Event-B Modeling in Isabelle/HOL

1 Introduction

This document describes the **model** command provided by this theory. The command integrates a dataspace representation using lenses [3] with standard Isabelle locales [2]. It supports refinement through additional proof obligations.

The approach follows the standard Event-B modeling methodology [1] and builds on top of the Igloo framework for event-based models of distributed systems [4].

2 The model Command

2.1 Syntax

The general syntax for an Event-B model definition looks like this:

model $('a_1,\ldots,'a_n)$ *model-name* =
[**extends** *locale-expression*]
variables var_1 :: $type_1$ $\cdots$ var_j :: $type_j$
initialise "$var_1 = v_1$" $\cdots$ "$var_j = v_j$"
invariants
inv_1 : "P_1 $var_1 \cdots var_j$"

inv_i : "P_i $var_1 \cdots var_j$" [**uses** inv_a ...]
events
E_1 $\alpha_{1,1} \ldots \alpha_{1,l_1}$ **when** $G_{1,1} \cdots G_{1,m_1}$ **do** $A_{1,1} \cdots A_{1,n_1}$

| E_k $\alpha_{k,1} \ldots \alpha_{k,l_k}$ **when** $G_{k,1} \cdots G_{k,m_k}$ **do** $A_{k,1} \cdots A_{k,n_k}$
[**refines** $model\text{-}name_0$ **via** $mediator\text{-}eq^+$ **where** π]

The syntax for *locale-expression* is described in [2]. In the guard and after expressions (G and A respectively) each variable *var* is bound. In after expressions A a primed version var' (the variable after an event E occurred) is bound too.

When declaring a refinement, each variable from $model\text{-}name_0$ must have a corresponding mediator equality *mediator-eq* of the form "$var_0 = \ldots$". This function π is specified infix:

$E_a \twoheadrightarrow model\text{-}name_0.E_b$

2.2 Semantics

Each invocation defines a new datatype *model-name.event* with constructors $\textit{model-name.}E_1, \ldots, \textit{model-name.}E_k$ and *model-name.Skip*.

A simple invocation (no refinement) of the command will propose the following proof obligations:

1. $var = v \Longrightarrow P\ var$
2. $[\![P\ var;\ G\ var;\ A\ var\ var']\!] \Longrightarrow P\ var'$

The invocation defines a new locale extension with an associated lens for each variable *var* in the sense of a dataspace [3]:

$$
\begin{array}{rl}
\textbf{locale} & \textit{model-name} = \\
\textbf{fixes} & var_{1\,L} :: type_1 \Longrightarrow {'st} \\
 & \vdots \\
\textbf{and} & var_{j\,L} :: type_j \Longrightarrow {'st} \\
\textbf{assumes} & \textit{dataspace-assms}
\end{array}
$$

The *dataspace-assms* state that all lenses var_L satisfy *vwb-lens* and mutual exclusivity.

2.2.1 Refinement

Refinement introduces additional proof obligations. It relies on a function π mapping the new model's event type to the old model's event type. This function is automatically defined. By default all constructors are skipped during refinement (π maps events to *old-model.Skip* unless another definition is provided).

The additional proof obligations will become:

1. $G\ (m\ (get_{var_v}\ s)) \wedge A\ (m\ (get_{var_v}\ s))\ (m\ (get_{var_v}\ s'))$

2.2.2 Generated Constants and Theorems

By default for each *inv* a standard invariant theorem is proved of the form:
inv: $reach\ ES\ ?s \Longrightarrow P\ (get_{var_{1\,v}}\ ?s) \cdots (get_{var_{j\,v}}\ ?s)$
Whenever a refinement took place a refinement theorem is proved:
$\textit{refines-model-name}_0$:
$\textit{model-name}_0\ ?var \Longrightarrow ES \sqsubseteq_\pi\textit{-model-name}_0\ \textit{model-name}_0.ES\ (m\ ?var)$

3 The *model-cases* Method and *eventb* Theorems

To keep proof obligations as simple as possible, the infrastructure needs to be able to split facts about the before-and-after predicates.

To this end this theory introduces a new named attribute *eventb*. Core theorems for splitting facts are:

lemma *eventb-if*[*case-names if-True if-False*, *eventb*]:
  assumes ‹$Q \Longrightarrow P\ x$› **and** ‹$\neg\ Q \Longrightarrow P\ y$›
  shows ‹P (*if* Q *then* x *else* y)›
  using *assms* **by** *simp*

lemma *eventb-case-sum*[*case-names Inl Inr*, *eventb*]:
  assumes ‹$\bigwedge a.\ x = \mathit{Inl}\ a \Longrightarrow P\ (f\ a)$› **and** ‹$\bigwedge b.\ x = \mathit{Inr}\ b \Longrightarrow P\ (g\ b)$›
  shows ‹P (*case-sum* $f\ g\ x$)›
  using *assms* **by** (*auto split*: *sum.split*)

lemma *eventb-case-option*[*case-names None Some*, *eventb*]:
  assumes ‹$mx = \mathit{None} \Longrightarrow P\ y$› **and** ‹$\bigwedge x.\ mx = \mathit{Some}\ x \Longrightarrow P\ (f\ x)$›
  shows ‹P (*case-option* $y\ f\ mx$)›
  using *assms* **by** (*auto split*: *option.split*)

These named theorems facilitate structured case analysis which can predictably be dealt with using *model-cases*. For example a refinement proof might look like this (only interesing cases need to be considered):

proof *model-cases*
  case *inv*:E_a
  then show *?thesis* **..**
next
  case *refine*:E_a
  then show *?thesis* **..**
qed *auto*

4 Example: Leader Election

This chapter compares the running example of the Leader Election using the newly defined command. The goal is to highlight, how the command is able to automate model definitions and goal construction. The user can focus on the interesting bits, no auxiliary constructions and lemmas are necessary.

4.1 Abstract System Model

model ($'b$::*countable*) *election0* =
  variables *leader* :: $'b \Rightarrow$ *bool*
  initialise *leader* = (λ-. *False*)
  invariants
    no-multiple-leaders: $\forall$ *i j* . *leader i* $\longrightarrow$ *leader j* $\longrightarrow$ *i* = *j*
  events
    Elect (*i*:$'b$) **when** $\forall$ *j*. *leader j* $\longrightarrow$ *i* = *j* **do** *leader*$'$ = *leader*(*i* := *True*)
  by *auto*

thm *election0.ES-def*
thm *election0.no-multiple-leaders*

4.1.1 Satisfiability

It easy to define non-sensical locales in Isabelle and the standard way to show that a locale is not non-sensical is to show that there is at least one inhabitant.

Indeed, the locale has a trivial inhabitant:

interpretation *ec0*: *election0* 1_L
  by (*rule election0.intro*, *simp*)

thm *ec0.no-multiple-leaders*[*unfolded id-lens-def*, *simplified*, *folded id-lens-def*]

4.2 Protocol Model

record $'a$ *local1* =
  elected1 :: *bool*
  chan1 :: $'a$ *set*

abbreviation *add-msg-to-chan1* **where**
  add-msg-to-chan1 s x msg $\equiv$ *s*(*x* := *s x*(| *chan1* := *insert msg* (*chan1* (*s x*)) |))

lemma (**in** *ringnetwork*) *topEqI*: $[\![\bigwedge z.\ z \neq x \Longrightarrow z < x]\!] \Longrightarrow x = \top$
  by *fastforce*

model ($'a$::*countable*) *election1* =
  extends *ringnetwork less* **for** *less* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** ‹<› *50*)
  variables *local1* :: $'a \Rightarrow 'a$ *local1*

initialise *local1* = (λ*a*. (|*elected1*=*False*, *chan1*={}|))
invariants
— if msg is in buffer of x, then all z in the interval between (msg,x) are strictly smaller than msg
valid-interval: ∀ *x msg*. *msg* ∈ *chan1* (*local1 x*) ⟶ (∀ *z* ∈ *collect msg x*. *z* < *msg*)
and
— only top can be elected leader
leader-top: ∀ *i*. *elected1* (*local1 i*) ⟶ *i* = ⊤ **uses** *valid-interval*
events
— add own ID to the next node's buffer
Setup (*x*::'*a*) **do** *local1'* = *add-msg-to-chan1 local1* (*nxt x*) *x*
| *Accept* (*x*::'*a*) (*msg*::'*a*)
— node x received a name msg higher than his own x.
when *msg* ∈ *chan1* (*local1 x*) *x* < *msg*
— forward to next node in ring.
do *local1'* = *add-msg-to-chan1 local1* (*nxt x*) *msg*
| *Elect* (*z*::'*a*)
— node x received its own name.
when *z* ∈ *chan1* (*local1 z*)
do *local1'* = *local1*(*z* := (*local1 z*)(|*elected1* := *True*|))
refines *election0*
via
leader = *elected1* ∘ *local1*
where
Elect y ↠ *election0.Elect y*
proof *model-cases*
case *leader-top*:*Elect*
then show *?case*
using *topEqI collect-self* **by** *simp*
next
case (*refine*:*Elect s a s' y*)
then show *?case*
apply *safe*
apply (*metis collect-self comp-def extremum-strict*)
by (*simp add*: *fun-upd-comp*)
qed ((*rule ext*)?; *clarsimp simp*: *fun-upd-comp*; *fastforce dest*: *collect-nxt-r*)+

4.3 Interface Model

We add local input buffers and output buffers. Here, they are modelled as sets and messages are never removed from the buffers.

4.3.1 Local Buffers Model tr2

record '*a local2* =
leader2 :: *bool* — has this node declared itself a leader?
ibuf2 :: '*a set* — incoming internal buffer of a node
obuf2 :: '*a set* — outgoing internal buffer of a node

abbreviation (*input*) *add-msg-to-ibuf2* **where**
add-msg-to-ibuf2 s x msg ≡ *s*(*x* := *s x* (|*ibuf2* := *insert msg* (*ibuf2* (*s x*))|))

abbreviation (*input*) *add-msg-to-obuf2* **where**
add-msg-to-obuf2 s x msg ≡ *s*(*x* := *s x* (|*obuf2* := *insert msg* (*obuf2* (*s x*))|))

abbreviation (*input*) *add-msg-to-chan2* **where**
add-msg-to-chan2 s x msg ≡ *s*(*x* := *insert msg* (*s x*))

model (*'a*::*countable*, *'ADDR*::*countable*) *election2* =
extends *addressedRingnetwork less top - - addr*
for *top* :: *'a* (‹⊤›)
and *less* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹<› *50*)
and *addr* :: *'a* ⇒ *'ADDR*
variables *local2* :: *'a* ⇒ *'a local2* **and** *chan* :: *'ADDR* ⇒ *'a set*
initialise
local2 = (λ-. (|*leader2*=*False*,*ibuf2*={},*obuf2*={}|)) *chan* = (λ-. {})
invariants
inv-buffers:
∀ *x msg*. (*msg* ∈ *obuf2* (*local2 x*) ⟶ *msg* ∈ *chan* (*addr x*) ∧ *x* < *msg* ∨ *msg* = *x*) ∧
(*msg* ∈ *ibuf2* (*local2 x*) ⟶ *msg* ∈ *chan* (*addr x*))
events
Setup (*x*:*'a*) **do** *local2'* = *add-msg-to-obuf2 local2 x x*
| *Receive* (*x*:*'a*) (*msg*:*'a*)
when *msg* ∈ *chan* (*addr x*)
do *local2'* = *add-msg-to-ibuf2 local2 x msg*
| *Accept* (*x*:*'a*) (*msg*:*'a*)
— node x received a name msg higher than his own x
when *msg* ∈ *ibuf2* (*local2 x*) *x* < *msg*
do *local2'* = *add-msg-to-obuf2 local2 x msg*
| *Elect* (*x*:*'a*)
when *x* ∈ *ibuf2* (*local2 x*)
do *local2'* = *local2*(*x* := *local2 x* (|*leader2* := *True*|))
| *Send* (*x*:*'a*) (*msg*:*'a*) (*a*:*'ADDR*)
when *msg* ∈ *obuf2* (*local2 x*) *a* = *addr* (*nxt x*)
do *chan'* = *add-msg-to-chan2 chan a msg*
refines *election1* ⊤ *ordr nxt less*
via
local1 = (λ*x*. (|*elected1* = *leader2* (*local2 x*), *chan1* = *chan* (*addr x*)|))
where
election2.Send x y - ↠
(*if x* = *y then election1.Setup x*
else election1.Accept x y)
| *election2.Elect x* ↠ *election1.Elect x*
proof *model-cases*
case (*refine*:*Send*:*if-False s a s' x y msg*)
moreover hence $y \in get_{chan_v}$ *s* (*addr x*) ∧ *x* < *y* ∨ *y* = *x* **by** *blast*

ultimately show *?case* **by** *force*
qed (*force intro*!: *ext*)+

thm *election2.refines-election1*

References

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[2] C. Ballarin. *Tutorial to Locales and Locale Interpretation.* https://isabelle.in.tum.de/website-Isabelle2023/dist/Isabelle2023/doc/locales.pdf.

[3] S. Foster, C. Pardillo-Laursen, and F. Zeyda. Optics. *Archive of Formal Proofs*, May 2017. https://isa-afp.org/entries/Optics.html, Formal proof development.

[4] C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P. Müller, M. Clochard, and D. Basin. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.