

School of Computer Science & Engineering —  
UNSW

<http://www.cse.unsw.edu.au/>

**An Introduction to the B Method**  
**A Simple Library**

**Revised 27th March 2001**

Ken Robinson

[mailto::k.robinson@unsw.edu.au](mailto:k.robinson@unsw.edu.au)k.robinson@unsw.edu.au

# Objectives of this Lecture

- to expand the mathematical toolkit to enable modelling of relations
- to develop a case study that will widen our abstract machine repertoire
- to discuss the notions of *fragile* and *robust* operations

# A Simple Library

Let us model a very simple library with the following requirements:

1. Prospective borrowers must register and are given a unique identifier.
2. Only registered users of the library may borrow books.
3. Borrowers may borrow more than one book at the same time.

To simplify the model we will assume that there is a set *BOOK* that contains all the books that could be in the library. This could be thought of as similar to the set of all *ISBN* numbers, but as there will be at most only one copy of any book in the library it's more appropriate to compare it with a shelf number, or a book barcode. When you borrow a book from a library, you don't borrow a book title, you borrow a specific physical book.

The specification we are going to develop is probably not wrong when compared with a real library, nor inappropriate, rather it is *incomplete*.

# Machine Parameters

Machine parameters represent information that is imported from outside the machine. There are two sorts of parameters:

*SET parameters* denoted by upper case identifiers, these represent non-empty finite sets

*numeric constants* denoted by lower case identifiers, these represent natural number constants

We will model the set *BOOK* as a set parameter of the machine.

We will also use a non-zero numeric constant *maxuser* to denote the size of the set of registered users.

Thus the machine header is

MACHINE

SimpleLibrary(BOOK,maxuser)

# CONSTRAINTS Clause

In order to constrain the machine parameter *maxuser* to be non-zero, we add a *CONSTRAINTS* clause to the machine containing the constraint

$$\textit{maxuser} \in \mathbb{N}_1$$

```
CONSTRAINTS          maxuser : NAT1
```

# Representing Registered Users

To represent the set of all possible registered users of the library we will specify a *deferred* set *USER*.

*SETS* *USER*

Such a set is local to the machine, and cannot be seen from outside the machine, although elements of the set may be passed as tokens through machine operations.

Such sets are described as *deferred*, as the exact structure of the set is deferred until the later design (refinement) phases.

We wish the set *USER* to have exactly *maxuser* elements, and we use the *PROPERTIES* clause to constrain the cardinality of the set.

$\text{card}(\text{USER}) = \text{maxuser}$

**SETS**

**USER**

**PROPERTIES**

**card(USER) = maxuser**

# Variables

We need to model

*the set of registered users:* the set of currently registered users.

*books in the library:* the set of books acquired by the library.

*books on the shelf:* the subset of the library books that are currently on the shelves, *ie* not on loan.

*books on loan:* information on what books have been borrowed and who has borrowed them.

# Registered Users

**Variable:**  $users$

**Constraint:**  $users \subseteq USER$

The variable  $USERS$  will “keep track” of the people who have been registered.

**Note:**  $users \subseteq USER$  is equivalent to  $users \in \mathbb{P}(USERS)$ .

# Books in the Library

**Variable:**  $books\_in\_library$

**Constraint:**  $books\_in\_library \subseteq BOOK$

The variable  $books\_in\_library$  will “keep track” of the books acquired by the library.

**Note:**  $books\_in\_library \subseteq BOOK$  is equivalent to  $books\_in\_library \in \mathbb{P}(BOOK)$ .

# Books on the Shelf

**Variable:**  $books\_on\_shelf$

**Constraint:**  $books\_on\_shelf \subseteq books\_in\_library$

Books must be acquired before they may appear on the shelf.

**Note:**  $books\_on\_shelf \subseteq books\_in\_library$  is equivalent to  
 $books\_on\_shelf \in \mathbb{P}(books\_in\_library)$ .

# Books on Loan

**Variable:** *books\_on\_loan*

**Constraint:**  $books\_on\_loan \in books\_in\_library \rightarrow users$

We should note the following:

Each book that is borrowed must be borrowed by exactly one registered user.

A borrower may borrow more than one book.

This indicates a functional relation between *books* and *borrowers*.

# Strengthening the Constraints

The constraints on the variables are not yet strong enough.

Clearly, a book that is borrowed may not also be on the shelf in the library.

Also, a book acquired by the library is either *on the shelf* or *on loan*. At least in our simple library.

Both of these can be expressed by saying that *books\_on\_shelf* must be exactly the difference between *books\_in\_library* and the *domain* of the *books\_on\_loan* function .

Thus we need the following constraint

$$books\_on\_shelf = books\_in\_library - \text{dom}(books\_on\_loan)$$

Thus we obtain the following header for the **SimpleLibrary** machine.

```
MACHINE          SimpleLibrary(BOOK,maxuser)
CONSTRAINTS     maxuser : NAT1
SETS            USER
PROPERTIES      card(USER) = maxuser
VARIABLES
  users, books_in_library, books_on_shelf, books_on_loan
INVARIANT
  users <: USER &
  books_in_library <: BOOK &
  books_on_shelf <: books_in_library &
  books_on_loan : books_in_library +-> users &
  books_on_shelf = books_in_library - dom(books_on_loan)
```

# Initialisation

An appropriate initialisation of the variables that will satisfy the machine state invariant is to set all the variables to the empty set.

## INITIALISATION

```
users,  
books_in_library,  
books_on_shelf,  
books_on_loan := {}, {}, {}, {}
```

# Adding a Book to the Library

We want to model an operation *AddBook(book)* that adds book to the libraries collection, the set *books\_in\_library*.

*book* must be a new book —one that is not already contained in the libraries collection— to the library collection.

We will assume that at the same time as we add the book to the library collection we add it to the library shelves.

```
AddBook(book) =
```

```
  PRE book : BOOK & book /: books_in_library
```

```
  THEN books_in_library := books_in_library \/ {book} ||
```

```
       books_on_shelf := books_on_shelf \/ {book}
```

```
END
```

# Registering a New User

We want to model an operation *NewUser* that will register a new user, provided that we have not exhausted our set of user tokens.

This operation will return the user token that must be used when borrowing a book.

In modelling this operation we choose any user token that has not yet been allocated. We then add this to the set *USER* and return the value to the invoker of the operation.

```
newuser <-- NewUser =  
  PRE users /= USER  
  THEN  
    ANY user  
    WHERE user : (USER - users)  
    THEN users := users \ {user} ||  
      newuser := user  
    END  
  END
```

# Borrowing a Book

We want to model a borrow operation,  $Borrow(user, book)$ , that involves a borrower,  $USER$ , and a book to be borrowed,  $book$ .

The  $USER$  must be a registered user.

The  $book$  must be available for loan, *ie* it must be a book that is currently on the shelf.

After the operation, the book is no longer on the shelf of the library, and the state records the relation between the book and the borrower.

Notice that we replace  $books\_on\_loan$  by the union of two functions, and this must be a function. In general, the union of two functions is not a function. Why?

Why is it in this case?

```
Borrow(user,book) =  
  PRE user : users & book : books_on_shelf  
  THEN books_on_shelf := books_on_shelf - {book} ||  
        books_on_loan := books_on_loan \ / {book |-> user}  
  END
```

# Returning a Book

We want to model a return operation,  $\textit{Return}(\textit{book})$ , that returns a book to the library.

The book to be returned must currently be on loan.

We do not care who returns the book, so only a book appears as an argument to the operation.

Note the use of *domain subtraction* to remove all maplets  $\textit{book} \mapsto \textit{anyone}$  that records the borrowing of the book by *anyone*. Since this is a function there will be at most one such maplet. In this case there will be exactly one. Why?

```
Return(book) =
```

```
  PRE book : dom(books_on_loan)
```

```
  THEN books_on_shelf := books_on_shelf \ / {book} ||
```

```
       books_on_loan := {book} << | books_on_loan
```

```
END
```

# Who's Borrowed this Book?

We will model an enquiry operation that reports the borrower of a book.

Clearly, this operation has to assume that the book has been borrowed.

An enquiry operation is an operation that does not change the state of the machine.

```
user <-- Borrowed(book) =  
  PRE book : dom(books_on_loan)  
  THEN user := books_on_loan(book)  
  END
```

# A Note on Constraining Predicates

Predicates constraining a set, constant, variable, or operation argument must contain a *constraining predicate*. A constraining predicate allows the determination of the basic set to which the entity belongs. This is required by the type analyzer in the BToolkit. It's also required by mere mortals reading a specification.

Constraining predicates have the form:

$x \in S, x \subseteq S, x \subset S, \text{ or } x = E$  where  $x \setminus S$  and  $x \setminus E$ <sup>a</sup>

Consider the *book* argument to any operation.

We could write  $book \in BOOK \wedge book \in books\_on\_shelf$

but we could also write simply  $book \in books\_on\_shelf$

since  $books\_on\_shelf \subseteq books\_in\_library \subseteq BOOK$ .

We cannot write simply  $book \notin books\_on\_shelf$ , we must write

$book \in BOOK \wedge book \notin books\_on\_shelf$ .

---

<sup>a</sup> $x \setminus E$  ( $x$  “not free in”  $E$ ) means that any instances of  $x$  in  $E$  are bound by quantifiers such as  $\exists x$ , or  $\forall x$ .

# Check Proof Obligations

Having completed the machine:

*Analyze the machine* until the machine is syntactically and type correct.

*Generate the proof obligations* This step after analysis should be standard.

*Run the AutoProver* to determine any “residual” proof obligations.

*Run the BToolProver* on the remaining proof obligations —possibly not doing complete proofs— to determine if there are any proof obligations that are *unprovable*. These will be the consequence of inconsistencies, or incompleteness in the specification.

In the case of the **SimpleLibrary** machine there is one undischarged *Context* proof obligation

$$cst(\text{SimpleLibrary}) \Rightarrow \exists \text{USER}. (\text{card}(\text{USER}) = \text{maxuser} \wedge \text{card}(\text{USER}) \in \mathbb{N}_1)$$

A very simple rewrite rule,  $(P \ \& \ Q) == (Q \ \& \ P)$  leads to a proof!

# Animation

Try animating the **SimpleLibrary** machine.

Instantiate *maxuser* to something small, say 10. It is always wise to instantiate constants to something small.

Don't bother instantiating any of the deferred sets.

Populate the books and users of the library by using symbolic names such as BigBlueBook, LittleRedBook for books and john, jill for users. All deferred sets are in reality sets of natural numbers and the names suggested above are natural number constants.

# *books\_on\_shelf*: A Dependent Variable

Given

$$books\_in\_library \subseteq BOOK \wedge$$

$$books\_on\_loan \in books\_in\_library \rightarrow users \wedge$$

$$books\_on\_shelf = books\_in\_library - \text{dom}(books\_on\_loan)$$

we can derive

$$books\_on\_shelf \subseteq books\_in\_library$$

This shows that *books\_on\_shelf* is a *dependent* variable.

There is nothing wrong with having a dependent variable, but we could remove *books\_on\_shelf* as a variable, and leave the concept of *books\_on\_shelf* by inserting a *definitions* clause:

$$books\_on\_shelf \hat{=} books\_in\_library - \text{dom}(books\_on\_loan)$$

`books_on_shelf == books_in_library - dom(books_on_loan)` in  
ASCII.

# Fragile and Robust Operations

Operations with non-trivial preconditions are *fragile* and must be used in contexts in which the preconditions can be proved to be satisfied.

Invoking a fragile operation in a context in which the preconditions are not known to be satisfied will lead to unpredictable results.

It should be observed that developments conducted completely within the B Method will entail proving that all preconditions are satisfied.

# Adding an Interface to Simplelibrary

We can develop a machine with robust operations that could be used as an application programmer interface (API).

A robust operation is an operation that has only a trivial precondition. Such an operation may be invoked in any state of the machine and for any argument values.

The standard technique for converting a fragile operation to a robust operation is to add a response value to the return list of the operation. The response value indicates whether the operation has been successful.

It should be noted that all return values must have appropriate values, although the validity of those values will depend on the response value.

The following slides show an API version of **SimpleLibrary**.

MACHINE *SimpleLibraryAPI* ( *BOOK* , *maxuser* )

CONSTRAINTS *maxuser*  $\in \mathbb{N}_1$

INCLUDES *SimpleLibrary* ( *BOOK* , *maxuser* )

SETS

*RESPONSE* = { *OK* ,  
    *BookInLibrary* ,  
    *NoNewUsers* ,  
    *NotRegisteredUser* ,  
    *BookNotForLoan* ,  
    *BookNotOnLoan* }

OPERATIONS

```
response  $\leftarrow$  AddBookR ( book )  $\hat{=}$   
  PRE book  $\in$  BOOK  
  THEN IF book  $\notin$  books_in_library THEN  
    AddBook ( book ) ||  
    response := OK  
  ELSE response := BookInLibrary  
  END  
END ;
```

```
response , newuser ← NewUserR ≐  
IF users ≠ USER  
THEN newuser ← NewUser ||  
    response := OK  
ELSE newuser ∈ USER ||  
    response := NoNewUsers  
END ;
```

```
response  $\leftarrow$  BorrowR ( user , book )  $\hat{=}$   
PRE user  $\in$  USER  $\wedge$  book  $\in$  BOOK  
THEN  
  SELECT  
    user  $\notin$  users THEN response := NotRegisteredUser  
  WHEN  
    book  $\notin$  books_on_shelf THEN response := BookNotF  
  ELSE  
    Borrow ( user , book ) ||  
    response := OK  
  END  
END ;
```

```
response  $\leftarrow$  ReturnR ( book )  $\hat{=}$   
  PRE book  $\in$  BOOK  
  THEN IF book  $\in$  dom ( books_on_loan ) THEN  
    Return ( book ) ||  
    response := OK  
  ELSE response := BookNotOnLoan  
  END  
END ;
```

```
response , user  $\leftarrow$  BorrowedR ( book )  $\hat{=}$   
PRE book  $\in$  BOOK  
THEN IF book  $\in$  dom ( books_on_loan ) THEN  
    response := OK ||  
    user  $\leftarrow$  Borrowed ( book )  
ELSE response := BookNotOnLoan ||  
    user  $\in$  USER  
END  
END  
END
```

# Use of *SELECT* Substitution

Notice the use of a *SELECT* substitution within the *BorrowR* operation.

This achieves *non-deterministic* choice in the case that both guards  $user \notin users$  and  $book \notin books\_on\_shelf$  are true, *ie* a person who is not a registered user is attempting to borrow a book that is not available for loan.

The specification says that either *NotRegisteredUser* or *BookNotForLoan* are valid responses and we don't care which is chosen.

# Enumerated Sets

Notice the use of an enumerated set for the response values.

Enumerated sets are sets of natural numbers with the symbolic values being mapped onto 0, 1 etc.

# Machine Inclusion

Notice that **SimpleLibraryAPI** includes the **SimpleLibrary** machine.

At the point where the machine is included, any parameters of the included machine must be instantiated. In this case the parameters of **SimpleLibrary** are instantiated to the parameters of **SimpleLibraryAPI**.

The including machine inherits the state and operations of the included machine.

The state variables of the included machine may be referenced in predicates, but the values may be changed only by using operations of the included machine.

Thus we have *partial hiding* of the state.

Notice that when a machine operation is used, the syntax is the same as that used for the specification of the operation. See, for example, the use of *Borrowed* within *BorrowedR*.

# Non-deterministic Choice from a Set

*ASCII*  $x :: S$

*Publication*  $x \in S$

Arbitrarily choose a value from the set  $S$ , and substitute in the variable  $v$ .

This substitution is used in `NewUserR`, the robust version of the operation `NewUser`, in the event that it is not possible to choose a new user token.

# Proof Obligations of the Robust Operations

When the proof obligations for **SimpleLibraryAPI** are generated it will be noted that there are no proof obligations for the operations.

This is a consequence of the guards on the *IF THEN ELSE* substitutions satisfying the preconditions of the referenced fragile operations from **SimpleLibrary**.

If you wish you can reset the machine, and choose *generate all proof obligations* in the **Options/Provers** menu, and then regenerate the proof obligations. You will now get proof obligations for the operations. They are trivial, but display the proof obligations thrown up by the preconditions of the fragile operations.