System Modelling and Design
A Whirlwind Introduction to the B Method

Revision: 1.2, March 5, 2003

Ken Robinson

19th August 2005
©Ken Robinson 2005
mailto::k.robinson@unsw.edu.au

Contents

1 A Homily: Formality is inexorable 2
2 Some Terminology 2
3 Notation 3
   3.1 Abstract Machines 3
   3.2 Machine Variables in B 3
   3.3 Object based 3
   3.4 Substitutions 4
   3.5 Abstract Machine Notation 4
4 The B-Toolkit 4
   4.1 The B-Toolkit interface 4
   4.2 Introducing a new machine 5
5 A Simple Model 5
   5.1 Machine Structure 6
   5.2 Machine Structure 6
   5.3 Machine Parameters 6
   5.4 Operations 7
   5.5 Invariant and Preconditions 7
   5.6 Trivial preconditions 7
   5.7 Problem with the PiggyBank Machine 7
   5.8 Proof obligation generation and proof 8
1 A Homily: 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. 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.*

2 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. \( \text{amount} < \text{balance(customer)} \).

**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 superset the set of values that satisfy the predicate.
3 Notation

All components of a B Method (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,

\[ \text{account : ACCOUNT} \]

means the variable account is an element of the set ACCOUNT.

The notation for publication will is marked up high quality mathematics. For example,

\[ \text{account} \in \text{ACCOUNT}, \]

which has the same meaning as the ASCII example.

3.1 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.

3.2 Machine Variables in B

For technical reasons that will not be explained now, machine variables in B must have at least two characters. Thus \( \times x \) 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.

3.3 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.
3.4 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 basic 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.

3.5 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.

4 The B-Toolkit

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

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

4.1 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.

4.2 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.

5 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 ← 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 ∈ \( \mathbb{N} \)

INITIALISATION piggybank := 0

OPERATIONS
FeedBank (amount) ≡

PRE amount ∈ ℤ THEN
piggybank := piggybank + amount
END;

RobBank (amount) ≡
PRE amount ∈ ℤ THEN
piggybank := piggybank - amount
END;

money ← CashLeft ≡
BEGIN
money := piggybank
END

5.1 Machine Structure

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>name</th>
<th>set and numeric parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRAINTS</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INCLUDES/SEES/USES</td>
<td>machine parameters</td>
<td></td>
</tr>
<tr>
<td>SETS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>CONSTANTS</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>VARIABLES</td>
<td>names</td>
<td></td>
</tr>
<tr>
<td>INVARINT</td>
<td>predicate</td>
<td></td>
</tr>
<tr>
<td>INITIALISATION</td>
<td>substitution</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

5.2 …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.

5.3 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
5.4 Operations

The form of an operation is:

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

An operation-signature has the form:

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

In both cases the operation may have no arguments.

5.5 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.

5.6 Trivial preconditions

Although the specification of \text{FeedBank} and \text{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.

5.7 Problem with the PiggyBank Machine

There is a problem with the \text{PiggyBank} machine.

\[ \text{See if you can spot it.} \]

Alternatively, generate the proof obligations and try to discharge them.
5.8 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.*

5.9 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.

5.10 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*.

5.11 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.

8
5.12 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.

6 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.

6.1 A CoffeeClub machine

MACHINE CoffeeClub0 (NAME)
INCLUDES PiggyBank
PROMOTES RobBank, CashLeft
CONSTANTS coffee
PROPERTIES coffee = 120
VARIABLES finances
INVARIANT finances ∈ NAME → ℕ
INITIALISATION finances := {} 

OPERATIONS
NewMember (member) ≜
    PRE member ∈ NAME
    THEN finances (member) := 0
    END ;
Contribute (member, amount) \equiv 
\text{PRE } member \in \text{NAME} \land amount \in \mathbb{N} 
\text{THEN} 
\text{finances} (member) := \text{finances} (member) + amount \parallel 
\text{FeedBank} (amount) 
\text{END} ; 

BuyCoffee (member) \equiv 
\text{PRE } member \in \text{NAME} 
\text{THEN} 
\text{finances} (member) := \text{finances} (member) - \text{coffee} 
\text{END} ; 

credit \leftarrow \text{Credit (member)} \equiv 
\text{PRE } member \in \text{NAME} 
\text{THEN} \text{credit} := \text{finances} (member) 
\text{END} 

\text{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.

6.2 Some notes on machine inclusion

\text{Included machine state:} the included machine’s state is “added” to the state of the including machine.

\text{Referencing included state:} the variables in the state of the included machine may be referenced by the including machine.

\text{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.

\text{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.

\text{Included machine parameters:} if the included machine has parameters they must be instantiated by the including machine.
6.3 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.

6.4 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 \( member \notin \text{dom}(\text{finances}) \), that is, the prospective member is not an existing member.

Contribute the function \( \text{finances} \) is partial, so the expression used to update the member’s finances:

\[
\text{finances}(\text{member}) := \text{finances}(\text{member}) + \text{amount}
\]

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

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

1. the person must be a member, otherwise \( \text{finances}(\text{member}) \) 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 \( \text{finances}(\text{member}) - \text{coffee} \) will not be a natural number, breaking the invariant.

So the following precondition is required:

\[
\text{member} \in \text{dom}(\text{finances}) \land \\
\text{finances}(\text{member}) \geq \text{coffee}
\]

Credit \( \text{finances}(\text{member}) \) assumes \( \text{member} \in \text{dom}(\text{finances}) \), 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 ) \equiv
\text{PRE } amount \in \mathbb{N} \text{ THEN } \text{piggybank} := \text{piggybank} + amount \\
\text{END ;}
RobBank ( amount ) \equiv
\text{PRE } amount \in \mathbb{N} \land amount \leq \text{piggybank} \text{ THEN } \\
\text{piggybank} := \text{piggybank} - amount \\
\text{END ;}
money \leftarrow\text{CashLeft } \equiv
\text{BEGIN} \\
\text{money} := \text{piggybank} \\
\text{END}
\text{END}

MACHINE \text{CoffeeClub ( NAME )}
\text{INCLUDES PiggyBank}
\text{PROMOTES RobBank, CashLeft}
\text{CONSTANTS coffee}
\text{PROPERTIES coffee} = 120
\text{VARIABLES finances}
\text{INVARIANT} \text{finances} \in \text{NAME} \rightarrow \mathbb{N}
\text{INITIALISATION} \text{finances} := \{\}

\text{OPERATIONS}
\text{NewMember ( member ) } \equiv
\text{PRE } \text{member} \in \text{NAME} \land \text{member} \notin \text{dom ( finances )} \\
\text{THEN} \\
\text{finances ( member )} := 0 \\
\text{END ;}
\text{Contribute ( member , amount ) } \equiv
\text{PRE } \text{member} \in \text{NAME} \land \\
\text{member} \in \text{dom ( finances )} \land amount \in \mathbb{N} \\
\text{THEN} \\
\text{finances ( member )} := \text{finances ( member )} + amount || \\
\text{FeedBank ( amount )} \\
\text{END ;}
\text{BuyCoffee ( member ) } \equiv
\text{PRE } \text{member} \in \text{NAME} \land \text{member} \in \text{dom ( finances )} \land \\
\text{finances ( member )} \geq \text{coffee} \\
\text{THEN} \\
\text{finances ( member )} := \text{finances ( member )} - \text{coffee} \\
\text{END ;}
\text{credit} \leftarrow\text{Credit ( member ) } \equiv
\text{PRE } \text{member} \in \text{NAME} \land \text{member} \in \text{dom ( finances )} \\
\text{THEN} \text{credit} := \text{finances ( member )} \\
\text{END}
\text{END}
7 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 ) ≡
\textbf{PRE} \text{ member } \in \text{ NAME } \text{ THEN}
\begin{align*}
\text{IF} \quad \text{ member } \notin \text{ dom ( finances ) } \text{ THEN} \\
\text{response} & := \text{not}\_\text{member} \parallel \text{ credit } \in \mathbb{N} \\
\text{ELSE} & \quad \text{response} := \text{OK} \parallel \text{ credit } \leftarrow \text{Credit} ( \text{ member } ) \\
\text{END}
\end{align*}
\text{END ;}

\text{response} \leftarrow \text{RobBankAPI} ( \text{ amount } ) \triangleq
\begin{align*}
\text{PRE} \quad \text{ amount } \in \mathbb{N} \text{ THEN}
\text{IF} \quad \text{ piggybank } < \text{ amount } \text{ THEN} \\
\text{response} & := \text{not}\_\text{enough}\_\text{in}\_\text{bank} \\
\text{ELSE} & \quad \text{response} := \text{OK} \parallel \text{ RobBank } ( \text{ amount } ) \\
\text{END}
\end{align*}
\text{END ;}

\text{money} \leftarrow \text{CashLeftAPI} \triangleq \text{money} \leftarrow \text{CashLeft}
\text{END}