

# Modèles Événementiels

Spécification et Validation de Programmes  
(SVP) - M2 STL 2015

# Approches méthodologiques

Deux approches complémentaires de modélisation formelle :

## 1. Approche ascendante

- Programmation : programme + tests
- Spécification : propriétés du programme
- Validation : preuve de propriété

⇒ **Coq** “classique” : fonctions récursives → preuves inductives

## 2. Approche descendante

- Spécification : modèle formel d'un système
- Programmation : implantation de la spécification
- Validation : preuve que l'implantation respecte la spec.

⇒ **méthodes formelles** : Z, VDM, B, **Event B**, ASM, etc.

# Event B

Source : [event-b.org](http://event-b.org)

*“Event-B is a formal method for system-level modelling and analysis. Key features of Event-B are the use of set theory as a modelling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels.”*

Environnements :

- **Atelier B** (plutôt B “classique”)  
⇒ Clearsy (commercial, gratuit) : [www.atelierb.eu](http://www.atelierb.eu)
- **Rodin platform** (Event B)  
⇒ Consortium Deploy (logiciel libre) : [event-b.org](http://event-b.org)

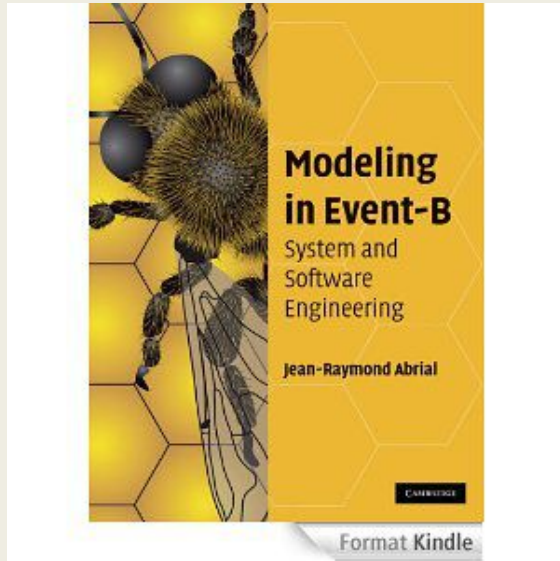
# B dans l'industrie

Quelques usages industriels :

- B “classique”
  - RATP (Meteor, etc.), Alstom, Siemens, etc.  
cf. [www.atelierb.eu/en/industrial-references-for-atelier-b/](http://www.atelierb.eu/en/industrial-references-for-atelier-b/)
- Event B
  - Bosch, Siemens, SAP, etc.  
cf. [wiki.event-b.org/index.php/Industrial\\_Projects](http://wiki.event-b.org/index.php/Industrial_Projects)

# Littérature

## Event B



### Modeling in Event - B

Jean-Raymond Abrial

© 2010 Cambridge University Press

## B classique



### The B-Book

Jean-Raymond Abrial

© 2005 Cambridge University Press

# Dans ce cours ...

## Event B “light”

- Machines indépendantes (pas de composition)
- Théory des types vs. Ensembles naïfs
- Raffinement “manuel” (cf. prochains cours)

## Environnement

- Encodage des machines en Coq
- Obligations de preuves (PO) “manuelles”
- Démonstration assistée des PO en Coq

# Event B “light” : concepts

**Event B** est un langage de spécification

**Objectif** : à partir d'un cahier des charges informel, spécifier de manière formelle un système (logiciel/matériel/mixte) et son environnement.

⇒ *Closed system modeling*

Une spécification contient :

- une partie statique : le **contexte**
- une partie dynamique : la **machine** (abstraite/concrète)

# Spécification Coq

**Module** *<Nom de la spécification>*.

**Module** Context.

*<spécification du contexte>*

**End** Context.

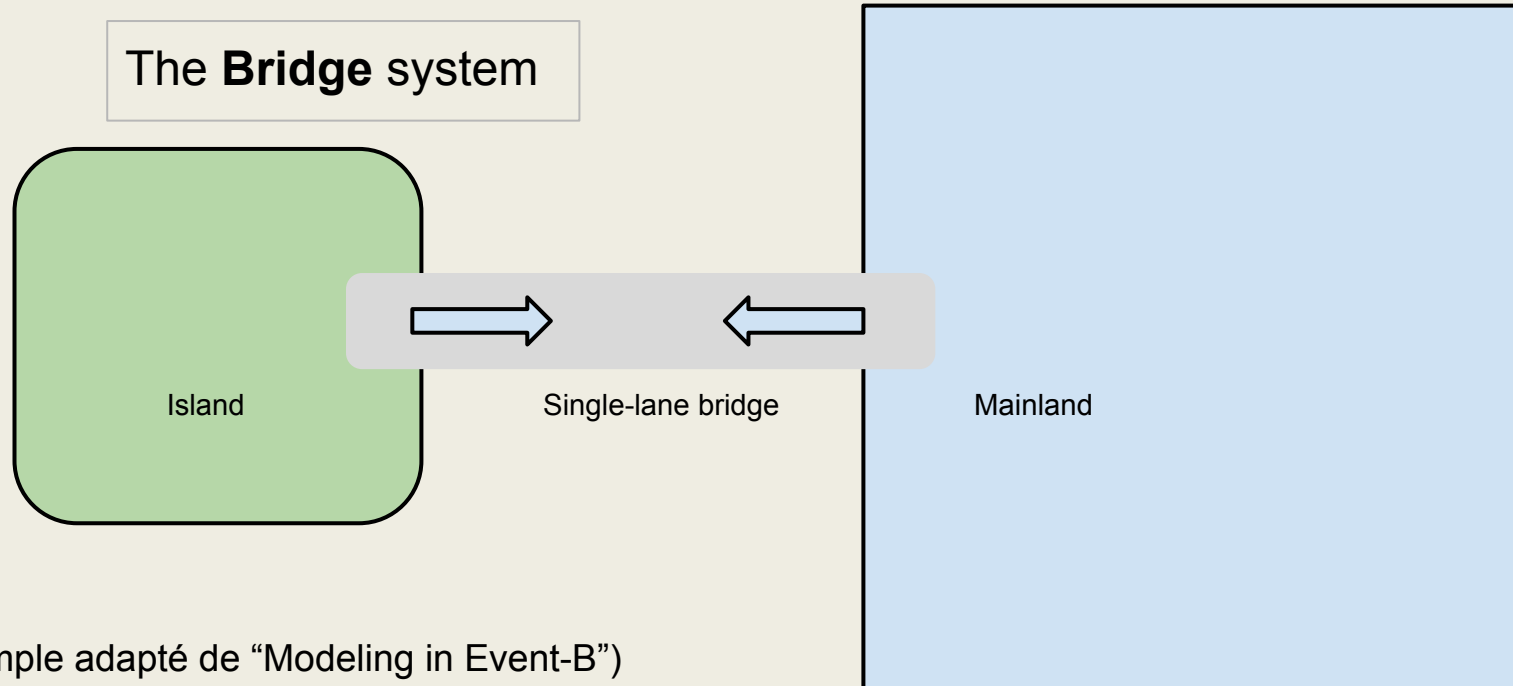
*<spécification de la machine>*

**End** *<Nom de la spécification>*.



# Étude de cas

Pour illustrer les concepts : on considère un système très simple.



(exemple adapté de “Modeling in Event-B”)

# Le contexte

**Contexte** = partie statique du système

- **Constantes**
  - paramètre(s) du système
- **Axiomes**
  - propriétés statiques sur les constantes

# Contexte en Coq

```
Module Context.
```

```
(* constante *)
```

```
Parameter <nom de la constante> : <type>.
```

```
...
```

```
(* axiome *)
```

```
Axiom <nom de l'axiome> : <proposition>.
```

```
...
```

```
End Context.
```

# Structure d'une machine

⇒ La **machine** (abstraite/concrète) exprime la dynamique du système.

Une machine B est spécifiée par :

- un état
  - **variables** d'état
- des invariants d'état
- des événements comprenant :
  - une **garde** : condition d'application de l'événement
  - une **action** : description d'un changement d'état
- des preuves de propriétés
  - guidées par des **obligations de preuves (PO)**

# État d'une machine en Coq

```
Record State : Set := mkState {  
  
  (* variable *)  
  <nom de la variable> : <type>.  
  
  ...  
  
}
```

# Invariants (d'état)

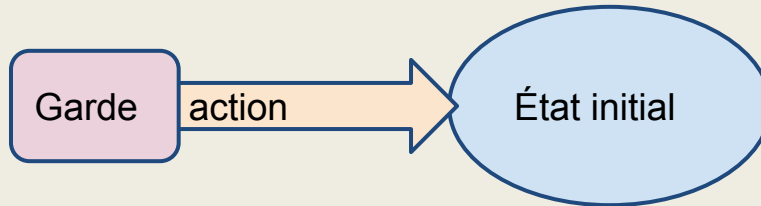
Un invariant (d'état) exprime une propriété importante devant être toujours vérifiée par l'état de la machine.

En Coq :

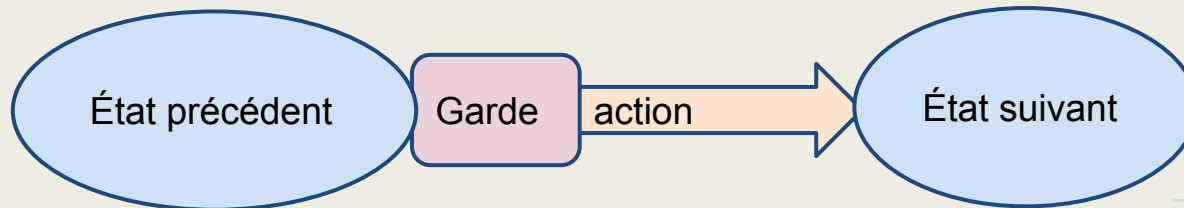
**Definition** `Inv_<id numérique> (<état>:State) : Prop :=`  
`<proposition invariante sur l'état>.`

# Évènements

- Construction d'un état initial de la machine  
⇒ **événement d'initialisation**



- Changement d'état  
⇒ **événement de transition**



# Initialisations en Coq

**Évènement d'initialisation** : description d'un état initial

```
Module <Nom de l'évènement>. (* généralement : Init *)
```

```
Definition Guard <paramètres> : Prop :=  
  <proposition de garde>.
```

```
Definition action <paramètres> : State :=  
  <définition de l'action : construction de l'état initial>.
```

```
<obligations de preuves (PO)>
```

```
End <nom de l'évènement>.
```

**Remarques** : ① l'action d'initialisation s'applique sous l'hypothèse de la garde.  
② la garde et l'action ont les mêmes paramètres



# Transitions en Coq

**Évènement de transition** : description d'un changement d'état atomique :

**Module** *<Nom de l'évènement>*.

**Definition** Guard (*<état précédent>* : State) *<autres paramètres>* : **Prop** :=  
*<proposition de garde>*.

**Definition** action (*<état précédent>* : State) *<autres paramètres>* : State :=  
*<définition de l'action : construction de l'état suivant>*.

*<obligations de preuves (PO)>*

**End** *<nom de l'évènement>*.

**Remarques** : ① l'action de transition s'applique sous l'hypothèse de la garde.  
② la garde et l'action ont les mêmes paramètres

# Obligations de preuves (PO)

La spécification d'un événement induit des obligations de preuves.

- Sûreté : **Safety PO** (obligatoire, initialisation ou transition)
  - un état construit est correct  $\Rightarrow$  les invariants sont vérifiés.
- Fairness : **Convergence PO** (optionnelle, transition)
  - un événement convergent ne peut être appliqué indéfiniment.
- Vivacité : **Deadlock Freedom PO** (optionnelle, globale)
  - pour tout état au moins un événement est applicable.

**Remarque** : les constantes et axiomes du contexte sont des hypothèses implicites des POs.

# Safety PO : initialisation

**Definition** **Guard** (p1:T1) ... (pN:TN) : **Prop** := <prop>.

**Definition** **action** (p1:T1) ... (pN:TN) : **State** := <expr>.

**Theorem** PO\_Safety:

(\* paramètres \*)

**forall** p1 : T1, ..., **forall** pN : TN,

(\* Garde \*)

**Guard** p1 ... pN

-> (\* état initial \*)

**let** S := **action** p1 ... pN

**in** (\* Invariants \*)

Inv\_1 S /\ ... /\ Inv\_M S.

**Proof.**

# Safety PO : transition

**Definition** `Guard` (S:State) (p1:T1) ... (pN:TN) : **Prop** := <prop>.

**Definition** `action` (S:State) (p1:T1) ... (pN:TN) : State := <expr>.

**Theorem** `PO_Safety`:

```
(* état précédent *)
forall S : State,
(* paramètres *)
forall p1 : T1, ..., forall pN : TN,
(* Invariants pre *)
Inv_1 S -> ... -> Inv_M S
-> (* Garde *)
  Guard S p1 ... pN
  -> (* état suivant *)
    let S' := action S p1 ... pN
    in (* Invariants post *)
      Inv_1 S' /\ ... /\ Inv_M S'.
```

**Proof.**

# Convergence PO (Optionnelle)

Convergence d'un événement de transition :

- l'événement ne peut s'appliquer indéfiniment dans un état donné

Preuve de convergence :

- Définition d'un **variant** d'état pour la transition
  - Mesure définie sur un ordre strict bien fondé  
⇒ exemple : (nat, <)
  - Synthétisé à partir d'un état de la machine
- Preuve de **décroissance stricte du variant** durant la transition  
⇒ **variant** <état suivant> < variant <état précédent>

# Convergence PO en Coq

**Definition** Guard (S:State) (p1:T1) ... (pN:TN) : Prop := <prop>.

**Definition** action (S:State) (p1:T1) ... (pN:TN) : State := <expr>.

**Definition** variant (S:State) : nat := <expr>.

**Theorem** PO\_Convergence:

(\* état précédent \*)

**forall** S : State,

(\* paramètres \*)

**forall** p1 : T1, ..., **forall** pN : TN,

(\* Invariants pre \*)

Inv\_1 S -> ... -> Inv\_M S

-> (\* Garde \*)

Guard S p1 ... pN

-> (\* état suivant \*)

**let** S' := action S p1 ... pN

**in** (\* Invariants post \*)

variant S' < variant S.

**Proof.**

# Deadlock Freedom

PO optionnelle de la machine :

⇒ dans tout état au moins une transition est possible.

Preuve de Deadlock Freedom:

- Soit un état  $S$  quelconque,
  - En supposant les invariants d'état de la machine (et implicitement les axiomes)

⇒ montrer que la disjonction des gardes de tous les événements est vraie.

# Deadlock Freedom en Coq

**Definition** `Guard` (p1:T1) ... (pN:TN) : **Prop** := <prop>.

**Definition** `action` (p1:T1) ... (pN:TN) : **State** := <expr>.

**Theorem** `PO_Deadlock_Freedom`:

(\* état \*)

**forall** S : **State**,

(\* invariants \*)

Inv\_1 S -> ... -> Inv\_M S,

(\* Disjonction des gards \*)

-> Event1.**Guard** S \/ ... \/ EventN.**Guard** S

**Proof.**

**Remarque** : en général la preuve est une décomposition par cas non-triviale.



# ... to be continued

Prochain cours :

Approche descendante itérative

⇒ **Raffinement** de machine

