Introduction à la Méthode B

Preuves et Constructions Formelles

J. Christian Attiogbé

Université de Nantes, November 2020

Plan

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)
- **J-R. ABRIAL**

B Method: some references

- *The B Method*,
J-R. Abrial, Cambridge University Press, 1996

- *Formal Development in B of a Minimum Spanning Tree Algorithm*,
R. Fraer, in 1st Conference on the B Method, 1996

- *Formal Derivation of Spanning Trees Algorithms*,
J-R. Abrial and D. Cansell and D. Mery, ZB'2003

- *Faultless Systems: Yes We Can!*,
Jean-Raymond Abrial, Computer, vol. 42, no. 9, pp. 30-36, Sept. 2009

Industrial Applications

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

☞ Highly needed skills in the Industry
(Aeronautics, Telecoms, Space, Health, Automotive, etc).

Method in Software Engineering

Formal Method=

- Formal Specification or Modelling Language
- Formal reasoning System (Logics)

B Method=

- Specification Language
  - Logic, Set Theory: data language
  - Generalized Substitution Language: Operations'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 ones

  = Refinement into code in a programming language.

B: Formal Method

+ refinement theory (of abstract machines)

⇒ formal development method

The GCD Example

Formal development

mathematical model → programming model

Illustration: From an abstract machine to its refinement into code.

gcd(x,y) is d | x mod d = 0 ∧ y mod d = 0
∧ ∀ other divisors dx d > dx
∧ ∀ other divisors dy d > dy

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) = /* specification of gcd */
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 od are : od < d */
    & !od.((od : INT & od < MAXINT
        & ((xx- (xx/od)*od) = 0)   &  (yy-(yy/od)*od) = 0) => od < dd)
    THEN rr := dd
    END
END
```

Constructing the GCD: refinement

```
REFINEMENT /* refinement of ...*/
    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 */
        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 divids 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
            ; rr := cr
        END
```

State space

The value of a variable may be changed ⇒ it takes different *states*

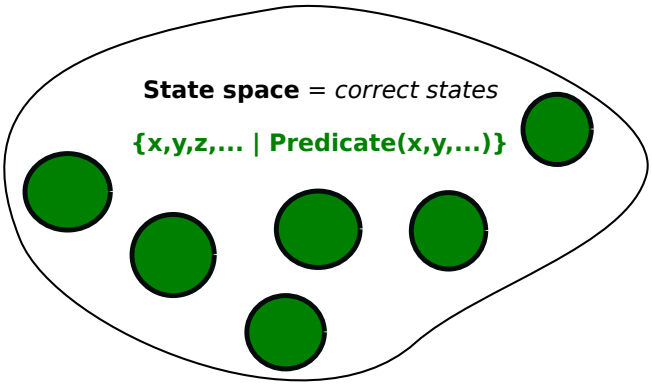


Figure: A view of a state space

A correct software: composition of operations that relate the states.

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

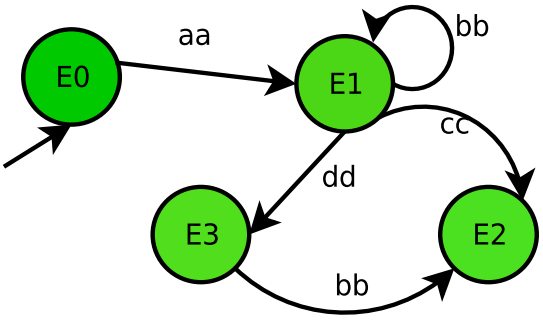


Figure: Evolution of a software system

The B Method

Concepts and basic principles :

- abstract machine (state space + abstract operations).
  A **consistency proof is required** (+ safety properties).

- proved refinement (from abstract to concrete model)
  Each refinement step **should be proved correct**.

Refinement = Development method = Design

B Method: Global Approach

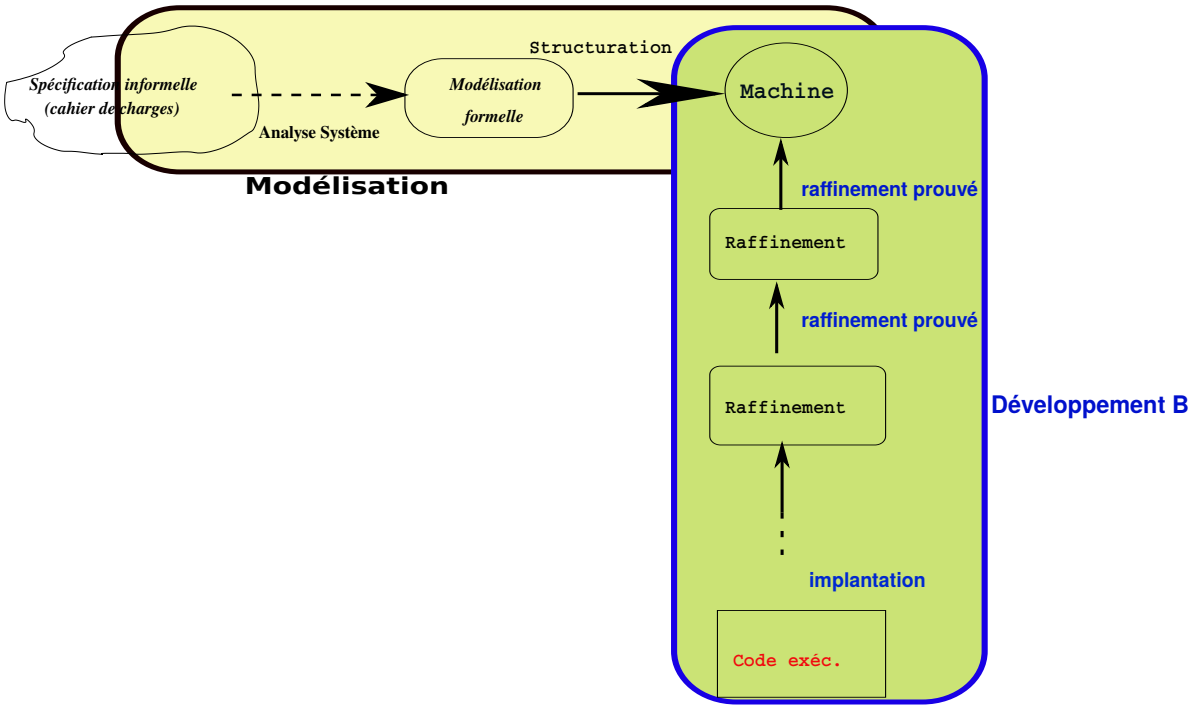


Figure: Analysis and B development

B Method: the foundations

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

Machine abstraite : exemple de la jauge

Exemple

Soit à contrôler la valeur d'une variable (jauge) entre 2 et 45 alors qu'on y effectue des opérations d'incrémentation et de décrémentation.

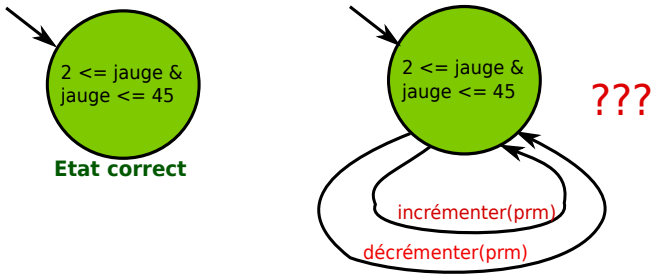


Figure: Rester dans un état correct

Exprimer les états corrects par un prédicat (invariant)
Exprimer les conditions de correction des opérations.

Machine abstraite : exemple de la jauge

```
MACHINE NomMach
VARIABLES
    jauge
INVARIANT
    jauge : NAT
&    jauge >= 2
&    jauge <= 45
INITIALISATION
jauge := 1 // quoi ???
```

```
OPERATIONS
decrementer1 =
PRE jauge > 2
THEN jauge := jauge - 1
END
; decrementer(pas) =
PRE pas : NAT
& jauge - pas >= 2
THEN
     jauge := jauge - pas
END
...
incrementer ...
...
END
```


Light Regulation System

Case Study

Requirements:
A room is equipped with lights and air condition. We will distinguish day mode and night mode. The devices are controlled by a software.

- The light can be turned off or on.
- The temperature can be incresead or decreased.
  (we will give more constraints later)
- ...

Specification Approach

Light and temperature regulation in a room.

- 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 = models a transition system
(with a logical approach)

Abstract Machine

variables

predicates

operation

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

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
;
turnOn =
    BEGIN light := on END
;
turnOff =
    light := off
;
decreaseTemp = temp := temp - 1
;
increaseTemp = temp := temp +1
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
// note a very big space!
```

- 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.
  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 simultaneaously for all the variables.
  They can be modified later by the operations.

Abstract Machine

```
OPERATIONS
changeMode =
CHOICE mode := day
OR mode := night
END
;
turnOn =
     light := on
;
turnOff =
     light := off
;
decreaseTemp = temp := temp - 1
;
increaseTemp = temp := temp +1
END // machine
```

- 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).

B: principle of the method

To **use an invariant to control** the behaviour a software (or a system)

- To model the space of correct states with a predicate (a conjunction of properties).
- To maintain the system within these states! it runs safely.
  (ie to avoid the system going out from the defined state space).
- To 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: everything in Logic (data and operations)

- state space= Invariant= Predicate : $P(x, y, z)$
  A state is a valuation of variables
  $x := v_x \quad y := v_y \quad z := v_z$ in $P(x, y, z)$
  $\Rightarrow$ Logical substitution
- An operation: transforms a correct (state) into another one.
  $\Rightarrow$ predicate transformer (the invariant is transformed)
  B Operation = predicate transformer (with substitutions)
  other effects than assignment $\Rightarrow$ generalized substitutions

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
    $u \leftarrow O(pp) =$ /* 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 variables that meet the Invariant

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

The initialisation establishes the invariant (a state meets the invariant)

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

Each operation called under its precondition preserves the invariant

Proof Obligations (PO)

There are the predicates to be proven to ensure the consistency (correction) of the mathematical model defined by the abs. machine.

The designer of the machine has two types of proof obligations:

- prove that the INITIALISATION establishes the invariant;

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

- 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 master the proof obligations.

Data Modelling Language

- First order Logic

$p \wedge q$ $p \vee q$ $\neg p$ $p \Rightarrow q$ $\forall x.p(x)$ $\exists x.p(x)$ $\cdots$

- Predefined sets - there are types / User-defined Sets

$\mathbb{N}$ $\mathbb{Z}$ RESSOURCES PROCESSUS OBJETS $\cdots$

- Set operators + theory :

$E \cup F$ $E \cap F$ $x \in F$ $E \setminus F$ $E \subseteq F$ $\cdots$

- Relations, Functions :

$r : S \leftrightarrow T$ $f : S \rightarrow T$ $f : S \nrightarrow T$ $\cdots$

Nous étudierons en TD/TP

Révisions : Maths discètes (L1, L2) - cf fiche mémo sur le site du cours.

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 $\mathscr{P}(E)$
  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

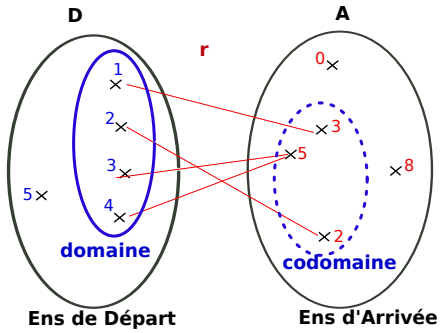
B - Data Language

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`

Relations definition, vocabulary

A relation r over D and A is a subset of the cartesian product $D \times A$
it is noted $r : D \leftrightarrow A$ or $r \subseteq D \times A$
r is a set of couples (d, a) also denoted by $d \mapsto a$



$r = \{(1,3),(2,2),(3,5),(4,5)\}$ ou
$r = \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 5, 4 \mapsto 5\}$

dom(r) = {1, 2, 3, 4}
ran(r) = {3, 5, 2}

Figure: Euler-Venn diagram of r

Domaine : domaine Codomaine : range

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	$S \lhd r$	S <<| r
range antirestriction	$r \rhd T$	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	$S \pfun T$	S +-> T
total function	$S \fun T$	S --> T
partial injection	$S \pinj T$	S >+-> T
total injection	$S \inj T$	S >-> T
partial surjection	$S \psur T$	S +->> T
total surjection	$S \sur T$	S -->> T
total bijection	$S \bij T$	S >->> T
lambda abstraction	$\%x.(P \mid E)$	

Modelling Operations : generalized substitutions

Basic concepts of the dynamic part of a B abstract machine

- Change of states; state transitions; with logical substitution! Generalisation of the classical substitution of the Logic - GSL to model the behaviours of operations.
- + Abstractions with Non-determinism
- + Proof Obligations to guarantee the Invariant

Before-after predicates.

$$\mathrm{Prd}_x(S) \mathrel{\widehat{=}} \neg\,[S](x' \neq x)$$

$\mathrm{Prd}_x(x := E) \Leftrightarrow (x' = E)$
Then induction on the structure of S (skip, P|S, S [] T, ...)
+ Termination and feasibility : $fis(S) \Leftrightarrow \exists x'.\mathrm{Prd}_x(S)$

B: Generalized Substitutions - Axioms

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:

Let 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 ...

Extending the basic substitution set: non-deterministic

Nondeterministic @

$@x.(P_x \Longrightarrow S_x)$

Syntactic extension

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

Nondeterministic $x :\in U$

(becomes member)

x :: U

$@y.(y \in U \Longrightarrow x := y)$

Syntactic extension

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

Nondeterministic $x : P(x)$

x : P(x) // (x such that P)

Extending the GSL set to programming substitutions

- **Concretisation** ⇒ refinement into code
- Extending the basic GSL set to other substitutions closed to programming
  - CASE OF
  - SELECT
  - IF THEN ELSE

B: CASE Tools

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

Available tools:

- Atelier B
- Rodin
- ProB
- ...

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) ;
- ...