Formal Software Engineering

The B Method
for correct-by-construction software

J. Christian Attiogbé

November 2012

UNIVERSITÉ DE NANTES

NANTES

The B Method: an introduction

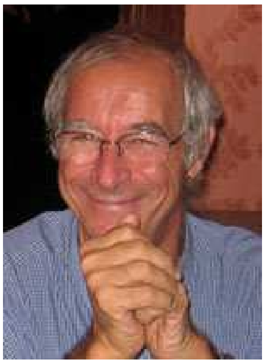
- Introduction: what it is?
    - A quick overview
    - Example of specification
        - Light Control in a Room
- How to develop using B
    - System Analysis
    - Structuration: Abstract Machines
    - Modeling Data
    - Modeling Data Operations
    - Refinements
    - Implementation

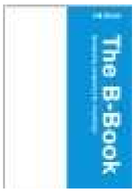
Examples of development

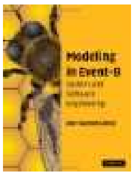
- Exemples
  - GCD (PGCD), euclidian division,
  - Sorting
- Basic concepts of the method
  - Modeling the static part (data)
  - Modeling the dynamic part(operations)
  - Proof of consistency
  - Refinement
  - Proofs of refinement
- Case studies (with AtelierB, Rodin)

B Method

- (..1996) A Method to specify, design and build sequential software.
- (1998..) Event B ... distributed, concurrent systems.
- (...) still evolving, with more sophisticated tools (aka Rodin) ;-(







Examples of application in railways systems

Figure: Synchronisation of platform screen doors - Paris Metro

Industrial Applications

- Applications in Transportation Systems (Alsthom>Siemens) braking systems, platform screen doors(line 13, Paris metro),
- KVS, Calcutta Metro (India), Cairo
- INSEE (french population recensement)
- Meteor RATP : automatic pilote, generalization of platform screen doors
- SmartCards (*Cartes à puce*) : securisation, ...
- Peugeot
- etc

☞ Highly needed competencies in Industries.

A Context that imposes Formal Method

The standard EN51128 "Systèmes de signalisation, de télécommunication et de traitement" :
Cette norme traite en particulier des méthodes qu'il est necessaire d'utiliser pour fournir des logiciels répondant aux exigences d'intégrité de la sécurité appliquées au domaine du ferroviaire. L'intégrité d'un logiciel est répartie sur cinq niveaux SIL, allant de SIL 0 à SIL 4. Ces niveaux SIL sont définis par association, dans la gestion du risque, de la fréquence et de la conséquence d'un événement dangereux. Afin de définir précisément le niveau de SIL d'un logiciel, des techniques et des mesures sont définies dans cette norme.
(cf. ClearSy)

SIL : Safety Integrity Level

Introduction to B

The Standard EN 50128 : Software Aspect of the Control

Standard NF EN 50128
Titre : Railway Applications, system of signaling, telecommunication and processing equipped with software for the control and the security of railway systems.
Domain: Exclusively applicable to software and to the interaction between software and physical devices;
5 levels of criticity:
Not critical: SIL0,
No dead danger for humans: SIL1, SIL2,
Critical : SIL3, SIL4
Applicable to: the software application; the operating systems ; the CASE[1] tools;

Depending on the projects and the contexts, we will need formal methods to build the dependable software or systems.

Method in Software Engineering

Formal Method=

- Formal Specification or Modeling Langagauge
- Formal reasoning System

B Method=

- Specification Language
  - Logic, Set Theory: data language
  - Generalized Substitution Language: Operations's language
- Formal reasoning System
  - Theorem Prover

Introduction to B

Formal Development

Formel Software Development=

- Systematic transformation of a mathematical model into an executable code.

  = Transformation from the abstract to the concrete

  = Passing from mathematical structures to programming structures

  = Refinement into code in a programming language.

B: Formal Method
  + refinement theory (of abstract machines)
$\Rightarrow$ formal development method

Correct Development (no overflow, for a trajectory)

```
MACHINE
    CtrlThreshold /* to control two naturals X and Y */
            /* 0 <= x <= threshold
            ∧ ∀ y . 0 < y < threshY */
CONSTANTS     threshX, threshY
PROPERTIES threshX : INT & threshX = 10 ...
VARIABLES
        xx, yy
INVARIANT
        xx : INT & 0 <= xx & xx <= threshX
        yy : INT & 0 < yy & yy < threshY
INITIALISATION     xx := 0 || yy := 1
OPERATIONS
        computeY =
            yy := ... /* an expression */
END
```


Correct Development....

```
OPERATIONS (continued)
setXX(nx) = /* specification of an operation with PRE */
PRE
    nx : INT & nx >= 0 & nx <= threshX
THEN
    xx := nx
END ;

rx <-- getXX = /* specification of an operation */
BEGIN
    rx := xx
END
```

The GCD Example

From the abstract machine to its refinement into executable code.

mathematical model –> programming model

Constructing the GCD: abstract machine

```
MACHINE
    pgcd1 /* the GCD of two naturals */
            /* gcd(x,y) is d | x mod d = 0 ∧ y mod d = 0
            ∧ ∀ other divisors dx d > dx
            ∧ ∀ other divisors dy d > dy */
OPERATIONS
        rr <-- pgcd(xx,yy) = /* OUTPUT : rr ; INPUT xx, yy */
            ...
END
```

Constructing the GCD: abstract machine

```
OPERATIONS
rr <-- pgcd(xx,yy) = /* spécification du pgcd */
PRE
    xx : INT & xx >= 1 & xx < MAXINT
& yy : INT & yy >= 1 & yy < MAXINT
THEN
    ANY dd WHERE
    dd : INT
    & (xx - (xx/dd)*dd) = 0 /* d is a divisor of x */
    & (yy - (yy/dd)*dd) = 0 /* d is a divisor of y */
        /* and the other common divisors are < d */
    & !dx.((dx : INT & dx < MAXINT
        & (xx- (xx/dx)*dx) = 0 & (yy-(yy/dx)*dx)=0) => dx < dd)
    THEN rr := dd
    END
END
```


Constructing the GCD: refinement

```
REFINEMENT /* raffinement de ...*/
    pgcd1_R1
REFINES pgcd1 /* the former machine */
OPERATIONS
rr <-- pgcd (xx, yy) = /* the interface is not changed */
    BEGIN
        ... Body of the refined operation
    END
END
```

Constructing the GCD: refinement

```
rr <-- pgcd (xx, yy) = /* the refined operation */
     BEGIN
          VAR cd, rx, ry, cr IN
               cd := 1
               ; WHILE ( cd < xx & cd < yy) DO
                    ; rx := xx - (xx/cd)*cd ; ry := yy - (yy/cd)*cd
                    IF (rx = 0 & ry = 0)
                    THEN /* cd divises x and y, possible GCD */
                         cr := cd /* possible rr */
                    END
                    ; cd := cd + 1 ; /* searching a greater one */
               INVARIANT
                    xx : INT & yy : INT & rx : INT & rx < MAXINT
                    & ry : INT & ry < MAXINT & cd < MAXINT
                    & xx = cr*(xx/cr) + rx & yy = cr*(y/cr) + ry
               VARIANT
                    xx - cd
               END
```



```
     END
```


B Method: Global Approach

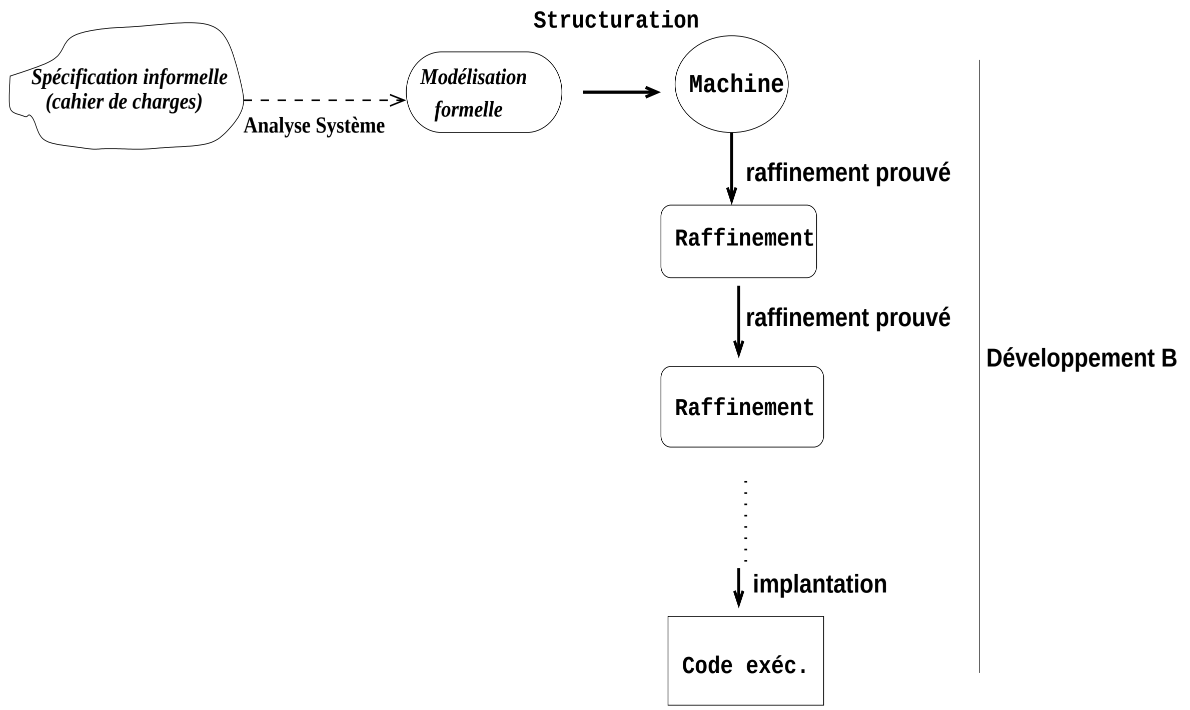


Figure: Analysis and B development

The B Method

Concepts and basic principles :

- abstract machine (state space + abstract operations),
- proved refinement (from abstract to concrete model)

State and State Space

- Observe a variable in a logical model;
- It can take different values through the time, or several states through the time;
- For example a natural variable I: one can (logically) observe $I = 2$, $I = 6$, $I = 0$, $\cdots$ provided that I is modified;
- Following a modification, the state of I is changed;
- The change of states of a variable can be modeled by an action that substitutes a new value to the current one.
- More generally, for a natural I, there are possibly all the range or the naturals as the possible states for I: hence the state space.
- One generalises to several variables $\langle I, J \rangle$, $\langle V1, V2, V4, ... \rangle$

Development Approach

The approaches of Z, TLA, B, ... are said: model (or state) oriented

- Describe a state space
- Describe operations that explore the space
- Transition system between the states

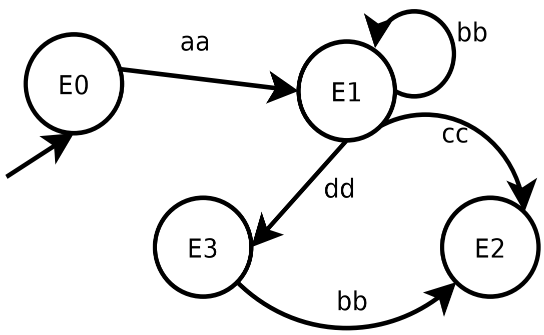


Figure: Evolution of a software system

Specification Approach

- A tuple of variables describes a state

$$\langle mode = day,\ light = off,\ temp = 20 \rangle$$

- A predicate (with the variables) describes a state space

$$light = off \wedge mode = day \wedge temp > 12$$

- An operation that affects the variables changes the state

$$mode := day$$

Specification in B = model a transition system
(with a logical approach)

Abstract Machine

variables

predicates

operation

```
MACHINE ...
SETS ...
VARIABLES
...
INVARIANT
... predicates
INITIALISATION
...
OPERATIONS
...
END
```


Abstract Machine

```
MACHINE ReguLight
SETS
DMODE = {day, night}
; LIGHTSTATE = {off, on}
```

- An abstract machine has a name
- The SETS clause enables ones to introduce abstract or enumerated sets;
  These sets are used to type the variables
- The predefined sets are : NAT, INTEGER, BOOL, etc

Abstract Machine

```
VARIABLES
mode
, light
, temp
INVARIANT
mode : DMODE
& light : LIGHTSTATE
& temp : NAT
```

- The VARIABLES clause gathers the variables to be used in the specification
- The INVARIANT clause is used to give the predicate that describe the invariant properties of the abstract machine; its should be always true
- Both clauses go together.

Abstract Machine

```
INITIALISATION
mode := day
|| temp := 20
|| light := off
```

- An abstract machine should contain, an initial state of the specified system.
  This initial state should ensures the invariant properties.
  The INITIALISATION clause enebales one to initialise ALL the variables used in the machine
  The initialisation using substitutions, is done simultaneaously for all the variables.
  They can be modified later by the operations.

Abstract Machine

```
OPERATIONS
changeMode =
CHOICE mode := day
OR mode := night
END
;
putOn =
light := on
;
putOff =
light := off
;
decreaseTemp = temp := temp - 1
;
increaseTemp = temp := temp +1
END
```

- Within the clause OPERATIONS one provides the operations of the abstract machine.
  The operations model the change of state variables with logical substitutions (noted :=).
  The logical substitutions are generalised for more expressivity.
  The operations has a PREcondition (the POST is implicitely the invariant).

Abstract Machine : example of Light Regulation

```
MACHINE ReguLight
SETS
DMODE = {day, night}
; LIGHTSTATE = {off, on}

VARIABLES
mode
, light
, temp
INVARIANT
mode : DMODE
& light : LIGHTSTATE
& temp : NAT

INITIALISATION
mode := day || temp := 20
|| light := off
```

```
OPERATIONS
changeMode =
CHOICE mode := day
OR mode := night
END
;
putOn =
light := on
;
putOff =
light := off
;
decreaseTemp = temp := temp - 1
;
increaseTemp = temp := temp +1
END
```

Abstract Machine: provides operations

An abstract machine provides operations which are callable from other external operations/programmes.

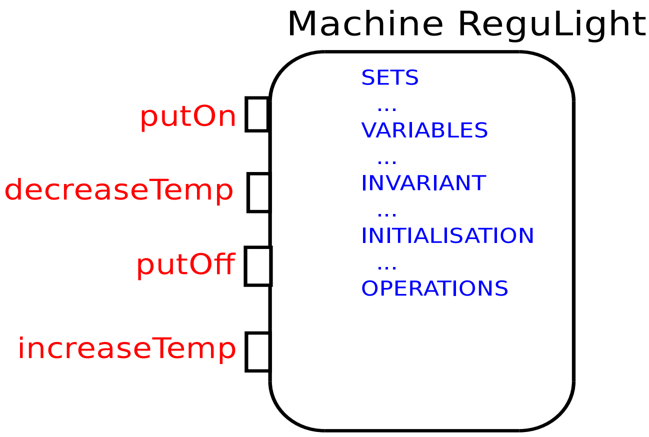


Figure: The operations are callable from outside

☛ An operation of a machine cannot call another operation of the same machine

Introduction to B

Interface of operations

(operations with or without input/output parameters)

- No parameter:

  nameOfOperation = ...
- Input parameters only:

  nameOfOperation(p1, p2, $\cdots$) = ...
- Output parameters only:

  r1, r2, $\cdots$ <— nameOfOperation = ...
- Input and Output parameters:

  r1, r2, $\cdots$ <— nameOfOperation(p1, p2, $\cdots$) = ...

Light Regulation System

Study

Requirements:

- The light should not be on during daylight.
- The temperature should not exceed 29 degrees during daylight.
- ...

⇒ Find and formalise the properties of the invariant.

Abstract Machine: example of the gauge

```
MACHINE MyGauge
VARIABLES
gauge
INVARIANT
gauge : NAT
&   gauge >= 2
& gauge <= 45
INITIALISATION
gauge := 1 // !! what?
```

```
OPERATIONS
decrease1 =
PRE gauge > 2
THEN gauge := gauge - 1
END
; decrease(st) =
PRE st : NAT
& gauge - st >= 2
THEN
gauge := gauge - st
END
...
increase ...
...
END
```

Abstract Machine: example of ressources

```
MACHINE
Resrc
SETS
RESC
CONSTANTS
maxRes // a parameter
PROPERTIES
maxRes : NAT & maxRes > 1
VARIABLES
rsc
INVARIANT
rsc <: RESC // a subset
&  card(rsc) <= maxRes //bound
INITIALISATION
rsc := {}
```

```
OPERATIONS
addRsc(rr) = // adding
PRE
rr : RESC & rr /: rsc &
card(rsc) < maxRes
THEN
rsc := rsc \/ {rr}
END
;
rmvRsc(rr) = // removing
PRE
rr : RESC & rr : rsc
THEN
rsc := rsc - {rr}
END
END
```


Basics of correct program construction

Consider operations on a bank account:

- a withdrawal of `givenAmount`

```
begin
   account := account - givenAmount
end
```

- a deposit on the account of `newAmount`

```
begin
   account := account + newAmount
end
```

☞ these operations are not satisfactory, they don't take care of the constraints (the threshold to not overpass).

Basics of correct program construction

- a withdrawal `givenAmount`

```
withdrawal(account, givenAmount)=
pre
   account - givenAmount >= 0 //unauthorised overdraft
begin
   account := account - givenAmount
end
```

☛ Before calling the operation, we should ensure that it does not overpass the autorised amount.

```
IF withdrawalPossible(account, givenAmount)
   THEN withdrawal(account, givenAmount)
END
```


Basics of correct program construction (before B)

Consider two naturals `natN` and `natD`.
What happens with the following statement?

```
res := natN / natD
```

What was expected:

```
IF (natD /= 0)
  THEN res := natN / natD
END
```

Indeed, the division operation has a precondition : (denom /= 0)

B: principle of the method

The control with an invariant of a system (or of a software)

- one models the space of correct states with a property (a conjunction of properties).
- While the system is in these states, it runs safely; it should be maintain within these states!
- We should avoid the system going out from the state space
- Hence, be sure to reach a correct state before performing an operation.

Examples: trajectory of a robot (avoid collision points before moving).

☛ The operations that change the states has a precondition.

B: logical approach

Originality of B: every thing in logics (data and operations)

- state space: Invariant: Predicate : $P(x, y, z)$
  A state: a valuation of variables
  $x := v_x \;\; y := v_y \;\; z := v_z$ in $P(x, y, z)$
  $\Rightarrow$ Logical substitution
- An operation: transforms a correct (state) into another one.

  Transform a state = predicate transformer (invariant)
  Operation = predicate transformer = substitution
  other effects than affectation $\Rightarrow$ generalized substitutions

B: the practice

A few specification rules in B

- An operation of a machine cannot call another operation of the same machine (violation of PRE);
- One cannot call in parallel from outside a machine two of its operation (for example : incr || decr) ;
- A machine should contain auxilliary operations to check the preconditions of the principal provided operations;
- The caller of an operation should check its precondition before the call ("One should not divide by 0") ;
- During refinement, PREconditions should be weaken until they desappear(Be careful, this is not the case with Event-B) ;
- ...

B: the foundations

- First Order Logic
- Set Theory (+ types)
- Theory of generalized substitutions
- Theory of refinement
- and a good taste of: abstraction and composition!

B: CASE Tools

- Modularity:
  Abstract Machine, Refinement, Implementation
- Architecture of complex applications:
  with the clauses **SEES**, **USES**, **INCLUDES**, **IMPORTS**, ...
- CASE:
  Editors, analysers, provers, ...

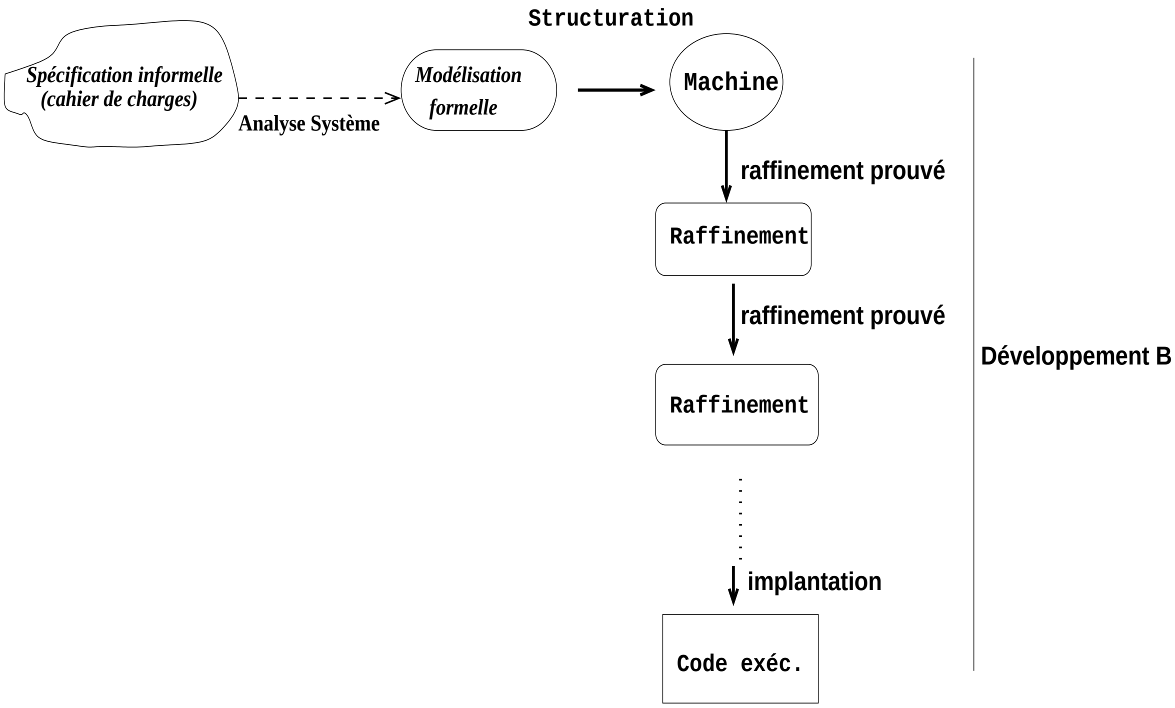


Figure: Analysis and B development

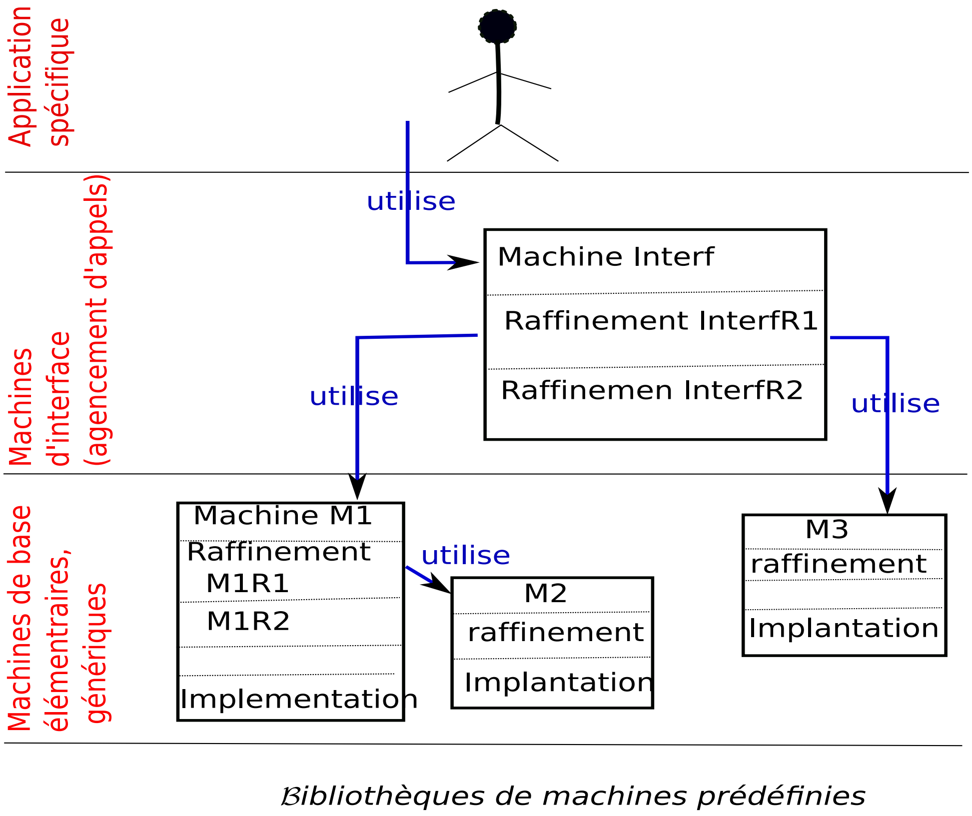


Figure: Structure of a B Development

Position - other methods

- B: Correct-by-construction Approach → *proofs*
- B: Unique framework for (software lifecycle):
  - Analysis
  - Specification/Modeling
  - Design
  - Development
- B: Stepwise Refinements from abstract model to concrete one.
- (Other) Approaches: development, test à postériori → *tests*

Constructing the GCD: abstract machine

```
MACHINE
    pgcd1 /* the GCD of two naturals */
            /* gcd(x,y) is d | x mod d = 0 ∧ y mod d = 0
            ∧ ∀ other divisors dx d > dx
            ∧ ∀ other divisors dy d > dy */
OPERATIONS
        rr <-- pgcd(xx,yy) = /* OUTPUT : rr ; INPUT xx, yy */
            ...
END
```


Constructing the GCD: abstract machine

```
OPERATIONS
rr <-- pgcd(xx,yy) = /* spécification du pgcd */
PRE
    xx : INT & xx >= 1 & xx < MAXINT
& yy : INT & yy >= 1 & yy < MAXINT
THEN
    ANY dd WHERE
    dd : INT
    & (xx - (xx/dd)*dd) = 0 /* d is a divisor of x */
    & (yy - (yy/dd)*dd) = 0 /* d is a divisor of y */
        /* and the other common divisors are < d */
    & !dx.((dx : INT & dx < MAXINT
        & (xx- (xx/dx)*dx) = 0 & (yy-(yy/dx)*dx)=0) => dx < dd)
    THEN rr := dd
    END
END
```

Constructing the GCD: refinement

```
REFINEMENT /* raffinement de ...*/
    pgcd1_R1
REFINES pgcd1 /* the former machine */
OPERATIONS
rr <-- pgcd (xx, yy) = /* the interface is not changed */
    BEGIN
        ... Body of the refined operation
    END
END
```


Constructing the GCD: refinement

```
rr <-- pgcd (xx, yy) = /* the refined operation */
    BEGIN
        VAR cd, rx, ry, cr IN
            cd := 1
            ; WHILE ( cd < xx & cd < yy) DO
                ; rx := xx - (xx/cd)*cd ; ry := yy - (yy/cd)*cd
                IF (rx = 0 & ry = 0)
                THEN /* cd divises x and y, possible GCD */
                    cr := cd /* possible rr */
                END
                ; cd := cd + 1 ; /* searching a greater one */
            INVARIANT
                xx : INT & yy : INT & rx : INT & rx < MAXINT
                & ry : INT & ry < MAXINT & cd < MAXINT
                & xx = cr*(xx/cr) + rx & yy = cr*(y/cr) + ry
            VARIANT
                xx - cd
            END
```



```
    END
```

After the examples

... Let's dig a bit ...

A simplified general shape of an abstract machine

MACHINE
  M (prm) /* Name and parameters */
CONSTRAINTS
  C /* Predicate on X and x */
/* clauses USES, SEES, INCLUDES, EXTENDS, ***/**
SETS
  ENS /* list of basic sets identifiers */
CONSTANTS
  K /* list of constants identfiers */
PROPERTIES
  B /* preedicate(s) on K */
VARIABLES
  V /* list of variables identifiers */
DEFINITIONS
  D /* list of definitions (macros) */

A simplified shape of an abstract machine (cont'd)

```
...
INVARIANT
        I                      /* a predicate */
INITIALISATION  U              /* the initialisation */
OPERATIONS
        u ← O(pp) =            /* an operation O */
        PRE
          P
        THEN
          Subst                /* body of the operation*/
        END;
        ...
end
```


Semantics: consistency of a machine

$$\exists\, prm.C$$

It is possible to have values f parameters that meet the constraints

$$C \;\Rightarrow\; \exists\,(ENS, K).B$$

There are sets and constants that meet the properties of the machine

$$B \;\wedge\; C \;\Rightarrow\; \exists\, V.I$$

There are a state that meets the invariant

$$B \;\wedge\; C \;\Rightarrow\; [U]I$$

The initialisation establishes the invariant

For each operation of the machine

$$B \;\wedge\; C \;\wedge\; I \;\wedge\; P \;\Rightarrow\; [Subst]I$$

Each operation called under its precondition preserves the invariant

Proof Obligations (PO)

There are the predicates to be proven to ensure the consistency (and the correction) of the mathematical model defined by the abstract machine.
The designer of the machine has two types of proof obligations:

- prove that the INITIALISATION establishes the invariant;
- prove that each OPERATION, when called under its precondition, preserves the invariant.

$$I \ \wedge \ P \ \Rightarrow \ [Subst]I$$

In practice, one has tools assistance to discharge the proof obligations.

Semantics of a machine - Consistency

To formally establish the condition for the correct functionning of a machine, one uses proof obligations.
To guaranty the correction of a machine, we have two main proof obligations:

- The initialisation establishes the invariant
- Each operation of the machine, when called under its precondition, preserves the invariant.

These are logical expressions, predicates, which are proved.

New Example

...SORTING...

Example of Specifying Sorting with B

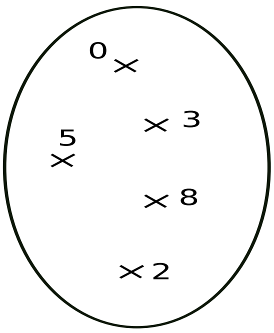


Figure: Modeling the Sorting of (a set of) Naturals

Example of Specifying Sorting with B

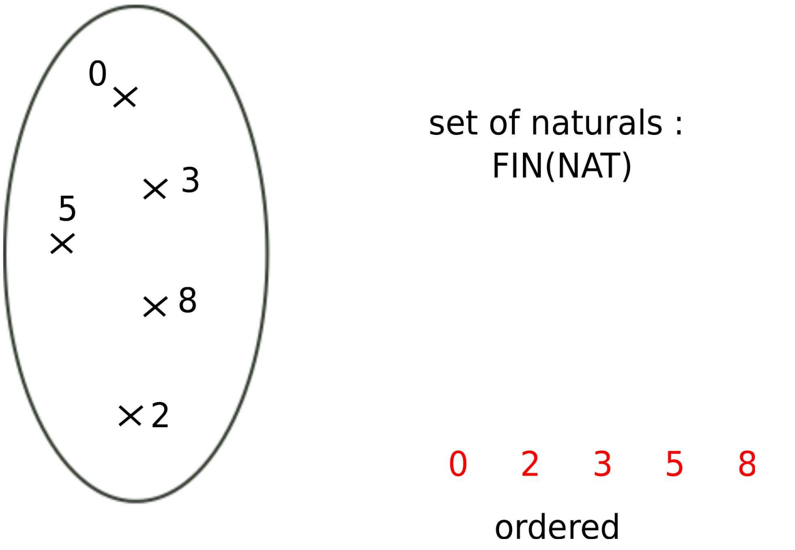


Figure: Modeling the Sorting: ordering the set of Naturals

Example of Specifying Sorting with B

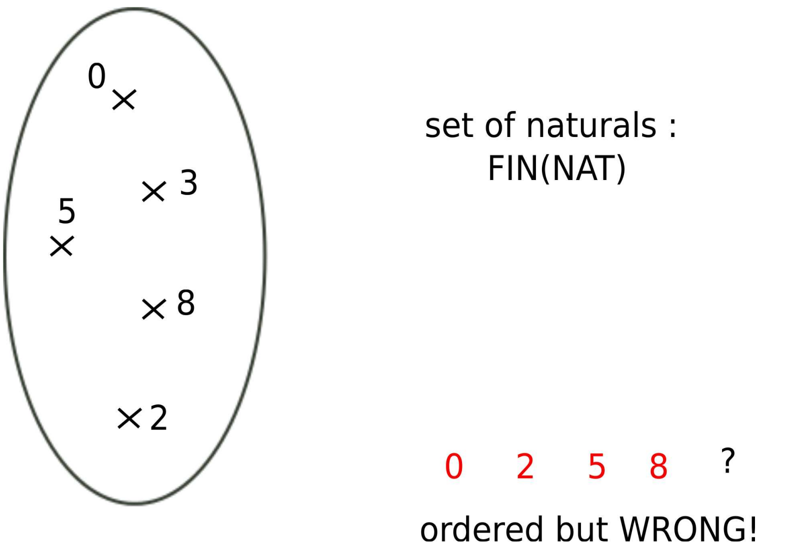


Figure: Modeling the Sorting: be careful!

Example of Specifying Sorting with B

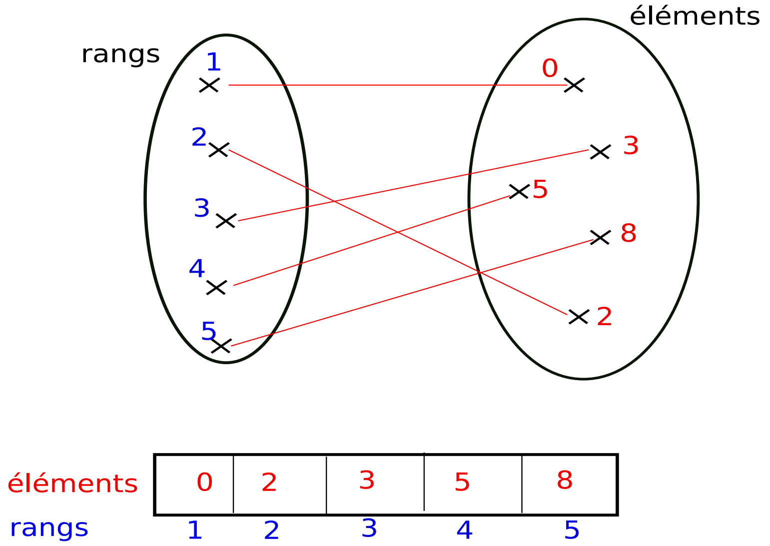


Figure: Modeling the Sorting

Example of Specifying Sorting with B

```
MACHINE /* Specify the sorting of a set of naturals */
         Sort
CONSTANTS
         sortOf /* defining a function */
PROPERTIES
   sortOf : FIN(NAT) +-> seq(NAT) &
     %ss.(ss : FIN(NAT) =>
     (ran(sortOf(ss)) = ss &
      %(ii,jj).(ii : dom(sortOf(ss)) &    jj : dom(sortOf(ss)) &
      ii < jj    =>    (sortOf(ss))(ii) < (sortOf(ss))(jj) )
    ) )
END
```

Example of Specifying Sorting with B

```
MACHINE
    SpecSort
    /* specify an appli that gets naturals and then sort them */
SEES
            Sort /* To use the previous machine */
SETS
            SortMode = {insertion, extraction}
VARIABLES
            unsorted, sorted, mode
INVARIANT
             unsorted : FIN(NAT)
&           sorted : seq(NAT)
&           mode : SortMode
&           ((mode = extraction) => (sorted= sortOf(unsorted)))
```


Example of Specifying Sorting with B

```
/* MACHINE SpecSort
                    (continued ...) */

INITIALISATION
            unsorted := {} || sorted:= [] || mode := extraction

OPERATIONS
           moveToInsertion =
            PRE
                        mode = extraction
            THEN
                        mode := insertion ||
                        unsorted := {} ||
                        sorted :: seq(NAT)
            END
;
...
```

Example of Specifying Sorting with B

```
/* MACHINE SpecTri
(continued ...) */

         input(xx) =
         PRE
                 xx : NAT & mode = insertion
         THEN
                 unsorted := unsorted \/ {xx} ||
                 sorted :: seq(NAT)
         END
;
         moveToExtraction() =
         PRE
                 mode = insertion
         THEN
                 mode := extraction ||
                 sorted := sortof(unsorted)
         END
```


Example of Specifying Sorting with B

```
/* MACHINE SpecTri
(continued ...) */


yy <- extract(ii) =
         PRE
                 ii : dom(sorted) & mode = extraction
         THEN
                 yy := sorted(ii)
         END
END
```

B - Data Language - sets and typing

- Predefined Sets (work as types)
  BOOL, CHAR,
  INTEGER ($\mathbb{Z}$), NAT ($\mathbb{N}$), NAT1 ($\mathbb{N}^*$) ,
  STRING
- Cartesian Product E $\times$ F
- The set of subsets (powerset) of E $\mathcal{P}(E)$
  written POW(E)

B - Data Language

With the data language

- we model the state space of a system with its data
- we describe the invariant properties of a system

Modeling the state:

- Abstraction, modeling (abstract sets, relations, functions, ...)
- Logical Properties, or algebraic properties.

B - Data Language

- When we model a system (with the set of its states) and make explicit its (right) properties, we ensure thereafter that the system only goes through the set of states that respect the defined properties: it is the consistency of the system.
- To show that it is possible to have states satisfying the given properties, one builds at least one state (it is the initial state).
- The specified system is correct if after each operation, the reached state is a state satisfying the given invariant properties.

B - Data Language

First Order Logic

Description	Notation	Ascii
and	$p \wedge q$	`p & q`
or	$p \vee q$	`p or q`
not	$\neg p$	`not p`
implication	$p \Rightarrow q$	`(p) ==> (q)`
univ. quantif.	$\forall x.p(x)$	`!x.(p(x))`
exist. quantif.	$\exists x.p(x)$	`#x.(p(x))`

Variables should be typed:
`#x.(x : T ==> p(x))` and `!x.(x : T ==> p(x))`

B - Data Language

The standard set operators

E, F and T are sets, x an member of F

Description	Notation	Ascii
union	$E \cup F$	`E \/ F`
intersection	$E \cap F$	`E /\ F`
membership	$x \in F$	`x : F`
difference	$E \setminus F$	`E - F`
inclusion	$E \subseteq F$	`E <: F`
selection	`choice(E)`	`choice(E)`

+ generalised Union and intersection
+ quantified Union et intersection

B - Data Language

In ascii notation, the negation is written with $/$.

Description	Notation	Ascii
not member	$x \notin F$	`x /: F`
non inclusion	$E \not\subseteq F$	`E /<: F`
non equality	$E \neq F$	`E /= F`

Generalised Union

an operator to achieve the generalised union of well-formed *set expressions*.

$S \in \mathcal{P}(\mathcal{P}(T))$

$\Rightarrow$

$union(S) = \{x \mid x \in T \wedge \exists u.(u \in S \wedge x \in u)\}$

Example

$$union(\{\{aa, ee, ff\}, \{bb, cc, gg\}, \{dd, ee, uu, cc\}\})$$
$$= \{aa, ee, ff, bb, cc, gg, dd, uu\}$$

Quantified Union

an operator to achieve the quantified union of well-formed *set expressions*.

$\forall x.(x \in S \Rightarrow E \subseteq T)$

$\Rightarrow$

$$\bigcup x.(x \in S \mid E) = \{y \mid y \in T \wedge \exists x.(x \in S \wedge y \in E)\}$$

Exemple

$$UNION(x).(x \in \{1, 2, 3\} \mid \{y \mid y \in NAT \wedge y = x * x\})$$

$$= \{1\} \cup \{4\} \cup \{9\} = \{1, 4, 9\}$$

Generalised Intersection

an operator to achieve the generalised intersection of of well-formed *set expressions*.

$S \in \mathcal{P}(\mathcal{P}(T))$

$\Rightarrow$

$inter(S) = \{x \mid x \in T \wedge \forall u.(u \in S \Rightarrow x \in u)\}$

Example

$inter(\{\{aa, ee, ff, cc\}, \{bb, cc, gg\}, \{dd, ee, uu, cc\}\} = \{cc\}$

Quantified Intersection

an operator to achieve the quantified intersection of of well-formed *set expressions*.

$\forall x.(x \in S \Rightarrow E \subseteq T)$

$\Rightarrow$

$\bigcap x.(x \in S \mid E)$
$= \{y \mid y \in T \wedge \forall x.(x \in S \Rightarrow y \in E)\}$

Example

$INTER(x).(x \in \{1, 2, 3, 4\} \mid \{y \mid y \in \{1, 2, 3, 4, 5\} \wedge y > x\})$

$= inter(\{\{1, 2, 3, 4, 5\}, \{2, 3, 4, 5\}, \{3, 4, 5\}, \{4, 5\}\})$

Relations

Description	Notation	Ascii
relation	$r : S \leftrightarrow T$	`r : S <-> T`
domain	$dom(r) \subseteq S$	`dom(r) <: S`
range	$ran(r) \subseteq T$	`ran(r) <: T`
composition	$r;s$	`r;s`
composition r(s)	$r \circ s$	`r(s)`
identity	$id(S)$	`id(S)`

Relations (continued)

Description	Notation	Ascii
domain restrictition	$S \lhd r$	`S <| r`
range restriction	$r \rhd T$	`r |> T`
domain antirestriction	$S \ntriangleleft r$	`S <<| r`
range antirestriction	$r \ntriangleright T$	`r |>> T`
inverse	$r^{\sim}$	`r ~`
relationnelle image	$r[S]$	`r[S]`
overiding	$r1 \oplus r2$	`r1 <+ r2`
direct product of rel.	$r1 \otimes r2$	`r1 >< r2`
closure	$closure(r)$	`closure(r)`
reflexive trans. closure	$closure1(r)$	`closure1(r)`

Functions

Description	Notation	Ascii
partial function	$S \nrightarrow T$	S +-> T
total function	$S \rightarrow T$	S --> T
partial injection	$S \rightarrowtail\!\!\!\!\nrightarrow T$	S >+-> T
total injection	$S \rightarrowtail T$	S >--> T
partial surjection	$S \twoheadrightarrow\!\!\!\!\nrightarrow T$	S +->> T
total surjection	$S \twoheadrightarrow T$	S -->> T
total bijection	$S \rightarrowtail\!\!\!\!\twoheadrightarrow T$	S >->> T
lambda abstraction	$\%x.(P \mid E)$	

Sequences

Description	Notation
sequence of elements of T	$seq(T) = union(n).(n \in N \mid 1..n \rightarrow T)$
empty sequence	[]
injective sequence of element of T T	$iseq(T)$
bijective sequence of element of T T	$perm(T)$
size of a sequence s	$size(s) = card(dom(s))$

Sequences (continued)

Description	Notation
first element of a seq. s	$first(s) = s(1)$
last element of a seq. s	$last(s) = s(size(s))$
restrict. of s t its s n first elem. elments	$s \uparrow n$
elimination of the first n elements of s	$s \downarrow n$

Modeling Operations

Basic Concepts of the Dynamic Part

Weakest preconditions

Context: *Hoare/Floyd/Dijkstra Logic* Hoare triple
(State, state space, statements, execution, Hoare triple)

$$\{P\}\ S\ \{R\}$$

S a **statement** and R a **predicate that denotes the result of** S.
$wp(S, R)$, is the predicate that descrives:
the set of all states | the execution of S begining with one of them **terminates** in a *finite time* din a state satisfaying R,
$wp(S, R)$ is the *weakest precondition* of S with respect to R.

Basic Concepts of the Dynamic Part

Some examples

Let S be an assignment and
R the predicate $i \leq 1$

$$wp(i := i + 1, i \leq 1) \;=\; (i \leq 0)$$

Let S be the conditional:
if $x \geq y$ then $z := x$ else $z := y$
and R the predicate $z = max(x, y)$

$$wp(S, R) \;=\; Vrai$$

Weakest preconditions - meaning

The meaning of $wp(S, R)$ can be make precise with two properties:

- $wp(S, R)$ is a precondition guarantying R after the execution of S, that is:

$$\{wp(S, R)\}\ S\ \{R\}$$

- $wp(S, R)$ is the weakest of such preconditions, that is:
  if $\{P\}\ S\ \{R\}$ then $P \Rightarrow wp(S, R)$

Weakest preconditions - meaning

In practice a program S establishes a postcondition R.
Hence the interest for the precondition that permits to establish R.
wp is a function with two parameters:

a statement (or a program) S and

a predicate R.

For a fixed S, we can view $wp(S,\ R)$ as a function with only one parameter $wp_S(R)$.
The function wp_S is called *predicate transformer* - Dijkstra
It is the function which associates to every predicate R the weakest precondition such that $\{P\}\ S\ \{R\}$.

B: Generalized Substitutions - Axioms

Generalisation of the classical substitution of the Logic
(to model the behaviours of operations).
Consider a predicate R to be established, the semantics of generalized substitution is defined by the predicate transformer.

- **Simple Substitution** S
  Semantics **[S]R** is read : S establishes R
- **Multiple Substitution** $x, y := E, F$
  Semantics **[x, y := E, F]R**

B: generalized substitutions - Basic set of GS

The abstract syntax language to specify the operations:

Le R be the invariant, S, T substitutions

Name	Abs. Synt.	definition	equivalent in logic
neutral (id.)	$skip$	$[skip]R$	R
Pre-condition	$P \mid S$	$[P \mid S]R$	$P \wedge [S]R$
Bounded choice	$S [] T$	$[S [] T]R$	$[S]R \wedge [T]R$
Guard	$P \Longrightarrow T$	$[P \Longrightarrow T]R$	$P \Rightarrow [T]R$
Unbounded	$@x.S$	$[@x.S]R$	$\forall x.[S]R$ x bounded (not free) in R

enough as B specification language but ...

Non determinism - Substitutions

- **Abstraction** ⇒ (possible)non determinism. OK for specifying.
- **Concretisation** ⇒ refinement into code
- Extending the basic GSL set to other substitutions closed to programming

  CASE OF
  SELECT
  IF THEN ELSE

B - Generalized substitutions language

Syntactic extension of substitutions: basic substititution set

Basis Substitution

noted S

Syntactic Extension

```
BEGIN
   S
END
```

Simultaneous Substitutions

Consider S and T two substitutions.

S being $x := E$ and
T being $y := F$

note `S || T`

B - generalized substitution Language

Neutral Substitution

skip

Syntactic extension

skip

Subst. with precondition

$P \mid S$

Syntactic extension

```
PRE
P
THEN
S
END
```


B - generalized substitution Language

Bounded choice

$S [] T$

Syntactic extension

```
CHOICE
S
OR
T
END
```

Guarded Substitution

$(P \Longrightarrow T) [] (\neg P \Longrightarrow S)$

Syntactic extension

```
IF P
THEN T
ELSE S
END
```

B - generalized substitution Language

Unbounded Choice Substitution Syntactic extension

$@x.S_x$

```
VAR x IN
Sx
END
```


Extending the basic substution set: non-deterministic

Nondeterministic @
$@x.(P_x \implies S_x)$

Syntactic extension

```
ANY x
WHERE Px
THEN Sx
END
```

Extending the basic substution set : non-deterministic

Nondeterministic $x :\in U$
(becomes member)
x:: U
$@y.(y \in U \implies x := y)$

Syntactic extension

```
ANY y
WHERE y : U
THEN x := y
END
```


Basic Concepts of the Dynamic Part

B - generalized substitution Language

Extensions... non-deterministic

Nondeterministic $x : P(x)$
(x such that P)
x: P(x)

Proof Obligations

...Proof Obligation (PO)...

Consistency Proof Obligations

```
MACHINE ThreshCtrl
CONSTANTS    thresX, threshY
PROPERTIES     thresX : INT & thresX = 10 ...
VARIABLES     xx
INVARIANT        xx : INT & 0 <= xx & xx <= thresX
INITIALISATION     xx := 0
OPERATIONS
setXX(nx) = /* an operation with PRE */
PRE     nx : INT & nx >= 0 & nx <= thresX
THEN
    xx := nx
END
; incrXX(px) = /* incrementation of xx with px */
PRE      px : INT & xx+px >= 0 & xx+px <= thresX
THEN
    xx := xx+px
END
END
```

Proof Obligations (recall)

The predicates to be proved to ensure the consistency (and the correction) of the mathematical model defined by the abstract machine.
The machine developer has two kinds of PO:

- to prouve that the INITIALISATION establishes the invariant: $[Init]I$
- to prove that each OPERATION, when it is called under its precondition, preserves the invariant.

$$I \wedge P \Rightarrow [Subst]I$$

In practice, CASE tools are used to help in discharging the proofs.

Proof of the operation setXX(nx)

We must prove that $I \wedge P \Rightarrow [Subst]I$

```
INVARIANT      xx : INT & 0 <= xx & xx <= thresX
setXX(nx) =
PRE
    nx : INT & nx >= 0 & nx <= thresX
THEN
    xx := nx    /*  Subst  */
END
INVARIANT      xx : INT & 0 <= xx & xx <= thresX
```

(use white/blackboard)

Precondition computation / preservation of the invariant

xx : INT & 0 <= xx & xx <= thresX
setXX(nx) = PRE ... ? THEN xx := nx /* Subst */ END
nx : INT & 0 <= nx & nx <= thresX

We express $[Subst]I$ and obtain a predicate which should be true!

nx : INT & 0 <= nx & nx <= thresX ?

It is the precondition!

Precondition computation / preservation of the invariant

xx : INT & 0 <= xx & xx <= thresX
incrXX(px) = PRE ... ? THEN xx := xx+px /* Subst */ END
xx : INT & 0 <= xx & xx <= thresX

We express $[Subst]I$ and obtain a predicate which should be true!

xx+px : INT & 0 <= xx+px & xx+px <= thresX ?

hence the precondition: px : INT & 0 <= xx+px & xx+px <= thresX

Example of ressources allocation (recall)

MACHINE
Resrc
SETS
RESC
CONSTANTS
maxRes // a parameter
PROPERTIES
maxRes : NAT & maxRes > 1
VARIABLES
rsc
INVARIANT
rsc <: RESC // subset
& card(rsc) <= maxRes //
bounded
INITIALISATION
rsc := {}
	OPERATIONS
	addRsc(rr) = // adding
	ressources
	PRE
	rr : RESC & rr /: rsc &
	card(rsc) < maxRes
	THEN
	rsc := rsc \/ {rr}
	END
	;
	rmvRsc(rr) = //
	allocation
	PRE
	rr : RESC & rr : rsc
	THEN
	rsc := rsc - {rr}
	END
	END

Consistency of a machine: proof obligation

The Initialisation establishes the invariant: $[U]I$;

[rsc := {}] (rsc <: RESC & card(rsc) <= maxRes) ?

Replace variables with their values:

{} <: RESC & card({}) <= maxRes ?

Reduce

{} <: RESC & 0 <= maxRes ?

TRUE

Consistency of a machine: proof obligation

Preservation of the invariant by: addRsc(rr)

```
rsc <: RESC &  card(rsc) <= maxRes
PRE
     rr : RESC & rr /: rsc & card(rsc) < maxRes
THEN
     rsc := rsc \/ {rr}
END
rsc <: RESC &  card(rsc) <= maxRes
```

Replace variables with their values in I:

rsc \/ {rr} <: RESC & card(rsc \/ {rr}) <= maxRes ?

(use white/blackboard)

Case Studies

...Cas Euclide...

Démo division euclidienne

```
   Euclid Pgm demo
        +----------------------------------+
        +   Menu de l'application         +
        +----------------------------------+
            Nouvelle division          : 1
        +----------------------------------+
            Quitter                    : 0
        +----------------------------------+
         choix ? 1
 Division euclidienne
 Donnez le dividende (entre 3 et 78)
56
 Donnez le diviseur (entre 1 et 78)
78
 Resultat de la division : 0
 Reste de la division : 56
```


suite démo

```
        +----------------------------------+
        +   Menu de l'application         +
        +----------------------------------+
            Nouvelle division          : 1
        +----------------------------------+
            Quitter                    : 0
        +----------------------------------+

         choix ? 1
 Division euclidienne
 Donnez le dividende (entre 3 et 78)
67
 Donnez le diviseur (entre 1 et 78)
6
 Resultat de la division : 11
 Reste de la division : 1
```

Spécification de Euclide

```
MACHINE
        euclide
OPERATIONS
 reste, quot ⟵ calculReste(divis, divid) =
 PRE
         divis ∈ NAT ∧ divid ∈ NAT  ∧  divis > 0
 ∧      divis ≤ divid  /* sinon B le trouve */
 THEN
        ANY vq, vr WHERE
                vq  ∈ NAT
        ∧       vr  ∈ NAT
        ∧       divid = vq*divis + vr
        THEN
                quot  := vq
        ||      reste  := vr
        END
 END
END
```


Example of development with B

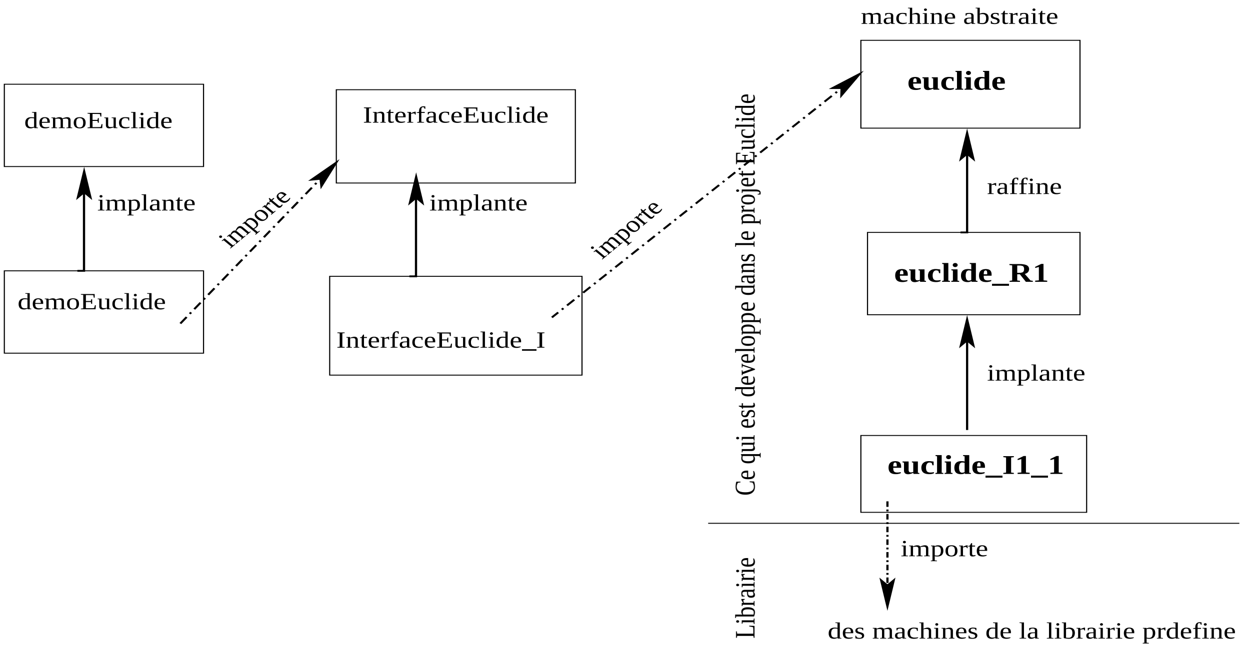


Figure: Architecture of applications with B

Refinement: development technique

Idea of refinement :

- We start with an abstract machine defining an abstract mathematical model,
- we refine this model to obtain a concrete model :
  - the abstract model is not executable.
    Why? (it is defined with mathematical objects)
  - to obtain an equivalent model,wrt to functionalities, but more concrete.
    (it is described with programming objects)

There is a well-defined Theory of refinement [Morgan 1990; R-J. Back 1980; C. Ralph-Johan Back, Joakim Wright, 1998]

Refinement: development technique

- The objective of refinement is the construction of executable code.
- We should guaranty that the refinement is correct:
  (refinement proof).

⇒ refinement proof obligations

Approach of refinement

What to refine in the model?

- The **variables and the invariant**
  Static Part - state space
  Changes of variables (replacement with more concrete ones):
- The **operations**
  Dynamic Part - generalized substitutions
  refinement of substitutions.
  Introduce **refinement substitutions**
  (until reaching programming substitutions).

Approach of refinement: How to refine?

Introducing data structures and
replacing abstract structures by concrete ones.

- Use the clause REFINES to link the abstract machine witj its refinement

```
REFINEMENT
        MM_R1
REFINES
        MM
...
END
```

- Refining the state space:
  - introduce new (concrete) variables,
  - choice of (less abstract) structures,
  - binding abstract and concrete variables
    bay a binding invariant

Approach of refinement: How to refine?

- Refinement of the operations:
  - The interface should not be modified.
  - Rewrite the abstract operations with the new variables and the appropriate substitutions (introducing sequences, loop, local variables).
  - Introduce refinement substitutions.
  - Remove non-determinism
  - Weak in the concrete refined machine, the preconditions of the abstract operations, until they disappear.

$\Rightarrow$ extending the substitution language.

Examples of refinement

Aready seen:

- Resource Allocation
- Euclidian Division

Example refinement

- Modeling and development of a resource allocation system
- There are N resources to allocate/free
- The allocation is done according to the availability of the resources
- the allocated resources are free after a while

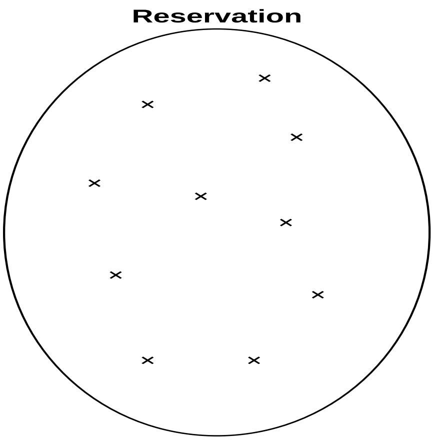
Example : resource allocation

$\mathtt{n\_rsrc} \in \mathtt{0..100}$

$\mathtt{n\_rsrc}$ = cardinal of the set

`allocate` $\rightarrow$ - 1 element

`free` $\rightarrow$ + 1 element



```
MACHINE
     Allocation

VARIABLES
     n_rsrc

INVARIANT
     n_rsrc : 0..100

INITIALISATION
     n_rsrc := 100
```

```
 OPERATIONS
allocate =
    PRE  n_rsrc > 0
    THEN
         n_rsrc := n_rsrc - 1
    END
;
free =
    PRE  n_rsrc < 100
    THEN
         n_rsrc := n_rsrc + 1
    END
;
bb <-- available =
     bb :: BOOL
// ou  bb := bool(0 < n_rsrc)
 END
```


Consistency Proof

The developer of the abstract machine has to kinds of PO:
To prove that the INITIALISATION establishes the invariant

$$[n\_rsrc := 100](n\_rsrc \in 0..100)$$

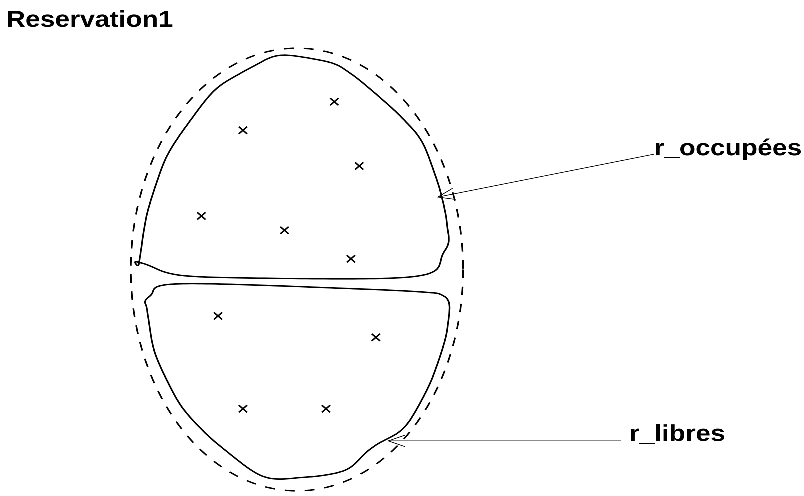
we should prove that $100 \in 0..100$

Consistency Proof

We have to prove that each operation called under its PREcondition, preserve the invariant.

- for the operation *allocate* we should prove:
  $n\_rsrc \in 0..100 \wedge 0 < n\_rsrc \Rightarrow n\_rsrc - 1 \in 0..100$
- for the operation *available* we should prove:
  $n\_rsrc \in 0..100 \wedge (n\_rsrc > 0 \vee \neg (n\_rsrc > 0))$
  $\Rightarrow$
  $n\_rsrc \in 0..100$

Resource allocation (Refinement)



`allocate` $\longrightarrow$ find 1 free element

`free` $\longrightarrow$ find 1 unavailable element

```
REFINEMENT
  Allocation_R1
REFINES
  Allocation
VARIABLES
   rs_free, rs_unavailable // n_rscrc est incluse
// new less abstract variables
INVARIANT
  rs_free : POW(INTEGER)
& rs_unavailable : POW(INTEGER)
& rs_free /\ rs_unavailable = {}
& n_rsrc = card(rs_free) // binding invariant
INITIALISATION
  rs_free, rs_unavailable, n_rsrc := 1..100, {}, 100
```

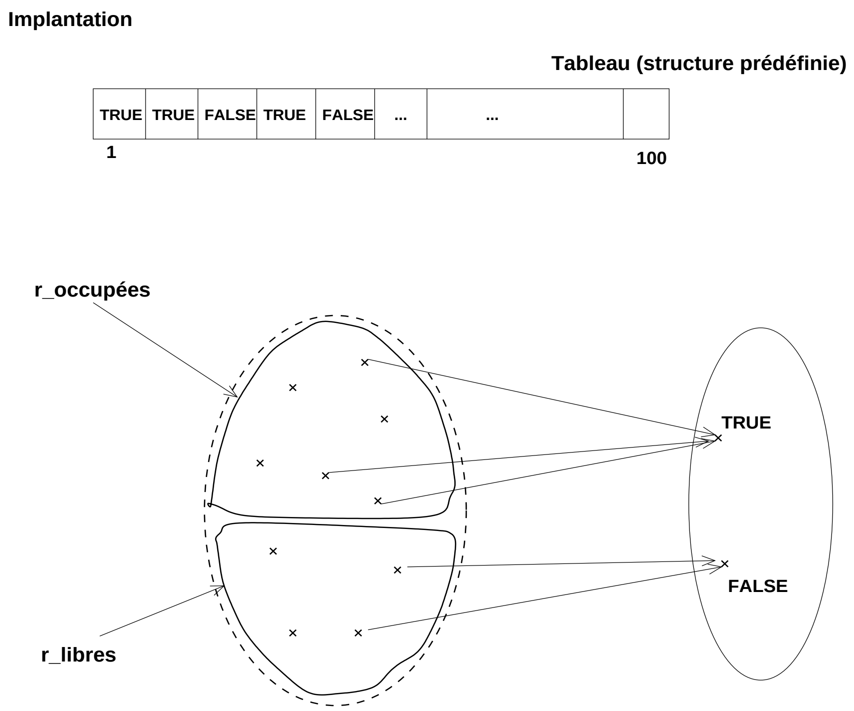


```
OPERATIONS
 allocate =     // rewritten with the new variables
  ANY ss WHERE
     ss : rs_free  // non-dterministic way
  THEN
      rs_free := rs_free - {ss}
   || rs_unavailable := rs_unavailable \/  {ss}
   || n_rsrc := n_rsrc - 1
  END
;
```

```
free =  // rewritten with the  new variables
 ANY ss WHERE
    ss : rs_unavailable
 THEN
     rs_free := rs_free \/ {ss}
 || rs_unavailable :=
           rs_unavailable - {ss}
 || n_rsrc := n_rsrc + 1
 END
;
```

```
bb <-- available =
   IF 0 < n_rsrc
   THEN
      bb := TRUE
   ELSE
      bb := FALSE
   END
END
```


Resource allocation (Implementation)



Structure of the implementation

```
IMPLEMENTATION
  Allocation_I1
REFINES
  Allocation_R1
IMPORTS
 ... // import predefined machines
VARIABLES
 ... // new concrete variables
INVARIANT
 ...
INITIALISATION
 ...
OPERATIONS
 ... // They are now rewritten with refinement subst.
        and programming substitutions
```


Refinement substitutions

- **Sequential substitutions**
  Let S and T be substitutions,
  the sequential substitution is noted: $S\ ;\ T$
  Its semantic definition is expressed with:

$$
\begin{aligned}
[S;T]R &\equiv [S][T]R \\
&\equiv [S]([T]R) \\
&S \text{ establishes } [T]R
\end{aligned}
$$

Refinement substitutions

- **Loop substitution**
  The loop substitution has the following shape:

while P do
S
invariant
I
variant
V
end

Semantic of the loop substitution

Semantically, it is

$I \wedge$

/* the variant is a natural */

$\forall x.(I \Rightarrow V \in NATURAL) \wedge$

/* the variant decreases after each step */

$\forall (x,n).(I \wedge P \Rightarrow [n := V][S](V < n)) \wedge$

/* continuation of the loop */

$\forall x.(I \wedge P \Rightarrow [S]I) \mid$

$@x'.([x := x'](I \wedge \neg P) \Rightarrow x := x'))$

Substitution VAR ... IN

- **Block with local variables**
  The notation is :

  var x in // introduction of local variables
  S
  end

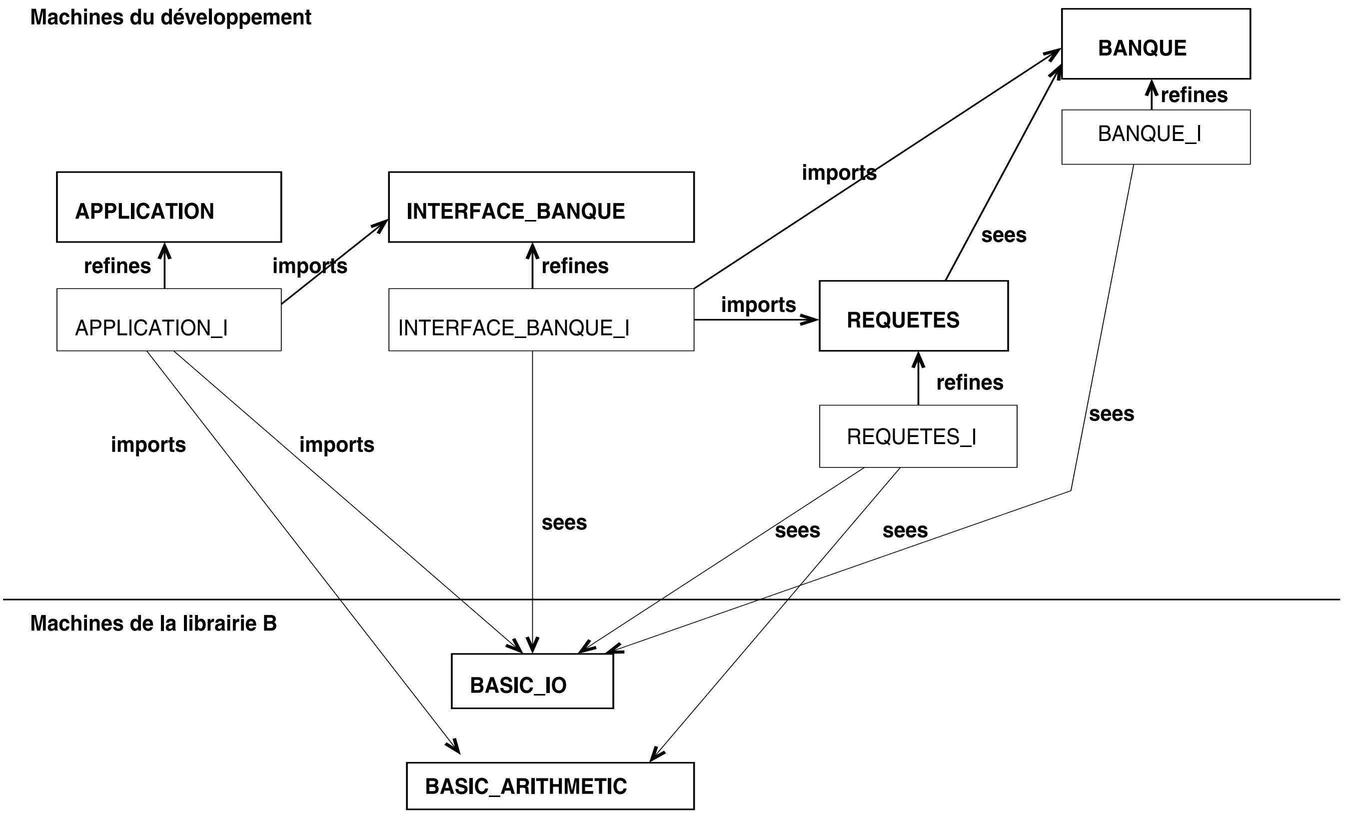
Architecture of Large Systems

Composition of machines → large machines.

- Modules - Composition - Layered Architecture
- Modularity

Composition of machines

- Hierarchy
  with the clauses INCLUDES, EXTENDS, PROMOTES
- Sharing
  with the clauses SEES, USES



Hierarchy

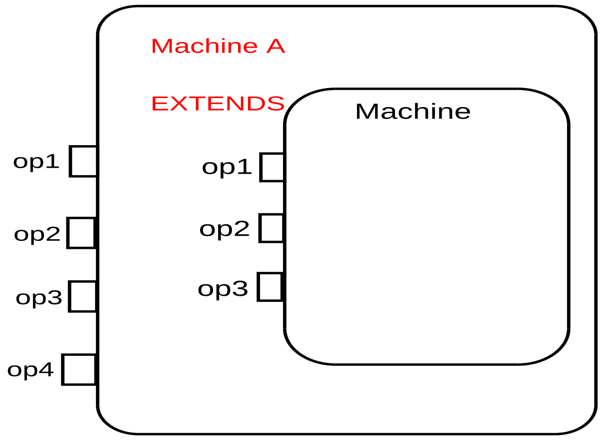
INCLUDES to include a machine in another one
+ promotion of some operations PROMOTES

```
MACHINE
       MA
 INCLUDES
       MB               /* access by Opmb to varB */
 PROMOTES
       Opmb1, Opmb3     /* become operations of MA */
...
END
```

Hierarchy

EXTENDS, inclusion but no need to promote

```
MACHINE
        MA
 EXTENDS
        MB
...
END
```



Sharing

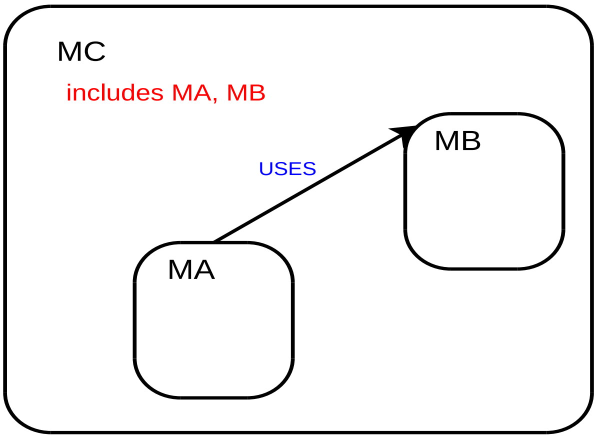
SEES for a read only sharing

```
MACHINE
        MA
 SEES
        MB
...
END
```

Sharing

USES for a read/write sharing

```
MACHINE
       MA
 USES
       MB
...
END
```



MA et MB should be included in another machine.