

School of Computer Science & Engineering —
UNSW

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

An Introduction to the B Method

A Simple ATM

Beyond Specification

Revision: 1.2, May 31, 2002

©Ken Robinson

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

Objectives of this Lecture

- to introduce the concepts of refinement and implementation;
- to introduce detail through design (refinement & implementations);
- to illustrate model decomposition using multiple machines;
- to illustrate the use —now seen for the first time— of the results returned by operations;
- to illustrate all the above using a simple ATM example.

A Simplistic Model of an ATM

We want to produce a model of an ATM. The model will be kept reasonably simple, but also reasonably realistic.

Required ATM operations:

- an operation to insert the card and provide a password;
- an operation to withdraw money;
- an operation to deposit money;
- an operation to get the account balance.

Additionally, we will need an operation, that is really a bank operation, to open an account so that the above operations can be executed.

The initial attempt might be as shown in the *ATMO* machine. This is likely to be the type of specification produced by someone familiar only with machine level development.

ATM0

A simplistic and incorrect model of an ATM

MACHINE *ATM0*

SETS

ACCOUNT ;

PASSWORD ;

RESPONSE = { *OK* , *REFUSED* }

VARIABLES

accounts ,
password ,
balance ,
loggedin

INVARIANT

$accounts \subseteq ACCOUNT \wedge$
 $password \in accounts \rightarrow PASSWORD \wedge$
 $balance \in accounts \rightarrow \mathbb{N} \wedge$
 $loggedin \in seq(accounts) \wedge size(loggedin) \leq 1$

INITIALISATION $accounts, password, balance, loggedin := \{\}, \{\},$

OPERATIONS

```
response ← InsertCard ( account , pass ) ≐  
PRE account ∈ ACCOUNT ∧ pass ∈ PASSWORD  
THEN IF loggedin = [] ∧  
      account ∈ accounts ∧  
      pass = password ( account )  
THEN response := OK ||  
      loggedin := [ account ]  
ELSE response := REFUSED  
END  
END ;
```

response , *bal* \leftarrow Withdraw (*amount*) $\hat{=}$

PRE *amount* $\in \mathbb{N}$ THEN

IF *loggedin* $\neq []$ THEN

LET *account* BE *account* = first (*loggedin*) IN

IF *amount* \leq *balance* (*account*) THEN

balance (*account*) := *balance* (*account*) - *amo*

bal := *balance* (*account*) - *amount* ||

response := OK

ELSE *bal* $\in \mathbb{N}$ || *response* := REFUSED

END

END

ELSE *bal* $\in \mathbb{N}$ || *response* := REFUSED

END

END ;

response \leftarrow Deposit (*amount*) $\hat{=}$

PRE *amount* $\in \mathbb{N}$ THEN

IF *loggedin* $\neq []$ THEN

LET *account* BE *account* = first (*loggedin*) IN

balance (*account*) := *balance* (*account*) + *amount*

response := *OK*

END

ELSE *response* := *REFUSED*

END

END ;

```
response , bal  $\leftarrow$  Balance  $\hat{=}$   
IF loggedin  $\neq$  [ ] THEN  
  LET account BE account = first ( loggedin ) IN  
    bal := balance ( account ) ||  
    response := OK  
  END  
ELSE bal : $\in$   $\mathbb{N}$  ||  
  response := REFUSED  
END ;
```

```

response , account  $\leftarrow$  OpenAccount ( pass )  $\hat{=}$ 
  PRE pass  $\in$  PASSWORD THEN
    IF accounts  $\neq$  ACCOUNT
      THEN ANY acc WHERE acc  $\in$  ACCOUNT – accounts
        THEN accounts := accounts  $\cup$  { acc } ||
          balance ( acc ) := 0 ||
          password ( acc ) := pass ||
          account := acc ||
          response := OK
        END
      ELSE account  $\in$  ACCOUNT ||
        response := REFUSED
      END
    END ;

```

WithdrawCard $\hat{=}$ *loggedin* := []

END

Improving the Model

This *ATMO* model is seriously ill-conceived. It puts bank-like state inside the ATM. This is obviously wrong: ATMs have no banking knowledge, they are simply boxes in the wall that interact with a card user and communicate with a remote banking system.

We will attempt to build a more realistic model that separates the ATM and the remote banking system.

First, we need to specify the context information that is common to both the ATM and the remote banking system. This is shown in *BankContext* and *Password*. It's split into two machines because the account, service card and response modelling “belongs” to the banking system, but the modelling of passwords is global.

We are now modelling a service card, distinct from the account. We assume that the service card can be represented by information that is generated from the account, and that the account can be extracted from the service card.

Looking ahead to secure implementation to numeric operations we use the finite

SCALAR type from the *Scalar_TYPE* machine instead of \mathbb{N} .

Bank Context machine

MACHINE *BankContext* (*maxaccount*)

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

SETS

The set of account ids

ACCOUNT ;

The set of services cards

SCARD ;

The set of responses

RESPONSE = { *OK* , *REFUSED* }

CONSTANTS

A constant, injective function, that maps accounts to service cards

GENSCARD

PROPERTIES

$\text{card} (\textit{ACCOUNT}) = \textit{maxaccount} \wedge$

$\text{card} (\textit{SCARD}) \geq \textit{maxaccount} \wedge$

$\textit{GENSCARD} \in \textit{ACCOUNT} \mapsto \textit{SCARD}$

OPERATIONS

The operations implement the *GENSCARD* function and its inverse.

```
scard ← GenScard ( account ) ≐
  PRE account ∈ ACCOUNT
  THEN scard := GENSCARD ( account )
  END ;
account ← ExtractAccount ( scard ) ≐
  PRE scard ∈ SCARD
  THEN account := GENSCARD-1 ( scard )
  END
END
```

Password machine

```
MACHINE Password  
SETS PASSWORD  
END
```

ATM

This machine models the observed behaviour of an ATM. It does not attempt to model the reasons for the behaviour.

Note the extensive use of non-determinism.

MACHINE *ATM*

SEES *BankContext* , *Password* , *Scalar_TYPE*

VARIABLES *loggedin*

INVARIANT $loggedin \subseteq ACCOUNT \wedge \text{card}(loggedin) \leq 1$

INITIALISATION $loggedin := \{\}$

OPERATIONS

$response \leftarrow \text{InsertCard} (scard , pass) \hat{=}$

PRE $scard \in SCARD \wedge pass \in PASSWORD$ THEN

IF $loggedin = \{\}$ THEN

CHOICE

$response := OK \parallel$

$loggedin := \{ GENSCARD^{-1} (scard) \}$

OR

$response := REFUSED$

END

ELSE $response := REFUSED$

END

END ;

```
response , money , bal  $\leftarrow$  ATMWithdraw ( amount )  $\hat{=}$   
PRE amount  $\in$  SCALAR THEN  
  IF loggedin  $\neq$  {} THEN  
    CHOICE  
      response := OK || money := amount  
    OR  
      response := REFUSED || money := 0  
    END  
  ELSE response := REFUSED || money := 0  
  END || bal  $\in$  SCALAR  
END ;
```

```
response , bal  $\leftarrow$  ATMDeposit ( amount )  $\hat{=}$   
  PRE amount  $\in$  SCALAR THEN  
    IF loggedin  $\neq$  {} THEN  
      response  $:\in$  RESPONSE  
    ELSE response := REFUSED  
    END || bal  $:\in$  SCALAR  
  END ;
```

response , *bal* \leftarrow ATMBalance $\hat{=}$

BEGIN

IF *loggedin* \neq {} THEN

response $:\in$ *RESPONSE*

ELSE *response* $:=$ *REFUSED*

END || *bal* $:\in$ *SCALAR*

END ;

WithdrawCard $\hat{=}$ *loggedin* := { }

END

Bank

This machine provides a simple model of a remote banking system. The machine operations represent transactions and queries that can be requested by a client system.

```
MACHINE Bank ( maxaccount )  
CONSTRAINTS maxaccount  $\in$  1 .. MaxScalar  
SEES Password , Scalar_TYPE  
INCLUDES BankContext ( maxaccount )  
PROMOTES ExtractAccount
```

OPERATIONS

$response, account, scard \leftarrow \text{OpenAccount}(pass) \hat{=}$

PRE $pass \in \text{PASSWORD}$ THEN

$account : \in \text{ACCOUNT} \parallel scard : \in \text{SCARD} \parallel response :$

END ;

$response \leftarrow \text{CheckPassword} (account , pass) \hat{=}$

PRE $account \in ACCOUNT \wedge pass \in PASSWORD$ THEN

$response \in RESPONSE$

END ;

response , *bal* \longleftarrow Withdraw (*account* , *amount*) $\hat{=}$
PRE *account* \in ACCOUNT \wedge *amount* \in SCALAR THEN
 bal : \in SCALAR || *response* : \in RESPONSE
END ;

response \leftarrow Deposit (*account* , *amount*) $\hat{=}$

PRE *account* \in *ACCOUNT* \wedge *amount* \in *SCALAR* THEN

response $:\in$ *RESPONSE*

END ;

response , *bal* \longleftarrow Balance (*account*) $\hat{=}$

PRE *account* \in *ACCOUNT* THEN

bal $:\in$ *SCALAR* || *response* $:\in$ *RESPONSE*

END

END

A Note on structure

Notice that the *ATM* machine SEES both *BankContext* and *Password*, while the *Bank* machine SEES *Password*, but INCLUDES *BankContext*.

This reflects “ownership”. Neither machine owns *Password*, but the *Bank* machine owns the contents of *BankContext*.

The consequence is that further along the development, *BankContext* will be implemented within the implementation of *Bank*, and *Password* will be implemented independently.

Refinement

Refinement is the name given to *design* in **B**. In general refinement is conducted as follows:

1. The old variables are replaced by new variables.
2. Part of the invariant, called the *refinement relation*, relates the new and old variables.
3. The operations in the refinement machine are modified so that the new operations simulate the old operations under the refinement relation.

The idea of refinement is that any user who is satisfied with the behaviour of an operation, must be satisfied with the behaviour of the refinement of that operation.

Notice that refinement is not equivalence. Outside the precondition of an operation, the refined operation can do anything. Refinement can also discard non-determinism.

Refinement allows an operations to be *strengthened*, but not *weakened*.

Implementation

Implementation is a form of refinement, subject to the following extra constraints:

1. the implementation machine can have no variables;
2. the implementation machine should import machines and use the variables and operations of the imported machines to implement the machine being refined;
3. *full hiding* of the variables of the imported machines is enforced, *ie* the variables of the imported machines can be neither referenced or modified directly within operations. All references and modifications within operations must be achieved through the use of operations.

Implementing the ATM machine

We now demonstrate that we can implement the ATM machine by importing the Bank machine. We also import two instances of the renameable variable machine, *Rename_Vvar*, from the standard library. These two machines implement a Boolean variable and an ACCOUNT variable, respectively, and provide a refinement of the singleton or empty set of ACCOUNT. The implementation is a programming exercise in which we are limited to using the operations of the imported machines. Notice that we are allowed to use sequential composition.

ATMI

IMPLEMENTATION *ATMI*

REFINES *ATM*

SEES *Password* , *Bool_TYPE*

IMPORTS

Bank (*100*) ,

loggedin_Vvar (*ACCOUNT*) , *islogged_Vvar* (*BOOL*)

INVARIANT

(*islogged_Vvar* = *TRUE* \Rightarrow *loggedin* \neq {}) \wedge

(*islogged_Vvar* = *FALSE* \Rightarrow *loggedin* = {}) \wedge

(*islogged_Vvar* = *TRUE* \Rightarrow { *loggedin_Vvar* } = *loggedin*)

INITIALISATION *islogged_STO_VAR* (*FALSE*)

OPERATIONS

```
response ← InsertCard ( scard , pass ) ≐
VAR bb , res , account
IN bb ← islogged_VAL_VAR ;
  IF bb = FALSE THEN
    account ← ExtractAccount ( scard ) ;
    res ← CheckPassword ( account , pass ) ;
    IF res = OK THEN
      islogged_STO_VAR ( TRUE ) ;
      loggedin_STO_VAR ( account )
    ELSE islogged_STO_VAR ( FALSE )
    END ; response := res
  ELSE response := REFUSED
  END
END ;
```

```
response , money , bal  $\leftarrow$  ATMWithdraw ( amount )  $\hat{=}$   
VAR bb , res , acc  
IN response := REFUSED ; money := 0 ; bal := 0 ;  
  bb  $\leftarrow$  islogged_VAL_VAR ;  
  IF bb = TRUE THEN  
    acc  $\leftarrow$  loggedin_VAL_VAR ;  
    res , bal  $\leftarrow$  Withdraw ( acc , amount ) ;  
    IF res = OK THEN  
      money := amount  
    END ;  
    response := res  
  END  
END ;
```

```
response , bal ← ATMDeposit ( amount ) ≐
VAR bb , res , acc
IN bb ← islogged_VAL_VAR ;
  IF bb = TRUE THEN
    acc ← loggedin_VAL_VAR ;
    response ← Deposit ( acc , amount ) ;
    res , bal ← Balance ( acc )
  ELSE bal := 0 ; response := REFUSED
  END
END ;
```

```
response , bal ← ATMBalance ≐  
VAR bb , acc  
IN bb ← islogged_VAL_VAR ;  
  IF bb = TRUE THEN  
    acc ← loggedin_VAL_VAR ;  
    response , bal ← Balance ( acc )  
  ELSE bal := 0 ; response := REFUSED  
  END  
END ;
```

WithdrawCard $\hat{=}$ *islogged_STO_VAR* (*FALSE*)

END

ATMBankSys

In order to exercise the ATM machine it is necessary to be able to create accounts and service cards, so we extend the ATM machine by adding a clone of the *OpenAccount* operation.

MACHINE *ATMBankSys*

SEES *BankContext* , *Password* , *Scalar_TYPE*

EXTENDS *ATM*

OPERATIONS

response , *account* , *scard* \leftarrow *OpenAccount* (*pass*) $\hat{=}$

PRE *pass* \in *PASSWORD* THEN

account : \in *ACCOUNT* || *scard* : \in *SCARD* || *response* :

END

END

ATMBankSysI

And we implement the ATMBankSys machine.

```
IMPLEMENTATION ATMBankSysI
REFINES ATMBankSys
SEES
  Bool_TYPE ,
  Password
IMPORTS
  Bank ( 100 ) ,
  loggedin_Vvar ( ACCOUNT ) ,
  islogged_Vvar ( BOOL )
PROMOTES OpenAccount
```

INVARIANT

$(islogged_Vvar = TRUE \Rightarrow loggedin \neq \{ }) \wedge$

$(islogged_Vvar = FALSE \Rightarrow loggedin = \{ }) \wedge$

$(islogged_Vvar = TRUE \Rightarrow \{ loggedin_Vvar \} = loggedin)$

INITIALISATION $islogged_STO_VAR (FALSE)$

OPERATIONS

```
response ← InsertCard ( scard , pass ) ≐
  VAR bb , res , account
  IN bb ← islogged_VAL_VAR ;
    IF bb = FALSE THEN
      account ← ExtractAccount ( scard ) ;
      res ← CheckPassword ( account , pass ) ;
      IF res = OK THEN
        islogged_STO_VAR ( TRUE ) ;
        loggedin_STO_VAR ( account )
      ELSE islogged_STO_VAR ( FALSE )
      END ; response := res
    ELSE response := REFUSED
  END
END ;
```

```
response , money , bal ← ATMWithdraw ( amount ) ≐
VAR bb , res , acc
IN response := REFUSED ; money := 0 ; bal := 0 ;
   bb ← islogged_VAL_VAR ;
   IF bb = TRUE THEN
       acc ← loggedin_VAL_VAR ;
       res , bal ← Withdraw ( acc , amount ) ;
       IF res = OK THEN
           money := amount
       END ;
       response := res
   END
END ;
```

```
response , bal ← ATMDeposit ( amount ) ≐
VAR bb , res , acc
IN bb ← islogged_VAL_VAR ;
  IF bb = TRUE THEN
    acc ← loggedin_VAL_VAR ;
    response ← Deposit ( acc , amount ) ;
    res , bal ← Balance ( acc )
  ELSE bal := 0 ; response := REFUSED
  END
END ;
```

```
response , bal ← ATMBalance ≐
  VAR bb , acc
  IN bb ← islogged_VAL_VAR ;
    IF bb = TRUE THEN
      acc ← loggedin_VAL_VAR ;
      response , bal ← Balance ( acc )
    ELSE bal := 0 ; response := REFUSED
    END
  END ;
  WithDrawCard ≐ islogged_STO_VAR ( FALSE )
END
```

Password Encryption

In *BankR*, the refinement of *Bank*, we model the mapping from account to password with a function $accounts \rightarrow PASSWORD$.

Looking ahead to implementation, we recognise that it would be unwise to implement a mapping from account to a plaintext password. It would be more secure to encrypt the password. To provide facilities for this we introduce a new machine *Encryption*.

We also specify the operation *CheckPassword* as comparing encrypted passwords, rather than comparing plain passwords. Notice that we need to “think ahead” on this issue: if we specified the operation as comparing plain passwords, we could not later decide to implement the operation using comparison of encrypted passwords as this is weaker than comparing plain passwords and is hence not a refinement.

Encryption

MACHINE *Encryption*

SEES *Password*

SETS *CRYPT*

CONSTANTS *ENCRYPT*

PROPERTIES $ENCRYPT \in PASSWORD \rightarrow CRYPT$

OPERATIONS

$encrypted \longleftarrow \text{Encrypt} (password) \hat{=}$

PRE $password \in PASSWORD$

THEN $encrypted := ENCRYPT (password)$

END

END

BankR

BankR is a refinement of Bank. This machine adds variables to resolve the non-determinism in the operations of Bank.

REFINEMENT *BankR*

REFINES *Bank*

SEES *Password* , *Encryption* , *Scalar_TYPE*

VARIABLES *accounts* , *password* , *balance*

INVARIANT

$accounts \subseteq ACCOUNT \wedge$

$password \in accounts \rightarrow PASSWORD \wedge$

$balance \in accounts \rightarrow SCALAR$

INITIALISATION *accounts* , *password* , *balance* := {}, {}, {}

OPERATIONS

$response, account, scard \leftarrow \text{OpenAccount}(pass) \hat{=}$

CHOICE

IF $accounts \neq ACCOUNT$

THEN ANY acc WHERE $acc \in ACCOUNT - accounts$

THEN $accounts := accounts \cup \{acc\}$ ||

$balance(acc) := 0$ ||

$password(acc) := pass$ ||

$account := acc$ ||

$scard := \text{GENSCARD}(acc)$ ||

$response := OK$

END

ELSE $account \in ACCOUNT$ || $scard \in SCARD$ ||

$response := REFUSED$

END

OR

$account \in ACCOUNT$ || $scard \in SCARD$ || $response :$

END ;

```
response  $\leftarrow$  CheckPassword ( account , pass )  $\hat{=}$   
  IF account  $\in$  accounts  $\wedge$   
    ENCRYPT ( password ( account ) ) = ENCRYPT ( pass  
  THEN response := OK  
  ELSE response := REFUSED  
  END ;
```

```

response , bal  $\leftarrow$  Withdraw ( account , amount )  $\hat{=}$ 
  IF account  $\in$  accounts  $\wedge$  amount  $\leq$  balance ( account )
  THEN balance ( account ) := balance ( account ) - amount
       bal := balance ( account ) - amount ||
       response := OK
  ELSE bal  $\in$  SCALAR || response := REFUSED
  END ;

response  $\leftarrow$  Deposit ( account , amount )  $\hat{=}$ 
  IF account  $\in$  accounts  $\wedge$  balance ( account ) + amount  $\leq$  M
       balance ( account ) := balance ( account ) + amount ||
       response := OK
  ELSE response := REFUSED
  END ;

```

```
response , bal ← Balance ( account ) ≐  
IF account ∈ accounts THEN  
    bal := balance ( account ) ||  
    response := OK  
ELSE bal ∈ SCALAR ||  
    response := REFUSED  
END ;
```

account \longleftarrow ExtractAccount (*scard*) $\hat{=}$

PRE *scard* \in SCARD

THEN *account* := GENSCARD⁻¹ (*scard*)

END

END

Implementing Bank

To implement the Bank machine we will use the B-Toolkit's base generator capability.

The database definition shown in *BankDB base* consists of a database called *bankdb* containing records with mandatory fields *accountf*, *passwordf* and *balancef*.

The B-Toolkit takes the base definition and generates the machine *BankDB*, which is imported into the implementation *BankRI*.

Notice that having given service cards the possibility of having an identity different from accounts, we implement service cards as identical to accounts. We can do this without loss of generality.

BankDB base

SYSTEM *BankDB*
SUPPORTS *BankRI*
IS

BASE

bankdb

MANDATORY

accountf \in *ACCOUNT* ;

passwordf \in *CRYPT* ;

balancef \in \mathbb{N}

END

END

BankDB

MACHINE *BankDB* (*max_bankdb* , *ACCOUNT* , *CRYPT*)

CONSTRAINTS

$max_bankdb \in 1 .. 2147483646$

SEES

BankDBCtx , *Bool_TYPE* , *Scalar_TYPE*

VARIABLES

bankdb ,

locate_bankdb ,

accountf ,

passwordf ,

balancef

INVARIANT

$bankdb \subseteq bankdb_ABSOBJ \wedge$

$card (bankdb) \leq max_bankdb \wedge$

$locate_bankdb \in 1 .. card (bankdb) \mapsto bankdb \wedge$
 $accountf \in bankdb \rightarrow ACCOUNT \wedge$
 $passwordf \in bankdb \rightarrow CRYPT \wedge$
 $balancef \in bankdb \rightarrow SCALAR$

INITIALISATION

$bankdb := \{ \} \parallel$
 $locate_bankdb := \{ \} \parallel$
 $accountf := \{ \} \parallel$
 $passwordf := \{ \} \parallel$
 $balancef := \{ \}$

OPERATIONS

$rep, Base_bankdb \leftarrow make_bankdb (Val_accountf, Val_passwordf)$

PRE

$Val_accountf \in ACCOUNT \wedge$
 $Val_passwordf \in CRYPT \wedge$

$Val_balancef \in SCALAR \wedge$
 $card (bankdb) < max_bankdb$

THEN

CHOICE

ANY $Base_bankdbx, loc$ WHERE

$Base_bankdbx \in bankdb_ABSOBJ - bankdb \wedge$

$loc \in 1 .. card (bankdb) + 1 \rightsquigarrow bankdb \cup \{ Base_ban$

THEN

$bankdb := bankdb \cup \{ Base_bankdbx \} \parallel$

$accountf (Base_bankdbx) := Val_accountf \parallel$

$passwordf (Base_bankdbx) := Val_passwordf \parallel$

$balancef (Base_bankdbx) := Val_balancef \parallel$

$Base_bankdb := Base_bankdbx \parallel$

$locate_bankdb := loc \parallel$

$rep := TRUE$

END

OR

ANY *Base_bankdbx* WHERE
Base_bankdbx \in *bankdb_ABSOBJ*

THEN

Base_bankdb := *Base_bankdbx* ||
rep := *FALSE*

END

END

END ;

rep , *Base_bankdb* \leftarrow *key_search_accountf* (*Elem_ACCOUNT*) \neq

PRE *Elem_ACCOUNT* \in *ACCOUNT* THEN

IF *Elem_ACCOUNT* \in *ran* (*accountf*) THEN

ANY *Base_bankdbx* WHERE

Base_bankdbx \in *bankdb* \wedge *accountf* (*Base_bankdbx*)

THEN

Base_bankdb := *Base_bankdbx* || *rep* := *TRUE*

```

    END
  ELSE
    Base_bankdb :∈ bankdb_ABSOBJ || rep := FALSE
  END
END ;
rep ← eql_passwordf ( Base_bankdb , Elem_CRYPT ) ≐
PRE Base_bankdb ∈ bankdb ∧ Elem_CRYPT ∈ CRYPT THEN
  rep := bool ( passwordf ( Base_bankdb ) = Elem_CRYPT )
END ;
nat ← val_balancef ( Base_bankdb ) ≐
PRE Base_bankdb ∈ bankdb THEN
  nat := balancef ( Base_bankdb )
END ;
mod_balancef ( Base_bankdb , nat ) ≐
PRE Base_bankdb ∈ bankdb ∧ nat ∈ SCALAR THEN
  balancef ( Base_bankdb ) := nat

```

END

END

BankRI

IMPLEMENTATION *BankRI*

REFINES *BankR*

SEES

Password , *Encryption* ,
Scalar_TYPE , *Scalar_TYPE_Ops* ,
Bool_TYPE ,
BankDBCtx

IMPORTS

account_Nvar (*maxaccount*) ,
BankDB (*maxaccount* , *ACCOUNT* , *CRYPT*)

PROPERTIES

ACCOUNT = 1 .. *maxaccount* \wedge
SCARD = 1 .. *MaxScalar* \wedge
GENSCARD = id (*SCARD*)

INVARIANT

$accounts = 1 .. account_Nvar \wedge$

$(password ; ENCRYPT) = (accountf^{-1} ; passwordf) \wedge$

$accountf \in bankdb \rightsquigarrow accounts \wedge$

$balance = (accountf^{-1} ; balancef)$

OPERATIONS

```
response , account , scard ← OpenAccount ( pass ) ≐
VAR bb , db , nn , encpass
IN account := maxaccount ; scard := maxaccount ; response := ERROR ;
   bb ← account_NEQ_NVAR ( maxaccount ) ;
   IF bb = TRUE THEN
       account_INC_NVAR ;
       nn ← account_VAL_NVAR ;
       encpass ← Encrypt ( pass ) ;
       bb , db ← make_bankdb ( nn , encpass , 0 ) ;
       IF bb = TRUE THEN
           account := nn ; scard := nn ;
           response := OK
       ELSE account_DEC_NVAR
       END
   END
END
END ;
```

```
response ← CheckPassword ( account , pass ) ≐
VAR bb , db , encpass
IN bb , db ← key_search_accountf ( account ) ;
  IF bb = TRUE THEN
    encpass ← Encrypt ( pass ) ;
    bb ← eql_passwordf ( db , encpass )
  END ;
  IF bb = TRUE THEN
    response := OK
  ELSE response := REFUSED
  END
END ;
```

```
response , bal ← WithDraw ( account , amount ) ≐
VAR bb , db , balv
IN response := REFUSED ; bal := 0 ;
   bb , db ← key_search_accountf ( account ) ;
   IF bb = TRUE THEN
       balv ← val_balancef ( db ) ;
       IF amount ≤ balv THEN
           balv ← SUB ( balv , amount ) ;
           mod_balancef ( db , balv ) ;
           bal := balv ;
           response := OK
       END
   END
END ;
```

```
response ← Deposit ( account , amount ) ≐
  VAR bb , db , bal
  IN response := REFUSED ;
    bb , db ← key_search_accountf ( account ) ;
    IF bb = TRUE THEN
      bal ← val_balancef ( db ) ;
      IF bal ≤ MaxScalar – amount
        THEN
          bal ← ADD ( bal , amount ) ;
          mod_balancef ( db , bal ) ;
          response := OK
        END
      END
    END ;
```

```
response , bal ← Balance ( account ) ≐
VAR bb , db
IN bal := 0 ; response := REFUSED ;
   bb , db ← key_search_accountf ( account ) ;
   IF bb = TRUE THEN
       bal ← val_balancef ( db ) ;
       response := OK
   END
END ;
```

account \leftarrow ExtractAccount (*scard*) $\hat{=}$

account := *scard*

END

Implementing Encryption

Having set the scene for encryption, we will implement encryption by making the encrypted password the same as the password. Clearly, we can do this without loss of generality as we could use any other encryption we wish.

IMPLEMENTATION *EncryptionI*

REFINES *Encryption*

SEES *Password*

PROPERTIES

$CRYPT = PASSWORD \wedge$

$ENCRYPT = \text{id} (PASSWORD)$

OPERATIONS

$encrypted \leftarrow \text{Encrypt} (password) \hat{=}$

$encrypted := password$

END

APPENDIX

The following slides show the instantiation of the machines *loggedin_Vvar* and *islogged_Vvar.mch*.

loggedin_Vvar

MACHINE *loggedin_Vvar* (*VALUE*)
SEES *Bool_TYPE*
VARIABLES *loggedin_Vvar*
INVARIANT *loggedin_Vvar* \in *VALUE*
INITIALISATION *loggedin_Vvar* $:\in$ *VALUE*

OPERATIONS

$VV \longleftarrow$ *loggedin_VAL_VAR* $\hat{=}$

BEGIN

$vv :=$ *loggedin_Vvar*

END ;

loggedin_STO_VAR (*VV*) $\hat{=}$

PRE

$VV \in VALUE$

THEN

$loggedin_Vvar := vv$

END ;

$bb \leftarrow loggedin_EQL_VAR (VV) \hat{=}$

PRE

$VV \in VALUE$

THEN

$bb := \text{bool} (loggedin_Vvar = vv)$

END ;

$bb \leftarrow loggedin_NEQ_VAR (VV) \hat{=}$

PRE

$VV \in VALUE$

THEN

$bb := \text{bool} (loggedin_Vvar \neq vv)$

END ;

loggedin_SAV_VAR $\hat{=}$

BEGIN skip END ;

loggedin_RST_VAR $\hat{=}$

BEGIN

loggedin_Vvar : \in *VALUE*

END ;

loggedin_SAVN_VAR $\hat{=}$

BEGIN skip END ;

loggedin_RSTN_VAR $\hat{=}$

BEGIN

loggedin_Vvar : \in *VALUE*

END

END

islogged_Vvar

MACHINE *islogged_Vvar* (*VALUE*)
SEES *Bool_TYPE*
VARIABLES *islogged_Vvar*
INVARIANT *islogged_Vvar* \in *VALUE*
INITIALISATION *islogged_Vvar* : \in *VALUE*

OPERATIONS

$VV \longleftarrow$ *islogged_VAL_VAR* $\hat{=}$

BEGIN

$vv :=$ *islogged_Vvar*

END ;

islogged_STO_VAR (VV) $\hat{=}$

PRE

$VV \in VALUE$

THEN

$islogged_Vvar := vv$

END ;

$bb \leftarrow islogged_EQL_VAR (VV) \hat{=}$

PRE

$VV \in VALUE$

THEN

$bb := \text{bool} (islogged_Vvar = vv)$

END ;

$bb \leftarrow islogged_NEQ_VAR (VV) \hat{=}$

PRE

$VV \in VALUE$

THEN

$bb := \text{bool} (islogged_Vvar \neq vv)$

END ;

islogged_SAV_VAR $\hat{=}$

BEGIN skip END ;

islogged_RST_VAR $\hat{=}$

BEGIN

islogged_Vvar : \in *VALUE*

END ;

islogged_SAVN_VAR $\hat{=}$

BEGIN skip END ;

islogged_RSTN_VAR $\hat{=}$

BEGIN

islogged_Vvar : \in *VALUE*

END

END