Ken Robinson

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

May 18, 2010

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

The Specification of Sorting

Abstract Specification of Sorting

Insertion Sort

The Insertion Sort Plan

InsertionSortR1: More Refinement
InsertionSortR2: Discovering m and n
InsertionSortR3: The Concrete Algorithm

HeapSort
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
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
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
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
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 $\text{isSorted}(s)(m)(n)$ for determining whether the sequence $s$ is sorted (monotonically increasing) in the domain subrange $m .. n$. 
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 $m..n$. 
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

\textbf{axm1: } \text{length} \in \mathbb{N}_1

\textbf{axm2: } \text{DOM0} = 0 \ldots \text{length}

\textbf{axm3: } \text{DOM1} = 1 \ldots \text{length}

\textbf{axm4: } \text{ISEQ} = \text{DOM1} \mapsto \mathbb{N} \quad \text{All injective sequences of natural numbers with domain } \text{DOM1}

\textbf{axm8: } \text{PERM} = \text{DOM1} \mapsto \text{DOM1}

\textbf{axm9: } \text{isSORTED} \in \text{ISEQ} \rightarrow (\text{DOM1} \rightarrow (\text{DOM0} \rightarrow \text{BOOL}))

\textbf{axm10: }

\forall s, m, n \cdot s \in \text{ISEQ} \\
\quad \wedge m \in \text{DOM1} \\
\quad \wedge n \in \text{DOM0} \\
\Rightarrow \\
\text{isSORTED}(s)(m)(n) = \text{bool}(\forall i, j \cdot i \in m \ldots n \wedge j \in m \ldots n \wedge i < j \Rightarrow s(i) < s(j))

\textbf{axm11: } \text{UNSORTED} \in \text{ISEQ}
THEOREMS

\textbf{thm1:} \quad \forall m, n \cdot m \in \text{DOM}1 \land n \in \text{DOM}1 \\
\quad \Rightarrow \\
\quad \{i \cdot i \in m .. n|i + 1 \mapsto i\} \in m + 1 .. n + 1 \implies m .. n

\textbf{thm2:} \quad \forall m, n \cdot m \in \text{DOM}1 \land n \in \text{DOM}1 \\
\quad \Rightarrow \\
\quad \text{dom}(\{i \cdot i \in m .. n|i + 1 \mapsto i\}) = m + 1 .. n + 1

\textbf{thm3:} \quad \forall m, n \cdot m \in \text{DOM}1 \land n \in \text{DOM}1 \\
\quad \Rightarrow \\
\quad \text{ran}(\{i \cdot i \in m .. n|i + 1 \mapsto i\}) = m .. n

\textbf{thm4:} \quad \text{dom(isSORTED)} = \text{ISEQ}
Outline
Objectives of this Lecture
The Specification of Sorting

\( \text{thm5: } \forall s \cdot s \in ISEQ \)
\[ \Rightarrow \]
\[ \text{dom}(\text{isSORTED}(s)) = \text{dom}(s) \]

\( \text{thm6: } \forall s, m \cdot s \in ISEQ \)
\[ \land m \in \text{dom}(s) \]
\[ \Rightarrow \]
\[ \text{dom}(\text{isSORTED}(s)(m)) = \text{DOM0} \]

\( \text{thm7: } \forall s, m, n \cdot s \in ISEQ \)
\[ \land m \in \text{DOM1} \]
\[ \land n \in \text{DOM0} \]
\[ \land m > n \]
\[ \Rightarrow \]
\[ \text{isSORTED}(s)(m)(n) = \text{TRUE} \]

\( \text{thm8: } \forall s, m \cdot s \in ISEQ \)
\[ \land m \in \text{DOM1} \]
\[ \Rightarrow \]
\[ \text{isSORTED}(s)(m)(m) = \text{TRUE} \]
**thm9**: \[\forall s, m, n \cdot s \in ISEQ \]
\[\land m \in DOM1\]
\[\land n \in DOM1\]
\[\land n + 1 \in DOM1\]
\[\land isSORTED(s)(m)(n) = TRUE\]
\[\land s(n) < s(n + 1)\]
\[\Rightarrow\]
\[isSORTED(s)(m)(n + 1) = TRUE\]
**thm10:** \( \forall s, l, m, n \cdot s \in ISEQ \)
\[ \land l \in \text{DOM1} \]
\[ \land m \in \text{DOM1} \]
\[ \land n \in \text{DOM0} \]
\[ \land l \leq m \land m \leq n \]
\[ \land \text{isSORTED}(s)(l)(m) = \text{TRUE} \]
\[ \land \text{isSORTED}(s)(m)(n) = \text{TRUE} \]
\[ \Rightarrow \]
\[ \text{isSORTED}(s)(l)(n) = \text{TRUE} \]
\textbf{thm11:} \quad \forall s \cdot s \in ISEQ \\
\Rightarrow \quad s^{-1}; s = \text{id}(\text{ran}(s))

\textbf{thm12:} \quad \forall s, t \cdot s \in ISEQ \\
\land t \in ISEQ \\
\land \text{ran}(s) = \text{ran}(t) \\
\Rightarrow \quad s; t^{-1}; t = s

\textbf{thm13:} \quad \forall s \cdot s \in ISEQ \\
\Rightarrow \quad s; \text{id}(\text{ran}(s)) = s
thm14: \( \forall p, s \cdot p \in \text{PERM} \land s \in \text{ISEQ} \Rightarrow p; s \in \text{ISEQ} \)

thm15: \( \forall m, p, s \cdot 1 \leq m \land m \leq \text{length} \land p \in 1..m \Rightarrow 1..m \land s \in \text{ISEQ} \Rightarrow \text{dom}(p; s) = 1..m \)
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
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.
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.
Abstract Specification of Sorting

In the following \textbf{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.
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.
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

\[
1..sorted \triangle tosort
\]

is currently sorted.
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
  
  $1..\text{sorted} \triangleleft \text{tosort}$

  is currently sorted.
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

\[ 1..\text{sorted} \triangleleft \text{tosort} \]

is currently sorted.
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

\[ 1..\text{sorted} \prec tosort \]

is currently sorted.
MACHINE Sort

SEES Sorting_ctx

VARIABLES

tosort
Sequence to be sorted

sorted
tosort is ordered from 1 to sorted

INVARINTS

inv2: tosort ∈ ISEQ

inv1: sorted ∈ DOM1

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

inv4: tosort; UNSORTED⁻¹ ∈ PERM
EVENTS
Initialisation
begin
act1: tosort := UNSORTED
act2: sorted := 1
end
Event $\text{sort} \Leftarrow$

any $\text{perm}$

when

$\text{grd1} : \text{perm} \in \text{PERM}$

$\text{grd2} : \text{isSORTED}(\text{perm}; \text{UNSORTED})(1)(\text{length}) = \text{TRUE}$

then

$\text{act1} : \text{tosort} := (\text{perm}; \text{UNSORTED})$

$\text{act2} : \text{sorted} := \text{length}$

end

END
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:

\[ \text{SHIFTUP}(d) \] shifts elements in the set \( d \) up one position in the sequence;

\[ \text{SWAP}(m \mapsto n) \] swaps the values at positions \( m \) and \( n \).

Some theorems are also presented that express preservation of the notion of sorting under \text{SHIFTUP}.
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 $SHIFTUP$. 
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.
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:

\( \text{SHIFTUP}(d) \) shifts elements in the set \( d \) up one position in the sequence;

\( \text{SWAP}(m \mapsto n) \) swaps the values at positions \( m \) and \( n \).

Some theorems are also presented that express preservation of the notion of sorting under SHIFTUP.
InsertionSort: The Insertion Sort Plan

The plan for insertion sort, shown in 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.
InsertionSort: The Insertion Sort Plan

The plan for insertion sort, shown in 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.
InsertionSort: The Insertion Sort Plan

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

1. We discover that the sequence is sorted in the range $1 \ldots m$, 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 $1 \ldots m + 1$

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.
InsertionSort: The Insertion Sort Plan

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

1. We discover that the sequence is sorted in the range $1 \ldots m$, 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 $1 \ldots m + 1$

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.
InsertionSort: The Insertion Sort Plan

The plan for insertion sort, shown in 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.
InsertionSort: The Insertion Sort Plan

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

1. We discover that the sequence is sorted in the range \(1 \ldots m\), 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 \(1 \ldots m+1\)

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
INVIARANTS

inv1: tosorti ∈ ISEQ
inv2: sortedi ∈ DOM1
inv3: isSORTED(tosorti)(1)(sortedi) = TRUE
inv4: tosorti; UNSORTED⁻¹ ∈ PERM
inv5: sortedi = length ⇒ sortedi = sorted
inv6: sortedi = length ⇒ tosorti = tosort
EVENTS

Initialisation

begin

act1: tosorti := UNSORTED

act2: sortedi := 1

end
Event \( \text{sort} \triangleq \)
refines \( \text{sort} \)
when
\( \text{grd}1: \text{sortedi} = \text{length} \)
with
\( \text{perm}: \text{perm} = \text{tosorti}; \text{UNSORTED}^{-1} \)
then \( \text{skip} \)
end
Event \textit{insert} \supseteq

Status convergent

\begin{align*}
\text{any} & \quad m \\
\text{perm} & \\
\text{when} & \\
\text{grd}1 & : \quad \text{sortedi} \neq \text{length} \\
\text{grd}2 & : \quad m \in \text{DOM1} \\
\text{grd}3 & : \quad m \geq \text{sortedi} \\
\text{grd}4 & : \quad m \neq \text{length} \\
\text{grd}5 & : \quad \text{isSORTED(tosorti)(1)(m)} = \text{TRUE} \\
\text{grd}6 & : \quad \text{tosorti}(m + 1) < \text{tosorti}(m) \\
\text{grd}7 & : \quad \text{perm} \in 1..m + 1 \rightarrow 1..m + 1
\end{align*}
\textit{grd8:} \quad \texttt{isSORTED}(perm; tosorti)(1)(m + 1) = \texttt{TRUE} \\
\text{then} \\
\textit{act1:} \quad \texttt{tosorti} := \texttt{tosorti} \leftarrow (perm; \texttt{tosorti}) \\
\textit{act2:} \quad \texttt{sortedi} := m + 1 \\
\text{end}
VARIANT

\[
\text{length} - \text{sorted}\]

END
InsertionSortR1: More Refinement

To the previous plan we add:

1. A discovery that there is an element at position $n$, in the range $1 \ldots m$ with $\text{tosort}_0(n) < \text{tosort}_0(m + 1) < \text{tosort}_0(n + 1)$.

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 $n + 1 \ldots m$.

There are still two discoveries: the positions $m$ and $n$. 
InsertionSortR1: More Refinement

To the previous plan we add:

1. A discovery that there is an element at position $n$, in the range $1 \ldots m$ with $\text{tosort}0(n) < \text{tosort}0(m + 1) < \text{tosort}0(n + 1)$.

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 $n + 1 \ldots m$.

There are still two discoveries: the positions $m$ and $n$. 
To the previous plan we add:

1. A discovery that there is an element at position \( n \), in the range 
   \( 1 \ldots m \) with \( \text{tosort}_0(n) < \text{tosort}_0(m + 1) < \text{tosort}_0(n + 1) \).

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 \( n + 1 \ldots m \).

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 \( \text{sort} \supseteq \)

extends \( \text{sort} \)

when

\( \text{grd1} : \text{sortedi} = \text{length} \)

then \( \text{skip} \)

end
Event \( \text{insert} \) \( \supseteq \) refines \( \text{insert} \)

any \( m \)

\( \text{perm1} \) \( a \text{ new perm} \)

\( \text{n} \)

when

\( \text{grd1: } \) \( \text{sortedi} \neq \text{length} \)

\( \text{grd2: } m \in \text{DOM1} \)

\( \text{grd3: } m \geq \text{sortedi} \)

\( \text{grd4: } m \neq \text{length} \)

\( \text{grd5: } \text{isSORTED(tosorti)(1)(m)} = \text{TRUE} \)

\( \text{grd6: } \text{tosorti}(m + 1) < \text{tosorti}(m) \)
grd7: \( perm_1 \in n .. m + 1 \mapsto n .. m + 1 \)

grd8: \( n \in 1 .. m \) \hspace{1cm} \text{position for insertion of \((m+1)\)th element}

grd9: \( tosort_i(m + 1) < tosort_i(n) \)

grd10: \( n \neq 1 \Rightarrow tosort_i(n - 1) < tosort_i(m + 1) \)

grd11: \( perm_1 = \text{ROTATEUP}(n \mapsto m + 1) \)

with

\( perm: \) \( perm = 1 .. n - 1 \triangleleft id \triangleleft perm_1 \)

then

act1: \( tosort_i := tosort_i \triangleleft (perm_1; tosort_i) \)

act2: \( sorted_i := m + 1 \)

end

END
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 (sorted1) 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 \texttt{ROTATEUP} to insert the $m + 1$th elements in the correct place.

Notice the witnesses for $m$, $n$ and \texttt{perm1} as these parameters have been deleted.
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 (sortedi1 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.
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 (\( \text{sorted}\_i1 \) 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 \( \text{perm1} \) as these parameters have been deleted.
**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 (*sortedi1* 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.
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 (\( sortedi1 \) 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.
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 (*sor*ed1 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.
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 ($\text{sortedi1}$ 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 $\text{perm1}$ as these parameters have been deleted.
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 (sortedi1 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
INukiNANTS

inv1: \( \text{sortedi1} \in \text{DOM1} \)
inv2: \( \text{isSORTED(tosorti)(1)(sortedi1)} = \text{TRUE} \)
inv3: \( \text{sortedi1} = \text{length} \Rightarrow \text{sortedi} = \text{sortedi1} \)
inv4: \( \text{moveto} \in \text{DOM0} \)
inv5: \( \text{moveto} \neq 0 \Rightarrow \text{moveto} \in 1 \ldots \text{sortedi1} \)
inv6: \( \text{moveto} \neq 0 \Rightarrow \text{tosorti(sortedi1 + 1)} < \text{tosorti(moveto)} \)
EVENTS

Initialisation

begin
  act1: tosorti := UNSORTED
  act2: sortedi1 := 1
  act3: moveto := 0
end
Event $\text{sort} \trianglerighteq$
refines $\text{sort}$
when
$\text{grd1}: \text{sortedi1} = \text{length}$
then $\text{skip}$
end
Event  \textit{insert} \supseteq \\
refines  \textit{insert} \\
when \\
\textit{grd1}:  \textit{sortedi1} \neq \textit{length} \\
\textit{grd2}:  \textit{moveto} \neq 0 \\
\textit{grd3}:  \textit{tosorti}(\textit{moveto}) < \textit{tosorti}(\textit{sortedi1} + 1) \\
\textit{grd4}:  \textit{moveto} \neq 1 \Rightarrow \textit{tosorti}(\textit{moveto} - 1) < \textit{tosorti}(\textit{sortedi1} + 1) \\
with \\
m:  \textit{m} = \textit{sortedi1} \\
n:  \textit{n} = \textit{moveto} \\
\textit{perm1}:  \textit{perm1} = \textit{ROTATEUP}(\textit{moveto} \mapsto \textit{sortedi1} + 1) \\
then
act1: \( tosorti := tosorti \)

\[ \iffalse \]
\( \text{ROTATEUP}(\text{moveto} \leftrightarrow \text{sortedi1} + 1); \text{tosorti} \) \[ \fi \]

act2: \( \text{sortedi1} := \text{sortedi1} + 1 \)

act3: \( \text{moveto} := 0 \)

end
Event \( scanforward \) \( \triangleq \)

Status convergent

when

grd1: \( \text{sortedi}1 \neq \text{length} \)
grd2: \( \text{tosorti} (\text{sortedi}1) < \text{tosorti} (\text{sortedi}1 + 1) \)
grd3: \( \text{moveto} = 0 \)

then

act1: \( \text{sortedi}1 := \text{sortedi}1 + 1 \)

end
Event  $\textit{reverse} \triangleq$

when

$\textit{grd1}$:  $\textit{sortedi1} \neq \textit{length}$

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

then

$\textit{act1}$:  $\textit{moveto} := \textit{sortedi1}$

end
Event  \textit{scanbackward} \eq \top

Status convergent

when

\textit{grd1: } \textit{moveto} > 1

\textit{grd2: } \textit{tosorti} \left( \textit{sortedi}1 + 1 \right) < \textit{tosorti} \left( \textit{moveto} – 1 \right)

then

\textit{act1: } \textit{moveto} := \textit{moveto} – 1

end
VARIANT

length + moveto - sortedi1

END
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

\( \text{sortedi1} \)
\( \text{tosorti1} \) is ordered from 1 to \( \text{sortedi1} \)

\( \text{tosorti1} \)
Sequence to be sorted in InsertionSortR3

\( \text{tomove} \)
Element to be moved
INVARINTS

\( inv1: \) \( tosorti1 \in ISEQ \)

\( inv2: \)

\( tomove = 0 \Rightarrow \text{isSORTED}(tosorti1)(1)(sortedi1) = TRUE \)

\( inv3: \) \( tomove \neq 0 \Rightarrow \)

\( \text{isSORTED}(tosorti1)(1)(\text{moveto} - 1) = TRUE \)
\( \land \)

\( \text{isSORTED}(tosorti1)(\text{moveto})(\text{sortedi1} + 1) = TRUE \)

\( inv4: \) \( \text{sortedi1} = \text{length} \Rightarrow tosorti1 = tosorti \)
EVENTS

Initialisation

begin

act1:  sortedi1 := 1
act2:  tosorti1 := UNSORTED
act3:  tomove := 0

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

refines \( \text{scanforward} \)

when

\( \text{grd1: } \) \( \text{sortedi1} \neq \text{length} \)

\( \text{grd2: } \) \( \text{tosorti1} (\text{sortedi1}) < \text{tosorti1} (\text{sortedi1} + 1) \)

\( \text{grd3: } \) \( \text{tomove} = 0 \)

then

\( \text{act1: } \) \( \text{sortedi1} \leftarrow \text{sortedi1} + 1 \)

end
Event \( \text{reverse} \) ≃
refines \( \text{reverse} \)
when
\( \text{grd1:} \) \( \text{sortedi1} \neq \text{length} \)
\( \text{grd2:} \) \( \text{tosorti1} (\text{sortedi1}) > \text{tosorti1} (\text{sortedi1} + 1) \)
then
\( \text{act1:} \) \( \text{tomove} := \text{sortedi1} + 1 \)
end
Event \( \text{backwardinsert} \models \) refines \( \text{scanbackward} \)

when

\( \text{grd1: } \) \( \text{tomove} > 1 \)

\( \text{grd2: } \) \( \text{tosorti1}(\text{tomove}) < \text{tosorti1}(\text{tomove} - 1) \)

then

\( \text{act1: } \) \( \text{tosorti1} := \text{tosorti1} \)

\( \downarrow \)

\( (\text{SWAP}(\text{tomove} \leftrightarrow \text{tomove} - 1); \text{tosorti1}) \)

\( \text{act2: } \) \( \text{tomove} := \text{tomove} - 1 \)

end

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