinC} = \text{Head} \land \text{coinC} = \text{Tail}) \land\]
\[ \text{[coinC} := \text{Tail}](\text{coinC} = \text{Head} \land \text{coinC} = \text{Tail}) \]
\[ \implies (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land\]
\[ (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \]
\[ \implies (\text{Head} = \text{Tail}) \]
\[ \implies \text{false} \]

Thus, it appears that our initial intuition doesn’t work.

What went wrong?
The effect of nondeterminism

The problem is, we have failed to take into account the effect of nondeterminism.

Both the abstract and concrete events may be nondeterministic.

In our initial intuition we took abstract and concrete initial states that were related by the refinement relation \( R \).

We then considered abstract and concrete final states obtained by respectively invoking the abstract and concrete events and then requiring the final states to be related by \( R \).

The Flip event clearly demonstrates that this is not a valid expectation.
The effect of nondeterminism

The problem is, we have failed to take into account the effect of nondeterminism.

Both the abstract and concrete events may be nondeterministic.

In our initial intuition we took abstract and concrete initial states that were related by the refinement relation $R$.

We then considered abstract and concrete final states obtained by respectively invoking the abstract and concrete events and then requiring the final states to be related by $R$.

The Flip event clearly demonstrates that this is not a valid expectation.
The effect of nondeterminism

The problem is, we have failed to take into account the effect of nondeterminism.

Both the abstract and concrete events may be nondeterministic.

In our initial intuition we took abstract and concrete initial states that were related by the refinement relation $R$.

We then considered abstract and concrete final states obtained by respectively invoking the abstract and concrete events and then requiring the final states to be related by $R$.

The Flip event clearly demonstrates that this is not a valid expectation.
The effect of nondeterminism

The problem is, we have failed to take into account the effect of nondeterminism.

Both the abstract and concrete events may be nondeterministic.

In our initial intuition we took abstract and concrete initial states that were related by the refinement relation $R$.

We then considered abstract and concrete final states obtained by respectively invoking the abstract and concrete events and then requiring the final states to be related by $R$.

The Flip event clearly demonstrates that this is not a valid expectation.
A revised formalisation of refinement

Figure: Accommodation of nondeterminism
Towards a formal understanding of refinement

A revised formalisation of refinement

**Figure:** Accommodation of nondeterminism
Consider any final state of the concrete event, contained in the *blue* set. Since $R$ is, in general, a relation the blue set of concrete states will be related to a the *green* set of abstract states. The informal refinement condition stated earlier requires only that at least one of these states can be reached by the abstract event invoked in an initial abstract state related by $R$ to the initial concrete state.

The problem with the initial simple formalisation is that it requires *all* reachable abstract states, the *red* set, to be reachable from the from the concrete event. The requirement is that the green set of states should be a subset of the red set.
Consider any final state of the concrete event, contained in the blue set. Since $R$ is, in general, a relation the blue set of concrete states will be related to a the green set of abstract states. The informal refinement condition stated earlier requires only that at least one of these states can be reached by the abstract event invoked in an initial abstract state related by $R$ to the initial concrete state.

The problem with the initial simple formalisation is that it requires all reachable abstract states, the red set, to be reachable from the concrete event. The requirement is that the green set of states should be a subset of the red set.
The story presented by the above diagram is as follows:

- from a single initial concrete state, \( \bullet \), the event \( Evt_C \) terminates in any state in blue set of states;
- the refinement relation \( R \) maps the initial concrete state to the set of states shown by the orange set of states;
- starting in any state in the orange set of states, the abstract event \( Evt_A \) terminates in some state in the red set of states;
- the refinement relation \( R \) maps the blue set of states to the green set of states

For \( Evt_C \) to be a refinement of \( Evt_A \), the green set must be contained in the red set. This needs to be true for all initial concrete states.
In the substitution $[B_C][B_A](l_A \land l_C \land R)$, the substitution $[B_A](l_A \land l_C \land R)$ is the problem.

By definition, $[S](P)$ yields the weakest precondition that guarantees that $S$ will terminate in a state satisfying $P$.

This is too strong, we require only that $B_A$ can terminate in a state satisfying $l_C \land R$. We need something weaker.

Consider $\neg[S] \neg P$:

$[S](\neg P)$ gives the weakest precondition guaranteeing that $S$ will terminate in a state satisfying $\neg P$.

$\neg[S](\neg P)$ gives the weakest precondition guaranteeing that $S$ will not terminate in a state satisfying $\neg P$, that is, it may satisfy $P$. 

Isolating the problem I
\neg[S] \neg P \text{ is sometimes called the conjugate weakest precondition of } S \text{ with respect to } P.

We can now recast 1 as:

\[ l_A \land l_C \land R \land G \implies [B_C] \neg[B_A](\neg(l_C \land R)) \]  \hspace{1cm} (2)
Final formulation

Currently we have ignored event results; we now have to take them into account. Suppose that the abstract event is

\[ result \leftarrow Evt_A = P \mid B_A \]

and the concrete event is

\[ result \leftarrow Evt_C = P \mid B_C \]

It is clearly a requirement of refinement that the value of the results of an event and its refinement must be equal. The results have the same name, so to differentiate we will temporarily rename the result of the concrete event to \( result' \) and let

\[ B'_C = [\text{result} := \text{result'}]B_C \]

then the general refinement condition becomes

\[ I_A \land I_C \land R \land G \implies [B'_C] \neg[B_A](\neg(\text{result'} = \text{result} \land I_C \land R)) \quad (3) \]
Final formulation

Currently we have ignored event results; we now have to take them into account. Suppose that the abstract event is

\[\text{result} \leftarrow Evt_A = P \mid B_A\]

and the concrete event is

\[\text{result} \leftarrow Evt_C = P \mid B_C\]

It is clearly a requirement of refinement that the value of the results of an event and its refinement must be equal. The results have the same name, so to differentiate we will temporarily rename the result of the concrete event to \(\text{result}'\) and let

\[B'_C = [\text{result} := \text{result}']B_C\]

then the general refinement condition becomes

\[I_A \land I_C \land R \land G \implies [B'_C][\neg[B_A](\neg(\text{result}' = \text{result} \land I_C \land R))]\]  (3)
Towards a formal understanding of refinement

Final formulation

Currently we have ignored event results; we now have to take them into account. Suppose that the abstract event is

\[ \text{result} \leftarrow \text{Evt}_A = P \mid B_A \]

and the concrete event is

\[ \text{result} \leftarrow \text{Evt}_C = P \mid B_C \]

It is clearly a requirement of refinement that the value of the results of an event and its refinement must be equal. The results have the same name, so to differentiate we will temporarily rename the result of the concrete event to \( \text{result}' \) and let

\[ B'_C = [\text{result} := \text{result}']B_C \]

then the general refinement condition becomes

\[ I_A \land I_C \land R \land G \implies [B'_C] \neg[B_A]((\neg(\text{result}' = \text{result} \land I_C \land R)) \quad (3) \]
Final formulation

Currently we have ignored event results; we now have to take them into account. Suppose that the abstract event is

\[
\text{result} \leftarrow Evt_A = P \mid B_A
\]

and the concrete event is

\[
\text{result} \leftarrow Evt_C = P \mid B_C
\]

It is clearly a requirement of refinement that the value of the results of an event and its refinement must be equal. The results have the same name, so to differentiate we will temporarily rename the result of the concrete event to \(\text{result}'\) and let

\[
B'_C = [\text{result} := \text{result}'] B_C
\]

then the general refinement condition becomes

\[
l_A \land l_C \land R \land G \implies [B'_C] \neg[B_A](\neg(\text{result}' = \text{result} \land l_C \land R)) \quad (3)
\]
Final formulation

Currently we have ignored event results; we now have to take them into account. Suppose that the abstract event is

\[
\text{result} \leftarrow Evt_A = P \mid B_A
\]

and the concrete event is

\[
\text{result} \leftarrow Evt_C = P \mid B_C
\]

It is clearly a requirement of refinement that the value of the results of an event and its refinement must be equal. The results have the same name, so to differentiate we will temporarily rename the result of the concrete event to \(\text{result}'\) and let

\[
B'_C = [\text{result} := \text{result}']B_C
\]

then the general refinement condition becomes

\[
l_A \land l_C \land R \land G \implies [B'_C][\neg B_A](\neg(\text{result}' = \text{result} \land l_C \land R)) \tag{3}
\]
Final formulation

Currently we have ignored event results; we now have to take them into account. Suppose that the abstract event is

\[ \text{result} \leftarrow Evt_A = P \mid B_A \]

and the concrete event is

\[ \text{result} \leftarrow Evt_C = P \mid B_C \]

It is clearly a requirement of refinement that the value of the results of an event and its refinement must be equal. The results have the same name, so to differentiate we will temporarily rename the result of the concrete event to \( \text{result}' \) and let

\[ B_C' = [\text{result} := \text{result}']B_C \]

then the general refinement condition becomes

\[ I_A \land I_C \land R \land G \implies [B_C'] \neg[B_A] \neg(\text{result}' = \text{result} \land I_C \land R) \] (3)
Validating Flip under the new formulation

\[
\begin{align*}
\text{coinC} &= \text{coinA} \\
\implies [\text{coinC} \in \text{COIN}] &\implies [\text{coinA} \in \text{COIN}] \implies (\text{coinC} = \text{coinA}) \\
\implies [\text{coinC} \in \text{COIN}] &\implies [\text{coinA} \in \text{COIN}] \implies (\text{coinC} \neq \text{coinA}) \\
\implies [\text{coinC} \in \text{COIN}] &\implies ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail})) \\
\implies [\text{coinC} \in \text{COIN}] &\implies ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail})) \\
\implies ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) &\land \\
((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \\
\implies \text{true}
\end{align*}
\]
Validating Flip under the new formulation

\[ \text{coinC} = \text{coinA} \]

\[ \implies \quad \neg \text{[coinC} \in \text{COIN]} \land \neg \text{[coinA} \in \text{COIN]} \land \neg (\text{coinC} = \text{coinA}) \]

\[ \implies \quad \neg \text{[coinC} \in \text{COIN]} \land \neg \text{[coinA} \in \text{COIN]} \land (\text{coinC} \neq \text{coinA}) \]

\[ \implies \quad \neg ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail})) \]

\[ \implies \quad (\text{coinC} \in \text{COIN} \land (\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail})) \]

\[ \implies \quad ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \]

\[ \implies \quad \text{true} \]
Validating Flip under the new formulation

\[ coinC = coinA \]

\[ \implies [coinC \in COIN] \land [\neg (coinC = coinA)] \]

\[ \implies [coinC \in COIN] \land \neg [coinA \in COIN] \land (coinC \neq coinA) \]

\[ \implies [coinC \in COIN] \land \neg ((coinC \neq \text{Head}) \land (coinC \neq \text{Tail})) \]

\[ \implies [coinC \in COIN] \land ((coinC = \text{Head}) \lor (coinC = \text{Tail})) \]

\[ \implies ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \]

\[ \implies \text{true} \]
Validating Flip under the new formulation

\[ \text{coinC} = \text{coinA} \]

\[ \implies [\text{coinC} \in \text{COIN}] \neg [\text{coinA} \in \text{COIN}] \neg (\text{coinC} = \text{coinA}) \]

\[ \implies [\text{coinC} \in \text{COIN}] \neg [\text{coinA} \in \text{COIN}] (\text{coinC} \neq \text{coinA}) \]

\[ \implies [\text{coinC} \in \text{COIN}] \neg ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail})) \]

\[ \implies [\text{coinC} \in \text{COIN}] ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail})) \]

\[ \implies ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land \\
(\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \]

\[ \implies \text{true} \]
Validating Flip under the new formulation

\[
\text{coinC} = \text{coinA}
\]

\[
\implies [\text{coinC} \in \text{COIN}] \land [\text{coinA} \in \text{COIN}] \land (\text{coinC} = \text{coinA})
\]

\[
\implies [\text{coinC} \in \text{COIN}] \land [\text{coinA} \in \text{COIN}] \land (\text{coinC} \neq \text{coinA})
\]

\[
\implies [\text{coinC} \in \text{COIN}] \land ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail}))
\]

\[
\implies [\text{coinC} \in \text{COIN}] \land ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail}))
\]

\[
((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land
((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail}))
\]

\[
\implies \text{true}
\]
Validating Flip under the new formulation

\[\text{coinC} = \text{coinA}\]

\[\implies [\text{coinC} \in \text{COIN}] \neg [\text{coinA} \in \text{COIN}] \neg (\text{coinC} = \text{coinA})\]

\[\implies [\text{coinC} \in \text{COIN}] \neg [\text{coinA} \in \text{COIN}] (\text{coinC} \neq \text{coinA})\]

\[\implies [\text{coinC} \in \text{COIN}] \neg ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail}))\]

\[\implies [\text{coinC} \in \text{COIN}] ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail}))\]

\[\implies ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail}))\]

\[\implies \text{true}\]
Notes on the conjugate weakest precondition

If $S$ is deterministic then $\lnot [S] \lnot P = [S] P$.

The behaviour of $\lnot [S] \lnot P$ can be demonstrated as follows:

Assume that $S$ has the form $v \in s^1$, then

$$
\lnot [S] \lnot P
= \lnot [v \in s] \lnot P
= \lnot \forall xx. (xx : s \implies [v := xx](\lnot P)) \{\text{semantics of } v \in s\}
= \exists xx. (xx : s \land \lnot ([v := xx](\lnot P))) \{\forall z.P \implies Q = \exists z.(P \land \lnot Q)\}
= \exists xx. (xx : s \land [v := xx]P) \{\lnot \text{ distributes through simple substitution}\}
$$

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

\(^1\)more generally we could use $S = S_1 \parallel \ldots \parallel S_n$
Notes on the conjugate weakest precondition

If \( S \) is deterministic then \( \neg [S] \neg P = [S] P \).

The behaviour of \( \neg [S] \neg P \) can be demonstrated as follows:

Assume that \( S \) has the form \( v \in s^1 \), then

\[
\neg [S] \neg P \\
= \neg [v \in s] \neg P \\
= \neg \forall xx. (xx : s \implies [v := xx](\neg P)) \{\text{semantics of } v \in s\} \\
= \exists xx. (xx : s \land \neg([v := xx](\neg P))) \{\neg \forall z. (P \implies Q) = \exists z. (P \land \neg Q)\} \\
= \exists xx. (xx : s \land [v := xx]P) \{\neg \text{ distributes through simple substitution}\}
\]

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

\(^1\)more generally we could use \( S = S_1 \parallel \ldots \parallel S_n \)
Towards a formal understanding of refinement

Notes on the conjugate weakest precondition

If $S$ is deterministic then $\neg[S] \neg P = [S] P$.

The behaviour of $\neg[S] \neg P$ can be demonstrated as follows:

Assume that $S$ has the form $v \in s^1$, then

\[
\begin{align*}
\neg[S] \neg P &= \neg[v \in s] \neg P \\
&= \neg\forall xx. (xx : s \implies [v := xx](\neg P)) \{\text{semantics of } v \in s\} \\
&= \exists xx. (xx : s \land \neg([v := xx](\neg P))) \{\neg\forall z.(P \implies Q) = \exists z.(P \land \neg Q)\} \\
&= \exists xx. (xx : s \land [v := xx] P) \{\neg \text{ distributes through simple substitution}\}
\end{align*}
\]

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

\(^1\)more generally we could use $S = S_1 \parallel \ldots \parallel S_n$
Notes on the conjugate weakest precondition

If $S$ is deterministic then $\neg[S] \neg P = [S] P$.

The behaviour of $\neg[S] \neg P$ can be demonstrated as follows:

Assume that $S$ has the form $v \in s^1$, then

\[
\neg[S] \neg P
= \neg[ v \in s] \neg P
= \neg \forall xx. \ (xx : s \implies [v := xx](\neg P)) \quad \{ \text{semantics of } v \in s \}
= \exists xx. \ (xx : s \land \neg ([v := xx](\neg P))) \quad \{ \forall z. (P \implies Q) = \exists z. (P \land \neg Q) \}
= \exists xx. \ (xx : s \land [v := xx] P) \quad \{ \neg \text{ distributes through simple substitution} \}
\]

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

\footnote{More generally we could use $S = S_1 \parallel \ldots \parallel S_n$.}
Notes on the conjugate weakest precondition

If \( S \) is deterministic then \( \neg[S] \neg P = [S]P \).

The behaviour of \( \neg[S] \neg P \) can be demonstrated as follows:

Assume that \( S \) has the form \( v \in s^1 \), then

\[
\neg[S] \neg P \\
= \neg[v \in s] \neg P \\
= \neg \forall xx. (xx : s \implies [v := xx](\neg P)) \{ \text{semantics of } v \in s \} \\
= \exists xx. (xx : s \land \neg ([v := xx](\neg P))) \{ \forall z.(P \implies Q) = \exists z.(P \land \neg Q) \} \\
= \exists xx. (xx : s \land [v := xx] P) \{ \neg \text{ distributes through simple substitution} \}
\]

Existential quantification captures the arbitrary choice from nondeterminism during refinement.
Notes on the conjugate weakest precondition

If $S$ is deterministic then $\neg[S]\neg P = [S]P$.

The behaviour of $\neg[S]\neg P$ can be demonstrated as follows:

Assume that $S$ has the form $v \in s^1$, then

$$\neg[S]\neg P = \neg[v \in s]\neg P = \neg\forall xx.(xx : s \implies [v := xx](\neg P)) \{\text{semantics of } v \in s\}$$

$$= \exists xx.(xx : s \land \neg([v := xx](\neg P))) \{\neg\forall z.(P \implies Q) = \exists z.(P \land \neg Q)\}$$

$$= \exists xx.(xx : s \land [v := xx]P) \{\neg \text{ distributes through simple substitution}\}$$

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

1 more generally we could use $S = S_1 \parallel \ldots \parallel S_n$
Notes on the conjugate weakest precondition

If $S$ is deterministic then $\neg[S]\neg P = [S]P$.

The behaviour of $\neg[S]\neg P$ can be demonstrated as follows:

Assume that $S$ has the form $v \in s^1$, then

$$\neg[S]\neg P$$

$$= \neg[v \in s]\neg P$$

$$= \neg \forall xx. (xx : s \rightarrow [v := xx](\neg P)) \{ \text{semantics of } v \in s \}$$

$$= \exists xx. (xx : s \land \neg([v := xx](\neg P))) \{ \neg \forall z.(P \rightarrow Q) = \exists z.(P \land \neg Q) \}$$

$$= \exists xx. (xx : s \land [v := xx]P) \{ \neg \text{ distributes through simple substitution} \}$$

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

$^1$more generally we could use $S = S_1 \parallel \ldots \parallel S_n$
Notes on the conjugate weakest precondition

If $S$ is deterministic then $\neg[S] \neg P = [S] P$.

The behaviour of $\neg[S] \neg P$ can be demonstrated as follows:

Assume that $S$ has the form $v \in s^1$, then

\[
\neg[S] \neg P \\
\quad = \neg[v \in s] \neg P \\
\quad = \neg \forall xx. (xx : s \implies [v := xx](\neg P)) \{\text{semantics of } v \in s\} \\
\quad = \exists xx. (xx : s \land \neg([v := xx](\neg P))) \{\neg \forall z. (P \implies Q) = \exists z. (P \land \neg Q)\} \\
\quad = \exists xx. (xx : s \land [v := xx] P) \{\neg \text{ distributes through simple substitution}\}
\]

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

\(^1\)more generally we could use $S = S_1 \parallel \ldots \parallel S_n$
In contrast, \([S]P = \forall xx. (xx : s \implies [v := xx]P)\)

Assume that \(S\) has the form \(v := e\)

\[
\begin{align*}
\neg[S]\neg P &= \neg[v := e]\neg P \\
&= \neg(\neg[v := e]P) \\
&= [v := e]P \\
&= [S]P
\end{align*}
\]
In contrast, \([S]P = \forall xx. (xx : s \implies [v := xx]P)\)

Assume that \(S\) has the form \(v := e\)

\[
\neg[S]
\neg P
\begin{align*}
  &= \neg[v := e]\neg P \\
  &= \neg(\neg[v := e]P) \\
  &= [v := e]P \\
  &= [S]P
\end{align*}
\]
In contrast, $[S]P = \forall xx. (xx : s \implies [v := xx]P)$

Assume that $S$ has the form $v := e$

\[-[S] \neg P\]
\[= \neg [v := e] \neg P\]
\[= \neg (\neg [v := e] P)\]
\[= [v := e] P\]
\[= [S] P\]
In contrast, \([S]P = \forall xx.(xx : s \implies [v := xx]P)\)

Assume that \(S\) has the form \(v := e\)

\[
\neg[S] \neg P \\
= \neg[v := e] \neg P \\
= \neg(\neg[v := e]P) \\
= [v := e]P \\
= [S]P
\]
In contrast, \([S]P = \forall xx.(xx : s \implies [v := xx]P)\)

Assume that \(S\) has the form \(v := e\)

\[
\neg[S] \neg P \\
= \neg[v := e] \neg P \\
= \neg(\neg[v := e] P) \\
= [v := e] P \\
= [S] P
\]
In contrast, \( [S]P = \forall xx.(xx : s \implies [v := xx]P) \)

Assume that \( S \) has the form \( v := e \)

\[
\neg[S] \neg P \\
= \quad \neg[v := e] \neg P \\
= \quad \neg(\neg[v := e] P) \\
= \quad [v := e] P \\
= \quad [S] P
\]
In contrast, \([S]P = \forall xx.(xx : s \implies [v := xx]P)\)

Assume that \(S\) has the form \(v := e\)

\[
\neg [S] \neg P \\
= \neg [v := e] \neg P \\
= \neg (\neg [v := e] P) \\
= [v := e] P \\
= [S] P
\]
Refinement and feasibility

Refinements form a partial ordering stretching from Abort to Magic, where

1. for all $P$, $[\text{Abort}]P = \text{false}$, and
2. for all $P$, $[\text{Magic}]P = \text{true}$

Let $\sqsubseteq$ represent refinement then a sequence of refinements $R_i$ is ordered as follows:

$$\text{Abort} \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq \text{Magic}$$

Thus, $\text{Abort}$ is refined by anything, while $\text{Magic}$ is a refinement of anything. $\text{Abort}$ is easy to implement, while $\text{Magic}$ is impossible to implement. $\text{Magic}$ is infeasible.

---

$^2$The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

Refinements form a partial ordering stretching from Abort to Magic, where

1. for all $P$, $\lbrack \text{Abort} \rbrack P = \text{false}$, and
2. for all $P$, $\lbrack \text{Magic} \rbrack P = \text{true}$

Let $\sqsubseteq$ represent refinement then a sequence of refinements $R_i$ is ordered as follows:

$$\text{Abort} \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq \text{Magic}$$

Thus, $\text{Abort}$ is refined by anything, while $\text{Magic}$ is a refinement of anything. $\text{Abort}$ is easy to implement, while $\text{Magic}$ is impossible to implement. $\text{Magic}$ is infeasible.

---

2 The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

Refinements form a partial ordering stretching from Abort to Magic, where

1. for all $P$, $[\text{Abort}]P = \text{false}$, and
2. for all $P$, $[\text{Magic}]P = \text{true}$

Let $\sqsubseteq$ represent refinement then a sequence of refinements $R_i$ is ordered as follows:

$\text{Abort} \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq \text{Magic}$

Thus, Abort is refined by anything, while Magic is a refinement of anything. Abort is easy to implement, while Magic is impossible to implement. Magic is infeasible.

---

$^2$The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

Refinements form a partial ordering stretching from Abort to Magic, where

1. for all $P$, $[Abort]P = false$, and
2. for all $P$, $[Magic]P = true$

Let $\sqsubseteq$ represent refinement then a sequence of refinements $R_i$ is ordered as follows:\(^2\)

$$Abort \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq Magic$$

Thus, $Abort$ is refined by anything, while $Magic$ is a refinement of anything. $Abort$ is easy to implement, while $Magic$ is impossible to implement. $Magic$ is infeasible.

\(^2\)The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

Refinements form a partial ordering stretching from Abort to Magic, where

1. for all $P$, $[\text{Abort}] P = \text{false}$, and
2. for all $P$, $[\text{Magic}] P = \text{true}$

Let $\sqsubseteq$ represent refinement then a sequence of refinements $R_i$ is ordered as follows:

$$\text{Abort} \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq \text{Magic}$$

Thus, $\text{Abort}$ is refined by anything, while $\text{Magic}$ is a refinement of anything. $\text{Abort}$ is easy to implement, while $\text{Magic}$ is impossible to implement. $\text{Magic}$ is infeasible.

---

$^2$The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

Refinements form a partial ordering stretching from Abort to Magic, where

1. for all $P$, $[\text{Abort}]P = false$, and
2. for all $P$, $[\text{Magic}]P = true$

Let $\sqsubseteq$ represent refinement then a sequence of refinements $R_i$ is ordered as follows:\(^2\)

$$\text{Abort} \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq \text{Magic}$$

Thus, $\text{Abort}$ is refined by anything, while $\text{Magic}$ is a refinement of anything. $\text{Abort}$ is easy to implement, while $\text{Magic}$ is impossible to implement. $\text{Magic}$ is infeasible.

\(^2\)The ordering, in general, is not linear as shown here, but a lattice.
Avoiding the infeasible

The refinement ordering demonstrates that at any refinement step the refinement may become infeasible.

But, it is important to understand that a construct that is feasible can always be refined to a feasible construct.

That is, infeasibility can always be avoided—provided, of course, that the original construct was feasible.
Avoiding the infeasible

The refinement ordering demonstrates that at any refinement step the refinement may, become infeasible.

But, it is important to understand that a construct that is feasible can always be refined to a feasible construct.

That is, infeasibility can always be avoided — provided, of course, that the original construct was feasible.
Avoiding the infeasible

The refinement ordering demonstrates that at any refinement step the refinement may, become infeasible.

But, it is important to understand that a construct that is feasible can always be refined to a feasible construct.

That is, infeasibility can always be avoided —provided, of course, that the original construct was feasible.
Avoiding the infeasible

The refinement ordering demonstrates that at any refinement step the refinement may, become infeasible.

But, it is important to understand that a construct that is feasible can always be refined to a feasible construct.

That is, infeasibility can always be avoided —provided, of course, that the original construct was feasible.
Formality does not guarantee feasibility

To drive home the fact that specifying something using predicates does not preclude infeasibility, here is a specification of an event that defies Fermat’s last theorem, which conjectures,

"there is no integer solutions for $x, y, z$ to the equation $x^n + y^n = z^n$ for integer $n$ with $n > 2$,"

This conjecture was presented in 1637 and not proved until 1995.
Towards a formal understanding of refinement

Coin flip machine and refinement
Event refinement
Simple refinement intuition
Checking our intuition
The effect of... the infeasible

Formality does not guarantee feasibility
Proving feasibility
The Infeasible cannot be made Feasible

Fermat I

MACHINE Fermat
SEES Fermat_ctx
VARIABLES
  a
  b
  c

INVARIANTS
  inv1: a ∈ N ∧ b ∈ N ∧ c ∈ N

EVENTS
Initialisation
  begin
  act1: a, b, c := 0, 0, 0
  end
Event \( \text{Fermat} \triangleq \)

any \( n \)
\( x \)
\( y \)
\( z \)

when

\( \text{grd1} : n \in \mathbb{N} \land n > 2 \)
\( \text{grd2} : EXP(x)(n) + EXP(y)(n) = EXP(z)(n) \)

then

\( \text{act1} : a, b, c := x, y, z \)

end

END
Feasibility proof obligations can be generated, but generally they are existential proof obligations. The general strategies for discharging existential proof obligations involve producing *witnesses*, that is giving values that demonstrate that there is at least one solution. This, of course, is equivalent to producing an implementation.

Thus, proof of the feasibility of producing an implementation can involve producing an implementation. This is not a productive solution.

But the situation can be inverted:

if an implementation —with accompanying discharged proof obligations— can be produced then the feasibility proof obligations could have been discharged. Conversely, if the feasibility proof obligations cannot be discharged, then any attempts at implementation will fail.
Proving feasibility

Feasibility proof obligations can be generated, but generally they are existential proof obligations. The general strategies for discharging existential proof obligations involve producing *witnesses*, that is giving values that demonstrate that there is at least one solution. This, of course, is equivalent to producing an implementation.

Thus, proof of the feasibility of producing an implementation can involve producing an implementation. This is not a productive solution.

But the situation can be inverted:

if an implementation —with accompanying discharged proof obligations— can be produced then the feasibility proof obligations could have been discharged. Conversely, if the feasibility proof obligations cannot be discharged, then any attempts at implementation will fail.
Feasibility proof obligations can be generated, but generally they are existential proof obligations. The general strategies for discharging existential proof obligations involve producing witnesses, that is giving values that demonstrate that there is at least one solution. This, of course, is equivalent to producing an implementation.

Thus, proof of the feasibility of producing an implementation can involve producing an implementation. This is not a productive solution.

But the situation can be inverted:

if an implementation— with accompanying discharged proof obligations— can be produced then the feasibility proof obligations could have been discharged. Conversely, if the feasibility proof obligations cannot be discharged, then any attempts at implementation will fail.
Feasibility proof obligations can be generated, but generally they are existential proof obligations. The general strategies for discharging existential proof obligations involve producing witnesses, that is giving values that demonstrate that there is at least one solution. This, of course, is equivalent to producing an implementation.

Thus, proof of the feasibility of producing an implementation can involve producing an implementation. This is not a productive solution.

But the situation can be inverted:

if an implementation —with accompanying discharged proof obligations— can be produced then the feasibility proof obligations could have been discharged. Conversely, if the feasibility proof obligations cannot be discharged, then any attempts at implementation will fail.
The Infeasible cannot be made Feasible

While refinement of a feasible specification can produce an infeasible refinement, the converse cannot happen.

Put more strongly: if you start with an infeasible specification, you will not be able to implement it through refinement. This may not be obvious given that infeasibility may be cloaked behind magic at the specification stage.

This can be simply demonstrated.

Consider an event whose body is represented by the nondeterministic assignment

$$v_A \in s$$

Assume that the state invariant is $I_A$, then the proof obligation will be

$$I_A \implies \forall xx.(xx : s \implies [v_A := xx]I_A)$$

and this is true, if $s$ is empty, that is the event is infeasible.
The Infeasible cannot be made Feasible

While refinement of a feasible specification can produce an infeasible refinement, the converse cannot happen.

Put more strongly: if you start with an infeasible specification, you will not be able to implement it through refinement. This may not be obvious given that infeasibility may be cloaked behind *magic* at the specification stage.

This can be simply demonstrated.

Consider an event whose body is represented by the nondeterministic assignment

\[ v_A \in s \]

Assume that the state invariant is \( I_A \), then the proof obligation will be

\[ I_A \implies \forall xx. (xx : s \implies [v_A := xx] I_A) \]

and this is true, if \( s \) is empty, that is the event is infeasible.
The Infeasible cannot be made Feasible

While refinement of a feasible specification can produce an infeasible refinement, the converse cannot happen.

Put more strongly: if you start with an infeasible specification, you will not be able to implement it through refinement. This may not be obvious given that infeasibility may be cloaked behind *magic* at the specification stage.

This can be simply demonstrated.

Consider an event whose body is represented by the nondeterministic assignment

\[ v_A \in s \]

Assume that the state invariant is \( I_A \), then the proof obligation will be

\[ I_A \implies \forall xx. (xx : s \implies [v_A := xx]I_A) \]

and this is true, if \( s \) is empty, that is the event is infeasible.
The Infeasible cannot be made Feasible

While refinement of a feasible specification can produce an infeasible refinement, the converse cannot happen.

Put more strongly: if you start with an infeasible specification, you will not be able to implement it through refinement. This may not be obvious given that infeasibility may be cloaked behind *magic* at the specification stage.

This can be simply demonstrated.

Consider an event whose body is represented by the nondeterministic assignment

\[ v_A \in s \]

Assume that the state invariant is \( I_A \), then the proof obligation will be

\[ I_A \implies \forall xx. (xx : s \implies [v_A := xx]I_A) \]

and this is true, if \( s \) is empty, that is the event is infeasible.
Now suppose that we claim to refine the body of that event to $ν_C := e$, i.e., a deterministic refinement, with invariant $I_B$ and refinement relation $ν_A = ν_C$,

then we would have to prove

$$l_A \land l_C \land ν_A = ν_C \implies [ν_C := e][ν_A \in s]\neg(l_A \land l_C \land ν_A = ν_C)$$

$$= l_A \land l_C \land ν_A = ν_C \implies [ν_C := e][ν_A \in s](\neg l_A \lor (ν_A \neq ν_C))$$

$$= l_A \land l_C \land ν_A = ν_C \implies [ν_C := e](\forall xx.(xx : s \implies [ν_A := xx](\neg l_A \lor (ν_A \neq ν_C))))$$

$$= l_A \land l_C \land ν_A = ν_C \implies [ν_C := e]\exists xx.(xx : s \land l_A \land l_C \land xx = ν_C)$$

$$= l_A \land l_C \land ν_A = ν_C \implies \exists xx.(xx : s \land l_A \land l_C \land xx = e)$$

$$= \text{false}, \quad \text{if } s \text{ is empty}$$

This demonstrates the sting in the tail of magic: it is truly impossible to implement.
Now suppose that we claim to refine the body of that event to \( v_C := e \), ie a deterministic refinement, with invariant \( I_B \) and refinement relation \( v_A = v_C \),

then we would have to prove

\[
I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg [v_A \in s] \neg (I_A \land I_C \land v_A = v_C)
\]

\[
= I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg [v_A \in s] (\neg I_A \lor (v_A \neq v_C))
\]

\[
= I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg (\forall xx. (xx : s \implies [v_A := xx] (\neg I_A \lor (v_A \neq v_C))))
\]

\[
= I_A \land I_C \land v_A = v_C \implies [v_C := e] \exists xx. (xx : s \land I_A \land I_C \land xx = v_C)
\]

\[
= I_A \land I_C \land v_A = v_C \implies \exists xx. (xx : s \land I_A \land I_C \land xx = e)
\]

\[
= false, \quad \text{if } s \text{ is empty}
\]

This demonstrates the sting in the tail of magic: it is truly impossible to implement.
Now suppose that we claim to refine the body of that event to \( v_C := e \), ie a deterministic refinement, with invariant \( I_B \) and refinement relation \( v_A = v_C \),

then we would have to prove

\[
I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg [v_A \in s] \neg (I_A \land I_C \land v_A = v_C)
\]
\[
= I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg [v_A \in s] (\neg I_A \lor (v_A \neq v_C))
\]
\[
= I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg (\forall xx. (xx : s \implies [v_A := xx] (\neg I_A \lor (v_A \neq v_C))))
\]
\[
= I_A \land I_C \land v_A = v_C \implies [v_C := e] \exists xx. (xx : s \land I_A \land I_C \land xx = v_C)
\]
\[
= I_A \land I_C \land v_A = v_C \implies \exists xx. (xx : s \land I_A \land I_C \land xx = e)
\]
\[
= false, \quad \text{if } s \text{ is empty}
\]

This demonstrates the sting in the tail of magic: it is truly impossible to implement.
Now suppose that we claim to refine the body of that event to \( v_C := e \), i.e., a deterministic refinement, with invariant \( I_B \) and refinement relation \( v_A = v_C \),

then we would have to prove

\[
I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg [v_A \in s] \neg (I_A \land I_C \land v_A = v_C) = I_A \land I_C \land v_A = v_C \implies [v_C := e] \neg (\forall xx. (xx : s \implies [v_A := xx] (\neg I_A \lor (v_A \neq v_C))))
\]

\[
= I_A \land I_C \land v_A = v_C \implies [v_C := e] \exists xx. (xx : s \land I_A \land I_C \land xx = v_C)
\]

\[
= I_A \land I_C \land v_A = v_C \implies \exists xx. (xx : s \land I_A \land I_C \land xx = e)
\]

\[
= false, \quad \text{if } s \text{ is empty}
\]

This demonstrates the sting in the tail of *magic*: it is truly impossible to implement.
The second implementation will involve data refinement.

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.
The second implementation will involve data refinement.

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

The second implementation will involve data refinement.

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, \textit{queuetokens}. A function, \textit{queueitem}, maps from \textit{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, \textit{queuetokens}. A function, \textit{queueitem}, maps from \textit{queuetokens} to the actual items.
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, \textit{queuetokens}. A function, \textit{queueitem}, maps from \textit{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. This is done in the Queue_ctx machine.

The Item_ctx machine contains the carrier set ITEM, which is used for item identifiers.
CONTEXT Item_ctx

SETS

ITEM the set from which items in the queue are taken

END
CONTEXT Queue_ctx Context machine for queues modelled as sequences

SETS

QUEUE identifiers for items stored in the queue

CONSTANTS

maxqueue upper limit to number of items in the queue
tail sequence tail function
delete sequence element deletion
AXIOMS

\( axm1: \) finite(QUEUE)

\( axm2: \) maxqueue \( \in \mathbb{N}_1 \)

\( axm3: \) tail \( \in (\mathbb{N}_1 \mapsto \text{QUEUE}) \rightarrow (\mathbb{N} \mapsto \text{QUEUE}) \)

\( axm4: \) \( \forall n, q \cdot n \in \mathbb{N}_1 \land q \in 1 \ldots n \rightarrow \text{QUEUE} \)
\( \Rightarrow \) \( \text{dom(tail}(q)) = 1 \ldots n - 1 \)

\( axm5: \) \( \forall n, q, i \cdot n \in \mathbb{N}_1 \land q \in 1 \ldots n \rightarrow \text{QUEUE} \)
\( \land i \in \text{dom(tail}(q)) \land i + 1 \in \text{dom}(q) \)
\( \Rightarrow \) \( \text{tail}(q)(i) = q(i + 1) \)

\( axm6: \) \( \forall n, q \cdot n \in \mathbb{N}_1 \land q \in 1 \ldots n \rightarrow \text{QUEUE} \)
\( \Rightarrow \) \( \text{dom(tail}(q)) = 1 \ldots n - 1 \)

\( axm7: \) delete \( \in \mathbb{N}_1 \rightarrow ((\mathbb{N}_1 \mapsto \text{QUEUE}) \rightarrow (\mathbb{N} \mapsto \text{QUEUE})) \)

\( axm8: \) \( \forall d, q \cdot d \in \mathbb{N}_1 \land q \in \mathbb{N}_1 \rightarrow \text{QUEUE} \land d \notin \text{dom}(q) \)
\( \Rightarrow \) \( \text{delete}(d)(q) = q \)

\( axm9: \) \( \forall d, q \cdot d \in \mathbb{N}_1 \land q \in \mathbb{N}_1 \rightarrow \text{QUEUE} \land d \in \text{dom}(q) \)
\( \Rightarrow \) \( \text{ran(delete}(d)(q)) = \text{ran}(q) \setminus \{q(d)\} \)
axm10: ∀ n, d, q · n ∈ ℕ₀ ∧ d ∈ 1..n ∧ q ∈ 1..n → QUEUE
⇒ dom(delete(d)(q)) = 1..n − 1

axm11: ∀ n, d, q, i · n ∈ ℕ₀ ∧ d ∈ 1..n ∧ q ∈ 1..n → QUEUE
∧ i ∈ 1..d − 1
⇒ delete(d)(q)(i) = q(i)

axm12: ∀ n, d, q, i · n ∈ ℕ₀ ∧ d ∈ 1..n ∧ q ∈ 1..n → QUEUE
∧ i ∈ d..n − 1
⇒ delete(d)(q)(i) = q(i + 1)

axm13: ∀ n, q · n ∈ ℕ₀ ∧ q ∈ 1..n → QUEUE
⇒ tail(q) = delete(1)(q)
THEOREMS

**thm1:** \( \forall n, q : n \in \mathbb{N}_1 \land q \in 1 .. n \rightarrow QUEUE \Rightarrow q \in \text{dom}(\text{tail}) \)

**thm2:** \( \forall n, q : n \in \mathbb{N}_1 \land q \in 1 .. n \leftrightarrow QUEUE \Rightarrow \text{ran}(\text{tail}(q)) = \text{ran}(q) \setminus \{q(1)\} \)

**thm3:** \( \forall n, d, q : n \in \mathbb{N}_1 \land d \in 1 .. n \land q \in 1 .. n \rightarrow QUEUE \Rightarrow q \in \text{dom}(\text{delete}(d)) \)

END
MACHINE Queue

SEES Queue_ctx, Item_ctx
The Queue machine II

VARIABLES

*queuetokens*
- tokens currently in queue

*queue*
- the queue of tokens

*queueitems*
- a function for fetching the item associated with a token

*qsize*
- current size of queue
INVARIANTS

- **inv1**: \( \text{queuetokens} \subseteq \text{QUEUE} \)
- **inv2**: \( \text{queue} \in 1 .. \text{qsize} \rightarrow \text{queuetokens} \)
- **inv3**: \( \text{qsize} \in 0 .. \text{maxqueue} \)
- **inv4**: \( \text{queueitems} \in \text{queuetokens} \rightarrow \text{ITEM} \)
THEOREMS

\textit{thm1}: \quad \textit{queuetokens} = \textit{ran(queue)}
EVENTS

Initialisation

begin

act1:  queuetokens := ∅

act2:  queue := ∅

act3:  qsize := 0

act4:  queueitems := ∅

end
**The Queue machine VI**

**Event** $Enqueue \eq$

\[
\begin{align*}
\text{any} & \quad \text{item} \\
\text{qid} & \\
\text{when} & \\
\text{grd1:} & \quad \text{item} \in ITEM \\
\text{grd2:} & \quad \text{qid} \in QUEUE \setminus \text{queuetokens} \\
\text{grd3:} & \quad \text{qsize} \neq \text{maxqueue} \\
\text{then} & \\
\text{act1:} & \quad \text{qsize} := \text{qsize} + 1 \\
\text{act2:} & \quad \text{queuetokens} := \text{queuetokens} \cup \{\text{qid}\} \\
\text{act3:} & \quad \text{queue}(\text{qsize} + 1) := \text{qid} \\
\text{act4:} & \quad \text{queueitems}(\text{qid}) := \text{item} \\
\text{end} & 
\end{align*}
\]
The Queue machine VII

Event  Dequeue  ≜

  when

  grd1:  qsize \neq 0

  then

  act1:  qsize := qsize − 1

  act2:  queuetokens := queuetokens \{queue(1)\}

  act3:  queue := tail(queue)

  act4:  queueitems := \{queue(1)\} \ priced \ queueitems

  end
The Queue machine VIII

Event  \textit{Unqueue} \equiv

any  \quad qid

when

\textit{grd1}:  \quad qsize \neq 0

\textit{grd2}:  \quad qid \in \text{queuetokens}

then

\textit{act1}:  \quad qsize := qsize - 1

\textit{act2}:  \quad \text{queuetokens} := \text{queuetokens} \setminus \{qid\}

\textit{act3}:  \quad queue := \text{delete}(queue^{-1}(qid))(queue)

\textit{act4}:  \quad \text{queueitems} := \{qid\} \triangleleft \text{queueitems}

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.

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.

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

\begin{itemize}
  \item \texttt{qfirst} the first element of the list;
  \item \texttt{qlast} the last element of the list;
  \item \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;
  \item \texttt{qsize} the size of the list.
\end{itemize}

Additionally, the refinement uses the variable \texttt{queueitem} in the same role as in the \textit{Queue} machine. Although this variable 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.

Refinements may not use variables of the refined machine except in invariants. \textit{Complete hiding} is enforced.
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.

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

Additionally, the refinement uses the variable $\texttt{queueitem}$ in the same role as in the $\textit{Queue}$ 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.

Refinements may not use variables of the refined machine except in invariants. Complete hiding is enforced.
Refining the Queue machine

The refinement replaces the monolithic sequence model by a list model, in which the discrete elements of the set $\texttt{queuetokens}$ 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.

Additionally, the refinement uses the variable $\texttt{queueitem}$ in the same role as in the Queue machine. Although this variable 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.

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

Additionally, the refinement uses the variable *queueitem* in the same role as in the *Queue* machine. Although this variable 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. Refinements may not use variables of the refined machine except in invariants. *Complete hiding* is enforced.
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:

- \( q_{\text{first}} \): the first element of the list;
- \( q_{\text{last}} \): the last element of the list;
- \( q_{\text{next}} \): 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;
- \( q_{\text{size}} \): the size of the list.

Additionally, the refinement uses the variable *queueitem* in the same role as in the *Queue* machine. Although this variable 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.

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

Additionally, the refinement uses the variable \textit{queueitem} in the same role as in the \textit{Queue} machine. Although this variable 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. Refinements may not use variables of the refined machine except in invariants. \textit{Complete hiding} is enforced.
Relational composition and iteration

Since we are modelling a list structure we will use \textit{relational composition} on the $qnext$ function to describe paths along the list, and we will use transitive \textit{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 \textit{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 $r^{n+1} = r^n \cdot r$. 
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

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

Multiple composition is expressed by iteration: \( qnext^n \) (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 \( r^{n+1} = r^n \circ r \).
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

\[
q\text{first} \quad \text{gives the identity of the first item in the list}
\]

\[ q\text{next}(q\text{first}) \quad \text{gives the identity of the second item in the list} \]

\[(q\text{next} ; q\text{next})(q\text{first}) \quad \text{gives the identity of the third item in the list} \]

\[ \ldots \quad \text{etc} \]

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 \( r^{n+1} = r^n ; r \).
Relational composition and iteration

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

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

- \texttt{qfirst} gives the identity of the first item in the list
- \texttt{qnext(qfirst)} gives the identity of the second item in the list
- \texttt{(qnext ; qnext)(qfirst)} gives the identity of the third item in the list
- ... etc

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

If \( r \in X \leftrightarrow X \), then \( r^0 = \text{id}(X) \) 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

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

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 *qnext* with itself \(n\) times.

If \(r \in X \leftrightarrow X\), then \(r^0 = \text{id}(X)\) and \(r^{n+1} = r^n \circ r\).
Since we are modelling a list structure we will use *relational composition* on the $q\text{next}$ function to describe 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

$$q\text{first}$$
$$q\text{next}(q\text{first})$$
$$(q\text{next} ; q\text{next})(q\text{first})$$
$$\ldots$$

gives the identity of the first item in the list

gives the identity of the second item in the list

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 $iterate(q\text{next} \mapsto n)$), is the result of composing $q\text{next}$ with itself $n$ times.

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

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

Multiple composition is expressed by iteration: \( qnext^n \) (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 \( r^{n+1} = r^n ; r \).
Relational composition and iteration

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

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

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

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

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

Multiple composition is expressed by *iteration*: `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 \( r^{n+1} = r^n ; r \).
Closure

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

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

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

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

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

\[
\begin{align*}
    r^* &= \bigcup n.(n \in \mathbb{N} \mid r^n) \uparrow
\end{align*}
\]

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

\[
\begin{align*}
    r^+ &= \bigcup n.(n \in \mathbb{N}_1 \mid r^n),
\end{align*}
\]

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 $f \cdot f$ 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.
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.
The definitions required by the list machine are given in `List_ctx`.

```
CONTEXT List_ctx  Context machine for queues modelled as lists
EXTENDS Queue_ctx
CONSTANTS iterate closure1 closure0
```
The List Context machine II

AXIOMS

axm1:  \( iterate \in (\text{QUEUE} \leftrightarrow \text{QUEUE}) \times \mathbb{N} \)
\( \rightarrow (\text{QUEUE} \leftrightarrow \text{QUEUE}) \)

axm2:  \( \forall r \cdot r \in \text{QUEUE} \leftrightarrow \text{QUEUE} \Rightarrow iterate(r \mapsto 0) = id(\text{QUEUE}) \)

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

axm4:  \( \text{closure1} \in (\text{QUEUE} \leftrightarrow \text{QUEUE}) \rightarrow (\text{QUEUE} \leftrightarrow \text{QUEUE}) \)

axm5:  \( \forall r \cdot r \in \text{QUEUE} \leftrightarrow \text{QUEUE} \)
\( \Rightarrow \text{closure1}(r) = (\bigcup n \cdot n \in \mathbb{N}_1 | iterate(r \mapsto n)) \)

axm6:  \( \text{closure0} \in (\text{QUEUE} \leftrightarrow \text{QUEUE}) \rightarrow (\text{QUEUE} \leftrightarrow \text{QUEUE}) \)

axm7:  \( \forall r \cdot r \in \text{QUEUE} \leftrightarrow \text{QUEUE} \)
\( \Rightarrow \text{closure0}(r) = (\bigcup n \cdot n \in \mathbb{N} | iterate(r \mapsto n)) \)
THOREMS

thm1: \( \forall f, n. f \in \text{QUEUE} \mapsto \text{QUEUE} \land n \in \mathbb{N}_1 \Rightarrow iterate(f \mapsto n) = f \circ (iterate(f \mapsto n - 1)) \)

thm2: \( \forall f. f \in \text{QUEUE} \iff \text{QUEUE} \Rightarrow iterate(f \mapsto 1) = f \)

thm3: \( \forall f, q, n. f \in q \mapsto q \land q \subseteq \text{QUEUE} \land n \in \mathbb{N} \Rightarrow iterate(f \mapsto n) \in q \mapsto q \)

thm4: \( \forall r. r \in \text{QUEUE} \iff \text{QUEUE} \Rightarrow closure_0(r) = closure_1(r) \cup id(\text{QUEUE}) \)

END
The invariant of QueueR

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

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

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:

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

The list consists of the elements of \textit{queues\textit{tokens}} hence

\[
qsize = \text{card}(\text{queues\textit{tokens}})
\]

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

\[
\text{queues\textit{tokens}} \neq \emptyset \implies q\textit{first} \in \text{queues\textit{tokens}}
\]
\[
\text{queues\textit{tokens}} \neq \emptyset \implies q\textit{last} \in \text{queues\textit{tokens}}
\]

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

\[
q\text{next} \in \text{queues\textit{tokens}} \setminus \{q\text{last}\} \implies \text{queues\textit{tokens}} \setminus \{q\text{first}\}
\]
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 *queuetokens* hence

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

For non-empty lists, \textit{qfirst} and \textit{qlast} are elements of *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 qnext is injective, but it is also surjective and therefore bijective:

\[
\text{qnext} \in \text{queuetokens} \setminus \{\text{qlast}\} \implies \text{queuetokens} \setminus \{\text{qfirst}\}
\]
The Refinement relation

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

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

The following should follow from the invariant:

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

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

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

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

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

\[
\text{closure}_1(qnext)[\{\text{qfirst}\}] = \text{queuetokens}
\]

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

\[
\begin{align*}
\text{qsize} \neq 0 & \Rightarrow \text{queue}(1) = \text{qfirst} \\
\text{qsize} \neq 0 & \Rightarrow \text{queue}(\text{qsize}) = \text{qlast} \\
\text{qsize} \neq 0 & \Rightarrow \forall i \cdot i \in 1 \ldots \text{qsize} - 1 \Rightarrow \text{queue}(i + 1) = \text{qnext}(\text{queue}(i))
\end{align*}
\]
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 $\text{queuetokens}$
   \[ \text{closure}_1(qnext)[\{qfirst\}] = \text{queuetokens} \]

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

The following should follow from the invariant:

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

\[
\forall t \cdot t \in \text{queuetokens} \land q\text{size} > 1 \land t \neq q\text{last} \Rightarrow t \in \text{dom}(q\text{next})
\]

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

\[
\forall t \cdot t \in \text{queuetokens} \land q\text{size} > 1 \land t \neq q\text{first} \Rightarrow t \in \text{ran}(q\text{next})
\]

3. Following all sequences of \( q\text{next} \) from \( q\text{first} \) should give all tokens in \( \text{queuetokens} \)

\[
\text{closure1}(q\text{next})[\{q\text{first}\}] = \text{queuetokens}
\]

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

\[
q\text{size} \neq 0 \Rightarrow \text{queue}(1) = q\text{first}
\]

\[
q\text{size} \neq 0 \Rightarrow \text{queue}(q\text{size}) = q\text{last}
\]

\[
q\text{size} \neq 0 \Rightarrow \forall i \cdot i \in 1 \ldots q\text{size} - 1 \Rightarrow \text{queue}(i + 1) = q\text{next}(\text{queue}(i))
\]
QueueR Theorems

The following should follow from the invariant:

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

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

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

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

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

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

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

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

Traversing the list from \( qfirst \) should cover all the elements of \( \text{queuetokens} \)

\[
qnext^+[\{qfirst\}] = \text{queuetokens}
\]
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 \textit{qnext}, so the following should be a theorem:

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

Traversing the list from \textit{qfirst} should cover all the elements of \textit{queuetokens}

\[ qnext^+ \{qfirst\} = \text{queuetokens} \]
The QueueR machine

MACHINE QueueR
REFINES Queue
SEES List_ctx, Item_ctx
The QueueR machine II

VARIABLES

`queuetokens`
- tokens currently in queue

`queueitems`
- a function for fetching the item associated with a token

`qsize`
- current size of queue

`qfirst`
- the first item, if any, in the queue

`qnext`
- link to the next item, if any, in the queue

`qlast`
- the last item, if any, in the queue
The QueueR machine III

INVARINTS

\begin{align*}
\text{inv1:} & \quad \text{queuetokens} \subseteq \text{QUEUE} \\
\text{inv2:} & \quad \text{qsize} = \text{card} (\text{queuetokens}) \\
\text{inv3:} & \quad \text{qfirst} \in \text{QUEUE} \\
\text{inv4:} & \quad \text{qsize} \neq 0 \Rightarrow \text{qfirst} \in \text{queuetokens} \\
\text{inv5:} & \quad \text{qlast} \in \text{QUEUE} \\
\text{inv6:} & \quad \text{qsize} \neq 0 \Rightarrow \text{qlast} \in \text{queuetokens} \\
\text{inv7:} & \quad \text{qnext} \in \text{queuetokens} \setminus \{\text{qlast}\} \mapsto \text{queuetokens} \setminus \{\text{qfirst}\} \\
\text{inv8:} & \quad \text{qnext} \cap \text{id} (\text{queuetokens}) = \emptyset \\
\text{inv9:} & \quad \text{qsize} \neq 0 \Rightarrow \text{qlast} = \text{iterate} (\text{qnext} \mapsto \text{qsize} - 1)(\text{qfirst}) \\
\text{inv10:} & \quad \forall i \cdot i \in 1 .. \text{qsize} \Rightarrow \text{queue}(i) = \text{iterate} (\text{qnext} \mapsto i - 1)(\text{qfirst})
\end{align*}

Refinement relation starts here
THEOREMS

thm1: $qsize \neq 0 \Rightarrow queue(1) = qfirst$

thm2: $qsize \neq 0 \Rightarrow queue(qsize) = qlast$

thm3: $qsize \neq 0 \Rightarrow (\forall i \cdot i \in 1..qsize-1 \Rightarrow queue(i+1) = qnext(queue(i)))$

thm4: $closure1(qnext)[\{qfirst\}] = queuetokens$

thm5:

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

thm6:

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

thm7: $closure1(qnext) \cap id(queuetokens) = \emptyset$

thm8: $qnext \in QUEUE \Rightarrow QUEUE$
The QueueR machine V

EVENTS

Initialisation

begin

act1: queuetokens := ∅
act2: qsize := 0
act3: queueitems := ∅
act4: qfirst ∈ QUEUE
act5: qlast ∈ QUEUE
act6: qnext := ∅

end
The QueueR machine VI

Event $Enqueue0 \cong$
refines $Enqueue$

any item
qid

when

$grd1$: $qsize = 0$
$grd2$: $item \in ITEM$
$grd3$: $qid \in QUEUE \setminus queuetokens$
$grd4$: $qsize \neq maxqueue$

then

$act1$: $queuetokens := queuetokens \cup \{qid\}$
$act2$: $qsize := qsize + 1$
$act3$: $qlast := qid$
$act4$: $qfirst := qid$

end
Event $Enqueue1 \equiv$
refines $Enqueue$

any $item$
$qid$

when
$grd1$: $qsize \neq 0$
$grd2$: $item \in ITEM$
$grd3$: $qid \in QUEUE \setminus queuetokens$
$grd4$: $qsize \neq maxqueue$

then
$act1$: $queuetokens := queuetokens \cup \{qid\}$
$act2$: $qsize := qsize + 1$
$act3$: $qnext := qnext \leftarrow \{qlast \mapsto qid\}$
$act4$: $qlast := qid$

end
Event  $\text{Deque}0 \sqsupseteq$
refines  $\text{Deque}$
when
$grd1$:  $qsize = 1$
then
$act1$:  $qsize := 0$
$act2$:  $queuetokens := \emptyset$
$act3$:  $queueitems := \emptyset$
end
Event $Dequeue_1 \equiv$
refines $Dequeue$
when
$grd_1$: $qsize > 1$
then
$act_1$: $qsize := qsize - 1$
$act_2$: $queuetokens := queuetokens \ \{qfirst\}$
$act_3$: $queueitems := \{qfirst\} \sqsubseteq queueitems$
$act_4$: $qfirst := qnext(qfirst)$
$act_5$: $qnext := \{qfirst\} \sqsubseteq qnext$
end
The Queue\textsubscript{R} machine $X$

**Event** \textit{Unqueue0} \textit{refines} \textit{Unqueue}

\textbf{any} \: \textit{qid}

\textbf{when}

\textit{grd1}: \: \textit{qsize} = 1

\textit{grd2}: \: \textit{qid} \in \textit{queuetokens}

\textbf{then}

\textit{act1}: \: \textit{qsize} := \textit{qsize} − 1

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

\textit{act3}: \: \textit{queueitems} := \{\textit{qid}\} \lefttriangle \textit{queueitems}

\textbf{end}
Event Unqueue1 \cong 
refines Unqueue 
any qid 
when
grd1: \( qsize = 1 \) 
grd2: \( qid = qfirst \) 
grd3: \( qid \in queuetokens \) 
then
act1: \( qsize := qsize - 1 \) 
act2: \( queuetokens := queuetokens \setminus \{qid\} \) 
act3: \( queueitems := \{qid\} \triangleleft queueitems \) 
act4: \( qfirst := qnext(qfirst) \) 
act5: \( qnext := \{qid\} \triangleleft qnext \) 
end
Event $Unqueue2$ $\cong$
refines $Unqueue$

any $qid$

when

$grd1$: $qsize > 1$

$grd2$: $qid = qlast$

$grd3$: $qid \in queuetokens$

then

$act1$: $qsize := qsize - 1$

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

$act3$: $queueitems := \{qid\} \leftarrow queueitems$

$act4$: $qnext := qnext \triangleright \{qid\}$

$act5$: $qlast := qnext^{-1}(qid)$

end
The QueueR machine XIII

Event Unqueue3 ⇔
refines Unqueue

any qid

when

grd1: qsize > 2

grd2: qid ≠ qfirst

grd3: qid ≠ qlast

grd4: qid ∈ queuetokens

then

act1: qsize := qsize − 1

act2: queuetokens := queuetokens \ {qid}

act3: queueitems := {qid} ⋈ queueitems

act4: qnext(qnext⁻¹(qid)) := qnext(qid)

end

END
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 \textit{prev} for free because \textit{qnext} is an injective function, so \textit{prev} has been obtained by simply inverting \textit{qnext}. In an implementation we have no such luxury. In the refinement of Unqueue3 we implement \textit{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 \textit{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, i.e. implement *qprev*. 
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.
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 1

MACHINE QueueRR
REFINES QueueR
SEES List_ctx, Item_ctx
QueueRR II

VARIABLES

\textit{queue\text{tokens}}
\begin{itemize}
\item tokens currently in queue
\end{itemize}

\textit{queue\text{items}}
\begin{itemize}
\item a function for fetching the item associated with a token
\end{itemize}

\textit{qsize}
\begin{itemize}
\item current size of queue
\end{itemize}

\textit{qfirst}
\begin{itemize}
\item first item, if any, in queue
\end{itemize}

\textit{qnext}
\begin{itemize}
\item link to next item, if any, in queue
\end{itemize}

\textit{qlast}
\begin{itemize}
\item last item, if any, in queue
\end{itemize}

\textit{qprev}
\begin{itemize}
\item previous item
\end{itemize}
deleting

BOOL switch to prevent interference during distributed deletion
Queue RR IV

IN VARIANTS

inv1: $q_{prev} \in \text{QUEUE}$

inv2: $\text{deleting} \in \text{BOOL}$
EVENTS

Initialisation

\textit{extended}

\begin{verbatim}
BEGIN
  act1: queuetokens := \emptyset
  act2: qsize := 0
  act3: queueitems := \emptyset
  act4: qfirst \in QUEUE
  act5: qlast \in QUEUE
  act6: qnext := \emptyset
  act7: qprev \in QUEUE
  act8: deleting := FALSE
END
\end{verbatim}
Event $Enqueue0 \triangleq$
extends $Enqueue0$

any $item$
$qid$

when

grd1: $qsize = 0$
grd2: $item \in ITEM$
grd3: $qid \in QUEUE \setminus \text{queuetokens}$
grd4: $qsize \neq \text{maxqueue}$

then

act1: $\text{queuetokens} := \text{queuetokens} \cup \{qid\}$
act2: $qsize := qsize + 1$
act3: $qlast := qid$
act4: $qfirst := qid$

end
QueueRR VII

Event \textit{Enqueue1} \triangleq
extends \textit{Enqueue1}
  any \item
    qid
  when
  grd1: \text{qsize} \neq 0
  grd2: \text{item} \in \text{ITEM}
  grd3: qid \in \text{QUEUE} \setminus \text{queuetokens}
  grd4: qsize \neq \text{maxqueue}
  grd5: \text{deleting} = \text{FALSE}
then
act1: \text{queuetokens} := \text{queuetokens} \cup \{\text{qid}\}
act2: qsize := qsize + 1
act3: qnext := qnext \leftarrow \{qlast \mapsto qid\}
act4: qlast := qid
end
Event \( Dequeue0 \triangleq \)
extends \( Dequeue0 \)
when
\[ \text{grd1: } qsize = 1 \]
\[ \text{grd2: } \text{deleting} = \text{FALSE} \]
then
\[ \text{act1: } qsize := 0 \]
\[ \text{act2: } \text{queuetokens} := \emptyset \]
\[ \text{act3: } \text{queueitems} := \emptyset \]
end
Event \( Dequeue1 \) 

extends \( Dequeue1 \)

when

\( grd1 : qsize > 1 \)

\( grd2 : \) deleting = FALSE

then

\( act1 : qsize := qsize - 1 \)

\( act2 : \) queuetokens := queuetokens \( \setminus \{ qfirst \} \)

\( act3 : \) queueitems := \( \{ qfirst \} \leftarrow queueitems \)

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

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

end
Queue Events Modelling of queue Context Machines The Queue machine Refining the Queue machine The List Context machine The QueueR invariant The Refinement relation QueueR Theorems

Event $Unqueue0 \equiv$

extends $Unqueue0$

any $qid$

when

$grd1: qsize = 1$
$grd2: qid \in\ queuetokens$
$grd3: deleting = FALSE$

then

$act1: qsize := qsize - 1$
$act2: queuetokens := queuetokens \ \{qid\}$
$act3: queueitems := \{qid\} \leftarrow queueitems$

end
QueueRR XI

Event $Unqueue1 \equiv$

extends $Unqueue1$

any $qid$

when

grd1: $qsize = 1$
grd2: $qid = qfirst$
grd3: $qid \in queuetokens$

$grd4$: deleting = FALSE

then

act1: $qsize := qsize - 1$
act2: queuetokens := queuetokens \ {qid}
act3: queueitems := {qid} $\ll$ queueitems
act4: $qfirst := qnext(qfirst)$
act5: $qnext := \{qid\} \ll qnext$

end
QueueRR XII

Event \( Unqueue2 \) \( \cong \)
extends \( Unqueue2 \)
  any \( qid \)
when
  \( grd1 : \text{qsize} > 1 \)
  \( grd2 : qid = qlast \)
  \( grd3 : qid \in \text{queuetokens} \)
  \( grd4 : \text{deleting} = \text{FALSE} \)
then
  \( act1 : \text{qsize} := \text{qsize} - 1 \)
  \( act2 : \text{queuetokens} := \text{queuetokens} \setminus \{qid\} \)
  \( act3 : \text{queueitems} := \{qid\} \bowtie \text{queueitems} \)
  \( act4 : qnext := qnext \triangleright \{qid\} \)
  \( act5 : qlast := qnext^{-1}(qid) \)
end
Event \( Unqueue3 \) \( \triangleq \) refines \( Unqueue3 \)

any \( qid \)

when

\( grd1: \) \( qid \in \text{queuetokens} \)
\( grd2: \) \( \text{qsize} > 2 \)
\( grd3: \) \( qid \neq \text{qfirst} \)
\( grd4: \) \( qid \neq \text{qlast} \)
\( grd7: \) \( \text{qprev} \in \text{dom}(\text{qnext}) \)
\( grd5: \) \( \text{qnext}(\text{qprev}) = qid \)
\( grd6: \) \( \text{deleting} = \text{TRUE} \)

then

\( act1: \) \( \text{queueitems} := \{qid\} \triangleleft \text{queueitems} \)
Queue Events Modelling of queue Context Machines The Queue machine Refining the Queue machine The List Context machine The QueueR invariant The Refinement relation QueueR Theorems

QueueRR XIV

\[
\begin{align*}
\text{act2: } & \quad \text{queuetokens} := \text{queuetokens} \setminus \{ \text{qid} \} \\
\text{act3: } & \quad \text{qsize} := \text{qsize} - 1 \\
\text{act4: } & \quad \text{qnext}(\text{qprev}) := \text{qnext}(\text{qid}) \\
\text{act5: } & \quad \text{deleting} := \text{FALSE} \\
\text{end}
\end{align*}
\]
Event $Unqueue3I \equiv$

any $qid$

when

$grd1$: $qid \in \text{queuetokens}$
$grd2$: $qsize > 1$
$grd3$: $qid \neq qfirst$
$grd4$: $qid \neq qlast$
$grd5$: $\text{deleting} = \text{FALSE}$

then

$act1$: $qprev := qfirst$
$act2$: $\text{deleting} := \text{TRUE}$

end
Event \( \text{Unqueue3S} \) \( \triangleq \)

Status convergent

any \( qid \)

when

\( grd1: \ qid \in \text{queuetokens} \)
\( grd2: \ qsize > 1 \)
\( grd3: \ qid \neq qfirst \)
\( grd4: \ qid \neq qlast \)
\( grd5: \ qprev \in \text{dom}(qnext) \)
\( grd6: \ qnext(qprev) \neq qid \)
\( grd7: \ deleting = \text{TRUE} \)

then

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

end
VARIANT

closure1(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 \( \text{prev} \). Clearly we expect that the number of remaining items in that set is finite and decreasing. The set of items is obtained by applying \( \text{closure(qnext)} \) to \( \text{prev} \).
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.