An Introduction to the B Method

An Overview

Revision: 1.2, March 5, 2003

©Ken Robinson
mailto::k.robinson@unsw.edu.au
Introduction to B

In this course we will be introducing you to the formal method, whose full name is the B Method, but which is usually known simply as B.

B is one of the few formal software development methods that covers the complete software lifecycle, from requirements (specification), through design (refinement) to implementation, code generation, and maintenance.

In this course we will be concerned only with the use of B for specification.

To assist developments in B, we will be use the B-Toolkit, a configuration management tool.
What do we mean by Formal Methods?

In our use of the term Formal Methods we mean the application of mathematics—set theory and logic—to specify, design and implement software—or more generally systems—in such a way that the resulting code has been proved to be consistent with the original specification.

In B, a specification is a mathematical model of the required behaviour of a system. Specifications are generally abstract.

We then transform the specification through a sequence of formally defined refinement steps towards a concrete implementation.

During the process we have a number of proof obligations that must be discharged.
What do we mean by Formal Methods?

In our use of the term Formal Methods we mean the application of mathematics—set theory and logic—to specify, design and implement software—or more generally systems—in such a way that the resulting code has been proved to be consistent with the original specification.

In B, a specification is a mathematical model of the required behaviour of a system. Specifications are generally abstract.

We then transform the specification through a sequence of formally defined refinement steps towards a concrete implementation.

During the process we have a number of proof obligations that must be discharged.

Connection with “conventional” methods

Conventional software development methods usually express the requirements informally, in either structured or unstructured English, or using some structured notation: dataflow, entity relationship diagrams, or the unified modelling language (UML). Specifi-
cations are frequently expressed directly in programming code. Design then consists of “fleshing out” the code to produce an implementation.

In contrast, when using a formal development method, like B, the specification is an abstract description of the requirements, expressing what behaviour is required, rather than how to produce that behaviour.

It follows the design phase must affect a radical transformation of the specification in order to obtain executable code.
cations are frequently expressed directly in programming code. Design then consists of “fleshing out” the code to produce an implementation.

In contrast, when using a formal development method, like B, the specification is an abstract description of the requirements, expressing what behaviour is required, rather than how to produce that behaviour.

It follows the design phase must affect a radical transformation of the specification in order to obtain executable code.

**Testing within traditional engineering disciplines**

If we examine traditional engineering disciplines, such as electrical engineering, or civil engineering, we find the following:

- Designs are based on a mathematical theory of the materials, components, structures to be used in the implementation of bridges, buildings, electronic circuits.
- Then the testing consists of physical testing of the implementation, or a model of the implementation.
This is a successful strategy because the domains can be described by continuous mathematics: if a model conforms for some specific test input, it will also conform for input that is “less than” that input. Thus testing can be conducted for extreme values within domains.

This strategy does not work for discrete valued domains.
This is a successful strategy because the domains can be described by continuous mathematics: if a model conforms for some specific test input, it will also conform for input that is “less than” that input. Thus testing can be conducted for extreme values within domains.

This strategy does not work for discrete valued domains.

Contrast with testing of software

Software executes over discrete domains, and testing usually consists of probing points within that space. Thus testing can only confirm conformance of behaviour at specific points. Testing is therefore incapable, in general, of demonstrating conformance over the complete application domain.

Thus testing may confirm the presence of bugs, but not their absence.
Advantages of formal development and proof

Using a formal development method, we build a model using constructs that are described by precise mathematical theories. These models capture the behaviour in a complete application domain.

As we develop our specifications into implementations, the formal method produces proof obligations that basically describe the complete set of tests that confirm that the behaviours of the specification and the design are consistent—more correctly, not inconsistent.

Discharging the proof obligations is thus the counterpart of testing in other engineering disciplines.

A proof validates behaviour in a complete domain, not simply at a single point.
What has been done with formal methods?

Formal methods have been used in various mission critical applications, for example train control systems and smart cards. In some countries the use of formal methods is mandatory for certain critical systems.


The distributed control system handled the critical parts of the central control room, the wayside equipment along the track and on the platform, the onboard train control. The Météor system was developed by Matra Transport, now owned by Siemens. The following pages on the Siemens site mention the B-Method in connection with Safety and Innovation http://www.siemens-ts.com/pagesUS/Engagements/Securite.htm http://www.siemens-ts.com/pagesUS/Engagements/Innovation.htm

The use of B was mandated by the Paris transit authority, RATP.
The following page talks about the announcement of the award of a contract to install the Météor system on the New York subway.


For another application see


GEMPLUS is a company that develops smart cards.
Formality is inexorable

The increase of the use formality 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.

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

There is no reason to believe that this “progress” will not continue.
Course Objectives

The objective of the course is to get you to think more carefully about the specification phase of software development.

The objective of the use of a formal method is to encourage you to think more rigorously—even formally—about specification in particular, and to extend that to other phases of software system development.
The objectives of this lecture

- to introduce the mathematical toolkit: set theory and logic
- to introduce the standard notations
- to introduce the basic B substitutions
- to introduce abstract machines and abstract machine notation (AMN)
- to gain an overview of the complete process: specification, animation, refinement, implementation, proof
- to introduce the B-Toolkit
Proof, Infallibility and Documentation

We have already said that B allows us to prove that the implementation is consistent with the specification, but the words prove and proof have unfortunate connotations of infallibility.

This is not intended.

Nor should the B-Toolkit be regarded as an oracle, despite the fact that it contains a number of theorem provers—really proof assistants.

The B-Toolkit is a documentation tool that allows the whole development to be documented as human readable evidence of the developers efforts to ensure a sound implementation.
Specifications vs Implementations

In this course we will be talking mainly about specification, although in this lecture a complete overview will be presented. We frequently will be emphasizing the difference between specification and implementation.

In the specification domain we will be dealing with abstract concepts.

In the implementation domain we need to deal with finite constructs.

A good example of the difference between these two domains is the infamous Y2K bug.

Why?
The Y2K bug

The Y2k bug is one of a class of bugs that arise because of the following problem:

Many of the concepts and functions that we wish to implement in our computer systems are unbounded. For example, the concept of date does not have any known bound: the life of our solar system presumably is limited, but we don’t know what it is. In the 1960’s, when this bug was born, the end of the century looked a long way away.

On the other hand, the physical computer systems on which we have to implement our systems are finite. We have to map our unbounded concepts onto finite physical hardware. This is part of a process called design, and sometimes we get it wrong, very wrong!
Pedigree of B

The author of the B Method is Jean-Raymond Abrial and B is built on the foundations established by

Tony Hoare, Edsgar Dijkstra: Weakest Preconditions

Cliff Jones: Pre and Post conditions, VDM

Ralph Back, Carroll Morgan: Refinement Calculus

Jean-Raymond Abrial: Z specification notation and mathematical toolkit

Many others have, of course, made significant contributions to related formal methods.
The Development History of B

1985  Initial B concepts developed by J-R Abrial
1985–1988  BP-funded research project at PRG Oxford and BP IT Division, headed by J-R Abrial
1988–92  Commercial development of tool kit (with subcontract to GEC-Alsthom, France)
1992–  Alpha tests in industry
1994–  Release of toolkits
As its name implies, B (the B Method) is a method, not simply a notation. B is supported by a toolkit, which should be regarded as an essential requisite for using the method. There are two toolkits:

- B-Toolkit distributed by B-Core [http://www.b-core.com](http://www.b-core.com)

We will be using the B-Toolkit.
Notation

All components of a B development will have a source form, used to specify machines and other input to the B-Toolkit, and a publication form used in documentation.

The notation for the source form will be ascii. For example, account : ACCOUNT means the variable account is an element of the set ACCOUNT.

The notation for publication will be \LaTeX markup, yielding high quality mathematical documents. For example, account \in ACCOUNT, which has the same meaning as ascii example.
Abstract Machines

B uses Abstract Machines, which are machines that encapsulate:

state consisting of a set of variables constrained by an invariant
and
operations operations may change the state, while maintaining the invariant, and may return a sequence of results.
Machine Variables in B

For technical reasons that will not be explained now, machine variables in B must have at least two characters. Thus xx is a valid variable, while x is not.

Warning: this is likely to cause many mysterious problems in your first attempts to write B machines. The error messages of the B-Toolkit will not clearly identify the problem!

Where single letters are used in describing the notation, those letters are placeholders, representing some form of expression, which might be a proper variable.
Object based

Abstract machines are sometimes described as object-based, rather than object-oriented. You will notice that a machine can be compared with an object, an instance of a class. Importantly, a machine does not behave as a class, although it is possible to model a class.
Abstract Machine Notation (AMN) is the notation used to describe Abstract Machines. AMN also incorporates a syntactic dressing up of the basic generalized substitution language (GSL).

AMN gives B an appearance and a feel of a programming language, although the level of abstraction is not changed by this syntactic sugaring.
Substitutions

The foundation of B machines is the substitution, and a set of substitutions that Abrial calls the Generalised Substitution Language or GSL. The GSL notation will not be described in this lecture.

A substitution is a construct that, in some way, changes the values of a set of variables, that is the current values of a set of variables are substituted by a new set of values.

The concept of the substitution is founded on the basic notion that the only way a state machine makes progress is by changing the value of the state.

Substitutions are given a formal semantics that also involves substitution; thus the word “substitution” is a pun.
AMN form of substitutions

We won’t describe the GSL at this stage, but we will note that there are only 11 basis substitutions in the GSL.

Instead we will use the AMN version of a few substitutions. The AMN syntax makes the substitutions look like a programming language, albeit an abstract one.
### The B-Toolkit

The **B-Toolkit** is a configuration management tool that provides the following facilities:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Facility</th>
</tr>
</thead>
<tbody>
<tr>
<td>introduction of new machines</td>
<td>syntax and type analysis</td>
</tr>
<tr>
<td>animation of specifications</td>
<td>generation of proof obligations</td>
</tr>
<tr>
<td>automatic &amp; interactive proof</td>
<td>introduction of user theories</td>
</tr>
<tr>
<td>markup of machines</td>
<td>maintenance of documents</td>
</tr>
<tr>
<td>generation of code</td>
<td>generation of interfaces</td>
</tr>
<tr>
<td>execution of generated code</td>
<td>generation of base machines</td>
</tr>
<tr>
<td>automatic remakes</td>
<td>browsing of designs &amp; specifications</td>
</tr>
<tr>
<td>hypertext displays of machines</td>
<td>online help</td>
</tr>
</tbody>
</table>
The B-Toolkit interface

The interface of the B-Toolkit is very compact, but has a large number of configurations.

Menu bar the top line contains menus that control the functions of the toolkit.

Environments Below the menu bar is a set of environments: Main, Provers, etc that present different views on the development process.

Machine panel below the Environments is a panel that contains the names of machines or other constructs. This panel contains colour coded buttons that provide access to one of the functions of the toolkit.

Log panel at the bottom is another panel that contains a log of the interactions for the current session.
Introducing a new machine

To introduce a new machine you would select Introduce/New/Machine in the Main environment of the B-Toolkit.

Having introduced the machine, a template will appear in your editor. The machine should be “filled in” and saved.

Then the machine should be committed and analyzed, by selecting the cmt (commit) and anl (analyze) buttons in the Main environment.
A Simple Model

As a first simple model we will take a simple coffee club, but we will do it in two steps.

First we will model a “piggy bank” into which we can feed money and also take money out.

In our model we will use a variable \texttt{piggybank} whose value is a natural number, representing the contents in cents.

Also, we will use operations

\begin{align*}
\text{FeedBank}(\text{amount}) & \quad \text{add } \text{amount} \text{ to the piggy bank;} \\
\text{RobBank}(\text{amount}) & \quad \text{take } \text{amount} \text{ out of the piggy bank;} \\
\text{money} & \leftarrow \text{CashLeft} \quad \text{an enquiry operation that returns, in } \text{money}, \text{ the total of the contents of the piggy bank.}
\end{align*}
Let's step through the specification of a machine that “owns” and manages the piggy bank.

**MACHINE PiggyBank**

- **VARIABLES piggybank**
  - We need a variable, piggybank

- **INVARIANT piggybank : NAT**
  - piggybank is a natural number

- **INITIALISATION piggybank := 0**
  - we will start with the piggybank set to 0

**OPERATIONS**

- **FeedBank(amount) =**
  - we need an operation FeedBank with a single argument amount

- **PRE amount : NAT**
  - amount is a natural number

- **THEN piggybank := piggybank + amount**
  - we set piggybank

- **piggybank + amount**

- **END;**
  - end of preconditioned substitution
RobBank(amount) = now operation RobBank with a single argument amount

PRE amount : NAT

THEN piggybank := piggybank - amount

we set piggybank

piggybank - amount

END; end of substitution

money <-- CashLeft = an operation CashLeft that returns a value,

which is the amount in piggybank

BEGIN money := piggybank END set the return value money to piggybank

END end of machine.
Let’s put that all together and show the machine marked up.

MACHINE PiggyBank
VARIABLES piggybank
INVARIANT piggybank $\in \mathbb{N}$
INITIALISATION piggybank := 0

OPERATIONS
FeedBank ( amount ) $\cong$
PRE amount $\in \mathbb{N}$
THEN piggybank := piggybank + amount
END ;
RobBank ( amount ) $\cong$
PRE amount $\in \mathbb{N}$
THEN piggybank := piggybank − amount
END ;
\[ \text{money} \leftarrow \text{CashLeft} \equiv \]

BEGIN \hspace{1em} \text{money} := \text{piggybank} \hspace{1em} \text{END} \]

END
Machine Structure

MACHINE name set and numeric parameters
CONSTRAINTS predicate
INCLUDES/SEES/USES machine parameters
SETS names
CONSTANTS names
PROPERTIES predicate
VARIABLES names
INVARIANT predicate
INITIALIZATION substitution
OPERATIONS operations
END
...Machine Structure

Note the hierarchy of constraints in the machine structure:

constraints constrain the machine parameters

properties constrain the sets and constants

invariant constrains the variables

Notice that constants and variables are not typed at the point of declaration, but their type must be constrained by the corresponding constraining predicate.
Machine Parameters

Machine parameters enable the specification of generic machines.

The parameters are either:

- **sets** upper case identifiers; denote finite non-empty sets
- **numeric** natural number constants
Operations

The form of an operation is

\[
\text{operation-signature} \triangleq \text{substitution}
\]

An \text{operation-signature} has the form

- \text{name(args)} for an operation that only makes a state substitution, or
- \text{results} \leftarrow \text{name(args)}, where \text{results} is a list of identifiers that represent result values.

In both cases the operation may have no arguments.
**Invariant and Preconditions**

The invariant of a machine is an expression of **safety** or **integrity** conditions. Satisfying the invariant should ensure the integrity or consistency of the information modelled by the state of a machine.
Invariant and Preconditions

The invariant of a machine is an expression of safety or integrity conditions. Satisfying the invariant should ensure the integrity or consistency of the information modelled by the state of a machine.

It is an obligation that each operation maintains the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.
Invariant and Preconditions

The invariant of a machine is an expression of safety or integrity conditions. Satisfying the invariant should ensure the integrity or consistency of the information modelled by the state of a machine.

It is an obligation that each operation maintains the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that would lead to the invariant being broken after the operation.
Invariant and Preconditions

The invariant of a machine is an expression of **safety** or **integrity** conditions. Satisfying the invariant should ensure the integrity or consistency of the information modelled by the state of a machine.

It is an obligation that each operation **maintains** the invariant: it is guaranteed that the invariant is true before an operation is invoked and it is the duty of the operation to ensure that the invariant is true after the operation.

The precondition of an operation should capture all combinations of state and operation arguments before an operation that would lead to the invariant being broken after the operation.

It is desirable that the invariant is as strong as possible, and the precondition is as weak as possible.
**Trivial preconditions**

Although the specification of FeedBank and RobBank use a preconditioned substitution the precondition is used only to carry the type of the parameter to the operation.

This is a trivial precondition.
Problem with the PiggyBank Machine

There is a problem with the PiggyBank machine.

See if you can spot it.

Alternatively, generate the proof obligations and try to discharge them.
Proof obligation generation and proof

Having analyzed a machine, you should routinely generate the proof obligations by selecting the pog (proof obligation generator) button in the Main environment.

Then move to the Provers environment, select the prv (provers) button for the machine, and select AutoProver. If there are unproved obligations then you should either try to discharge the proof obligation using the BToolProver, or at least inspect the obligation to see if it is true.

This should be a routine validation step.
Viewing the proof obligations

Select the **Provers** environment and select the **ppf** (prettyprint proof) button for the machine of interest.

Select the proof obligations from the list.

Select the **Documents** environment, and notice that there is a green `.prf` construct for the chosen machine.

Mark-up the proof obligations by selecting the **dmu** (document markup) button; the view by selecting the **shw** (show) button.
Adding a non-trivial precondition

An attempt to discharge the outstanding proof obligation for the operation RobBank will leave \( \text{amount} \leq \text{piggybank} \) unprovable.

This occurs because the machine invariant says that \( \text{piggybank} \in \mathbb{N} \), that is \( 0 \leq \text{piggybank} \) both before and after an operation.

Thus we need to add the conjunct \( \text{amount} \leq \text{piggybank} \) to the precondition of RobBank.
Total and Partial operations: preconditions

Operations without non-trivial preconditions are total operations: that is the operation may be invoked in any state of the machine, and for any value of the arguments of the operation. Such operations are also called robust.

Operations with non-trivial preconditions are partial operations: that is the operation may not be defined outside of the precondition. Such operations are also called fragile.

A precondition is an assumption, it is not a condition that is going to be tested by the implementer of the operation.

It is the obligation of the invoker of the operation to ensure that the precondition holds. The precondition is the part of the contract that applies to the client of the operation.
Modelling a Coffee Club

We will now model a coffee club with the following facilities for members:

Joining a person can join the club. For the purpose of this simple exercise we identify each member by an element of the set $\text{NAME}$. Of course we want all members to be distinct.

Contributing members can contribute money to the club. This is used to increase the credit of the member, which in turn is used to pay for cups of coffee.

Buy coffee a member can buy a cup of coffee. The price of a cup of coffee is deducted from the members credit.

Credit a member can obtain their current credit balance.
A CoffeeClub machine

The above behaviour is modelled by the machine CoffeeClub. Aspects of this machine are:

- The NAME set is represented by a machine parameter.
- The PiggyBank machine is included into this machine. This embeds the state of PiggyBank into this machine, and gives CoffeeClub access to the operations of PiggyBank.
- The operations RobBank and CashLeft are promoted to the interface of CoffeeClub.
- A constant coffee is used for the cost of a cup of coffee.
- The state of the machine consists of a variable finances, which is a partial function from NAME to N.
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.
MACHINE CoffeeClub ( NAME )
INCLUDES PiggyBank
PROMOTES RobBank, CashLeft
CONSTANTS coffee
PROPERTIES coffee = 120
VARIABLES finances
INVARIANT finances ∈ NAME → N
INITIALISATION finances := {}
THEN finances ( member ) := finances ( member ) + amount \parallel

    FeedBank ( amount )
END;

BuyCoffee ( member ) \equiv

    PRE member ∈ NAME

THEN finances ( member ) := finances ( member ) − coffee
END;

credit ← Credit ( member ) \equiv

    PRE member ∈ NAME

THEN credit := finances ( member )
END

END
Problems with CoffeeClub

The specification given by this machine is not adequate. It is easy to show that the operations can break the invariant.

Generating the proof obligations and attempting to discharge them will illustrate some of the problems.
A Simple Complete Development

Now we will ignore all the components of a machine except the machine header and the operations.

We will concentrate on a simple state-less machine with operations that evaluate simple numerical functions, and pursue a complete development.

We will give a complete development through to code.
Operation to compute Square

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 \texttt{sqr} operation in the \texttt{Square} machine.
MACHINE Square

OPERATIONS

\[
\text{sqr} \leftarrow \text{Sqr}(\text{num}) = \\
\text{PRE num} : \text{NAT} \\
\text{THEN sqr} := \text{num} \times \text{num} \\
\text{END}
\]
A square root operation

Suppose we wish to have an operation $\text{ApproxSqrt}(num)$ 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 $\text{ApproxSqrt}$ operation of the $\text{Square}$ machine.
MACHINE       Square
OPERATIONS
      sqr <-- Sqr(num) =
      PRE  num  :  NAT
              THEN  sqr := num*num
              END;

      sqrt <-- ApproxSqrt(num) =
      PRE  num  :  NAT
      THEN
        ANY  approx
        WHERE  approx  :  NAT  &
                   approx*approx <= num &
                   num  <  (approx+1)*(approx+1)
        THEN  sqrt := approx
        END
      END
      END
END
The use of non-determinism in specification

Notice that the specification of result of ApproxSqrt uses a non-deterministic substitution.

This is a common strategy in writing formal specification.

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.
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 Square

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.
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:

\[ \text{approx} \times \text{approx} \leq \text{num} \quad \text{and} \quad \text{num} < (\text{approx} + 1) \times (\text{approx} + 1) \]

It is relatively easy 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 \( \text{low} + 1 \leq \text{high} \), as shown in the refinement, and implementation machine **SQUAREL**. Notice that this machine uses the services of (IMPORTS) of a new machine **SqrtUtility**.
IMPLEMENTATION SquareI
REFINES Square
IMPORTS SqrtUtility

OPERATIONS

\[ \text{sqr} \leftarrow \text{Sqr}(\text{num}) = \]
BEGIN
\[ \text{sqr} := \text{num} \times \text{num} \]
END;
sqrt <-- ApproxSqrt(num) =
VAR low, high
IN low, high <-- ChooseInitApprox(num);
WHILE low+1 /= high
DO  low, high <-- ChooseBetterApprox(num,low,high)
INVARIANT
   low : NAT & high : NAT &
   low+1 <= high &
   low*low <= num &
   num < high*high
VARIANT
   high-low
END;
sqrt := low
END
MACHINE            SqrtUtility
OPERATIONS
low,high <-- ChooseInitApprox(num) =
   PRE num : NAT
   THEN
      ANY xx, yy
      WHERE xx : NAT & yy : NAT &
         xx + 1 <= yy &
         xx*xx <= num &
         num < yy*yy
      THEN low, high := xx, yy
   END
END;
low, high ← ChooseBetterApprox(num, oldlow, oldhigh) =

PRE num : NAT & oldlow : NAT & oldhigh : NAT &
oldlow*oldlow <= num & num < oldhigh*oldhigh &
oldlow+1 < oldhigh

THEN

ANY xx, yy
WHERE xx : NAT & yy : NAT &
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 \textit{SqrtUtility} we now have to refine it to an implementation.

This is done in three refinement steps as shown in machines \textit{SqrtUtilityR}, \textit{SqrtUtilityR}, and \textit{SqrtUtilityRRI}.
REFINEMENT          SqrtUtilityR
REFINES             SqrtUtility

OPERATIONS

low, high <-- ChooseInitApprox(num) =
    low, high := 0, (num+1)/2 + 1;

low, high <-- ChooseBetterApprox(num, oldlow, oldhigh) =
    ANY mid
    WHERE mid : NAT & oldlow < mid & mid < oldhigh
    THEN SELECT
        mid*mid <= num
        THEN low, high := mid, oldhigh
        WHEN
        num < mid*mid
        THEN low, high := oldlow, mid
    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  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

END
Generating code

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 following frames.
#include "Square.h"

#include "SqrtUtility.h"

void
INI_Square()
{
    INI_SqrtUtility();
}

void
Sqr(_sqr, _num)
int * _sqr, _num;
{
    * _sqr = _num * _num;
}
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;
}
#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;
    }
}
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.
An extended machine

We now extend \texttt{Square} to \texttt{Square1}, which adds an exact square root operation, \texttt{Sqrt}.

The exact square root operation can be implemented by using the approximate square root operation in the \texttt{Square} machine. This follows from the assumption that \texttt{num} has an exact square root.

This example gives a simple illustration of the re-use of specifications/implementations.

Note carefully: the implementation \texttt{Square1I} is dependent on the specification machine \texttt{Square} not on the implementation of that machine. This ensures that we can use any implementation of \texttt{Square}.
MACHINE Square1
EXTENDS Square

OPERATIONS

sqrt <-- Sqrt(num) =
PRE num : NAT &
  #anum.(anum : NAT & anum*anum = num)
THEN
  ANY xx
  WHERE xx : NAT &
    xx*xx = num
  THEN sqrt := xx
END
END

END
IMPLEMENTATION Square1I
REFINES Square1
IMPORTS Square
PROMOTES Sqr, ApproxSqrt

OPERATIONS
  sqrt <-- Sqrt(num) =
    BEGIN sqrt <-- ApproxSqrt(num) END
END
Feasibility

Just because we write a predicate to specify an operation, does not mean that we can implement the specification.

Consider the machine Fermat.
MACHINE Fermat
CONSTANTS EXP

PROPERTIES
  EXP : NAT*NAT --> NAT &
  !xx.(xx:NAT => EXP(xx,0) = 1) &
  !(xx,nn).(xx:NAT & nn:NAT1 =>
    EXP(xx,nn) = xx*EXP(xx,nn-1))

OPERATIONS
  aa,bb,cc <-- Fermat(nn) =

      PRE nn : NAT
      THEN ANY xx,yy,zz
             WHERE xx:NAT & yy:NAT & zz:NAT &
                   EXP(xx,nn)+EXP(yy,nn) = EXP(zz,nn)
      THEN aa,bb,cc := xx,yy,zz

      END

      END

END