Construction de systèmes corrects : Introduction à la Méthode B

Preuves et Constructions Formelles

J. Christian Attiogbé

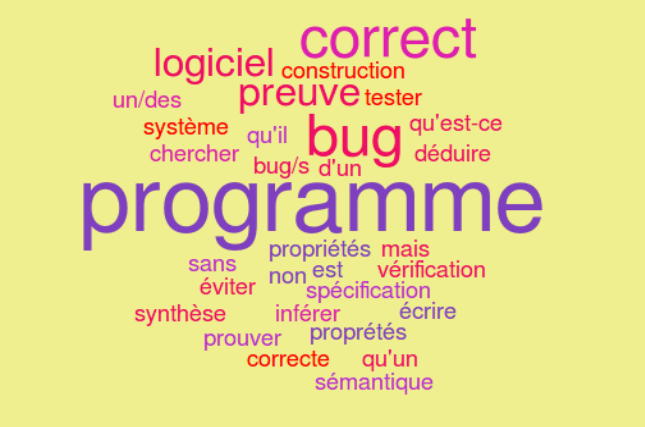
Nantes Université, Octobre 2023

Nantes Université

LS2N LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

Plan

De quoi allons nous parler



Tout un (petit) univers

Qu'est-ce qu'un logiciel ? qu'est-ce qu'un système ? qu'est-ce un *bug* ? logiciel correct ? logiciel non correct ? éviter bug ? chercher un/des bug/s ? mais qu'est-ce qu'un bug ? tester programme ? écrire programme sans bug ? prouver programme (qu'il est correct) ? spécification ? vérification (de proprétés) ? preuve de propriétés ? système de preuve ? construction correcte ? synthèse de programme ? sémantique d'un programme ? inférer ? déduire ?

☞ Logique, Modèle, Langage

☞ Analyser, exprimer, raisonner, concevoir, construire

“Ce que l’on conçoit bien s’énonce clairement,
et les mots pour le dire arrivent aisément”,

(Art poétique, Nicolas Boileau, 1674)

Vérification et preuves formelles

Vérification de propriétés
- par la preuve
- sur la base de modèles (formels)

des systèmes
- programmes, logiciels, applications,
- industriels,
- naturels

propriétés
- intrinsèques : cohérence, complétude
- spécifiques : selon les fonctionnalités du système

ex. dans des systèmes
- un seul véhicule dans 1 tunnel
- drone au dessus du public
- choc entre drones dans 1 flotte

Vérification et preuves formelles

Modèles formels
- par des abstractions : humaines
- formalisation : mathématiques

Propriétés formelles
- en Logique
- fonctionnelles, non fonctionnelles

Correction de système
- par rapport à ce qu'il doit faire !
- qu'est-ce qu'il doit faire ?

Spécification
- de ce qu'un système doit faire ou pas faire
- hypothèses sur son environnement

Modélisation - preuve

Modéliser (Logique, Graphes, Automates, Equations différentielles, etc)

Spécifier (Logique, Graphes, etc)

Analyser - Vérifier (preuve de théorèmes - exploration de modèles)

Construire / Développer (raffinements, compositions)

☞ Quelques compétences à acquérir dans cet enseignement

Formal Methods are required in some contexts

Common Criteria Evaluation Assurance Level	Process rigor required for development of an IT product
EAL1	Functionally tested
EAL2	Structuraly tested
EAL3	Methodically tested and checked
EAL4	Methodically designed, tested and reviewed
EAL5	Semiformally designed and tested
EAL6	semiformally verified, designed and tested
EAL7	Formally designed and tested

https://militaryembedded.com/avionics/safety-certification/whats-needed-ensure-safety-security-uav-software

Formal Methods are required in some contexts

Jobs, Jobs, Jobs!

ClearSy (Ingénieurs méthodes formelles, Logiciel, SdF, ...),
Systerel(Ingénieurs méthodes formelles, Logiciel, SdF, ...),
Siemens, RATP, ALSTOM, Thales...
Hitachi Rail STS France, TechnicAtome, Railenium,...

https://fr.indeed.com/+MF

Formal Methods are required in some contexts

Norme EN 50128 : "Systèmes de signalisation, de télécommunication et de traitement" :
Cette norme traite en particulier des méthodes qu'il est nécessaire 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

Standards in Software construction

EN xxx : Normes européennes / IEC xxx : Normes internationales (avec des correspondances) et aussi des DO xxx
DO178C : *Software Considerations in Airborne Systems and Equipment Certification*

Normes	Domaines
IEC 62304	*Medical Device Software Development* Systèmes embarqués, médical
IEC 62279 (EN 50128)	*Railway applications*
IEC 61506	*Industrial-process measurement and control*
...	...

SIL : Safety Integrity Level ☞ Formal methods

The EN 50128 : Software Aspect of Control Systems

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/contexts, we will **need formal methods** to build dependable systems.

[1] computer-aided software engineering

La logique : construire et raisonner

Logique du premier ordre :

- propositions + Calcul propositionnel (procédure de décision)
- prédicats + Système de preuve

Propriétés :

- exprimées à l'aide de prédicats
  ⚠ différentes catégories de propriétés (différentes logiques).
- utilisées pour construire/analyser des objets

Opérations de base sur les prédicats :

- Substitution $P_X[X = V]$ ou $[V/X]P_X$
- Quantification $\forall x.(P(x) \Rightarrow G(x))$ $\quad$ $\exists y.(P(y) \wedge F(y))$

Logique : Construire et Raisonner sur des objets.

Hoare Logic

Spécification de correction partielle d'un prog. : "triplet de Hoare"
(Floyd/Hoare-Dijkstra - système de déduction / axiomes + règles)

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

S : instruction ou programme (*statement*)
P : Précondition et R : Postcondition ; ce sont des prédicats.
Correction partielle : car hypothèse de terminaison.
Règles de Hoare (ax. affectation, séquence, conditionnelle, boucle).
Raisonnement sur la structure du programme.

⚠ Terminaison (à prouver, si on veut la correction totale !)

Trouver ou **calculer les préconditions** et les postconditions.
Des conditions plus faibles (*weakening*), plus fortes (*strengthening*)?

Weakest preconditions (Les plus faîbles préconditions)

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 such that the execution of S begining with one of them **terminates** in a *finite time* in a state satisfaying R.

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

Some examples

Let S be an assignment (i:=i+1) and R the predicate $i \leq 1$

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

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

$$wp(S, R) \;=\; \textbf{Vrai}$$

The wp is a **predicate**; it is computed!

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 - Predicate Transformer

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 given S,
$wp(S,\ R)$ viewed as a function with only **one parameter** $wp_S(R)$.
This function wp_S is called *predicate transformer* - DIJKSTRA

It is the function which associates to every predicate R the weakest precondition such that $\{wp_S(R)\}\ S\ \{R\}$.

Predicate Transformer - Programming

- Un prédicat définit des états.
- P sur les entrées du programme, R sur les sorties du programme.
- Lorsque $\{P\}$ **S** $\{R\}$ vrai, **S** transforme l'état décrit par P en l'état décrit par R.

Un programme S se comporte comme un transformateur de prédicat : il transforme un état en un autre état.

☞ Sémantique (des instructions) des programmes ! en Logique !

Introduction: proving the correction of a software

Build correctly a software or
Prove the correction of a software S via its model.

- The **model** of the software : M
- The **properties** : P
- $M \models P$
  proof depending on the structure of the model
  ex: prove that P is true in all the (reachable) states of M
  (if M is a state model)

Anyway, you need a formal model; or rigorous dev. methods.
Do you know some?

☞ Learn how to build M, P and how to prove (Modelling + Verification) (using dedicated tools or not).

Deductive approaches (logic-based) - an overview

- Build a logic model (formal specification) of a system M
  (or software, product, system...)
- State the required properties for the system P.
- Prove that the model satisfies the properties $M \models P$.
  ?= demonstrate a theorem (from which axioms?)

The proved properties become theorems!

Need the **assistance of provers** (interactive theorem proving)
But, we may want to build [**correct**] programs/software!

Heavy trends: Correct-by-construction (model, prove, refine until code)

Foundation of formal approaches (proof)

Interpretation of the Curry-Howard's Isomorphism:

$$\text{Proof}\ \frac{\textit{Axioms}}{\textit{Theorems}} \quad \text{equiv. to} \quad \text{Development}\ \frac{\textit{Specifications}}{\textit{Programs}}$$

♡♡♡

→ Proof assistants needed! not only editors and compilers

Models and what they are good for?

What is a model?

- An approximation of a reality. It can be built mathematically or not.
- Useful (meaningful) or not.

Given a model M, it can be a reference:

- to build a product/software;
- to reason about a system (a software): predict its behavior: $M \models P$

But

- Can we use it as the program?
- can we transform it into the program?

☺ ☺ ☺

Models: a few definitions

Modelling:
HOARE: A **scientific theory is** formalised as **a mathematical model of reality**, from which can be deduced or calculated the observable properties of a well-defined class of processes in the physical world.

There are two main notions of models in computer science.

1. Model = an approximation of the reality by a math. structure.
   An object O is a model of a reality R, if O allows one to answer all the questions about R.

In Mathematics, Physics, ... models are built with equation systems using quantities (masses, energy, ...) or hypothetic laws.
⇒ State exploration, simulation

Models: a few definitions (continued)

2. Logics, Theory of models
   A model of a theory T is a structure in which the axioms of T are valid.
   A *structure* S is a model of a theory T, or S satisfies T if all formula of T is satisfied in S.
   The reality is a model of a theory!
   First Order Theory = any set of logic formula in a given language (precisely defined).
   Model as an interpretation of a specification - an algebra as a model of an algebraic specification (or an axiomatisation).

⇒ deductive approach, theorem proving

Models: a few definitions (continued ... end)

These two notions of *model* are encountered in the model-oriented (or state-oriented) and property-oriented approaches of Soft. Eng.

In current practical/pragmatic uses,

- *model* = (archetype), what serves or is used for imitation to reproduce orther instances.
- *model* = (paradigm), declination model, conjugation model, etc
- *model* = (reference), . . .

Introduction to the B Method

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*

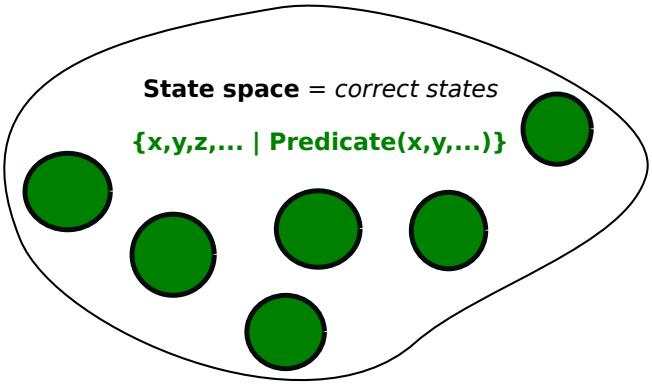


Figure: A view of a state space

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

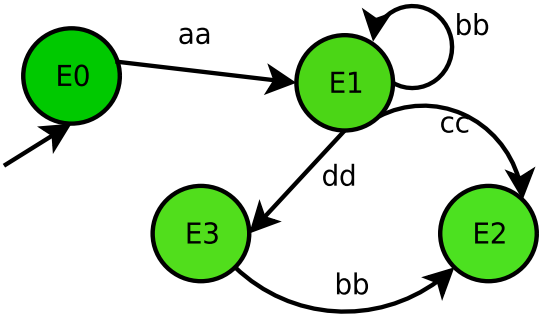


Figure: Evolution of a software system

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

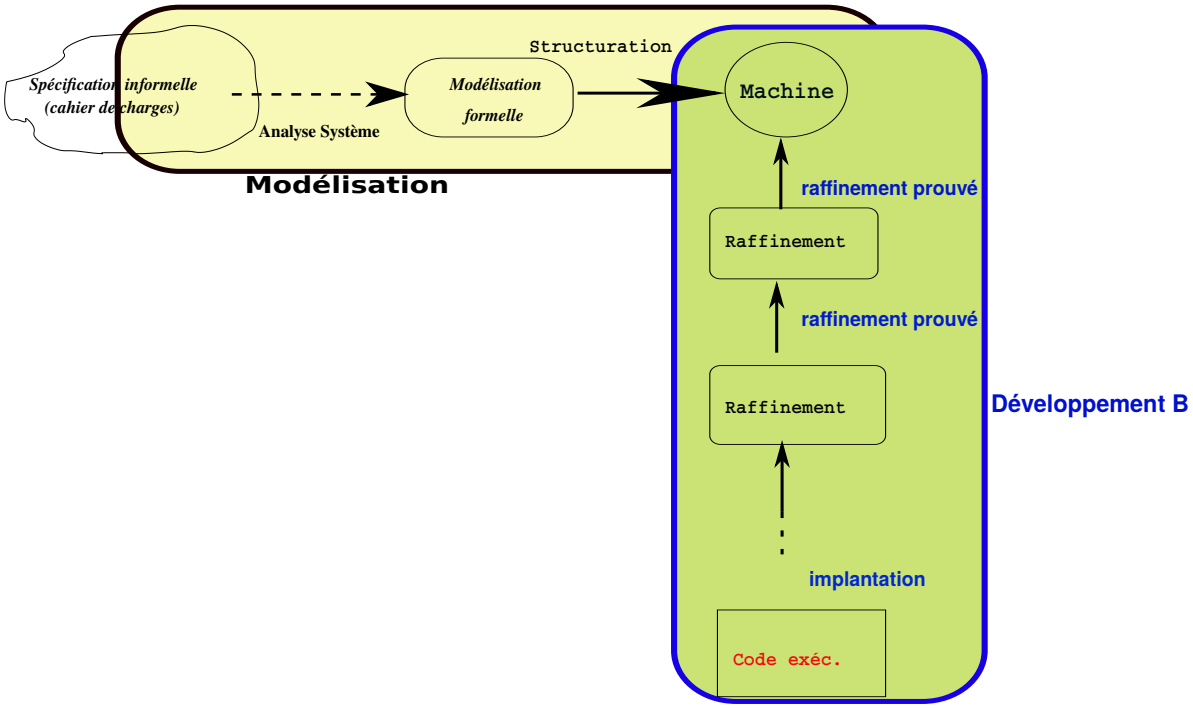


Figure: Analysis and B development

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.

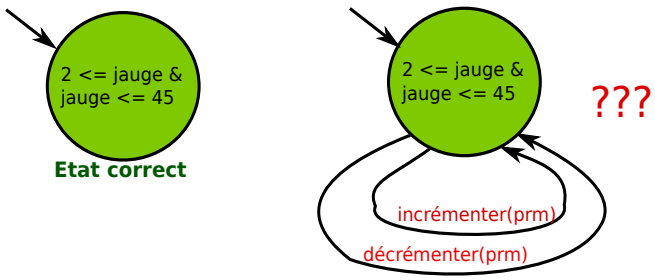


Figure: Rester dans un état correct

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

Semantics - Consistency

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.

Modelling Data

Data Modelling Language

- First order Logic

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

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

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

- Set operators + theory :

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

- Relations, Functions :

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

Nous étudierons en TD/TP

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

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) ;
- ...

Refinement

Refinement: development technique

The objective is the construction of correct executable code.

- Start with an abstract machine : an abstract mathematical model,
- Refine this model to obtain : a concrete model
  - the abstract model is not executable.
    Why? (it is defined with mathematical objects)
  - to an equivalent model, wrt to functionalities, but more concrete.
    (it is described with programming objects)
- We should guaranty that the refinement is correct:
  ⇒ refinement proof obligations

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

Refinement: development technique

Principle of refinement (S_c refines S_a)
Abstract machine : INV_{abs}, U_{abs}, OP_{abs} =(PRE_{abs} | Subst_{abs})
Refining machine : INV_{conc}, U_{conc}, OP_{conc} =(PRE_{conc} | Subst_{conc})

Refinement of : the invariant, the initialisation, the operations.
Each abstract operation is refined by a concrete operation.

$$\textit{if} \quad Subst_{abs} \sqsubseteq Subst_{conc} \quad \textit{then} \quad \forall R.([R]Subst_{abs} \Rightarrow [R]Subst_{conc})$$

The refinement preserves the invariant.

Approach of refinement

What to refine in the model?

- The **variables and the invariant** (refining the state space)
- The **operations** (refinement of the generalized substitutions). Introduce **refinement substitutions** *(until programming substs)*.
- Use the clause REFINES to link the abstract machine with its refinement

```
REFINEMENT  MM_R1
REFINES     MM
...
INVARIANT
+binding abstract and concrete
variables
...
END
```


Approach of refinement: How to refine?

Introduce data structures and replace abstract by concrete ones.

- **Think a lot about a strategy**
- Refining the state space:
  - introduce new (concrete) variables,
  - choice of (less abstract) structures,
  - link abstract and concrete variables by a binding invariant
- Refinement of the operations:
  - The operation interfaces should not be modified.
  - Introduce refinement substitutions to rewrite abstract operations with the new variables.
  - Remove non-determinism
  - Weak the preconditions of the abstract oper, until they disappear.

⇒ Need of the extension of the substitution language.

Refinement substitutions

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

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

  S establishes $[T]R$

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

  var x in // introduction of local variables
  S
  end

Refinement substitutions

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

  while P do S
  invariant I
  variant V
  end

Semantically, it is

$$\begin{array}{l} I \wedge \\ \qquad \text{/* the variant is a natural */} \\ \forall\, x.(I \Rightarrow V \in NATURAL) \wedge \\ \qquad \text{/* the variant decreases after each step */} \\ \forall\, (x,n).(I \wedge P \Rightarrow [n := V][S](V < n)) \wedge \\ \qquad \text{/* continuation of the loop */} \\ \forall\, x.(I \wedge P \Rightarrow [S]I) \mid \\ \qquad @x'.([x := x'](I \wedge \neg P) \Rightarrow x := x') \end{array}$$

Refinement proof obligations

Intuition: the concrete machine does not contradict the abstract one.
PO for the initialisation:

$$[U_{conc}]\neg([U_{abs}]\neg(Inv_{conc}))$$

PO for each operation:

$$INV_{abs} \wedge INV_{conc} \wedge PRE_{abs} \Rightarrow PRE_{conc}$$

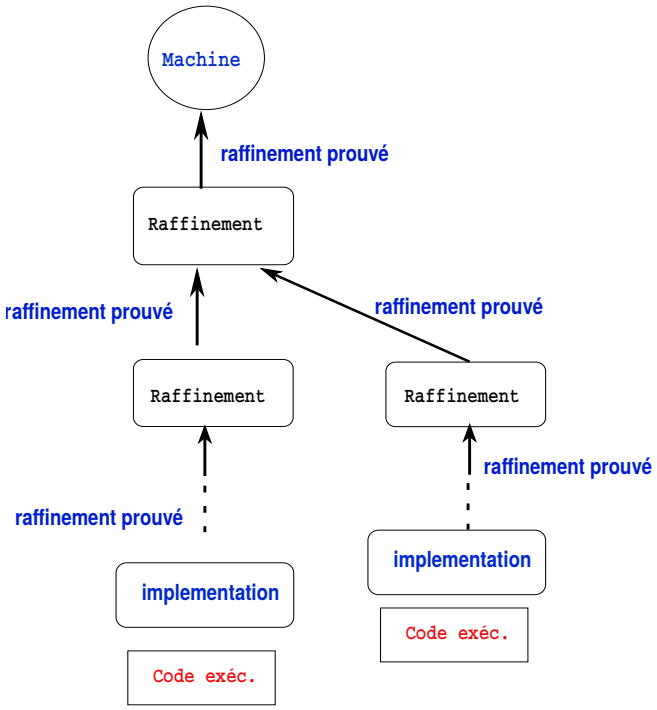
$$INV_{abs} \wedge INV_{conc} \wedge PRE_{abs} \Rightarrow [Subst_{conc}]\neg([Subst_{abs}]\neg(INV_{conc}))$$

Refinement of operations: practically

Abstract	Refinement
S ‖ T	S ; T
PRE P THEN Subst END	BEGIN Concrete(Subst) END
ANY ... WHERE ... END	WHILE ... DO ... END
CHOICE S OR T END	IF (...) THEN S ELSE T
...	...

Implementation

- Stop the refinement when no more abstract structures
- Implementation = the last refinement step
- Several alternative refinements
- B0 Language
- Translation (at least) into C code.



Exercices

En TD/TP, mais pas seulement !

Synthèse

Une introduction à la Méthode B

- Modélisation formelle : approche par états définis par Invariant ; machine abstraite/concrète
- Approche de la correction par construction (à priori)
- Très forte expressivité : Logique, Théorie des ensembles
- Raffinement au code du logiciel (de l'abstrait au concret)
- Preuves de cohérence et des raffinements
- Méthode outillée

Mais on fait encore mieux avec Event-B !