

FIT3013 Formal Specification in Software Engineering
Semester 1 2009
Event-B Tutorial Exercises 3
Machines

The objective of this set of tutorial exercises is to develop specifications as Event-B machines. In all cases the resulting machines should be introduced into the Event-B Rodin Toolkit, analyzed, proof obligations generated, the autoprover run and any remaining undischarged proof obligations inspected carefully.

Not all of the following exercises will be discussed in tutorial periods. Selected solutions will be made available.

1. **A simple bank** Produce a model of a very simple bank with the following requirements. Follow the English very carefully.

accounts the bank customers are represented by accounts. Having obtained an account a customer may use the other operations supported by the bank.

balance the bank needs to maintain a balance for all accounts.

curaccount the current account number being processed by the system.

curbalance the current balance being processed by the system.

NewAccount an event by which a customer obtains an account identifier. Account identifiers are allocated from a pool (set) of identifiers maintained by the bank. The new account number should be saved as the current account number *curaccount*.

Deposit an operation to add an *amount* to an *account* balance.

Withdraw an operation withdraw an *amount* from an *account*. Customers cannot withdraw more than the balance in their account.

Balance an enquiry event for a customer to obtain the *balance* in their *account*. The balance should be saved as the current balance *curbalance*.

Holdings an operation that returns the total sum of all the balances held by the bank. Clearly this should be a privileged operation not able to be run by anyone, but we will keep things simple. Return its value in *curbalance*.

Note: the balance and all other money amounts can be represented by natural numbers, recording cents, rather than dollars and cents.

Note that the event *Holdings* requires a **sum** operation. How do you specify this in Event-B? (B has an explicit sigma Σ operator.)

Answer:

```

MACHINE SimpleBank_M0
SEES SimpleBank_C0
VARIABLES
  accounts
  balance
  curaccount
  curbalance
INVARIANTS
  inv1 :  $accounts \subseteq ACCOUNTS$ 
  inv2 :  $balance \in accounts \rightarrow \mathbb{N}$ 
  inv3 :  $curaccount \in ACCOUNTS$ 
  inv4 :  $curbalance \in \mathbb{N}$ 
EVENTS
Initialisation
  begin
    act1 :  $accounts := \emptyset$ 
    act2 :  $balance := \emptyset$ 
    act3 :  $curaccount \in ACCOUNTS$ 
    act4 :  $curbalance := 0$ 
  end
NewAccount  $\hat{=}$ 
  when
    grd1 :  $ACCOUNTS \setminus accounts \neq \emptyset$ 
  then
    act1 :  $curaccount \in ACCOUNTS \setminus accounts$ 
    act2 :  $accounts := accounts \cup \{curaccount\}$ 
  end
Deposit  $\hat{=}$ 
  any
    amount
    account
  where
    grd1 :  $amount \in \mathbb{N}$ 
    grd2 :  $account \in accounts$ 
  then
    act1 :  $balance(account) := balance(account) + amount$ 
  end

```

Answer:

```

Withdraw  $\hat{=}$ 
  any
    amount
    account
  where
    grd1 :  $amount \in \mathbb{N}$ 
    grd2 :  $account \in accounts$ 
  then
    act1 :  $balance(account) := balance(account) - amount$ 
  end
Balance  $\hat{=}$ 
  any
    account
  where
    grd1 :  $account \in accounts$ 
  then
    act1 :  $curaccount := account$ 
    act2 :  $curbalance := balance(account)$ 
  end
Holdings  $\hat{=}$ 
Status anticipated
  begin
    act1 :  $curbalance := \in \mathbb{N}$ 
  end
END

```

The way to do repeated addition is to use the **iteration** operator, and to form its **closure** over all element of a set. The set we want closure over is the set of all *accounts*, and the values we want to sum are the *balances* for each account.

So we look for a relation that we can iterate over, that picks off an element of the set *accounts* at each iteration, computing the balance, and summing that. The iteration stops when there are no elements left in the set.

This implies a relation (function) of the form $k \in \mathbb{P}(S \times \mathbb{N})$, where the two arguments are the set of accounts and the accumulating balance.

$$\begin{aligned}
 k &= \lambda q \mapsto n. q \subseteq S \wedge n \in \mathbb{N} \wedge \exists x. r \in S \times \mathbb{N} \wedge \\
 &\quad (q = \phi \Rightarrow x = \phi \mapsto n) \vee \\
 &\quad (q \neq \phi \Rightarrow \exists t, m. t \in s \wedge m \in \mathbb{N} \wedge m = f(t) \wedge x = (S - t \mapsto n + m)) \\
 &\quad | \quad x
 \end{aligned}$$

If f is replaced with *balance*, then the sum expression is

$$sum = k^*(accounts, 0)$$

2. **Traffic lights** Exercises in the use of preconditions to ensure safety. In a real system we would need more complex protocols and reactive systems. For example, when a light is switched on, it is necessary to receive feedback on whether the light did switch on. But we will not worry about any of that here.

- (a) **A simple four way intersection** Consider a four-way intersection with traffic light control in the two directions *EastWest* and *NorthSouth*. The lights in the *East* and *West* directions are identical; similarly with *North* and *South*.

Specify a machine consisting of the following:

- i. A *SETS* component containing two enumerated sets

$$\begin{aligned} DIRECTION &= \{NorthSouth, EastWest\}; \\ LIGHT &= \{Red, Green, Amber\} \end{aligned}$$
- ii. A *VARIABLES* component containing *lights*, which will model the relation between directions and the lights showing in those directions.
- iii. An *INVARIANT* component containing

$$lights \in DIRECTION \rightarrow LIGHT$$

and an expression of the safety condition

when a *Green* or *Amber* light is showing in any direction then *Red* must be showing in the other direction.

The light sequencing is: *Red, Green, Amber, Red, ...*

- iv. An appropriate *INITIALISATION* substitution.
- v. *OPERATIONS* containing:
 - A. ToRed(*dir*): an operation that sets the light to *Red* in direction *dir*; similarly
 - B. ToGreen(*dir*) and
 - C. ToAmber(*dir*),

Each operation must have appropriate preconditions that ensure that the safety conditions will be met. The proof obligations will contain checks of the consistency of the machine invariant and the operations.

- (b) **An intersection with right-turn lights** Suppose that we now wish to add *right-turn* lights in the *North* and *South* directions.

Change *DIRECTION* to contain $\{North, NorthRight, South, SouthRight, EastWest\}$, recognizing that we now need to split the *NorthSouth* direction into four directions.

Change the rest of the machine.

- (c) **A generic intersection with multiple undetermined directions** Change *DIRECTION* to a deferred set, ie no identified specific directions.

Add

- i. a *CONSTANTS* section containing:

$$conflicts$$
- ii. a *PROPERTIES* section containing:

$$conflicts \in DIRECTION \leftrightarrow DIRECTION$$

that is, *conflicts* relates a *direction* to all other *directions* that conflict with that *direction*. See box on relations.

Add more predicates that identify properties of *conflicts*.

3. **Interpreting preconditions** The machine **Lift.mch** in *BDEMO/DEMO2_LIFT* specifies the behaviour of a lift. The machine is not documented and it can be difficult to understand the intention of the preconditions. The following exercises are suggested:

- (a) Obtain the *BDEMOS* using the Unix command `installBdemos`, if you don't already have them.
- (b) Move to the directory *BDEMO/DEMO2_LIFT* and run the B-Toolkit (`btkit`).
- (c) Introduce **Lift.mch** from *SRC*, and analyze the machine.
- (d) Inspect the machine in your editor, and animate the machine.
- (e) Document the machine in English by describing the machine invariant and the preconditions in behavioural terms, not mathematical terms. For example you might explain the invariant predicate `dor[mov] <: {c1o}` by saying, *Doors of moving lifts are always closed*.

4. Examples from Wordsworth

If you haven't already got them, install the Wordsworth demos by running `installBdemos` in an appropriate directory. This will install the demos from the floppy disc provided with the book. They have been edited to make them compatible with the current releases for the B-Toolkit.

- (a) Exercises 3.1 and 3.2. Extend the *CMA*, Class Manager Assistant, machine by providing operations for
 - i. recording that a student has done the exercises.
 - ii. recording that a student has left the class.
- (b) Exercise 3.4. Define a machine to represent an array. The machine should have two parameters, a natural number to fix the upper limit of the indexes (the lower limit is 1) and a set for the type of values to be stored in the array. Use a partial function from indexes to values for the model. There should be operations to read the value at a given index, to store a value at a given index, and to exchange the values at two indexes. The initialization and the operations to read values from the array and to exchange values in the array should warn the user of the array not to look at the values at an index that has not previously had a value stored.

Relations and Functions

Given sets X and Y , $X \leftrightarrow Y$ is the set of all possible relations between elements of X and Y . A relation is a set of pairs relating—in this case—elements of X to elements in Y . Notice that relations are directional: from X to Y ; even though the symbol \leftrightarrow might suggest bidirectional mappings. Notice that $X \leftrightarrow Y$ is identical to $\mathbb{P}(X \times Y)$.

Relations are, in general, *many-to-many* expressing the possibility of one element of X mapping to many elements of Y , and many elements X mapping to the same element of Y .

An example of a relation is $factors \in \mathbb{N}_1 \leftrightarrow \mathbb{N}_1$ the set of all mappings from non-zero natural numbers to their factors.

$$factors = \{1 \mapsto 1, 2 \mapsto 1, 2 \mapsto 2, 3 \mapsto 1, 3 \mapsto 3, 4 \mapsto 1, 4 \mapsto 2, 4 \mapsto 4, \dots \}$$

If $r \in X \leftrightarrow Y$ then r^{-1} is the *inverse* of r —written $r\sim$ in ASCII—in which all the maplets are reversed.

In many situation we want a relation that is at most *many-to-one*, that is a relation where elements of the *domain* map to at most one element in the range. This is called a *functional* mapping.

$X \twoheadrightarrow Y$ is the set of all *partial* functions mapping from X to Y . A function is *partial* if not all elements of the domain set, X , are required to have mappings.

$X \rightarrow Y$ is the set of all *total* functions mapping from X to Y . A *total* function must map every element of X to some element of Y .

$X \mapsto Y$ is the set of all *partial injective* functions from X to Y . An *injective* function is *one-to-one* in which at most one element of X maps to any particular element of Y . Notice that, if $f \in X \twoheadrightarrow Y$ then, in general, f^{-1} is not a function, but if $f \in X \mapsto Y$ then f^{-1} is a function.

For functions we have the notation of *function application*, $f(x)$. If $f \in X \twoheadrightarrow Y$ then $\exists(x, y).(x \in X \wedge y \in Y \wedge x \mapsto y \in f \Rightarrow f(x) = y)$. What do we have for relations?

Since general relations are many-to-many, function application cannot work. Instead we have *relational image*. If $r \in X \leftrightarrow Y$ and $s \subseteq X$, then $r[s]$, the *image of s under r* , is the subset of Y to which the elements of s are mapped under r .

$$factors[\{6\}] = \{1, 2, 3, 6\}$$

Note: $r[s] = \text{ran}(s \triangleleft r)$