Circus Solution of ESELSystem1 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 ESEL System 1

section $ESELSystem1$ **parents** $ESELHeader$

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

$$\textbf{process}\ \mathit{Controller1} \mathrel{\widehat=} \textbf{begin}$$

Controller has three state components: *pumap* for mapping from ESELs 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 ESELs to products map according to the input map, while the *UpdateAllMap* schema discards all 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 ESELs 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 an ESEL (given by the formal *uid* parameter). It sends the price to the specified ESEL by the *uupdate* event, and then waits for the response from the ESEL. If the return status is not successful (*ufail*), it sends the result to response collection action *CollectResp* below, then terminates. Otherwise, it terminates immediately.

$$\begin{aligned}&\mathit{AUpdateUnitPrice} \mathrel{\widehat=} \mathit{uid} : \mathit{ESID}\ ;\ \mathit{pid} : \mathit{PID} \bullet \\ &\quad \mathit{uupdate.uid.(ppmap\ pid)} \rightarrow \mathit{ures.uid?rst} \rightarrow \\ &\quad \big((\mathit{rst} = \mathit{ufail}\big)\ \&\ \mathit{resp.pid.(fail\ uid)} \rightarrow \mathbf{Skip} \\ &\quad \Box\ \big(\mathit{rst} = \mathit{uok}\big)\ \&\ \mathbf{Skip})\end{aligned}$$

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

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

Otherwise, if the product has not been allocated the corresponding ESELs, 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 ESELs, guarded *AUpdateProductUnits*, or without associated ESELs, guarded *AUpdateNoUnit*.

$$\begin{aligned} AUpdateProduct \mathrel{\widehat=} pid : PID \bullet \\ (pid \in \mathrm{ran}\ pumap) \ \&\ AUpdateProductUnits(pid) \\ \Box\ (pid \notin \mathrm{ran}\ pumap) \ \&\ AUpdateNoUnit(pid) \end{aligned}$$

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.

$$\begin{aligned} AUpdateProducts \mathrel{\widehat=} ((|||\ pid : (\mathrm{dom}\ ppmap) \ [\![\varnothing]\!] \bullet AUpdateProduct(pid)) \\ ; terminate \rightarrow \mathbf{Skip}) \end{aligned}$$

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

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

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

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. Finally, these internal events are hidden.

$$
\begin{array}{l}
AUpdateResp \mathrel{\widehat=} \\
\quad (AUpdateProducts \llbracket \varnothing \mid RespInterface \mid \{response\} \rrbracket CollectResp) \\
\quad \setminus RespInterface
\end{array}
$$

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.

$$
\begin{array}{l}
AUpdatePrice \mathrel{\widehat=} update \rightarrow response := \varnothing; \\
\quad AUpdateResp \,;\, display \rightarrow finishdisplay \rightarrow failures.response \rightarrow \mathbf{Skip}
\end{array}
$$

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

$$
\begin{array}{l}
\bullet \left(Init\right) ; init \rightarrow \mathbf{Skip}; \\
\quad (\mu X \bullet (AUpdatemap \,\Box\, ANewPrice \,\Box\, AUpdatePrice) \,;\, X)
\end{array}
$$

end

ESEL Process Each ESEL is defined as a parameterised process with the formal parameter—ESEL ID.

$$\mathbf{process}\ ESEL1 \mathrel{\widehat=} eid : ESID \bullet \mathbf{begin}$$

The process has two state components: *price* for the price to display, and *status* for the status of ESEL.

$$\mathbf{state}\ State == [\, price : Price \,;\, status : UStatus \,]$$

Initially, the price is equal to 0 and the status is *uok*.

$$Init == [\, (State)' \mid price' = 0 \land status' = uok \,]$$

The *Update* action provides its environment (*Controller*) the update of price for the associated product. It accepts the *uupdate* event with the price, then writes the price to *price*. After that, it writes the price to the display unit, and reads back the value to compare with the original price. If it is equal, it sends back status *uok* by the *ures* event. Otherwise, it sends back status *ufail*. Accordingly, *status* is updated.

$$
\begin{array}{l}
Update \mathrel{\widehat=} uupdate.eid?x \rightarrow price := x \,;\, write.eid.price \rightarrow read.eid?y \\
\quad \rightarrow ((y = price) \,\&\, ures.eid.uok \rightarrow status := uok \\
\qquad \Box\, (y \neq price) \,\&\, ures.eid.ufail \rightarrow status := ufail)
\end{array}
$$

The *Display* action accepts the *display* event. If the status is *uok*, then the associated display is turned on. Otherwise, the display is turned off.

$$
\begin{array}{l}
Display \mathrel{\widehat=} display \rightarrow (\\
\quad \big(status = uok\big) \,\&\, ondisplay.eid \rightarrow \mathbf{Skip} \\
\quad \Box \, \big(status = ufail\big) \,\&\, offdisplay.eid \rightarrow \mathbf{Skip}) \\
\quad ; finishdisplay \rightarrow \mathbf{Skip}
\end{array}
$$

$$NotUpdateDisplay \mathrel{\widehat=} display \rightarrow offdisplay.eid \rightarrow finishdisplay \rightarrow \mathbf{Skip}$$

The initial behaviour of the process is given in the action *AInit* which initialises the state at first, and then turns off the display.

$$AInit \mathrel{\widehat=} \big(Init\big) \,;\, offdisplay.eid \rightarrow init \rightarrow \mathbf{Skip}$$

The overall behaviour of the process is given by its main action. It specifies that after initialisation the process repeatedly provides update or display to its environment.

$$\bullet \; AInit \,;\, (\mu X \bullet ((Update \,;\, Display) \,\Box\, NotUpdateDisplay) \,;\, X)$$

end

The behaviour of all ESELs together is formed by iterated parallel composition of *ESEL* process. The only communication between them is to display at the same time.

$$
\begin{array}{l}
\mathbf{channelset} \;\; InterESELInterface1 == \{\!| \, init, display, finishdisplay \, |\!\} \\
\mathbf{process} \;\; ESELS1 \mathrel{\widehat=} \Big|\Big| \; eid : ESID \; [\![\, InterESELInterface1]\!] \bullet ESEL1(eid)
\end{array}
$$

System Finally, the whole system is defined as the parallel between the *Controller* process and the *ESELS* process. They synchronise with the *uupdate*, *ures*, *display*, and *finishdisplay* events.

$$
\begin{array}{l}
\mathbf{channelset} \;\; ESELInterface1 == \{\!| \, uupdate, ures, init, display, finishdisplay \, |\!\} \\
\mathbf{process} \;\; ESELSystem1 \mathrel{\widehat=} (Controller1 \; [\![\, ESELInterface1 \,]\!] \; ESELS1) \setminus ESELInterface1
\end{array}
$$