

# System Modelling and Design

Modelling:

Sorting Algorithms

Revision: 1.3, May 9, 2008

Ken Robinson

May 18, 2010

©Ken Robinson 2005-2010

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

## Contents

<b>1</b>	<b>The Specification of Sorting</b>	<b>1</b>
1.1	Abstract Specification of Sorting . . . . .	4
<b>2</b>	<b>Insertion Sort</b>	<b>5</b>
2.1	The Insertion Sort Plan . . . . .	6
2.2	InsertionSortR1: More Refinement . . . . .	8
2.3	InsertionSortR2: Discovering m and n . . . . .	9
2.4	InsertionSortR3: The Concrete Algorithm . . . . .	12
<b>3</b>	<b>HeapSort</b>	<b>14</b>

## Objectives of this Lecture

- to model a number of sorting algorithms to illustrate the use of modelling to gain understanding of a proposed design;
- the objective is understanding, not sorting algorithms in themselves;
- to illustrate the refinement process

## 1 The Specification of Sorting

In this development we will describe sorting an injective sequence of numbers. Making the sequence injective avoids having to deal with multiple instances of the same value in the sequence. This is done

to make the process just a little simpler.

The following context contains the definitions required for specifying sequences and a predicate function  $isSorted(s)(m)(n)$  for determining whether the sequence  $s$  is sorted (monotonically increasing) in the domain subrange  $m .. n$ .

**CONTEXT** `Sorting_ctx`

**CONSTANTS** *length*    The length of a sequence    *ISEQ*    The set of injective sequences  
*DOM0*    Domain of possibly empty sequences    *DOM1*    Domain of non-empty sequences  
*PERM*    The set of sequence permutations    *isSORTED*    Predicate for determining sortedness  
*UNSORTED*    An arbitrary sequence

**AXIOMS**

*axm1:*  $length \in \mathbb{N}_1$

*axm2:*  $DOM0 = 0 .. length$

*axm3:*  $DOM1 = 1 .. length$

*axm4:*  $ISEQ = DOM1 \rightarrow \mathbb{N}$     All injective sequences of natural numbers with domain *DOM1*

*axm8:*  $PERM = DOM1 \rightarrow DOM1$

*axm9:*  $isSORTED \in ISEQ \rightarrow (DOM1 \rightarrow (DOM0 \rightarrow BOOL))$

*axm10:*  $\forall s, m, n. s \in ISEQ$   
 $\wedge m \in DOM1$   
 $\wedge n \in DOM0$   
 $\Rightarrow$   
 $isSORTED(s)(m)(n)$   
 $= bool(\forall i, j. i \in m .. n \wedge j \in m .. n \wedge i < j \Rightarrow s(i) < s(j))$

*axm11:*  $UNSORTED \in ISEQ$

**THEOREMS**

*thm1:*  $\forall m, n. m \in DOM1 \wedge n \in DOM1$   
 $\Rightarrow$   
 $\{i \cdot i \in m .. n \mid i + 1 \mapsto i\} \in m + 1 .. n + 1 \rightarrow m .. n$

*thm2:*  $\forall m, n. m \in DOM1 \wedge n \in DOM1$   
 $\Rightarrow$   
 $dom(\{i \cdot i \in m .. n \mid i + 1 \mapsto i\}) = m + 1 .. n + 1$

*thm3:*  $\forall m, n. m \in DOM1 \wedge n \in DOM1$   
 $\Rightarrow$   
 $ran(\{i \cdot i \in m .. n \mid i + 1 \mapsto i\}) = m .. n$

*thm4:*  $dom(isSORTED) = ISEQ$

*thm5:*  $\forall s. s \in ISEQ$   
 $\Rightarrow$   
 $dom(isSORTED(s)) = dom(s)$

*thm6:*  $\forall s, m \cdot s \in ISEQ$   
 $\wedge m \in \text{dom}(s)$   
 $\Rightarrow$   
 $\text{dom}(\text{isSORTED}(s)(m)) = \text{DOM0}$

*thm7:*  $\forall s, m, n \cdot s \in ISEQ$   
 $\wedge m \in \text{DOM1}$   
 $\wedge n \in \text{DOM0}$   
 $\wedge m > n$   
 $\Rightarrow$   
 $\text{isSORTED}(s)(m)(n) = \text{TRUE}$

*thm8:*  $\forall s, m \cdot s \in ISEQ$   
 $\wedge m \in \text{DOM1}$   
 $\Rightarrow$   
 $\text{isSORTED}(s)(m)(m) = \text{TRUE}$

*thm9:*  $\forall s, m, n \cdot s \in ISEQ$   
 $\wedge m \in \text{DOM1}$   
 $\wedge n \in \text{DOM1}$   
 $\wedge n + 1 \in \text{DOM1}$   
 $\wedge \text{isSORTED}(s)(m)(n) = \text{TRUE}$   
 $\wedge s(n) < s(n + 1)$   
 $\Rightarrow$   
 $\text{isSORTED}(s)(m)(n + 1) = \text{TRUE}$

*thm10:*  $\forall s, l, m, n \cdot s \in ISEQ$   
 $\wedge l \in \text{DOM1}$   
 $\wedge m \in \text{DOM1}$   
 $\wedge n \in \text{DOM0}$   
 $\wedge l \leq m \wedge m \leq n$   
 $\wedge \text{isSORTED}(s)(l)(m) = \text{TRUE}$   
 $\wedge \text{isSORTED}(s)(m)(n) = \text{TRUE}$   
 $\Rightarrow$   
 $\text{isSORTED}(s)(l)(n) = \text{TRUE}$

*thm11:*  $\forall s \cdot s \in ISEQ$   
 $\Rightarrow$   
 $s^{-1}; s = \text{id}(\text{ran}(s))$

*thm12:*  $\forall s, t \cdot s \in ISEQ$   
 $\wedge t \in ISEQ$   
 $\wedge \text{ran}(s) = \text{ran}(t)$   
 $\Rightarrow$   
 $s; t^{-1}; t = s$

*thm13:*  $\forall s \cdot s \in ISEQ$   
 $\Rightarrow$   
 $s; \text{id}(\text{ran}(s)) = s$

*thm14:*  $\forall p, s \cdot p \in PERM$   
 $\wedge s \in ISEQ$   
 $\Rightarrow$   
 $p; s \in ISEQ$

*thm15:*  $\forall m, p, s \cdot 1 \leq m$   
 $\wedge m \leq length$   
 $\wedge p \in 1..m \mapsto 1..m$   
 $\wedge s \in ISEQ$   
 $\Rightarrow$   
 $dom(p; s) = 1..m$

*thm16:*  $\forall s \cdot s \in ISEQ$   
 $\Rightarrow$   
 $s; s^{-1} = id(DOM1)$

*thm17:*  $\forall s \cdot s \in ISEQ$   
 $\Rightarrow$   
 $s^{-1}; s = id(ran(s))$

**END**

## 1.1 Abstract Specification of Sorting

In the following **Sort** machine, the **Sort** event creates a sorted sequence by proposing a permutation  $p$  that will transform the unsorted sequence  $u$  into a sorted sequence  $s$  by relational composition:

$$p; u = s$$

Composing both sides of the equality on the right by  $u^{-1}$  gives

$$p; u; u^{-1} = s; u^{-1}$$

giving the required permutation as

$$p = s; u^{-1}$$

Of course, that's all very well after the fact, and it is the job of a sorting algorithm to effectively compute that permutation.

In the specification of **Sort** and subsequent refinements,

- *tosort* represents the sequence being sorted.
- *tosort* contains the same values as the unsorted sequence, *UNSORTED*, and
- and at any time the subsequence

$$1..sorted \triangleleft tosort$$

is currently sorted.

**MACHINE** Sort

**SEES** Sorting\_ctx

**VARIABLES** *tosort* Sequence to be sorted *sorted* *tosort* is ordered from 1 to *sorted*

**INVARIANTS**

*inv2*:  $tosort \in ISEQ$

*inv1*:  $sorted \in DOM1$

*inv3*:  $isSORTED(tosort)(1)(sorted) = TRUE$

*inv4*:  $tosort; UNSORTED^{-1} \in PERM$

**EVENTS**

**Initialisation**

**begin**

*act1*:  $tosort := UNSORTED$

*act2*:  $sorted := 1$

**end**

**Event**  $sort \hat{=}$

**any**  $perm$

**when**

*grd1*:  $perm \in PERM$

*grd2*:  $isSORTED(perm; UNSORTED)(1)(length) = TRUE$

**then**

*act1*:  $tosort := (perm; UNSORTED)$

*act2*:  $sorted := length$

**end**

**END**

## 2 Insertion Sort

A very simple sorting algorithm is insertion sort. In modelling insertion sort we are interested in motivating the algorithm, rather than in the algorithm itself.

The InsertSort context specifies some functions that build permutations that can be used to move elements of a sequence:

*SHIFTUP*( $d$ ) shifts elements in the set  $d$  up one position in the sequence;

*SWAP*( $m \mapsto n$ ) swaps the values at positions  $m$  and  $n$ .

Some theorems are also presented that express preservation of the notion of sorting under SHIFTUP.

## 2.1 The Insertion Sort Plan

The plan for insertion sort, shown in the the refinement InsertionSort is as follows:

1. We discover that the sequence is sorted in the range  $1 .. m$ , but
2.  $tosorti(m + 1) < tosorti(m)$ , so  $tosorti(m + 1)$  is not in the correct position for being sorted.
3. We discover a permutation that will result in  $tosorti$  being sorted in the range  $1 .. m + 1$

This plan is represented by the new convergent event **Insert**.

It contains two discoveries, which will be the subject of further refinement.

The refinement of the original event **Sort** simply waits until  $sortedi = length$ , that is the sequence is sorted.

**MACHINE** InsertionSort

This refinement of Sort leads to the insertion sort solution

**REFINES** Sort

**SEES** Sorting\_ctx

**VARIABLES**  $tosorti$  Sequence to be sorted by InsertionSort  $sortedi$   $tosorti$  is ordered  
from 1 to  $sortedi$

### INVARIANTS

*inv1:*  $tosorti \in ISEQ$

*inv2:*  $sortedi \in DOM1$

*inv3:*  $isSORTED(tosorti)(1)(sortedi) = TRUE$

*inv4:*  $tosorti; UNSORTED^{-1} \in PERM$

*inv5:*  $sortedi = length \Rightarrow sortedi = sorted$

*inv6:*  $sortedi = length \Rightarrow tosorti = tosort$

### EVENTS

#### Initialisation

**begin**

*act1:*  $tosorti := UNSORTED$

*act2:*  $sortedi := 1$

**end**

**Event**  $sort \hat{=}$

**refines**  $sort$

**when**

**grd1:**  $sortedi = length$   
**with**  
**perm :**  $perm = tosorti; UNSORTED^{-1}$   
**then**  $skip$   
**end**  
**Event**  $insert \hat{=}$   
**Status**  $convergent$   
**any**  
 $m$   
 $perm$   
**when**  
**grd1:**  $sortedi \neq length$   
**grd2:**  $m \in DOM1$   
**grd3:**  $m \geq sortedi$   
**grd4:**  $m \neq length$   
**grd5:**  $isSORTED(tosorti)(1)(m) = TRUE$   
**grd6:**  $tosorti(m + 1) < tosorti(m)$   
**grd7:**  $perm \in 1 .. m + 1 \mapsto 1 .. m + 1$   
**grd8:**  $isSORTED(perm; tosorti)(1)(m + 1) = TRUE$   
**then**  
**act1:**  $tosorti := tosorti \Leftarrow (perm; tosorti)$   
**act2:**  $sortedi := m + 1$   
**end**  
**VARIANT**  $length - sortedi$   
**END**

## 2.2 InsertionSortR1: More Refinement

To the previous plan we add:

1. A discovery that there is an element at position  $n$ , in the range  $1 .. m$  with  $tosort0(n) < tosort0(m + 1) < tosort0(n + 1)$ .

The required permutation would then move the item at position  $m + 1$  to position  $n + 1$  and to shift up all items in the range  $n + 1 .. m$ .

There are still two discoveries: the positions  $m$  and  $n$ .

**MACHINE** InsertionSortR1

**REFINES** InsertionSort

**SEES** InsertSort\_ctx

**VARIABLES** *tosorti* Sequence to be sorted by InsertionSort *sortedi* *tosorti* is ordered from 1 to *sortedi*

**EVENTS**

**Initialisation** *extended*

**begin**

*act1* : *tosorti* := UNSORTED

*act2* : *sortedi* := 1

**end**

**Event** *sort*  $\hat{=}$

**extends** *sort*

**when**

*grd1* : *sortedi* = *length*

**then** *skip*

**end**

**Event** *insert*  $\hat{=}$

**refines** *insert*

**any**

*m*

*perm1* a new perm

*n*

**when**

*grd1:*  $sortedi \neq length$   
*grd2:*  $m \in DOM1$   
*grd3:*  $m \geq sortedi$   
*grd4:*  $m \neq length$   
*grd5:*  $isSORTED(tosorti)(1)(m) = TRUE$   
*grd6:*  $tosorti(m + 1) < tosorti(m)$   
*grd7:*  $perm1 \in n .. m + 1 \mapsto n .. m + 1$   
*grd8:*  $n \in 1 .. m$  position for insertion of (m+1)th element  
*grd9:*  $tosorti(m + 1) < tosorti(n)$   
*grd10:*  $n \neq 1 \Rightarrow tosorti(n - 1) < tosorti(m + 1)$   
*grd11:*  $perm1 = ROTATEUP(n \mapsto m + 1)$

**with**

*perm* :  $perm = 1 .. n - 1 \triangleleft id \triangleleft perm1$

**then**

*act1:*  $tosorti := tosorti \triangleleft (perm1; tosorti)$   
*act2:*  $sortedi := m + 1$

**end**

**END**

### 2.3 InsertionSortR2: Discovering m and n

In this refinement we add events that discover the  $m$  and  $n$  values, rather than them being declaratively specified.

This discovery is achieved by three new events:

**scanforward** scans forward from the current maximum sorted position (*sortedi*) to discover  $m$ ;

**reverse** sets up the backward scan once  $m$  has been found;

**scanbackward** scans backward to determine  $n$

Once  $m$  and  $n$  have been determined the *Insert* event uses *ROTATEUP* to insert the  $m + 1$ th elements in the correct place.

Notice the witnesses for  $m$ ,  $n$  and *perm1* as these parameters have been deleted.

**MACHINE** InsertionSortR2

**REFINES** InsertionSortR1

**SEES** InsertSort\_ctx

**VARIABLES** *tosorti* Sequence to be sorted by InsertionSort *sortedi1* *tosorti1* is ordered  
from 1 to *sortedi1* *moveto* where to move unsorted element to

**INVARIANTS**

*inv1:*  $sortedi1 \in DOM1$

*inv2:*  $isSORTED(tosorti)(1)(sortedi1) = TRUE$

*inv3:*  $sortedi1 = length \Rightarrow sortedi = sortedi1$

*inv4:*  $moveto \in DOM0$

*inv5:*  $moveto \neq 0 \Rightarrow moveto \in 1 .. sortedi1$

*inv6:*  $moveto \neq 0 \Rightarrow tosorti(sortedi1 + 1) < tosorti(moveto)$

**EVENTS**

**Initialisation**

**begin**

*act1:*  $tosorti := UNSORTED$

*act2:*  $sortedi1 := 1$

*act3:*  $moveto := 0$

**end**

**Event**  $sort \hat{=}$

**refines**  $sort$

**when**

*grd1:*  $sortedi1 = length$

**then**  $skip$

**end**

**Event**  $insert \hat{=}$

**refines**  $insert$

**when**

*grd1:*  $sortedi1 \neq length$

*grd2:*  $moveto \neq 0$

*grd3:*  $tosorti(moveto) < tosorti(sortedi1 + 1)$

*grd4:*  $moveto \neq 1 \Rightarrow tosorti(moveto - 1) < tosorti(sortedi1 + 1)$

**with**

*m* :  $m = \text{sortedi1}$

*n* :  $n = \text{moveto}$

*perm1* :  $\text{perm1} = \text{ROTATEUP}(\text{moveto} \mapsto \text{sortedi1} + 1)$

**then**

*act1*:  $\text{tosorti} := \text{tosorti}$   
 $\Leftarrow$   
 $(\text{ROTATEUP}(\text{moveto} \mapsto \text{sortedi1} + 1); \text{tosorti})$

*act2*:  $\text{sortedi1} := \text{sortedi1} + 1$

*act3*:  $\text{moveto} := 0$

**end**

**Event** *scanforward*  $\hat{=}$

**Status** convergent

**when**

*grd1*:  $\text{sortedi1} \neq \text{length}$

*grd2*:  $\text{tosorti}(\text{sortedi1}) < \text{tosorti}(\text{sortedi1} + 1)$

*grd3*:  $\text{moveto} = 0$

**then**

*act1*:  $\text{sortedi1} := \text{sortedi1} + 1$

**end**

**Event** *reverse*  $\hat{=}$

**when**

*grd1*:  $\text{sortedi1} \neq \text{length}$

*grd2*:  $\text{tosorti}(\text{sortedi1}) > \text{tosorti}(\text{sortedi1} + 1)$

**then**

*act1*:  $\text{moveto} := \text{sortedi1}$

**end**

**Event** *scanbackward*  $\hat{=}$

**Status** convergent

**when**

*grd1*:  $\text{moveto} > 1$

*grd2*:  $tosorti(sortedi1 + 1) < tosorti(moveto - 1)$

**then**

*act1*:  $moveto := moveto - 1$

**end**

**VARIANT**

$length + moveto - sortedi1$

**END**

## 2.4 InsertionSortR3: The Concrete Algorithm

This refinement refines *scanbackward*—rename *backwardinsert* to perform swaps as it scans backward. The idea is that by the time the final insertion point is found the sequence is sorted in the range  $1..m+1$ . Consequently, the *insert* event has only bookkeeping to do.

**MACHINE** InsertionSortR3

**REFINES** InsertionSortR2

**SEES** InsertSort.ctx

**VARIABLES** *sortedi1* *tosorti1* is ordered from 1 to *sortedi1* *tosorti1* Sequence to be sorted  
in InsertionSortR3 *tomove* element to be moved

**INVARIANTS**

*inv1*:  $tosorti1 \in ISEQ$

*inv2*:  $tomove = 0 \Rightarrow isSORTED(tosorti1)(1)(sortedi1) = TRUE$

*inv3*:  $tomove \neq 0 \Rightarrow$   
 $isSORTED(tosorti1)(1)(moveto - 1) = TRUE$   
 $\wedge$   
 $isSORTED(tosorti1)(moveto)(sortedi1 + 1) = TRUE$

*inv4*:  $sortedi1 = length \Rightarrow tosorti1 = tosorti$

**EVENTS**

**Initialisation**

**begin**

*act1*:  $sortedi1 := 1$

*act2*:  $tosorti1 := UNSORTED$

*act3*:  $tomove := 0$

**end**

```

Event sort  $\hat{=}$ 
extends sort
when
grd1: sortedi1 = length
then skip
end
Event insert  $\hat{=}$ 
refines insert
when
grd1: sortedi1  $\neq$  length
grd2: tomove  $\neq$  0
grd3: tomove  $\neq$  1  $\Rightarrow$  tosorti1(tomove - 1) < tosorti1(tomove)
then
act1: sortedi1 := sortedi1 + 1
act2: tomove := 0
end
Event scanforward  $\hat{=}$ 
refines scanforward
when
grd1: sortedi1  $\neq$  length
grd2: tosorti1(sortedi1) < tosorti1(sortedi1 + 1)
grd3: tomove = 0
then
act1: sortedi1 := sortedi1 + 1
end
Event reverse  $\hat{=}$ 
refines reverse
when
grd1: sortedi1  $\neq$  length
grd2: tosorti1(sortedi1) > tosorti1(sortedi1 + 1)

```

```

then
act1: tomove := sortedi1 + 1
end

Event backwardinsert  $\hat{=}$ 
refines scanbackward

when
grd1: tomove > 1
grd2: tosorti1(tomove) < tosorti1(tomove - 1)
then
act1: tosorti1 := tosorti1
       $\Leftarrow$ 
      (SWAP(tomove  $\mapsto$  tomove - 1); tosorti1)
act2: tomove := tomove - 1
end
END

```

### 3 HeapSort

We will now refine the initial **Sort** machine to use the *heapsort* —sometimes called *treesort*— algorithm.