Modularisation/Group Refinement/Views

Alexei Iliasov

Newcastle University

Modularisation

What the plugin does

- The plugin extends the Event B modelling language with the concept of a module
- A module is a parametrised Event B development associated with a module **interface**
- An interface defines a number of **operations**
- A specification is decomposed by including a module in a machine and connecting the two using operation calls and gluing invariants

What the plugin provides

- a new type of Event B component - a module interface (editor, pretty-printer and proof obligations generator)
- new machine constructs:**IMPLEMENTS** and **USES**
- new event attributes: **group** and **final**
- the ability to write operation calls in event actions
- additional proof obligations for operation calls
- additional proof obligations for implementation machines

Parking Lot

The task is to developep an access control and payment collection mechanisms for a parking lot. The following main requirements were identified:

1. no car may enter when there is no space left in the parking lot
2. a fare must be paid when a car leaves the parking lot
3. each time a car leaves the parking lot, the fare to be paid is determined by multiplying the total length of stay since the midnight (that is, including any previous stay(s)) by the cost of parking per unit of time
4. the amount paid in any single transaction is capped
5. at midnight, the accumulated parking time of all cars is reset to zero

Solution overview:

1. two gates are placed to control entry and exit
2. a payment collection machine is placed near to the exit gate in such a manner that a driver may use it before going through the exit gate
3. the exit gate does not open until the full payment is collected
4. the entrance gate does not open if the car park is full

Abstract model

the initial model describes the phenomena of cars entering and leaving the parking lot. It addresses the capacity restrictions although without exhibiting a concrete mechanism for controlling the number of cars entering the parking lot.

Model variables

- LOT_SIZE - the parking lot capacity (constant)
- $entered$ - the number of cars that have entered the parking lot
- $left$ - the number of cars that have left the parking lot
- hence, $left - entered$ is the current number of cars in the parking lot

INVARIANT

$entered \in \mathbb{N}$

$left \in \mathbb{N}$

$entered - left \in 0 \ldots LOT\_SIZE$

Model events

a new car appears:

$$\text{enter} \quad = \quad \begin{array}{l} \text{WHEN} \\ \qquad \mathit{entered} - \mathit{left} < \mathit{LOT\_SIZE} \\ \text{THEN} \\ \qquad \mathit{entered} := \mathit{entered} + 1 \\ \text{END} \end{array}$$

a car leaves:

$$\text{leave} \quad = \quad \begin{array}{l} \text{WHEN} \\ \qquad \mathit{entered} - \mathit{left} > 0 \\ \text{THEN} \\ \qquad \mathit{left} := \mathit{left} + 1 \\ \text{END} \end{array}$$

First refinement

In the first refinement the entrance is controlled by a gate. The the gate prevents a car from entering when there is no free space and also records the registration plate of an entering car.

Gate Module

The logic controlling a gate is easily decoupled from the main model. We decompose the model into the controller part and an entry gate

The first step of this decomposition is to define a gate module interface.

Gate variables

- *CAR* - car id (registration plate)
- *mcars* - the number of cars that has passed through the gate
- *current* - the id of the car in the front of the gate

INVARIANT

$$mcars \in \mathbb{N}$$
$$current \in CAR$$

Gate operations

when there is no car in front of the gate, a driver may press the gate button to try to open the gate:

$$
\boxed{
\begin{array}{lll}
\mathsf{carid} \leftarrow \mathsf{Button} & = & \mathrm{PRE} \\
 & & \quad \mathit{current} = \mathit{empty} \\
 & & \mathrm{POST} \\
 & & \quad \mathit{current}' \in \mathit{CAR} \setminus \{\mathit{empty}\} \\
 & & \quad \mathit{carid}' = \mathit{current} \\
 & & \mathrm{END}
\end{array}
}
$$

Gate operations

the car park controller orders the gate to open; the gate has sensors to observe whether the car has moved through the gate ($moved = \mathrm{TRUE}$) or stayed in front of the gate:

$$
\begin{array}{lcl}
\mathsf{moved} \leftarrow \mathsf{OpenGate} & = & \textsc{pre} \\
 & & \quad current \neq empty \\
 & & \textsc{post} \\
 & & \quad (moved' = \mathrm{TRUE} \land mcars' = mcars + 1 \land \\
 & & \qquad current' = empty) \lor \\
 & & \quad (moved' = \mathrm{FALSE} \land mcars' = mcars \land \\
 & & \qquad current' = current) \\
 & & \textsc{end}
\end{array}
$$

Gate operations

predicate $mcars' = mcars \wedge current' = current$ in

$$\boxed{\begin{array}{l}(moved' = \mathrm{TRUE} \wedge mcars' = mcars + 1 \wedge current' = empty) \vee \\ (moved' = \mathrm{FALSE} \wedge mcars' = mcars \wedge current' = current)\end{array}}$$

is necessary to indicate that *mcars* and *current* remain unchanged in the second branch of the post-condition. This is only required when a disjunction is used and not all variables are assigned new values in the disjunction branches

Operating the gate

to open the gate and let a car through it, the following has to happen:

- a driver must press the gate button (operation *Button*)
- the controller must activate the gate (operation *OpenGate*)

in our model, the main development models both driver's and controller's behaviour

First refinement machine

The first refinement imports the gate module interface. Prefix entry is used to avoid name clashes (with another gate added later on).

When a prefixed interface is imported, all its constants and sets appear prefixed in the importing context. This is not always convenient. We use type instantiation to replace the type of an imported module by a typing expression known in the importing context. We also define a property (an axiom) that equates a prefixed and unprefixed versions of constant *empty*.

USES entry : ParkingGate
  TYPES
    $entry\_CAR \mapsto CAR$
  PROPERTIES
    $entry\_empty = empty$

First refinement machine

two new variables are defined in the refinement machine. They help to link the states of the controller and the entry gate.

- *incar* - the id of an entering car
- *inmoved* - a flag indicating whether a car has passed through the (open) entry gate

INVARIANT

$incar \in CAR$
$inmoved \in BOOL$

Import invariant

it is necessary to provide an invariant relating the states of an imported module and the importing machine (import invariant)

without this, a module import does not make much sense as an overall model would be composed of two independently evolving systems

Import invariants

when there is no car at the gate, the gate car counter has the same value as the controller counter:

$$\boxed{\textit{inmoved} = \mathrm{FALSE} \implies \textit{entered} = \textit{entry\_mcars}}$$

when a car is passing through the entrance gate, only the gate counter has been incremented:

$$\boxed{\textit{inmoved} = \mathrm{TRUE} \implies \textit{entered} + 1 = \textit{entry\_mcars}}$$

Import invariants

when a car is passing through the gate there must be no other car at the gate:

$$\boxed{\mathit{inmoved} = \mathrm{TRUE} \implies \mathit{entry\_current} = \mathit{empty}}$$

when a car is coming through the entrance gate there is certainly free space in the parking lot:

$$\boxed{\mathit{inmoved} = \mathrm{TRUE} \land \mathit{entry\_current} \neq \mathit{empty} \implies \mathit{entered} - \mathit{left} < \mathit{LOT\_SIZE}}$$

Model events

a driver presses the gate button at the entrance gate (new event):

$$
\boxed{
\begin{array}{lll}
\text{UserPressButton} & = & \textsc{when} \\
& & \quad \mathit{entered} - \mathit{left} < \mathit{LOT\_SIZE} \\
& & \quad \mathit{entry\_current} = \mathit{empty} \\
& & \quad \mathit{inmoved} = \mathit{FALSE} \\
& & \textsc{then} \\
& & \quad \mathit{incar} := \textbf{entry\_Button} \\
& & \textsc{end}
\end{array}
}
$$

here **entry_Button** is a call of the *Button* operation from the *entry* module.

Model events

the parking lot controller orders the gate to open (new event):

$$\text{CtrlOpenGate} \quad = \quad \begin{array}{l} \textsc{when} \\ \quad \mathit{entry\_current} \neq \mathit{empty} \wedge \mathit{inmoved} = \mathrm{FALSE} \\ \textsc{then} \\ \quad \mathit{inmoved} := \textbf{entry\_OpenGate} \\ \textsc{end} \end{array}$$

Model events

finally, the *enter* event is refined to reflect the model changes:

$$
\begin{array}{lcl}
\mathsf{enter} & = & \textsc{when} \\
 & & \quad \mathit{inmoved} = \mathrm{TRUE} \\
 & & \textsc{then} \\
 & & \quad \mathit{entered} := \mathit{entered} + 1 \\
 & & \quad \mathit{inmoved} := \mathrm{FALSE} \\
 & & \textsc{end}
\end{array}
$$

Development structure

all proof obligations are discharged automatically (18 total)

Second refinement

the second refinement is very similar: we add another gate - an exit gate. the same module is imported with a new prefix to obtain two separate modules modelling two gates.

Second refinement

one interactive proof (17 total)

Third refinement

the third refinement step is concerned with keeping the record of car stays; this step introduces the notion of time

the definition of time will be used more than once and thus it is convenient to place in an interface

Third refinement

Fourth refinement

in this step, before a car may leave, the car driver must pay the amount determined by the length of stay since the midnight

the functionality of a device collecting payment is decoupled from the controller logic and is placed in a separate module

Fourth refinement

all proof obligations are discharged automatically (34 total)

Implementing Modules

A machine providing the realisation of an interface is said to *implement* the interface. This is recorded by adding the interfaces into the **IMPLEMENTS** section of a machine. The fact that a machine provides a correct implementation of interfaces is established by a number of static checks and a set of proof obligations. The latter appear automatically in the list of machine proof obligations. The implementation relation is maintained during machine refinement (subject to some syntactic constraints) and thus the bulk of the module implementation activity is the normal Event B refinement process.

Event Group

The first step of implementing an interface is to provide at least one event for each interface operation. In general, an operation is realised by a set of events (an event **group**). Some events play a special role of operation termination events and are called **final** events. A final event returns the control to a caller. It must satisfy the operation post-conditions but there is no need to prove the convergence of a final event.

Implementing **ParkingGate**: abstract machine

To simplify proofs, the initial implementation is a simple machine with few events mirroring the interface operations. The machine retains interface variables *current* and *mcars* and also defines the operation return variables $Button\_carid$ and $OpenGate\_moved$.

The names of the operation return variables are fixed for the first machine of a module implementation. In further refinements they may be replaced or removed using data refinement.

Implementing **ParkingGate**: abstract machine

The *button* event implements operation *Button* in a single atomic step. The fact that it is associated with operation *Button* is stated by GROUP Button. Being the only event in its operation group it is also a FINAL event.

$$
\begin{array}{l}
\textsc{machine}\ \textsf{iParkingGate}\ \textsc{implements} \\
\textsc{variables}\ \mathit{current}\ \mathit{mcars}\ \mathit{Button\_carid}\ \mathit{OpenGate\_moved} \\
\textsc{events} \\
\quad \textsf{button} \quad = \quad \textsc{final group}\ \textsf{Button} \\
\qquad\qquad\qquad\quad \textsc{when} \\
\qquad\qquad\qquad\qquad \mathit{current} = \mathit{empty} \\
\qquad\qquad\qquad\quad \textsc{then} \\
\qquad\qquad\qquad\qquad \mathit{current} :\in \mathit{CAR} \setminus \{\mathit{empty}\} \\
\qquad\qquad\qquad\qquad \mathit{Button\_carid} := \mathit{current} \\
\qquad\qquad\qquad\quad \textsc{end}
\end{array}
$$

Implementing **ParkingGate**: abstract machine

The machine declares two more events, both realising the *OpenGate* operation. The events are final and each one handles one of the cases of the *OpenGate* operation post-condition.

$$
\begin{array}{rcl}
\mathsf{gate\_succ} & = & \textsc{final group}\ \mathsf{OpenGate} \\
& & \textsc{when} \\
& & \quad \mathit{current} \neq \mathit{empty} \\
& & \textsc{then} \\
& & \quad \mathit{OpenGate\_moved} := \mathit{TRUE} \\
& & \quad \mathit{mcars} := \mathit{mcars} + 1 \\
& & \quad \mathit{current} := \mathit{empty} \\
& & \textsc{end} \\
\mathsf{gate\_nocar} & = & \textsc{final group}\ \mathsf{OpenGate} \\
& & \textsc{when} \\
& & \quad \mathit{current} \neq \mathit{empty} \\
& & \textsc{then} \\
& & \quad \mathit{OpenGate\_moved} := \mathit{FALSE} \\
& & \textsc{end}
\end{array}
$$

Development structure

one interactive proof (5 total)

Implementing **ParkingGate**: first refinement

new variables:

- *gate* - the gate state: open or closed
- *sensor* - the state of the car sensor placed; the sensor is placed on the parking lot of a gate
- *stage* - the current step of the gate operation

INVARIANT

$$gate \in GATE$$
$$sensor \in BOOL$$
$$stage \in 0 \ldots 3$$
$$stage = 1 \implies gate = OPEN$$
$$stage = 2 \implies gate = CLOSED$$

Implementing **ParkingGate**: first refinement

The refined implementation of the *OpenGate* operation includes events for opening and closing the gate.

$$
\begin{array}{lcl}
\text{open\_gate} & = & \textsc{group}\ \text{OpenGate} \\
& & \textsc{when} \\
& & \quad \mathit{gate} = \mathit{CLOSED} \wedge \mathit{stage} = 0 \\
& & \textsc{then} \\
& & \quad \mathit{gate} := \mathit{OPEN} \\
& & \quad \mathit{stage} := 1 \\
& & \textsc{end} \\
\text{close\_gate} & = & \textsc{group}\ \text{OpenGate} \\
& & \textsc{when} \\
& & \quad \mathit{stage} = 2 \\
& & \textsc{then} \\
& & \quad \mathit{gate} := \mathit{CLOSED} \\
& & \quad \mathit{stage} := 3 \\
& & \textsc{end}
\end{array}
$$

the gate detects whether a car has passed through the gate while the gate was open:

$$
\boxed{
\begin{array}{lll}
\text{readSensor} & = & \textsc{group}\ \text{OpenGate} \\
 & & \textsc{when} \\
 & & \quad \mathit{stage} = 1 \\
 & & \textsc{then} \\
 & & \quad \mathit{sensor} :\in \mathit{BOOL} \\
 & & \quad \mathit{stage} :\in 1, 2 \\
 & & \textsc{end}
\end{array}
}
$$

Development structure

all proof obligations are discharged automatically (26 total)

Implementing **ParkingGate**: second refinement

To prove the convergence of anticipated event *readSensor*, the car sensor waits for a car for a given time interval. The time model is imported from the *Clock* interface.

$$
\begin{array}{lcl}
\text{readSensor} & = & \textsc{group}\ \text{OpenGate} \\
 & & \textsc{when} \\
 & & \quad \mathit{stage} = 1 \\
 & & \quad \mathit{prev} < \mathit{delay} \\
 & & \textsc{then} \\
 & & \quad \mathit{sensor}, \mathit{stage} :\mid (\mathit{sensor}' = \mathrm{TRUE} \wedge \mathit{stage}' = 2) \vee \\
 & & \qquad\qquad (\mathit{sensor}' = \mathrm{FALSE} \wedge \mathit{stage}' = 1) \\
 & & \quad \mathit{time} := \textbf{currentTime} \\
 & & \textsc{end}
\end{array}
$$

Development structure

two interactive proofs (8 total)

The overall development structure

Implementation

- New syntatic elements at the level of unchecked machines
- Pure Event-B at the level of statically checked machines
- Custom static checking rules
- Additional proof obligations

Version 2.0

- Improved PO generation
- Process notion
- Bug fixes
- ProB and Camille integration

Developments/Case Studies

- SSF AOCS (Abo, bscw)
- Parking lot tutorial (Ncl)
- BepiColombo 2 (SSF, internal)
- NFS (Ncl, ongoing)

Extending Event-B

Event-B is a set of plugins to the Rodin platform

good:

- extending the Rodin database is easy
- contributing new static checker rules and proof obligations is easy
- + lots of documentation appeared recently on the Event-B wiki

bad:

- Eclipse/Java/OSes/bits (getting better)
- inconsistently exposed API (not enough abstraction layers?)
- extensions integration (text editor)
- the status and the future of EMF integration
- some Event-B plugins could be Rodin plugins (some progress: new pretty-printer)

Questions?

The full version and the development are available at:

http://wiki.event-b.org/index.php/Modularisation_Plug-in

next: Group refinement

Abstract Model

MACHINE m0

VARIABLES v

INVARIANT

$v \in \mathbb{N}$

EVENTS

$\mathsf{e} \quad = \quad$ BEGIN $v := 2$ END

END

Refinement objective

- v is to be replaced by a and b: $v = a + b$
- a and b may not be updated at the same time

Refinement

MACHINE m1_classic
  REFINES m0
  VARIABLES a, b, pc, al, bl
  INVARIANT

$$a + b = v \land pc \in 1..3$$

  EVENTS

$$\begin{array}{lcl}
\text{e1} & = & \text{WHEN } pc = 1 \text{ THEN } al := 1 \| pc := 2 \text{ END} \\
\text{e2} & = & \text{WHEN } pc = 2 \text{ THEN } bl := 1 \| pc := 3 \text{ END} \\
\text{e3 REF } e & = & \text{WHEN} \\
 & & \quad pc = 3 \\
 & & \text{THEN} \\
 & & \quad a, b := al, bl \\
 & & \quad pc := 1 \\
 & & \text{END}
\end{array}$$

END

Refinement proof + model

MACHINE m1_classic
REFINES m0
VARIABLES $\underline{a, b, pc}, al, bl$
INVARIANT
$a + b = v \wedge \underline{pc \in 1..3}$
EVENTS

$$
\begin{array}{lcl}
e1 & = & \text{WHEN } \underline{pc = 1} \text{ THEN } al := 1 \| \underline{pc := 2} \text{ END} \\
e2 & = & \text{WHEN } \underline{pc = 2} \text{ THEN } bl := 1 \| \underline{pc := 3} \text{ END} \\
e3 \text{ REF } e & = & \text{WHEN} \\
 & & \quad \underline{pc = 3} \\
 & & \text{THEN} \\
 & & \quad \underline{a, b := al, bl} \\
 & & \quad \underline{pc := 1} \\
 & & \text{END}
\end{array}
$$

END

Another example: swap

MACHINE m
VARIABLES a, b
INVARIANT $a \in \mathbb{N} \land b \in \mathbb{N}$
INITIALISATION $a :\in \mathbb{N} \parallel b :\in \mathbb{N}$
EVENTS

$$\mathsf{swap} \quad = \quad \text{BEGIN } a, b := b, a \text{ END}$$

END

Refinement objective

- a and b may not be updated at the same time

Refinement objective

$$
\begin{array}{l}
\textsc{refinement}\ \mathsf{m1a} \\
\quad \textsc{refines}\ \mathsf{m} \\
\quad \textsc{variables}\ a, b, x, y, pc \\
\quad \textsc{invariant} \\
\qquad x \in \mathbb{N} \land y \in \mathbb{N} \land pc \in 0 \ldots 3 \\
\qquad pc = 0 \implies x = a \land y = b \\
\qquad pc = 1 \implies x = a + b \land y = b \\
\qquad pc = 2 \implies x = a + b \land y = a \\
\qquad pc = 3 \implies x = b \land y = a \\
\quad \textsc{initialisation} \\
\qquad a, x :\mid a' \in \mathbb{N} \land x' = a' \\
\qquad b, y :\mid b' \in \mathbb{N} \land y' = b' \\
\qquad pc := 0 \\
\quad \textsc{events} \\
\qquad \mathsf{step1} \ = \ \textsc{when}\ pc = 0\ \textsc{then}\ x := x + y \parallel pc := 1\ \textsc{end} \\
\qquad \mathsf{step2} \ = \ \textsc{when}\ pc = 1\ \textsc{then}\ y := x - y \parallel pc := 2\ \textsc{end} \\
\qquad \mathsf{step3} \ = \ \textsc{when}\ pc = 2\ \textsc{then}\ x := x - y \parallel pc := 3\ \textsc{end} \\
\qquad \mathsf{swap} \ = \ \textsc{when}\ pc = 3\ \textsc{then}\ a, b := x, y \parallel pc := 0\ \textsc{end} \\
\textsc{end}
\end{array}
$$

Refinement objective

REFINEMENT m1a

REFINES m

VARIABLES $\underline{a, b}, x, y, pc$

INVARIANT

$$\underline{x \in \mathbb{N} \land y \in \mathbb{N} \land pc \in 0 \ldots 3}$$
$$\underline{pc = 0 \implies x = a \land y = b}$$
$$\underline{pc = 1 \implies x = a + b \land y = b}$$
$$\underline{pc = 2 \implies x = a + b \land y = a}$$
$$\underline{pc = 3 \implies x = b \land y = a}$$

INITIALISATION

$$\underline{a, x :\mid a' \in \mathbb{N} \land x' = a'}$$
$$\underline{b, y :\mid b' \in \mathbb{N} \land y' = b'}$$
$$pc := 0$$

EVENTS

$$\begin{array}{lcl}
\mathsf{step1} & = & \text{WHEN } pc = 0 \text{ THEN } x := x + y \parallel \underline{pc := 1} \text{ END} \\
\mathsf{step2} & = & \text{WHEN } pc = 1 \text{ THEN } y := x - y \parallel \underline{pc := 2} \text{ END} \\
\mathsf{step3} & = & \text{WHEN } pc = 2 \text{ THEN } x := x - y \parallel \underline{pc := 3} \text{ END} \\
\mathsf{swap} & = & \text{WHEN } \underline{pc = 3} \text{ THEN } \underline{a, b := x, y \parallel pc := 0} \text{ END}
\end{array}$$

END

A pattern

- atomicity refinement
- $+$ data refinement
- $=$ housekeeping event $+$ new variables $+$ gluing invariant

Pattern mechanisation

- new proof obligations for event split refinement for the scope of a given refinement step
- override split refinement semantics: the combined effect of all the refinement events achieves the effect of the abstract event
- some ordering constraints on events

Alternative Refinement

MACHINE m1_alt

REFINES m0

VARIABLES a, b

INVARIANT

$a + b = v$

EVENTS

e1 REF e $=$ BEGIN $a := 1$ END

e2 REF e $=$ BEGIN $b := 1$ END

END

Refinement

MACHINE m1_classic

REFINES m0

VARIABLES $\underline{a, b, pc}, al, bl$

INVARIANT

$a + b = v \wedge \underline{pc \in 1..3}$

EVENTS

$$
\begin{array}{lcl}
\mathsf{e1} & = & \text{WHEN } \underline{pc = 1} \text{ THEN } al := 1 \| \underline{pc := 2} \text{ END} \\
\mathsf{e2} & = & \text{WHEN } \underline{pc = 2} \text{ THEN } bl := 1 \| \underline{pc := 3} \text{ END} \\
\mathsf{e3} \text{ REF } e & = & \text{WHEN} \\
 & & \quad \underline{pc = 3} \\
 & & \text{THEN} \\
 & & \quad \underline{a, b := al, bl} \\
 & & \quad \underline{pc := 1} \\
 & & \text{END}
\end{array}
$$

END

Refinement objective

$$
\begin{array}{l}
\textsc{refinement}\ \mathsf{m1a} \\
\quad \textsc{refines}\ \mathsf{m} \\
\quad \textsc{variables}\ \underline{a, b}, x, y, pc \\
\quad \textsc{invariant} \\
\qquad \underline{x \in \mathbb{N} \land y \in \mathbb{N} \land pc \in 0 \ldots 3} \\
\qquad \underline{pc = 0 \implies x = a \land y = b} \\
\qquad \underline{pc = 1 \implies x = a + b \land y = b} \\
\qquad \underline{pc = 2 \implies x = a + b \land y = a} \\
\qquad \underline{pc = 3 \implies x = b \land y = a} \\
\quad \textsc{initialisation} \\
\qquad \underline{a, x :\mid a' \in \mathbb{N} \land x' = a'} \\
\qquad \underline{b, y :\mid b' \in \mathbb{N} \land y' = b'} \\
\qquad pc := 0 \\
\quad \textsc{events} \\
\qquad \begin{array}{lcl}
\mathsf{step1} & = & \textsc{when}\ pc = 0\ \textsc{then}\ x := x + y \parallel \underline{pc := 1}\ \textsc{end} \\
\mathsf{step2} & = & \textsc{when}\ pc = 1\ \textsc{then}\ y := x - y \parallel \underline{pc := 2}\ \textsc{end} \\
\mathsf{step3} & = & \textsc{when}\ pc = 2\ \textsc{then}\ x := x - y \parallel \underline{pc := 3}\ \textsc{end} \\
\mathsf{swap} & = & \textsc{when}\ \underline{pc = 3}\ \textsc{then}\ \underline{a, b := x, y \parallel pc := 0}\ \textsc{end}
\end{array} \\
\textsc{end}
\end{array}
$$

31/0 POs

Alternative Refinement

REFINEMENT m1b

REFINES m

VARIABLES a, b

INVARIANT $a \in \mathbb{N} \land b \in \mathbb{N}$

INITIALISATION $a :\in \mathbb{N} \parallel b :\in \mathbb{N}$

EVENTS

$$\begin{aligned} \mathsf{swap1} &= \text{BEGIN } a := a + b \text{ END} \\ \mathsf{swap2} &= \text{BEGIN } b := a - b \text{ END} \\ \mathsf{swap3} &= \text{BEGIN } a := a - b \text{ END} \end{aligned}$$

END

2/0 POs

demo

Questions?

The plugin and some documentation is here:

http://wiki.event-b.org/index.php/Group_refinement_plugin

next: Views