Introduction to the **Event-B** method and the **Rodin** development tool

Pierre Castéran
Bordeaux, HCMC
A critical system or safety-critical system is a system whose failure or malfunction may result in:

- death or serious injury to people, or
- loss or severe damage to equipment or
- environmental harm,
- etc.
A critical system or safety-critical system is a system whose failure or malfunction may result in:

- death or serious injury to people, or
- loss or severe damage to equipment or
- environmental harm,
- etc.

Safety-critical systems are increasingly computer-based.
Examples (c) Wikipedia

**infrastructure**: Emergency services dispatch systems, Fire alarm, Electricity generation, transmission and distribution

**medicine**: Heart-lung machines, Robotic surgery machines

**Nuclear engineering**: Nuclear reactor control systems

**Transport**: Railway signalling and control systems, Braking systems, avionics, Human spaceflight vehicles
Safety critical software need to be trusted
Tests and model-checkers don’t cover all software.
Safety critical software need to be trusted
Tests and model-checkers don’t cover all software.

Solutions?
Interactive Program Proving: Frama-C, Jessie
Program Synthesis by stepwise refinements: Atelier B, Rodin
Introduction to the B Method

Let’s look at Wikipedia again

The B method is a method of software development based on B, a tool-supported formal method based around an abstract machine notation, used in the development of computer software. It was originally developed by Jean-Raymond Abrial in France and the UK.
Introduction to the B Method

Let’s look at Wikipedia again

The B method is a method of software development based on B, a tool-supported formal method based around an abstract machine notation, used in the development of computer software. It was originally developed by Jean-Raymond Abrial in France and the UK.

B has been used in major safety-critical system applications in Europe (such as the Paris Métro Line 14), and is attracting increasing interest in industry. It has robust, commercially available tool support for specification, design, proof and code generation.
Principles of the B Method

This method proposes the following cycle of project development:

1. Translation from an informal or semi-formal specification into the Abstract Machine Notation.
Principles of the B Method

This method proposes the following cycle of project development:

1. Translation from an informal or semi-formal specification into the Abstract Machine Notation. This step is not machine-checked, so it’s extremely important that this translation can be read and accepted by the “client”.
Principles of the B Method

This method proposes the following cycle of project development:

1. Translation from an informal or semi-formal specification into the Abstract Machine Notation. This step is not machine-checked, so it’s extremely important that this translation can be read and accepted by the “client”.

2. A sequence of (machine-checked) refinements: each version of the software is proved to be consistent with the previous one.
Principles of the B Method

This method proposes the following cycle of project development:

1. Translation from an informal or semi-formal specification into the Abstract Machine Notation. This step is not machine-checked, so it’s extremely important that this translation can be read and accepted by the “client”.

2. A sequence of (machine-checked) refinements: each version of the software is proved to be consistent with the previous one.

Some remarks

- The weak link of this method is obviously the translation from an informal or semi-formal requirement into a formal statement. It is important that the client can read and control this translation.

- The B-Method proposes to use the basic mathematical language (first-order logic, elementary set theory) for writing formal specifications.

- The first and most abstract step of the development is supposed to be readable. It should contain no implementation details, which should be introduced in further development steps.
Faire un dessin
Atelier B or Rodin?

Two development tools actually use the B method: Atelier B and Rodin.

- Atelier B is suitable for developing imperative programs using classical control structures: loops, sequences, conditionals, etc.

- Rodin implements the event-B formalism for describing event-driven reactive systems: the basic control structure is the event: Any event that satisfies some given condition: its guard can occur. Such systems are not bound to terminate, but their state is required to be consistent.
On-line documentation

- http://www.event-b.org/: wiki, downloads
- Some home-made developments
  http://www.labri.fr/perso/casteran/FM/Rodin
Let’s take some examples

▶ “Controling cars on a bridge” (J.R. Abrial)
▶ Searching an item in an array
Event-B components: contexts

- A context is a first-order theory that contains:
  - declarations of constants,
  - axioms about these constants
- The description language is first order logic + arithmetics + simple set theory.
- Typing is expressed through set membership.
An Event-B Specification of Maximum
Creation Date: 27 Nov 2011 @ 02:45:32 PM

CONTEXT Maximum

CONSTANTS

maxi Maximum number of cars in the island (bridge included)

AXIOMS

axm1: maxi ∈ N₁

END
Comments

- The previous slide has been generated from a context:
  - written with Rodin’s interactive (Eclipse-based) editor,
  - then translated into LaTeX
- It contains a constant declaration of maxi
- The “type” of this constant is expressed through an axiom (with the language of set theory).
- The symbol $\mathbb{N}_1$ denotes the set of strictly positive integers.
Beware of Axioms!

```
CONTEXT  AxiomsBad
CONSTANTS
    f

AXIOMS

axm1 : f ∈ ℕ → ℕ
axm2 : ∀ n · (n ∈ ℕ ⇒ f(n) < n)

END
```

From these declarations, one can infer contradictions, hence 2 = 3.
The following systems of axioms are OK:

\begin{align*}
\text{axm1} & : \quad f \in \mathbb{N} \rightarrow \mathbb{N} \\
\text{axm2} & : \quad \forall n \cdot (n \in \text{dom}(f) \Rightarrow f(n) < n) \\
\text{axm3} & : \quad \forall x, y \cdot ((x \mapsto y) \in f \Rightarrow y < x)
\end{align*}
Advice for building contexts

- Give axioms systems that are consistent (that have some model)
- Try to get minimal sets of axioms: if some property can be inferred from the other axioms, mark it as theorem.
Advice for building contexts

- Give axioms systems that are consistent (that have some model)
- Try to get minimal sets of axioms: if some property can be inferred from the other axioms, mark it as theorem.

\[
\text{axm1: } n \in \mathbb{N}_1
\]
\[
\text{thm1: } \forall i \cdot i \in 0..n - 1 \Rightarrow i > 0 \lor i < n
\]
A context for searching in an array

CONTEXT Array

CONSTANTS

\( n \)  array size
\( a \)  the array to search in
\( x \)  value to search in \( a \)

AXIOMS

axm1 :  \( n \in \mathbb{N}_1 \)
axm2 :  \( a \in 1..n \rightarrow \mathbb{Z} \)
axm3 :  \( x \in \mathbb{Z} \)

END
CONTEXT  SortedArray
EXTENDS  Array
AXIOMS

axm1 : \( \forall i,j. i \in 1 .. n \land j \in i .. n \Rightarrow a(i) \leq a(j) \)
END
Event-B’s description language is quite big, including:

- First Order Logic: connective, quantifiers,
- Naïve set theory: sets, relations, functions,
- Arithmetics (on $\mathbb{Z}$). $\mathbb{N}$ and $\mathbb{N}_1$ are subsets of $\mathbb{Z}$. 

Note that sets, relations, functions, are first-class objects of Event-B language. It is thus possible to quantify over them.
Event-B’s description language is quite big, including:

- First Order Logic: connective, quantifiers,
- Naïve set theory: sets, relations, functions,
- Arithmetics (on $\mathbb{Z}$). $\mathbb{N}$ and $\mathbb{N}_1$ are subsets of $\mathbb{Z}$.

Note that sets, relations, functions, are first-class objects of Event-B language. It is thus possible to quantify over them.

SETS

$U$

AXIOMS

$thm1 : \forall A, B, C \cdot A \subseteq C \land B \subseteq C \land C \subseteq U \Rightarrow A \cup B \subseteq C$
Abstract Machines

An abstract machine is a component of an Event-B project, which describes a reactive system.

Structure of an abstract machine

- Constants and axioms are imported from contexts (SEES clauses).
- The state of the machine is described by a set of variables,
- the consistency of the state is defined by a set of invariants: i.e. formulae that the variables must satisfy.
- A set of events describe the possible evolutions of the machine’s state.
An example

An Event-B Specification of Br0
Creation Date: 27 Nov 2011 @ 02:45:39 PM

MACHINE Br0
SEES Maximum

VARIABLES

  nb_cars total number of cars (bridge + island)

ININVARIANTS

  inv1 : nb_cars ∈ 0 .. maxi
Any abstract machine must define a special event called \texttt{initialisation} for giving an initial value to the variables of the machine. It takes the form of a set of (parallel) assignments.

\begin{verbatim}
EVENTS
Initialisation

act1 : nb_cars := 0
\end{verbatim}
Any abstract machine must define a special event called \textit{initialisation} for giving an initial value to the variables of the machine. It takes the form of a set of (parallel) assignments.

\begin{verbatim}
EVENTS
Initialisation

act1 : nb_cars := 0
\end{verbatim}

\textbf{Proof Obligation :}

The initial values of the variables must satisfy the invariants of the machine.

For each invariant $inv_i$, Rodin generates a \textit{proof obligation} \text{INITIALISATION/inv$_i$/INV} whose hypotheses are all the axioms and theorems of the seen contexts, and conclusion (goal) is $inv_i$ where every variable has been replaced by its initial value.

$$0 \in 0 .. maxi$$
Ordinary events

Events (other than initialisation) describe the possible changes of the machine’s state. They are composed of:

- A *guard*, that defines whether the event can be triggered,
- An *action* part, that reassigns (part of) the variables of the machine.
Event $Main_{out} \triangleq$
A car leaves the mainland
when
$\quad grd1: nb\_cars < maxi$
then
$\quad act1: nb\_cars := nb\_cars + 1$
end

Event $Main_{in} \triangleq$
when
$\quad grd1: nb\_cars > 0$
then
$\quad act1: nb\_cars := nb\_cars - 1$
end
Proof obligations associated with an event

- If the invariants are true before the event,
- and if the event’s guard is true,
- then the invariant must be true after the event.
Proof Obligations associated with an event

Let us consider some event \( e \) : for each invariant \( inv_i \) of the machine, a proof obligation \( e/inv_i/INV \) is built.

The hypotheses are formed by:

- The axioms and theorems of the imported contexts
- All the invariants,
- The guard of the considered event
- The before-after relations associated with the assignments (expressed as a relation between old variables \( v \) and new variables \( v' \))

The conclusion is \( inv_i' \) which is a copy of \( inv_i \) after replacing all assigned variable names \( v \) by \( v' \).
Example: the Main\_out event

Hypotheses

- \texttt{axm1} \( \maxi \in \mathbb{N}_1 \)
- \texttt{inv1} \( \textit{nb\_cars} \in 0..\maxi \)
- \texttt{grd1} \( \textit{nb\_cars} < \maxi \)

Before-after relation \( \textit{nb\_cars}' = \textit{nb\_cars} + 1 \)

Goal: \texttt{inv1'}

- \texttt{inv1'} \( \textit{nb\_cars}' \in 0..\maxi \)
A commented abstract machine: Searching in an array

MACHINE S0
SEES Array
VARIABLES
  result
ININVARIANT
  inv3: result ∈ 0 .. n
EVENTS
  begin
    act1: result := 0
  end

An Event-B Specification of S0
Creation Date: 2 Jan 2012 @ 03:01:14 PM
Events with parameters

Some events may use local parameters, introduced with the ANY...WHERE construct:

```
EVENT SUCCESS
    any
    r
    where
    grd1: r ∈ 1 .. n
    grd2: a(r) = x
    then
    act1: result := r
    end
```
The proof obligation associated to an invariant and a parameterized event just considers the parameter as a free variable.

Hypotheses

- **axm1** \( n \in \mathbb{N}_1 \)
- **axm2** \( a \in 1..n \rightarrow \mathbb{Z} \)
- **axm3** \( x \in \mathbb{Z} \)
- **inv3** \( \text{result} \in 0..n \)
- **grd1** \( r \in 1..n \)
- **grd2** \( a(r) = x \)
  \[ \text{result}' = r \]

Goal

- **inv3'** \( \text{result}' \in 0..n \)
The following event describes the case where \( x \) has no occurrence in the array \( a \). Notice that it is just an abstract but easy to read specification.

**EVENT FAILURE :**

\[
\text{when } \quad \text{grd1} : \quad \forall i \cdot i \in 1..n \Rightarrow a(i) \neq x
\]

\[
\text{then } \quad \text{skip}
\]

\[
\text{end}
\]
Introduction to the notion of refinement

- We will say that a machine $C$ refines a machine $A$ when all behaviours of $C$ correspond to behaviours of $A$.
- We will say that $C$ is more concrete than $A$. 
Rodin
On Refinements

Introduction to the notion of refinement

- We will say that a machine $C$ refines a machine $A$ when all behaviours of $C$ correspond to behaviours of $A$.
- We will say that $C$ is more concrete than $A$.
- Rodin helps us to build *machine-proven* refinements,
- Refinements are useful for deriving more concrete implementations from abstract specifications.
- We can also use refinements for expressing more precise specifications.
A first example

MACHINE L0
REFINES S0
SEES Array
VARIABLES
result
EVENTS
Initialisation
extended
begin
act1: result := 0
end
Event \( SUCCESS \) \( \triangleq \) extends \( SUCCESS \)

any

where

\( \text{grd1} : r \in 1..n \)

\( \text{grd2} : a(r) = x \)

\( \text{grd3} : \forall i \cdot i \in 1..n \land a(i) = x \Rightarrow r \leq i \)

then

\( \text{act1} : \text{result} := r \)

end

Event \( Failure \) \( \triangleq \) extends \( Failure \)

when

\( \text{grd1} : \forall i \cdot i \in 1..n \Rightarrow a(i) \neq x \)

then

skip

end

END
MACHINE LinearSearch
REFINES L0
SEES Array
VARIABLES
result
cursor
INVARINTANTS

inv1: c ∈ 1 .. n
inv2: ∀i. i ∈ 1 .. n ∧ a(i) = x ⇒ i ∈ c .. n
DLF: a(c) = x ∨ (c = n ∧ a(c) ≠ x) ∨ (c < n ∧ a(c) ≠ x)
EVENTS
Initialisation

extended
begin
  act1 : result := 0
  act2 : c := 1
end

Event SUCCESS \equiv
refines SUCCESS
when
  grd2 : a(c) = x
with
  r : r = c
then
  act1 : result := c
end
Event \( \text{Failure} \) \( \sqsubseteq \) 
refines \( \text{Failure} \) 
when 
grd2 : \( c = n \) 
grd3 : \( a(c) \neq x \) 
then 
\( \text{skip} \) 
end 

Event \( \text{Right} \) \( \sqsubseteq \) 
Status convergent 
when 
grd1 : \( c < n \) 
grd2 : \( a(c) \neq x \) 
then 
act1 : \( c := c + 1 \) 
end 

VARIANT 
\( n - c \) 
END
Proof Obligations

It is extremely important to know how Rodin builds and tries to solve proof obligations associated to abstract machines and their refinements.

- It helps to design correct machines and invariants,
- It allows to detect conception errors,
- In case of non-automatic proofs, it helps to interact with the tool.
- Last but not least, it is the basis of many questions in a written exam.
Let us consider a machine $A$ and a refinement $C$. We assume that $A$ sees a context $\Gamma_A$ and $C$ a context $\Gamma_C$ (which is often an extension of $\Gamma_A$).

We assume that all the POs of $A$ have been solved.

**Remarks**

- $C$ must declare every variable that occurs in events (in guards and/or assignments).
- Some variables occur both in $A$ and $C$.
- If a variable occur both in $A$ and $C$, one assumes that it has always the same value in both machines.
- An event in $A$ can be *refined* by one or several events in $C$.
- Intuitively, an event $evt_C$ in $C$ refines an event $evt_A$ in $A$ if one can associate to any behaviour of $evt_C$ a behaviour of $A$. 
Remark

Note that the initialisation of LinearSearch extends the initialisation of L0.
An example

- The machine LinearSearch refines the machine L0.
- The variable result is common to both machines
- The variable c belongs only to LinearSearch
- The event SUCCESS is parameterized in L0, but not in LinearSearch
- The event Right belongs only to LinearSearch, and corresponds to a non-event in L0.
Proof obligations for LinearSearch

All the invariants of the concrete machine must be satisfied by the initialisation event:

**Hypotheses**
- \( axm1 \): \( n \in \mathbb{N}_1 \)
- \( axm2 \): \( a \in 1..n \rightarrow \mathbb{Z} \)
- \( axm3 \): \( x \in \mathbb{Z} \)
- \( inv3 \): \( result \in 0..n \)
  - \( c = 1 \land result = 0 \)

**Goal**
- \( inv1 \): \( 1 \in 1..n \)
- \( inv2 \): \( \forall i \cdot i \in 1..n \land a(i) = x \Rightarrow i \in 1..n \)
- \( inv3' \): \( result' \in 0..n \)