System Modelling and Design
Introduction

Revision: 1.3, March 3, 2007

Ken Robinson

March 3, 2007

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

Contents

1 Introduction 2
   1.1 What is this course about? 2
   1.2 What? 2
   1.3 Why? 3
   1.4 Models must have measurements! 3

2 Software Modelling 3
   2.1 Software 3
   2.2 Specifications,... 3
   2.3 Go Back, You are Going the Wrong Way! 4
   2.4 Example 4
   2.5 Specification 5
   2.6 Animation 6
   2.7 Implementing SquareRoot 6
   2.8 Interface 9
   2.9 Verification of Implementation 9
   2.10 A Note on Proof 10
   2.11 An Important Caveat 10
   2.12 A Homily: Formality is inexorable 10

A Translations 11
   A.1 SquareRoot.c 11
   A.2 SquareRoot.h 11
   A.3 SqrUtility.c 11
1 Introduction

1.1 What is this course about?

This course is concerned with modelling, design and implementation of systems. We will be using mathematics in the form of the B Method \((B)\) to reason about the consistency of our models. Sometimes this would be called Formal Methods. We are going to resist that description as it brings with it many bad connotations and misconceptions. We wish this course to have a strong software engineering flavour. Certainly there isn’t a lot of use of B, but it’s not true to say there is none. Siemens make considerable use of B in their rail developments. Let’s take a look at a paper presented to the B2007 conference in Besançon, France in January 2007. This paper covers the period of use of B in rail development commencing with the driverless Meteor Line (line 14) of the Paris Metro completed in 1998 through to the refurbishment of the Carnesie Line in New York in 2006. In this course we will be concerned with the engineering design of systems, especially software systems. Engineering design has two concerns:

The world view: how the system should behave in the real world. We will call this specification.

The internal view: how to use technology to achieve the required real world behaviour. We will call this implementation.

We use modelling to deal with both, and for precise, rigorous modelling we will use B.

1.2 What is a model?

A model provides us with a demonstration of some properties of a system without necessarily building the system.

Indeed in the early stages of system design it is important that we don’t build the system.

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 capability of the bridge;
- etc;

We wouldn’t want to build the bridge to do that.
1.3 Why do we want a model?

All engineering and design disciplines use models.
We need simple ways of verifying our ideas of what it is we want to build, without going to the expense of building it, and perhaps finding that we didn’t understand what we really wanted.

1.4 Models must have measurements!

A model must provide us with the capability of doing measurements that will enable us to validate the behaviour of the “thing” being modelled.
Software designers sometimes use Unified Modelling Language (UML), which allows us to draw models, but there are no measurements!
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;
- and we want to be sure that our “thing” achieves the required behaviour.

2 Software Modelling

2.1 Is Software different?

Some think that software systems are different because they are soft. That is a partial answer: the softness of software systems does contrast with many hard systems that involve hard, natural materials and manufacturing. It’s not a total answer as software systems can be very expensive and they can go horribly wrong.
But, for similar reasons to why we build models of conventional engineering structures, it would be useful to build software models that concentrate on particular aspects of system behaviour.

2.2 Specification, Design and Implementation

We can differentiate between 3 phases of development:

Specification: is concerned with modelling, what a system should do. There is no requirement that a specification is executable. Specifications model the user or world view of a system, not internal system behaviour.

Refinement or Design: a process of transformation of the specification towards implementation. This must be achieved without changing the world view.

Implementation: the final refinement or design step in which we make the model executable. Implementation is concerned with how the system is to be realised.
2.3 Go Back, You are Going the Wrong Way!

People with a background in programming frequently have difficulty describing a specification model. They tend to think in implementation terms: how would I implement this thing.

As soon as you start thinking of loops or sequential composition (do this and then do that), stop, you are going the wrong way. You are thinking how, not what.

One useful way to think of a system is to think of yourself at an interface: What do you do? and What do you expect the system to do in response?

If you were explaining such an interface to a friend, it would be useless to launch into details of how it might be implemented internally.

As we will now demonstrate.

2.4 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.

ChooseInitApprox

\[
\text{ChooseInitApprox} \left( \text{num} \right) \triangleq \\
\begin{aligned}
\text{begin} & \quad \text{low} := 0 ; \\
& \quad \text{high} := (\text{num} + 1) / 2 + 1 \end{aligned}
\]

Mystery Loop

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

ChooseBetterApprox

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

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?

**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?

### 2.5 Specification
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 \textbf{Square} machine.

**MACHINE** \textbf{Square}

**OPERATIONS**
\[
\text{sqr} \leftarrow Sqr(\text{num}) \equiv \\
\quad \text{pre} \quad \text{num} \in \mathbb{N} \\
\quad \text{then} \quad \text{sqr} := \text{num} \times \text{num} \\
\text{end}
\]

\( \mathbb{N} \) is the set of natural numbers = \{0, 1, 2, \ldots\}

\textit{Square} is Deterministic and Computational

The specification \textit{Square} not only clearly explains what the operation does, it also shows how to compute it. In this case \textit{what} and \textit{how} coincide.

Suppose we wish to have an operation \( \text{ApproxSqrt}(\text{num}) \) that should return the largest natural number that does not exceed \( \sqrt{\text{num}} \), the mathematical square root of \( \text{num} \).

We could specify this as shown in the \( \text{ApproxSqrt} \) operation of the \textbf{SquareRoot} machine

**MACHINE** \textbf{SquareRoot}

**OPERATIONS**
\[
\text{sqrt} \leftarrow \text{ApproxSqrt}(\text{num}) \equiv \\
\quad \text{pre} \quad \text{num} \in \mathbb{N} \text{ then} \\
\quad \text{any} \quad \text{approx}
\]
\[
\text{where } \text{approx} \in \mathbb{N} \land \\
\text{approx} \times \text{approx} \leq \text{num} \land \\
\text{num} < (\text{approx} + 1) \times (\text{approx} + 1)
\]
\[
\text{then } \text{sqrt} := \text{approx}
\]

\end{end}

\text{END}

**ApproxSqrt is specified non-deterministically**

- What the specification says is:

<table>
<thead>
<tr>
<th>Find</th>
<th>a value of \text{approx}</th>
</tr>
</thead>
<tbody>
<tr>
<td>such that</td>
<td>\text{approx} \times \text{approx} \leq \text{num} and</td>
</tr>
<tr>
<td></td>
<td>(\text{approx} + 1) \times (\text{approx} + 1) &gt; \text{num}</td>
</tr>
<tr>
<td>then</td>
<td>\text{approx} = \text{ApproxSqrt}(\text{num})</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 \text{ApproxSqrt} is deterministic; it is a function.

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

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

- Rather, the specification provides an acceptance test.

### 2.6 Animation

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 \text{Square} and \text{SquareRoot}.

To animate a machine, select the \text{anm} button in the \text{Main} environment.

### 2.7 Implementing SquareRoot

Without going into detailed reasons here we can proceed to implement the the SquareRoot machine. To do that we specify a new machine SqrUtility to help with the implementation and then we proceed to implement that machine.

**IMPLEMENTATION** \text{SquareRootI}
REFINES SquareRoot
IMPORTS SqrtUtility

OPERATIONS

sqrt ←− ApproxSqrt ( num ) ≡

var low , high in
  low , high ←− ChooseInitApprox ( num ) ;
  while low + 1 ≠ high do
    low , high ←− ChooseBetterApprox ( num , low , high )
  invariant
    low ∈ N ∧ high ∈ N ∧
    low + 1 ≤ high ∧
    low × low ≤ num ∧
    num < high × high
  variant
    high − low
  end ;
  sqrt := low
end

END

MACHINE SqrtUtility

OPERATIONS

low , high ←− ChooseInitApprox ( num ) ≡

pre num ∈ N then
  any xx , yy
  where xx ∈ N ∧ yy ∈ 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 ∧
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**

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

\[
\begin{aligned}
\text{low, high} & \leftarrow \text{ChooseBetterApprox} \ (num, oldlow, oldhigh) \triangleq \\
& \text{any} \quad \text{mid} \quad \text{where} \quad \text{mid} \in N \land \text{oldlow} < \text{mid} \land \text{mid} < \text{oldhigh} \quad \text{then} \\
& \quad \text{select} \quad \text{mid} \times \text{mid} \leq num \quad \text{then} \\
& \quad \quad \text{low, high} := \text{mid}, \text{oldhigh} \\
& \quad \text{when} \quad num < \text{mid} \times \text{mid} \quad \text{then} \\
& \quad \quad \text{low, high} := \text{oldlow}, \text{mid} \\
& \text{end} \end{aligned}
\]

**REFINEMENT $SqrtUtilityRR$**

**REFINES $SqrtUtilityR$**

**OPERATIONS**

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

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

8
```plaintext
else  low := oldlow ; high := mid
end
end
END

IMPLEMENTATION SqrtUtilityRRI
REFINES SqrtUtilityRR

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 ≤ num / mid
          then  low := mid ; high := oldhigh
      else  low := oldlow ; high := mid
      end
END

Having produced implementation machines, SquareI 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.

2.8 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.

2.9 Verification of Implementation

At each stage in the implementation proof obligations are generated by the BToolkit. These are verification conditions that must be discharged by the toolkit theorem prover.
If we can do that, then provided we have made no mistakes —yes mistakes can be made— the implementation will faithfully implement the specification.
These proof obligations are only possible because all B constructs have a formal semantics.
```
2.10 A Note on Proof

The word proof carries unfortunate connotations. It might be thought to imply infallibility. And talk of theorem provers might suggest artificial intelligence, or some other mystic cult.

Nothing of the sort is intended. All we are doing is carrying out computations in the first order predicate calculus, just as other engineers do computations in the differential or integral calculus.

We are just using mathematics to verify our models.

2.11 An Important Caveat

An important issue was raised by a student in today’s lecture (27/02/2006). That issue was feasibility.

What if in the specification of ApproxSqrt we had specified that num was a negative integer, the remainder being left unchanged?

Then apparently we are specifying a natural number that is an approximation to the square root of a negative integer.

Is this possible and what does it mean?

Well it’s certainly possible, but what it would mean is that anything you put forward as an implementation would produce proof obligations that you would not be able to discharge.

Notice that the specification is still perfectly clear as to what it means. However the specification is infeasible, meaning that it cannot be implemented.

Notice also that saying that any implementation you put forward cannot be verified is weaker than proving the specification cannot be implemented.

2.12 A Homily: Formality is inexorable

The increase of the use formality, ie the use of mathematics, in software development has been continuous, from formal grammars to specify programming language syntax, to the semi-formal application of translator generators in compiler implementation.

High-level programming languages themselves are an instance of increased formality, over machine level (assembler) programming in this case. OO design is usually conducted informally, but many of the concepts derive from formal ideas of, for example, abstract data types.

Mutual exclusion; synchronisation provide further examples.

Everywhere rigour and formality has been used there has been an increase in the reliability of implementations.

Engineering in general depends on rigour and mathematics.

There is no reason to believe that this “progress” will not continue.
A Translations

A.1 SquareRoot.c

#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;
}

A.2 SquareRoot.h

void INI_SquareRoot();
void ApproxSqrt();

A.3 SqrUtility.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;
```c
int mid;  
mid = (_oldlow+_oldhigh)/2;  
if ( mid <= _num/mid ) {  
    *low = mid;  
    *high = _oldhigh;
}  
else {  
    *low = _oldlow;  
    *high = mid;
}
```

A.4 SqrUtility.h

```c
void INI_SqrtUtility();  
void ChooseInitApprox();  
void ChooseBetterApprox();
```