Circus Solution of ESELSpec for the ESEL System

Kangfeng Ye

April 27, 2016

1 Header

section *ESELHeader* **parents** *circus_toolkit*

This section gives all basic definitions that will be used in all three *Circus* models. And gateway related definitions are only used in the ESEL System 2.

First of all, three constants are defined. MAX_ESEL and MAX_PID stand for maximum number of displays and maximum number of product categories (or, products for short) in the system separately. And constant $MAX\_GATEWAY$ stands for maximum number of gateways.

$$MAX\_ESEL : \mathbb{N}$$
$$MAX\_PID : \mathbb{N}$$

Then all displays and products are identified by a tag plus a unique number which are defined in the free types $ESID$ and PID below where the constructors ES and PD are the tags for displays and products. For an instance, number ten of the display is given $ES\,10$ or $ES(10)$. Similarly, GID gives all identities for gateways.

$$ESID ::= ES\langle\langle 1 \mathinner{..} MAX\_ESEL\rangle\rangle$$
$$PID ::= PD\langle\langle 1 \mathinner{..} MAX\_PID\rangle\rangle$$

The type of product price is defined as an abbreviation to natural numbers $\mathbb{N}$.

$$Price == \mathbb{N}$$

The unit response is defined as a free type with two constants: uok and $ufail$.

$$UStatus ::= uok \mid ufail$$

The response from this program to the environment is a set of product identities of which the price is not updated successfully due to 1) no linked ESEL ID to the product or 2) failed update to its linked ESEL. The first reason is given the status constant NA and the second is provided the constructor $fail\langle\langle ESID\rangle\rangle$.

$$FStatus ::= fail\langle\langle ESID\rangle\rangle \mid NA$$

Two channels are provided to update the map from ESEL ID to product ID. *updateallmap* will clear all stored map and use the input map as new map, while *updatemap* just updates a partial map. In this map, one ESEL can be linked to up to one product. However, one product may associate with multiple ESELs.

channel $updateallmap : ESID \pfun PID$
channel $updatemap : ESID \pfun PID$

Similarly, two channels are provided to update the price information. *updateallprice* will clear all price information and use the input price information as new price, while *updateprice* just updates price partially.

channel $updateallprice : PID \pfun Price$
channel $updateprice : PID \pfun Price$

The *update* channel gives a signal to the program to start update process.

channel $update$

The *failures* channel returns all failed products and related error reasons after update. Since one product may associate with multiple displays, the return status is a power set of *FStatus* to denote which specific displays that the product links are updated unsuccessfully. But it is worth noting that *NA* and *fail* must not occur in a product's return set at the same time because they can not be both no associate display and associate display update fail.

channel $failures : PID \pfun \mathbb{P}\ FStatus$

The internal *resp* event is used to collect update responses from all displays and *terminate* event is for completing the collection.

channel $resp : PID \times FStatus$
channel $terminate$
channelset $RespInterface == \{\!|\ resp, terminate\ |\!\}$

This *uupdate* event is to update one ESEL to the specific price, and *ures* for update response from this ESEL. And *udisplay* is used to synchronise the show of price on all ESELs at the same time and *finishdisplay* is used to wait for display completion of all ESELs. That is the similar case for *uinit* and *ufinishinit* that are for initialisation synchronisation.

channel $uupdate : ESID \times Price$
channel $ures : ESID \times UStatus$
channel $uinit, finishuinit$
channel $udisplay, finishudisplay$

And *display* is used to synchronise the show of price on all gateways (or ESELs) at the same time and *finishdisplay* is used to wait for display completion of all gateways (or ESELs). That is the similar case for *init* and *finishinit* that are for initialisation synchronisation.

channel $init, finishinit$
channel $display, finishdisplay$

The channels below are for communication between the ESEL system and displays. The *write* event writes price to a display, and the *read* event reads price from the display. *ondisplay* turns on the related display and *offdisplay* turns off it conversely.

channel $write : ESID \times Price$
channel $read : ESID \times Price$
channel $ondisplay : ESID$
channel $offdisplay : ESID$

2 ESELSpec

section $ESELSpec$ **parents** $ESELHeader$

Controller Process The process for overall control of the system, named *Controller*, is defined as an explicitly defined process.

$$\textbf{process}\ \mathit{Controller} \mathrel{\widehat=} \textbf{begin}$$

Controller has three state components: *pumap* for mapping from displays to products, *ppmap* for mapping from products to their price, and *response* for the response of one update to the environment.

$$\textbf{state}\ \mathit{State} == [\, \mathit{pumap} : \mathit{ESID} \pfun \mathit{PID}\ ;\ \mathit{ppmap} : \mathit{PID} \pfun \mathit{Price};\ \mathit{response} : \mathit{PID} \pfun (\mathbf{P}\ \mathit{FStatus})\,]$$

Initially, these three state components all are empty.

$$\mathit{Init} == [\, (\mathit{State})' \mid \mathit{pumap}' = \varnothing \land \mathit{ppmap}' = \varnothing \land \mathit{response}' = \varnothing\,]$$

The *UpdateMap* schema updates part of the displays to products map according to the input map, while the *UpdateAllMap* schema discards stored map and uses new input map as *pumap*.

$$\begin{aligned}&\mathit{UpdateMap} == [\, \Delta \mathit{State}\ ;\ \mathit{map?} : \mathit{ESID} \pfun \mathit{PID} \mid \\ &\quad \mathit{pumap}' = \mathit{pumap} \oplus \mathit{map?} \land \mathit{ppmap}' = \mathit{ppmap} \land \mathit{response}' = \mathit{response}\,] \\ &\mathit{UpdateAllMap} == [\, \Delta \mathit{State}\ ;\ \mathit{map?} : \mathit{ESID} \pfun \mathit{PID} \mid \\ &\quad \mathit{pumap}' = \mathit{map?} \land \mathit{ppmap}' = \mathit{ppmap} \land \mathit{response}' = \mathit{response}\,]\end{aligned}$$

The *NewPrice* updates part of price information stored, while the *AllNewPrice* discards all price information stored and uses input price as *ppmap*.

$$\begin{aligned}&\mathit{NewPrice} == [\, \Delta \mathit{State}\ ;\ \mathit{price?} : \mathit{PID} \pfun \mathit{Price} \mid \\ &\quad \mathit{ppmap}' = \mathit{ppmap} \oplus \mathit{price?} \land \mathit{pumap}' = \mathit{pumap} \land \mathit{response}' = \mathit{response}\,] \\ &\mathit{AllNewPrice} == [\, \Delta \mathit{State}\ ;\ \mathit{price?} : \mathit{PID} \pfun \mathit{Price} \mid \\ &\quad \mathit{ppmap}' = \mathit{price?} \land \mathit{pumap}' = \mathit{pumap} \land \mathit{response}' = \mathit{response}\,]\end{aligned}$$

AUpdatemap is an action defined to update displays to products map: either partial update by *updatemap* event or complete update by *updateallmap* event.

$$\begin{aligned}&\mathit{AUpdatemap} \mathrel{\widehat=} \mathit{updatemap?map} \rightarrow \big(\mathit{UpdateMap}\big) \\ &\quad \Box\ \mathit{updateallmap?map} \rightarrow \big(\mathit{UpdateAllMap}\big)\end{aligned}$$

Similarly, *ANewPrice* is an action defined to update products to price map: either partial update by *updateprice* event or complete update by *updateallprice* event.

$$\begin{aligned}&\mathit{ANewPrice} \mathrel{\widehat=} \mathit{updateprice?price} \rightarrow \big(\mathit{NewPrice}\big) \\ &\quad \Box\ \mathit{updateallprice?price} \rightarrow \big(\mathit{AllNewPrice}\big)\end{aligned}$$

A parameterised action, *AUpdateUnitPrice*, is given to update the price (specified by the formal *pid* parameter) to a display (given by the formal *uid* parameter). It sends the price to the specified display by the *write* event, and then read back the price from the display by the *read* event. If the write price matchs with the read price, then the update is successful. Otherwise, it fails (*ufail*) and sends the result to response collection action *CollectResp* below, then terminates.

$$\begin{aligned}&\mathit{AUpdateUnitPrice} \mathrel{\widehat=} \mathit{uid} : \mathit{ESID}\ ;\ \mathit{pid} : \mathit{PID} \bullet \\ &\quad \mathit{write.uid.}(\mathit{ppmap}\ \mathit{pid}) \rightarrow \mathit{read.uid?}y \rightarrow \\ &\quad \big((y = (\mathit{ppmap}\ \mathit{pid})\big)\ \&\ \textbf{Skip} \\ &\quad \Box\ \big(y \neq (\mathit{ppmap}\ \mathit{pid})\big)\ \&\ \mathit{resp.pid.}(\mathit{fail}\ \mathit{uid}) \rightarrow \textbf{Skip})\end{aligned}$$

The parameterised action *AUpdateProductUnits* aims to update one product's price specified by the formal *pid* parameter in case the product has associated displays. Since one product may have more than one associated displays, this action updates the product's price to all associated displays. Furthermore, the update to each display is independent. Therefore, they are combined together into an interleave. It is worth noting that each *AUpdateUnitPrice* action will not update state or local variables and thus its name set is empty.

$$AUpdateProductUnits \mathrel{\widehat=} pid : PID \bullet \\ \left(\left|\left|\left| uid : (\mathrm{dom}\,(pumap \rhd \{pid\})) \right|\right|\right]\varnothing\left[\right|\right| \bullet AUpdateUnitPrice(uid, pid))$$

Otherwise, if the product has not been allocated the corresponding displays, it sends back a response to state this error *NA*. The behaviour is defined in the *AUpdateNoUnit* action.

$$AUpdateNoUnit \mathrel{\widehat=} pid : PID \bullet resp.pid.NA \rightarrow \mathbf{Skip}$$

The behaviour of the price update for a product given in *pid* is the update of product either with associated displays, guarded *AUpdateProductUnits*, or without associated displays, guarded *AUpdateNoUnit*.

$$AUpdateProduct \mathrel{\widehat=} pid : PID \bullet \\ \quad \big(pid \in \mathrm{ran}\ pumap\big) \mathbin{\&} AUpdateProductUnits(pid) \\ \quad \Box\ \big(pid \notin \mathrm{ran}\ pumap\big) \mathbin{\&} AUpdateNoUnit(pid)$$

Then the update of all products is given in the action *AUpdateProducts*. At first, it is an interleave of all updates of the products which have associated price, then follows a *terminate* event to finish the update.

$$AUpdateProducts \mathrel{\widehat=} ((\left|\left|\left| pid : (\mathrm{dom}\ ppmap) \right|\right|\right]\varnothing\left[\right|\right| \bullet AUpdateProduct(pid)) \\ \quad ; terminate \rightarrow \mathbf{Skip})$$

The *AddOneFailure* schema adds one failure (either update failure or no associate failure) for a product to the state component *response*.

$$AddOneFailure == [\,\Delta State\ ;\ pid? : PID\ ;\ fst? : FStatus \mid \\ \quad (pid? \in \mathrm{dom}\ response \Rightarrow \\ \quad\quad response' = response \oplus \{pid? \mapsto (response(pid?) \cup \{fst?\})\}) \land \\ \quad (pid? \notin \mathrm{dom}\ response \Rightarrow \\ \quad\quad response' = response \cup \{pid? \mapsto \{fst?\}\}) \land \\ \quad ppmap' = ppmap \land pumap' = pumap\,]$$

The *CollectResp* action is to collect responses from all units and write them into the *response* variable by *AddOneFailure* schema expression. It recursively waits for the response from the units, or terminates if required.

$$CollectResp \mathrel{\widehat=} \mu X \bullet \\ \quad ((resp?pid?fst \rightarrow \big(AddOneFailure\big)\ ;\ X) \Box\ terminate \rightarrow \mathbf{Skip})$$

Then update of all products and response collection behaviours are put together into *AUpdateResp* action. It is a parallel composition of *AUpdateProducts* and *CollectResp* actions and they are synchronised with *resp* and *terminate* events. Furthermore, the

left action *AUpdateProducts* will not update state variables (its name set is empty set) while the right action *CollectResp* will update *response* (its name set has only one variable *response*). Finally, these internal events are hidden.

$$
\begin{array}{l}
AUpdateResp \mathrel{\widehat=} \\
\quad (AUpdateProducts \;[\![\, \varnothing \mid RespInterface \mid \{response\} \,]\!]\; CollectResp) \\
\quad \setminus RespInterface
\end{array}
$$

All displays will synchronise on *display* event to show the price at the same time, which is defined in *ADisplay*. Whether a display should be turned on or off is decided based on the logic below.

- If the display is not mapped to a product, then turn it off.
- Otherwise, if the display linked product is not to be updated, then turn it off.
- Otherwise, if the display has been written the price successfully, then turn it on.
- Otherwise, then turn it off.

$$
\begin{array}{l}
ADisplay \mathrel{\widehat=} \\
\quad ([\![\{\!|\, display \,|\!\}]\!]\; uid : ESID \bullet [\![\varnothing]\!]\; display \rightarrow (\\
\qquad \textbf{if}\;\; uid \notin \mathrm{dom}\; pumap \longrightarrow offdisplay.uid \rightarrow \textbf{Skip} \\
\qquad [\!]\; uid \in \mathrm{dom}\; pumap \longrightarrow \\
\qquad\quad \textbf{if}\;\; pumap(uid) \notin \mathrm{dom}\; ppmap \longrightarrow offdisplay.uid \rightarrow \textbf{Skip} \\
\qquad\quad [\!]\; pumap(uid) \in \mathrm{dom}\; ppmap \longrightarrow \\
\qquad\qquad \textbf{if}\;\; pumap(uid) \notin \mathrm{dom}\; response \longrightarrow \\
\qquad\qquad\quad ondisplay.uid \rightarrow \textbf{Skip} \\
\qquad\qquad [\!]\; pumap(uid) \in \mathrm{dom}\; response \longrightarrow \\
\qquad\qquad\quad \textbf{if}\;\; (fail\; uid) \notin response(pumap(uid)) \longrightarrow \\
\qquad\qquad\qquad ondisplay.uid \rightarrow \textbf{Skip} \\
\qquad\qquad\quad [\!]\; (fail\; uid) \in response(pumap(uid)) \longrightarrow \\
\qquad\qquad\qquad offdisplay.uid \rightarrow \textbf{Skip} \\
\qquad\qquad\quad \textbf{fi} \\
\qquad\qquad \textbf{fi} \\
\qquad\quad \textbf{fi} \\
\qquad \textbf{fi} \\
\quad)) \setminus \{\!|\, display \,|\!\}
\end{array}
$$

The overall price update action is given in *AUpdatePrice*, which accepts a *update* event from its environment, then clears *response*, updates the price, sends *display* event to make all ESELs show their price at the same time, then feeds back the response to the environment.

$$
\begin{array}{l}
AUpdatePrice \mathrel{\widehat=} update \rightarrow response := \varnothing; \\
\quad AUpdateResp \;;\; ADisplay \;;\; failures.response \rightarrow \textbf{Skip}
\end{array}
$$

Initially, state components are cleared and all displays are turned off.

$$
AInit \mathrel{\widehat=} \big(Init\big) \;;\; (|||\; u : ESID \;[\![\varnothing]\!] \bullet offdisplay.u \rightarrow \textbf{Skip})
$$

The overall behaviour of the *Controller* process is given by its main action. It initializes at first, then repeatedly provides display map update, price map, or price update to its environment.

$$\bullet\ AInit\ ;\ (\mu X \bullet (AUpdatemap \Box ANewPrice \Box AUpdatePrice)\ ;\ X)$$

end

System The ESEL Specification is simply the *Controller* process.

process $ESELSpec \mathrel{\widehat{=}} Controller$