Rodin User and Developer Workshop

Transforming Guarded Events into Pre-conditioned Operations

Jean-Raymond Abrial (Marseille) and Wen Su (Shanghai)

June 2014

Outline

- The problem

- Its solution

- Questions???

- An example (time permitting)

In the Event-B Book (Chapter 15)

- Conditionals

- Loops

- Nothing for procedure calls

Different Usage of Operations and Events

- An operation can be called

- An event can be observed

Pre-conditioned Operation Definition

- Some (possibly missing) parameters

- A pre-condition: it must be true before the operation is called

- A post-condition

Guarded Event Definition

- Some (possibly missing) parameters

- A guard: it must be true for the event to be observable

- A post-condition

Guard and Pre-condition Refinements are Different

- Pre-conditions are weakened

- Guards are strengthened

Guard and Pre-condition Behaviours are Different

- An operation called with a false pre-condition results in a crash

- An event with a false guard is not observable

Operations and Events are both Useful

- Operations needed when specifying programs

- Events needed when modelling systems

A Challenge

- It seems impossible to define operations by means of events

- Because of the difference between pre-conditions and guards

- The intend of this presentation is to show how it is possible

Outline

- The problem

- Its solution

- Questions???

- An example (time permitting)

Defining an Operation with an Event

- The operations P is first defined as an event (and so refined):

P $\widehat{=}$
any
fp
when
$G(v, fp)$
then
$A(v, v', fp)$
end

- It can be proved to maintain some invariant $I(v)$:

$$I(v) \ \wedge \ G(v, fp) \ \wedge \ A(v, v', fp) \ \Rightarrow \ I(v')$$

Transforming the Operation into an Event (1)

We define the following set:

$$PR = \{call, return, undefined\}$$

We define a variable $prog$ (it is initialized to $undefined$):

$$prog \in PR$$

Transforming the Operation into an Event (2)

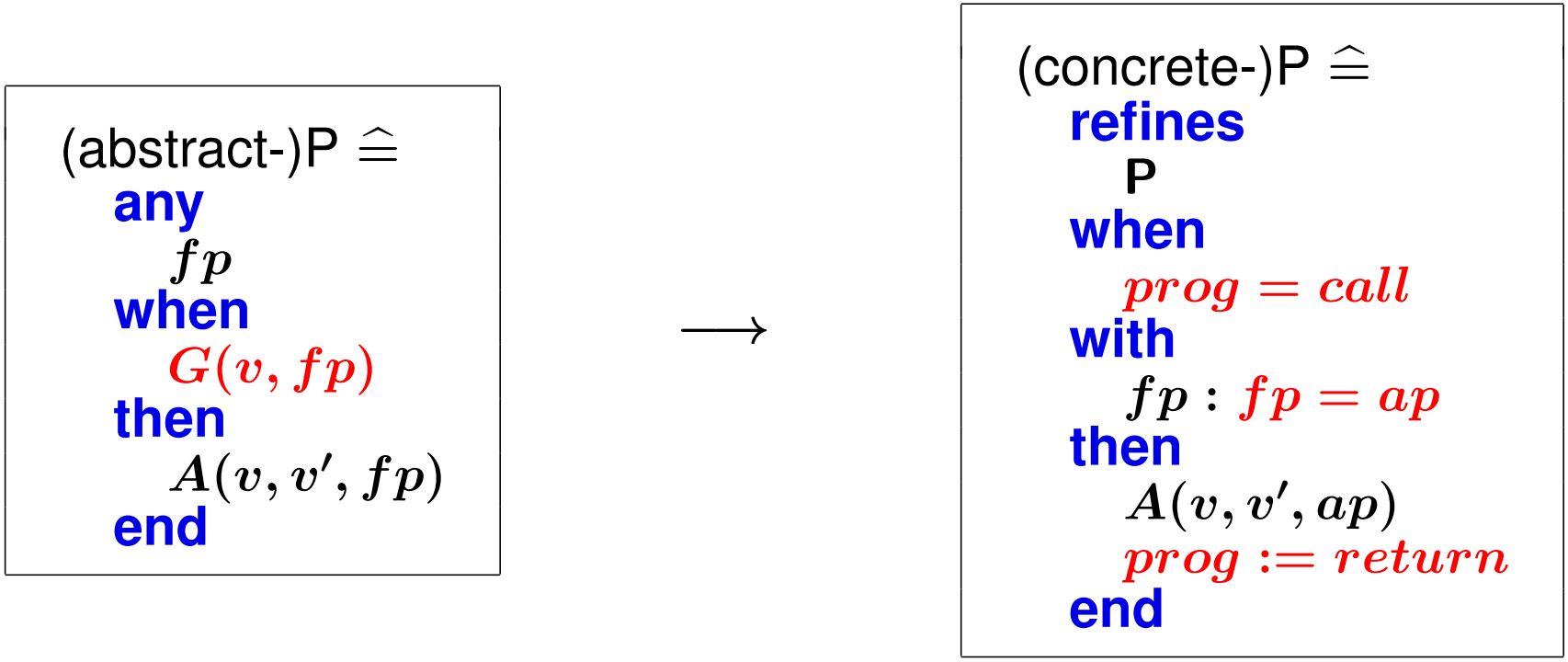
We define a call to P as follows:

Call_to_P $\widehat{=}$
when
$prog = undefined$
then
$ap := ...$
$prog := call$
end

- Variable ap contains the actual parameters of the operation

Transforming the Operation into an Event (3)

- We then refine P as follows:



- For proving the refinement:

$$prog = call \;\Rightarrow\; G(v, ap)$$

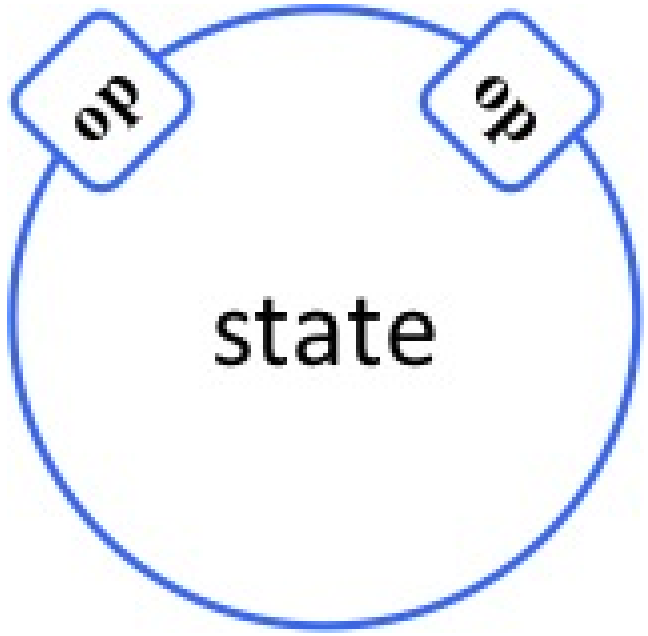
Transforming the Operation into an Event (4)

- Finally, we have a return from P:

Return_from_P $\widehat{=}$
when
$prog = return$
then
...
$prog := undefined$
end

Generalisation: Building a Module

- A module is made of a state surrounded by some operations



- Users of the modules can only use the operation

Outline

- The problem

- Its solution

- Questions???

- An example (time permitting)

Some Questions

- Have you got some examples where this would be useful?

- How about defining a plug-in for doing this?

- Would it be useful to generate corresponding code?

- Connection between Classical-B and Event-B?

Outline

- The problem

- Its solution

- Questions???

- An example (time permitting)

Example: Sorting (1)

$$\mathbf{axm1}: \quad n \in \mathbb{N}1$$

$$\mathbf{axm2}: \quad file\_0 \in 1 \mathinner{..} n \rightarrowtail \mathbb{N}$$

Example: Sorting (2) Defining the State of a Module

- In an initial machine we define the set $file$ to be sorted.

- We also define a variable k supposed to be in the domain of $file$.

$$\begin{array}{l} \mathbf{inv0\_1}: \quad file \in 1 \mathinner{..} n \rightarrowtail \mathbb{N} \\ \mathbf{inv0\_2}: \quad \mathrm{ran}(file) = \mathrm{ran}(file\_0) \\ \mathbf{inv0\_3}: \quad k \in \mathrm{dom}(file) \end{array}$$

These variables are initialised as follows:

INIT $\widehat{=}$
begin
$file := file\_0$
$k := 1$
end

Example: Sorting (3)

- Now, we surround this state with the following events, thus forming a module

search_min $\widehat{=}$
any i **where**
$i \in 1 \mathinner{..} n$
then
$k := file^{-1}(\min(file[i \mathinner{..} n]))$
end

swap $\widehat{=}$
any i j **where**
$i \in 1 \mathinner{..} n$
$j \in 1 \mathinner{..} n$
then
$file := file \Leftarrow \{i \mapsto file(j)\} \Leftarrow \{j \mapsto file(i)\}$
end

It is easy to prove that these events maintain the three invariants **inv0_1**, **inv0_2**, and **inv0_3**.

Example: Sorting (4)

We finally define the following sort event (it is defined so far in a non-deterministic way):

sort $\widehat{=}$
begin
$file :| file' \in 1 \mathinner{..} n \rightarrowtail \mathbb{N} \ \wedge \ \mathbf{ran}(file') = \mathbf{ran}(file\_0) \ \wedge$
$(\forall p, q \cdot p \in 1 \mathinner{..} n \ \wedge \ q \in 1 \mathinner{..} n \ \wedge \ p < q \ \Rightarrow \ file'(p) < file'(q))$
end

Example: Sorting (5)

- In the refinement, we first define the following enumerated set:

$$\begin{aligned} PR \;=\; \{&call\_search\_min,\\ &return\_search\_min,\\ &call\_swap,\\ &return\_swap,\\ &undefined\} \end{aligned}$$

- We define the following variables l and $prog$:

$$\mathbf{inv1\_1}: \; l \in 0 \,..\, n$$

$$\mathbf{inv1\_2}: \; prog \in PR$$

Example: Sorting (6)

- Besides the initialisation, we then define some new events

- The new events do no use the variables $file$ and k, they only call the operations

INIT $\widehat{=}$
begin
$file := file\_0$
$k := 1$
$l := 0$
$prog := undefined$
end

sort_1 $\widehat{=}$
when
$l < n$
$prog = undefined$
then
$prog := call\_search\_min$
end

sort_2 $\widehat{=}$
when
$prog = return\_search\_min$
then
$prog := call\_swap$
end

sort_3 $\widehat{=}$
when
$prog = return\_swap$
then
$l := l + 1$
$prog = undefined$
end

sort_4 $\widehat{=}$
refines **sort** **when** $l = n$ **then** skip **end**

Example: Sorting (7) Refining the Operations

search_min $\widehat{=}$
refines
search_min
when
$prog = call\_search\_min$
with
$i = l + 1$
then
$k := file^{-1}(\min(file[l+1\,..\,n]))$
$prog := return\_search\_min$
end

swap $\widehat{=}$
refines
swap
when
$prog = call\_swap$
with
$i = l + 1$
$j = k$
then
$file := file \lhd \{l+1 \mapsto file(k)\} \lhd \{k \mapsto file(l+1)\}$
$prog := return\_swap$
end

Example: Sorting (8)

We have then to prove the pre-conditions:

inv1_3: $prog = call\_search\_min \Rightarrow l + 1 \in 1 \,..\, n$

inv1_4: $prog = call\_swap \Rightarrow l + 1 \in 1 \,..\, n$

inv1_5: $prog = call\_swap \Rightarrow k \in 1 \,..\, n$

Example: Sorting (9)

- Finally, we want to prove **thm1_1** saying that $file$ is indeed sorted when $l = n$ holds:

thm1_1: $l = n \Rightarrow (\forall p, q \cdot p \in 1 \mathinner{..} n \wedge q \in 1 \mathinner{..} n \wedge p < q \Rightarrow file(p) < file(q))$

- This theorem is needed to prove that the event sort_4 refines the abstract event sort.
- More invariants are needed for this
- This development required 96 proof obligations
- All proved automatically, except 3 of them proved interactively

Example: Sorting (10) Final Program

- Events INIT, and sort_1 to sort_4, lead to the following program:

```
sort ≙
  l := 0;
  while l < n do
    swap(l + 1, search_min(l + 1));
    l := l + 1
  end
```