

# System Modelling and Design

## Introduction

Revision: 1.0, March 3, 2007

Ken Robinson

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

April 3, 2008

©Ken Robinson 2005

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

# Outline I

- 1 What is this lecture about?
  - The Problem
  
- 2 SquareRoot.ctx: context machine
  - The SquareRoot machine
  - SquareRoot event
  - First refinement
  - Improving low and high
  - Second refinement
  - The third refinement
  - Extracting code

# What is this lecture about?

This lecture presents the design of an integer square root algorithm. 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 greater 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.

# What is this lecture about?

This lecture presents the design of an integer square root algorithm. 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 greater 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.

# What is this lecture about?

This lecture presents the design of an integer square root algorithm. 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 greater 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 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

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

$$(sqrt(num) + 1) \times (sqrt(num) + 1) > num \quad (2)$$

# SquareRoot\_ctx: context machine

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

We also give a couple of number theorems:

- thm1** every natural number (in fact integer) is equal to either
- twice some other natural number, or
  - twice some other natural number plus one.

This is an expression of the notion that every number is either *even* or *odd*.

- thm2** every natural number is less than the square of its successor.

# The SquareRoot Machine

We now develop a SquareRoot machine.

The machine

- sees the context machine;
- 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* :  $sqrt \in \mathbb{N}$

# SquareRoot event

We use  $:|$  (“becomes such that”) assignment as in  $x :| P$ , which assigns a value to  $x$  that satisfies the predicate  $P$ . The predicate  $P$  can contain both  $x$ , representing the value of  $x$  before the assignment, and  $x'$ , representing the value of  $x$  after the assignment. This form of assignment ensures that  $sqrt$  is assigned a value consistent with the definition of square root given above.

## EVENTS

### Initialisation

begin

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

end

squareroot  $\hat{=}$

begin

$act1 : sqrt :| (sqrt' \in \mathbb{N} \wedge sqrt' * sqrt' \leq num \wedge num < (sqrt' + 1) * (sqrt' + 1))$

end

# First refinement

The current assignment to  $\text{sqrt}$ , while obviously correct, is abstract and we have to compute a concrete value.

The problem with the predicate

$$\text{sqrt}' * \text{sqrt}' \leq \text{num} \wedge \text{num} < (\text{sqrt}' + 1) * (\text{sqrt}' + 1)$$

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

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

$$\text{low} * \text{low} \leq \text{num}$$

and

$$\text{num} < \text{high} * \text{high}$$

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

$$\text{low} + 1 = \text{high}$$

at which point  $\text{low}$  will be the required square root.

MACHINE SquareRootR1

REFINES SquareRoot

SEES SquareRoot\_ctx

VARIABLES

*sqrt*

*low*

*high*

INVARIANTS

*inv1* :  $low \in \mathbb{N}$

*inv2* :  $high \in \mathbb{N}$

*inv3* :  $low + 1 \leq high$

*inv4* :  $low * low \leq num$

*inv5* :  $num < high * high$

## EVENTS

### Initialisation

begin

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

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

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

end

SquareRoot  $\hat{=}$

Refines SquareRoot

when

*grd1* : *low* + 1 = *high*

then

*act1* : *sqrt* := *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.

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 and we are required to give a variant expression, which must yield a natural number and is guaranteed to decrease on each execution of *Improve*.

Since either *low* is increased or *high* is increased but  $low + 1 \leq high$  an adequate expression is  $high - low$ .

Improve  $\hat{=}$

any

$l$

$h$

where

*grd1* :  $low + 1 \neq high$

*grd2* :  $l \in \mathbb{N} \wedge low \leq l \wedge l * l \leq num$

*grd3* :  $h \in \mathbb{N} \wedge h \leq high \wedge num < h * h$

*grd4* :  $l + 1 \leq h$

*grd5* :  $h - l < high - low$

then

*act1* :  $low, high := l, h$

end

VARIANT

$high - low$

END

# 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* :  $\in \mathbb{N}$

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

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

end

SquareRoot  $\hat{=}$

Refines SquareRoot

when

*grd1* :  $low + 1 = high$

then

*act1* :  $sqrt := low$

end

Improve1  $\hat{=}$

Refines Improve

any

*mid*

where

*grd1* :  $low + 1 \neq high$

*grd2* :  $mid \in \mathbb{N}$

*grd3* :  $low < mid \wedge mid < high$

*grd4* :  $mid * mid \leq num$

with

*l* :  $l = mid$

*h* :  $h = high$

then

*act1* :  $low := mid$

end

Improve2  $\hat{=}$

Refines Improve

any

*mid*

where

*grd1* :  $low + 1 \neq high$

*grd2* :  $mid \in \mathbb{N}$

*grd3* :  $low < mid \wedge mid < high$

*grd4* :  $num < mid * mid$

with

*l* :  $l = low$

*h* :  $h = mid$

then

*act1* :  $high := mid$

end

VARIANT

*high – low*

END

# 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 We compute *mid* as  $(low + high)/2$ , that is we take the midpoint value between *low* and *high*.  
This is not the only option, we could have used  $low + 1$  or  $high - 1$ .  
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*

THEOREMS

*thm1* :  $\exists m. (m \in \mathbb{N} \wedge (num = 2 * m \vee num = 2 * m + 1))$

## EVENTS

### Initialisation

begin

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

*act2* :  $low := 0$

*act3* :  $high := num + 1$

end

SquareRoot  $\hat{=}$

Refines SquareRoot

when

*grd1* :  $low + 1 = high$

then

*act1* :  $sqrt := low$

end

Improve1  $\hat{=}$

Refines Improve1

any

*mid*

where

*grd1* :  $low + 1 \neq high$

*grd2* :  $mid \in \mathbb{N}$

*grd3* :  $mid = (low + high)/2$

*grd4* :  $mid * mid \leq num$

then

*act1* :  $low := mid$

end

Improve2  $\hat{=}$

Refines Improve2

any

*mid*

where

*grd1* :  $low + 1 \neq high$

*grd2* :  $mid \in \mathbb{N}$

*grd3* :  $mid = (low + high)/2$

*grd4* :  $num < mid * mid$

then

*act1* :  $high := mid$

end

VARIANT  
*high – low*  
END

# 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  $\neq$  high {  
  mid := (low + high)/2  
  if mid * mid  $\leq$  num{  
    low := mid  
  }  
  else high := mid  
}  
sqrt := low
```