1 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.

2 The Problem

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

\begin{align}
\sqrt{num} \times \sqrt{num} & \leq num \tag{1} \\
(\sqrt{num} + 1) \times (\sqrt{num} + 1) & > num \tag{2}
\end{align}

2.1 Contexts

Theories: context

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

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)))

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

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

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

END

SquareRoot_ctx: context

We use another context to define a constant $num$ that is any natural number. $num$ is essentially used as an argument of the event $\sqrt{\cdot}$ in the SquareRoot machine.

This context extends the Theories context.

CONTEXT SquareRoot_ctx

EXTENDS Theories

CONSTANTS

num
AXIOMS

axm1: \( \text{num} \in \mathbb{N} \)

END

2.2 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.

2.3 The SquareRoot machine

We now develop a SquareRoot machine.

The machine

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

MACHINE SquareRoot

SEES SquareRoot_ctx

VARIABLES \( \text{sqrt} \)

INVARIANTS

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

EVENTS

Initialisation

begin

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

end

Event SquareRoot \( \triangleq \)

begin
A slightly different formulation
The following context and machine offer a slightly different formulation.

**CONTEXT** SquareRoot_ctx

**EXTENDS** Theories

**CONSTANTS**

SQRT

**AXIOMS**

axm1: \( SQRT \in \mathbb{N} \rightarrow \mathbb{N} \)

axm2: \( \forall m, n \cdot m \in \mathbb{N} \land n \in \mathbb{N} \implies (m = SQRT(n) \Leftrightarrow m \cdot m \leq n \land (m + 1) \cdot (m + 1) > n) \)

thm1: \( \forall n \cdot n \in \mathbb{N} \implies \\
\quad SQRT(n) \cdot SQRT(n) \leq n \\
\quad \land (SQRT(n) + 1) \cdot (SQRT(n) + 1) > n \)

**MACHINE** SquareRoot

**SEES** SquareRoot_ctx, SquareRootArg

**VARIABLES**

sqrt

final

**INvariants**

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

inv2: \( final = TRUE \implies sqrt = SQRT(num) \)

**EVENTS**

**Initialisation**

begin

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

\( \land sqrt' \in \mathbb{N} \)

\( \land sqrt' \cdot sqrt' \leq num \)

\( \land num < (sqrt' + 1) \cdot (sqrt' + 1) \)
act2: final := FALSE
end

Event SquareRoot :=
when
grd1: final = FALSE
then
act1: sqrt := SQRT(num)
act2: final := TRUE
end
END

2.4 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.

3 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

\[ sqrt' \times sqrt' \leq num \land num < (sqrt' + 1) \times (sqrt' + 1) \]

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:

\[ low \times low \leq num \]

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.

MACHINE SquareRootR1

REFINES SquareRoot
SEES  SquareRoot_ctx

VARIABLES  sqrt  low  high

INVARINTS
inv1: low ∈ N
inv2: high ∈ N
inv3: low + 1 ≤ high
inv4: low * low ≤ num
inv5: num < high * high

EVENTS
Initialisation
begin
act1: sqrt ∈ N
act2: low :| (low' ∈ N ∧ low' * low' ≤ num)
act3: high :| (high' ∈ N ∧ num < high' * high')
end

Event  SquareRoot  \cong
refines  SquareRoot
when
grd1: low + 1 = high
then
act1: sqrt := low
end

Event  Improve  \cong
Status  convergent
any  l  h
when
grd1: low + 1 ≠ high
grd2: l ∈ N ∧ low ≤ l ∧ l * l ≤ num
grd3: h ∈ N ∧ h ≤ high ∧ num < h * h
grd4: l + 1 ≤ h
grd5: h − l < high − low
then

act1: low, high := l, h
end

VARIANT high − low
END

3.1 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 \( high − low \).

4 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 \( low + 1 \leq high \) and also \( low + 1 \neq high \), 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
act1: sqrt ∈ N
act2: low :| (low' ∈ N ∧ low' * low' ≤ num)
act3: high :| (high' ∈ N ∧ num < high' * high')
end

Event $SquareRoot \hat{=}$
refines $SquareRoot$
when
grd1: low + 1 = high
then
act1: sqrt := low
end

Event $Improve1 \hat{=}$
refines $Improve$
any $m$
when
grd1: low + 1 \neq high
grd2: m ∈ N
grd3: low < m ∧ m < high
grd4: m * m ≤ num
with
l: l = m
h: h = high
then
act1: low := m
end

Event $Improve2 \hat{=}$
refines $Improve$
any $m$
when

grd1: \( low + 1 \neq high \)
grd2: \( m \in \mathbb{N} \)
grd3: \( low < m \land m < high \)
grd4: \( num < m \ast m \)

with

1: \( l = low \)

h: \( h = m \)

then

act1: \( high := m \)

end

END

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

Event SquareRoot ≅
refines SquareRoot
when
grd1: low + 1 = high
then
act1: sqrt := low
end

Event Improve1 ≅
refines Improve1
any m
when
grd1: low + 1 ≠ high
grd2: m = (low + high)/2
grd3: m * m ≤ num
then
act1: low := m
end

Event Improve2 ≅
refines Improve2
any m
when
grd1: low + 1 ≠ high
grd2: m = (low + high)/2
grd3: num < m * m
then
act1: high := m
end
END
6 The fourth refinement

In the third refinement we still have an abstract element in the form of the parameter \( m \) to the events \( \text{Improve1} \) and \( \text{Improve2} \).

The value of \( m \) has to be produced, nondeterministically, to satisfy the guards of \( \text{Improve1} \).

In \( \text{SquareRootR4} \) we introduce another variable, \( \text{mid} \), to replace the parameter \( m \).

\[
\text{MACHINE} \quad \text{SquareRootR4} \\
\text{REFINES} \quad \text{SquareRootR3} \\
\text{SEES} \quad \text{SquareRoot_ctx} \\
\text{VARIABLES} \quad \text{sqrt} \; \text{low} \; \text{high} \; \text{mid} \\
\text{INVARIANTS} \\
\text{inv1}: \text{mid} = (\text{low} + \text{high})/2 \\
\text{EVENTS} \\
\text{Initialisation} \\
\text{begin} \\
\text{act1}: \text{sqrt} :\in \mathbb{N} \\
\text{act2}: \text{low} := 0 \\
\text{act3}: \text{high} := \text{num} + 1 \\
\text{act4}: \text{mid} := (\text{num} + 1)/2 \\
\text{end} \\
\text{Event} \quad \text{SquareRoot} \triangleq \\
\text{refines} \quad \text{SquareRoot} \\
\text{when} \\
\text{grd1}: \text{low} + 1 = \text{high} \\
\text{then} \\
\text{act1}: \text{sqrt} := \text{low} \\
\text{end} \\
\text{Event} \quad \text{Improve1} \triangleq \\
\text{refines} \quad \text{Improve1} \\
\text{when} \\
\text{grd1}: \text{low} + 1 \neq \text{high} \\
\text{grd2}: \text{mid} \times \text{mid} \leq \text{num} \]
with
m : m = mid
then
act1 : low := mid
act2 : mid := (mid + high)/2
end
Event Improve2 :=
refines Improve2
when
grd1 : low + 1 ≠ high
grd2 : num < mid * mid
with
m : m = mid
then
act1 : high := mid
act2 : mid := (low + mid)/2
end
END

7 Alternative refinement sequence

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

Historical note: the presentation sequence of a model development is frequently different to the chronological sequence. Refinement to SquarerootR3B was the original refinement step from Square-rootR2. The extra step via SquarerootR3A to SquarerootR4 was added later.

MACHINE SquareRootR3B
REFINES SquareRootR2
SEES SquareRoot_ctx
VARIABLES sqrt low high mid
INVARIANTS
inv1 : mid ∈ N
\textbf{inv2}: \( \text{mid} = (\text{low} + \text{high})/2 \)

\textbf{inv3}: \( \text{low} \leq \text{mid} \)

\textbf{inv4}: \( \text{mid} < \text{high} \)

\textbf{EVENTS}

\textbf{Initialisation}

\begin{verbatim}
begin
act1: sqrt :\in \mathbb{N}
act2: low := 0
act3: high := num + 1
act4: mid := (num + 1)/2
end
\end{verbatim}

\textbf{Event} \texttt{SquareRoot} \equiv

\textbf{refines} \texttt{SquareRoot}

\textbf{when}

\begin{verbatim}
grd1: low + 1 = high
\end{verbatim}

\textbf{then}

\begin{verbatim}
act1: sqrt := low
\end{verbatim}

\textbf{end}

\textbf{Event} \texttt{Improve1} \equiv

\textbf{refines} \texttt{Improve1}

\textbf{when}

\begin{verbatim}
grd1: low + 1 \neq high
grd2: mid * mid \leq num
\end{verbatim}

\textbf{with}

\begin{verbatim}
m : m = mid
\end{verbatim}

\textbf{then}

\begin{verbatim}
act1: low := mid
act2: mid := (mid + high)/2
\end{verbatim}

\textbf{end}

\textbf{Event} \texttt{Improve2} \equiv

\textbf{refines} \texttt{Improve2}
when

grd1: low + 1 ≠ high

grd2: num < mid * mid

with

m: m = mid

then

act1: high := mid

act2: mid := (low + mid)/2

end

END

8 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.

```
low := 0;
high := num + 1;

while low + 1 ≠ high {
    mid := (low + high)/2
    if mid * mid ≤ num{
        low := mid
    }
    else high := mid
}

sqrt := low
```