The B Method: an introduction

- Introduction
  - What is B? Applications? SIL and Standard
- Positionning with Formal Methods
  - A quick overview of the B Method
  - Example of specification
    - GCD (PGCD) + refinement, euclidian division,
    - Light Regulation in a Room
- How to develop using B
  - Light regulation, Gauge, Resource Management

Examples of development

- Examples
  - GCD (PGCD), euclidian division,
  - Sorting
- Basic concepts of the method : abstract machine
  - Modeling the static part (data)
  - Modeling the dynamic part (operations)
  - Proof of consistency
  - Refinement of machine
  - Proofs of refinement
- Case studies (with AtelierB, Rodin)
Examples of application in railways systems

- Applications in Transportation Systems (Alstom–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.

The Standard EN 50128: Software Aspect of the Control

**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 criticality**:
- Not critical: SIL0
- No dead danger for humans: SIL1, SIL2
- Critical: SIL3, SIL4

**Applicable to**: the software application; the operating systems; the CASE tools;

Depending on the projects and the contexts, we will need formal methods to build the dependable software or systems.
**Introduction to B Method in Software Engineering**

**Formal Method**
- Formal Specification or Modeling Language
- Formal reasoning System

**B Method**
- Specification Language
  - Logic, Set Theory: data language
  - Generalized Substitution Language: Operation's language
- Formal reasoning System
  - Theorem Prover

**Formal Development**

**Formal Software Development**
- Systematic transformation of a mathematical model into an executable code.
- Transformation from the abstract to the concrete model
- Passing from mathematical structures to programming structures
- Refinement into code in a programming language.

**B**
- Formal Method
  - + refinement theory (of abstract machines)
  - ≤ formal development method

**Correct Development (no overflow, for a trajectory)**

```plaintext
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 (continued)**

```plaintext
OPERATIONS (continued)
setXX(nx) = /* specification of an operation with PRE */
  nx := 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: 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
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 */
  END
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
```

B Method: Global Approach

```
Figure: Analysis and B development
```

The B Method

- **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 $I = 2$, $I = 6$, $I = 0$, ... 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 $\langle I, J \rangle$, $\langle V_1, V_2, V_4, ... \rangle$
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

![Diagram of state transitions]

Figure: Evolution of a software system

Specification Approach

- A tuple of variables describes a state
  \[ \langle \text{mode} = \text{day}, \text{light} = \text{off}, \text{temp} = 20 \rangle \]
- A predicate (with the variables) describes a state space
  \[ \text{light} = \text{off} \land \text{mode} = \text{day} \land \text{temp} > 12 \]
- An operation that affects the variables changes the state
  \[ \text{mode} := \text{day} \]

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

Abstract Machine

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

variables
predicates
operation

- 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
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; it should be always true.

Both clauses go together.

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

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 implicitly the invariant).
Abstract Machine: provides operations

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

Machine ReguLight

SETS VARIABLES INVARIANT INITIALISATION OPERATIONS

putsOn decreaseTemp putOff increaseTemp

Figure : The operations are callable from outside

Interface of operations

(operations with or without input/output parameters)

- No parameter:
  
  ```plaintext
  nameOfOperation = ...
  ```

- Input parameters only:
  
  ```plaintext
  nameOfOperation(p1, p2, ...) = ...
  ```

- Output parameters only:
  
  ```plaintext
  r1, r2, ... ← nameOfOperation = ...
  ```

- Input and Output parameters:
  
  ```plaintext
  r1, r2, ... ← nameOfOperation(p1, p2, ...) = ...
  ```

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

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

```plaintext
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 resources

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

J. Christian Attiogbé (November 2014)

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 (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

☞ Before calling the operation, we should ensure that it does not overpass the authorised amount.

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

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

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 $P(x, y, z)$
  - 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 ⇒ 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 auxiliary 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: CASE Tools

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

Figure: Analysis and B development

Position - other methods

- B: Correct-by-construction Approach $\rightarrow$ \textit{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 $\rightarrow$ \textit{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
```

**OPERATIONS**

```
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: refinement

**REFINEMENT** /* raffinement de ...*/

```
pgcd1_R1
REFINES pgcd1 /* the former machine */
OPERATIONS
  rr <-> pgcd(xx,yy) = /* the interface is not changed */
  ...
END
```

```
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 */
  END
  END
```

```
```

```
```
After the examples

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 identifiers */
PROPERTIES
B /* preedicate(s) on K */
VARIABLES
V /* list of variables identifiers */
DEFINITIONS
D /* list of definitions (macros) */

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

∃ prm.C
It is possible to have values for parameters that meet the constraints

C ⇒ ∃ (ENS, K).B
There are sets and constants that meet the properties of the machine

B ∧ C ⇒ ∃ V.I
There are a state that meets the invariant

B ∧ C ⇒ [U]I
The initialisation establishes the invariant

For each operation of the machine

B ∧ C ∧ I ∧ P ⇒ [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.

\[ I \land P \Rightarrow [\text{Subst}]I \]

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

Semantics of a machine - Consistency

To formally establish the condition for the correct functioning 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

**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)))

INITIALISATION

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

OPERATIONS

moveToInsertion =

PRE

mode = extraction

THEN

mode := insertion ||

unsorted := {} ||

sorted :: seq(NAT)

END

...
Data Modeling Language

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 \(\text{POW}(E)\)

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.

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.

First Order Logic

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
<th>Ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>(p \land q)</td>
<td>(p &amp; q)</td>
</tr>
<tr>
<td>or</td>
<td>(p \lor q)</td>
<td>(p \lor q)</td>
</tr>
<tr>
<td>not</td>
<td>(\neg p)</td>
<td>(\neg p)</td>
</tr>
<tr>
<td>implication</td>
<td>(p \Rightarrow q)</td>
<td>((p) ==&gt; (q))</td>
</tr>
<tr>
<td>univ. quantif.</td>
<td>(\forall x.p(x))</td>
<td>!x.(p(x))</td>
</tr>
<tr>
<td>exist. quantif.</td>
<td>(\exists x.p(x))</td>
<td>#x.(p(x))</td>
</tr>
</tbody>
</table>

Variables should be typed:

\#x.(x : T ==> p(x)) and !x.(x : T ==> p(x))
The standard set operators

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
<th>Ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>$E \cup F$</td>
<td>$E \bigcup F$</td>
</tr>
<tr>
<td>intersection</td>
<td>$E \cap F$</td>
<td>$E \bigcap F$</td>
</tr>
<tr>
<td>membership</td>
<td>$x \in F$</td>
<td>$x : F$</td>
</tr>
<tr>
<td>difference</td>
<td>$E \setminus F$</td>
<td>$E \setminus F$</td>
</tr>
<tr>
<td>inclusion</td>
<td>$E \subseteq F$</td>
<td>$E \subseteq F$</td>
</tr>
<tr>
<td>selection</td>
<td>choice($E$)</td>
<td>choice($E$)</td>
</tr>
</tbody>
</table>

+ generalised Union and intersection
+ quantified Union et intersection

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
<th>Ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>not member</td>
<td>$x \notin F$</td>
<td>$x /: F$</td>
</tr>
<tr>
<td>non inclusion</td>
<td>$E \not\subseteq F$</td>
<td>$E /:\subseteq F$</td>
</tr>
<tr>
<td>non equality</td>
<td>$E \neq F$</td>
<td>$E /= F$</td>
</tr>
</tbody>
</table>

Generalised Union

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

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

\[ \Rightarrow \]

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

\textbf{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.

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

\[ \Rightarrow \]

\[ \bigcup x.(x \in S \mid E) = \{ y \mid y \in T \land \exists x.(x \in S \land y \in E)\} \]

\textbf{Example}

\[
\text{UNION}(x).(x \in \{1, 2, 3\} \mid \{ y \mid y \in \text{NAT} \land y = x \times x\})
\]

= \{1\} \cup \{4\} \cup \{9\} = \{1, 4, 9\}
Generalised Intersection

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

\[ S \in \mathcal{P}(\mathcal{P}(T)) \Rightarrow \text{inter}(S) = \{ x \mid x \in T \land \forall u.(u \in S \Rightarrow x \in u) \} \]

Example

\[ \text{inter}([\{aa, ee, ff, cc\}, \{bb, gg\}, \{dd, ee, uu, cc\}] = \{cc\} \]

Quantified Intersection

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

\[ \forall x.(x \in S \Rightarrow E \subseteq T) \Rightarrow \bigcap x.(x \in S \mid E) = \{ y \mid y \in T \land \forall x.(x \in S \Rightarrow y \in E) \} \]

Example

\[ \text{INTER}(x).(x \in \{1, 2, 3, 4\} \mid \{y \mid y \in \{1, 2, 3, 4, 5\} \land y > x\}) = \text{inter}((\{1, 2, 3, 4, 5\}, \{2, 3, 4, 5\}, \{3, 4, 5\}, \{4, 5\})) \]

Relations

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
<th>Ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>relation</td>
<td>( r : S \leftrightarrow T )</td>
<td>( r : S \leftrightarrow T )</td>
</tr>
<tr>
<td>domain</td>
<td>( \text{dom}(r) \subseteq S )</td>
<td>( \text{dom}(r) \subseteq S )</td>
</tr>
<tr>
<td>range</td>
<td>( \text{ran}(r) \subseteq T )</td>
<td>( \text{ran}(r) \subseteq T )</td>
</tr>
<tr>
<td>composition</td>
<td>( r \circ s )</td>
<td>( r \circ s )</td>
</tr>
<tr>
<td>composition r(s)</td>
<td>( r(s) )</td>
<td>( r(s) )</td>
</tr>
<tr>
<td>identity</td>
<td>( \text{id}(S) )</td>
<td>( \text{id}(S) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
<th>Ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain restriction</td>
<td>( S &lt; r )</td>
<td>( S &lt; r )</td>
</tr>
<tr>
<td>range restriction</td>
<td>( r \uparrow T )</td>
<td>( r \uparrow T )</td>
</tr>
<tr>
<td>domain antirestriction</td>
<td>( S &lt;</td>
<td>r )</td>
</tr>
<tr>
<td>range antirestriction</td>
<td>( r \downarrow T )</td>
<td>( r \downarrow T )</td>
</tr>
<tr>
<td>inverse</td>
<td>( r^{-1} )</td>
<td>( r^{-1} )</td>
</tr>
<tr>
<td>relationnelle image</td>
<td>( r[S] )</td>
<td>( r[S] )</td>
</tr>
<tr>
<td>overriding</td>
<td>( r1 \oplus r2 )</td>
<td>( r1 \oplus r2 )</td>
</tr>
<tr>
<td>direct product of rel.</td>
<td>( r1 \otimes r2 )</td>
<td>( r1 \otimes r2 )</td>
</tr>
<tr>
<td>closure</td>
<td>( \text{closure}(r) )</td>
<td>( \text{closure}(r) )</td>
</tr>
<tr>
<td>reflexive trans. closure</td>
<td>( \text{closure1}(r) )</td>
<td>( \text{closure1}(r) )</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
<th>Ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>partial function</td>
<td>( S \rightarrow T )</td>
<td>( S \rightarrow T )</td>
</tr>
<tr>
<td>total function</td>
<td>( S \rightarrow T )</td>
<td>( S \rightarrow T )</td>
</tr>
<tr>
<td>partial injection</td>
<td>( S \mapsto T )</td>
<td>( S \rightarrow T )</td>
</tr>
<tr>
<td>total injection</td>
<td>( S \rightarrow T )</td>
<td>( S \rightarrow T )</td>
</tr>
<tr>
<td>partial surjection</td>
<td>( S \rightarrow T )</td>
<td>( S \rightarrow T )</td>
</tr>
<tr>
<td>total surjection</td>
<td>( S \rightarrow T )</td>
<td>( S \rightarrow T )</td>
</tr>
<tr>
<td>total bijection</td>
<td>( S \rightarrow T )</td>
<td>( S \rightarrow T )</td>
</tr>
<tr>
<td>lambda abstraction</td>
<td>( %x.(P</td>
<td>E) )</td>
</tr>
</tbody>
</table>

### Sequences

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
<th>Ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence of elements of ( T )</td>
<td>( \text{seq}(T) )</td>
<td>( \text{seq}(T) )</td>
</tr>
<tr>
<td>empty sequence</td>
<td>( [] )</td>
<td>( [] )</td>
</tr>
<tr>
<td>injective sequence of element of ( T )</td>
<td>( \text{iseq}(T) )</td>
<td>( \text{iseq}(T) )</td>
</tr>
<tr>
<td>bijective sequence of element of ( T )</td>
<td>( \text{perm}(T) )</td>
<td>( \text{perm}(T) )</td>
</tr>
<tr>
<td>size of a sequence ( s )</td>
<td>( \text{size}(s) = \text{card(dom}(s)) )</td>
<td>( \text{size}(s) = \text{card(dom}(s)) )</td>
</tr>
</tbody>
</table>

### Sequences (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>first element of a seq. ( s )</td>
<td>( \text{first}(s) = s(1) )</td>
</tr>
<tr>
<td>last element of a seq. ( s )</td>
<td>( \text{last}(s) = s(\text{size}(s)) )</td>
</tr>
<tr>
<td>restrict. of ( s ) its ( n ) first elem. elements</td>
<td>( s \uparrow n )</td>
</tr>
<tr>
<td>elimination of the first ( n ) elements of ( s )</td>
<td>( s \downarrow n )</td>
</tr>
</tbody>
</table>

### Basic Concepts of the Dynamic Part

**Modeling Operations**
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 describes:

the set of all states | the execution of S beginning with one of them terminates in a finite time in a state satisfying R,

\(wp(S, R)\) is the **weakest precondition** of S with respect to R.

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
- 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 \(\{P\} S \{R\}\).

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 \(z = \text{max}(x, y)\)

\[wp(S, R) = \text{Vrai}\]
**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 $[x, y := E, F]R$

---

**Non determinism - Substitutions**

- **Abstraction** $\Rightarrow$ (possible)non determinism. OK for specifying.
- **Concretisation** $\Rightarrow$ refinement into code
- **Extending the basic GSL set to other substitutions closed to programming**
  - CASE OF
  - SELECT
  - IF THEN ELSE

---

**B: generalized substitutions - Basic set of GS**

The abstract syntax language to specify the operations:

Let $R$ be the invariant, $S$, $T$ substitutions

<table>
<thead>
<tr>
<th>Name</th>
<th>Abs. Synt.</th>
<th>definition</th>
<th>equivalent in logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>neutral (id.)</td>
<td>$skip$</td>
<td>$[skip]R$</td>
<td>$R$</td>
</tr>
<tr>
<td>Pre-condition</td>
<td>$P \mid S$</td>
<td>$[P \mid S]R$</td>
<td>$P \land [S]R$</td>
</tr>
<tr>
<td>Bounded choice</td>
<td>$S \parallel T$</td>
<td>$[S \parallel T]R$</td>
<td>$[S]R \land [T]R$</td>
</tr>
<tr>
<td>Guard</td>
<td>$P \Rightarrow T$</td>
<td>$[P \Rightarrow T]R$</td>
<td>$P \Rightarrow [T]R$</td>
</tr>
<tr>
<td>Unbounded</td>
<td>$@x.S$</td>
<td>$[@x.S]R$</td>
<td>$\forall x.[S]R$</td>
</tr>
</tbody>
</table>

$x$ bounded (not free) in $R$ enough as B specification language but ...

---

**Syntactic extension of substitutions**: basic substitution set

**Basis Substitution**

- noted $S$

**Simultaneous Substitutions**

Consider $S$ and $T$ two substitutions.

**Syntactic Extension**

- $S$ being $x := E$ and $T$ being $y := F$
- note $S \parallel T$
B - generalized substitution Language

**Neutral Substitution**

- Syntactic extension
  - skip

**Subst. with precondition**

- Syntactic extension
  - $P \mid S$

**Bounded choice**

- Syntactic extension
  - $S \parallel T$

**Guarded Substitution**

- Syntactic extension
  - $(P \Rightarrow T) \parallel (\neg P \Rightarrow S)$

**Unbounded Choice Substitution**

- Syntactic extension
  - $\@.x. S_x$

**Non-deterministic @**

- Syntactic extension
  - $\@.x. (P_x \Rightarrow S_x)$
Extending the basic substitution set: non-deterministic

Nondeterministic \( x \in U \) (becomes member)
\[ x : \in U \]
@\( y. (y \in U \rightarrow x := y) \)

Syntactic extension

ANY \( y \)
WHERE \( y : U \)
THEN \( x := y \)
END

Extensions... non-deterministic

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

Proof Obligations

MACHINE ThreshCtrl
CONSTANTS thresX, thresY
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
...Proof Obligation (PO)...
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 prove that the **INITIALISATION** establishes the invariant: $[\text{Init}]I$
- to prove that each **OPERATION**, when it is called under its precondition, preserves the invariant.

$$I \land P \Rightarrow [\text{Subst}]I$$

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

Proof of the operation \(\text{setXX}(nx)\)

We must prove that $I \land P \Rightarrow [\text{Subst}]I$

\[
\begin{align*}
\text{INVARIANT} & \quad xx : \text{INT} \land 0 \leq xx \land xx \leq \text{thresX} \\
\text{setXX}(nx) = \\
\text{PRE} & \quad \ldots \ ? \\
\text{THEN} & \quad xx := nx \quad /\* \quad \text{Subst} \quad */ \\
\text{END} & \\
\end{align*}
\]

\[
xx : \text{INT} \land 0 \leq xx \land xx \leq \text{thresX}
\]

(\text{use white/blackboard})

Precondition computation / preservation of the invariant

\[
\begin{align*}
xx : \text{INT} & \land 0 \leq xx \land xx \leq \text{thresX} \\
\text{setXX}(nx) = \\
\text{PRE} & \quad \ldots \ ? \\
\text{THEN} & \quad xx := nx \quad /\* \quad \text{Subst} \quad */ \\
\text{END} & \\
\end{align*}
\]

\[
nx : \text{INT} \land 0 \leq nx \land nx \leq \text{thresX}
\]

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

\[
nx : \text{INT} \land 0 \leq nx \land nx \leq \text{thresX} \quad ?
\]

It is the precondition!

Precondition computation / preservation of the invariant

\[
\begin{align*}
xx : \text{INT} & \land 0 \leq xx \land xx \leq \text{thresX} \\
\text{incrXX}(px) = \\
\text{PRE} & \quad \ldots \ ? \\
\text{THEN} & \quad xx := xx + px \quad /\* \quad \text{Subst} \quad */ \\
\text{END} & \\
\end{align*}
\]

\[
xx : \text{INT} \land 0 \leq xx \land xx \leq \text{thresX}
\]

\[
xx + px : \text{INT} \land 0 \leq xx + px \land xx + px \leq \text{thresX} \quad ?
\]

hence the precondition:

\[
px : \text{INT} \land 0 \leq xx + px \land xx + px \leq \text{thresX}
\]
Example of resources allocation (recall)

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

OPERATIONS
addRsc(rr) = // adding resources
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

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

Case Studies

...Cas Euclide...

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

Replace variables with their values in I:

\( rsc \backslash \{rr\} <\) RESC & card(rsc \backslash \{rr\}) <= maxRes

(use white/blackboard)
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

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

Example of development with B

raffine
implante
importe
machine abstraite

euclide

demoEuclide
implante

InterfaceEuclide
implante

InterfaceEuclide_I

machine abstraite

implante

euclide_R1

implante

euclide_I1_1

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

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).

- 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: How to refine?

Introducing data structures and replacing abstract structures by concrete ones.

- Use the clause **REFINES** to link the abstract machine with 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
    by 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.

⇒ extending the substitution language.

Examples of refinement

- Already seen:
  - Resource Allocation
  - Euclidean 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

```
\( n_{rsrc} \in 0..100 \)
\( n_{rsrc} = \text{cardinal of the set} \)
allocate \( \rightarrow -1 \text{ element} \)
free \( \rightarrow +1 \text{ element} \)
```
### Machine Allocation

#### Variables

- \( n_{rsrc} \)

#### Invariant

\( n_{rsrc} : 0..100 \)

#### Initialisation

\( n_{rsrc} := 100 \)

#### Operations

**allocate**

- **Pre**: \( n_{rsrc} > 0 \)
- **Then**: \( n_{rsrc} := n_{rsrc} - 1 \)

**free**

- **Pre**: \( n_{rsrc} < 100 \)
- **Then**: \( n_{rsrc} := n_{rsrc} + 1 \)

- \( bb \leftarrow \text{available} = \)
  - \( bb :: \text{BOOL} \)
  - // ou \( bb := \text{bool}(0 < n_{rsrc}) \)

### Consistency Proof

The developer of the abstract machine has to
two kinds of PO:

To prove that the INITIALISATION establishes the invariant

\[ [n_{rsrc} := 100] (n_{rsrc} \in 0..100) \]

we should prove that \( 100 \in 0..100 \)

### Resource allocation (Refinement)

- **allocate** → find 1 free element
- **free** → find 1 unavailable element

We have to prove that each operation called under its **Pre** condition, preserve the invariant.

- For the operation **allocate** we should prove:
  \[ n_{rsrc} \in 0..100 \land 0 < n_{rsrc} \Rightarrow n_{rsrc} - 1 \in 0..100 \]

- For the operation **available** we should prove:
  \[ n_{rsrc} \in 0..100 \land (n_{rsrc} > 0 \lor \neg (n_{rsrc} > 0)) \]
  \[ \Rightarrow \]
  \[ n_{rsrc} \in 0..100 \]
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

allocate = // rewritten with the new variables
ANY ss WHERE
ss : rs_free // non-deterministic 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}
br := TRUE
|| rs_unavailable := rs_unavailable - {ss}
|| n_rsrc := n_rsrc + 1
END

Implantation
Tableau (structure prédéfinie)
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

Sequential substitutions

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

$$[S; T]R \equiv [S][T]R$$

$$\equiv [S][T[R]$$

$S$ establishes $[T]R$

Loop substitution

The loop substitution has the following shape:

while $P$ do
  $S$
  invariant $I$
  variant $V$
end

Semantically, it is

$$I \land /* \text{the variant is a natural} */$$

$$\forall x.(I \Rightarrow V \in \text{NATURAL}) \land /* \text{the variant decreases after each step} */$$

$$\forall (x,n).(I \land P \Rightarrow [n \Leftarrow V][S](V < n)) \land /* \text{continuation of the loop} */$$

$$\forall x.(I \land P \Rightarrow [S]I) \land /* \text{continuation of the loop} */$$

$$@x'.([x := x'](I \land \neg P) \Rightarrow x := x'))$$
**Block with local variables**

The notation is:

\[
\text{var } x \text{ in } S \\
\text{end}
\]

**Composition of machines**
- Modules - Composition - Layered Architecture
- Modularity

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

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

**MACHINE**

\[
\text{MA} \\
\text{INCLUDES } \text{MB} \\
\text{PROMOTES Opmb1, Opmb3} \\
\text{/* access by Opmb to varB */} \\
\text{/* become operations of MA */} \\
\text{END}
\]
Hierachy

**EXTENDS**, inclusion but *no need to promote*

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

**SEES for a read only sharing**

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

Sharing

**USES** for a *read/write* sharing

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

MA et MB should be included in another machine.