Some notes about Event-B and Rodin

Résumé


Event-B is a language used to design abstract state machines and their relations, in particular simulation relations.

The specification of a software system, especially embedded systems reacting to events, can be turned into a formal specification using event-B state machines. See tutorials for examples.

1 Syntax

1.1 Contexts

A context is composed of
— a finite list of contexts that are extended by the current context,
— a finite list $x_1, x_2, \ldots$ of constants names,
— a finite list $S_1, S_2, \ldots$ of sets names,
— a list of logical properties $a_1, a_2, \ldots$ called axioms, referring to the constants and sets defined in this context and the inherited contexts, as well as native elements and symbols of the event-B language (integers, booleans, see http://wiki.event-b.org/images/EventB-Summary.pdf for a comprehensive list).

Intuitively, a context is used to define global variables that are constant, and specifies guarantees on these variables. For example, one may define a non-empty array $A$ of integers as follows.

```
CONTEXT
  array

CONSTANTS
  size
  A

AXIOMS
  axm1: size ∈ N not theorem
  axm2: size > 0 not theorem
  axm2: A \notin \phi \rightarrow \mathbb{N} not theorem

END
```
or, equivalently,

```
CONTEXT
    array
CONSTANTS
    ○ size
    ○ A
AXIOMS
    ○ axm1: size ∈ N1 not theorem
    ○ axm2: Aε0..size → Z not theorem
END
```

The axioms shall include some *typing invariants*, giving types to constants.
Intuitively, sets can be thought of as user-defined *types*, like *enum* types.
For example

```
CONTEXT
    colors
SETS
    ○ Colors
CONSTANTS
    ○ red
    ○ blue
AXIOMS
    ○ axm1: Colors = \{red, blue\} not theorem
    ○ axm2: red ≠ blue not theorem
END
```

defined an enum type with two colors. Remark that nothing guarantees that
red ≠ blue, for that we need to add a specific axiom:

```
CONTEXT
    colors
SETS
    ○ Colors
CONSTANTS
    ○ red
    ○ blue
AXIOMS
    ○ axm1: Colors = \{red, blue\} not theorem
    ○ axm2: red ≠ blue not theorem
END
```
or, more concisely,

```
CONTEXT
    colors
SETS
    ○ Colors
CONSTANTS
    ○ red
    ○ blue
AXIOMS
    ○ axm1: partition(Colors,\{red\},\{blue\}) not theorem
END
```
1.2 Abstract state machines

An abstract state machine is composed of
— the name of the state machine,
— a finite list \( CO_1, CO_2, \ldots \) of contexts seen by the machine,
— a finite list \( v_1, v_2, \ldots \) of variables names,
— a finite list \( I_1, I_2, \ldots \) of logical properties called invariants, which refer to
  — the variables of the machine
  — the constants and sets seen in the contexts and the inherited contexts,
  — as well as native elements and symbols of the event-B language
— a finite list \( E_0, E_1, E_2, \ldots \) of events, including the special event \( E_0 = \text{INITIALISATION} \).
— optionally, a variant \( V \).

The invariants shall include some typing invariants, giving types to variables. For example
\[
n \in \mathbb{N}
\]
is an invariant specifying that variable \( n \) takes non-negative integer values.

1.3 Events

Every event is composed of
— the name of the event,
— a finite list \( g_1, g_2, \ldots \) of logical properties called guards,
— a finite list \( a_1, a_2, \ldots \) of actions, which affects (possibly non-deterministically), a values to variables (not necessarily all of them).

Intuitively, guards are pre-conditions which are required for executing the event. For example, the following machine declares one variable \( n \), of integer type, initially set to 0, and two event that may increase or decrease \( n \), depending on its value.
Some affectations may be non-deterministic, for example we can modify the previous machine so that $n$ is initialized at some arbitrary value by replacing $n := 0$ by $n \in \mathbb{N}$. 

```plaintext
MACHINE
    Example
VARIABLES
    ○ $n$
INVARIANTS
    ○ $inv1$: $n \in \mathbb{N}$ not theorem
EVENTS
    ○ $INITIALISATION$: not extended ord
      THEN
      ○ $act1$: $n = 0$
    END
    ○ $INC$: not extended ordinary
      WHERE
      ○ $grd1$: $n \leq 5$ not theorem
      THEN
      ○ $act1$: $n = n + 1$
    END
    ○ $DEC$: not extended ordinary
      WHERE
      ○ $grd1$: $n > 0$ not theorem
      THEN
      ○ $act1$: $n = n - 1$
    END
END
```
Remark that there may be non-deterministic affectations even in C-programs, for example user input obtained by `cin` or from the arguments passed from command line (cf. `int main(int nargs, char ** args)`) is non-deterministic.

2 Semantics

2.1 Executions

The state machine executes in discrete time, leading to a sequence of states. By definition, the state of the machine is the values of its variables. This differs from usual programming languages where the execution keeps track of a program pointer. In Event-B there is no such pointer, every event whose guards are true (enabled event) may execute. If several events are enabled, then one of them is chosen non-deterministically: a single machine can have many different executions.
Formally, if the domain (the set of possible values) of every variable $v_n$ is denoted $D_n$, then a state of the machine is a sequence of variables values in $D_1, D_2, \ldots$. And an execution of the machine is a sequence of states $s_0, s_1, \ldots$

of the machines such that:

— $s_0$ is obtained by the actions of the **INITIALISATION** event,

— for every $n$, there exists an event $E_n$, whose all guards are true in state $s_n$, and such that $s_{n+1}$ is obtained from $s_n$ by executing the actions of $E_n$.

Let us stress that a single machine may have different executions, and that the order of events in the machine is not-significant: if several events are enabled, one of them will be picked up by the machine, independently of the order of the events of the machine. Stated otherwise, swapping two events does not modify the semantics of a machine.

For example, the following executions can occur in the Machine **example**:

$$0 \xrightarrow{\text{INC}} 1 \xrightarrow{\text{INC}} 2 \xrightarrow{\text{INC}} 3 \xrightarrow{\text{INC}} 4 \xrightarrow{\text{INC}} 5 \xrightarrow{\text{DEC}} 6 \xrightarrow{\text{DEC}} 5 \xrightarrow{\text{DEC}} 4 \xrightarrow{\text{DEC}} 3 \xrightarrow{\text{DEC}} 2 \ldots$$

$$0 \xrightarrow{\text{INC}} 1 \xrightarrow{\text{INC}} 2 \xrightarrow{\text{INC}} 3 \xrightarrow{\text{INC}} 4 \xrightarrow{\text{INC}} 5 \xrightarrow{\text{DEC}} 4 \xrightarrow{\text{INC}} 3 \xrightarrow{\text{INC}} 5 \xrightarrow{\text{DEC}} 4 \xrightarrow{\text{DEC}} 4 \ldots$$

however the following sequence of states is not an execution:

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \ldots$$

because the guard of the **INC** event ($n \leq 5$) is not true when $n = 6$.

### 2.2 Correct executions and machines

An execution is correct if at every step of the execution, all the invariants of the machine are true. Otherwise we say that the execution violates the invariant.

A machine is correct if all its executions are correct.

For example, if we augment the example machine with the invariant $n \leq 4$, then it is not correct because both executions above violate the invariant, when $n = 5$.

### 2.3 Testing

The plugin ProB can be used to generate executions of the machine and test on-the-fly whether the invariants are violated on the particular execution which is generated. Non-determinism is resolved by random choices, so what ProB does is exploring a randomly chosen single path of the whole execution tree.

ProB can detect invariant violations along the execution path, for example:
However, not all the bugs can be found using ProB, for example the following machine is not correct:
This is quite a realistic example, inspired by a common programming bug: forgetting to control that the value of a variable does not go beyond the bounds of its type.

However finding this bug with ProB will be hard because the proportion of executions leading to bugs over all possible executions is very small.

### 3 Refinement

A central notion is refinement. This is the key to go from the specification, given by a machine $M_0$ to the implementation, given by a machine $M_n$ through a series of intermediate machines $M_0, M_1, \ldots, M_n$, linked together by a refinement relation.