System Modelling and Design

An Introduction to Modelling

Revision: 1.2, May 19, 2004

Ken Robinson

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

12th March 2005

©Ken Robinson 2005
mailto:k.robinson@unsw.edu.au
Outline

1. Prelude
   - Why do we want a model?
   - What should a model do?
3. Software Modelling
   - Is Software different?
   - Specification, Design and Implementation
   - A Simple Numeric Example
   - Specification
   - Animation
   - Refinement
   - Generating an interface
This course is going to be concerned with the rigorous development of systems. We will be using *B Method* (*B*), a formal development method that spans development from specification through to implementation. *B* is based on set theory and logic, but more of that later. There will be some *B* in this lecture, but it will be given without introduction and hopefully will be reasonably easy to understand.

We are concerned here with the notion of *modelling*, but in particular with the need for a model to enable quantitative measurement. This isn’t always the case: *Unified Modelling Language* (*UML*) is a widespread modelling language for computing systems, but in *UML* we get pictures and no measurement.
Why do we want a model?

We are going to use a mathematics based notation to describe our models, but the intention is the same as in any application where a model is used. We want a model:

- to exhibit some important characteristics of some “thing” that we want to build or implement;
- to reveal what our “thing” will “look” like;
- to help us understand how our “thing” will behave.
Why do we want a model?

We are going to use a mathematics based notation to describe our models, but the intention is the same as in any application where a model is used. We want a model:

- to exhibit some important characteristics of some “thing” that we want to build or implement;
- to reveal what our “thing” will “look” like;
- to help us understand how our “thing” will behave.
We are going to use a mathematics based notation to describe our models, but the intention is the same as in any application where a model is used. We want a model:

- to exhibit some important characteristics of some “thing” that we want to build or implement;
- to reveal what our “thing” will “look” like;
- to help us understand how our “thing” will behave.
Why do we want a model?

We are going to use a mathematics based notation to describe our models, but the intention is the same as in any application where a model is used. We want a model:

- to exhibit some important characteristics of some “thing” that we want to build or implement;
- to reveal what our “thing” will “look” like;
- to help us understand how our “thing” will behave.
Why do we want a model?

We are going to use a mathematics based notation to describe our models, but the intention is the same as in any application where a model is used. We want a model:

- to exhibit some important characteristics of some “thing” that we want to build or implement;
- to reveal what our “thing” will “look” like;
- to help us understand how our “thing” will behave.
A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed Sydney Harbour Bridge we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
What should a model do?

A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed Sydney Harbour Bridge we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed *Sydney Harbour Bridge* we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
What should a model do?

A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed Sydney Harbour Bridge we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
What should a model do?

A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed *Sydney Harbour Bridge* we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
What should a model do?

A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed *Sydney Harbour Bridge* we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
What should a model do?

A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed *Sydney Harbour Bridge* we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
What should a model do?

A model must provide us with the capability of doing measurements that will enable us to establish the behaviour of the “thing” being modelled.

If we were modelling a proposed *Sydney Harbour Bridge* we would want to know:

- what the bridge looks like;
- the length of the bridge;
- the width of the bridge;
- the maximum load of the bridge;
- etc;
Models are not implementations

In general a model is not an implementation:

If we wanted a model of a proposed Sydney Harbour Bridge, we would not build the bridge.

Why not?
Models are not implementations

In general a model is not an implementation:

- if we wanted a model of a proposed Sydney Harbour Bridge, we would not build the bridge.

Why not?
Models are not implementations

In general a model is not an implementation:

- if we wanted a model of a proposed Sydney Harbour Bridge, we would not build the bridge.

Why not?
In general a model is not an implementation:

- if we wanted a model of a proposed Sydney Harbour Bridge, we would not build the bridge.

Why not?
Some think that software systems are different because of they are *soft*.

But, for similar reasons to why we build models of conventional engineering structures, it would be useful to build software models that concentrate on the system behaviour, not on how it is implemented.
Is Software different?

Some think that software systems are different because of they are soft.

But, for similar reasons to why we build models of conventional engineering structures, it would be useful to build software models that concentrate on the system behaviour, not on how it is implemented.
Some think that software systems are different because of they are *soft*.

But, for similar reasons to why we build models of conventional engineering structures, it would be useful to build software models that concentrate on the system behaviour, not on how it is implemented.
We can differentiate between 3 phases of development:

**Specification:** is concerned with describing, or modelling, what a system should do. There is no requirement that a specification is executable.

**Implementation:** is concerned with how the specification is to be realised. In a software implementation this must be executable.

**Design:** describes the process between specification and implementation. Until the final stage of design (implementation) intermediate stages may contain a mixture of specification and implementation. We will frequently call design refinement.
We can differentiate between 3 phases of development:

**Specification:** is concerned with describing, or modelling, what a system should do. There is no requirement that a specification is executable.

**Implementation:** is concerned with how the specification is to be realised. In a software implementation this must be executable.

**Design:** describes the process between specification and implementation. Until the final stage of design (implementation) intermediate stages may contain a mixture of specification and implementation. We will frequently call design refinement.
We can differentiate between 3 phases of development:

**Specification**: is concerned with describing, or modelling, what a system should do. There is no requirement that a specification is executable.

**Implementation**: is concerned with how the specification is to be realised. In a software implementation this must be executable.

**Design**: describes the process between specification and implementation. Until the final stage of design (implementation) intermediate stages may contain a mixture of specification and implementation. We will frequently call design refinement.
We can differentiate between 3 phases of development:

**Specification:** is concerned with describing, or modelling, what a system should do. There is no requirement that a specification is executable.

**Implementation:** is concerned with how the specification is to be realised. In a software implementation this must be executable.

**Design:** describes the process between specification and implementation. Until the final stage of design (implementation) intermediate stages may contain a mixture of specification and implementation. We will frequently call design refinement.
We can differentiate between 3 phases of development:

**Specification**: is concerned with describing, or modelling, what a system should do. There is no requirement that a specification is executable.

**Implementation**: is concerned with how the specification is to be realised. In a software implementation this must be executable.

**Design**: describes the process between *specification* and *implementation*. Until the final stage of design (implementation) intermediate stages may contain a mixture of specification and implementation. We will frequently call design *refinement*. 
A Simple Numeric Example

We mostly won’t be dealing with numerical algorithms, but the following simple example is complex enough to illustrate our point.

The following programming fragments are expressed in the low-level executable subset of B, the language we will be using to model systems. It doesn’t need much explanation, except to note that operations can return multiple results.
We mostly won’t be dealing with numerical algorithms, but the following simple example is complex enough to illustrate our point.

The following programming fragments are expressed in the low-level executable subset of B, the language we will be using to model systems. It doesn’t need much explanation, except to note that operations can return multiple results.
We mostly won’t be dealing with numerical algorithms, but the following simple example is complex enough to illustrate our point. The following programming fragments are expressed in the low-level executable subset of B, the language we will be using to model systems. It doesn’t need much explanation, except to note that operations can return multiple results.
ChooseInitApprox

\[
\text{low} , \text{high} \leftarrow \text{ChooseInitApprox} \ (\text{\textit{num}}) \\
\begin{align*}
\text{begin} & \quad \text{\textit{low}} := 0 \ ; \ 	ext{\textit{high}} := (\text{\textit{num}} + 1) / 2 + 1 \end{align*}
\]

Mystery Loop

\[
\text{while} \quad \text{low} + 1 \neq \text{high} \\
\text{do} \quad \text{low} , \text{high} \leftarrow \text{ChooseBetterApprox} \ (\text{\textit{num}} , \text{\textit{low}} , \text{\textit{high}}) \end{align*}
\]

ChooseBetterApprox

\[
\text{low} , \text{high} \leftarrow \text{ChooseBetterApprox} \ (\text{\textit{num}} , \text{\textit{oldlow}} , \text{\textit{oldhigh}}) \\
\begin{align*}
\text{var} & \quad \text{\textit{mid}} \ 	ext{in} \\
\text{\textit{mid}} & := (\text{\textit{oldlow}} + \text{\textit{oldhigh}}) / 2 \ ; \\
\text{if} & \quad \text{\textit{mid}} \leq \text{\textit{num}} / \text{\textit{mid}} \\
\text{then} & \quad \text{\textit{low}} := \text{\textit{mid}} \ ; \ 	ext{\textit{high}} := \text{\textit{oldhigh}} \\
\text{else} & \quad \text{\textit{low}} := \text{\textit{oldlow}} \ ; \ 	ext{\textit{high}} := \text{\textit{mid}} \\
\text{end}
\end{align*}
\]
**ChooseInitApprox**

\[
\text{low, high} \leftarrow \text{ChooseInitApprox}(\text{num}) \\
\begin{align*}
\text{begin} & \quad \text{low} := 0 ; \quad \text{high} := (\text{num} + 1) / 2 + 1 \text{ end ;}
\end{align*}
\]

**Mystery Loop**

\[
\text{while low} + 1 \neq \text{high} \\
\text{do low, high} \leftarrow \text{ChooseBetterApprox}(\text{num}, \text{low}, \text{high}) \text{ end}
\]

**ChooseBetterApprox**

\[
\text{low, high} \leftarrow \text{ChooseBetterApprox}(\text{num, oldlow, oldhigh}) \\
\begin{align*}
\text{var mid in} & \\
\quad \text{mid} := (\text{oldlow} + \text{oldhigh}) / 2 ; \\
\quad \text{if mid} \leq \text{num} / \text{mid} & \\
\quad \text{then low} := \text{mid} ; \quad \text{high} := \text{oldhigh} \\
\quad \text{else low} := \text{oldlow} ; \quad \text{high} := \text{mid} \\
\text{end}
\end{align*}
\]
ChooseInitApprox

\[ \text{low} , \text{high} \leftarrow \text{ChooseInitApprox} (\text{num}) \equiv \begin{align*}
\text{begin} & \quad \text{low} := 0 ; \quad \text{high} := (\text{num} + 1) / 2 + 1 \end{align*} ; \]

Mystery Loop

\[ \text{while} \quad \text{low} + 1 \neq \text{high} \]
\[ \text{do} \quad \text{low} , \text{high} \leftarrow \text{ChooseBetterApprox} (\text{num} , \text{low} , \text{high}) \quad \text{end} \]

ChooseBetterApprox

\[ \text{low} , \text{high} \leftarrow \text{ChooseBetterApprox} (\text{num} , \text{oldlow} , \text{oldhigh}) \equiv \begin{align*}
\text{var} & \quad \text{mid} \ \text{in} \\
\text{mid} & := (\text{oldlow} + \text{oldhigh}) / 2 ; \\
\text{if} & \quad \text{mid} \leq \text{num} / \text{mid} \\
\text{then} & \quad \text{low} := \text{mid} ; \quad \text{high} := \text{oldhigh} \\
\text{else} & \quad \text{low} := \text{oldlow} ; \quad \text{high} := \text{mid} \\
\text{end} \end{align*} \]
The Problem

What does it do?

Given an initial value of \(num\) as any natural number (non-negative integer), what does the loop compute?

How do you know?

How would you discover what it does?

Are you sure?

How would you verify that it does what you think it does?
ChooseInitApprox

\[
\text{low , high } \leftarrow \text{ ChooseInitApprox } ( \text{ num } ) \cong \\
\begin{align*}
\text{begin} & \quad \text{low} := 0 ; \\
& \quad \text{high} := ( \text{num} + 1 ) / 2 + 1 \text{ end ;}
\end{align*}
\]

Mystery Loop

\[
\text{while } \qquad \text{low} + 1 \neq \text{high} \\
\text{do } \quad \text{low , high } \leftarrow \text{ ChooseBetterApprox } ( \text{ num , low , high } ) \text{ end}
\]

ChooseBetterApprox

\[
\text{low , high } \leftarrow \text{ ChooseBetterApprox } ( \text{ num , oldlow , oldhigh } ) \cong \\
\begin{align*}
\text{var} & \quad \text{mid in} \\
& \quad \text{mid} := ( \text{oldlow} + \text{oldhigh} ) / 2 ; \\
& \quad \text{if} \quad \text{mid} \leq \text{num} / \text{mid} \\
& \quad \text{then} \quad \text{low} := \text{mid} ; \quad \text{high} := \text{oldhigh} \\
& \quad \text{else} \quad \text{low} := \text{oldlow} ; \quad \text{high} := \text{mid} \\
\text{end}
\end{align*}
\]
When the loop terminates the value of \( low \) is the largest natural number whose square does not exceed the value of \( num \), that is, the integer part of \( \sqrt{num} \).

Again, how would you verify that?
Indeed how do you read that from the algorithm?
Is it a good specification model?
When the loop terminates the value of \( low \) is the largest natural number whose square does not exceed the value of \( num \), that is, the integer part of \( \sqrt{num} \).

Again, how would you verify that?

Indeed how do you read that from the algorithm?

Is it a good specification model?
The Answer

When the loop terminates the value of low is the largest natural number whose square does not exceed the value of num, that is, the integer part of sqrt(num).

Again, how would you verify that?

Indeed how do you read that from the algorithm?

Is it a good specification model?
When the loop terminates the value of \(low\) is the largest natural number whose square does not exceed the value of \(num\), that is, the integer part of \(\sqrt{num}\).

Again, how would you verify that?

Indeed how do you read that from the algorithm?

Is it a good \textit{specification} model?
Consider an operation with a single natural number argument that is to compute the numerical square of the value of the argument.

The specification could look like the Sqr operation in the Square machine.
Consider an operation with a single natural number argument that is to compute the numerical square of the value of the argument. The specification could look like the Sqr operation in the **Square** machine.
Consider an operation with a single natural number argument that is to compute the numerical square of the value of the argument. The specification could look like the Sqr operation in the Square machine.
MACHINE Square

OPERATIONS

sqr ← Sqr (num) ⊑

PRE num ∈ ℕ

THEN sqr := num × num

END

END

Square is Deterministic and Computational

The specification Square not only clearly explains what the operation does, it also shows how to compute it.
MACHINE  Square

OPERATIONS
  \[ sqr \leftarrow Sqr \left( num \right) \triangleq \]
  PRE  \[ num \in \mathbb{N} \]
  THEN  \[ sqr := num \times num \]
  END
END

Square is Deterministic and Computational

The specification Square not only clearly explains what the operation does, it also shows how to compute it.
Suppose we wish to have an operation ApproxSqrt(num) that should return the largest natural number that does not exceed \( \sqrt{num} \), the mathematical square root of num.

We could specify this as shown in the ApproxSqrt operation of the SquareRoot machine.
Suppose we wish to have an operation \texttt{ApproxSqrt} (\texttt{num}) that should return the largest natural number that does not exceed $\sqrt{\texttt{num}}$, the mathematical square root of \texttt{num}.

We could specify this as shown in the \texttt{ApproxSqrt} operation of the \texttt{SquareRoot} machine.
Suppose we wish to have an operation ApproxSqrt(num) that should return the largest natural number that does not exceed \( \sqrt{num} \), the mathematical square root of \( num \).

We could specify this as shown in the ApproxSqrt operation of the \textbf{SquareRoot} machine.
MACHINE  *SquareRoot*

OPERATIONS

\[
\text{sqrt} \leftarrow \text{ApproxSqrt}\left(\text{num}\right) \triangleq \\
\text{PRE} \quad \text{num} \in \mathbb{N} \text{ THEN} \\
\quad \text{ANY} \quad \text{approx} \\
\quad \text{WHERE} \quad \text{approx} \in \mathbb{N} \land \\
\quad \quad \text{approx} \times \text{approx} \leq \text{num} \land \\
\quad \quad \text{num} < (\text{approx} + 1) \times (\text{approx} + 1) \\
\quad \text{THEN} \quad \text{sqrt} := \text{approx} \\
\quad \text{END} \\
\text{END} \\
\text{END}
\]
The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

What the specification says is:

\[
\text{if } \text{approx} \times \text{approx} \leq \text{num} \quad \text{and} \quad \text{(approx} + 1) \times (\text{approx} + 1) > \text{num} \\
\text{then } \text{approx} = \text{ApproxSqrt}(\text{num})
\]

Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

We use non-determinism to specify what not how.

The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.

Rather, the specification provides an acceptance test.
The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

What the specification says is:

if \( \text{approx} \times \text{approx} \leq \text{num} \) and
\(
(\text{approx} + 1) \times (\text{approx} + 1) > \text{num}
\)
then \( \text{approx} = \text{ApproxSqrt}(\text{num}) \)

The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

What the specification says is:

if \( \text{approx} \times \text{approx} \leq \text{num} \) and
\(
(\text{approx} + 1) \times (\text{approx} + 1) > \text{num}
\)
then \( \text{approx} = \text{ApproxSqrt}(\text{num}) \)
The use of non-determinism in specification

ApproxSqrt is specified non-determi

What the specification says is:

\[
\text{if } \text{approx} \times \text{approx} \leq \text{num} \quad \text{and} \quad \text{(approx} + 1) \times \text{(approx} + 1) > \text{num} \\
\text{then } \text{approx} = \text{ApproxSqrt}(\text{num})
\]

Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

We use non-determinism to specify what not how.

The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.
ApproxSqrt is specified non-deterministically

What the specification says is:

```
if     approx × approx ≤ num     and
       (approx + 1) × (approx + 1) > num

then   approx = ApproxSqrt(num)
```

Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic. They are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

We use non-determinism to specify what not how.

The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.

Rather, the specification provides an acceptance test.
The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

- What the specification says is:
  
  \[
  \text{if } \ approx \times approx \leq num \quad \text{and} \quad (approx + 1) \times (approx + 1) > num
  \]
  
  \[
  \text{then } \ approx = \text{ApproxSqrt}(num)
  \]

- Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

- In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

- We use non-determinism to specify what not how.

- The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.

- Rather, the specification provides an acceptance test.
The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

- What the specification says is:

<table>
<thead>
<tr>
<th>if</th>
<th>approx \times approx \leq num</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>(approx + 1) \times (approx + 1) &gt; num</td>
<td></td>
<td></td>
</tr>
<tr>
<td>then</td>
<td>approx = ApproxSqrt(num)</td>
<td></td>
</tr>
</tbody>
</table>

- Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

- In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

- We use non-determinism to specify what not how.

- The specification of ApproxSqrt is non-constructorive; it gives no clue to how we should compute the result.

- Rather, the specification provides an acceptance test.
ApproxSqrt is specified non-deterministically

What the specification says is:

\[
\begin{align*}
\text{if} & \quad \text{approx} \times \text{approx} \leq \text{num} \quad \text{and} \\
& \quad (\text{approx} + 1) \times (\text{approx} + 1) > \text{num} \\
\text{then} & \quad \text{approx} = \text{ApproxSqrt}(\text{num})
\end{align*}
\]

Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

- We use non-determinism to specify what not how.
- The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.
- Rather, the specification provides an acceptance test.
The use of non-determinism in specification

ApproxSQRT is specified non-deterministically

- What the specification says is:
  
  \[
  \text{if } \text{approx} \times \text{approx} \leq \text{num} \quad \text{and} \quad (\text{approx} + 1) \times (\text{approx} + 1) > \text{num} \\
  \text{then } \text{approx} = \text{ApproxSQRT} (\text{num})
  \]

- Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

- In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSQRT is deterministic; it is a function.

- **We use non-determinism to specify what not how.**

- The specification of ApproxSQRT is *non-constructive*; it gives no clue to how we should compute the result.

- Rather, the specification provides an acceptance test.
The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

- What the specification says is:
  - if \( \text{approx} \times \text{approx} \leq \text{num} \) and
  \[ (\text{approx} + 1) \times (\text{approx} + 1) > \text{num} \]
  - then \( \text{approx} = \text{ApproxSqrt}(\text{num}) \)

- Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

- In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

- We use non-determinism to specify what not how.

- The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.

- Rather, the specification provides an acceptance test.
The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

- What the specification says is:
  
  \[
  \text{if } \quad \text{approx} \times \text{approx} \leq \text{num} \quad \text{and} \quad (\text{approx} + 1) \times (\text{approx} + 1) > \text{num} \quad \text{then} \quad \text{approx} = \text{ApproxSqrt}(\text{num})
  \]

- Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

- In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

- We use non-determinism to specify what not how.

- The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.

- Rather, the specification provides an acceptance test.
The use of non-determinism in specification

ApproxSqrt is specified non-deterministically

What the specification says is:

| if | approx × approx ≤ num and (approx + 1) × (approx + 1) > num |
|    |                                                   |
|    | then approx = ApproxSqrt(num)                      |

Non-determinism is frequently used in writing formal specification. It allows the specification to concentrate on the properties of the operation.

In many, perhaps most, uses of non-determinism we are not specifying an operation that is itself non-deterministic; they are frequently deterministic. Certainly ApproxSqrt is deterministic; it is a function.

We use non-determinism to specify what not how.

The specification of ApproxSqrt is non-constructive; it gives no clue to how we should compute the result.

Rather, the specification provides an acceptance test.
While B promises an implementation that is consistent with the specification, there can be no promise that the specification is consistent with the informal requirements.

In general, a formal specification cannot be compiled and executed, but *animation* enables the specifier to validate the behaviour through—possibly human-assisted—interpretation of the specification.

We will animate the operations in `Square` and `SquareRoot`.

To animate a machine, select the `anm` button in the `Main` environment.
While B promises an implementation that is consistent with the specification, there can be no promise that the specification is consistent with the informal requirements.

In general, a formal specification cannot be compiled and executed, but *animation* enables the specifier to validate the behaviour through—possibly human-assisted—interpretation of the specification.

We will animate the operations in *Square* and *SquareRoot*.

To animate a machine, select the *anm* button in the *Main* environment.
While B promises an implementation that is consistent with the specification, there can be no promise that the specification is consistent with the informal requirements.

In general, a formal specification cannot be compiled and executed, but animation enables the specifier to validate the behaviour through—possibly human-assisted—interpretation of the specification.

We will animate the operations in Square and SquareRoot.

To animate a machine, select the anm button in the Main environment.
While B promises an implementation that is consistent with the specification, there can be no promise that the specification is consistent with the informal requirements.

In general, a formal specification cannot be compiled and executed, but *animation* enables the specifier to validate the behaviour through—possibly human-assisted—interpretation of the specification.

We will animate the operations in **Square** and **SquareRoot**.

To animate a machine, select the *anm* button in the *Main* environment.
While B promises an implementation that is consistent with the specification, there can be no promise that the specification is consistent with the informal requirements.

In general, a formal specification cannot be compiled and executed, but animation enables the specifier to validate the behaviour through —possibly human-assisted— interpretation of the specification.

We will animate the operations in **Square** and **SquareRoot**.

To animate a machine, select the **anm** button in the **Main** environment.
While B promises an implementation that is consistent with the specification, there can be no promise that the specification is consistent with the informal requirements.

In general, a formal specification cannot be compiled and executed, but animation enables the specifier to validate the behaviour through —possibly human-assisted— interpretation of the specification.

We will animate the operations in **Square** and **SquareRoot**.

To animate a machine, select the anm button in the **Main** environment.
Refinement is the name given to a design step in which we take a specification closer to an implementation.

A refinement of an operation can be thought of as the specification of an operation that is at least as acceptable as the operation being refined.
Refinement is the name given to a design step in which we take a specification closer to an implementation.

A refinement of an operation can be thought of as the specification of an operation that is at least as acceptable as the operation being refined.
Refinement is the name given to a design step in which we take a specification closer to an implementation.

A refinement of an operation can be thought of as the specification of an operation that is at least as acceptable as the operation being refined.
The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:

- $\text{approx} \times \text{approx} \leq \text{num}$ and
- $\text{num} < (\text{approx} + 1) \times (\text{approx} + 1)$

It is simple to choose a value of $\text{approx}$ to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, $\text{low}$ and $\text{high}$ with $\text{low} \leq \text{approx}$, $\text{low} < \text{high}$ and $\text{approx} < \text{high}$, as shown in the implementation machine `SquareRootI`, where the loop moves $\text{low}$ and $\text{high}$ closer together under the above invariant. Termination clearly occurs when $\text{low} + 1 = \text{high}$.

Notice that this machine uses the services of (IMPORTS) of a new machine `SqrtUtility`. 
The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:

1. \( \text{approx} \times \text{approx} \leq \text{num} \) and
2. \( \text{num} < (\text{approx} + 1) \times (\text{approx} + 1) \)

It is simple to choose a value of \( \text{approx} \) to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, \( \text{low} \) and \( \text{high} \) with \( \text{low} \leq \text{approx} \), \( \text{low} < \text{high} \) and \( \text{approx} < \text{high} \), as shown in the implementation machine \text{SquareRootI}, where the loop moves \( \text{low} \) and \( \text{high} \) closer together under the above invariant. Termination clearly occurs when \( \text{low} + 1 = \text{high} \).

Notice that this machine uses the services of (\text{IMPORTS}) of a new machine \text{SqrtUtility}. 
Refining ApproxSqrt

The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:

- \( \text{approx} \times \text{approx} \leq \text{num} \) and
- \( \text{num} < (\text{approx} + 1) \times (\text{approx} + 1) \)

It is simple to choose a value of \( \text{approx} \) to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, \( \text{low} \) and \( \text{high} \) with \( \text{low} \leq \text{approx} \), \( \text{low} < \text{high} \) and \( \text{approx} < \text{high} \), as shown in the implementation machine \( \text{SquareRootI} \), where the loop moves \( \text{low} \) and \( \text{high} \) closer together under the above invariant. Termination clearly occurs when \( \text{low} + 1 = \text{high} \).

Notice that this machine uses the services of \( \texttt{IMPORTS} \) of a new machine \( \texttt{SqrtUtility} \).
The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:

- $approx \times approx \leq num$ and
- $num < (approx + 1) \times (approx + 1)$

It is simple to choose a value of $approx$ to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, $low$ and $high$ with $low \leq approx$, $low < high$ and $approx < high$, as shown in the implementation machine $SquareRootI$, where the loop moves $low$ and $high$ closer together under the above invariant. Termination clearly occurs when $low + 1 = high$.

Notice that this machine uses the services of ($IMPORTS$) of a new machine $SqrtUtility$. 
Refining ApproxSqrt

The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:

- \( \text{approx} \times \text{approx} \leq \text{num} \) and
- \( \text{num} < (\text{approx} + 1) \times (\text{approx} + 1) \)

It is simple to choose a value of \( \text{approx} \) to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, \( \text{low} \) and \( \text{high} \) with \( \text{low} \leq \text{approx} \), \( \text{low} < \text{high} \) and \( \text{approx} < \text{high} \), as shown in the implementation machine \texttt{SquareRootI}, where the loop moves \( \text{low} \) and \( \text{high} \) closer together under the above invariant. Termination clearly occurs when \( \text{low} + 1 = \text{high} \).

Notice that this machine uses the services of \texttt{IMPORTS} of a new machine \texttt{SqrtUtility}. 
The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:

- \( \text{approx} \times \text{approx} \leq \text{num} \) and
- \( \text{num} < (\text{approx} + 1) \times (\text{approx} + 1) \)

It is simple to choose a value of \( \text{approx} \) to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, \( \text{low} \) and \( \text{high} \) with \( \text{low} \leq \text{approx} \), \( \text{low} < \text{high} \) and \( \text{approx} < \text{high} \), as shown in the implementation machine \textbf{SquareRootI}, where the loop moves \text{low} and \text{high} closer together under the above invariant. Termination clearly occurs when \( \text{low} + 1 = \text{high} \).

Notice that this machine uses the services of (\textbf{IMPORTS}) of a new machine \textbf{SqrtUtility}. 
The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:

- $approx \times approx \leq num$ and
- $num < (approx + 1) \times (approx + 1)$

It is simple to choose a value of $approx$ to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, $low$ and $high$ with $low \leq approx$, $low < high$ and $approx < high$, as shown in the implementation machine $SquareRootI$, where the loop moves $low$ and $high$ closer together under the above invariant. Termination clearly occurs when $low + 1 = high$.

Notice that this machine uses the services of (IMPORTS) of a new machine $SqrtUtility$. 
IMPLEMENTATION   *SquareI*

REFINES   *Square*

OPERATIONS

\[ \text{sq}r \leftarrow \text{Sqr}(\text{num}) \triangleq \]

\[
\begin{align*}
\text{BEGIN} \\
\text{sqr} & \leftarrow \text{num} \times \text{num} \\
\text{END}
\end{align*}
\]

END
IMPLEMENTATION SquareRootI
REFINES SquareRoot
IMPORTS SqrtUtility

OPERATIONS
SquareRootII

\[
\begin{align*}
\text{sqrt} & \Leftarrow \text{ApproxSqrt} \ (\ num \ ) \\
\text{VAR} & \quad \text{low}, \text{high} \ \text{IN} \\
\text{low}, \text{high} & \Leftarrow \text{ChooseInitApprox} \ (\ num \ ) \\
\text{WHILE} & \quad \text{low} + 1 \neq \text{high} \\
\text{DO} & \quad \text{low}, \text{high} \Leftarrow \text{ChooseBetterApprox} \ (\ num, \text{low}, \text{high} \ ) \\
\text{INVARIANT} & \quad \text{low} \in \mathbb{N} \land \text{high} \in \mathbb{N} \land \\
& \quad \text{low} + 1 \leq \text{high} \land \\
& \quad \text{low} \times \text{low} \leq \text{num} \land \\
& \quad \text{num} < \text{high} \times \text{high} \\
\text{VARIANT} & \quad \text{high} - \text{low} \\
\text{END} \ ; \\
\text{sqrt} := \text{low} \\
\text{END} \\
\text{END}
\end{align*}
\]
MACHINE SqrtUtility

OPERATIONS
  low, high ← ChooseInitApprox ( num ) ≜
  
  PRE num ∈ \mathbb{N} THEN
  
  ANY xx, yy
  
  WHERE xx ∈ \mathbb{N} ∧ yy ∈ \mathbb{N} ∧
    xx + 1 ≤ yy ∧
    xx × xx ≤ num ∧
    num < yy × yy
  
  THEN low, high := xx, yy
  
  END

END ;
low, high ← ChooseBetterApprox (num, oldlow, oldhigh) ≜ 
PRE num ∈ ℕ ∧ oldlow ∈ ℕ ∧ oldhigh ∈ ℕ ∧ 
oldlow × oldlow ≤ num ∧ num < oldhigh × oldhigh ∧ 
oldlow + 1 < oldhigh 
THEN
ANY xx, yy
WHERE xx ∈ ℕ ∧ yy ∈ ℕ ∧ 
xx + 1 ≤ yy ∧ 
xx × xx ≤ num ∧ 
num < yy × yy ∧
yy – xx < oldhigh – oldlow
THEN low, high := xx, yy
END
END
END
Refining SqrtUtility

Having introduced the machine SqrtUtility we now have to refine it to an implementation.

This is done in three refinement steps as shown in machines SqrtUtilityR, SqrtUtilityRR and SqrtUtilityRRI.
REFINEMENT  $SqrtUtilityR$

REFINES  $SqrtUtility$

OPERATIONS

$low, high \leftarrow \text{ChooseInitApprox}(\ num) \triangleq$

$low, high := 0, (num + 1) / 2 + 1;$
low, high ← ChooseBetterApprox (num, oldlow, oldhigh) ≜
ANY mid
WHERE mid ∈ ℕ ∧ oldlow < mid ∧ mid < oldhigh THEN
SELECT mid × mid ≤ num THEN
low, high := mid, oldhigh
WHEN num < mid × mid THEN
low, high := oldlow, mid
END
END
END
REFINEMENT SqrtUtilityRR
REFINES SqrtUtilityR

OPERATIONS
  low, high ← ChooseInitApprox ( num )

  BEGIN  low := 0 ; high := ( num + 1 ) / 2 + 1 END ;
low, high ← ChooseBetterApprox (num, oldlow, oldhigh) 

VAR mid IN

mid := (oldlow + oldhigh) / 2 ;

IF mid × mid ≤ num

THEN low := mid ; high := oldhigh

ELSE low := oldlow ; high := mid

END

END

END
IMPLEMENTATION  \texttt{SqrtUtilityRRI}
REFINES  \texttt{SqrtUtilityRR}

OPERATIONS
\begin{verbatim}
  low, high ← ChoselnInitApprox ( num ) ≡
  BEGIN
    low := 0 ; high := ( num + 1 ) / 2 + 1
  END ;
\end{verbatim}
low, high ← ChooseBetterApprox (num, oldlow, oldhigh)

VAR mid IN

mid := (oldlow + oldhigh) / 2;

IF mid ≤ num / mid
THEN low := mid; high := oldhigh
ELSE low := oldlow; high := mid
END
END
END
Having produced implementation machines, Squarel and SqrtUtilityRRI, we can produce code by selecting the `trl` button in the Translators environment of the B-Toolkit.

The C code is shown in the appendix.
To run the code we can introduce an interface to Square and then generate the interface in the **Generators** environment of the B-Toolkit. The interface can be run by selecting the **exe** (execute) button in the **Translators** environment.

The interface looks similar to that of the Animator, but this is not an interpretative execution; you are now running actual code.
Generating an interface

To run the code we can introduce an interface to Square and then generate the interface in the **Generators** environment of the B-Toolkit.

The interface can be run by selecting the **exe** (execute) button in the **Translators** environment.

The interface looks similar to that of the Animator, but this is not an interpretative execution; you are now running actual code.
#include "SquareRoot.h"

#include "SqrtUtility.h"

void
INI_SquareRoot()
{
    INI_SqrtUtility();
}

void ApproxSqrt(_sqrt, _num)
    int *__sqrt, _num;
{
    int low, high;
    ChooseInitApprox(&low, &high, _num);
    while (low+1 != high) {
        ChooseBetterApprox(&low, &high, _num, low, high);
    }
    *__sqrt = low;
}
void INI_SquareRoot();
void ApproxSqrt();
```c
#include "SqrtUtility.h"

void
INI_SqrtUtility()
{

}

void
ChooseInitApprox(_low , _high,_num)
int *_low,*_high,__num;
{
   *__low = 0;
   *__high = ((__num+1)/2+1);
}
```
void
ChooseBetterApprox(_low, _high, _num, _oldlow, _oldhigh)
int *__low, *__high, _num, _oldlow, _oldhigh;
{
    int mid;
    mid = (_oldlow+_oldhigh)/2;
    if ( mid <= _num/mid ) {
        *__low = mid;
        *__high = _oldhigh;
    }
    else {
        *__low = _oldlow;
        *__high = mid;
    }
}
void INI_SqrtUtility();
void ChooseInitApprox();
void ChooseBetterApprox();
Proof Obligations for SquareRootI.imp

\[
\text{ApproxSqrt.1} \\
\text{cst} \left( \text{SquareRootI}_1 \right) \land \text{ctx} \left( \text{SquareRootI}_1 \right) \land \\
\text{inv} \left( \text{SquareRootI}_1 \right) \land \text{asn} \left( \text{SquareRootI}_1 \right) \land \\
\text{pre} \left( \text{ApproxSqrt} \right) \Rightarrow \\
\text{low} \in \mathbb{N} \land \\
\text{high} \in \mathbb{N} \land \\
\text{low} + 1 \leq \text{high} \land \\
\text{low} \times \text{low} \leq \text{num} \land
\]
SquareRootI II

\[ \text{num} < \text{high} \times \text{high} \]
\[ \Rightarrow \]
\[ \text{high} - \text{low} \in \mathbb{N} \]

\underline{\text{ApproxSqrt.2}}
\[ \text{cst} ( \text{SquareRootI}_1) \land \text{ctx} ( \text{SquareRootI}_1) \land \]
\[ \text{inv} ( \text{SquareRootI}_1) \land \text{asn} ( \text{SquareRootI}_1) \land \]
\[ \text{pre} ( \text{ApproxSqrt} ) \]
\[ \Rightarrow \]
\[ \text{low} \in \mathbb{N} \land \]
\[ \text{high} \in \mathbb{N} \land \]
\[ \text{low} + 1 \leq \text{high} \land \]
\[ \text{low} \times \text{low} \leq \text{num} \land \]
SquareRootI III

\[ \text{num} < \text{high} \times \text{high} \land \\
\neg (\text{low} + 1 = \text{high}) \\
\Rightarrow \\
\text{low} + 1 < \text{high} \]

\[ \text{ApproxSqrt.3} \\
\text{cst} (\text{SquareRootI}_1) \land \text{ctx} (\text{SquareRootI}_1) \land \\
\text{inv} (\text{SquareRootI}_1) \land \text{asn} (\text{SquareRootI}_1) \land \\
\text{pre} (\text{ApproxSqrt}) \\
\Rightarrow \\
\text{low} \in \mathbb{N} \land \\
\text{high} \in \mathbb{N} \land \\
\text{low} + 1 \leq \text{high} \land \]
\[ \text{low} \times \text{low} \leq \text{num} \land \\
\text{num} < \text{high} \times \text{high} \land \\
\text{low} + 1 = \text{high} \Rightarrow \\
\text{num} < (\text{low} + 1) \times (\text{low} + 1) \]
Proof Obligations for SqrtUtilityR.ref

\[ \text{ChooseInitApprox.1} \]
\[
cst (\text{SqrtUtilityR}_1) \land ctx (\text{SqrtUtilityR}_1) \land
\]
\[
inv (\text{SqrtUtilityR}_1) \land asn (\text{SqrtUtilityR}_1) \land
\]
\[
pre (\text{ChooseInitApprox}) \Rightarrow
\]
\[ 0 \in \mathbb{N} \]

\[ \text{ChooseInitApprox.2} \]
\[
cst (\text{SqrtUtilityR}_1) \land ctx (\text{SqrtUtilityR}_1) \land \]
inv ( SqrtUtilityR₁ ) \land \text{asn} ( SqrtUtilityR₁ ) \land 
pre ( ChooseInitApprox ) 
\Rightarrow 
(\text{num} + 1) / 2 + 1 \in \mathbb{N} 

\hline

ChooseInitApprox.3 
\underline{cst} ( SqrtUtilityR₁ ) \land \text{ctx} ( SqrtUtilityR₁ ) \land 
inv ( SqrtUtilityR₁ ) \land \text{asn} ( SqrtUtilityR₁ ) \land 
pre ( ChooseInitApprox ) 
\Rightarrow 
1 \leq (\text{num} + 1) / 2 + 1 

\hline

ChooseInitApprox.4
cst ( SqrtUtilityR₁ ) ∧ ctx ( SqrtUtilityR₁ ) ∧
inv ( SqrtUtilityR₁ ) ∧ asn ( SqrtUtilityR₁ ) ∧
pre ( ChooseInitApprox )
⇒
0 ≤ num

ChooseInitApprox.5

______________________________

cst ( SqrtUtilityR₁ ) ∧ ctx ( SqrtUtilityR₁ ) ∧
inv ( SqrtUtilityR₁ ) ∧ asn ( SqrtUtilityR₁ ) ∧
pre ( ChooseInitApprox )
⇒
num < ( ( num + 1 ) / 2 + 1 ) × ( ( num + 1 ) / 2 + 1 )
ChooseBetterApprox.1
\[
\begin{align*}
\text{cst}(\text{SqrtUtilityR}_1) \land \text{ctx}(\text{SqrtUtilityR}_1) \land \\
\text{inv}(\text{SqrtUtilityR}_1) \land \text{asn}(\text{SqrtUtilityR}_1) \land \\
\text{pre}(\text{ChooseBetterApprox})
\end{align*}
\]
\[
\Rightarrow
\]
\[
\begin{align*}
\text{mid} \in \mathbb{N} \land \\
\text{oldlow} < \text{mid} \land \\
\text{mid} < \text{oldhigh} \land \\
\text{mid} \times \text{mid} \leq \text{num}
\end{align*}
\]
\[
\Rightarrow
\]
\[
\text{mid} + 1 \leq \text{oldhigh}
\]

ChooseBetterApprox.2
\( \text{cst} (\text{SqrtUtilityR}_1) \land \text{ctx} (\text{SqrtUtilityR}_1) \land \text{inv} (\text{SqrtUtilityR}_1) \land \text{asn} (\text{SqrtUtilityR}_1) \land \text{pre} (\text{ChooseBetterApprox}) \)

\[ \Rightarrow \]

\( \text{mid} \in \mathbb{N} \land \text{olldlow} < \text{mid} \land \text{mid} < \text{olldhigh} \land \text{mid} \times \text{mid} \leq \text{num} \)

\[ \Rightarrow \]

\( \text{olldhigh} - \text{mid} < \text{olldhigh} - \text{olldlow} \)

\( \overline{\text{ChooseBetterApprox.3}} \)

\( \text{cst} (\text{SqrtUtilityR}_1) \land \text{ctx} (\text{SqrtUtilityR}_1) \land \)
inv ( SqrtUtilityR ₁ ) ∧ asn ( SqrtUtilityR ₁ ) ∧
pre ( ChooseBetterApprox )
⇒
mid ∈ N ∧
oldlow < mid ∧
mid < oldhigh ∧
num < mid × mid
⇒
oldlow + 1 ≤ mid

ChooseBetterApprox.4
\[ \text{cst ( SqrtUtilityR ₁ ) ∧ ctx ( SqrtUtilityR ₁ ) ∧} \]
\[ \text{inv ( SqrtUtilityR ₁ ) ∧ asn ( SqrtUtilityR ₁ ) ∧} \]
pre (ChooseBetterApprox)

\[ mid \in \mathbb{N} \land \oldlow < mid \land mid < \oldhigh \land num < mid \times mid \]

\[ \Rightarrow mid - \oldlow < \oldhigh - \oldlow \]
Proof Obligations for SqrtUtilityRR.ref

\[\text{ChooseBetterApprox.1} \]
\[\text{cst} (\text{SqrtUtilityRR}_2) \land \text{ctx} (\text{SqrtUtilityRR}_2) \land \]
\[\text{inv} (\text{SqrtUtilityRR}_2) \land \text{asn} (\text{SqrtUtilityRR}_2) \land \]
\[\text{pre} (\text{ChooseBetterApprox}) \]
\[\Rightarrow \]
\[\text{num} < (\text{oldlow} + \text{oldhigh}) / 2 \times ((\text{oldlow} + \text{oldhigh}) / 2) \]
\[\Rightarrow \]
\[\exists \text{mid}_0 . (\]
\[\text{mid}_0 \in \mathbb{N} \land\]
oldlow \ < \ mid_0 \ \land \\
mid_0 \ < \ oldhigh \ \land \\
( ( \ mid_0 \times \ mid_0 \leq \ num \ \land \ ( \ oldlow + \ oldhigh ) / 2 = \ oldhigh \ \land \\
oldlow = \ mid_0 ) ) \\
\lor \\
( \ num < \ mid_0 \times \ mid_0 \ \land \ ( \ oldlow + \ oldhigh ) / 2 = \ mid_0 ) \\
)

\text{ChooseBetterApprox.2} \\
cst ( SqrtUtilityRR_2 ) \ \land \ ctx ( SqrtUtilityRR_2 ) \ \land \\
inv ( SqrtUtilityRR_2 ) \ \land \ asn ( SqrtUtilityRR_2 ) \ \land \\
pres ( \text{ChooseBetterApprox} ) \\
\Rightarrow
\[ \frac{(oldlow + oldhigh)}{2} \times \left(\frac{(oldlow + oldhigh)}{2}\right) \leq num \]

\[ \Rightarrow \]

\[ \exists \; mid_0 . \left( \right. \]

\[ mid_0 \in \mathbb{N} \land \]

\[ oldlow < mid_0 \land \]

\[ mid_0 < oldhigh \land \]

\[ \left( (mid_0 \times mid_0) \leq num \land \left(\frac{(oldlow + oldhigh)}{2}\right) = mid_0 \right) \]

\[ \lor \]

\[ (num < mid_0 \times mid_0 \land oldhigh = mid_0 \land \left(\frac{(oldlow + oldhigh)}{2}\right) = oldlow) \]

\[ ) \]
Proof Obligations for SqrtUtilityRRI.imp

ChooseBetterApprox.1
\[\text{cst (}\ SqrtUtilityRRI_3\text{)} \land \text{ctx (}\ SqrtUtilityRRI_3\text{)} \land \text{inv (}\ SqrtUtilityRRI_3\text{)} \land \text{asn (}\ SqrtUtilityRRI_3\text{)} \land \text{pre (}\ \text{ChooseBetterApprox}\ )}\]
\[\Rightarrow\]
\[\text{num} / ( (\ \text{oldlow} + \text{oldhigh}) / 2 ) < (\ \text{oldlow} + \text{oldhigh}) / 2\]
\[\Rightarrow\]
\[ (\ \text{oldlow} + \text{oldhigh}) / 2 \times ( (\ \text{oldlow} + \text{oldhigh}) / 2 ) \leq \text{num} \land (\ \text{oldlow} + \text{oldhigh}) / 2 = \text{oldhigh} \land \]
\[
oldlow = \frac{oldlow + oldhigh}{2}
\]
\[
\lor
\]
\[
num < \frac{oldlow + oldhigh}{2} \times \left(\frac{oldlow + oldhigh}{2}\right)
\]

---

\[
\text{ChooseBetterApprox.2}
\]
\[
cst\left(\text{SqrtUtilityRRI}_{3}\right) \land ctx\left(\text{SqrtUtilityRRI}_{3}\right) \land
\]
\[
inv\left(\text{SqrtUtilityRRI}_{3}\right) \land asn\left(\text{SqrtUtilityRRI}_{3}\right) \land
\]
\[
pre\left(\text{ChooseBetterApprox}\right)
\]
\[
\Rightarrow
\]
\[
\frac{oldlow + oldhigh}{2} \leq num / \left(\frac{oldlow + oldhigh}{2}\right)
\]
\[
\Rightarrow
\]
\[
\frac{oldlow + oldhigh}{2} \times \left(\frac{oldlow + oldhigh}{2}\right) \leq num
\]
\[
\lor
\]
num < (oldlow + oldhigh) / 2 \times ( (oldlow + oldhigh) / 2 ) \land 
oldhigh = (oldlow + oldhigh) / 2 \land 
( oldlow + oldhigh ) / 2 = oldlow