System Modelling and Design

Square Root

Revision: March 14, 2012

Ken Robinson

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

March 14, 2012

© Ken Robinson 2005-2010

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

What is this example about?

The Problem

Contexts

Substitution

The SquareRoot machine

SquareRoot event

First refinement

Improving low and high

Second refinement

The third refinement

The fourth refinement

Alternative refinement sequence

Extracting code
This example presents the modelling (design) of a simple numerical algorithm, namely integer square root.

The focus of the development is on understanding how each step fits, so that the correctness of the development as a whole is clear from the steps.

Discharge of proof obligations gives confidence in the correctness in the final algorithm.

It is important to appreciate that our focus is not principally on numeric algorithms; square root gives us a simple starting point.

The example is interesting because it is not easy to see how to verify the initial model. The verification is finally produced by refining to a concrete model (that is an algorithm) that produces the square root.
What is this example about?

This example presents the modelling (design) of a simple numerical algorithm, namely integer square root.

The focus of the development is on understanding how each step fits, so that the correctness of the development as a whole is clear from the steps.

Discharge of proof obligations gives confidence in the correctness in the final algorithm.

It is important to appreciate that our focus is not principally on numeric algorithms; square root gives us a simple starting point.

The example is interesting because it is not easy to see how to verify the initial model. The verification is finally produced by refining to a concrete model (that is an algorithm) that produces the square root.
This example presents the modelling (design) of a simple numerical algorithm, namely integer square root.

The focus of the development is on understanding how each step fits, so that the correctness of the development as a whole is clear from the steps.

Discharge of proof obligations gives confidence in the correctness in the final algorithm.

It is important to appreciate that our focus is not principally on numeric algorithms; square root gives us a simple starting point.

The example is interesting because it is not easy to see how to verify the initial model. The verification is finally produced by refining to a concrete model (that is an algorithm) that produces the square root.
This example presents the modelling (design) of a simple numerical algorithm, namely integer square root.

The focus of the development is on understanding how each step fits, so that the correctness of the development as a whole is clear from the steps.

Discharge of proof obligations gives confidence in the correctness in the final algorithm.

It is important to appreciate that our focus is not principally on numeric algorithms; square root gives us a simple starting point.

The example is interesting because it is not easy to see how to verify the initial model. The verification is finally produced by refining to a concrete model (that is an algorithm) that produces the square root.
This example presents the modelling (design) of a simple numerical algorithm, namely integer square root.

The focus of the development is on understanding how each step fits, so that the correctness of the development as a whole is clear from the steps.

Discharge of proof obligations gives confidence in the correctness in the final algorithm.

It is important to appreciate that our focus is not principally on numeric algorithms; square root gives us a simple starting point.

The example is interesting because it is not easy to see how to verify the initial model. The verification is finally produced by refining to a concrete model (that is an algorithm) that produces the square root.
This example presents the modelling (design) of a simple numerical algorithm, namely integer square root.

The focus of the development is on understanding how each step fits, so that the correctness of the development as a whole is clear from the steps.

Discharge of proof obligations gives confidence in the correctness in the final algorithm.

It is important to appreciate that our focus is not principally on numeric algorithms; square root gives us a simple starting point.

The example is interesting because it is not easy to see how to verify the initial model. The verification is finally produced by refining to a concrete model (that is an algorithm) that produces the square root.
The problem is to compute the integer square root of some natural number value \( num \).

The definition of the integer square root of \( num \) is the largest natural number that when squared does not exceed \( num \), that is

\[
\sqrt{num} \times \sqrt{num} \leq num \quad (1)
\]

\[
(\sqrt{num} + 1) \times (\sqrt{num} + 1) > num \quad (2)
\]
The problem is to compute the integer square root of some natural number value $num$.

The definition of the integer square root of $num$ is the largest natural number that when squared does not exceed $num$,

that is

$$\sqrt{num} \times \sqrt{num} \leq num \quad (1)$$

$$(\sqrt{num} + 1) \times (\sqrt{num} + 1) > num \quad (2)$$
The problem is to compute the integer square root of some natural number value \( num \).

The definition of the integer square root of \( num \) is the largest natural number that when squared does not exceed \( num \), that is

\[
\sqrt{num} \times \sqrt{num} \leq num \quad (1)
\]

\[
(\sqrt{num} + 1) \times (\sqrt{num} + 1) > num \quad (2)
\]
We use a context to present a number of theorems concerning natural numbers. These will be useful for assisting the discharge of proof obligations.

**CONTEXT**

**Theories**

**AXIOMS**

\[ \text{thm1: } (\forall n \cdot n \in \mathbb{N} \Rightarrow (\exists m \cdot m \in \mathbb{N} \land (n = 2 \times m \lor n = 2 \times m + 1))) \]

\[ \text{thm2: } (\forall n \cdot n \in \mathbb{N} \Rightarrow n < (n + 1) \times (n + 1)) \]

\[ \text{thm3: } \forall m, n \cdot m \in \mathbb{N} \land n \in \mathbb{N} \land m < n \Rightarrow (m + n)/2 < n \]

\[ \text{thm4: } \forall m, n \cdot m \in \mathbb{N} \land n \in \mathbb{N} \land m < n \Rightarrow (m + n)/2 \geq m \]

**END**
We use another context to define a constant \( num \) that is any natural number. \( num \) is essentially used as an argument of the event \( sqrt \) in the SquareRoot machine.

This context \( \textit{extends} \) the \( \textit{Theories} \) context.

\[
\begin{align*}
\text{CONTEXT} & \quad \text{SquareRoot\_ctx} \\
\text{EXTENDS} & \quad \text{Theories} \\
\text{CONSTANTS} & \\
& \quad \text{num} \\
\text{AXIOMS} & \\
& \quad axm1: \quad num \in \mathbb{N} \\
\text{END}
\end{align*}
\]
Substitution

Event-B has three forms of substitution for changing the value assigned to a variable:

1. becomes equal: $x := e$, $x$ becomes equal to the value of the expression $e$;

2. becomes such that: $x :| p$, $x$ becomes a value that satisfies the predicate $p$. Within the predicate $p$ $x$ represents the value of the variable $x$ before the substitution and $x'$ represents the value of the variable $x$ after the substitution.

3. becomes in: $x \in s$, $x$ becomes any value in the set $s$

$x := x + 1$ and $x :| x' = x + 1$ are equivalent.
Event-B has three forms of substitution for changing the value assigned to a variable:

1. *becomes equal*: \( x := e \), \( x \) becomes equal to the value of the expression \( e \);

2. *becomes such that*: \( x :| p \), \( x \) becomes a value that satisfies the predicate \( p \). Within the predicate \( p \) \( x \) represents the value of the variable \( x \) before the substitution and \( x' \) represents the value of the variable \( x \) after the substitution.

3. *becomes in*: \( x :\in s \), \( x \) becomes any value in the set \( s \)

\( x := x + 1 \) and \( x :| x' = x + 1 \) are equivalent.
Substitution

Event-B has three forms of substitution for changing the value assigned to a variable:

1. *becomes equal*: $x := e$, $x$ becomes equal to the value of the expression $e$;

2. *becomes such that*: $x : | p$, $x$ becomes a value that satisfies the predicate $p$. Within the predicate $p$ $x$ represents the value of the variable $x$ before the substitution and $x'$ represents the value of the variable $x$ after the substitution.

3. *becomes in*: $x : \in s$, $x$ becomes any value in the set $s$

$x := x + 1$ and $x : | x' = x + 1$ are equivalent.
Substitution

Event-B has three forms of substitution for changing the value assigned to a variable:

1. **becomes equal**: \( x := e \), \( x \) becomes equal to the value of the expression \( e \);

2. **becomes such that**: \( x :| p \), \( x \) becomes a value that satisfies the predicate \( p \). Within the predicate \( p \) \( x \) represents the value of the variable \( x \) before the substitution and \( x' \) represents the value of the variable \( x \) after the substitution.

3. **becomes in**: \( x \in s \), \( x \) becomes any value in the set \( s \)

\( x := x + 1 \) and \( x :| x' = x + 1 \) are equivalent.
Substitution

Event-B has three forms of substitution for changing the value assigned to a variable:

1. *becomes equal*: \( x := e \), \( x \) becomes equal to the value of the expression \( e \);

2. *becomes such that*: \( x :| p \), \( x \) becomes a value that satisfies the predicate \( p \). Within the predicate \( p \) \( x \) represents the value of the variable \( x \) before the substitution and \( x' \) represents the value of the variable \( x \) after the substitution.

3. *becomes in*: \( x :\in s \), \( x \) becomes any value in the set \( s \)

\( x := x + 1 \) and \( x :| x' = x + 1 \) are equivalent.
The SquareRoot Machine

We now develop a SquareRoot machine.

The machine

- sees the context;
- has a variable $sqrt$ that will be used to store the square root of the constant (parameter) $num$. Initially $sqrt$ is assigned any natural number ($sqrt \in \mathbb{N}$).
MACHINE SquareRoot

SEES SquareRoot_ctx

VARIABLES

\( sqrt \)

INVARIANTS

\( inv1 : \sqrt \in \mathbb{N} \)

EVENTS

Initialisation

begin

\( act1 : \sqrt : \in \mathbb{N} \)

end
Event  $SquareRoot$  ≜
begin
  act1:  $sqrt : | (sqrt' \in \mathbb{N})$
       $\land sqrt' \ast sqrt' \leq num$
       $\land num < (sqrt' + 1) \ast (sqrt' + 1)$
end
END
A slightly different formulation

The following context and machine offer a slightly different formulation.
**CONTEXT**  SquareRoot_ctx  Defines SQRT(n): the integer square root of a natural number n

**EXTENDS**  Theories

**CONSTANTS**

| SQRT |

**AXIOMS**

| axm1:  SQRT ∈ N → N |

| axm2:  \( \forall m, n \cdot m \in N \land n \in N \)  \( \Rightarrow (m = SQRT(n) \iff m \ast m \leq n \land (m + 1) \ast (m + 1) > n) \) |

| thm1:  \( \forall n \cdot n \in N \)  \( \Rightarrow SQRT(n) \ast SQRT(n) \leq n \land (SQRT(n) + 1) \ast (SQRT(n) + 1) > n \) |

**END**
MACHINE SquareRoot

REQ: final = TRUE ⇒ sqrt = SQRT(num)

SEES SquareRoot_ctx, SquareRootArg

VARIABLES

sqrt
final

ININVARIANTS

inv1: sqrt ∈ N
inv2: final = TRUE ⇒ sqrt = SQRT(num)

EVENTS

Initialisation

begin

act1: sqrt ∈ N
act2: final := FALSE

end
Event \textit{SquareRoot} \triangleright

\textbf{when}

\textit{grd1}: \quad \textit{final} = \text{FALSE}

\textbf{then}

\textit{act1}: \quad \textit{sqrt} := \text{SQRT}(\text{num})

\textit{act2}: \quad \textit{final} := \text{TRUE}

\textbf{end}

\textbf{END}
SquareRoot event

Note the use of :| to make a *declarative* assignment to *sqrt* using the predicates determined earlier for the definition of *square root*.

This form of assignment requires that *sqrt* is assigned a value consistent with the definition of square root given in (1) and (2) above.
Note the use of :| to make a *declarative* assignment to *sqrt* using the predicates determined earlier for the definition of *square root*. This form of assignment requires that *sqrt* is assigned a value consistent with the definition of square root given in (1) and (2) above.
First refinement

The current assignment to $sqrt$, while obviously correct — that is consistent with the definition given in predicates 1 and 2 — is abstract. We will refine to a concrete model, that is an algorithm.

The problem with the predicate

$$sqrt' \ast sqrt' \leq num \land num < (sqrt' + 1) \ast (sqrt' + 1)$$

is that it contains a conjunction of two properties for $sqrt'$ each easy to satisfy on their own, but together much more difficult.

We split this conjunction into two predicates, each with a different variable as follows:

$$low \ast low \leq num$$

and

$$num < high \ast high$$

and contrive to bring $low$ and $high$ closer together until

$$high = low + 1$$

at which point $low$ will be the required square root.
First refinement

The current assignment to \(\sqrt{\cdot}\), while obviously correct — that is consistent with the definition given in predicates 1 and 2 — is abstract. We will refine to a concrete model, that is an algorithm.

The problem with the predicate

\[
\sqrt{\prime} \times \sqrt{\prime} \leq num \land num < (\sqrt{\prime} + 1) \times (\sqrt{\prime} + 1)
\]

is that it contains a conjunction of two properties for \(\sqrt{\prime}\) each easy to satisfy on their own, but together much more difficult.

We split this conjunction into two predicates, each with a different variable as follows:

\[
low \times low \leq num
\]

and

\[
num < high \times high
\]

and contrive to bring \(low\) and \(high\) closer together until

\[
high = low + 1
\]

at which point \(low\) will be the required square root.
First refinement

The current assignment to \( \text{sqrt} \), while obviously correct — that is consistent with the definition given in predicates 1 and 2 — is abstract. We will refine to a concrete model, that is an algorithm.

The problem with the predicate

\[
\sqrt{\prime} \times \sqrt{\prime} \leq \text{num} \land \text{num} < (\sqrt{\prime} + 1) \times (\sqrt{\prime} + 1)
\]

is that it contains a conjunction of two properties for \( \sqrt{\prime} \) each easy to satisfy on their own, but together much more difficult.

We split this conjunction into two predicates, each with a different variable as follows:

\[
\text{low} \times \text{low} \leq \text{num}
\]

and

\[
\text{num} < \text{high} \times \text{high}
\]

and contrive to bring \( \text{low} \) and \( \text{high} \) closer together until

\[
\text{high} = \text{low} + 1
\]

at which point \( \text{low} \) will be the required square root.
MACHINE SquareRootR1

REFINES SquareRoot

SEES SquareRoot_ctx

VARIABLES

\[ \text{sqrt} \]
\[ \text{low} \]
\[ \text{high} \]

INvariants

\[
\text{inv1} : \text{low} \in \mathbb{N} \\
\text{inv2} : \text{high} \in \mathbb{N} \\
\text{inv3} : \text{low} + 1 \leq \text{high} \\
\text{inv4} : \text{low} \times \text{low} \leq \text{num} \\
\text{inv5} : \text{num} < \text{high} \times \text{high}
\]
EVENTS

Initialisation

begin

act1 : $sqrt \in \mathbb{N}$

act2 : $low : | (low' \in \mathbb{N} \land low' \times low' \leq num)$

act3 : $high : | (high' \in \mathbb{N} \land num < high' \times high')$

end
Event \( \textit{SquareRoot} \Rightarrow \)
refines \( \textit{SquareRoot} \)
when
\( \text{grd1: } low + 1 = high \)
then
\( \text{act1: } sqrt := low \)
end
Event \textit{Improve} 

\textbf{Status} convergent

\textbf{any} $l$

\textbf{any} $h$

\textbf{when}

\textbf{grd1}: $low + 1 \neq high$

\textbf{grd2}: $l \in \mathbb{N} \land low \leq l \land l \times l \leq num$

\textbf{grd3}: $h \in \mathbb{N} \land h \leq high \land num < h \times h$

\textbf{grd4}: $l + 1 \leq h$

\textbf{grd5}: $h - l < high - low$

\textbf{then}

\textbf{act1}: $low, high := l, h$

\textbf{end}
VARIANT

high – low

END
Improving low and high

As shown in the above, the refinement of $SquareRoot$ fires only when $low + 1 = high$. We need a new event to bring that about.

We introduce a new event $Improve$ that brings $low$ and $high$ closer together, while maintaining the invariant relation on those variables.

It is important to observe the parameters $l$ and $h$ together with appropriate guards (constraints) that describe necessary properties, without perhaps knowledge of how those parameters can be instantiated. This style of specification is sometimes called declarative.
Improving low and high

As shown in the above, the refinement of \( SquareRoot \) fires only when \( low + 1 = high \). We need a new event to bring that about.

We introduce a new event \( Improve \) that brings \( low \) and \( high \) closer together, while maintaining the invariant relation on those variables.

It is important to observe the parameters \( l \) and \( h \) together with appropriate guards (constraints) that describe necessary properties, without perhaps knowledge of how those parameters can be instantiated. This style of specification is sometimes called \( declarative \).
Improving low and high

As shown in the above, the refinement of $SquareRoot$ fires only when $low + 1 = high$. We need a new event to bring that about.

We introduce a new event $Improve$ that brings $low$ and $high$ closer together, while maintaining the invariant relation on those variables.

It is important to observe the parameters $l$ and $h$ together with appropriate guards (constraints) that describe necessary properties, without perhaps knowledge of how those parameters can be instantiated. This style of specification is sometimes called $declarative$. 
The specification of *Improve* is abstract, simply requiring that either *low* is increased, or *high* is decreased, or both, without showing how that may be achieved concretely.

In order to ensure that *SquareRoot* will eventually fire, the new event must be convergent, that is it must terminate in a finite number of firings. We give a variant expression, which must yield a natural number value that is guaranteed to decrease on each execution of *Improve*.

Since either *low* is increased or *high* is decreased and \(low + 1 \leq high\) an adequate expression is *high − low*. 
The specification of *Improve* is abstract, simply requiring that either *low* is increased, or *high* is decreased, or both, without showing how that may be achieved concretely.

In order to ensure that *SquareRoot* will eventually fire, the new event must be convergent, that is it must terminate in a finite number of firings. We give a variant expression, which must yield a natural number value that is guaranteed to decrease on each execution of *Improve*.

Since either *low* is increased or *high* is decreased and \( low + 1 \leq high \) an adequate expression is \( high - low \).
The specification of *Improve* is abstract, simply requiring that either *low* is increased, or *high* is decreased, or both, without showing how that may be achieved concretely.

In order to ensure that *SquareRoot* will eventually fire, the new event must be convergent, that is it must terminate in a finite number of firings. We give a variant expression, which must yield a natural number value that is guaranteed to decrease on each execution of *Improve*.

Since either *low* is increased or *high* is decreased and $low + 1 \leq high$ an adequate expression is $high - low$. 
Second refinement

We now do a second refinement in which we refine the *Improve* event, mapping out how we can produce a better value of *low* or *high*. The idea is to take a value *mid* that is strictly between *low* and *high* and test whether *mid* is a better replacement for *low* or *high* and replace those values accordingly.

We know that such a value exists because *low* + 1 ≤ *high* and also *low* + 1 ≠ *high*, so *low* + 1 < *high* implying that there is at least one value between *low* and *high*.

In this refinement *Improve* is refined two ways into *Improve1* and *Improve2* depending on whether *low* or *high* is improved, respectively.

**Witnesses:** The refinements of *Improve* have eliminated the parameters *l* and *h*, so we are required to show how those parameters are represented by the refinement. This is achieved by the *with* clause, which shows the values of those parameters.
MACHINE SquareRootR2
REFINES SquareRootR1
SEES SquareRoot_ctx
VARIABLES

sqrt
low
high
EVENTS

Initialisation

begin

act 1: \textit{sqrt} \in \mathbb{N}

act 2: \textit{low} : | (\textit{low'} \in \mathbb{N} \land \textit{low'} \times \textit{low'} \leq \textit{num})

act 3: \textit{high} : | (\textit{high'} \in \mathbb{N} \land \textit{num} < \textit{high'} \times \textit{high'})

end
Event $SquareRoot \sqsupseteq$
refines $SquareRoot$
when
$grd1 : low + 1 = high$
then
$act1 : sqrt := low$
end
Event \textit{Improve1} ≜
refines \textit{Improve}
\hspace{1cm} \text{any } m \text{ when}
\hspace{1cm} \text{grd1: } low + 1 ≠ high
\hspace{1cm} \text{grd2: } m ∈ \mathbb{N}
\hspace{1cm} \text{grd3: } low < m ∧ m < high
\hspace{1cm} \text{grd4: } m * m ≤ num
\hspace{1cm} \text{with}
\hspace{2cm} l: l = m
\hspace{2cm} h: h = high
\hspace{1cm} \text{then}
\hspace{1cm} \text{act1: } low := m
\hspace{1cm} \text{end}
**Event** Improve2 ⊆
refines Improve
  any m
  when
  grd1: low + 1 ≠ high
  grd2: m ∈ N
  grd3: low < m ∧ m < high
  grd4: num < m * m
  with
    l: l = low
    h: h = m
  then
  act1: high := m
  end
END
The third refinement refines the abstract specifications to concrete expressions.

1. We compute the initial values of low and high as 0 and num + 1 respectively. We could find better initial values, but since the performance is going to be logarithmic, it may not be worth the expense of more complicated computations.

2. In SquareRootR3 we refine Improve two ways, each with a parameter m set to the value (low + high)/2, that is we introduce the midpoint value between low and high. This is not the only option, we could have used low + 1 or high − 1; any value that is strictly between —and not equal to— low and high.

The difference is one of performance, not correctness. Taking the midpoint will give logarithmic performance, while incrementing or decrementing will give linear performance.
The third refinement

The third refinement refines the abstract specifications to concrete expressions.

1. We compute the initial values of low and high as 0 and num + 1 respectively. We could find better initial values, but since the performance is going to be logarithmic, it may not be worth the expense of more complicated computations.

2. In SquareRootR3 we refine Improve two ways, each with a parameter m set to the value \((low + high)/2\), that is we introduce the midpoint value between low and high. This is not the only option, we could have used low + 1 or high − 1; any value that is strictly between —and not equal to— low and high.

The difference is one of performance, not correctness. Taking the mid point will give logarithmic performance, while incrementing or decrementing will give linear performance.
The third refinement

The third refinement refines the abstract specifications to concrete expressions.

1. We compute the initial values of \( low \) and \( high \) as 0 and \( num + 1 \) respectively. We could find better initial values, but since the performance is going to be logarithmic, it may not be worth the expense of more complicated computations.

2. In \( SquareRootR3 \) we refine \( Improve \) two ways, each with a parameter \( m \) set to the value \( (low + high)/2 \), that is we introduce the midpoint value between \( low \) and \( high \).
   
   This is not the only option, we could have used \( low + 1 \) or \( high - 1 \); any value that is strictly between —and not equal to— \( low \) and \( high \).
   
   The difference is one of performance, not correctness. Taking the mid point will give logarithmic performance, while incrementing or decrementing will give linear performance.
The third refinement

The third refinement refines the abstract specifications to concrete expressions.

1. We compute the initial values of \( low \) and \( high \) as 0 and \( num + 1 \) respectively. We could find better initial values, but since the performance is going to be logarithmic, it may not be worth the expense of more complicated computations.

2. In \( \text{SquareRootR3} \) we refine \( \text{Improve} \) two ways, each with a parameter \( m \) set to the value \( (low + high)/2 \), that is we introduce the midpoint value between \( low \) and \( high \). This is not the only option, we could have used \( low + 1 \) or \( high - 1 \); any value that is strictly between —and not equal to— \( low \) and \( high \).

The difference is one of performance, not correctness. Taking the midpoint will give logarithmic performance, while incrementing or decrementing will give linear performance.
The third refinement

The third refinement refines the abstract specifications to concrete expressions.

1. We compute the initial values of \( low \) and \( high \) as 0 and \( num + 1 \) respectively. We could find better initial values, but since the performance is going to be logarithmic, it may not be worth the expense of more complicated computations.

2. In \( SquareRootR3 \) we refine \( Improve \) two ways, each with a parameter \( m \) set to the value \( (low + high)/2 \), that is we introduce the midpoint value between \( low \) and \( high \). This is not the only option, we could have used \( low + 1 \) or \( high - 1 \); any value that is strictly between —and not equal to— \( low \) and \( high \).

The difference is one of performance, not correctness. Taking the mid point will give logarithmic performance, while incrementing or decrementing will give linear performance.
MACHINE SquareRootR3
REFINES SquareRootR2
SEES SquareRoot_ctx
VARIABLES
  sqrt
  low
  high
EVENTS

Initialisation

begin

act1 : \( \sqrt{n} \in \mathbb{N} \)
act2 : \( low := 0 \)
act3 : \( high := num + 1 \)

end
Event $SquareRoot$ \( \trianglelefteq \)
refines $SquareRoot$
when
$grd1: \ low + 1 = high$
then
$act1: \ sqrt := low$
end
Event \( \text{Improve1} \equiv \)
refines \( \text{Improve1} \)

any \( m \)

when

\( \text{grd1} : \text{low} + 1 \neq \text{high} \)

\( \text{grd2} : m = (\text{low} + \text{high})/2 \)

\( \text{grd3} : m \times m \leq \text{num} \)

then

\( \text{act1} : \text{low} := m \)

end
Event \( \text{Improve2} \supset \)

refines \( \text{Improve2} \)

any \( m \)

when

\[ \text{grd1: } \text{low} + 1 \neq \text{high} \]

\[ \text{grd2: } m = (\text{low} + \text{high})/2 \]

\[ \text{grd3: } \text{num} < m \times m \]

then

\[ \text{act1: } \text{high} := m \]

end

END
The fourth refinement

In the third refinement we still have an abstract element in the form of the parameter $m$ to the events $Improve1$ and $Improve2$.

The value of $m$ has to be produced, nondeterministically, to satisfy the guards of $Improve1$.

In $SquareRootR4$ we introduce another variable, $mid$, to replace the parameter $m$. 
The fourth refinement

In the third refinement we still have an abstract element in the form of the parameter $m$ to the events $Improve1$ and $Improve2$.

The value of $m$ has to be produced, nondeterministically, to satisfy the guards of $Improve1$.

In $SquareRootR4$ we introduce another variable, $mid$, to replace the parameter $m$. 
The fourth refinement

In the third refinement we still have an abstract element in the form of the parameter $m$ to the events $\text{Improve}1$ and $\text{Improve}2$.

The value of $m$ has to be produced, nondeterministically, to satisfy the guards of $\text{Improve}1$.

In $\text{SquareRootR4}$ we introduce another variable, $\text{mid}$, to replace the parameter $m$. 
MACHINE SquareRootR4
REFINES SquareRootR3
SEES SquareRoot_ctx
VARIABLES
  sqrt
  low
  high
  mid
INVARIANTS
  inv1: mid = (low + high)/2
EVENTS

Initialisation

begin

act1 : \( \text{sqrt} \in \mathbb{N} \)

act2 : \( \text{low} := 0 \)

act3 : \( \text{high} := \text{num} + 1 \)

act4 : \( \text{mid} := (\text{num} + 1)/2 \)

end
Event $SquareRoot$ \( \supseteq \)
refines $SquareRoot$
when
$grd1 : \text{low} + 1 = \text{high}$
then
$act1 : \text{sqrt} := \text{low}$
end
Event \( \text{Improve1} \subseteq \)
refines \( \text{Improve1} \)
when
\[
\text{grd1} : \quad \text{low} + 1 \neq \text{high}
\]
\[
\text{grd2} : \quad \text{mid} \times \text{mid} \leq \text{num}
\]
with
\[
\text{m} : \quad m = \text{mid}
\]
then
\[
\text{act1} : \quad \text{low} := \text{mid}
\]
\[
\text{act2} : \quad \text{mid} := (\text{mid} + \text{high})/2
\]
end
Event $Improve2 \triangleq$
refines $Improve2$
when
$grd1 : low + 1 \neq high$
$grd2 : num < mid \ast mid$
with
$m : m = mid$
then
$act1 : high := mid$
$act2 : mid := (low + mid)/2$
end
END
Alternative refinement sequence

Instead of the refinement sequence $\text{SquarerootR3A, SquarerootR4}$ we could have effectively gone directly to $\text{SquareRootR4}$ from $\text{SquareRootR2}$ and this is shown in $\text{SquareRootR3B}$

**Historical note:** the presentation sequence of a model development is frequently different to the chronological sequence. Refinement to $\text{SquarerootR3B}$ was the original refinement step from $\text{SquarerootR2}$. The extra step via $\text{SquarerootR3A}$ to $\text{SquarerootR4}$ was added later.
MACHINE SquareRootR3B

REFINES SquareRootR2

SEES SquareRoot_ctx

VARIABLES

\[
\begin{align*}
\text{sqrt} \\
\text{low} \\
\text{high} \\
\text{mid}
\end{align*}
\]

INVARIANTS

\[
\begin{align*}
\text{inv1}: \text{mid} \in \mathbb{N} \\
\text{inv2}: \text{mid} = (\text{low} + \text{high})/2 \\
\text{inv3}: \text{low} \leq \text{mid} \\
\text{inv4}: \text{mid} < \text{high}
\end{align*}
\]
EVENTS

Initialisation

begin

act1: \( \sqrt{x} \in \mathbb{N} \)

act2: \( low := 0 \)

act3: \( high := num + 1 \)

act4: \( mid := (num + 1)/2 \)

end
Event \texttt{SquareRoot} \supseteq
refines \texttt{SquareRoot}
when
\texttt{grd1}: \texttt{low + 1 = high}
then
\texttt{act1}: \texttt{sqrt := low}
end
Event \( \text{Improve1} \) \( \sqsubseteq \) refines \( \text{Improve1} \)
when
\( \text{grd1} : \ low + 1 \neq high \)
\( \text{grd2} : \ mid \ast mid \leq num \)
with
\( m : \ m = mid \)
then
\( \text{act1} : \ low := mid \)
\( \text{act2} : \ mid := (mid + high)/2 \)
end
Event $\text{Improve2} \supseteq$
refines $\text{Improve2} $
when
$\text{grd1} : \text{low} + 1 \neq \text{high}$
$\text{grd2} : \text{num} < \text{mid} \times \text{mid}$
with
$\text{m} : \text{m} = \text{mid}$
then
$\text{act1} : \text{high} := \text{mid}$
$\text{act2} : \text{mid} := (\text{low} + \text{mid})/2$
end
END
If the final refinement is examined carefully, it is clear that the events can be replaced by a loop:

- the subsidiary events $\text{Improve}1$ and $\text{Improve}2$ execute until $low + 1 = high$
- at which point the loop terminates and the final square root is computed.

Thus we can manually refine the final refinement to the following pseudo code.
Extracting code

If the final refinement is examined carefully, it is clear that the events can be replaced by a loop:

- the subsidiary events \textit{Improve1} and \textit{Improve2} execute until \( low + 1 = high \)
- at which point the loop terminates and the final square root is computed.

Thus we can manually refine the final refinement to the following pseudo code.
Extracting code

If the final refinement is examined carefully, it is clear that the events can be replaced by a loop:

- the subsidiary events Improve1 and Improve2 execute until $low + 1 = high$
- at which point the loop terminates and the final square root is computed.

Thus we can manually refine the final refinement to the following pseudo code.
If the final refinement is examined carefully, it is clear that the events can be replaced by a loop:

- the subsidiary events *Improve*1 and *Improve*2 execute until \( \text{low} + 1 = \text{high} \)
- at which point the loop terminates and the final square root is computed.

Thus we can manually refine the final refinement to the following pseudo code.
low := 0;
high := num + 1;

while low + 1 \neq high {
    mid := (low + high)/2
    if mid \times mid \leq num{
        low := mid
    }
    else high := mid
}

sqrt := low