System Modelling and Design

Introduction to the B Method and B Toolkit

Revision: 1.1, March 3, 2007

Ken Robinson

March 3, 2007

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

Contents

1 B Mathematical Toolkit

The mathematical toolkit of *the B Method* (*B*) is based on

set theory simple set theory, consisting of aggregates having no ordering and no multiplicity. The only property possessed by a value and a set is membership of the set.

logic first-order predicate calculus. A predicate is a function from variables to Boolean. The first-order calculus allows quantification only over variables, not predicates for example.

Numbers Although B allows opaque types, essentially all numbers in a B development are eventually natural numbers, because real computers consist of binary numerals. B does not contain infinity and all implementable sets are finite. The set of natural numbers ($\mathbb{N}$) is infinite and hence is not implementable.

$\mathbb{N}_1$ is $\mathbb{N} - \{ 0 \}$

2 Set Theory

B uses sets to *model* other mathematical constructs such as: relations, functions, sequences.

The base for modelling with sets are

powerset $\mathbb{P}(S)$, the powerset of the set S, is the set of all subsets of S. $\mathbb{P}(S)$ always contains the empty set.

$\mathbb{P}_1(S)$ is the set of all *non-empty* subsets of S.

product $X \times Y$, the product of X and Y, is the set of ordered pairs with the first element from X and the second from Y, $X \times Y = \{ x, y \mid x \in X \land y \in Y \}$.

2.1 Relations

A *relation* is a set of ordered pairs between the members of two sets.

$X \leftrightarrow Y$ is the set of all many-to-many relations between X and Y.

$X \leftrightarrow Y = \mathbb{P}\, X \times Y$

2.2 Functions

$\nrightarrow$	set of partial functions
$\rightarrow$	set of total functions
$\rightarrowtail\!\!\!\!\!+$	set of partial injection (one-to-one)
$X \rightarrowtail Y$	set of total injection
$\nrightarrow\!\!\!\!\!\twoheadrightarrow$	set of partial surjection (onto)
$\twoheadrightarrow$	set of total surjection
$\rightarrowtail\!\!\!\!\!\twoheadrightarrow$	set of total bijection (one-to-one and onto)

3 Predicate Calculus

3.1 Some Terminology

The following terms will be used frequently:

predicate a predicate is a partial function from variables (state) to Boolean. The predicate is usually expressed as a closed expression, e.g. $amount < balance(customer)$.

satisfies we talk of some variables *satisfying* a predicate. This means that substituting the values of the variables into the predicate will make the predicate *true*.

stronger and weaker if $P \Rightarrow Q$ we frequently say that, "P is *stronger than* Q", although strictly we should say, "P is *at least as strong as* Q". Similarly, we might say "Q is *weaker than* P".

In the same vein we will talk of *strengthening* or *weakening* a predicate. Strengthening a predicate subsets the set of values that satisfy the predicate. Weakening a predicate supersets the set of values that satisfy the predicate.

4 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 is marked up high quality mathematics. For example,

$$account \in ACCOUNT,$$

which has the same meaning as the `ASCII` example.

4.1 Abstract Machines

B uses *Abstract Machines*, which are machines that encapsulate:

state consisting of a set of variables constrained by an *invariant*

operations operations may change the state, *while maintaining the invariant*, and may return a sequence of results.

4.2 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 represent context dependent expressions, which include proper variables.

4.3 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*, that is, an instance of a *class*.
- Importantly, a machine does not behave as a class, although it is possible to model a class.

4.4 Substitutions

The foundation of B operations is a language called the *Generalised Substitution Language* or *GSL*. The GSL notation will not be described in this lecture. The elements of GSL are called *substitutions*, which have a role similar to statements or commands in a conventional programming language.

A substitution is a construct that, in some way, changes the state by substituting values into variables of the state.

> *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.*

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

Substitutions are given a formal semantics that in turn is expressed in in terms of substitution of values; thus the word "substitution" is a pun.

4.5 Abstract Machine Notation

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.

We will use only a few AMN constructs here.

5 The B-Toolkit

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

introduction of new machines	*syntax and type analysis*
animation of specifications	*generation of proof obligations*
automatic & interactive proof	*introduction of user theories*
markup of machines	*maintenance of documents*
generation of code	*generation of interfaces*
execution of generated code	*generation of base machines*
automatic remakes	*browsing of designs & specifications*
hypertext displays of machines	*online help*

5.1 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.

5.2 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.

6 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 using the following operations:

Feedbank(*amount*)	Feed *amount* cents to the piggybank.
RobBank(*amount*)	Rob the piggybank of *amount* cents.
money $\longleftarrow$ **CashLeft**	Query the piggybank to obtain the amount of *money* left in the piggybank.

In order to model the operations we will use a variable *piggybank* whose value is a natural number, representing the contents of the piggybank in cents.

Let's step through the specification of a machine that "owns" and manages the piggy bank.

MACHINE *PiggyBank0*
VARIABLES *piggybank*
INVARIANT $piggybank \in \mathbb{N}$
INITIALISATION $piggybank := 0$

OPERATIONS

FeedBank (*amount*) $\widehat{=}$
  pre $amount \in \mathbb{N}$ **then**
    $piggybank := piggybank + amount$
  end ;
RobBank (*amount*) $\widehat{=}$
  pre $amount \in \mathbb{N}$ **then**
    $piggybank := piggybank - amount$
  end ;
money $\longleftarrow$ **CashLeft** $\widehat{=}$
  begin
    $money := piggybank$
  end
END

6.1 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	
INITIALISATION	substitution	
OPERATIONS	operations	
END		

In general, the clauses of a machine can appear in any order, although machines are stored and marked up according to a canonic structure.

6.2 ... Machine Structure

Note the hierarchy of constraints (clauses consisting of a *predicate* in the machine structure)

constraints constrains the machine *parameters*

properties constrains 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.

6.3 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

6.4 Operations

The form of an operation is

$$\boxed{operation\text{-}signature \mathrel{\widehat{=}} substitution}$$

An $operation\text{-}signature$ has the form

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

In both cases the operation may have no arguments.

6.5 Invariant and Preconditions

The invariant of a machine is an expression of the properties that the state has to satisfy for the operations to correctly model the required behaviour.

The invariant expresses what might be called *safety* or *integrity* conditions.

The initial state must satisfy the invariant, and 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 are required to ensure that the invariant is satisfied after the operation.

It is important that the invariant is as strong as necessary, and the precondition is as weak as possible, but no weaker than necessary.

6.6 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.

6.7 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.

6.8 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.

6.9 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.

6.10 Adding a non-trivial precondition

An attempt to discharge the outstanding proof obligation for the operation `RobBank` will leave $amount \leq piggybank$ unprovable.

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

Thus we need to add the conjunct $amount \leq piggybank$ to the precondition of `RobBank`.

6.11 Towards understanding preconditions

Run the following experiment:

1. run the animator on *PiggyBank* with `RobBank` having a trivial precondition;
2. run the animator on *PiggyBank* with `RobBank` having the non-trivial precondition.

In each case:

1. enable display invariant —the default is not display;
2. run:
   (a) FeedBank(5)
   (b) RobBank(10)
   (c) FeedBank(5)

Describe the results. Notice very carefully that failure of the precondition *does not stop* the operation from going ahead.

6.12 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.

7 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 `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.

The above behaviour is modelled by the machine *CoffeeClub*, initially named *CoffeeClub0*.

7.1 A CoffeeClub machine

MACHINE *CoffeeClub0* (*NAME*)
INCLUDES *PiggyBank*
PROMOTES *RobBank* , *CashLeft*
CONSTANTS *coffee*
PROPERTIES $coffee = 120$
VARIABLES *finances*
INVARIANT $finances \in NAME \rightarrow\!\!\!\!\!\!+ \mathbb{N}$
INITIALISATION $finances := \{\}$

OPERATIONS

NewMember (*member*) $\widehat{=}$
pre $member \in NAME$
then
$finances (member) := 0$
end ;
Contribute (*member* , *amount*) $\widehat{=}$
pre $member \in NAME \wedge amount \in \mathbb{N}$
then
$finances (member) := finances (member) + amount \ \|$
FeedBank (*amount*)
end ;

BuyCoffee (*member*) $\widehat{=}$
pre $member \in NAME$
then
$finances (member) := finances (member) - coffee$
end ;
$credit \longleftarrow$ **Credit** (*member*) $\widehat{=}$

pre *member* $\in$ *NAME*
then *credit* := *finances* (*member*)
end
END

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 $\mathbb{N}$.
- Three operations NewMember, Contribute, BuyCoffee and Credit are used to model the required behaviour.

7.2 Some notes on machine inclusion

Included machine state: the included machine's state is "added" to the state of the including machine.

Referencing included state: the variables in the state of the included machine may be referenced by the including machine.

Modifying the variables of included state: variables of the included machine may be modified by the included machine, *but only* by invoking operations of the included machine.

Export of operations: While operations of the included machine may be used by the including machines, they do not becomes operations of the including machine unless promoted by including machine.

Included machine parameters: if the included machine has parameters they must be instantiated by the including machine.

7.3 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. Run the AutoProver on the proof obligations and examine any undischarged proof obligations.

Animation may help to illustrate where the problems lie.

7.4 Identifying and fixing the problems

The problems are enumerated below:

NewMember this operation has an undesirable functional property: if an existing member —or a new member with the same name as an existing member— with credit runs this operation then their finances are set to 0! The specification alerts the user to this undesirable effect by adding a precondition $member \notin \mathsf{dom}(finances)$, that is, the prospective member is not an existing member.

Contribute the function $finances$ is partial, so the expression used to update the member's finances:

$$finances(member) := finances(member) + amount$$

will be undefined when $member \notin \mathsf{dom}(finances)$. A precondition that $member \in \mathsf{dom}(finances)$ is required.

BuyCoffee In order to buy a coffee, two things are required

1. the person must be a member, otherwise $finances(member)$ will be undefined;
2. a member must have enough finance to cover the price of a cup of coffee. If this is not the case then $finances(member) - coffee$ will not be a natural number, breaking the invariant.

So a precondition of:

$$member \in \mathsf{dom}(finances) \wedge \text{[1ex]}\, finances(member) \geq coffee$$

is required.

Credit $finances(member)$ assumes $member \in \mathsf{dom}(finances)$, so this needs to be added to the precondition.

The following versions of **PiggyBank** and **CoffeeClub** have appropriately strengthened preconditions.

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

OPERATIONS

  FeedBank (*amount*) $\widehat{=}$
    pre *amount* $\in \mathbb{N}$ **then**
      piggybank $:=$ *piggybank* $+$ *amount*
    end ;
  RobBank (*amount*) $\widehat{=}$
    pre *amount* $\in \mathbb{N} \wedge$ *amount* $\leq$ *piggybank* **then**
      piggybank $:=$ *piggybank* $-$ *amount*
    end ;
  money $\longleftarrow$ **CashLeft** $\widehat{=}$
    begin
      money $:=$ *piggybank*
    end
END

MACHINE *CoffeeClub* (*NAME*)
INCLUDES *PiggyBank*
PROMOTES *RobBank* , *CashLeft*
CONSTANTS *coffee*
PROPERTIES $coffee = 120$
VARIABLES *finances*
INVARIANT $finances \in NAME \rightarrowtail\!\!\!\!\!+ \ \mathbb{N}$
INITIALISATION $finances := \{\}$

OPERATIONS

NewMember (*member*) $\widehat{=}$
  pre $member \in NAME \wedge member \notin \mathsf{dom}\ (finances)$
  then
    $finances\ (member) := 0$
  end ;
Contribute (*member* , *amount*) $\widehat{=}$
  pre $member \in NAME \wedge$
    $member \in \mathsf{dom}\ (finances) \wedge amount \in \mathbb{N}$
  then
    $finances\ (member) := finances\ (member) + amount \ \|$
    FeedBank (*amount*)
  end ;

BuyCoffee (*member*) $\widehat{=}$
  pre $member \in NAME \wedge member \in \mathsf{dom}\ (finances) \wedge$
    $finances\ (member) \geq coffee$
  then
    $finances\ (member) := finances\ (member) - coffee$
  end ;
$credit \longleftarrow$ **Credit** (*member*) $\widehat{=}$
  pre $member \in NAME \wedge member \in \mathsf{dom}\ (finances)$
  then $credit := finances\ (member)$
  end
END

8 Specifying a Robust machine

Most of the operations of the **CoffeeClub** machine are fragile, that is the operations have non-trivial preconditions. This means that there are combinations of state and operations arguments for which the operation will fail.

Such operations are not safe to use in an application programmer interface (API) or user interface (UI).

We will build an API machine, **CoffeeClubAPI**, with robust versions of the operations of **CoffeeClub**. These operations will use guards that discharge the precondition of the fragile operation ensuring that it is safe to invoke the fragile operation.

Each operation returns a response that reports whether the operation was successful, or why the precondition failed.

```
MACHINE CoffeeClubAPI ( NAME )
INCLUDES CoffeeClub ( NAME )
SETS RESPONSE = { OK ,
  existing_member ,
  not_a_member ,
  not_enough_finance ,
  not_enough_in_bank }

OPERATIONS

  response ⟵ NewMemberAPI ( member ) ≙
    pre member ∈ NAME then
      if member ∈ dom ( finances )
      then response := existing_member
      else
        response := OK || NewMember ( member )
      end
    end ;
  response ⟵ ContributeAPI ( member , amount ) ≙
    pre member ∈ NAME ∧ amount ∈ ℕ then
      if member ∉ dom ( finances ) then
        response := not_a_member
      else response := OK || Contribute ( member , amount )
      end
    end ;


  response ⟵ BuyCoffeeAPI ( member ) ≙
    pre member ∈ NAME then
      select member ∉ dom ( finances ) then
        response := not_a_member
      when finances ( member ) < coffee then
        response := not_enough_finance
      else response := OK || BuyCoffee ( member )
      end
    end ;
  response , credit ⟵ CreditAPI ( member ) ≙
    pre member ∈ NAME then
      if member ∉ dom ( finances ) then
        response := not_a_member || credit :∈ ℕ
      else response := OK || credit ⟵ Credit ( member )
      end
    end ;
```

$response \longleftarrow$ **RobBankAPI** ($amount$) $\widehat{=}$

  pre $amount \in \mathbb{N}$ **then**

    if $piggybank < amount$ **then**

      $response := not\_enough\_in\_bank$

    else $response := OK \parallel RobBank (amount)$

    end

  end ;

$money \longleftarrow$ **CashLeftAPI** $\widehat{=}$ $money \longleftarrow CashLeft$

END

9 A Question of Identity

The CoffeeClub, in addition to being a very simple model, also exhibits a serious deficiency:

It uses names for the identity of members.

This is clearly inadequate. For example we have a restriction that two people with the same name cannot belong to the club.

In all *real* systems we need to allocate unique identifiers for each member of —for each component of— a system.

Subsequent system models will demonstrate this.