Rodin User and Developer Workshop

Transforming Guarded Events into Pre-conditioned Operations

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

June 2014

- The problem

- Its solution

- Questions???

- An example (time permitting)

- Conditionals

- Loops

- Nothing for procedure calls

- Almost nothing for sequencing

- An operation can be called

- An event can be observed

- Some (possibly missing) parameters

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

- A post-condition

- Some (possibly missing) parameters

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

- A post-condition

- Pre-conditions are weakened

- Guards are strengthened

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

- An event with a false guard is not observable

- Operations needed when specifying programs

- Events needed when modelling systems

- It seems impossible to define operations by means of events

- Because of the difference between pre-conditions and guards

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

- The problem

- Its solution

- Questions???

- An example (time permitting)

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

$$
\begin{array}{l}
\text{P} \ \widehat{=} \\
\quad \textbf{any} \\
\qquad fp \\
\quad \textbf{when} \\
\qquad G(v, fp) \\
\quad \textbf{then} \\
\qquad A(v, v', fp) \\
\quad \textbf{end}
\end{array}
$$

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

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

We define the following set:

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

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

$$prog \in PR$$

We define a call to P as follows:

$$
\begin{aligned}
&\text{Call\_to\_P} \;\widehat{=} \\
&\quad \textbf{when} \\
&\quad\quad \dots \\
&\quad\quad prog = undefined \\
&\quad \textbf{then} \\
&\quad\quad \dots \\
&\quad\quad ap := \dots \\
&\quad\quad prog := call \\
&\quad\quad \dots \\
&\quad \textbf{end}
\end{aligned}
$$

- Variable ap contains the actual parameters of the operation

- We then refine P as follows:

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

$\longrightarrow$

(concrete-)P $\widehat{=}$
   refines
    P
   when
    $prog = call$
   with
    $fp : fp = ap$
   then
    $A(v, v', ap)$
    $prog := return$
   end

- For proving the refinement:

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

- Finally, we have a return from P:

$$
\begin{aligned}
&\text{Return\_from\_P} \;\widehat{=} \\
&\quad \textbf{when} \\
&\qquad prog = return \\
&\quad \textbf{then} \\
&\qquad ... \\
&\qquad prog := undefined \\
&\quad \textbf{end}
\end{aligned}
$$

- A module is made of a state surrounded by some operations
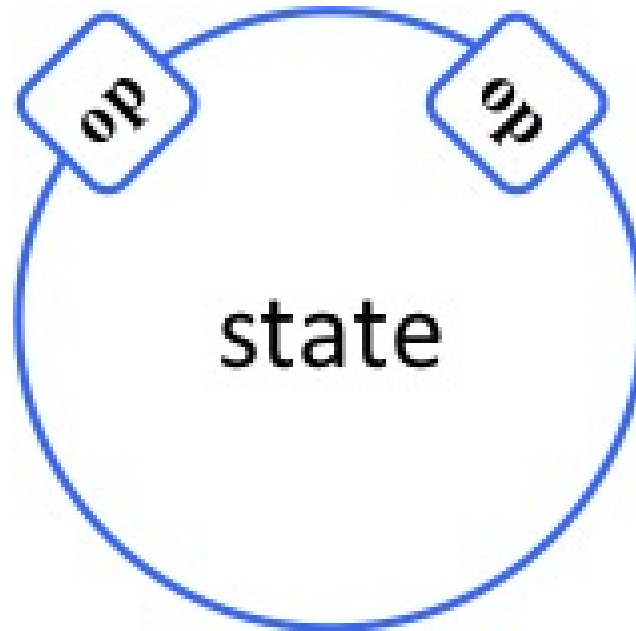
- Users of the modules can only use the operation

- The problem

- Its solution

- <u>Questions</u>???

- An example (time permitting)

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

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

- Would it be useful to generate corresponding code?

- Does this connect somehow Classical-B and Event-B?

- The problem

- Its solution

- Questions???

- An example (time permitting)

- The file to be sorted:

$$\text{axm1}: \ n \in \mathbb{N}1$$

$$\text{axm2}: \ file\_0 \in 1\mathinner{\ldotp\ldotp}n \rightarrowtail \mathbb{N}$$

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

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

$$\mathbf{inv0\_1}: \ file \in 1 \mathrel{.\,.} n \rightarrowtail \mathbb{N}$$

$$\mathbf{inv0\_2}: \ \mathrm{ran}(file) = \mathrm{ran}(file\_0)$$

$$\mathbf{inv0\_3}: \ k \in \mathrm{dom}(file)$$

These variables are initialised as follows:

$$
\begin{aligned}
&\text{INIT} \ \ \widehat{=} \\
&\quad \textbf{begin} \\
&\qquad file := file\_0 \\
&\qquad k := 1 \\
&\quad \textbf{end}
\end{aligned}
$$

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

$$
\begin{aligned}
&\text{search\_min} \quad \widehat{=} \\
&\quad \textbf{any } i \textbf{ where} \\
&\qquad i \in 1 \mathbin{..} n \\
&\quad \textbf{then} \\
&\qquad k := file^{-1}(\min(file[i \mathbin{..} n])) \\
&\quad \textbf{end}
\end{aligned}
$$

$$
\begin{aligned}
&\text{swap} \quad \widehat{=} \\
&\quad \textbf{any } i \ j \textbf{ where} \\
&\qquad i \in 1 \mathbin{..} n \\
&\qquad j \in 1 \mathbin{..} n \\
&\quad \textbf{then} \\
&\qquad file := file \mathbin{\vartriangleleft\mkern-14mu-} \{i \mapsto file(j)\} \mathbin{\vartriangleleft\mkern-14mu-} \{j \mapsto file(i)\} \\
&\quad \textbf{end}
\end{aligned}
$$

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

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

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

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

$$PR = \{call\_search\_min,$$
$$return\_search\_min,$$
$$call\_swap,$$
$$return\_swap,$$
$$undefined\}$$

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

$$inv1\_1 : l \in 0 .. n$$

$$inv1\_2 : prog \in PR$$

- 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

$$
\begin{aligned}
&\text{INIT } \ \widehat{=} \\
&\quad \textbf{begin} \\
&\qquad file := file\_0 \\
&\qquad k := 1 \\
&\qquad l := 0 \\
&\qquad prog := undefined \\
&\quad \textbf{end}
\end{aligned}
$$

$$
\begin{aligned}
&\text{sort\_1 } \ \widehat{=} \\
&\quad \textbf{when} \\
&\qquad l < n \\
&\qquad prog = undefined \\
&\quad \textbf{then} \\
&\qquad prog := call\_search\_min \\
&\quad \textbf{end}
\end{aligned}
$$

$$
\begin{aligned}
&\text{sort\_2 } \ \widehat{=} \\
&\quad \textbf{when} \\
&\qquad prog = return\_search\_min \\
&\quad \textbf{then} \\
&\qquad prog := call\_swap \\
&\quad \textbf{end}
\end{aligned}
$$

$$
\begin{aligned}
&\text{sort\_3 } \ \widehat{=} \\
&\quad \textbf{when} \\
&\qquad prog = return\_swap \\
&\quad \textbf{then} \\
&\qquad l := l + 1 \\
&\qquad prog = undefined \\
&\quad \textbf{end}
\end{aligned}
$$

$$
\begin{aligned}
&\text{sort\_4 } \ \widehat{=} \\
&\quad \textbf{refines sort when } l = n \textbf{ then } \text{skip } \textbf{end}
\end{aligned}
$$

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

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

We have then to prove the pre-conditions:

$$\mathbf{inv1\_3:} \quad prog = call\_search\_min \implies l + 1 \in 1 .. n$$

$$\mathbf{inv1\_4:} \quad prog = call\_swap \implies l + 1 \in 1 .. n$$

$$\mathbf{inv1\_5:} \quad prog = call\_swap \implies k \in 1 .. n$$

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

$$\textbf{thm1\_1:} \quad l = n \;\Rightarrow\; (\forall\, p, q \cdot p \in 1\mathbin{..}n \;\wedge\; q \in 1\mathbin{..}n \;\wedge\; p < q \;\Rightarrow\; file(p) < file(q))$$

- 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

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

$$
\begin{aligned}
&\text{sort} \quad \widehat{=} \\
&\quad l := 0; \\
&\quad \textbf{while } l < n \textbf{ do} \\
&\quad\quad \textbf{swap}(l + 1, \textbf{search\_min}(l + 1)); \\
&\quad\quad l := l + 1 \\
&\quad \textbf{end}
\end{aligned}
$$