

# System Modelling and Design

Refinement

Towards Implementation

Revision: 1.5, October 10, 2008

Ken Robinson

October 22, 2008

©Ken Robinson 2005

[mailto::k.robinson@unsw.edu.au](mailto:k.robinson@unsw.edu.au)

## Part I

# Informal notions of refinement

## Contents

<b>I</b>	<b>Informal notions of refinement</b>	<b>1</b>
<b>1</b>	<b>What is Refinement?</b>	<b>2</b>
1.1	What does refinement guarantee? . . . . .	3
1.2	Some things you can do in refinement . . . . .	3
1.3	Reducing nondeterminism: examples . . . . .	3
1.4	Refinement is not equivalence . . . . .	3
1.5	Refining the state . . . . .	4
1.6	A function machine . . . . .	4
1.7	A refinement of the function machine . . . . .	5
<b>II</b>	<b>Formalisation of refinement</b>	<b>8</b>
<b>2</b>	<b>Towards a formal understanding of refinement</b>	<b>8</b>
2.1	Coin flip machine and refinement . . . . .	8
2.2	Event refinement . . . . .	9
2.3	Simple refinement intuition . . . . .	9

2.4	Checking our intuition . . . . .	9
2.5	The effect of nondeterminism . . . . .	10
2.6	A revised formalisation of refinement . . . . .	10
2.7	Isolating the problem . . . . .	11
2.8	Final formulation . . . . .	11
2.9	Validating Flip under the new formulation . . . . .	12
2.10	Notes on the conjugate weakest precondition . . . . .	12
2.11	Refinement and feasibility . . . . .	13
2.12	Avoiding the infeasible . . . . .	13
2.13	Formality does not guarantee feasibility . . . . .	13
2.14	Proving feasibility . . . . .	15
2.15	The Infeasible cannot be made Feasible . . . . .	15

### **III A Queue Development 15**

2.16	Queue Events . . . . .	16
2.17	Modelling of queue . . . . .	16
2.18	Context Machines . . . . .	16
2.19	The Queue machine . . . . .	18
2.20	Refining the Queue machine . . . . .	20
2.21	The List Context machine . . . . .	22
2.22	The QueueR invariant . . . . .	23
2.23	The Refinement relation . . . . .	23
2.24	QueueR Theorems . . . . .	23
2.24.1	The QueueR machine . . . . .	24
2.24.2	Refinement of Unqueue3 . . . . .	30
2.24.3	Notes on the Variant . . . . .	34

## **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;

## **1 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.

### 1.1 What does refinement guarantee?

The refinement of a construct —for example an operation— 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.

### 1.2 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.

### 1.3 Reducing nondeterminism: examples

#### Specification

##### SimpleChoice

$result \in \{ 3, 6, 9, 12 \};$

#### Refinement

##### SimpleChoice

$result := 6;$

### 1.4 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

$COIN = \{ Head, Tail \}$

#### 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*

## 1.5 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.

## 1.6 A function machine

**MACHINE** Function

**SEES** Function\_ctx

**VARIABLES**

: *fun*

: *val*

**INVARIANTS**

**inv1** :  $fun \in DOM \leftrightarrow RAN$

**inv2** :  $val \in RAN$

**EVENTS**

**Initialisation**

**begin**

**act1** :  $fun := \emptyset$

**act2** :  $val \in RAN$

**end**

**Update**  $\hat{=}$

**any**

: *dval*

```

: rval

where
grd1 :  $dval \in DOM$ 
grd2 :  $rval \in RAN$ 

then
act1 :  $fun(dval) := rval$ 

end

Fetch  $\hat{=}$ 
any
: dval

where
grd1 :  $dval \in dom(fun)$ 

then
act1 :  $val := fun(dval)$ 

end

END

```

## 1.7 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** :  $fundom \in 1 .. maxdom \mapsto DOM$

Injective sequence

**inv2** :  $dom(fundom) = 1 .. card(fundom)$

**inv3** :  $funran \in 1 .. maxdom \mapsto RAN$

Sequence

**inv4** :  $dom(funran) = dom(fundom)$

**inv5** :  $fun = fundom^{-1}; funran$

Refinement relation

## THEOREMS

**thm1** :  $dom(fundom) = dom(funran)$

**thm2** :  $dom(fun) = ran(fundom)$

**thm3** :  $ran(fun) = ran(funran)$

**thm4** :  $dom(funran) = ran(fundom^{-1})$

## EVENTS

### Initialisation

**begin**

**act1** :  $fundom := \emptyset$

**act2** :  $funran := \emptyset$

**act3** :  $fun := \emptyset$

**act4** :  $val \in RAN$

**end**

**Update1**  $\hat{=}$

**Refines** Update

**any**  
 :  $dval$   
 :  $rval$   
**where**  
**grd1** :  $dval \notin \text{ran}(f_{\text{undom}})$   
**grd2** :  $rval \in \text{RAN}$   
**then**  
**act1** :  $f_{\text{undom}}(\text{card}(f_{\text{undom}}) + 1) := dval$   
**act2** :  $f_{\text{unran}}(\text{card}(f_{\text{undom}}) + 1) := rval$   
**end**

**Update2**  $\hat{=}$

**Refines** Update

**any**  
 :  $dval$   
 :  $rval$   
**where**  
**grd1** :  $dval \in \text{ran}(f_{\text{undom}})$   
**grd2** :  $rval \in \text{RAN}$   
**then**  
**act1** :  $f_{\text{unran}}(f_{\text{undom}}^{-1}(dval)) := rval$   
**end**

**Fetch**  $\hat{=}$

**Refines** Fetch

**any**  
 :  $dval$   
**where**  
**grd1** :  $dval \in \text{ran}(f_{\text{undom}})$

```

then
  act1 :  $val := funran(fundom^{-1}(dval))$ 
end
END

```

## Part II

# Formalisation of refinement

## Contents

### 2 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.

#### 2.1 Coin flip machine and refinement

```

MACHINE CoinFlip
SEES Coin_ctx
VARIABLES
: coin
INVARIANTS
inv1 :  $coin \in COIN$ 
EVENTS
Flip  $\hat{=}$ 
begin
act1 :  $coin \in COIN$ 
end
END

```

```

MACHINE CoinFlipR
REFINES CoinFlip
SEES Coin_ctx
VARIABLES
: coin
EVENTS
Flip  $\hat{=}$ 
Refines Flip
begin
act1 :  $coin \in COIN$ 
end
END

```

## 2.2 Event refinement

The refined event will be referred to as the *abstract* event and the refining event will be referred to as the *concrete* event.

	<b>Abstract</b>	<b>Concrete</b>
State	$v_A$	$v_C$
Invariant	$I_A$	$I_C$
Refinement relation		$R$
Event	$Op_A$	$Op_C$
Guard	$G$	$G$
Event body	$B_A$	$B_C$

Without any loss of generality we will assume that the states  $v_A$  and  $v_C$  are disjoint.

## 2.3 Simple refinement intuition

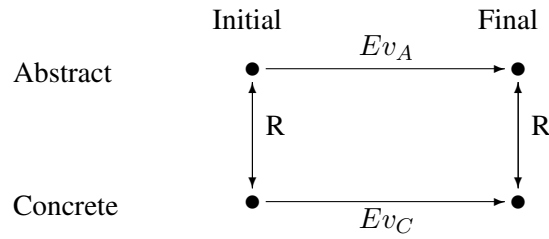


Figure 1: Initial refinement intuition

A simple intuition about refinement is shown in figure 1.

This leads to the following mathematical formulation of refinement

$$I_A \wedge I_C \wedge R \wedge G \implies [B_C ; B_A](I_A \wedge I_C \wedge R) \quad (1)$$

## 2.4 Checking our intuition

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

We have to prove:

$$\begin{aligned} & coinA \in COIN \wedge coinC \in COIN \wedge coinC = coinA \\ \implies & [coinC \in COIN ; coinA \in COIN] \\ & (coinA \in COIN \wedge coinC \in COIN \wedge coinC = coinA) \end{aligned}$$

To simplify the proof we will omit  $coinA \in COIN$  and  $coinC \in COIN$

$$\begin{aligned}
& coinC = coinA \\
& \implies [coinC \in COIN ; coinA \in COIN](coinC = coinA) \\
& \implies [coinC \in COIN][coinA \in COIN](coinC = coinA) \\
& \implies [coinC \in COIN]([coinA := Head](coinC = coinA) \wedge \\
& \quad [coinA := Tail](coinC = coinA)) \\
& \implies [coinC \in COIN]((coinC = Head) \wedge (coinC = Tail)) \\
& \implies [coinC := Head]((coinC = Head) \wedge (coinC = Tail)) \wedge \\
& \quad [coinC := Tail]((coinC = Head) \wedge (coinC = Tail)) \\
& \implies (Head = Head) \wedge (Head = Tail) \wedge \\
& \quad (Tail = Head) \wedge (Tail = Tail) \\
& \implies (Head = Tail) \\
& \implies false
\end{aligned}$$

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

What went wrong?

## 2.5 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 (figure 1) 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.

## 2.6 A revised formalisation of refinement

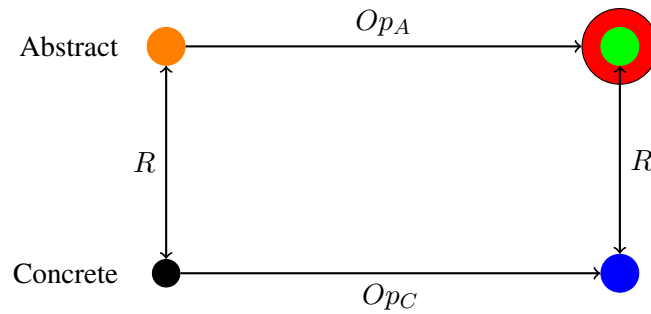


Figure 2: Accommodation of nondeterminism

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 (1) 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.

The story presented by

Fig 2

is as follows:

- from a single initial concrete state,  $\bullet$ , the event  $Op_C$  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  $Op_A$  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  $Op_C$  to be a refinement of  $Op_A$ , the **green** set must be contained in the **red** set. This needs to be true for all initial concrete states.

## 2.7 Isolating the problem

In the substitution  $[B_C][B_A](I_A \wedge I_C \wedge R)$ , the substitution  $[B_A](I_A \wedge I_C \wedge R)$  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  $I_C \wedge R$ . 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  $P$ .

$\neg[S]\neg P$  is sometimes called the *conjugate weakest precondition* of  $S$  with respect to  $P$ .

We can now recast 1 as:

$$I_A \wedge I_C \wedge R \wedge G \implies [B_C]\neg[B_A](\neg(I_C \wedge R)) \quad (2)$$

## 2.8 Final formulation

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

$$result \longleftarrow Op_A = P \mid B_A$$

and the concrete event is

$$result \leftarrow Op_C = P \mid B_C$$

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

$$B'_C = [result := result']B_C$$

then the general refinement condition becomes

$$I_A \wedge I_C \wedge R \wedge G \implies [B'_C] \neg [B_A] (\neg (result' = result \wedge I_C \wedge R)) \quad (3)$$

## 2.9 Validating Flip under the new formulation

$$\begin{aligned} coinC &= coinA \\ \implies [coinC \in COIN] \neg [coinA \in COIN] \neg (coinC = coinA) \\ \implies [coinC \in COIN] \neg [coinA \in COIN] (coinC \neq coinA) \\ \implies [coinC \in COIN] \neg ((coinC \neq Head) \wedge (coinC \neq Tail)) \\ \implies [coinC \in COIN] ((coinC = Head) \vee (coinC = Tail)) \\ \implies ((Head = Head) \vee (Head = Tail)) \wedge \\ &\quad ((Tail = Head) \vee (Tail = Tail)) \\ \implies true \end{aligned}$$

## 2.10 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

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

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

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

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

---

<sup>1</sup>more generally we could use  $S = S_1 \parallel \dots \parallel S_n$

$$\begin{aligned}
& \neg[S]\neg P \\
&= \neg[v := e]\neg P \\
&= \neg(\neg[v := e]P) \\
&= [v := e]P \\
&= [S]P
\end{aligned}$$

## 2.11 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:<sup>2</sup>

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

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*.

## 2.12 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.*

## 2.13 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.

**MACHINE** Fermat

**SEES** Fermat.ctx

**VARIABLES**

---

<sup>2</sup>The ordering, in general, is not linear as shown here, but a lattice.

:  $a$

:  $b$

:  $c$

## INVARIANTS

**inv1** :  $a \in \mathbb{N} \wedge b \in \mathbb{N} \wedge c \in \mathbb{N}$

## EVENTS

### Initialisation

**begin**

**act1** :  $a, b, c := 0, 0, 0$

**end**

**Fermat**  $\hat{=}$

**any**

:  $n$

:  $x$

:  $y$

:  $z$

**where**

**grd1** :  $n \in \mathbb{N} \wedge n > 2$

**grd2** :  $EXP(x)(n) + EXP(y)(n) = EXP(z)(n)$

**then**

**act1** :  $a, b, c := x, y, z$

**end**

**END**

## 2.14 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.

## 2.15 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  $v_C := e$ , ie a deterministic refinement, with invariant  $I_B$  and refinement relation  $v_A = v_C$ ,

then we would have to prove

$$\begin{aligned} I_A \wedge I_C \wedge v_A = v_C &\implies [v_C := e]\neg[v_A \in s]\neg(I_A \wedge I_C \wedge v_A = v_C) \\ &= I_A \wedge I_C \wedge v_A = v_C \implies [v_C := e]\neg[v_A \in s](\neg I_A \vee (v_A \neq v_C)) \\ &= I_A \wedge I_C \wedge v_A = v_C \implies [v_C := e]\neg(\forall xx.(xx : s \implies \\ &\quad [v_A := xx](\neg I_A \vee (v_A \neq v_C)))) \\ &= I_A \wedge I_C \wedge v_A = v_C \implies [v_C := e]\exists xx.(xx : s \wedge I_A \wedge I_C \wedge xx = v_C) \\ &= I_A \wedge I_C \wedge v_A = v_C \implies \exists xx.(xx : s \wedge I_A \wedge I_C \wedge xx = e) \\ &= \text{false, if } s \text{ is empty} \end{aligned}$$

This demonstrates the sting in the tail of *magic*: it is truly impossible to implement.

## Part III

# 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.

### 2.16 Queue Events

The machine has the following events:

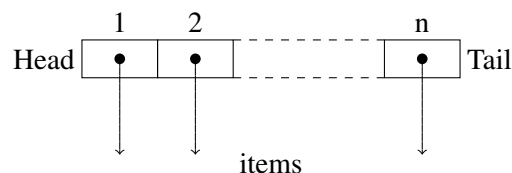
*Enqueue(item)* an operation that places an item on the end of the queue. The operation creates a unique queue identifier for this item. A unique item identifier is also generated for the item that is queued. A queue can contain multiple instances of the same item value.

*Dequeue* an operation 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.

### 2.17 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* operation 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.

### 2.18 Context Machines

#### Context Machines

Since EventB does not have a sequence type we need to define our own sequence type and accompanying functions for managing queues represented as sequences. 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 :**  $tail \in (\mathbb{N}_1 \mapsto QUEUE) \rightarrow (\mathbb{N} \mapsto QUEUE)$

**axm4 :**  $\forall n, q \cdot n \in \mathbb{N}_1 \wedge q \in 1 .. n \rightarrow QUEUE$   
 $\Rightarrow dom(tail(q)) = 1 .. n - 1$

**axm5 :**  $\forall n, q, i \cdot n \in \mathbb{N}_1 \wedge q \in 1 .. n \rightarrow QUEUE$   
 $\wedge i \in dom(tail(q)) \wedge i + 1 \in dom(q)$   
 $\Rightarrow tail(q)(i) = q(i + 1)$

**axm6 :**  $\forall n, q \cdot n \in \mathbb{N}_1 \wedge q \in 1 .. n \rightarrow QUEUE$   
 $\Rightarrow dom(tail(q)) = 1 .. n - 1$

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

**axm8 :**  $\forall d, q \cdot d \in \mathbb{N}_1 \wedge q \in \mathbb{N}_1 \mapsto QUEUE \wedge d \notin dom(q)$   
 $\Rightarrow delete(d)(q) = q$

**axm9 :**  $\forall d, q \cdot d \in \mathbb{N}_1 \wedge q \in \mathbb{N}_1 \mapsto QUEUE \wedge d \in dom(q)$   
 $\Rightarrow ran(delete(d)(q)) = ran(q) \setminus \{q(d)\}$

**axm10 :**  $\forall n, d, q \cdot n \in \mathbb{N}_1 \wedge d \in 1 .. n \wedge q \in 1 .. n \rightarrow QUEUE$   
 $\Rightarrow dom(delete(d)(q)) = 1 .. n - 1$

**axm11 :**  $\forall n, d, q, i \cdot n \in \mathbb{N}_1 \wedge d \in 1..n \wedge q \in 1..n \rightarrow QUEUE$   
 $\wedge i \in 1..d-1$   
 $\Rightarrow delete(d)(q)(i) = q(i)$

**axm12 :**  $\forall n, d, q, i \cdot n \in \mathbb{N}_1 \wedge d \in 1..n \wedge q \in 1..n \rightarrow QUEUE$   
 $\wedge i \in d..n-1$   
 $\Rightarrow delete(d)(q)(i) = q(i+1)$

**axm13 :**  $\forall n, q \cdot n \in \mathbb{N}_1 \wedge q \in 1..n \rightarrow QUEUE$   
 $\Rightarrow tail(q) = delete(1)(q)$

## THEOREMS

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

**thm2 :**  $\forall n, q \cdot n \in \mathbb{N}_1 \wedge q \in 1..n \rightarrow QUEUE$   
 $\Rightarrow ran(tail(q)) = ran(q) \setminus \{q(1)\}$

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

END

## 2.19 The Queue machine

MACHINE Queue

SEES Queue\_ctx, Item\_ctx

### VARIABLES

<i>queuetokens</i>	tokens currently in queue
<i>queue</i>	the queue of tokens
<i>queueitems</i>	a function for fetching the item associated with a token
<i>qsize</i>	current size of queue

### INVARIANTS

**inv1 :**  $queuetokens \subseteq QUEUE$

**inv2 :**  $queue \in 1..qsize \rightarrow queuetokens$

**inv3 :**  $qsize \in 0..maxqueue$

**inv4 :**  $queueitems \in queuetokens \rightarrow ITEM$

### THEOREMS

**thm1** :  $queuetokens = ran(queue)$

## EVENTS

### Initialisation

**begin**

**act1** :  $queuetokens := \emptyset$

**act2** :  $queue := \emptyset$

**act3** :  $qsize := 0$

**act4** :  $queueitems := \emptyset$

**end**

**Enqueue**  $\hat{=}$

**any**

*item*

*qid*

**where**

**grd1** :  $item \in ITEM$

**grd2** :  $qid \in QUEUE \setminus queuetokens$

**grd3** :  $qsize \neq maxqueue$

**then**

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

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

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

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

**end**

**Dequeue**  $\hat{=}$

**when**

**grd1** :  $qsize \neq 0$

**then**

```

act1 :  $queue := tail(queue)$ 

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

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

act4 :  $qsize := qsize - 1$ 

end

Unqueue  $\hat{=}$ 

any
   $qid$ 
where
  grd1 :  $qsize \neq 0$ 
  grd2 :  $qid \in queuetokens$ 
then
  act1 :  $queue := delete(queue^{-1}(qid))(queue)$ 
  act2 :  $queueitems := \{qid\} \triangleleft queueitems$ 
  act3 :  $queuetokens := queuetokens \setminus \{qid\}$ 
  act4 :  $qsize := qsize - 1$ 
end

END

```

## 2.20 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.*

### 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

<i>qfirst</i>	gives the identity of the first item in the list
<i>qnext</i> ( <i>qfirst</i> )	gives the identity of the second item in the list
<i>(qnext ; qnext)</i> ( <i>qfirst</i> )	gives the identity of the third item in the list
...	etc

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

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

### Closure

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

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

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

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

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

### 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.

---

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

## 2.21 The List Context machine

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*

### AXIOMS

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

**axm2 :**  $\forall r. r \in QUEUE \leftrightarrow QUEUE \Rightarrow iterate(r \mapsto 0) = id(QUEUE)$

**axm3 :**  $\forall r, n. r \in QUEUE \leftrightarrow QUEUE \wedge n \in \mathbb{N}_1$   
 $\Rightarrow iterate(r \mapsto n) = iterate(r \mapsto n - 1); r$

**axm4 :**  $closure1 \in (QUEUE \leftrightarrow QUEUE) \rightarrow (QUEUE \leftrightarrow QUEUE)$

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

**axm6 :**  $closure0 \in (QUEUE \leftrightarrow QUEUE) \rightarrow (QUEUE \leftrightarrow QUEUE)$

**axm7 :**  $\forall r. r \in QUEUE \leftrightarrow QUEUE$   
 $\Rightarrow closure0(r) = (\bigcup n. n \in \mathbb{N} | iterate(r \mapsto n))$

**axm8 :**  $\forall r. r \in QUEUE \leftrightarrow QUEUE$   
 $\Rightarrow closure0(r) = closure1(r) \cup id(QUEUE)$

### THEOREMS

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

**thm2 :**  $\forall f. f \in QUEUE \leftrightarrow QUEUE$   
 $\Rightarrow iterate(f \mapsto 1) = f$

**thm3 :**  $\forall f, q, n. f \in q \leftrightarrow q \wedge q \subseteq QUEUE \wedge n \in \mathbb{N}$   
 $\Rightarrow iterate(f \mapsto n) \in q \leftrightarrow q$

**END**

## 2.22 The QueueR invariant

The list consists of the elements of *queuetokens* hence

$$qsize = card(queuetokens)$$

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

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

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

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

## 2.23 The Refinement relation

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

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

## 2.24 QueueR Theorems

The following should follow from the invariant:

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

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

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

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

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

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

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

$$qsize \neq 0 \implies queue(1) = qfirst$$

$$qsize \neq 0 \implies queue(qsize) = qlast$$

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

## Loops

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

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

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

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

### 2.24.1 The QueueR machine

**MACHINE** QueueR

**REFINES** Queue

**SEES** List\_ctx, Item\_ctx

#### VARIABLES

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

#### INVARIANTS

- inv1** :  $queuetokens \subseteq QUEUE$
- inv2** :  $qsize = card(queuetokens)$
- inv3** :  $qfirst \in QUEUE$
- inv4** :  $qsize \neq 0 \Rightarrow qfirst \in queuetokens$
- inv5** :  $qlast \in QUEUE$
- inv6** :  $qsize \neq 0 \Rightarrow qlast \in queuetokens$
- inv7** :  $qnext \in queuetokens \setminus \{qlast\} \mapsto queuetokens \setminus \{qfirst\}$
- inv8** :  $qsize \neq 0 \Rightarrow qlast = iterate(qnext \mapsto qsize - 1)(qfirst)$
- inv9** :  $\forall i \cdot i \in 1 .. qsize \Rightarrow queue(i) = iterate(qnext \mapsto i - 1)(qfirst)$

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

**thm5 :**  $\forall t. t \in \text{queuetokens} \wedge \text{qsize} > 1 \wedge t \neq qfirst \Rightarrow t \in \text{ran}(\text{qnext})$

**thm6 :**  $\forall t. t \in \text{queuetokens} \wedge \text{qsize} > 1 \wedge t \neq qlast \Rightarrow t \in \text{dom}(\text{qnext})$

**thm7 :**  $\text{closure1}(\text{qnext}) \cap \text{id}(\text{queuetokens}) = \emptyset$

**thm8 :**  $\text{qnext} \in \text{QUEUE} \mapsto \text{QUEUE}$

## EVENTS

### Initialisation

**begin**

**act1 :**  $\text{queuetokens} := \emptyset$

**act2 :**  $\text{qsize} := 0$

**act3 :**  $\text{queueitems} := \emptyset$

**act4 :**  $qfirst \in \text{QUEUE}$

**act5 :**  $qlast \in \text{QUEUE}$

**act6 :**  $\text{qnext} := \emptyset$

**end**

**Enqueue0**  $\hat{=}$

**Refines** Enqueue

**any**

*item*

*qid*

**where**

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

**grd2 :**  $qid \in \text{QUEUE} \setminus \text{queuetokens}$

**grd3 :**  $\text{qsize} \neq \text{maxqueue}$

**grd4 :**  $\text{qsize} = 0$

**then**

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

**act2** :  $queueitems(qid) := item$

**act3** :  $qsize := qsize + 1$

**act4** :  $qfirst := qid$

**act5** :  $qlast := qid$

**act6** :  $qnext := \emptyset$

**end**

**Enqueue1**  $\hat{=}$

**Refines** Enqueue

**any**

*item*

*qid*

**where**

**grd1** :  $item \in ITEM$

**grd2** :  $qid \in QUEUE \setminus queuetokens$

**grd3** :  $qsize \neq maxqueue$

**grd4** :  $qsize = 1$

**then**

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

**act2** :  $queueitems(qid) := item$

**act3** :  $qsize := qsize + 1$

**act4** :  $qnext := \{qfirst \mapsto qid\}$

**act5** :  $qlast := qid$

**end**

**Enqueue2**  $\hat{=}$

**Refines** Enqueue

**any**

*item*

*qid*

**where**

**grd1** :  $item \in ITEM$

**grd2** :  $qid \in QUEUE \setminus queuetokens$

**grd3** :  $qsize \neq maxqueue$

**grd4** :  $qsize > 1$

**then**

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

**act2** :  $queueitems(qid) := item$

**act3** :  $qsize := qsize + 1$

**act4** :  $qnext := qnext \Leftarrow \{qlast \mapsto qid\}$

**act5** :  $qlast := qid$

**end**

**Dequeue0**  $\hat{=}$

**Refines** Dequeue

**when**

**grd1** :  $qsize = 1$

**then**

**act1** :  $qsize := 0$

**act2** :  $queuetokens := \emptyset$

**act3** :  $queueitems := \emptyset$

**end**

**Dequeue1**  $\hat{=}$

**Refines** Dequeue

**when**

**grd1** :  $qsize > 1$

**then**

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

**end**

**Unqueue0**  $\hat{=}$

**Refines** Unqueue

**any**

$qid$

**where**

**grd1** :  $qid \in queuetokens$

**grd2** :  $qsize = 1$

**then**

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

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

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

**end**

**Unqueue1**  $\hat{=}$

**Refines** Unqueue

**any**

$qid$

**where**

**grd1** :  $qid \in queuetokens$

**grd2** :  $qsize > 1$

**grd3** :  $qid = qfirst$

**then**

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

**end**

**Unqueue2**  $\hat{=}$

**Refines** Unqueue

**any**

$qid$

**where**

**grd1** :  $qid \in queuetokens$

**grd2** :  $qsize > 1$

**grd3** :  $qid = qlast$

**then**

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

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

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

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

**act5** :  $qnext := qnext \triangleright \{qid\}$

**end**

**Unqueue3**  $\hat{=}$

**Refines** Unqueue

**any**

$qid$

**where**

**grd1** :  $qid \in queuetokens$

```

grd2 :  $qsize > 2$ 
grd3 :  $qid \neq qfirst$ 
grd4 :  $qid \neq qlast$ 

then
act1 :  $queueitems := \{qid\} \triangleleft queueitems$ 
act2 :  $queuetokens := queuetokens \setminus \{qid\}$ 
act3 :  $qsize := qsize - 1$ 
act4 :  $qnext(qnext^{-1}(qid)) := qnext(qid)$ 

end
END

```

## 2.24.2 Refinement of Unqueue3

### Refinement of Unqueue3

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

### Implementing `prev`

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

### Preventing Interference

The refinement of `Unqueue3` consists of three events:

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

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

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

### The purpose of `deleting`

Until the deletion is complete the other queue events must not run as the state of the queue is not yet correct. Until now `Unqueue3` was an atomic event; in this refinement the actions of that event are spread across three events.

## QueueRR

**MACHINE** QueueRR

**REFINES** QueueR

**SEES** List\_ctx, Item\_ctx

### VARIABLES

<i>queuetokens</i>	tokens currently in queue
<i>queueitems</i>	a function for fetching the item associated with a token
<i>qsize</i>	current size of queue
<i>qfirst</i>	first item, if any, in queue
<i>qnext</i>	link to next item, if any, in queue
<i>qlast</i>	last item, if any, in queue
<i>qprev</i>	previous item
<i>deleting</i>	

### INVARIANTS

**inv1** :  $qprev \in QUEUE$

**inv2** :  $deleting \in BOOL$

### EVENTS

#### Initialisation

*inherited*

**Enqueue0**  $\hat{=}$

*inherited*

**Enqueue1**  $\hat{=}$

*inherited*

**Enqueue2**  $\hat{=}$

*inherited*

**Dequeue0**  $\hat{=}$

*inherited*

**Dequeue1**  $\hat{=}$

*inherited*

**Unqueue0**  $\hat{=}$

*inherited*

**Unqueue1**  $\hat{=}$

*inherited*

**Unqueue2**  $\hat{=}$

*inherited*

**Unqueue3**  $\hat{=}$

**Refines** Unqueue3

**any**

*qid*

**where**

**grd1** :  $qid \in \text{queuetokens}$

**grd2** :  $qsize > 2$

**grd3** :  $qid \neq qfirst$

**grd4** :  $qid \neq qlast$

**grd7** :  $qprev \in \text{dom}(qnext)$

**grd5** :  $qnext(qprev) = qid$

**grd6** :  $deleting = TRUE$

**then**

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

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

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

**act4** :  $qnext(qprev) := qnext(qid)$

**act5** :  $deleting := FALSE$

**end**

**Unqueue3I**  $\hat{=}$

**any**

$qid$   
**where**  
**grd1** :  $qid \in queuetokens$   
**grd2** :  $qsize > 1$   
**grd3** :  $qid \neq qfirst$   
**grd4** :  $qid \neq qlast$   
**grd5** :  $deleting = FALSE$   
**then**  
**act1** :  $qprev := qfirst$   
**act2** :  $deleting := TRUE$   
**end**  
**Unqueue3S**  $\hat{=}$   
**Which is convergent**  
**any**  
 $qid$   
**where**  
**grd1** :  $qid \in queuetokens$   
**grd2** :  $qsize > 1$   
**grd3** :  $qid \neq qfirst$   
**grd4** :  $qid \neq qlast$   
**grd5** :  $qprev \in dom(qnext)$   
**grd6** :  $qnext(qprev) \neq qid$   
**grd7** :  $deleting = TRUE$   
**then**  
**act1** :  $qprev := qnext(qprev)$   
**end**  
**VARIANT**  
 $closure1(qnext)[\{qprev\}]$   
**END**

### 2.24.3 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 be *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*.