

System Modelling and Design

Event B Semantics

Revision: 1.0, March 3, 2007

Ken Robinson

May 29, 2008

©Ken Robinson 2005

[mailto::k.robinson@unsw.edu.au](mailto:k.robinson@unsw.edu.au)

Contents

1	What is this lecture about?	2
2	Semantics in Event B	2
2.1	State Change	2
2.1.1	Before-After Predicates	3
2.2	Substitution	3
2.2.1	Other Forms of Substitution	3
2.3	Context Machines	4
2.3.1	Context Machines: Semantics & Proof Obligations	4
2.3.2	Theorems	5
2.4	Machines	5
2.4.1	Machine POs: Invariant and Theorems	5
2.4.2	Initialisation	6
2.5	Events	6
2.5.1	Event: Proof Obligations	6
2.5.2	Event: Maintaining Invariant	6
2.6	Machine Refinements	6
2.6.1	Variables and Invariant	7
2.6.2	Proof Obligations	7
2.7	Refined Events	7
2.7.1	Proof Obligations for Refined Events	8
2.7.2	The Variant and Convergent Events	8
A	One Point Rule	8

1 What is this lecture about?

1. This lecture presents the semantics of Event-B.
2. The various *Proof Obligations* (*Proof Obligations (PO)*) that result from those semantics.
3. An understanding of “what those POs mean”.
4. The roles of POs in verifying a refinement.
5. The classification of POs, which identify what a particular PO is “all about”.

2 Semantics in Event B

- Each construct in B is given a formal semantics.
- Additionally, machines must satisfy a set of constraints.

These rules provide for

- the verification of the consistency of a machine;
- the verification that the behaviour of a refinement machine is *consistent with* the behaviour of the machine it refines.

Note that it is not possible to prove that the behavior of the initial abstract machine is *correct*, that is, conforms with the written requirements.

2.1 State Change

There are three principle constructions —that Event B calls *substitutions*— for changing the state of a machine:

$x := e$ x becomes equal to the value of e

This rule may be used recursively to assign to any number of variables.

$x :| P$ x becomes such that it satisfies the *before-after* predicate P

$x : \in s$ x becomes in the set s

All of the above, except apparently $:\in$, can be extended to *multiple assignment*: $x, y := e_1, e_2$ and $x, y :| P$, and recursively to many variables. The variables must be distinct.

Note: all assignments can be written in the form: $x, y :| P$.

2.1.1 Before-After Predicates

Before-after predicates contain primed and unprimed variables, for example

$$x' = x + 1$$

where the primed variables represent the *after* value of a variable and the unprimed variables the *before* value.

Thus,

$$x : | x' = x + 1$$

and

$$x := x + 1$$

are equivalent.

Similarly we can write

$$x, y : | x' = x + 1 \wedge y' = y + 1$$

or

$$x, y := x + 1, y + 1.$$

2.2 Substitution

We will frequently need to compute, for example in computing POs, the weakest predicate on the state *before* a state given a required predicate on the *after* state.

We can do this by *substituting* into the after state.

We will write

$$[x, y := e_1, e_2]R$$

to denote the *concurrent* substitution of e_1 and e_2 for x and y in R , respectively.

For example,

$$\begin{aligned} & [x, y := y - 1, x + 1]x - y < x + y \\ & = (y - 1) - (x + 1) < (y - 1) + (x + 1) \\ \text{or } & y - x - 2 < y + x \end{aligned}$$

This gives the weakest constraint on the *before* state such that $x, y := y - 1, x + 1$ will give an *after* state satisfying $x - y < x + y$.

2.2.1 Other Forms of Substitution

For each of the 3 change of state substitutions, substitution into a predicate takes the following form:

$$\begin{array}{l}
\textit{Substitution} \qquad \qquad \qquad [\textit{Substitution}]R \\
v := E \qquad \qquad \qquad [v := E]R \\
v :| P \qquad \qquad \forall v' \cdot P \implies [v := v']R \\
v : \in S \qquad \forall v' \cdot v' \in S \implies [v := v']R
\end{array}$$

where:

1. v in general is a list of variables, and E a list of expressions;
2. P is a predicate containing both v and v' , where v' ; represents the value of v *after* the action.

2.3 Context Machines

Context machines are used to define abstract carrier sets (S) and constants (C).

The form of a context machine is:

Sets	S
Constants	C
Axioms	A
Theorems	T_c

Notice that S and C should be augmented with “builtin” sets and constants such as $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ etc and constants from those sets, but we will elide any explicit extension.

2.3.1 Context Machines: Semantics & Proof Obligations

The semantics of the sets and constants are specified in the axioms. The essential proof obligations is one of *feasibility*: show that sets and constants exist that will satisfy the axioms. That is:

$$(\exists S, C \cdot A)$$

The axioms are usually broken up into a sequence of sub-axioms $a_1, a_2, \dots a_n$, which are effectively conjuncted into a single A . The POs can be recursively split into separate POs based on

$$\begin{aligned}
& (\exists S, C \cdot a_1 \wedge a_2 \wedge \dots \wedge a_n) \\
& \equiv \exists S, C \cdot a_1) \wedge \exists S, C \cdot a_1) \implies (\exists S, C \cdot a_2 \wedge \dots \wedge a_n)
\end{aligned}$$

This may require the sub-axioms to be ordered

Of course, components of S, C that are not referenced in a_i can be eliminated from $\exists S, C \cdot a_i$.

2.3.2 Theorems

Theorems describe properties that follow from the axioms, so the general PO for the theorems is

$$(\forall S, C.A \implies T_c)$$

The theorems will, in general, be broken in sub-theorems t_1, t_2, \dots, t_n , and since universal quantification distributes through conjunction this breaks into multiple POs:

$$(\forall S, C.A \implies t_1), \dots, (\forall S, C.A \implies t_n)$$

2.4 Machines

The form of a machine is:

Context	S, C
Variables	V
Invariant	I
Theorems	T_v
Variant	Var
Events	E

2.4.1 Machine POs: Invariant and Theorems

The invariant as for the axioms for context machines, the invariant may raise feasibility proof obligations:

$$(\exists S, C \cdot A) \implies (\exists V \cdot I)$$

The theorems must follow from the set/constant axioms and the invariant:

$$\forall S, C, V \cdot A \wedge I \implies T_v$$

Note: where we have A we could also have $A \wedge T_c$, but since $A \implies C$ this does not gain any extra strengthening.

2.4.2 Initialisation

Initialisation, which is a special part of the events, must establish a state in which the variables satisfy the invariant.

Let us represent the initialisation by a multiple substitution

$$V := E(S, C)$$

where $E(S, C)$ emphasises that the initialising expressions can only reference sets and constants: E must not reference any variables, since all variables at this point are undefined.

Then the proof obligation for initialisation is

$$\forall S, C \cdot A \implies [V := E(S, C)] I$$

2.5 Events

Events have the following form

ANY	x
WHEN	G
THEN	$Action$

2.5.1 Event: Proof Obligations

There may be feasibility POs: that there exist parameters P that will satisfy the guards G

$$\forall S, C \cdot A \wedge \exists V, x \cdot I \wedge G$$

2.5.2 Event: Maintaining Invariant

The event must maintain the invariant of the machine: essentially the invariant will be true before the event is scheduled and must remain true when the event terminates.

$$\forall S, C, V, x \cdot A \wedge I \wedge G \implies [Action] I$$

2.6 Machine Refinements

The form of a refinement machine is

Context	S_r, C_r
Variables	V_r
Invariant	I_r
Theorems	T_v^+
Events	$E_r \text{ refines } E$
Variant	E
	Var

where E_r represents a refined event and E represents new normal events.

2.6.1 Variables and Invariant

The variable V_r are in general a superset of the variables in the machine being refined.

The invariant is the invariant of the refined machine plus invariants for the new variables. In addition the invariant contains the refinement relation relating the state of the refined machine to the variables of the refining machine. This gives a *simulation* relation.

The proof obligations for the variables, invariant and theorems are similar to those for the machine given above. We will concentrate on the new proof obligations that arise from the refined events.

2.6.2 Proof Obligations

$$\forall V_i, V \cdot I_r \implies I$$

the new invariant must not allow behaviour that was not part of the refined machine's behaviour, excepting where the state of the refining machine is "orthogonal" to the refined machine.

2.7 Refined Events

Refined Events

Refined Events have the following form

ANY	x_r
WHEN	G_r
WITH	$w : W$
THEN	$Action_r$

2.7.1 Proof Obligations for Refined Events

guard refinement

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge A_r \wedge I \wedge I_r \implies G_r \implies G$$

witness

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \implies \exists w \cdot W$$

Simulation

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \wedge [Action_r]I_r \implies [Action]I$$

2.7.2 The Variant and Convergent Events

The variant (Var) is an expression that denotes either a finite set or a natural number.

The purpose of the variant is to show that all convergent events must terminate. This is achieved by showing that the size of the set, or the natural number value is strictly decreasing.

Natural number variant

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \implies Var \in \mathbb{N}$$

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \implies [Action_r]Var < Var$$

Set variant

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \implies \text{finite}(Var)$$

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge I \wedge I_r \wedge W \implies \text{card}([Action_r]Var) < \text{card}(Var)$$

A One Point Rule

Consider $\forall x \cdot x \in X \wedge x = e \implies P(x)$.

For any x in S , $x = e$ is either *true* or *false*. If it is *false* then the universal quantification is trivially *true*; if it is *true* then the quantification reduces to $P(e)$. So

$$(\forall x \cdot x \in X \wedge x = e \implies P(x)) = P(e)$$

By a similar argument,

$$(\exists x \cdot x \in X \wedge x = e \wedge P(x)) = P(e)$$

Strictly, each should be conjuncted with $\exists x \cdot x \in X \wedge x = e$.