# The B Method

**Dang Van Hung**

United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau
http://www.iist.unu.edu

## Acknowledgements

*The B Method is the result of the inspirational work of Jean-Raymond Abrial, who developed it with a team of researchers at BP Sunbury in the late 1980s. He is also one of the creators of the specification language Z.*

*For the definitive reference to the B Method and its underlying theory, readers are referred to Abrial's The B Book (Cambridge University Press).*

## Aspects of a Design Method

1. A notation for specification of software.
2. A technique for software design.
3. Toolkit in support of software development.

## What is B?

- **B is a method for specifying, designing and coding software systems**
- **Main features:**
  - Abstract Machine, data and operations
  - Specification of data, specification of operations
  - Refinement towards an implementation, Refinement techniques
  - Library, reuse, code generation
  - Proof
  - B-Toolkit

# Contents

1. **How to specify software using the notations of abstract machine.**
   (a) How to ensure the consistency of a machine.
   (b) How to construct a large machine from smaller ones.
   (c) How to derive software from its specification.
   (d) How to use the concept of refinement to link a machine with its implementation
2. **ProB: an animator and model checker for B.**

# A Model for Pocket Calculator

1. A *visible* screen for output.
2. An *invisible* memory (or better the values stored in it) forms the *state* of the Calculator.
3. Various keys are the *operations*, which a user is able to activate in order to modify the memory.
4. A software system can always be regarded as Pocket Calculator, and modelled by **Abstract Machine**

# Abstract Machine Notation

An Abstract Machine consists of

1. **Machine name (probably with parameters)** specified by **MACHINE** clause
2. **State**
   - **VARIABLES** clause: a list of names denoting the components of the state.
   - **INVARIANT** clause: a logical statement making clear what the static laws of the machine are.
   - **INITIALISATION** clause: a pseudo program showing how the machine is initialised.
3. **Operations**
   - **OPERATIONS** clause: a list of operation definitions which for each operation specifies the input, the output and its effect on the state variables.

# Counter

**MACHINE**
    $counter$
**VARIABLES**
    $c$
**INVARIANT**
    $c \in NAT$
**INITIALISATION**
    $c := 0$
**OPERATIONS**
    $inc \;=\;$ **BEGIN** $c := c + 1$ **END**
    ;
    $reset \;=\;$ **BEGIN** $c := 0$ **END**
**END**

# Parameters and Constraints

**Example: A reservation system**

**MACHINE**
  $booking(max\_seat)$
**CONSTRAINTS**
  $max\_seat \in NAT$
**VARIABLES**
  $seat$
**INVARIANT**
  $seat \in NAT \wedge$
  $seat \leq max\_seat$
**INITIALISATION**
  $seat := max\_seat$

**OPERATIONS**
  $book \ = \ \ldots$
  ;
  $cancel \ = \ \ldots$
**END**

*When this machine is used as part of a large machine, the formal parameter $max\_seat$ has to be assigned a value satisfying the predicate in the* **CONSTRAINTS** *clause*

# Simple Substitution

**Example: A Resource Management System**

**MACHINE**
  $rms(RES)$
**CONSTRAINTS**
  $RES \subseteq NAT$
**VARIABLES**
  $rfree$
**INVARIANT**
  $rfree \subseteq RES$
**INITIALISATION**
  $rfree := \emptyset$
**OPERATIONS**
  $\ldots$
**END**

*$rfree := \emptyset$ is not the kind of things to be found in most programming languages, and is referred to a simple substitution.*

# PRE-THEN Substitution (1)

The $book$ operation is used to reserve a seat in the booking system.

Written without special care:

$$book \ = \ \textbf{BEGIN}\ seat := seat - 1\ \textbf{END}$$

When $seat$ is already equal to 0, the effect of the operation will *break the invariant $seat \in NAT$.*

To make this specification work, we have to add to it an ad-hoc **pre-condition** explaining in which case one is entitled to activate the $booking$ operation.

$booking \ \widehat{=}$ **PRE**
  $0 < seat$
**THEN**
  $seat := seat - 1$
**END**

# PRE-THEN Substitution (2)

The $cancel$ operation is used to cancel a previous reservation. It can be activated only when the number of available ticket is less than $max\_seat$

$cancel \ \widehat{=}$ **PRE**
  $seat < max\_seat$
**THEN**
  $seat := seat + 1$
**END**

# Operation Input Parameter (1)

**Example: Operations of the machine $rms$**

The $free(rr)$ operation has input parameter $rr$, and
it is used to free the resource $rr$.
The **precondition** is introduced to establish the type of $rr$.
The **then-part** says that the input is added to $rfree$.

$$\begin{array}{lll} free(rr) & \widehat{=} & \textbf{PRE} \\ & & \quad rr \in RES \ \wedge \ rr \notin rfree \\ & & \textbf{THEN} \\ & & \quad rfree := rfree \cup \{rr\} \\ & & \textbf{END} \end{array}$$

**The input parameters of an operation must be a list of distinct variables distinct from the state variables.**

# Operation Input Parameter (2)

**Example: Operations of the machine $rms$ (Cont'd)**

The $alloc(rr)$ allocates a free resource $rr$.

$$\begin{array}{lll} alloc(rr) & \widehat{=} & \textbf{PRE} \\ & & \quad rr \in rfree \\ & & \textbf{THEN} \\ & & \quad rfree := rfree - \{rr\} \\ & & \textbf{END} \end{array}$$

$alloc$ and $free$ seem to be doing useful things, *but each has a precondition that the user of the machines must guarantee*. We need additional information to be able to use our machine in a safe way, that is to test the pre-condition on the user's side.

1. Adding an enquiry operation to report whether a resource is free.
2. Making the operations robust by giving them outputs that report whether they worked, or why they did not.

# Testing a Precondition

The $isfree$ operation has an input parameter $rr$ and an output parameter $avail$. As a syntactic restriction, **the output parameter must be distinct from input parameters and state variables**.

$$\begin{array}{lll} avail \longleftarrow isfree(rr) & \widehat{=} & \textbf{PRE} \\ & & \quad rr \in RES \\ & & \textbf{THEN} \\ & & \quad avail := (rr \in rfree) \\ & & \textbf{END} \end{array}$$

# Conditional Substitution

We may redefine $alloc$ operation in the following way:

$$\begin{array}{lll} report \longleftarrow alloc(rr) & \widehat{=} & \textbf{PRE} \\ & & \quad rr \in RES \\ & & \textbf{THEN} \\ & & \quad \textbf{IF} \\ & & \qquad rr \in rfree \\ & & \quad \textbf{THEN} \\ & & \qquad report, rfree := good,\ rfree - \{rr\} \\ & & \quad \textbf{ELSE} \\ & & \qquad report := bad \\ & & \quad \textbf{END} \\ & & \textbf{END} \end{array}$$

# Robust Machine *booking*

$report \longleftarrow book \ \widehat{=}$
**IF**
$0 < seat$
**THEN**
$report, seat := good, \ seat - 1$
**ELSE**
$report := bad$
**END**

$report \longleftarrow cancel \ \widehat{=}$
**IF**
$seat < max\_seat$
**THEN**
$report, seat := good, \ seat + 1$
**ELSE**
$report := bad$
**END**

Check $report$ before using the results!

# Exercises

1. Write a PRE-THEN substitution for the operation $req(nbr)$ of the machine $booking$ which is used to request $nbr$ tickets.
2. Write an enquiry operation $available(nbr)$ which informs the user whether there are $nbr$ tickets available.
3. Redefine the operation $req(nbr)$ in a robust style which reports the user whether the operation $req$ has worked.

# Data Type Arrays (1)

We build a machine for an array, which is modelled as a total function $table$, from a set $INDEX$ to a set $VALUE$, both of which are parameters of the machine.

**MACHINE**
$array(INDEX, \mathit{VALUE})$
**CONSTRAINTS**
$INDEX \subseteq NAT \wedge$
$VALUE \subseteq INT$
**VARIABLES**
$table$
**INVARIANTS**
$table \in \mathit{INDEX} \rightarrow \mathit{VALUE}$
**INITIALISATION**
$table := \lambda x : INDEX \bullet 0$

# Data Type Arrays (2)

We first offer two operations, one for entering a new value into the $array$ at a specific index, and one for accessing the value stored in the array at a given index.

$enter(value, \ index) \ \widehat{=}$
**PRE**
$index \in INDEX \ \wedge$
$value \in VALUE$
**THEN**
$table(index) := value$
**END**

$value \longleftarrow access(index) \ \widehat{=}$
**PRE**
$index \in INDEX$
**THEN**
$value := table(index)$
**END**

# Nondeterministic Choice

The operation $index \longleftarrow search(value)$ searches an index of the array at which the value of its input parameter is stored. The operation is specified as a **non-deterministic choice** in the set of indexes where $value$ is stored.

$index \longleftarrow search(value) \;\widehat{=}$
**PRE**
$\quad value \in \mathsf{range}(table)$
**THEN**
$\quad index :\in table^{-1}(\{value\})$
**END**

($index :\in table^{-1}(\{value\})$: $index$ is assigned with any value in $table^{-1}(\{value\})$)

# Consistency Check

An abstract machine expresses precisely the properties of its state and its operations. The **VARIABLES** and **INVARIANT** clauses together characterise the state of the machine. If a machine is to represent anything worthwhile, it must ensure the invariant properties as respected, i.e.

- **Consistency of initialisation – the initialisation must establish the invariant.**
- **Consistency of operation – each operation must preserve the invariant.**

(Checking can be done automatically with tools!)

# Substitution Predicates

The statement "**the initialisation** $rfree := \emptyset$ **establishes the invariant** $rfree \subseteq RES$" can be formalised in a predicate:

$$[rfree := \emptyset](rfree \subseteq RES)$$

where the program $rfree := \emptyset$ in square brackets acts as a function name and the invariant $ref \subseteq RES$ is its argument. The value of this function can be calculated by replacing variable $rfree$ in the argument with constant $\emptyset$:

$$\begin{aligned} [rfree := \emptyset](rfree \subseteq RES) &= \emptyset \subseteq RES \\ &= true \end{aligned}$$

We call $[rfree := \emptyset](rfree \subseteq RES)$ a **substitution predicate**.

# Proof Obligation 1

The initialisation of the machine $booking$ is consistent, because

$$\begin{aligned} [seat := max\_seat](seat \leq max\_seat) &= max\_seat \leq max\_seat \\ &= true \end{aligned}$$

**In general, if I is the invariant of the machine, and G is a program for the INITIALISATION clause, to establish initialisation consistency we must prove**

$$[\mathbf{G}](\mathbf{I}) = true$$

# Preserving Invariant

An operation operates under two assumptions:

- The invariant is true when the user of the machine invokes the operation.
- The precondition of the operation is true.

We have to prove under these assumptions the execution of the operation will re-establish the invariant.

# The operation $free$

Consider the $free$ operation of the machine $rms$. The invariant is $rfree \subseteq RES$, and the precondition of $free$ is $rr \in RES \wedge rr \notin rfree$:

$$
\begin{array}{l}
(\mathbf{rfree} \subseteq \mathbf{RES}) \wedge (\mathbf{rr} \in \mathbf{RES} \wedge \mathbf{rr} \notin \mathbf{rfree}) \;\Rightarrow \\
[rfree := rfree \cup \{rr\}](\mathbf{rfree} \subseteq \mathbf{RES}) \\
\equiv \\
(rfree \subseteq RES) \wedge (rr \in RES \wedge rr \notin rfree) \;\Rightarrow \\
(rfree \cup \{rr\}) \subseteq RES \\
\equiv \\
true
\end{array}
$$

# The operation $alloc$

For the $alloc$ operation of $rms$ we have to show

$$
\begin{array}{l}
(\mathbf{rfree} \subseteq \mathbf{RES}) \wedge (\mathbf{rr} \in \mathbf{free}) \;\Rightarrow \\
[rfree := rfree - \{rr\}](\mathbf{rfree} \subseteq \mathbf{RES}) \\
\equiv \\
(rfree \subseteq RES) \;\Rightarrow \\
(rfree - \{rr\}) \subseteq RES \\
\equiv \\
true
\end{array}
$$

# Proof Obligation 2

**If I is the invariant,**
**if G is the pseudo program for the operation,**
**and if P is the stated pre-condition,**
**the proof obligation is**

$$(\mathbf{I} \wedge \mathbf{P}) \Rightarrow [\mathbf{G}](\mathbf{I})$$

# Calculate Substitution Predicates

1. $[x := E](\mathbf{I}) \;=\; \mathbf{I}[E/x]$
   where $\mathbf{I}$ is obtained by substituting for all free occurrences of $x$ in $\mathbf{I}$:
   $$[x := 1](x \neq y) \;=\; y \neq 1$$
   $$[x := 0](y \geq 1) \;=\; y \geq 1$$
2. $[\mathbf{PRE\ P\ THEN\ G}](\mathbf{I}) \;=\; \mathbf{P} \wedge [\mathbf{G}](\mathbf{I})$
   $$\begin{aligned}[\mathbf{PRE}\, x > 0\, \mathbf{THEN}\, x := x - 1\, \mathbf{END}](x > y) \quad = \quad & (x > 0) \wedge (x - 1 > y) \\ & (x > 0) \wedge (x > y + 1) \\ & x > \mathbf{max}(0,\, y + 1)\end{aligned}$$
3. $[\mathbf{IF\ P\ THEN\ G1\ ELSE\ G2\ END}](\mathbf{I}) \;=\; (\mathbf{P} \wedge [\mathbf{G1}](\mathbf{I})) \;\vee\; (\neg\mathbf{P} \wedge [\mathbf{G2}](\mathbf{I}))$

# Inference Rules

1. $P \Rightarrow true$
2. $false \Rightarrow R$
3. If $P \Rightarrow R1$ and $P \Rightarrow R2$, then $P \Rightarrow (R1 \wedge R2)$.
4. If $P1 \Rightarrow R$ and $P2 \Rightarrow R$, then $(P1 \vee P2) \Rightarrow R$

# Homework

1. Define an abstract machine of a single scalar $vv$. The variable $vv$ is supposed to belong to a certain set $VAL$ which is the parameter of the machine. The operations of machine should be relevant operations modifying, accessing and testing the variable $vv$. You are also required to check the consistency of the abstract machine.
2. Make the indicated substitution in the following predicates.

   (a)$[x := x + 1](x > 0)$
   (b)$[x := x \cup \{r\}](x \subseteq y)$
   (c)$[\mathbf{PRE}\ x > y\ \mathbf{THEN}\ x := y\ \mathbf{END}](x > 0)$
   (d)$[\mathbf{IF}\ x > y\ \mathbf{THEN}\ x := y\ \mathbf{END}](x > 0)$

# The Class Manager's Assistant

The informal requirement of the system includes

- to keep track of students enrolled in the class, up to a certain number (denoted by $class\_size$).
- to record whether a student has done the exercises set for the class.

Students may leave the class at any time, but only those who have done the exercises are entitled to a leaving certificate. The class manager needs to check from time to time whether a student has done the exercises.

# The **SEES** Clause

The **SEES** clause is used to introduce the names of machines that supply information we need later in the machine definition.

**MACHINE**
$cma(class\_size,\ STUDENT)$
**CONSTRAINTS**
$class\_size \in NAT\ \wedge$
$class\_size > 0 \wedge$
$\mathbf{card}(STUDENT) \in NAT$
**SEES**
$BOOL-TYPE$

**VARIABLES**
$enrolled,\ tested$
**INVARIANT**
$enrolled \subseteq STUDENT\ \wedge$
$tested \subseteq enrolled\ \wedge$
$\mathbf{card}(enrolled) \leq class\_size$
**INITIALISATION**
$enrolled,\ tested\ := \emptyset,\ \emptyset$

# Consistency of Initialisation

We need to establish

$$
\begin{aligned}
&[enrolled,\ tested\ := \emptyset,\ \emptyset] \\
&(\mathbf{enrolled} \subseteq \mathbf{STUDENT}\ \wedge \\
&\mathbf{tested} \subseteq \mathbf{enrolled}\ \wedge \\
&\mathbf{card}(\mathbf{enrolled}) \leq \mathbf{class\_size}) \quad = \quad (\emptyset \subseteq STUDENT\ \wedge \\
&\qquad\qquad \emptyset \subseteq \emptyset\ \wedge \\
&\qquad\qquad \mathbf{card}(\emptyset) \leq class\_size) \\
&\quad = \quad true
\end{aligned}
$$

# Enrolling a Student (1)

**Input Parameter** : $st : STUDENT$

**Output Parameter** : $res : Seq[Char]$

**Function**:

- If the class is full, the output is *"No room"*.
- If the class is not full, but the student $st$ is already enrolled, it outputs *"Already enrolled"*.
- If the class is not full and $st$ is not already enrolled, the output is *"Student enrolled"*, and $st$ is added to the enrolled set, but not to the tested set.

# Enrolling a Student (2)

$res \longleftarrow enroll(st)\ \widehat{=}$
**PRE**
$st \in STUDENT$
**THEN**
**IF** $st \in enrolled$
**THEN** $res := ''already\ enrolled''$
**ELSE**
**IF** $\mathbf{card}(enrolled) = class\_size$
**THEN** $res := ''no\ room''$
**ELSE** $res := ''student\ enrolled''\ \|$
$enrolled := enrolled \cup \{st\}$
**END**
**END**
**END**

# Enquires

**Input Parameter**: $st : STUDENT$
**Output Parameter**: $res : BOOL$
**Function**: If the student has not been enrolled, the response is $FALSE$. Otherwise, the response is $TRUE$.

$res \longleftarrow enquiry(st) \quad \widehat{=}$ **PRE** $st \in STUDENT$
**THEN**
**IF** $st \in enrolled$
**THEN**
$res := TRUE$
**ELSE**
$res := FALSE$
**END**
**END**

# Exercise

Specify the following $enquiry$ operation.

**Input Parameter**: $st : STUDENT$

**Output Parameter**: $res : Seq[Char]$

**Function**: If the student is not enrolled, the response is "Not enrolled". If the student has already been tested, the response is "Student tested". If the student is enrolled, but not tested, the response is "Enrolled but not tested".

# Parallel Substitution

The leave operation is to record that a student has left the class.

$leave(st) \ \widehat{=}$
**PRE**
$st \in enrolled$
**THEN**
**IF**
$st \in tested$
**THEN**
$tested := tested - \{st\} \parallel$
$enrolled := enrolled - \{st\}$
**ELSE**
$enrolled := enrolled - \{st\}$
**END**
**END**

where
$(x := E) \| (y := F) \ \widehat{=}$
$(x, y := E, F)$

# More Operations

$test(st) \ \widehat{=}$
**PRE**
$st \in enrolled \land$
$st \notin tested$
**THEN**
$tested := tested \cup \{st\}$
**END**

$res \longleftarrow istested(st) \ \widehat{=}$
**PRE**
$st \in enrolled$
**THEN**
**IF**
$st \in tested$
**THEN**
$res := TRUE$
**ELSE**
$res := FALSE$
**END**
**END**

# Exercises

Specify a machine $NatSet$ whose model is a finite set of natural numbers in a limited range. There are two parameters: the size of the largest set to be modelled, and the largest natural number to be saved in the set. There are four operations

1. Find the size of the set.
2. Add a number to the set if the number is in the appropriate range and the set is not full.
3. Remove a number if the number is in the appropriate range.
4. Find a number in the set: if the number is in the appropriate range, return 1 if it is in the set, and 0 if it is not.

# Deferred and Enumerated Sets

When we use sets as parameters, their values are decided when the machine is instantiated. Sometimes it is convenient to define a set within the machine, and to leave decisions about the contents of the set to the implementer.

Consider a machine which might be a component machine in a banking system, and keeps track of the relationship between accounts and customers.

**MACHINE**
 $owners$
**SETS**
 $ACCTNO; \ CUSTNO$
 $O\_REP \ = \ \{success, noroom\}$

# **CONSTANTS** and **PROPERTIES**

For administrative purposes in the bank, we need to have a special account number and a special customer number.

**CONSTANTS**
 $specacctno,$
 $speccustno$
**PROPERTIES**
 $specacctno \in ACCTNO \wedge$
 $speccustno \in CUSTNO$

The **PROPERTIES** clause is a predicate which expresses any desired relationship between the constants and the deferred sets. It can also refer to the parameter of the machine.

# State Components

**VARIABLES**
 $owner$
**INVARIANT**
 $owner \in ACCTNO \leftrightarrow CUSTNO \wedge$
 $specacctno \mapsto speccustno \in owner$

The **INVARIANT** clause gives the type of the variable $owner$, and says that the special account is owned by the special customer.

The **INITIALISATION** clause has to establish this invariant

**INITIALISATION**
 $owner := \{specacctno \mapsto speccustno\}$

# DEFINITIONS Clause

The **DEFINITIONS Clause** simplifies the expression of preconditions and other aspects of the operations. For example

**DEFINITIONS**

$accounts \mathrel{\widehat{=}} \mathbf{dom}(owner)\,;$

$customers \mathrel{\widehat{=}} \mathbf{ran}(owner)$

# CHOICE-OR-END (1)

Consider an operation to add a new account for an existing customer, which has that customer number as its input. As for the account number, we can make a choice: either the new account number is given as an input, or the machine does not allocate a new account.

$res \longleftarrow new\_acct\_old\_cust(cust,\, acct) \ \mathrel{\widehat{=}}$
**PRE**
  $cust \in customers \wedge$
  $acct \in ACCTNO - accounts$
**THEN**
  **CHOICE**
    $res := \text{"}success\text{"} \ \| owner(acct) := cust$
  **OR**
    $res := \text{"}noroom\text{"}$
  **END**
**END**

# CHOICE-OR-END (2)

$$\boxed{\begin{array}{l}[\textbf{CHOICE G OR H END}\,](P) \\ = \\ [\mathbf{G}](P) \ \wedge \ [\mathbf{H}](P)\end{array}}$$

The rule says that since either $\mathbf{G}$ or $\mathbf{H}$ might happen, each of them must be able to establish the predicate $P$.

# Functional Overriding

The substitution $owner(acct) := cust$ is a functional overriding substitution. It has the form of a simple substitution

$$f(x) := E$$

But its meaning is more elaborate

$$f \ := \ f \ \oplus \{x \mapsto E\}$$

where

$$\begin{array}{lcll}(f \oplus \{x \mapsto E\})(y) & = & f(y) & \text{if } y \neq x \\ (f \oplus \{x \mapsto E\})(y) & = & E & \text{if } y = x\end{array}$$

# Non-deterministic Output

$res, acct \longleftarrow alt\_new\_acct\_old\_cust(cust) \ \widehat{=}$

**PRE**

$cust \in customers \ \wedge \ accounts \neq ACCTNO$

**THEN**

**CHOICE**

**ANY** $ac$

**WHERE**

$ac \in ACCTNO - accounts$

**THEN**

$res := \ ''success'' \ \|$

$owner(ac) := cust \ \| acct := ac$

**END**

**OR**

$res := \ ''noroom'' \ \| acct :\in ACCTNO$

**END**

**END**

# ANY-WHERE-THEN-END

The substitution has three parts

1. The **ANY** part introduces local variables used in this substitution.
2. The **WHERE** part gives the constraints to local variables.
3. The **THEN** part introduces the substitution which uses the local variables.

$[\textbf{ANY } x \textbf{ WHERE } P \textbf{ THEN } G \textbf{ END}](Q)$

$=$

$\forall x \bullet (P \Rightarrow [G](Q))$

# Choice from a Set

The pseudo program $acct :\in ACCTNO$ says that any member of the set $ACCTNO$ can be assigned to $acct$. This is another way of expression of non-determinism.

$[x :\in S](Q)$

$=$

$\forall v \bullet (v \in S \Rightarrow Q[v/x])$

# Storage Management System

A storage management system has a certain number $maxblocks$ of blocks. A block can only be allocated to one user at a time, and a user must free a block before it can be allocated to another user. The blocks are identified by natural numbers in the range $1..maxblocks$. There are user identifiers, but what these are has not yet been decided ($\nrightarrow$ denotes partial mapping).

**MACHINE**

$storman$

**SETS**

$USER$

**CONSTANTS**

$maxblocks$

**PROPERTIES**

$maxblocks \in NAT - \{0\}$

**VARIABLES**

$alloc$

**INVARIANT**

$alloc \in 1..maxblocks \nrightarrow USER$

**DEFINITIONS**

$allocated \ \widehat{=} \ \mathbf{dom}(alloc)$

**INITIALISATION**

$alloc := \emptyset$

# Acquire a Free Block

**Input Parameter**: $user : USER$
**Output Parameter**: $block : NAT$
**Function**: It can be activated where there is at least one free block. Its output is the number of any free block, and that block is allocated to the input user.

$block \longleftarrow acquire(user) \ \widehat{=}$
**PRE**
  $allocated \subset (1..maxblocks) \ \wedge user \in USER$
**THEN**
  ANY $x$
  **WHERE**
    $x \in (1..maxblocks) - allocated$
  **THEN**
    $block := x \parallel alloc(x) := user$
  **END**
**END**

# Release a Block of Storage

**Input Parameter**: $user : USER, block : NAT$
**Function**: The input block number must be the number of a block allocated to the input user. The block is freed.

$release(user, block) \ \widehat{=}$
**PRE**
  $user \in USER \wedge$ $block \in allocated \wedge$
  $alloc(block) = user$
**THEN**
  $alloc := alloc - \{block \mapsto user\}$
**END**

**Exercise**: Write a robust specification for the $release$ operation which tests the preconditions.

# Release all the Storage

**Input Parameter**: $user : USER$
**Function**: It releases all the blocks which are allocated to the input user.

$free(user) \ \widehat{=}$
**PRE**
  $user \in USER$
**THEN**
  $alloc := alloc \rhd (USER - \{user\})$
**END**

# Range Restriction

Let $f$ be a mapping from the set $S$ to the set $T$. Let $Y$ be a subset of $T$.
The notation $f \rhd Y$ denotes the function $f$ restricted its range to the set $Y$.

$$\boxed{\begin{array}{l} f \rhd Y \ \widehat{=} \ \{x \mapsto f(x) \mid f(x) \in Y\} \\ f \ntriangleright Y \ \widehat{=} \ \{x \mapsto f(x) \mid f(x) \notin Y\} \end{array}}$$

**Example**

Let $f(x) \widehat{=} x^2$ be a mapping from $NAT$ to $NAT$. Let $EVEN$ be the set of even numbers, and $ODD$ the set of odd numbers.
Then $f \rhd EVEN$ becomes a mapping from $EVEN$ to $EVEN$, and $f \ntriangleright EVEN$ becomes a mapping from $ODD$ to $ODD$.

# Number of Free Blocks

**Output Parameter**: $number : NAT$
**Function**: The output is the number of free blocks.

$number \longleftarrow find \ \widehat{=}$
**BEGIN**
$\quad number := maxblocks - \mathbf{card}(allocated)$
**END**

# The Resource Manager Revised

**MACHINE**
$\quad rms(RES)$
**CONSTRAINTS**
$\quad RES \subseteq NAT$
**VARIABLES**
$\quad rfree$
**INVARIANT**
$\quad rfree \subseteq RES$

**OPERATIONS**
$\quad free(rr) \ \widehat{=}$
$\quad$ **PRE** $rr \in RES \land rr \notin rfree$
$\quad$ **THEN** $rfree := rfree \cup \{rr\}$
$\quad$ **END**
$\quad ;$
$\quad alloc(rr) \ \widehat{=}$
$\quad$ **PRE** $rr \in rfree$
$\quad$ **THEN** $rfree := rfree - \{rr\}$
$\quad$ **END**;
$\quad setfree(X) \ \widehat{=}$
$\quad$ **PRE** $X \subseteq RES$
$\quad$ **THEN** $rfree := X$
$\quad$ **END**
**END**

# Recoverable RM

We are going to construct a recoverable resource manager $rrms$ from the simple machine $rms$ by adding following operations:

**Back up**

**Function:** The current state of resource is saved.

**Restore**

**Function:** The most recently saved state of resources is restored.

# Recoverable RM (2)

**Allocate a resource**

**Output Parameter:** $response, resource$

**Function:** If there are any free resources, one is chosen, and the resource and a response of *"Resource allocated"* are output.
Otherwise the response is *"No free resource"*.

**Deallocate a resource**

**Input Parameter:** $resource$.

**Output Parameter:** $response$.

**Function:** If the resource is allocated, it is made free, and the response is *"Resource freed"*. Otherwise the response is *"Already free"*.

# New Features of $rrms$

The machine $rrms$ develops the machine $rms$ in the two ways:
(1) It includes two versions of $rms$, one to represent the current state of resources and one to represent the most recently saved state.
(2) It provides enquiry operation to test the precondition.

**MACHINE**
$rrms(RESOURCE, max\_res)$
**CONSTRAINTS**
$RESOURCE \subseteq NAT$
$\mathbf{card}(RESOURCE) \leq max\_res$
**SEES**
$BOOl\_TYPE$
**INCLUDES**
$rms(RESOURCE),$
$bkup.rms(RESOURCE)$

# The INCLUDES Clause

It introduces previously defined machines whose variables are to be variables of the new machine. If the same machine is to be included several times, it is necessary to distinguish the several instances by renaming them. The distinguished names are prefixed to the machine names.

**The INCLUDES clause is much more powerful than the SEES clause, where the latter has limited reference to the variables of the seen machine, but is NOT allowed to change them.**

(Current version of ProB does not support multiple machines!)

# Renaming and Encapsulation

Renaming affects the names of variables, so the variable *rfree* of $rms$ becomes *bkup.rfree* in the machine *bkup.rms*. The variables of the included machines can be used in predicates like the invariant of the new machine. However, their values can only be changed by using the operations of the included machines in the definitions of the operations of the new machine.

The operations are also renamed. **However, the operations of the included machines are not necessary operations of the including machine.**

# Invariant and Initialisation

The $rrms$ machine has no variables other than $rfree$ and $bkup.rfree$. However, we add new constraint over these variables in the including machine.

**INVARIANT**
$\mathbf{card}(rfree) \leq max\_res \wedge$
$\mathbf{card}(bkup.rfree) \leq max\_res$

There is NO new initialisation in the including machine. It is given by the included machines

$$rfree, bkup.rfree := \emptyset, \emptyset$$

# Allocate a New Resource

The operation to allocate a resource is $rec\_alloc$. It produces an output, the resource to be allocated.

$$
\begin{array}{l}
res \longleftarrow rec\_alloc \;\;\widehat{=} \\
\textbf{PRE} \\
\quad rfree \neq \emptyset \\
\textbf{THEN} \\
\quad \textbf{ANY}\ rr \\
\quad \textbf{WHERE} \\
\quad\quad rr \in rfree \\
\quad \textbf{THEN} \\
\quad\quad res := rr \parallel \\
\quad\quad alloc(rr) \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

# Operation Consistency

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\mathbf{card}(rfree) \leq max\_res) \;\wedge\; (rfree \neq \emptyset) \;\Rightarrow} \\
\multicolumn{2}{l}{[\textbf{ANY}\ rr\ \textbf{WHERE}\ rr \in rfree\ \textbf{THEN}\ alloc(rr)\ \textbf{END}](\mathbf{card}(rfree) \leq max\_res)} \\
\equiv & (\mathbf{card}(rfree) \leq max\_res) \;\wedge\; (rfree \neq \emptyset) \;\Rightarrow \\
& \forall rr \bullet (rr \in rfree \Rightarrow \\
& \qquad [alloc(rr)](\mathbf{card}(rfree) \leq max\_res)) \\
\equiv & (\mathbf{card}(rfree) \leq max\_res) \;\wedge\; (rfree \neq \emptyset) \;\Rightarrow \\
& \forall rr \bullet (rr \in rfree \Rightarrow \\
& \qquad [\ \textbf{PRE}\ rr \in rfree \\
& \qquad \textbf{THEN}\ rfree := rfree - \{rr\}\ \textbf{END}\ ](\mathbf{card}(rfree) \leq max\_res)) \\
\equiv & (\mathbf{card}(rfree) \leq max\_res) \;\wedge\; (rfree \neq \emptyset) \;\Rightarrow \\
& \forall rr \bullet (rr \in rfree \Rightarrow \\
& \qquad (rr \in rfree) \;\wedge \\
& \qquad (\mathbf{card}(rfree - \{rr\}) \leq max\_res)) \\
\equiv & true
\end{array}
$$

# The Boolean Substitution

To help the user of $rrms$ make use of $rec\_alloc$, the system provides an enquiry operation to see if there are any free resources.

$$
\begin{array}{l}
ack \longleftarrow is\_any\_free \;\;\widehat{=} \\
\textbf{BEGIN} \\
\quad ack := bool(rfree \neq \emptyset) \\
\textbf{END}
\end{array}
$$

where the Boolean substitution $x := bool(B)$ is defined by

$$
\boxed{
\begin{array}{l}
\textbf{IF}\ B \\
\textbf{THEN}\ x := TRUE \\
\textbf{ELSE}\ x := FALSE
\end{array}
}
$$

# Deallocate a Resource

The operation to free a resource is $rec\_free$. This has a single input $rr$, and the precondition says that $rr$ must be an allocated resource.

$$
\begin{array}{l}
rec\_free(rr) \;\;\widehat{=} \\
\textbf{PRE} \\
\quad rr \in RESOURCE - rfree \;\wedge \\
\quad \mathbf{card}(rfree) < max\_res \\
\textbf{THEN} \\
\quad free(rr) \\
\textbf{END}
\end{array}
$$

## An Enquiry Operation

The operation $is\_free$ takes a resource as input and outputs a Boolean value $ack$ to indicate whether the input is free.

$ack \longleftarrow is\_free(rr) \;\widehat{=}$
**PRE**
$\quad rr \in RESOURCE$
**THEN**
$\quad ack := bool(rr \in free)$
**END**

## The Back Up Operations

$rec\_backup$ makes a back up copy of the system by saving the value of $rfree$ in the variable $bkup.rfree$

$rec\_backup \;\widehat{=}$
**BEGIN**
$\quad bkup.setfree(rfree)$
**END**

$rec\_restore$ recovers the system by assigning the variable $rfree$ the value recorded in $bkup.rfree$.

$rec\_restore \;\widehat{=}$
**BEGIN**
$\quad setfree(bkup.rfree)$
**END**

## INCLUDES Clause: Summary

**MACHINE**
$\quad M_1(X_1, x_1)$
**CONSTRAINTS**
$\quad C_1$
$\quad \ldots$
**END**

**MACHINE**
$\quad M_2(X_2, x_2)$
**CONSTRAINTS**
$\quad C_2$
$\quad \ldots$
**END**

- **formal parameters of $M_i$ are actualised**
- **$M$ can access all variables, constants, sets of $M_i$**
- **$M$ can modify variables of $M_i$ via operations of $M_i$**
- **an operation in $M$ can call to at most ONE operation of $M_i$ (why?)**
- **Renaming**

**MACHINE**
$\quad M(X, x)$
**CONSTRAINTS**
$\quad C$
**SETS**
$\quad S; T = \{a, b\}$
**CONSTANTS**
$\quad c$
**PROPERTIES**
$\quad P$
**INCLUDES**
$\quad M_1(N_1, n_1), M_2(N_2, n_2)$
$\quad \ldots$
**END**

## The EXTENDS Clause

In the $rrms$ machine, the only operations are those defined in the **OPERATIONS** clause, namely $rec\_alloc$, $is\_any\_free$, $rec\_free$, $is\_free$ $rec\_backup$ and $rec\_restore$.

The operations of $rms$ and $bkup.rms$ are NOT operations of $rrms$.

**The EXTENDS clause allows a machine to be included in another, and at the same time makes**
**ALL ITS OPERATIONS**
**to be operations of the large machine.**

# The **PROMOTES** Clause

The **PROMOTES** clause allows some of the operations of the included machine to become operations of the large machine.

**INCLUDES**, **EXTENDS** and **PROMOTES** clauses are subject to the following rules:

1. **The same name cannot appear in both the INCLUDES and EXTENDS clauses.**
2. **For any operation name appears in the PROMOTES clause, its corresponding machine must appear in the INCLUDES clause.**

# The **USES** Clause

The **USES** clause is to let one machine have access to information in another before both are included in a large machine.

Consider a specification of the oil terminal system. We propose to present two small machines for its specification first.:

1. The $TankerM$ machine is used to manage the queue of tankers. It contains a deferred set $TANKER$. Its state is an injective sequence of tankers, i.e., a sequence without repetitions.
2. The $BerthM$ machine is used to manage the berths. It contains a deferred set $BERTH$ for the set of berths to be managed. Its state is a function relating berths and tankers, so it needs to use the set $TANKER$.

# Valdez Oil Terminal

# The $TankM$ Machine

**MACHINE**
    $TankerM$
**SETS**
    $TANKER$
**VARIABLES**
    $waiting$
**INVARIANT**
    $waiting \in \textbf{iseq}(TANKER)$
**OPERATIONS**
. . .
**END**

# The $BerthM$ Machine

**MACHINE**
$BerthM$
**SETS**
$BERTH$
**USES**
$TankerM$
**VARIABLES**
$docked$
**INVARIANT**
$docked \in BERTH \rightarrowtail\!\!\!\!\twoheadrightarrow TANKER \wedge$
$\mathbf{ran}(docked) \cap \mathbf{ran}(waiting) = \emptyset$
**OPERATIONS**
$\ldots$
**END**

**The USES clause allows $BerthM$ to use the set $TANKER$ and the variable $waiting$ in the invariant.**

# OilTerminalControl Machine

After these two machines have been separately presented, the specification of the oil terminal control system can be constructed by including both machines

**MACHINE**
$OTCS$
**INCLUDES**
$BerthM, TankerM$
**OPERATIONS**
$\ldots$
**END**

# The SEES Clause

The **SEES** clause allows one machine to access information in another that is to be *separately implemented*. Three kinds of machine appear in the **SEES** clause.

1. Stateless machines that define widely used types such as BOOL and INT.
2. Machines that define deferred and enumerated sets used in common by many machines in a development.
3. Mathematical context machines that define mathematical functions.

# A MathematicalContext Machine

**MACHINE**
$Mathfac$
**CONSTANTS**
$mathfac$
**PROPERTIES**
$mathfac \in NAT \rightarrow NAT \wedge$
$masthfac(0) = 1 \wedge$
$\forall n \bullet (n \in NAT - \{0\} \Rightarrow$
$mathfac(n) = n \times mathfac(n-1))$
**END**

# Native and Included Variables

**Native Variables:** defined in the **VARIABLES** clause.

They can appear in the invariant, and can be modified in the substitutions that define the operations.

**Included Variables:** defined in the machines named in the **EXTENDS** and **INCLUDES** clauses.

They can be used in the invariant, and can be referred to in the substitutions that define the operations.
**BUT they can only be modified by the operations of their own machines.**

# Used and Seen Variables

**Used Variables:** defined in the machines named in the **USES** clause.

They can be used in the invariant, and can be referred in to in the substitutions that define the operations.
**BUT they cannot be modified in the operations.**

**Seen Variables:** defined in the machines named in the **SEES** clause.

They can be referred to in the substitutions that define the operations.
**BUT they cannot be used in the invariant, nor modified in the operations.**

# Sets

Many machines have sets associated with them.

1. **Parametric Sets**: used as the parameters of the machine.
2. **Native Sets**: introduced in the **SETS** clause. They can be either deferred or enumerated.
3. **Included Sets**: the deferred and enumerated sets of the machines named in the **INCLUDES** and **EXTENDS** clauses.
4. **Used Sets**: the native and included sets of the machines named in the **USES** clause.
5. **Seen Sets**: the native and included sets of the machines named in the **SEES** clause.

# Constants

1. **Parametric Constants**: the parameters which are not sets. If they are not typed in the **CONSTRAINT** clause, they are assumed to be natural numbers.
2. **Native Constants**: introduced in the **CONSTANTS** clause, or the enumerated sets defined in the **SETS** clauses.
3. **Included Constants**: the native constants of the machines named in the **INCLUDES** and **EXTENDS** clauses.
4. **Used Constants**: the native constants of the machines named in the **USES** clause.
5. **Seen Constants**: the native constants of the machines named in the **SEES** clause.

# Constraints

The constraints of a machine are predicates about parameters, and they are introduced *explicitly* in the **CONSTRAINTS** clause.

Some constraints are *implicit* – these are the constraints that say **a parameter set is non-empty**.

# Context

The context of a machine is a set of predicates about the sets and constants.

1. For each native constant in the **CONSTANTS** clause, there is a constraint predicate, which must be introduced explicitly in the **PROPERTIES** clause. The deferred and enumerated sets CANNOT be typed or given values in the **PROPERTIES** clause.
2. An enumerated set can be given an implicit predicate. For example the enumerated set $BOOL$ has the following predicate in its context:
   $$BOOL \neq \emptyset$$
   $$BOOL = \{TRUE, FALSE\}$$
   $$FALSE \neq TRUE$$
   The set of implicit predicates and the content of the **PROPERTIES** clause form the **native context**.

# Invariant

1. **Native Invariant** : given in the **INVARIANT** clause, which fixes the types of native variables, and specifies the relationships of the native variables and the included and used variables. It cannot refer to the seen variables.
2. **Included Invariant** : the invariants of machines named in the **INCLUDES** and **EXTENDS** clause.
3. **Used Invariant** : the invariants of machines named in the **USES** clause.
4. **Seen Invariant** : the invariants of machines named in the **SEES** clause.

# Initialisation

1. **Native Initialisation** : given in the **INITIALISATION** clause.
2. **Included Initialisation** : the initialisations of machines named in the **INCLUDED** and **EXTENDS** clause.
3. **Used Initialisation** : the initialisations of machines named in the **USES** clause.

**The native, included and used initialisations MUST TOGETHER ESTABLISH the native invariant.**

# Operations

A machine includes three kinds of operations.

1. Operations of machines named in the **EXTENDS** clause.
2. Operations named in the **PROMOTES** clause.
3. Operations defined in the **OPERATION** clause.
   An operation definition can
   (a) refer to the native, included, used and seen variables, and any of the machine's sets and constants.
   (b) modify the native variables in any way.
   (c) change the included variables through the operations of the machines in which they are native.
   (d) use any of the operations of used and seen machines that do not modify variables.

   However, it CANNOT change used and seen variables.

# Summary

We have learnt

- how to include a machine in a large machine.
- how to use the operations of included machines in operations of including operations.
- the visibility and usability rules for the elements of machines in combination.

We will investigate some large examples, and show how to build the technical specification in an incremental way.

# An Invoice System (1)

We are going to specify a system capable of handling invoices for customers in a commercial environment.

**(1)** A **client** is recorded in the system together with his category and also with his maximum **allowance**. To each category of clients, there corresponds a certain **discount** applicable to the corresponding invoices.

**(2)** A **product** is recorded together with its **price**, its **status** (*available* or *sold out*), and its possible **substitute**, which is another product guaranteed not to be sold out.

# An Invoice System (2)

**(3)** An **invoice** is first concerned with the **client** to whom it is issued. An invoice also has a **discount** to be applied to the total of the invoice. Finally, it is characterised by the maximum amount of money that is *allowed* to it. The last two attributes are taken originally from the similar attributes of the client.

**(4)** Each **line** of an invoice is concerned with a certain **article**, the **quantity** and the **unit cost** of the article. The last attribute is taken originally from the **price** attribute of the product.

# Invoice System Operations

(1) Create and modify a client.

(2) Create and modify a product.

(3) Create and destroy an invoice.

(4) Add a new line to an invoice.

# Informal Requirements

(R1) A sold out product cannot be made part of an invoice.

(R2) If there exists a substitute for such a sold out product, the system must replace in the invoice the product in question by its substitute.

(R3) No two distinct lines of the same invoice may correspond to the same article.

(R4) No invoice can be made for *dubious* clients.

(R5) The discounted total of an invoice must not be greater than the maximum amount of money allocated for that invoice.

(R6) *Friend* get $20$ % discount, whereas others get no discount at all.

# Error Handling

The system may produce an error report when entering a new product in an invoice:

(1) the product might be sold out and there might be not corresponding substitute (see (R1) and (R2)).

(2) the product, or its substitute might be present already in the invoice, and there might be no more lines available in the system ((R2) and (R3)).

(3) the maximum allowance of the invoice would be reached by the introduction of the new product, or of its substitute ((R2), (R5) and (R6)).

# The Client Machine

The machine encapsulates the clients, which introduces the following two sets and one constant

(1) $CLIENT$ is a deferred set denoting all the possible (present and future) clients.

(2) $CATEGORY$ stands for the various categories of clients.

(3) $discount$ is a function linking each category of client to the corresponding percentage that will be applied to the total of their invoice.

# State of the $Client$ Machine

**MACHINE**
$Client$
**SETS**
$CLIENT$;
$CATEGORY = \{friend, dubious, normal\}$
**CONSTANTS**
$discount$
**PROPERTIES**
$discount \in CATEGORY \to (0..100) \wedge$
$discount = \{friend \mapsto 80, dubious \mapsto 100, normal \mapsto 100\}$

**VARIABLES**
$client, category, allowance$
**INVARIANT**
$client \subseteq CLIENT \wedge$
$category \in client \to CATEGORY \wedge$
$allowance \in client \to NAT$
**INITIALISATION**
$client, category, allowance := \emptyset, \emptyset, \emptyset$

# The $create\_client$ Operation

$create\_client$ has an input parameter $a : NAT$ representing the allowance given to the newly registered client, who is given *normal* as his category.

$c \longleftarrow create\_client(a) \ \widehat{=}$
**PRE**
$a \in NAT \ \wedge \ client \neq CLIENT$
**THEN**
**ANY**
$cc$
**WHERE**
$cc \in CLIENT - client$
**THEN**
$client := client \cup \{cc\} \parallel category(cc) := normal \parallel$
$allowance(cc) := a \parallel c := cc$
**END**
**END**

# More Operations

$c \longleftarrow read\_client \ \widehat{=}$
**PRE**
$client \neq \emptyset$
**THEN**
$c :\in client$
**END**

$modify\_category(c, k) \ \widehat{=}$
**PRE**
$c \in client \wedge$
$k \in CATEGORY$
**THEN**
$category(c) := k$
**END**

$modify\_allowance(c, a) \ \widehat{=} \ \ldots$

# The Product Machine

The $Product$ machine is used to encapsulate the product, and introduce its attributes.

**MACHINE**
$Product$
**SETS**
$PRODUCT$;
$STATUS = \{available, sold\_out\}$
**VARIABLES**
$product, price, status, substitute$

**INVARIANT**
$product \subseteq PRODUCT \wedge$
$price \in product \to NAT \wedge$
$status \in product \to STATUS \wedge$
$substitute \in product \rightarrowtail status^{-1}[\{available\}]$
**INITIALISATION**
$product, price, status, substitute := \emptyset, \emptyset, \emptyset, \emptyset$

# The $create\_product$ Operation

Function: to create a product. Parameter $c$ is supposed to denote the price of the product. The status of the new product is supposed to be *available*.

$p \longleftarrow create\_product(c) \ \widehat{=}$
**PRE**
  $c \in NAT \ \wedge product \neq PRODUCT$
**THEN**
  **ANY**
    $pp$
  **WHERE**
    $pp \in PRODUCT - product$
  **THEN**
    $price(pp) := c \ \| status(pp) := available \ \|$
    $product := product \cup \{pp\} \ \| p := pp$
  **END**
**END**

# More Operations

$make\_unavailable(p) \ \widehat{=}$
  **PRE**
    $p \in product$
  **THEN**
    $status(p) := sold\_out \ \|$
    $substitute := substitute \rhd \{p\}$
  **END**

$assign\_substitute(p, \ q) \ \widehat{=}$
  **PRE**
    $p \in product \ \wedge$
    $q \in product \ \wedge$
    $status(q) = available$
  **THEN**
    $substitute(p) := q$
  **END**

$modify\_price(p, \ c) \ \widehat{=} \ \ldots$
$p \longleftarrow read\_product \ \widehat{=}$

# The $Invoice$ Machine

The $Invoice$ machine

- **uses** $Client$ and $Product$ (because it needs to access some of the variables of these machines).
- needs two **sets** $INVOICE$ and $LINE$ for denoting all the possible invoices and lines.

# The $Invoice$ Machine (con'd)

**MACHINE**
  $Invoice$
**USES**
  $Client, \ Product$
**SETS**
  $INVOICE \, ;$
  $LINE$
**VARIABLES**
  $invoice, \ customer, percentage,$
  $allowed, \ total, line, \ origin,$
  $article, \ quantity, \ unit\_cost$

**INVARIANT**
  $invoice \subseteq INVOICE \ \wedge$
  $customer \in invoice \rightarrow client \ \wedge$
  $percentage \in invoice \rightarrow (0..100) \ \wedge$
  $allowed \in invoice \rightarrow NAT \ \wedge$
  $total \in invoice \rightarrow NAT \ \wedge$
  $\mathbf{ran}(total \otimes allowed) \subseteq \mathbf{leq} \ \wedge$
  $line \subseteq LINE \ \wedge \ origin \in line \rightarrow invoice \ \wedge$
  $article \in line \rightarrow product \ \wedge$
  $quantity \in line \rightarrow NAT \ \wedge$
  $unit\_cost \in line \rightarrow NAT \ \wedge$
  $origin \otimes article \in line \rightarrowtail\!\!\!\!\!\!\!\!\;+\;\, invoice \times product$

*where $(f \otimes g)(x) \ \widehat{=} \ (f(x), \ g(x))$, all the variables are set to $\emptyset$ initially, and $\rightarrowtail\!\!\!\!\!\!\!\!\;+\;$ denotes partial injective mapping*

# Creating an Invoice

This operation is designed to create an invoice for a given non-dubious client ((R4)).

$inv \longleftarrow create\_invoice\_header(c)\ \widehat{=}$

**PRE**

$c \in client\ \wedge category(c) \neq dubious\ \wedge invoice \neq INVOICE$

**THEN**

**ANY** $j$

**WHERE**

$j \in INVOICE - invoice$

**THEN**

$invoice := invoice \cup \{j\}\ \| customer(j) := c\ \|$

$percentage(j) := discount(category(c))\ \|$

$allowed(j) := allowance(c)\ \| inv := j$

**END**

**END**

# Adding a Line

It adds a line to an invoice when the product has not appeared yet.

$l \longleftarrow new\_line(i,\, p)\ \widehat{=}$

**PRE**

$i \in invoice\ \wedge p \in product\ \wedge status(p) = available\ \wedge$

$(i,\, p) \notin \mathbf{ran}(origin \otimes article)\ \wedge$

$line \neq LINE$

**THEN**

**ANY** $m$

**WHERE** $m \in LINE - line$

**THEN**

$l := m\ \|\ line := line \cup \{m\}\ \|$

$origin(m) := i\ \|\ article(m) := p\ \|$

$quantity(m) := 0\ \| unit\_cost(m) := price(p)$

**END**

**END**

# Obtaining a Line

The operation is to obtain the line of an available product, which is supposed to appear already in a certain line of the given invoice.

$l \longleftarrow the\_line(i,\, p)\ \widehat{=}$

**PRE**

$i \in invoice\ \wedge$

$p \in product\ \wedge$

$status(p) = available\ \wedge$

$(i,\, p) \in \mathbf{ran}(origin \otimes article)$

**THEN**

$l := (origin \otimes article)^{-1}(i,\, p)$

**END**

# Incrementing a Line

The operation is for incrementing a line, corresponding to an available product, with an extra quantity.

$increment(l,\, q)\ \widehat{=}$

**PRE**

$l \in line\ \wedge$

$q \in NAT\ \wedge$

$status(article(l)) = available\ \wedge$

$quantity(l) + q \in NAT\ \wedge$

$total(origin(l)) +$

$(q \times unit\_cost(l) \times percentage(origin(l))/100)$

$\leq\ allowed(origin(l)))$

**THEN**

$quantity(l) := quantity(l) + q\ \|$

$total(origin(l)) := total(origin(l)) +$

$(q \times unit\_cost(l) \times percentage(origin(l))/100)$

# Removing All Lines

$$
\begin{array}{l}
remove\_all\_lines(i) \ \widehat{=} \\
\textbf{PRE} \\
\quad i \in invoice \\
\textbf{THEN} \\
\quad line := line - origin^{-1}[\{i\}] \ \| \\
\quad origin := (origin^{-1}[\{i\}] \lhd origin) \ \| \\
\quad article := (origin^{-1}[\{i\}] \lhd article) \ \| \\
\quad quantity := (origin^{-1}[\{i\}] \lhd quantity) \ \| \\
\quad unit\_cost := (origin^{-1}[\{i\}] \lhd unit\_cost)
\end{array}
$$

where for a function $f$ and a set $S$ we define

$$
\boxed{
\begin{array}{lcl}
\mathbf{dom}(S \lhd f) & \widehat{=} & \mathbf{dom}(f) - S \\
(S \lhd f)(x) & \widehat{=} & f(x)
\end{array}
}
$$

# Removing an Invoice

The operation $remove\_invoice\_header$ is used to remove an invoice header.

$$
\begin{array}{l}
remove\_invoice\_header(i) \ \widehat{=} \\
\textbf{PRE} \\
\quad i \in invoice \wedge \\
\quad i \notin \mathbf{ran}(origin) \\
\textbf{THEN} \\
\quad invoice := invoice - \{i\} \ \| \\
\quad customer := \{i\} \lhd customer \ \| \\
\quad percentage := \{i\} \lhd percentage \ \| \\
\quad allowed := \{i\} \lhd allowed \ \| \\
\quad total := \{i\} \lhd total \\
\textbf{END}
\end{array}
$$

An invoice can be deleted **after removal of all its lines using the $remove\_all\_lines$ operation, i.e. $i \notin \mathbf{ran}(origin)$**.

# The $Invoice\_System$ Machine

The $Invoice\_System$ machine is just the joining together of all the previous machines. We add a number of operations to help in defining good error reporting.

$$
\begin{array}{l}
\textbf{MACHINE} \\
\quad Invoice\_System \\
\textbf{EXTENDS} \\
\quad Client, Product, Invoice
\end{array}
$$

$$
\begin{array}{l}
\textbf{OPERATIONS} \\
\quad b \longleftarrow some\_client\_exists \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow clients\_not\_saturated \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow client\_not\_dubious(c) \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow some\_product\_exists \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow products\_not\_saturated \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow product\_avaliable(p) \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow product\_has\_substitute(p) \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow invoices\_not\_saturated \ \widehat{=} \ \ldots; \\
\quad b \longleftarrow new\_product\_in\_invoice(i, p) \ \widehat{=} \ \ldots \\
\textbf{END}
\end{array}
$$

# A Multi-Lift Control System (1)

An $n$ lift system is to be installed in a building with $m$ floors. Design the logic to move lifts between floors according to the following rules:

(R1) Each lift has a set of buttons, one button for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the floor is arrived.

(R2) Each floor (except ground and top) has two buttons, one to request an up-lift and one to request a down-lift. They are illuminated when pressed. The buttons are cancelled when a lift visits the floor and is either travelling in the desired direction or visiting the floor with no requests outstanding. In the latter case, if both buttons are illuminated, only one should be cancelled.

# A Multi-Lift Control System (2)

(R3) When a lift has no requests to service, it should remain at its final destination with its door closed and await further request.

(R4) All requests for lifts from floors must be serviced eventually.

(R5) All requests for floors within lifts must be serviced eventually.

(R6) Each lift has an emergency button which, when pressed, causes a warning to be sent to the site manager. The lift is then deemed out of service. Each lift has a mechanism to cancel its out of service status.

# The $Lift$ Machine

We introduce two sets $LIFT$, which is a deferred set, and the enumerated set $DIRECTION$ in to the $Lift$ machine. We also define two constants, $top$ and $ground$, yielding the top and ground floors.

**MACHINE**
$\quad Lift$
**SETS**
$\quad LIFT\ ;$
$\quad DIRECTION \ = \ \{up,\ dn\}$
**CONSTANTS**
$\quad ground,\ top$
**PROPERTIES**
$\quad ground \in NAT \ \wedge top \in NAT \ \wedge top \ > \ ground$
**DEFINITIONS**
$\quad FLOOR \ \widehat{=} \ ground..top$

# Variables

1. Variable $moving$ is used to denote the set of moving lifts.
2. For each lift $l$, we have a corresponding floor: $floor(l)$, which is supposed to be the floor at which $l$ is stopped, if $l$ is not moving, or the floor the lift is about to arrive.
3. For each lift $l$, we have a direction, $dir(l)$, supposed to be the direction in which $l$ is travelling, if it is moving, or the direction in which the floor is intended to travel next, if it is not moving.
4. The variable $in$ is a binary relation from $FLOOR$ to $DIRECTION$. When $(f,\ d) \in in$, it means that some people want to travel from floor $f$ in direction $d$.
5. The variable $out$ is a binary relation from $LIFT$ to $FLOOR$. When $(l,\ f) \in out$, this means that some people in the lift $l$ want to leave $l$ at the floor $f$.

# Variables (Cont'd)

**VARIABLES**
$\quad moving,\ \ floor,\ \ dir,\ in,\ \ out$
**INVARIANT**
$\quad moving \subseteq LIFT \ \wedge$
$\quad floor \in LIFT \rightarrow FLOOR \ \wedge$
$\quad dir \in LIFT \rightarrow DIRECTION \ \wedge$
$\quad in \in FLOOR \leftrightarrow DIRECTION \ \wedge$
$\quad out \in LIFT \leftrightarrow FLOOR \ \wedge$
$\quad (ground \mapsto dn) \notin in \ \wedge$
$\quad (top \mapsto up) \notin in \ \wedge$
$\quad moving \lhd (out \cap floor) \ = \ \emptyset \ \wedge$
$\quad in \cap \mathbf{ran}(moving \lhd (floor \otimes dir)) \ = \ \emptyset$

**INITIALISATION**
$\quad in,\ out,\ moving := \emptyset,\ \emptyset,\ \emptyset \ ||$
$\quad floor,\ dir :=$
$\quad LIFT \times \{ground\},\ LIFT \times \{up\}$

# Request a Floor

This operation is used to correspond to the event of pressing buttons to request a floor inside a lift.

$Request\_Floor(l, f) \;\widehat{=}$
**PRE**
$\quad l \in LIFT \;\wedge$
$\quad f \in FLOOR \;\wedge$
$\quad (l \notin moving \Rightarrow (floor(l) \neq f))$
**THEN**
$\quad out := out \cup \{l \mapsto f\}$
**END**

# Request a Lift

This operation corresponds to actions of pressing a button to request a lift on a floor.

$Request\_Lift(f, d) \;\widehat{=}$
**PRE**
$\quad f \in FLOOR \;\wedge$
$\quad d \in DIRECTION \;\wedge$
$\quad (f, d) \neq (ground, dn) \;\wedge$
$\quad (f, d) \neq (top, up) \;\wedge$
$\quad (f, d) \;\notin\; \mathbf{ran}(moving \lhd (floor \otimes dir))$
**THEN**
$\quad in := in \cup \{f \mapsto d\}$
**END**

# The Control Events

We now define some events by which the system decides whether a moving lift, which is about to arrive a certain floor, has to continue moving or to stop at that floor. Predicate $attracted\_up(l)$ holds when a lift $l$ is situated or just arriving at $floor(l)$, and at least one of the following conditions holds:

- Some ones inside lift $l$ have expressed their intention to leave at some floor ahead of $floor(l)$ in the corresponding direction,

  $$out[\{l\}] \cap ((floor(l) + 1)...top) \neq \emptyset$$

- People are waiting for a lift at some floor ahead $floor(l)$ in the corresponding direction

  $$\mathbf{dom}(in) \cap ((floor(l) + 1)...top) \neq \emptyset$$

# The **DEFINITIONS** Clause

Predicate $attracted\_dn(l)$ is defined in a similar way.

**DEFINITIONS**
$\quad attracted\_up(l) \;\widehat{=}$
$$(\mathbf{dom}(in) \cup out[\{l\}]) \;\cap\; ((floor(l) + 1)...top) \;\neq\; \emptyset \;;$$

$\quad attracted\_dn(l) \;\widehat{=}$
$$(\mathbf{dom}(in) \cup out[\{l\}]) \;\cap\; (ground...(floor(l) - 1)) \;\neq\; \emptyset$$

# Control of Movement

$can\_continue\_up(l)$ and $can\_continue\_down(l)$ are supposed to hold when a moving lift $l$ has no reason to stop at $floor(l)$, where it is about to arrive. Obviously, this is when the following three conditions hold *simultaneously*:

(1) nobody wants to get out from $l$ at $floor(l)$

$$(l \mapsto floor(l)) \notin out$$

(2) nobody wants to get in from $floor(l)$ to travel in $dir(l)$

$$(floor(l) \mapsto dir(l)) \notin in$$

(3) lift $l$ is still attracted in the $dir(l)$ direction.

# Continue to Move Up

$$\begin{array}{l} can\_continue\_up(l) \;\widehat{=} \\ \qquad (l \mapsto floor(l)) \notin out \wedge \\ \qquad (floor(l) \mapsto dir(l)) \notin in \wedge \\ \qquad attracted\_up(l) \; ; \end{array}$$

$$\begin{array}{l} can\_continue\_dn(l) \;\widehat{=} \\ \qquad (l \mapsto floor(l)) \notin out \wedge \\ \qquad (floor(l) \mapsto dir(l)) \notin in \wedge \\ \qquad attracted\_dn(l) \; ; \end{array}$$

# Lift Control Operations

$Continue\_up(l) \;\widehat{=}$
**PRE**
$\quad l \in moving \wedge$
$\quad dir(l) = up \wedge$
$\quad can\_continue\_up(l)$
**THEN**
$\quad floor(l) := floor(l) + 1$
**END** ;

$Stop\_up(l) \;\widehat{=}$
**PRE**
$\quad l \in moving \wedge$
$\quad dir(l) = up \wedge$
$\quad \neg can\_continue\_up(l)$
**THEN**
$\quad moving := moving - \{l\} \parallel$
$\quad out := out - \{l \mapsto floor(l)\} \parallel$
$\quad in := in - \{floor(l) \mapsto dir(l)\}$
**END**

# Departure Operations

The decision for the departure of a lift from a floor in a certain direction or for the change of direction of a lift: The key idea is that a lift gives the priority to continuing its travel in the direction it was travelling when it stopped at the floor. If the lift has no reason to travel in the same direction then it is free to change to the opposite direction.

$Depart\_up(l) \;\widehat{=}$
**PRE**
$\quad l \in LIFT - moving \wedge$
$\quad dir(l) = up \wedge$
$\quad attracted\_up(l)$
**THEN**
$\quad moving := moving \cup \{l\} \parallel$
$\quad floor(l) := floor(l) + 1$
**END**

$Change\_up\_to\_dn(l) \;\widehat{=}$
**PRE**
$\quad l \in LIFT - moving \wedge$
$\quad dir(l) = up \wedge$
$\quad \neg attracted\_up(l) \wedge$
$\quad attracted\_dn(l)$
**THEN**
$\quad in := in - \{floor(l) \mapsto dn\} \parallel$
$\quad dir(l) := dn$
**END**

# Exercises

Define the following operations

(1) $Continue\_down(l)$
(2) $Stop\_dn(l)$
(3) $Depart\_dn(l)$
(4) $Change\_dn\_to\_up(l)$

# General Substitution (1)

(1) Simple Substitution

$[x := E]R \mathrel{\widehat{=}} R[E/x]$
$[skip]R \mathrel{\widehat{=}} R.$

(2) Multiple Simple Substitution

$[x, y := E, F]R \mathrel{\widehat{=}} R[E, F/x, y]$

(3) Precondition Substitution

$[\textbf{PRE}\ \ P\ \ \textbf{THEN}\ \ S\ \ \textbf{END}]R$

$\mathrel{\widehat{=}} [P \mid S]R \ (\iff (P \wedge [S]R))$

(4) Bounded Choice Substitution

$[\textbf{CHOICE}\ \ S\ \ \textbf{OR}\ \ T\ \ \textbf{END}]R$

$\mathrel{\widehat{=}} [S \,[]\, T]R \ (\iff ([S]R \wedge [T]R))$

# General Substitution (2)

(5) Guarded Substitution

$[\textbf{SELECT}\ \ P\ \ \textbf{THEN}\ \ S\ \ \textbf{END}]R$

$\mathrel{\widehat{=}} [P \Rightarrow S]R \ (\iff (P \Rightarrow [S]R))$

$\textbf{SELECT}\ P\ \textbf{THEN}\ S\ \textbf{WHEN}\ Q\ \textbf{THEN}\ T\ \textbf{END}$

$\mathrel{\widehat{=}} (P \Rightarrow S) \,[]\, (Q \Rightarrow T)$

(6) Conditional Substitution

$\textbf{IF}\ \ P\ \ \textbf{THEN}\ \ S\ \ \textbf{ELSE}\ \ T\ \ \textbf{END}$

$\mathrel{\widehat{=}} (P \Rightarrow S) \,[]\, (\neg P \Rightarrow T)$

$\textbf{IF}\ \ P\ \ \textbf{THEN}\ \ S\ \ \textbf{END}$

$\mathrel{\widehat{=}} \textbf{IF}\ P\ \textbf{THEN}\ \ S\ \ \textbf{ELSE}\ \ skip\ \ \textbf{END}$

# General Substitution (3)

(7) Unbounded Choice Substitution

$[\textbf{ANY}\ \ z\ \ \textbf{WHERE}\ \ P\ \ \textbf{THEN}\ \ S\ \ \textbf{END}]R$
$\mathrel{\widehat{=}} \forall z \bullet (P \Rightarrow [S]R)$

(8) Generalised Assignment

$[x :\in U]R \mathrel{\widehat{=}} \forall u \bullet (u \in U \Rightarrow R[u/x])$

(9) Boolean Substitution

$[x := \textbf{bool}(P)]R$

$\mathrel{\widehat{=}} P \wedge R[true/x] \vee \neg P \wedge R[false/x]$

(10) Update of Function

$(f(x) := E) \mathrel{\widehat{=}} (f := f \oplus \{x \mapsto E\})$

# Sequencing and Loop

(11) Sequencing

$[S;T]R \mathrel{\widehat{=}} [S]([T]R)$

(12) Loop

$[\mathbf{WHILE}\ P\ \mathbf{DO}\ S\ \mathbf{END}]$

$\mathrel{\widehat{=}} (P \Rightarrow S)\,\hat{}\ ;\ (\neg P \Rightarrow skip)$

where

$$T\,\hat{} \;=\; (T;T\,\hat{}\,)\;[]\;skip$$

# Refinement of Substitutions (1)

(13) Reduce Non-determinism.

$$S \;\widehat{=}\; (x := 0)\,[]\,(x := 1)$$
$$T \;\widehat{=}\; (x := 1)$$

where

$$\begin{aligned} [S]R &= (R[0/x] \;\wedge\; R[1/x]) \\ &\Rightarrow R[1/x] \\ &= [T]R \end{aligned}$$

(14) Weaken Pre-condition.

$$S \;\widehat{=}\; x > 5 \mid (x := y + 1)$$
$$T \;\widehat{=}\; x > 0 \mid (x := y + 1)$$

where

$$\begin{aligned} [S]R &= ((x > 5) \;\wedge\; R[y + 1/x]) \\ &\Rightarrow ((x > 0) \;\wedge\; R[(y + 1)/x]) \\ &= [T]R \end{aligned}$$

# Refinement of Substitutions (3)

(15) Refinement in both ways.

$$S \;\widehat{=}\; x > 5 \mid (x := 0)[](x := 1)$$
$$T \;\widehat{=}\; x > 0 \mid (x := 1)$$

where

$$\begin{aligned} (x > 0) \;\wedge\; R[1/x] &= [T]R \\ [S]R &= (x > 5) \;\wedge\; R[0/x] \;\wedge\; R[1/x] \\ &\Rightarrow (x > 0) \;\wedge\; R[1/x] \\ &= [T]R \end{aligned}$$

**Let $S$ and $T$ working with the same machine $M$ (with invariant $INV$). $S$ is said to be refined by $T$, denoted by $S \sqsubseteq T$ if**

$[S]R \Rightarrow [T]R$ **whenever** $R \Rightarrow INV$

# Monotonicity

(16) $S \sqsubseteq T \implies (P|S) \sqsubseteq (P|T)$

(17) $S \sqsubseteq T \implies (P \Rightarrow S) \sqsubseteq (P \Rightarrow T)$

(18) $S \sqsubseteq T \implies S\,\hat{} \sqsubseteq T\,\hat{}$

(19) $(U \sqsubseteq V) \wedge (S \sqsubseteq T) \implies (U[]S) \sqsubseteq (V[]T)$

(20) $(U \sqsubseteq V) \wedge (S \sqsubseteq T) \implies (U;S) \sqsubseteq (V;T)$

# Refining an Assignment

$$(P|(x :\in U)) \sqsubseteq T$$
$$\Longleftrightarrow$$
$$\forall x \bullet (P \Rightarrow [T](x \in U))$$

Let $E$ be an expression supposed to contain no occurrence of $x$.

$$(P|(x := E)) \sqsubseteq T$$
$$\Longleftrightarrow$$
$$\forall x \bullet (P \Rightarrow [T](x = E))$$

# Refinement and Invariant

$$\begin{array}{ll} (P|S) \textit{ preserves } INV & \\ \Longleftrightarrow & \forall x \bullet (INV \wedge P) \Rightarrow [P|S]INV \\ \Longleftrightarrow & \forall x \bullet (INV \wedge P) \Rightarrow [S]INV \\ \Longleftrightarrow & (INV \wedge P)|(x :\in \{m|INV(m)\}) \sqsubseteq S \end{array}$$

# Data Refinement

Let $NAT_1 \;\widehat{=}\; \{n \mid n \in NAT \;\wedge\; n \geq 1\}$. $\mathbf{FS}(NAT_1)$ is the power set of $NAT_1$.

**MACHINE**
Little-Example-1
**VARIABLES**
$y$
**INVARIANT**
$y \in \mathbf{FS}(NAT_1)$
**INITIALISATION**
$y := \emptyset$

**OPERATIONS**
$enter(n) \;\widehat{=}$
**PRE** $n \in NAT_1$
**THEN** $y := y \cup \{n\}$
**END**;
$m \longleftarrow maximum \;\widehat{=}$
**PRE** $y \neq \emptyset$
**THEN** $m := \mathbf{max}(y)$
**END**
**END**

# Loss of Information

**MACHINE**
Little-Example-2
**VARIABLES**
$z$
**INVARIANT**
$z \in NAT$
**INITIALISATION**
$z := 0$

**OPERATIONS**
$enter(n) \;\widehat{=}$
**PRE** $n \in NAT_1$
**THEN** $z := \mathbf{max}(z, n)$
**END**;

$m \longleftarrow maximum \;\widehat{=}$
**PRE** $z \neq 0$
**THEN** $m := z$ **END**
**END**

# External Substitutions

It is any generalised substitution that does not contain any explicit reference to the state variable, only calls to operations.

$$
\begin{array}{l}
prog \ \widehat{=} \\
\qquad enter(5); \\
\qquad enter(3); \\
\qquad x_1 \longleftarrow maximum; \\
\qquad enter(2); \\
\qquad enter(6); \\
\qquad x_2 \longleftarrow maximum
\end{array}
$$

Running $prog$ on each of the two machines we obtain the generalised substitutions $S$ and $T$ (on following slides)

# Substitutions $S$

$$
\begin{array}{l}
S \ \widehat{=} \\
\qquad y := \emptyset; \\
\qquad \textbf{PRE } 5 \in NAT_1 \textbf{ THEN } y := y \cup \{5\} \textbf{ END}; \\
\qquad \textbf{PRE } 3 \in NAT_1 \textbf{ THEN } y := y \cup \{3\} \textbf{ END}; \\
\qquad \textbf{PRE } y \neq \emptyset \textbf{ THEN } x_1 := \textbf{max}(y) \textbf{ END}; \\
\qquad \textbf{PRE } 2 \in NAT_1 \textbf{ THEN } y := y \cup \{2\} \textbf{ END}; \\
\qquad \textbf{PRE } 6 \in NAT_1 \textbf{ THEN } y := y \cup \{6\} \textbf{ END}; \\
\qquad \textbf{PRE } y \neq \emptyset \textbf{ THEN } x_2 := \textbf{max}(y) \textbf{ END}
\end{array}
$$

# Substitutions $T$

$$
\begin{array}{l}
T \ \widehat{=} \\
\qquad z := 0; \\
\qquad \textbf{PRE } 5 \in NAT_1 \textbf{ THEN } z := \textbf{max}(z, 5) \textbf{ END}; \\
\qquad \textbf{PRE } 3 \in NAT_1 \textbf{ THEN } z := \textbf{max}(z, 3) \textbf{ END}; \\
\qquad \textbf{PRE } z \neq 0 \textbf{ THEN } x_1 := z \textbf{ END}; \\
\qquad \textbf{PRE } 2 \in NAT_1 \textbf{ THEN } z := \textbf{max}(z, 2) \textbf{ END}; \\
\qquad \textbf{PRE } 6 \in NAT_1 \textbf{ THEN } z := \textbf{max}(z, 6) \textbf{ END}; \\
\qquad \textbf{PRE } z \neq 0 \textbf{ THEN } x_2 := z \textbf{ END}
\end{array}
$$

# Claim for $S$ and $T$

$$
\begin{array}{l}
[S](x_1, x_2 = 5, 6) \\
\Longleftrightarrow \\
true \\
\Longleftrightarrow \\
[T](x_1, x_2 = 5, 6)
\end{array}
$$

# Formal Definition

Let $M$ and $N$ be abstract machines with the same signature. The variables, named respectively $y$ and $z$, of these two machines are distinct. We suppose that these variables are members of sets $b$ and $c$ respectively.

The first machine $M$ is said to be refined by the second one, $N$, if each external substitution, working with its own variable $x$ supposed to be a member of $a$, and implemented on $M$ and $N$ in the form of the two respective substitutions $S$ and $T$, is such that the following holds

$$@y \bullet (y \in b \Rightarrow S) \sqsubseteq @z \bullet (z \in c \Rightarrow T)$$

(where @ is unbounded choice operator: $[@z \bullet A]B \mathrel{\widehat{=}} \forall z[A]B$)

# Sufficient Conditions

The definition of abstract machine refinement is not very helpful in practice because it is far too general.

Let $z = f(y)$ be a total function between $c$ and $b$.
Define a relation $W \;:\; (a \times c) \leftrightarrow (a \times b)$ by

$$W \;\widehat{=}\; \mathbf{id}(a) \| (z = f(y)).$$

Define for any set $R \subseteq (a \times c)$

$$[W]R \;\widehat{=}\; [z := f(y)]R \;=\; R[f(y)/z]$$

The proposed sufficient condition is:

**For all the operations $op$ (including the initialisation):**

$$(M.op \,;\, W) \;\sqsubseteq\; (W \,;\, N.op)$$

# Little-Examples

Let $z = \mathbf{max}(y \cup \{0\})$ be the relation between the states of $N$ and $M$. Define

$$[W]R \;=\; R[\mathbf{max}(y \cup \{0\})/z]$$

(1) Initialisation:
$(y := \emptyset; W) \sqsubseteq (W; z := 0)$

(2) $Entry$ operation:
$(n \in NAT_1 \mid y := y \cup \{n\}) \;;\; W$
$\sqsubseteq\; W \;;\; (n \in NAT_1 \mid z := \mathbf{max}(z,\, n))$

(3) $Maximum$ operation:
$(y \neq \emptyset \mid m := \mathbf{max}(y)) \;;\; W$
$\sqsubseteq\; W;\; (z \neq 0 \mid m := z)$

# Proof Obligation (1)

$$[y := \emptyset; W]R$$

$$\begin{aligned}
&= [y := \emptyset]([W]R)\\
&= [y := \emptyset]R[\mathbf{max}(y \cup \{0\})/z]\\
&= R[\mathbf{max}(\emptyset \cup \{0\})/z]\\
&= R[0/z]\\
&= [z := \mathbf{max}(y \cup \{0\})](R[0/z])\\
&= [z := \mathbf{max}(y \cup \{0\})]([z := 0]R)\\
&= [W; (z := 0)]R
\end{aligned}$$

# Proof Obligation (2)

$[M.entry(n); W]R$

$= [M.entry(n)]([W]R)$
$= [M.entry(n)](R[\mathbf{max}(y \cup \{0\})/z])$
$= n \in NAT_1 \ \wedge\ R[\mathbf{max}(y \cup \{n\} \cup \{0\})/z]$
$= n \in NAT_1 \ \wedge\ R[\mathbf{max}(y \cup \{n\})/z]$
$= [z := \mathbf{max}(y \cup \{0\})](n \in NAT_1 \ \wedge\ R[\mathbf{max}(z, n)/z])$
$= [z := \mathbf{max}(y \cup \{0\})]([N.entry(n)]R)$
$= [W; N.entry(n)]R$

# Local Variables

Swapping $x$ and $y$:

$t := x; x := y; y := t$

$t$ is introduced only for this swapping. So, the variable $t$ is declared to be local:

**VAR** $t$ **IN** $t := x; x := y; y := t$ **END**

(Note: $t$ is assigned some value (x) before it is read (to $y$)

**VAR clause:**

**VAR** $t_1, \ldots, t_n$ **IN** $S$ **END**

**Predicate Substitution for VAR substitutions**

$\boxed{[\mathbf{VAR}\ t_1, \ldots, t_n\ \mathbf{IN}\ S\ \mathbf{END}]P = \forall t_1, \ldots, t_n [S]P}$

# Data Refinement Machines

- **AMN (Abstract Machine Notations):** Specification machines describe functional requirements; users of such machines should understand the behaviour of these machines in term of the specification. User do not need to know about how the information is represented and handled in such machines.
- **Refinement machines:** a refinement machine describes the design decisions taken so far with regard to a particular specification. It describes the way that the abstract information is represented by means of a **linking invariant** relating the abstract states to the refinement states.
- A refinement machine should have the same interface as the machine it refines



**MACHINE** $Team$
**SETS** $ANSWER = \{in, out\}$
**VARIABLES** $team$
**INVARIANT** $team \subseteq 1..22 \wedge$
    **card**$(team) = 11$
**INITIALISATION** $team := 1..11$
**OPERATIONS**
$substitute(pp, rr) \mathrel{\widehat{=}}$
  **PRE** $pp \in 1..22 \wedge rr \notin team$
  **THEN** $team := (team \cup \{rr\}) - \{pp\}$
  **END**;
$aa \longleftarrow query(pp) \mathrel{\widehat{=}}$
  **PRE** $pp \in 1..22$
  **THEN**
  **IF** $pp \in team$   **THEN** $aa := in$
  **ELSE** $aa := out$ **END**
  **END**
**END**

**REFINEMENT** $TeamR$
**REFINES** $Team$
**VARIABLES** $teamr$
**INVARIANT**
    $teamr \in 1..11 \rightarrowtail 1..22 \wedge$
    **ran**$(teamr) = team$
**INITIALISATION**
  $teamr := \lambda nn.(nn \in 1..11 | nn)$
**OPERATIONS**
$substitute(pp, rr) \mathrel{\widehat{=}}$

  $teamr(teamr^{-1}(pp)) := rr;$
$aa \longleftarrow query(pp) \mathrel{\widehat{=}}$
  **IF** $pp \in$ **ran**$(teamr)$
  **THEN** $aa := in$
  **ELSE** $aa := out$
  **END**
**END**

# Refinement Machines

A Refinement Machine

- Has the machine name given as in **REFINEMENT** $TeamR$.
- May have its own variable ($teamr$), and must describe the relationship between its own state and the state of the machine it refines ($Team$) as the linking invariant (e.g. $team = \textbf{ran}(teamr)$)
- Has the same interface as the machine it refines ($TeamR$ and $Team$ have the same operations $substitute$ and $query$ with the same input/output parameters).
- Operations in the refenement machine should refines coresponding operations in the machine it refines under invariants of the two machines.

A refinement machine may make use of other abstract machines through **INCLUDES**, **PROMOTES** and **SEES**, but only abstract machines may be referred to (not refinement or implelentation ones)

# A Specification Machine

**MACHINE** $Exam$
**SETS** $CANDIDATE$
**VARIABLES** $marks$
**INVARIANT**
$marks \in CANDIDATE \nrightarrow 1..100$
**INITIALISATION** $marks := \emptyset$
**OPERATIONS**
$enter(cc, nn) \mathrel{\widehat{=}}$
  **PRE** $cc \in CANDIDATE \wedge$
  $cc \notin \textbf{dom}(marks) \wedge nn \in 0..100$
  **THEN** $marks(cc) := nn$
  **END**;

$aa \longleftarrow average \mathrel{\widehat{=}}$
  **PRE** $marks = \emptyset$
  **THEN** $aa :=$
  $\sum zz.(zz \in \textbf{dom}(marks) |$
  $marks(zz))/$
  $\textbf{card}(\textbf{dom}(marks))$
  **END**;
$nn \longleftarrow number \mathrel{\widehat{=}}$
  $nn := \textbf{card}(\textbf{dom}(marks))$
  **END**

# and Its Refinement Machine

**REFINEMENT** $ExamR$
**REFINES** $Exam$
**VARIABLES** $total, num$
**INVARIANT**
  $num = \textbf{card}(\textbf{dom}(mark)) \wedge$
  $total = \sum zz.(zz \in \textbf{dom}(marks) | marks(zz))$
**INITIALISATION**
  $total := 0; num := 0$
**OPERATIONS**
$enter(cc, nn) \mathrel{\widehat{=}}$
  **BEGIN**
  $total := total + nn || num := num + 1$
  **END**;
$aa \longleftarrow average \mathrel{\widehat{=}} aa := total/num;$
$nn \longleftarrow number \mathrel{\widehat{=}} nn := num$
**END**

# Implementaion Machines

- The AMN clauses for implementaion machines:

  **IMPLEMENTATION** $M2$
  **IMPORTS** $M1$

  A machine can only be imported by one implementation.
- An implementation machine does not have any variables listed in a **VARIABLES** clause (any state needed to be maintained by the implementation must be kept in imported machines.
- The **INVARIANT** clause contains a linking invariant between the imported state variables and the variables of the machine refined by the implementation.
- Statements in implementaion machines are restricted (to executable statements: simple asignments, sequential composition, conditional, case statements, while loop, use of local variables, operations of imported machines and query operations of seen machines) . So they do not have preconditions.

**Example**: Specification machine $Custumer$, implementation machine $CustumerI$ and imported machine $Set$.

# A Specification Machine

**MACHINE** $Custumer$
**SEES** $Price, Goods$
**CONSTANTS** $limit$
**PROPERTIES**
$limit \in GOODS \rightarrow NAT$
**VARIABLES** $purchases$
**INVARIANT** $purchases \subseteq GOODS$
**INITIALISATION** $purchases := \emptyset$

**OPERATIONS**
$pp \longleftarrow buy(gg) \widehat{=}$
  **PRE**
$gg \in GOODS \wedge price(gg) \leq limit(gg)$
  **THEN** $purchases \ := \ purchases \cup gg ||$
    $pp \longleftarrow pricequery(gg)$
  **END**
**END**

# and Its Implementation Machine

**IMPLEMENTATION** $CustumerI$
**REFINES** $Custumer$
**SEES** $Price, Goods$
**IMPORTS** $Set(GOODS)$
**INVARIANT** $set = purchases$
**OPERATIONS**
$pp \longleftarrow buy(gg) \widehat{=}$
  **BEGIN**
    $pp \longleftarrow pricequery(gg);$
    **IF** $pp \leq limit(gg)$
    **THEN** $add(gg)$
  **END**
**END**

**MACHINE** $Set(ELEM)$
**VARIABLES** $set$
**INVARIANT** $set \subseteq ELEM$
**INITIALISATION** $set := \emptyset$
**OPERATIONS**
$add(ee) \widehat{=}$
  **PRE** $ee \in ELEM$
  **THEN** $set := set \cup ee$
  **END**
**END**

# ProB: A Model Checker for B

ProB provides two ways of **modelchecking**

- **Temporal model checking:** try to find a sequence of operations that leads to a state that violates the invariant from the initial state (counter-example)
- **State-based model checking (constraint-based checking):** try to find a state that satisfies the invariant, but where can apply a single operation to reach a state that violates the invariant

# ProB: An Animator for B

- Provide immediate visual feedback about one's specification
- Provide a good feel for the B language
- Easily to find problems with one's specifications, errors are easily identified by ProB

# Animator Menu in ProB

- Open lift.mch with ProB, experiment with:
  - syntax check, syntax error reports
  - enabled operations whose preconditions and guards are satisfiable at the current state that can be selected for execution (deadlock states have no enabled operation
  - system traces
  - invariant violation
  - different views of system states, state space
- Preference settings

# Using Temporal Model Checker

- Checking for **invariant violations**
- Checking for **deadlock**
- For a machine with invariant violations AND deadlock, ONLY invariant violation is reported for the first time, and we have to invoke TMC again to discover deadlock.
- TMC is by exausted search, so we have to provide **the number of nodes** "max nr" to limit the search space, and the result
- **Inspect Existing Node** option (on/off) are to be used in combination with Check for Deadlock and Check for Invariant Violation in Two Phase Testing way.

(experiment with invoice.mch and lift.mch!)

# Advanced Features

Specifying GOALs

- Through the use of GOAL pragma in the DEFINITIONS section, specified with B syntax as
  `GOAL == (padlock={} & box_contains_gem=TRUE & hasbox=natasha)`
- Checking if a state satisfying the GOAL is reachable, and showing a sequence of operations leading to the state
- Example: Russian postal puzzle

# Russian Postal Puzzle

**Assumption:** Russian Postal System is corrupt, packages containing valuables are stolen. Padlocked boxes can prevent items, but keys can be stolen.

**How can Boris send a diamond ring to Natasha?**

- Boris places the ring in a box and locks it, send the box to Natasha and retains his key
- Natasha receives the box, puts another padlock to the box and retains her key, sends it back to Boris
- Boris remove his padlock, send the box to Natasha
- Natasha then removes her lock and obtains the ring

Write a B machine for the protocol and check if
`GOAL == (padlock={} & box_contains_gem=TRUE & hasbox=natasha)`
is reachable

# Typing in ProB

- Any variable or constant must be given a type for ProB to function properly
- Basic type in B (written in ProB): `BOOL, INT, POW(`$\tau$`), A*B, POW(A*B), POW(INT*A)`
- To declare typing: "var: Type"
  `x<:S` instead of `x:POW(S)`
  `x: A<->B` gives `x<:POW(A*B)`

# Controlling Animation

Animating and verifying B is undecidable in principle. ProB restricts them to finite sets and finite ranges of numbers.

- Enumeration
- Integer
- Sets

# Constraint Based Checking

- Checking whether applying an individual operation can result in an invariant violation, independently of the particular initialisation of the B machine (done by symbolic constraint solving).
- Model checking tries to find a reachable state violating an invariant, however Constraint Based Checking tries to find a state violating an invariant that reachable from an another state (which may not be or reachable from an initial state).
- Constraint Based Checking shows the existence of a model which makes the formula false, and we are not able to use the proof rules for B to prove the correctness of the machine if such a model found!

# Current Limitations of ProB

- `Definitions` with argument are not supported
- Multiple Machines are not fully supported
- Some constructs and type (e.g. STRING) that are not B standard are not supported.

# ProB Website

http://www.stups.uni-duesseldorf.de/ProB/

ProB is maintained by Michael Leuschel. It is based on research and implementation effort by Michael Leuschel, Michael Butler, Carla Ferreira, Leonid Mikhailov, Edward Turner, Phil Turner, and Laksono Adhianto. Part of the research and development was conducted within the EPSRC funded projects ABCD and iMoc.

Several versions of ProB are available for download.

# References


- **The B-Book:** Assigning Programs to Meanings, J.-R. Abrial, Cambridge University Press, 1996. ISBN 0-521-49619-5. 850 pages.
  - **Contents:** Mathematical reasoning; Set notation; Mathematical objects; Introduction to abstract machines; Formal definition of abstract machines; Theory of abstract machines; Construction large abstract machines; Example of abstract machines; Sequencing and loop; Programming examples; Refinement; Construction large software systems; Example of refinement;
  - **Appendices:** Summary of the most current notations; Syntax; Definitions; Visibility rules; Rules and axioms; Proof obligations.
- **The B-Method: An Introduction**, Steve Schneider, Palgrave, Cornerstones of Computing series, October 2001. ISBN 0-333-79284-X.