Introduction à la spécification et à la conception: la Méthode B

Formal Software Construction

J. Christian Attiogbé

Université de Nantes, November 2017

Plan

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

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)

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.

Method in Software Engineering

Formal Method=

- Formal Specification or Modeling Langaguage
- Formal reasoning System

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 structures

  = 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

Refinement = Development method = design

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 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 /* 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
```

B Method: Global Approach

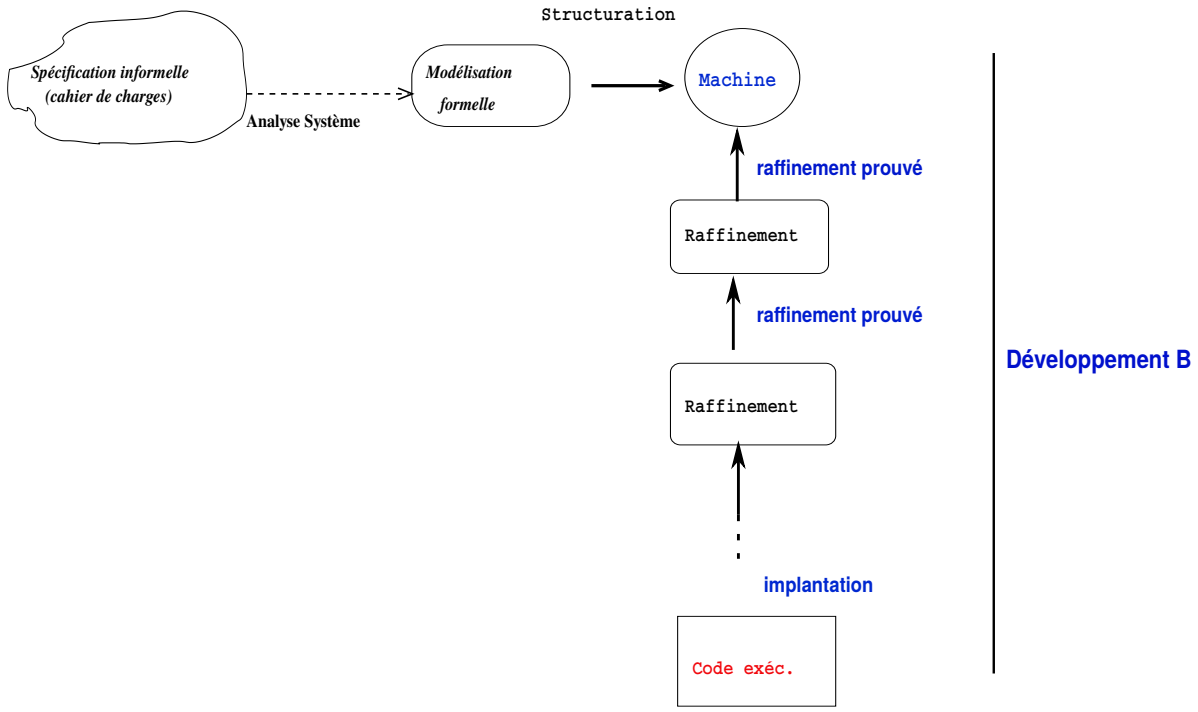


Figure: Analysis and B development

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 $\langle I,\ J\rangle$, $\langle V1,\ V2,\ V4,\ ...\rangle$

State space

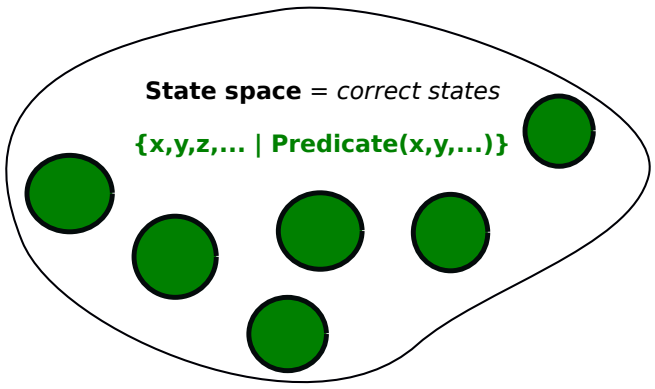


Figure: A view a state space

A software provides operations that relate the correct 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

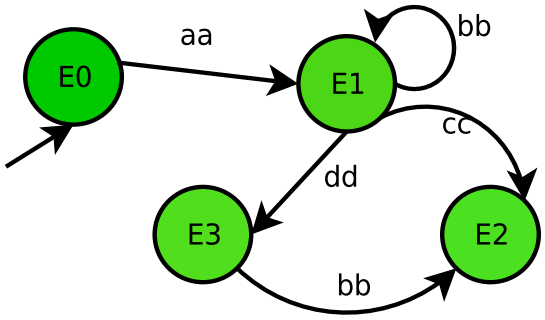


Figure: Evolution of a software system

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 = 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
// 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 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 // 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).

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
```


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: provides operations

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

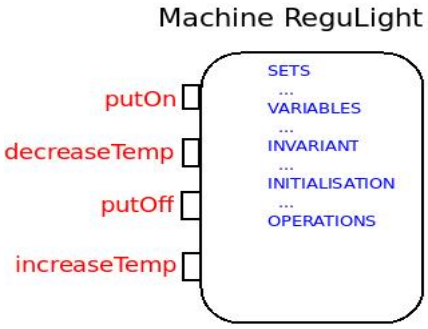


Figure: The operations are callable from outside

☛ An operation cannot call another operation of the same machine

Interface of operations

(operations with or without input/output parameters)

- No parameter:
  $$nameOfOperation = ...$$
- Input parameters only:
  $$nameOfOperation(p1, p2, \cdots) = ...$$
- Output parameters only:
  $$r1, r2, \cdots \longleftarrow nameOfOperation = ...$$
- Input and Output parameters:
  $$r1, r2, \cdots \longleftarrow nameOfOperation(p1, p2, \cdots) = ...$$

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)

Modelling a ressource allocator

Case Study

Requirements:

- An allocator should manage a stock of resources.
- One can add a ressource to the stock, or remove a resource from the stock

Modelling a ressource allocator

Case Study

Requirements:

- An allocator should manage a stock of resources.
- One can add a ressource to the stock, or remove a resource from the stock

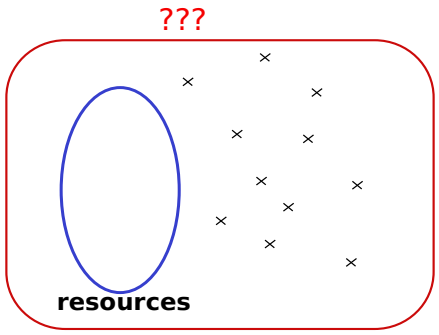


Figure: Types, Sets ad subsets

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
```

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
  $x := v_x \quad y := v_y \quad z := v_z$ in $P(x, y, z)$
  $\Rightarrow$ Logical substitution
- An operation: transforms a correct (state) into another one.

  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, ...

Available tools:

- Atelier B
- Rodin
- ProB
- ...

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 a state that meets the invariant

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

The initialisation establishes the invariant

Each operation called under its precondition **preserves the invariant**

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

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

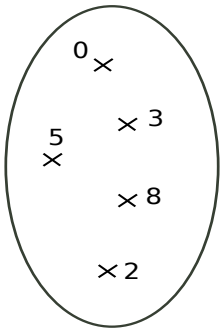
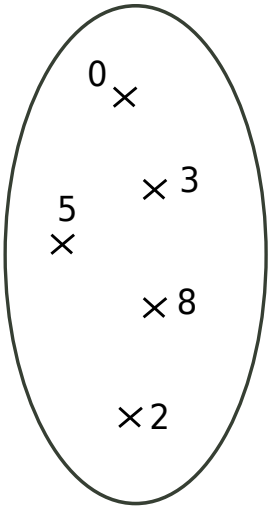


Figure: Modeling the Sorting of (a set of) Naturals

Example of Specifying Sorting with B



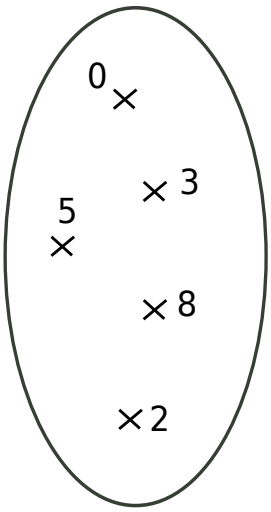
set of naturals :
FIN(NAT)

0 2 3 5 8

ordered

Figure: Modeling the Sorting: ordering the set of Naturals

Example of Specifying Sorting with B



set of naturals :
FIN(NAT)

0 2 5 8 ?

ordered but WRONG!

Figure: Modeling the Sorting: be careful!

Example of Specifying Sorting with B

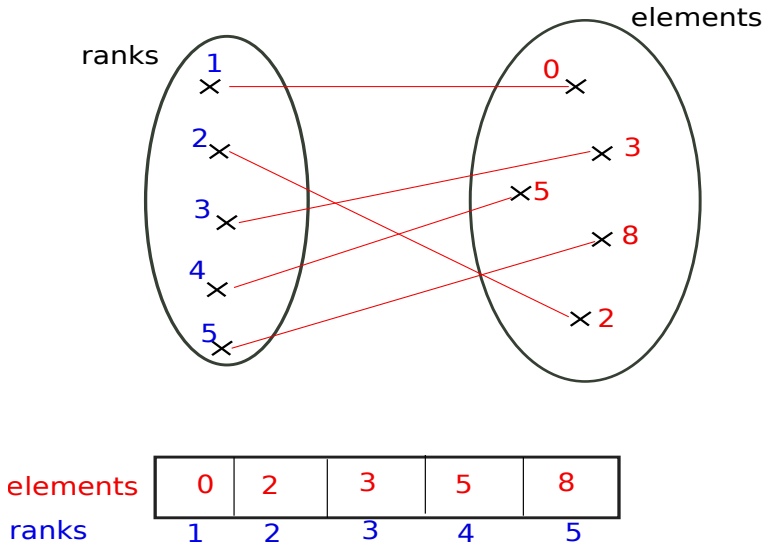


Figure: Modeling the Sorting

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
```

Modeling Operations

Basic Concepts of the Dynamic Part

- change of states; state transitions; logical substitution!
- Non-determinism
- Proof Obligations

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.

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 ...

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
```

B - generalized substitution Language

Extensions... non-deterministic

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

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

Modélisation de l'évolution de processus

Considérons le contexte d'un système d'exploitation.
On a des processus qui vivent et se terminent.
Soit PROCESSUS l'ensemble de base de tous les processus. On consirédera un sous-ensemble *processus* correspondants au processus effectivement manipulés dans le système.
Ainsi un processus est caractérisé par un numéro unique et un numéro caractérise un seul processus ; tous les processus manipulés ont un état : prêt, actif, bloqué.
A tout moment tout processus est dans un seul état ; plusieurs processus peuvent être dans le même état.
Tout nouveau processus créé se trouvera dans l'état prêt et a un numéro unique (différent de ceux des processus existants déjà). On ne pourra pas créer plus de maxP processus.

Modélisation de l'évolution de processus

On suppose qu'il y a un ordonnanceur qui fait le changement d'état des processus ; par exemple l'ordonnanceur peut, lorsque c'est possible (on ne détaille pas ici comment), faire passer un quelconque des processus de l'état prêt vers l'état actif ; un quelconque des processus de l'état bloqué vers l'état prêt.

Dessinez le diagramme de Euler-Venn exprimant la relation/fonction entre les processus et leurs états.

Modélisation de l'évolution de processus

Soit PROCESSUS l'ensemble de processus ; $processus \subseteq PROCESSUS$

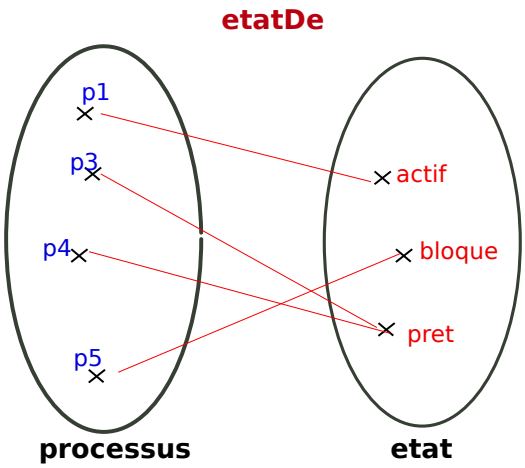


Figure: Etat des processus

Modélisation de l'évolution de processus

- fonction totale

$$etatDe : processus \rightarrow etats$$

- $etatDe^{-1}[\{actif\}] =$???
- $prets = etatDe^{-1}[\{pret\}]$
-

```
ANY pp
WHERE prets /= {} ∧ pp ∈ prets
      ∧ etatDe⁻¹[{actif}] = {}
THEN
    etatDe(pp) := actif
END
```

Conclusion

- Un tour rapide de la méthode B
- Approche orientée modèle (à états) - Spécification formelle
- Preuve de propriétés de sûreté (les invariants)
- Place à la pratique
- Employer les outils disponibles en domaine public (AtelierB, Rodin, ProB)
- Suite du cours : Systèmes de contrôle embarqués - Event B
- Atelier pratique Jeudi