Chapter 5

Case Study - Production Cell

5.1 Introduction

In this chapter, we present a real life control system case-study modelled in Event-B, to be used as an example for our formal featured-oriented reuse framework. We discuss how we can decompose and compose Event-B models in different ways and at different abstraction levels to explore product line reuse approach in Event-B. The case-study has been useful in exploring reusability, defining the notion of features for feature-based modelling in Event-B and to figure out the tooling requirements for product line modelling using Rodin.

5.2 Production Cell

The production cell (PC) [100] is an industrial metal processing plant which falls under the category of critical systems. This is an example of a reactive system which has been modelled in more than 30 formalisms [108, 110, 70, 101, 133], including the B formal method which is a predecessor of the Event-B language [121]. The production cell has also been specified at an abstract level using the RAISE [107] formal method and a stepwise refinement approach has been used to generate the implementation. This was supported by proofs to verify the safety properties of the system [98]. The production cell has various components which must be controlled in real time while maintaining safety properties to avoid any loss to human beings and damage to the equipment.

The production cell processes metal blanks which are routed to a press for forging, then routed away from it after processing. Figure 5.1 shows top view of the production cell plant. Metal blanks enter into the system through the feed belt and are dropped on to the elevating-rotary table when the table is empty and in the loading position. The table has two positions. The first position is for receiving blanks from the feed belt when

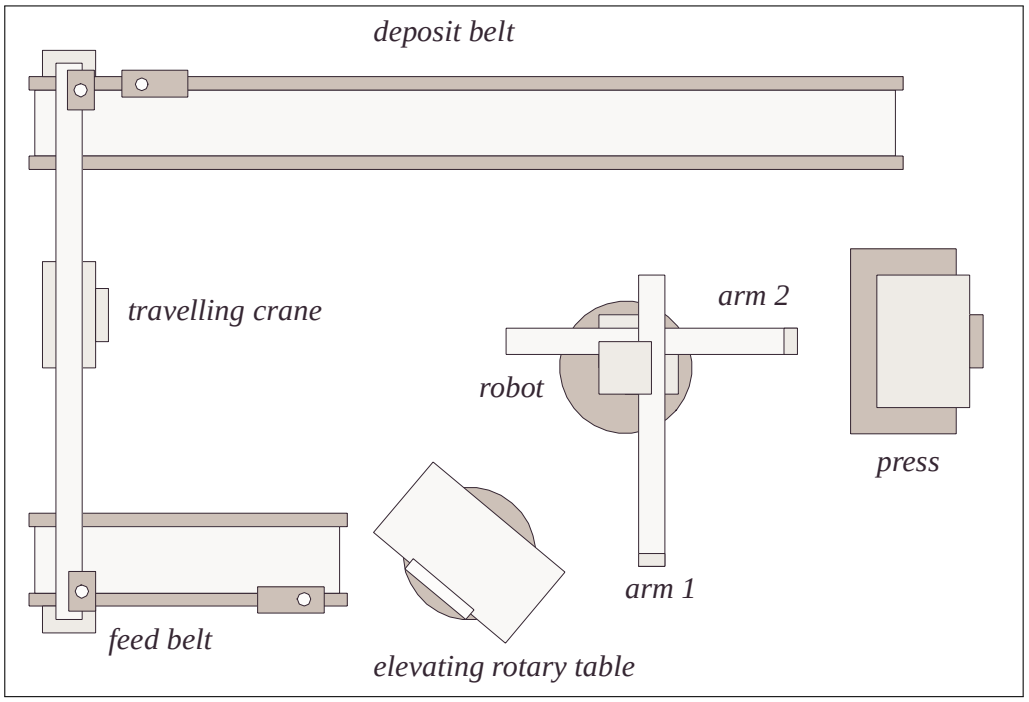


Figure 5.1: Top view of production cell plant [100]

the table is neither elevated nor rotated. The table, once loaded, elevates and rotates to the second position so that the first robot arm can pick up blanks as the robot arms cannot move vertically. The robot rotates around its own axis. It has two arms which can extend and retract horizontally and independently of each other. The first arm picks a blank from the table and drops it on the press. The press has three positions as the robot arms are at different planes to avoid collision. The first robot arm drops the blank when the press is in the middle position. The press forges the blank at the high position and moves to the lower position from where the blank is picked up by the second robot arm. The blank is then dropped onto the deposit belt. There is a flaw in the description of the production cell presented in [100] which does not define the mechanism for finding out the blanks that are not properly forged and fed back into the system for reprocessing. In reality, forged blanks should leave the system after being transported on to the deposit belt. So, we assume that the forged blanks on the deposit belt are removed from the system by some external mechanism, which is not included in our modelling, and the ones that are not forged properly travel towards the end of the deposit belt. A moving crane then picks these unforged blanks from the deposit belt and brings them back to the feed belt for reprocessing. This completes one cycle of the production cell. There are 13 actuators and 14 sensors for controlling and monitoring various components of the plant. The detailed description of the production cell and various requirements are narrated by [100]. The requirements specification subset we used for modelling the production cell is given in Appendix A.

5.3 Roadmap

We discuss Event-B development of the production cell in three different ways. We start with the physical component-based modelling of PC in Section 5.4, where we model and refine PC system as integral model and then decompose it into various physical components of the plant. We use both types of Event-B decomposition techniques, i.e., shared-variable decomposition (SVD) and shared-event decomposition (SED); and discuss the workload of modelling the system to be decomposed later using the two styles. We then discuss controller-based (functional) modelling of PC in Section 5.5, where we generalise PC requirements to model generic controllers of the plant or functional features that could then be specialised for modelling various components of the PC. This approach shows more reuse opportunity compared to the components-based PC. We then discuss domain-specific modelling approach based on static variability in Section 5.6, where we can model variants of PC by switching the contexts. This allowed us to use different methods of modelling the same system in Event-B and analysing our approach to feature-based modelling using existing tools and techniques in Event-B. This also enabled us to explore the amount of reuse that can be achieved using different modelling styles.

Refinement is a process of introducing more details in each step from abstract specification to the concrete one which is closer to implementation. By using refinement, we can model a system in multiple steps and deal with small number of requirements in each step rather than modelling the entire system in a single model. This becomes very difficult to manage and prove when modelling everything in a single refinement step. Hence, step-wise refinement is the approach we have used to model the production cell system.

5.4 PC Component-based

5.4.1 Development Structure

In the physical component-based modelling approach, we started with an abstract model of the production cell and refined it in a number of steps. The first four levels are horizontal refinements where we introduced further requirements in each refinement step and the later ones are the vertical refinements after we have decomposed the integral model into various physical components using SVD.

We have also modelled the PC in a way that we could decompose it using SED. This required us to prepare the model so that there are no shared variables and different

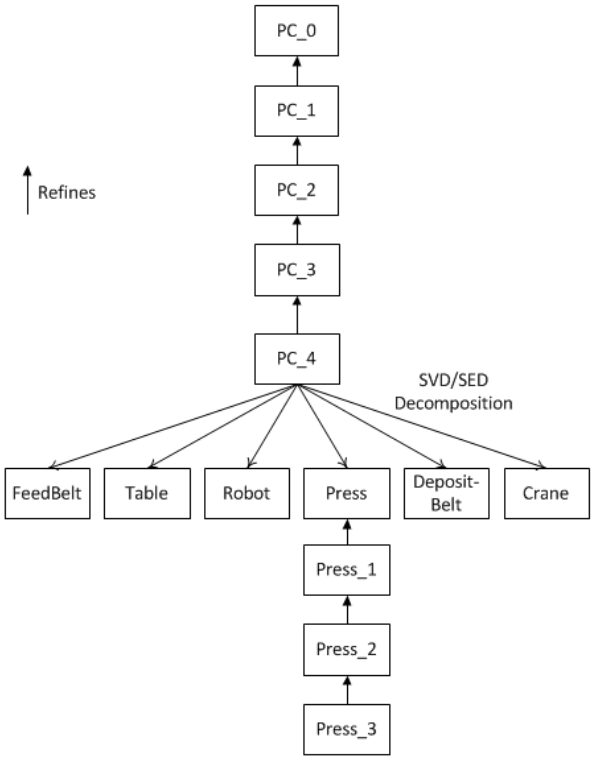


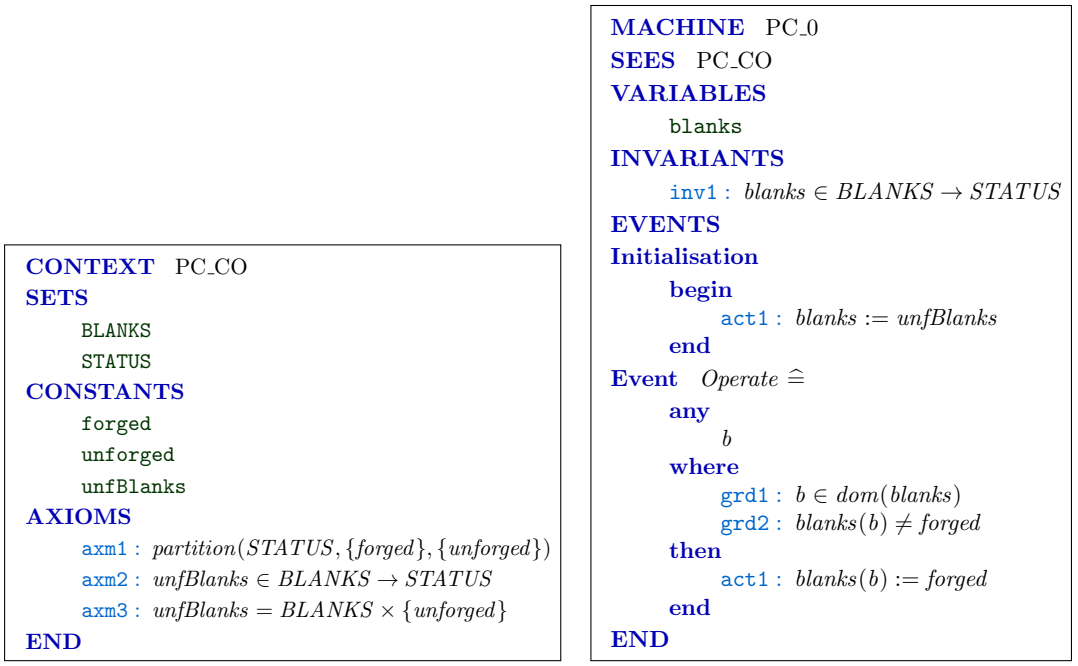
Figure 5.2: Component-based PC modelling

components communicate via shared-events. This means that we had to perform vertical refinements earlier on, unlike the development which used SVD where we only do horizontal refinement before the decomposition.

Each of these components can then be refined separately while maintaing the constraints of the decomposition technique. We only refined one component (i.e., *Press*, resulting from SVD of PC) further as the other components can be refined in the similar manner. During the vertical refinements, actuators and sensors were introduced to model the production cell closer to implementation. Figure 5.2 shows Event-B refinement architecture for component-based PC modelling and the detail of each refinement step is discussed below.

5.4.2 PC Abstract Model (PC_0)

The basic model for the production cell without much detail is specified in the abstract model. Here we specify the whole production cell cycle described above in one event *Operate* which models the processing of blanks from `forged` to `unforged` state through the variable `blanks`. The Event-B model is shown in Figure 5.3 where *PC_CO* is the abstract context (Figure 5.3(a)) seen by the machine (Figure 5.3(b)). The machine has a variable `blanks` and an invariant describing the variable type, i.e., every blank must have a status. All the blanks are initialised to unforged state in the *Initialisation* event and the blanks are forged in the *Operate* event. The abstract context contains



(a) Context (b) Machine

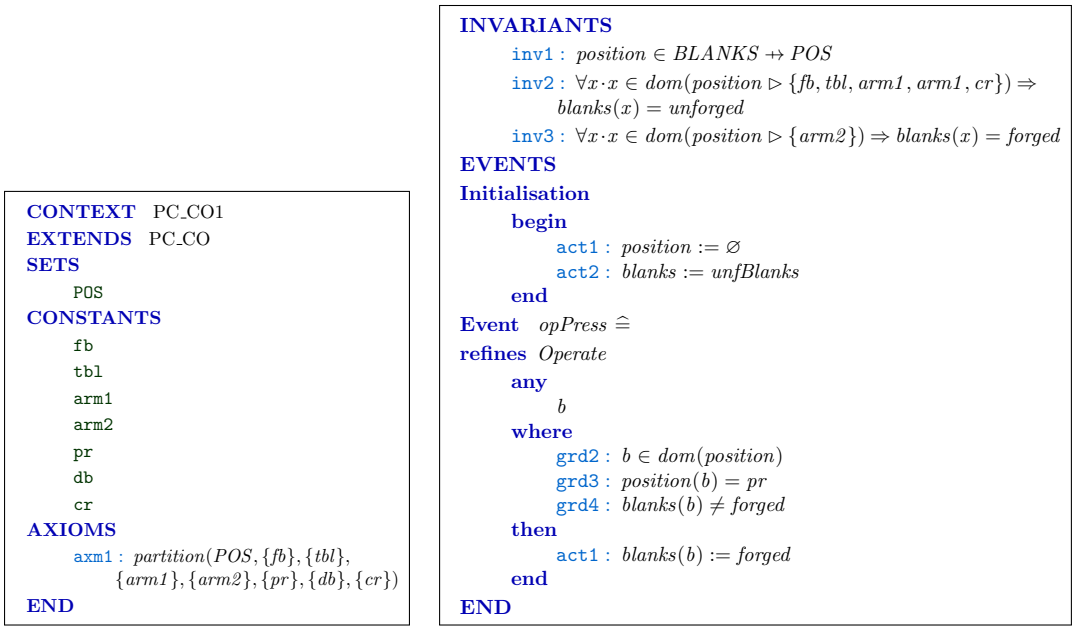
Figure 5.3: PC abstract model

the sets `BLANKS` and `STATUS`. The set `STATUS` has two elements defined as constants, i.e., `forged` and `unforged`. The constant `unfBlanks` sets the status of all the blanks to unforged as used in the *`Initialisation`* event of the machine.

Here we model requirement 1 of the PC requirement specification given in Appendix A.

5.4.3 PC First Refinement (PC_1)

In the first refinement, we look a bit further and introduce system components such as *feed belt, robot arms, press, deposit belt, crane* and introduce position for each blank. So at any time, we know where a particular blank is positioned and its status, i.e., forged or unforged. A part of the Event-B model after first refinement is given in Figure 5.4. The machine on the right refines PC_0 and sees the context PC_CO1 shown on the left which extends the context PC_CO of Figure 5.3(a). The context PC_CO1 defines an enumerated set `POS` which contains the positions (`axm1`) that blanks can have during the entire cycle of the production cell system. The machine contains a variable for the position which is a partial function from `BLANKS` to `POS` (`inv1`), so a blank can be positioned at any one component of the system at at any time. The invariant `inv2` specifies that the blanks on the feed belt, table, arm1 and crane should be unforged and those on arm2 must be forged as stated in the invariant `inv3`. The deposit belt



(a) Extended context

(b) PC_1 Refined partial machine

Figure 5.4: Partial model of PC first refinement

receives forged blanks from arm2 which are removed from the system where as the status of improperly forged blanks is changed to unforged through some external mechanism as stated in problem description earlier. Several new events were introduced for adding blanks to the feed belt (*loadFeedBelt*), loading the table (*loadTable*), dropping blanks on the press (*loadPress*), loading the robot arms (*loadArm1, loadArm2*), dropping blanks on the deposit belt (*loadDepositBelt*), removing forged blanks from the system (*removeBlanks*) and changing status of improperly forged blanks present on the deposit belt to unforged (*unforgeBlanks*). The abstract event *Operate* is refined by *opPress* event which actually processes the blanks. Below is the *loadFeedBelt* event for adding blanks into the system for processing. The guard `grd3` means adding a new blank which is not already in the plant or a blank already in the plant coming from the crane that was not forged properly.

Event *loadFeedBelt* ≙
any
b
where
grd1 : $b \in BLANKS$
grd2 : $blanks(b) \neq forged$
grd3 : $b \notin dom(position) \lor position(b) = cr$
then
act1 : $position := position \lhd\!\!+ \{b \mapsto fb\}$
end

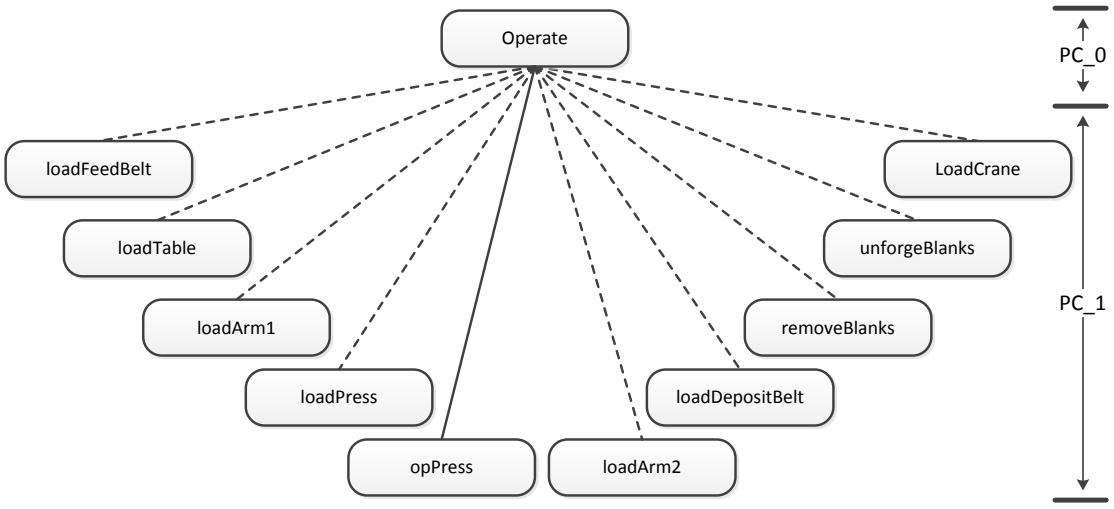


Figure 5.5: Atomicity refinement of PC_0 event in PC_1

Figure 5.5 shows the atomicity refinement of *Operate* event in this level of refinement. The dotted lines show new events introduced in the refinement whereas the solid line shows refined event. The order in which events can take place is read from left to right, as provided in the notation for atomicity decomposition diagram [50].

This refinement models requirements 2, 6, 13, 21, 28 and 35 of the PC requirement specification.

5.4.4 PC Second Refinement (PC_2)

In the second refinement, we further refined the operations taking place at different components of the production cell and introduced positions/states for components such as the *table, robot* and *press*. The context at this level of refinement defines the enumerated sets; `PRESSPOS` for press positions (i.e., low, mid, high) and `RBTPOS` for three robot positions (i.e., pos1, pos2, pos3). Following are some of the invariants at this refinement level:

$$\begin{aligned}
&\texttt{inv5}: \ tblElevated = FALSE \Rightarrow tblRotated = FALSE \\
&\texttt{inv6}: \ tblRotated = TRUE \Rightarrow tblElevated = TRUE \\
&\texttt{inv7}: \ position \rhd \{fb, db\} \in BLANKS \rightarrowtail POS
\end{aligned}$$

The invariants (`inv5` and `inv6`) control the correct positions of the table at any time which should only allow two of the possible four positions as described earlier. The invariant '`Inv7`' makes sure that the components other than feed belt and deposit belt should not have more than one blank on them. The operations defined in the first refinement are refined further to include more details, e.g., the event *loadTable* of PC_1 is refined by the events *loadTable, moveTableUp, moveTableDown, rotateTableFwd*

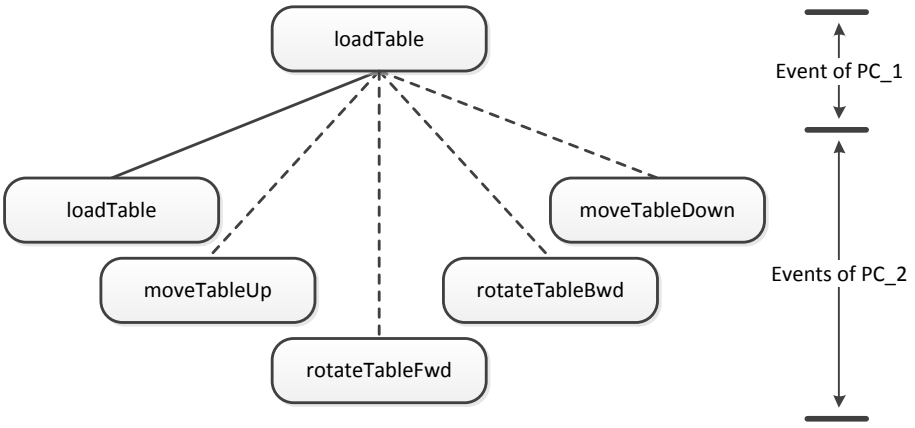


Figure 5.6: Atomicity refinement of *loadTable* event in PC_2

and *rotateTableBwd* to complete the functionality of the table as shown in Figure 5.6. Below is the *loadFeedBelt* event at this refinement level. The guard grd5 ensures that a blank is only added to the feed belt if it has not already reached its capacity.

Event *loadFeedBelt* $\widehat{=}$
refines *loadFeedBelt*
any
b
where
grd1 : $b \in BLANKS$
grd2 : $blanks(b) \neq forged$
grd3 : $b \notin dom(position) \lor position(b) = cr$
grd4 : $finite(dom(position \rhd \{fb\}))$
grd5 : $card(dom(position \rhd \{fb\})) < fbMax$
then
act1 : $position := position \lhd\!\!+ \{b \mapsto fb\}$
end

This refinement models requirements 5, 7, 8, 9, 10, 11, 16, 22, 23, 24, 25, 26, 27, 29 and 33 of PC requirement specification.

5.4.5 PC Third Refinement (PC_3)

In third refinement, we introduced control functionality of the robot arms and movement of the belts for the blanks to be delivered to the next stage. We introduced sets ARMSTATUS for recording the status of robot arms (i.e., extended, retracted) and BELTSTATUS to record whether a belt is running or stopped. Figure 5.7 shows some invariants for the machine at this refinement level. We also introduced some safety requirements for the components here. For example, safety requirements for the movement of robot arms to avoid collision include:

inv5 : $arm1State = extended \Rightarrow (robotPos = pos1 \lor robotPos = pos3)$
inv6 : $arm2State = extended \Rightarrow (robotPos = pos2 \lor robotPos = pos3)$
inv9 : $fbStatus = running \Rightarrow ((blankOnFbEnd = FALSE) \lor (tblElevated = FALSE \land tblRotated = FALSE \land tbl \notin ran(position)))$
inv10 : $dbStatus = running \Rightarrow ((blankOnDbEnd = FALSE) \land (card(dom(position \rhd \{db\})) > 0))$

Figure 5.7: Invariants of PC_3 for safety requirements of production cell

- *Arm1* must not extend at all when the robot is in position 2 (`inv5`).
- Similarly, *Arm2* must not be extended at all when the robot is in position 1 (`inv6`).

The invariant '`inv9`' specifies that the feed belt should only move to make a blank available at its end or if it already has a blank and the table is in a position to receive the blank. The invariant '`inv10`' ensures that the deposit belt should only move to make a blank available at its end to be picked up by the crane.

The ***moveFeedBelt*** event is shown below which moves the belt if it has a blank on it (`grd2`). It only moves the belt if there is no blank at the end of the belt to be dropped on to the table or if the table is vacant and in a position to receive a blank (`grd2`). By moving the feed belt, it makes a blank available on its end to be delivered to the next component.

Event $moveFeedBelt \mathrel{\widehat{=}}$
  when
    grd1 : $finite(dom(position \rhd \{fb\}))$
    grd2 : $card(dom(position \rhd \{fb\})) > 0$
    grd3 : $fbStatus = stopped$
    grd4 : $blankOnFbEnd = FALSE \lor (tblElevated = FALSE \land tblRotated = FALSE \land tbl \notin ran(position))$
  then
    act1 : $fbStatus := running$
  end

The ***loadCrane*** event below allows the crane to pick a blank from the deposit belt and also serves as unloading of the belt.

Event *loadCrane* $\widehat{=}$
refines *loadCrane*
any
b
where
grd1 : $b \in BLANKS$
grd2 : $b \in dom(position)$
grd3 : $position(b) = db$
grd4 : $blanks(b) \neq forged$
grd5 : $cr \notin ran(position)$
grd6 : $blankOnDbEnd = TRUE$
then
act1 : $position(b) := cr$
act2 : $blankOnDbEnd := FALSE$
end

This refinement models requirements 4, 14, 19, 20, 31 and 32 of the PC requirement specification.

5.4.6 PC Fourth Refinement (PC_4)

This is a small refinement step where we introduced further requirements for the travelling crane which includes controlling its movement between the two belts and loading/unloading functionality for blanks. A variable `cranePos` is introduced for the position of crane which can be either at the deposit belt of or at the feed belt. Event *loadFeedBelt* of PC_3 was refined by two events (as shown in Figure 5.8), i.e., *loadFeedBelt* and *unloadCrane*, for loading new blanks or improperly forged blanks for reprocessing by the crane. The guard `grd4` of *loadFeedBelt* event at the top is split into guard `grd4` in both the events at the bottom. New events added in this refinement were *moveCraneToFb* and *moveCraneToDb* as shown below:

Event *moveCraneToFb* $\widehat{=}$
when
grd1 : $cranePos = crposdb$
//crane is at deposit belt
grd2 : $cr \in ran(position)$
//crane has a blank on it
then
act1 : $cranePos := crposfb$
end

Event *moveCraneToDb* $\widehat{=}$
when
grd1 : $cranePos = crposfb$
//crane is at feed belt
grd2 : $cr \notin ran(position)$
//crane is vacant
then
act1 : $cranePos := crposdb$
end

This refinement models requirements 36, 38, 39, 40 and 41 of the PC requirement specification.

Event *loadFeedBelt* $\widehat{=}$
refines *loadFeedBelt*
//Event of PC_3
any
b
where
grd1 : $b \in BLANKS$
grd2 : $b \in dom(blanks)$
grd3 : $blanks(b) \neq forged$
grd4 : $b \notin dom(position) \lor position(b) = cr$
grd5 : $card(dom(position \rhd \{fb\})) < fbMax$
then
act1 : $position := position \lhd\!\!+ \{b \mapsto fb\}$
end

Event *loadFeedBelt* $\widehat{=}$
refines *loadFeedBelt*
//Event of PC_4
any
b
where
grd1 : $b \in BLANKS$
grd2 : $b \in dom(blanks)$
grd3 : $blanks(b) \neq forged$
grd4 : $b \notin dom(position)$
grd5 : $card(dom(position \rhd \{fb\})) < fbMax$
then
act1 : $position := position \lhd\!\!+ \{b \mapsto fb\}$
end

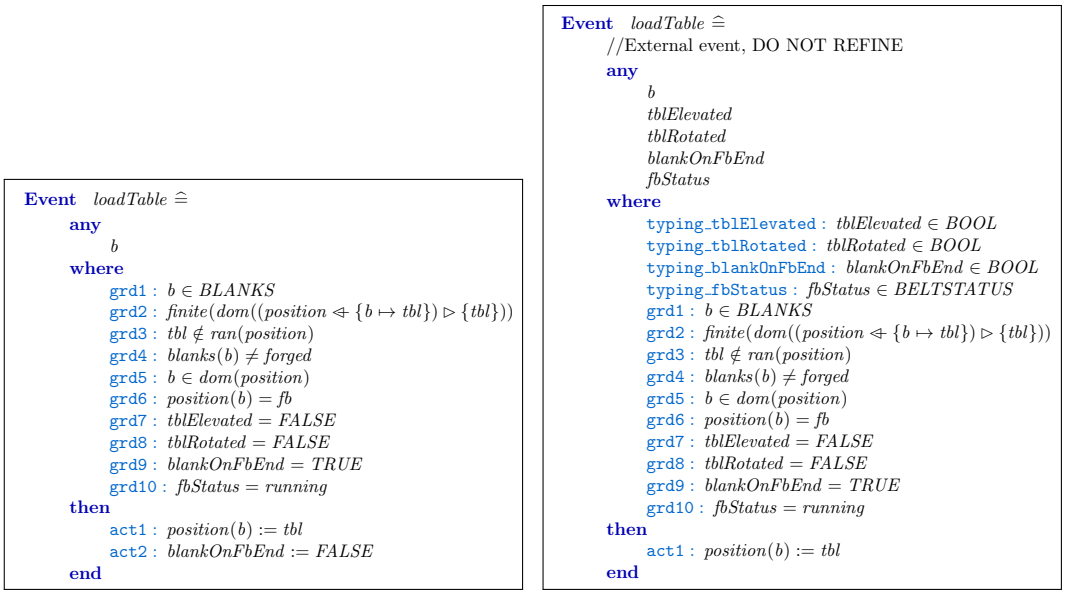
Event *unloadCrane* $\widehat{=}$
refines *loadFeedBelt*
//Event of PC_4
any
b
where
grd1 : $b \in BLANKS$
grd2 : $b \in dom(blanks)$
grd3 : $blanks(b) \neq forged$
grd4 : $position(b) = cr$
grd5 : $card(dom(position \rhd \{fb\})) < fbMax$
grd6 : $cranePos = crposfb$
then
act1 : $position := position \lhd\!\!+ \{b \mapsto fb\}$
end

Figure 5.8: Refinement of event `loadFeedBelt` in PC_4

There are various other requirements for efficiency and flexibility which we have not modelled due to time limitations and can be introduced in further refinement steps, as discussed by [100].

5.4.7 Decomposition

After four horizontal refinements where most of the PC requirements were modelled, we then wanted to introduce actuators and sensors for different components of the PC and perform vertical refinement. The functionality of sensors and actuators is independent for each of the components. Also, the model became quite big which was difficult to understand and refine further as a whole. So, we decomposed the model into various physical components (sub-models) of the PC (i.e., *feed belt, table, robot, press, deposit belt* and *crane*). Following is the detail for applying both styles of Event-B decomposition to the component-based production cell development.



(a) `LoadTable` Event of *Table*

(b) `LoadTable` event of *Press*

Figure 5.9: `LoadTable` as internal and external event in *Table* and *Press* components respectively after SVD

Table 5.1: Event distribution among components of PC based on SVD

Event Type	Feed Belt	Table	Robot	Press	Deposit Belt	Crane
Internal Events	3	5	9	5	3	4
External Events	13	11	14	10	10	7

5.4.7.1 Shared-Variable Decomposition (SVD)

At first, we used shared-variable decomposition (SVD) since different components were sharing variables. For example, all components shared the variable `blanks`, which models the status of blanks at any component. The variables `tblRotated` and `tblElevated` are shared among feed belt, table and robot components and appear as event parameters in the other components rather than as shared-variables. Same is the case with the variable `pressPos` which is shared between press and robot components. During the decomposition, events related to a particular physical component became events of that sub-model and any events of the sub-model involving the shared-variable became external events in all other components. For instance, event `loadTable` moved to the *table* sub-model, became an external event in all the sub-models for other physical components of PC. Figure 5.9 shows `loadTable` as an internal and external event in the table and press components respectively. Table 5.1 shows the distribution of internal/external events among various components of PC after SVD.

Table 5.2: Number of local and shared events of PC components based on SED

Event Type	Feed Belt	Table	Robot	Press	Deposit Belt	Crane
Local Events	0	4	7	4	3	2
Shared Events	4	4	4	4	2	2

5.4.7.2 Shared-Event Decomposition (SED)

In order to explore whether we can use shared-event decomposition (SED) to decompose the integral model into sub-models, we had to prepare the model to be decomposed using the SED style, right from the abstract model. For this, we had to partition (localise) any variables that are shared between different components, so that there is no more shared variables. For example, the shared variable `position` in PC_4 (i.e., $position \in BLANKS \nrightarrow POS$), which maintains the position of a blank on a component within the plant, would be partitioned for each component. So, each component knows the blank(s) present on it, e.g., variable `blanksOnFb` (i.e., $blanksOnFb \in \mathbb{P}(BLANKS)$) models the blanks present on the feed belt and so on for the rest of the plant components.

During the decomposition, we partitioned the variables into various sub-models along with their related events. Figure 5.10 shows how an event *`loadTable`* is split into two events for the *feed belt* and *table* components. We simply split the guards and actions into two. If a guard or action of an event is complex and can not be split then it must be simplified during the preparatory step to be split into two. For example, Figure 5.11 shows the event *`moveFeedBelt`* prepared in a refinement step to be split, as the guard `grd3` of the event on the left could not be partitioned. So, we had to introduce extra parameters (`t1, t2, t3`) and guards (`grd3, grd4, grd5`) to simplify this guard. Hence, the guard `grd3` of event on the left is simplified by guard `grd6` on the right. The decomposition of this event is shown in Figure 5.12 for feed belt and table components. Note that we had to do vertical refinement in order for us to perform SED unlike SVD where we only carried out horizontal refinements before the decomposition. So, it depends on the type of system being modelled and for distributed systems, the SED approach seems more appropriate.

Table 5.2 shows the local and shared events of various components of PC after SED. Table 5.3 shows the distribution of internal/external events among various components of PC if we had performed SVD instead of SED. In this case, there are no globally shared variables and the variables are only shared between connecting components, e.g., the *table* component shares variables with the *feed belt* and *robot* as it interacts with the two in order to receive and deliver blanks.

Table 5.3: Events distribution of PC components when performed SVD on a model prepared for SED

Event Type	Feed Belt	Table	Robot	Press	Deposit Belt	Crane
Internal Events	3	5	9	5	4	3
External Events	9	4	11	5	2	3

Event *loadTable* $\widehat{=}$
//Event before decompsoition
any
b
where
grd1 : $blankOnTbl = \varnothing$
grd2 : $b \in blanksOnFb$
grd3 : $tblElevated = FALSE$
grd4 : $tblRotated = FALSE$
grd5 : $blankOnFbEnd = TRUE$
grd6 : $fbStatus = running$
then
act1 : $blankOnTbl := blankOnTbl \cup \{b\}$
act2 : $blanksOnFb := blanksOnFb \setminus \{b\}$
act3 : $blankOnFbEnd := FALSE$
end

Event *loadTable* $\widehat{=}$
//Event of Feed Belt Component
any
b
where
typing_b : $b \in BLANKS$
grd2 : $b \in blanksOnFb$
grd5 : $blankOnFbEnd = TRUE$
grd6 : $fbStatus = running$
then
act2 : $blanksOnFb := blanksOnFb \setminus \{b\}$
act3 : $blankOnFbEnd := FALSE$
end

Event *loadTable* $\widehat{=}$
//Event of Table Component
any
b
where
typing_b : $b \in BLANKS$
grd1 : $blankOnTbl = \varnothing$
grd3 : $tblElevated = FALSE$
grd4 : $tblRotated = FALSE$
then
act1 : $blankOnTbl := blankOnTbl \cup \{b\}$
end

Figure 5.10: Event splitting example for SED

Event *moveFeedBelt* $\widehat{=}$
when
grd1 : $card(blanksOnFb) > 0$
grd2 : $fbStatus = stopped$
grd3 : $blankOnFbEnd = FALSE \lor (tblElevated = FALSE \land tblRotated = FALSE \land blankOnTbl = \varnothing)$
then
act1 : $fbStatus := running$
end

Event *moveFeedBelt* $\widehat{=}$
refines *moveFeedBelt*
any
$t1, t2, t3$
where
grd1 : $card(blanksOnFb) > 0$
grd2 : $fbStatus = stopped$
grd3 : $t1 = tblElevated$
grd4 : $t2 = tblRotated$
grd5 : $t3 = blankOnTbl$
grd6 : $(blankOnFbEnd = FALSE) \lor (t1 = FALSE \land t2 = FALSE \land t3 = \varnothing)$
then
act1 : $fbStatus := running$
end

Figure 5.11: Example of preparing an event for SED

Event *moveFeedBelt* $\widehat{=}$
//Event of Feed Belt Component
any
$t1, t2, t3$
where
typing_t3 : $t3 \in \mathbb{P}(BLANKS)$
typing_t2 : $t2 \in BOOL$
typing_t1 : $t1 \in BOOL$
grd1 : $card(blanksOnFb) > 0$
grd2 : $fbStatus = stopped$
grd6 : $(blankOnFbEnd = FALSE) \lor (t1 = FALSE \land t2 = FALSE \land t3 = \varnothing)$
then
act1 : $fbStatus := running$
end

Event *moveFeedBelt* $\widehat{=}$
// Event of Table Component
any
$t1, t2, t3$
where
typing_t3 : $t3 \in \mathbb{P}(BLANKS)$
typing_t2 : $t2 \in BOOL$
typing_t1 : $t1 \in BOOL$
grd3 : $t1 = tblElevated$
grd4 : $t2 = tblRotated$
grd5 : $t3 = blankOnTbl$
then
skip
end

Figure 5.12: Event of Figure 5.11 decomposed into two events

5.4.7.3 Discussion

Modelling a system with no shared variables seems to be a better option here. This is because after SVD we get lots of external events which must not be refined. This increases size of the model making it difficult to manage/understand after some refinements. For example, as shown in Table 5.1, the feed belt component of PC has 3 internal events and 13 external events. Table 5.1 and Table 5.3 show the number of internal/external events of PC components when decomposed using the two styles for the shared and disjoint variable modelling approaches respectively. The total number of events for a component in Table 5.3 is less than that of the same component in Table 5.1.

On the other hand, sub-models resulting from SED may have some events which do not make much sense on their own due to their other parts being present in another sub-model as a consequence of event sharing. For example, event `moveFeedBelt` in table component of Figure 5.12 is only meaningful when presented along with its other half in the feed belt component. The sub-models resulting from SED could be modelled independently without any restrictions. Though an extra preparatory refinement step

Table 5.4: Proofs statistics of component-based PC modelled for SED/SVD

Model	SVD-based	SED-based
PC_0	3	2
PC_1	43	29
PC_2	25	21
PC_3	43	30
PC_4	10	18
Press_1	83	57
Press_2	21	20
Press_3	107	89

is usually required in order to make the state variables disjoint to be decomposed using SED.

After decomposing the model into sub-models, we could then refine each of these sub-models independently. In case of SVD, we had to maintain the restrictions of the SVD style while refining these sub-models, i.e., to ensure that the shared variables and external events were not refined. We further refined the *press* sub-model vertically by introducing actuators and sensors to model it closer to implementation. This involved another three levels of refinement and was done using the refinement pattern for control systems [51]. We refined both of the press sub-models resulting from the two styles of decomposition, i.e., SVD and SED. The two press sub-models are different in a way that one has external events and shared-variables whereas the other only has local variables and shared events. Their refinement approach was similar since we followed the same refinement pattern for the two developments. We only discuss the press refinement resulting from SVD (i.e., *Press_0*) below as the other followed the same refinement structure. Figure 5.13 shows variables and internal events of this model (*Press_0*). Other sub-models of component-based PC (i.e., table, robot, etc.) could also be refined similarly. Table 5.4 shows the number proof obligations at each refinement level for modelling component-based PC from the view of SVD/SED.

5.4.8 Press First Refinement Model (Press_1)

The press model (*Press_0* of Figure 5.13) resulting from SVD of component-based PC (*PC_4*) was further refined by introducing actuators and sensors for handling the press. Since the press can have three positions (i.e., *Low, Mid* and *High*), there were three variables for the position sensors (i.e., `pressPosLowSensor, pressPosMidSensor, pressPosHighSensor`). A variable for an electric motor of the press (`pressMotor`) was introduced for its state, i.e., moving the motor forward, backward or stopped. We added events for turning the sensors on and off (i.e., *`pressLowSensorOn/Off, pressMidSensorOn/Off, pressHighSensorOn/Off`*). Events were added for starting

VARIABLES
`blanks, position, pressPos, robotPos`
//Shared variables, DO NOT REFINE
EVENTS
Event *movePressToHigh* $\widehat{=}$
when
grd1 : $pr \in ran(position)$
grd2 : $pressPos = mid$
then
act1 : $pressPos := high$
end
Event *movePressToMid* $\widehat{=}$
when
grd1 : $pressPos = low$
grd2 : $pr \notin ran(position)$
then
act1 : $pressPos := mid$
end
Event *movePressToLow* $\widehat{=}$
when
grd1 : $pr \in ran(position)$
grd2 : $pressPos = high$
then
act1 : $pressPos := low$
end
Event *loadPress* $\widehat{=}$
any
b
where
grd1 : $b \in BLANKS$
grd2 : $pr \notin ran(position)$
grd3 : $b \in dom(position)$
grd4 : $position(b) = arm1$
grd5 : $pressPos = mid$
grd6 : $robotPos = pos2$
then
act1 : $position(b) := pr$
end
Event *opPress* $\widehat{=}$
any
b
where
grd1 : $b \in BLANKS$
grd4 : $b \in dom(position)$
grd2 : $position(b) = pr$
grd3 : $blanks(b) \neq forged$
grd5 : $pressPos = high$
grd6 : $finite(blanks \Leftarrow \{b \mapsto forged\})$
then
act1 : $blanks(b) := forged$
end
END

Figure 5.13: Variables and events of Press_0 resulting from SVD of PC_4

and stopping the motor at different press positions (e.g., ***startPressMotorDown***, ***stopPressMotorAtLowPos*** etc.). Figure 5.14 shows how the event ***movePressToLow*** was refined using atomicity decomposition in various refinement steps. So, an event for moving the press from high to low position in *Press_0* was decomposed into four events in *Press_1*. The motor starts from high position that leads to the high position sensor turning off. When the low position sensor turns on, the motor is stopped. These events are shown in Figure 5.15. Other events were introduced in a similar way. Below are some of the invariants at this refinement model.

inv4 : $pressPosMidSensor = on \Rightarrow pressPosLowSensor = off \wedge pressPosHighSensor = off \wedge pressPos \in \{low, mid\}$
//Turn on mid sensor when the press arrives at mid position from low position

inv5 : $pressPosLowSensor = on \Rightarrow pressPosMidSensor = off \wedge pressPosHighSensor = off \wedge pressPos \in \{high, low\}$
//Turn on low sensor when the press arrives at the low position from high position

inv6 : $pressPosHighSensor = on \Rightarrow pressPosMidSensor = off \wedge pressPosLowSensor = off \wedge pressPos \in \{mid, high\}$
//Turn on high sensor when the press arrives at the high position from mid position

inv8 : $pressMotor = bwd \Rightarrow (\exists b \cdot b \in BLANKS \wedge b \mapsto pr \in position \wedge blanks(b) = forged) \wedge pressPos = high$

inv9 : $pressMotor = fwd \wedge pressPos = mid \Rightarrow (\exists b \cdot b \in BLANKS \wedge b \mapsto pr \in position \wedge blanks(b) = unforged)$

inv10 : $pressMotor = fwd \wedge pressPos = low \Rightarrow pr \notin ran(position)$

Some gluing invariants were required to relate the abstract state to the new variables. For example, invariant '`inv8`' shown above makes sure that the press will only move

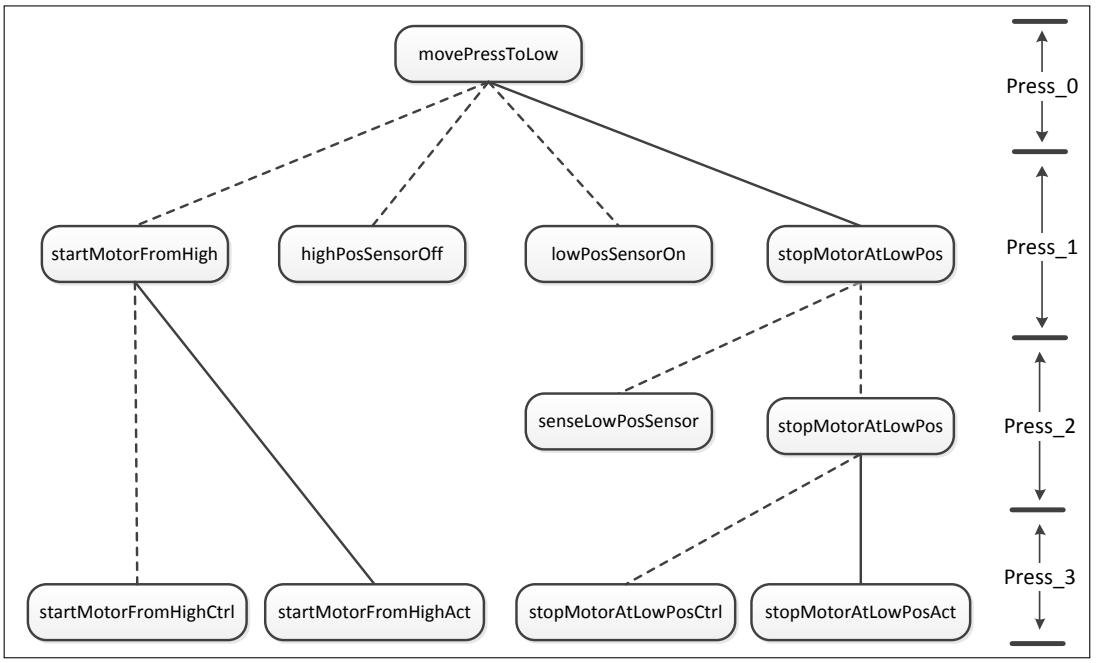


Figure 5.14: Event refinement for press component

down if it is at the high position and has a blank on it which has been processed. The invariant '`inv9`' specifies that the press should only move up when it is at the mid position and has received a blank to be processed. Similarly, the invariant '`inv10`' ensures that the press only moves up from the low position when the blank on it has been removed by the robot. We managed to discharge all the proof obligations. There were 83 POs generated by the Rodin, only 18 were discharged interactively and the rest were discharged automatically by the Rodin provers.

5.4.9 Press Second Refinement Model (Press_2)

In the second refinement, using the sensing pattern from the cookbook [51], we refined the model so the controller uses the sensed values of the sensors. This is done to model the system closer to reality as the value of the sensors at some point in time will be different from the sensed values. We have not yet introduced the notion of time which can be introduced later on. We added variables for sensed values of the sensors and a flag for each sensor to monitor whether a sensor has been sensed or not. Gluing invariants were added, e.g., a sensor flag will be true if the sensed value is same as the actual value of the sensor and false otherwise, as shown below:

`inv7` : $prMidSensedFlag = TRUE \Rightarrow pressMidSensed = pressPosMidSensor$
`inv8` : $prLowSensedFlag = TRUE \Rightarrow pressLowSensed = pressPosLowSensor$
`inv9` : $prHighSensedFlag = TRUE \Rightarrow pressHighSensed = pressPosHighSensor$

Event *startMotorFromHigh* $\widehat{=}$
  any
    b
  where
    grd1 : $pressMotor = stop$
    grd2 : $pressPosHighSensor = on$
    grd5 : $pressPos = high$
    grd6 : $b \in dom(position)$
    grd7 : $position(b) = pr$
    grd8 : $blanks(b) = forged$
  then
    act2 : $pressMotor := bwd$
  end

Event *highPosSensorOff* $\widehat{=}$
  when
    grd1 : $pressPosHighSensor = on$
    grd2 : $pressPos = high$
    grd3 : $pressMotor = bwd$
  then
    act1 : $pressPosHighSensor := off$
  end

Event *lowPosSensorOn* $\widehat{=}$
  when
    grd1 : $pressPosHighSensor = off$
    grd2 : $pressPosMidSensor = off$
    grd3 : $pressPosLowSensor = off$
    grd4 : $pressMotor = bwd$
    grd5 : $pressPos = high$
    grd6 : $pr \in ran(position)$
  then
    act1 : $pressPosLowSensor := on$
  end

Event *stopMotorAtLowPos* $\widehat{=}$
refines *movePressToLow*
  when
    grd1 : $pr \in ran(position)$
    grd2 : $pressPos = high$
    grd3 : $pressMotor = bwd$
    grd4 : $pressPosLowSensor = on$
  then
    act1 : $pressPos := low$
    act2 : $pressMotor := stop$
  end

Figure 5.15: Events of $Press\_1$ resulting from atomicity decomposition of *movePressToLow* event of $Pres\_0$

Event *stopPressMotorAtLowPos* $\widehat{=}$
refines *stopPressMotorAtLowPos*
  when
    grd1 : $pr \in ran(position)$
    grd2 : $pressPos = high$
    grd3 : $pressMotor = bwd$
    grd4 : $prLowSensedFlag = TRUE$
    grd5 : $pressLowSensed = on$
  then
    act1 : $pressPos := low$
    act2 : $pressMotor := stop$
    act3 : $prLowSensedFlag := FALSE$
  end

Event *sensePressLowSensorVal* $\widehat{=}$
  when
    grd1 : $prLowSensedFlag = FALSE$
  then
    act1 : $pressLowSensed := pressPosLowSensor$
    act2 : $prLowSensedFlag := TRUE$
  end

Figure 5.16: Events of $Press\_2$ resulting from atomicity decomposition of *stopMotorAtLowPos* event of $Pres\_1$

The event decomposition diagram of Figure 5.14 shows how *stopMotorAtLowPos* event of $Press\_1$ was decomposed into two events in $Press\_2$ for sensing the sensor value (new event: *SenseLowPosSensor*) before stopping the motor (refined event:*StopMotorAtLowPos*), as shown in Figure 5.16. Similarly, other events were introduced for different sensors. This was a smaller refinement step as compared to $Press\_1$. There were 21 POs and all were discharged automatically.

Event *stopPressMotorAtLowPosAct* $\widehat{=}$
refines *stopPressMotorAtLowPos*
when
grd1 : $pr \in ran(position)$
grd2 : $pressPos = high$
grd3 : $pressMotor = bwd$
grd4 : $prLowSensedFlag = TRUE$
grd5 : $pressLowSensed = on$
grd6 : $pressMotorCtrlFlag = TRUE$
grd7 : $pressMotorCtrl = stop$
then
act1 : $pressPos := low$
act3 : $prLowSensedFlag := FALSE$
act2 : $pressMotor := pressMotorCtrl$
act4 : $pressMotorCtrlFlag := FALSE$
end

Event *stopPressMotorAtLowPosCtrl* $\widehat{=}$
when
grd1 : $pr \in ran(position)$
grd2 : $pressPos = high$
grd3 : $pressMotorCtrl = bwd$
grd4 : $prLowSensedFlag = TRUE$
grd5 : $pressLowSensed = on$
grd6 : $pressMotorCtrlFlag = FALSE$
then
act2 : $pressMotorCtrl := stop$
act3 : $pressMotorCtrlFlag := TRUE$
end

Figure 5.17: Events of *Press_3* resulting from atomicity decomposition of `stopMotorAtLowPos` event of *Pres_2*

5.4.10 Press Third Refinement Model (Press_3)

At this level of refinement, the actuation was refined where a controller sets the actuation of a motor before the motor is actuated. Again, this brings the model closer to implementation as in reality there will be a delay in setting the actuator and its actual movement based on the actuation set by the controller. An abstract actuation event is decomposed into two events for a controller and actuator. For example, in Figure 5.14, the event `stopMotorAtLowPos` of *Press_2* is split into `stopMotorAtLowPosCtrl` (new event) and `stopMotorAtLowPosAct` (refined event) in *Press_3*. These two events are shown in Figure 5.17.

There were 107 POs generated at this refinement level where most of these POs were discharged automatically with a few discharged interactively. This model can be further refined vertically by introducing the notion of time.

Other components (e.g., *table, robot, feed belt* etc.) can also be modelled in the same way as the press using the refinement pattern for control systems. Their refinements will be similar to the press as we will have to model the actuators and sensors for these components in the same way.

5.4.11 Building Variants of PC

The component-based modelling approach discussed above gives us a product of PC which models a particular topology. This topology is presented in Figure 5.18 which shows the component-based PC model at fourth refinement level, before decomposition into various components. The figure shows how the physical components are connected

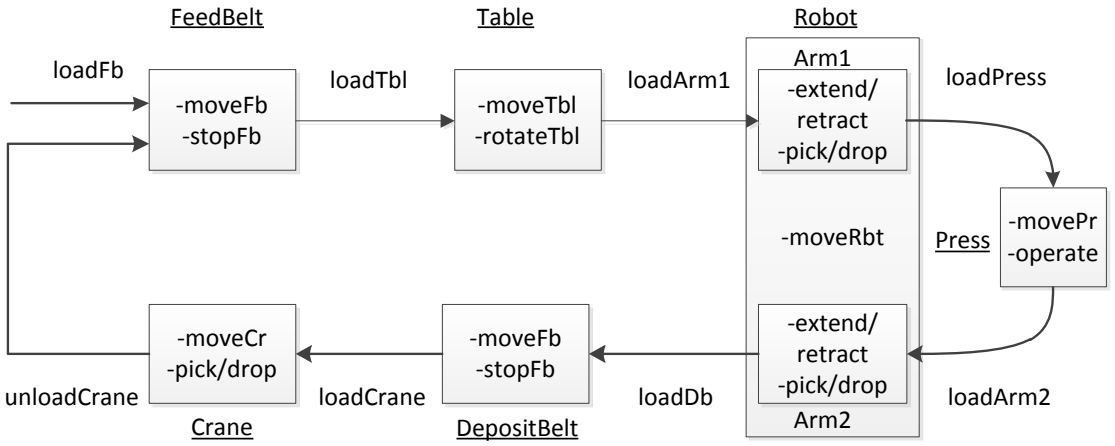


Figure 5.18: PC Topology1

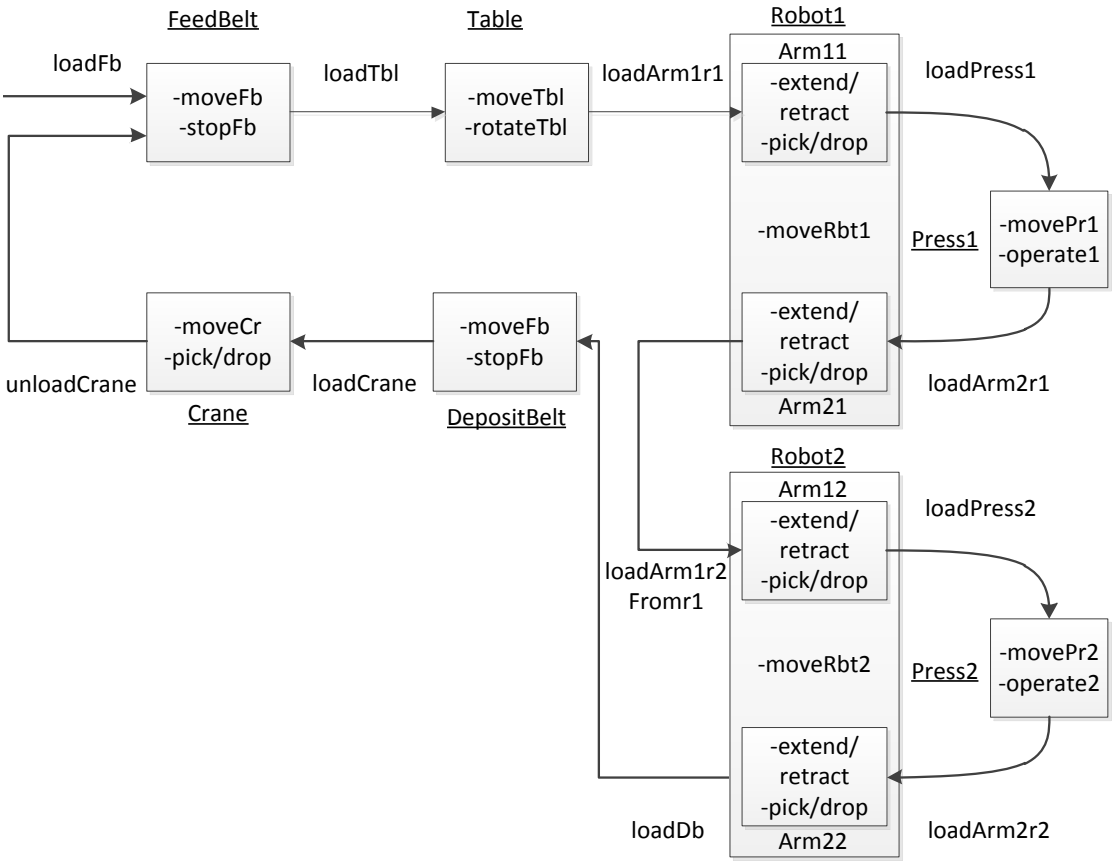


Figure 5.19: PC Topology2

with each other to model the production cell plant. It is designed to visualise the topology in terms of components and their events and helps in reuse for instantiation of alternative topologies. The boxes in the figure represent the physical components of production cell. The events placed inside the boxes are internal (local) for the components and those placed between the boxes represent shared events containing topological information for modelling the connectivity of components. This is an example of domain specific instance modelling with Event-B.

Each different topology is an instance of PC product line and we can build more variants of PC by selecting a different configuration (or topology) of these physical components. For example, consider a production cell with two press components for processing blanks twice and using two robots. We can call this 'topology 2' where *robot1* picks a blank from the *table* and drops onto *press1* and *robot2* takes the blank from *robot1* which picks it from *press1* and drops on *press2* (see Figure 5.19). Here we are interested in exploring to what extent we can reuse the models of *topology1* while modelling *topology2* and hence the proof effort. In terms of the requirements specification listed in Appendix A, we reused most of the requirements of *topology1* with the requirements for the robot and press components were duplicated. The additional requirement is the collection of blanks by *robot2* from *robot1* for processing in the second *press*.

We had to do instantiation and refactoring to simply duplicate the functionality of existing components for modelling *topology2*. This means that we would not have to reprove the models which have already been proved for *topology1*. This is because renaming of elements would not affect the proof obligations (POs) and is currently supported by the refactory plug-in [122] in Rodin. We only had to prove the POs generated for any additional information modelled in the second topology. For example, we specify that *arm1* of *robot2* collects the blank from *arm2* of *robot1* unlike picking it from the *table* in *topology1*.

For topology2 development, we started with the abstract model of topology1 and duplicated the events `Operate` for processing blanks twice, i.e. front and back (events `OperateFront, OperateBack`). Both events could be achieved through refactoring. We added an additional guard in `OperateBack` which ensures that it happens after `OperateFront` event, as shown below:

Event *Operate* $\widehat{=}$
any
b
where
grd1 : $b \in dom(blanks)$
grd2 : $blanks(b) \neq forged$
then
act1 : $blanks(b) := forged$
end

Event *OperateFront* $\widehat{=}$
any
b
where
grd1 : $b \in dom(blanks)$
grd2 : $blanks(b) = unforged$
then
act1 : $blanks(b) := forgedF$
end

Event *OperateBack* $\widehat{=}$
any
b
where
grd1 : $b \in dom(blanks)$
grd3 : $blanks(b) = forgedF$
then
act1 : $blanks(b) := forgedB$
end

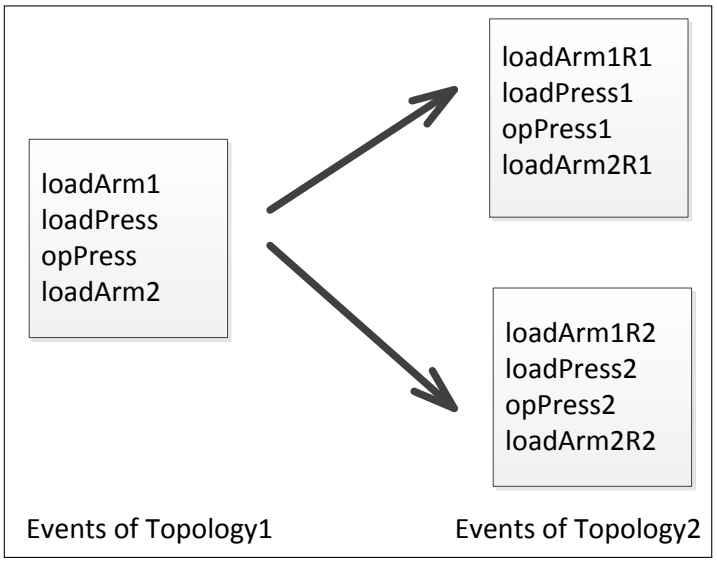


Figure 5.20: Events refactoring for modelling topology2 after topology1 - 1st refinement

In the first refinement, we duplicated the functionality for press and robot which means we had to replicate all the events related to both of these components. Figure 5.20 shows events of topology1 and that of topology2 at first refinement level. We also had to duplicate variables and invariants while duplicating a component. For example, a variable `arm1State` in Topology1 became `arm1r1State` and `arm1r2State` for robot1 and robot2 respectivey in Topology2 along with duplication of their typing and other invariants, as shown below:

INVARIANTS

//Topology 1

inv1 : $arm1State \in ARMSTATE$

inv5 : $arm1State = extended \Rightarrow (robotPos = pos1 \lor robotPos = pos3)$

INVARIANTS

//Topology 2

inv1 : $arm1r1State \in ARMSTATE$

inv2 : $arm1r2State \in ARMSTATE$

inv7 : $arm1r1State = extended \Rightarrow (robot1Pos = pos1 \lor robot1Pos = pos3)$

inv9 : $arm1r2State = extended \Rightarrow (robot2Pos = pos1 \lor robot2Pos = pos3)$

Figure 5.21 show events of *topology1* and that of *topology2* as a result of refactoring at second, third and fourth level of refinement.

Figure 5.22 shows the refinement architecture for modelling the two topologies and their components as achieved after decomposition. An example of event instantiation while modelling *topology2* after *topology1* is shown in Figure 5.23 where event `loadPress` is

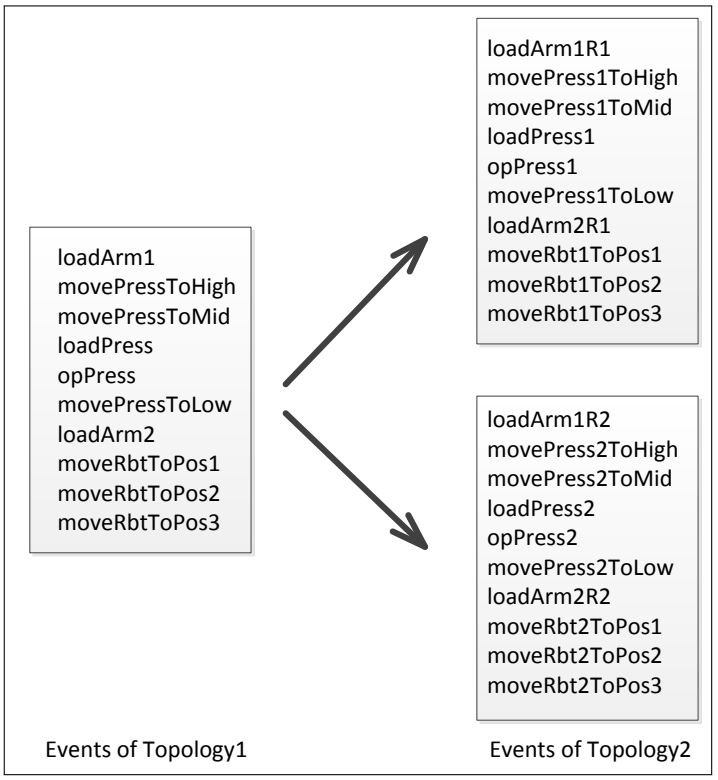


Figure 5.21: Events refactoring for modelling topology2 after topology1 - 2nd, 3rd and 4th refinement

duplicated for the two presses (events: *loadPress1* & *loadPress2*). This type of instantiation and refactoring has no proof burden. Figure 5.25 shows the proof obligations for the event that were refactored when modelling the second topology along with the additional POs. Figure 5.24 shows another example of event instantiation for loading robot arm1 in *topology1* (i.e., event *loadArm1*) . Here, the instantiation is not trivial. We refactor the instantiated events for arm1 of robot1 (*loadArm1R1*) and robot2 (*loadArm1R2*), add and modify some guards and actions accordingly. This would require us to prove that the refactored events preserve invariants.

The POs for *topology2* were discharged in the same way as *topology1*. Table 5.5 shows the number of POs for both topology developments at different refinement levels and how these were discharged, i.e., automatically or interactively. It also shows the amount of newly generated POs, reused (i.e., appear as these were in *topology1*) when modelling the second topology and those generated as a result of refactoring. It shows the percentage of reused POs as well. This means that refactoring utility could save the effort of reproving these refactored POs. Table 5.6 shows the number of events and invariants for the two topology developments along with the number of events reused from *topology1* (i.e., appear without any modification) and those refactored (i.e., duplicated along with some renaming). This experiment shows that we can reuse existing models and their proofs if we have tools to automate the instantiation and refactoring processes; hence, generating additional tooling requirements which are discussed later.

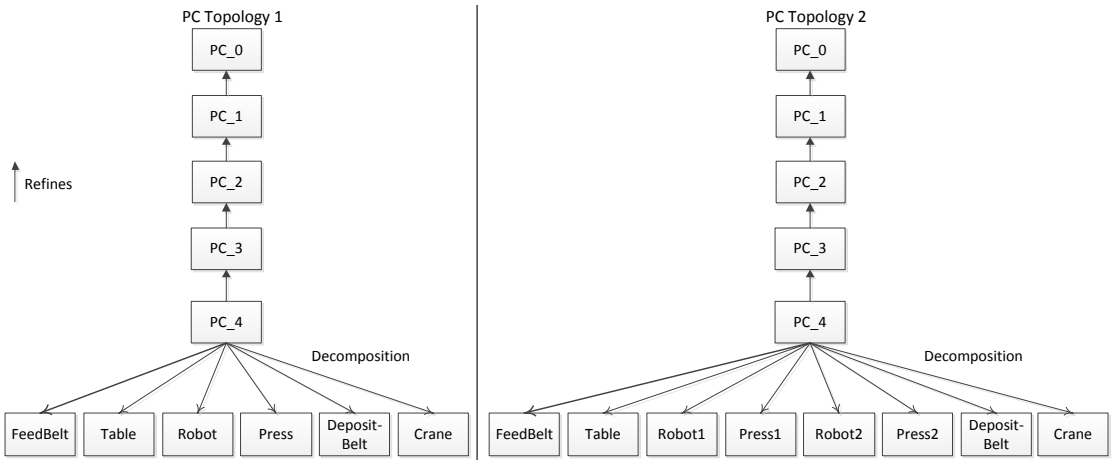


Figure 5.22: Refinement architecture for modelling the two topologies

```
event loadPress
  any b
  where
    @grd1 b∈BLANKS
    @grd2 pr ∉ran(position)
    @grd3 b∈dom(position)
    @grd4 position(b) = arm1
    @grd5 pressPos=mid
    @grd6 robotPos=pos2
  then
    @act1 position(b) = pr
end
```

```
event loadPress1
  any b
  where
    @grd1 b∈BLANKS
    @grd2 pr1 ∉ran(position)
    @grd3 b∈dom(position)
    @grd4 position(b) = arm1r1
    @grd5 press1Pos=mid
    @grd6 robot1Pos=pos2
  then
    @act1 position(b) = pr1
end
```

```
event loadPress2
  any b
  where
    @grd1 b∈BLANKS
    @grd2 pr2 ∉ran(position)
    @grd3 b∈dom(position)
    @grd4 position(b) = arm1r2
    @grd5 press2Pos=mid
    @grd6 robot2Pos=pos2
  then
    @act1 position(b) = pr2
end
```

Figure 5.23: Event instantiation example for PC topology2

5.4.12 Evaluation

We have discussed component-based modelling approach for PC. We refined PC both horizontally and vertically and decomposed the integral model into various components using both styles of Event-B decomposition (SVD/SED). After decomposition, we refined the press component further to model actuators and sensors in various refinement steps, using the pattern for modelling control systems in Event-B [51]. Other components could be modelled in the same way and hence not discussed here. In order to explore how we could reuse models of this development (which we call topology1), we modelled another variant of PC that has more components than in the first one. Here we wanted to model a PC which has two presses and two robots to process a blank twice. This was done by simply duplicating the functionality of these two components which means

```
loadArm1:
    REFINES
        loadArm1
    ANY
        b
    WHERE
        grd1:   b∈BLANKS
        grd2:   arm1∉ran(position)
        grd3:   b∈dom(position)
        grd4:   position(b) = tbl
        grd5:   robotPos=pos1
        grd6:   tblElevated=TRUE
        grd7:   tblRotated=TRUE
    THEN
        act1:   position(b) = arm1
    END
```

```
loadArm1R1:
    REFINES
        loadArm1R1
    ANY
        b
    WHERE
        grd1:   b∈BLANKS
        grd2:   arm1r1∉ran(position)
        grd3:   b∈dom(position)
        grd4:   position(b) = tbl
        grd5:   robot1Pos=pos1
        grd6:   tblElevated=TRUE
        grd7:   tblRotated=TRUE
    THEN
        act1:   position(b) = arm1r1
    END
```

```
loadArm1R2:
    REFINES
        loadArm1R2
    ANY
        b
    WHERE
        grd1:   b ∈ BLANKS
        grd2:   arm1r2 ∉ran(position)
        grd3:   b∈dom(position)
        grd4:   position(b) = arm2r1
        grd5:   blanks(b)=forgedF
        grd6:   robot1Pos=pos3
        grd7:   robot2Pos=pos1
    THEN
        act1:   position(b) = arm1r2
    END
```

Figure 5.24: Event instantiation example for PC topology2

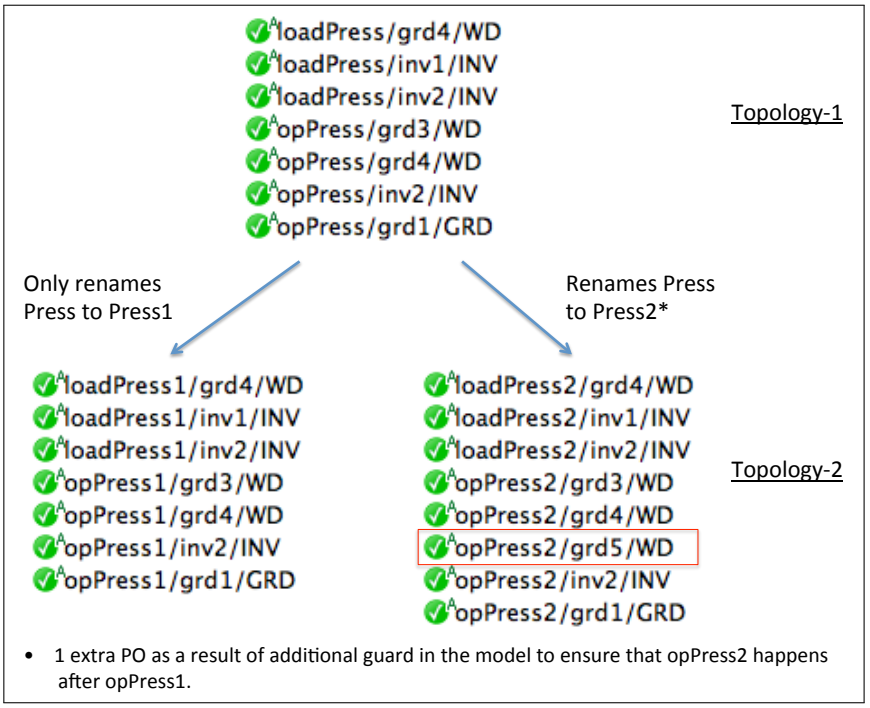


Figure 5.25: Proof obligations refactoring for an event of topology 2

Table 5.5: Proof obligations statistics for two topology developments

PC Topology 1				PC Topology 2						
Ref. level	Auto	Manual	Total	Total	Auto	Manual	New POs	Reuse Refa-ctored POs	Reused POs	Reuse %
PC-0	3	-	**3**	**6**	6	-	-	5	1	100
PC-1	35	8	**43**	**79**	58	21	15	39	25	81
PC-2	18	7	**25**	**25**	15	10	-	6	19	100
PC-3	31	12	**43**	**57**	40	17	2	32	23	96
PC-4	10	-	**10**	**8**	8	-	-	-	8	100
PC-5	81	-	**81**	**184**	116	68	23	138	23	87
PC-6	20	-	**20**	**40**	40	-	6	34	-	85
PC-7	167	15	**190**	**420**	387	13	39	342	39	90
Total	365	42	**415**	**819**	769	28	85	596	138	89

Table 5.6: Events and invariants statistics for topology 1 and 2

PC Topology 1			PC Topology 2			
Ref. level	Invariants	Events	Invariants	Events	Refactored Events	Reused Events
PC-0	1	2	1	3	2	1
PC-1	3	9	4	13	4	9
PC-2	7	19	9	29	20	9
PC-3	10	27	14	41	28	13
PC-4	3	30	1	44	28	16
PC-5	10	39	20	62	46	16
PC-6	9	42	19	68	52	16
PC-7	14	48	28	80	64	16

we had to duplicate the events related to these components through refactoring. We can define variation points as the connecting/shared events (events connecting boxes in Figure 5.18). These variation points may vary for any different topology which means these would behave differently unlike event duplication which are only refactored to make these disjoint. We have shown the reuse statistics when modelling topology2 after topology1, which clearly shows the amount of reuse achieved. This includes parts of Event-B specification (events) as well as proofs so that the proof effort could also be avoided.

This topology development approach seems feasible only when we need to duplicate a couple of components or to model a couple of topologies. Imagine a case where we have to model a topology having many instances of a component (e.g., 10 presses and 10 robots). This would become very hard to model, refine and manage the overall development with the current state of tool support. One way to deal with the issue of

modelling any number of topologies with multiple replication of components is to raise the abstraction level of our base topology model. We can specify generic components and generic events in our model that could be specialised as and when required to model various components of a particular topology through reuse and generic instantiation. For example, we can have an event that models the transfer of blanks from one component to another (i.e., *passBlankBtwCpt1Cpt2*) as shown below:

Event *passBlankBtwCpt1Cpt2* $\widehat{=}$
any
b
where
grd1 : $b \in blanks$
grd2 : $position(b) = Cpt1$
grd1 : $Cpt2 \notin ran(position)$
then
act1 : $position(b) := Cpt2$
end

This event could be specialised for passing blanks between any two components of the plant where we could refactor *Cpt1* and *Cpt2* with feed belt and table and so on for the rest of the components. This generic base topology could be refined to model internal mechanise of a component. For example, in a refinement model, a generic event for moving a component between two positions could be introduced. This event could then be specialised for table or crane components, etc.

This experiment generated further tool requirements and also helped us suggest guidelines for users (discussed later) who would like to benefit from reuse when modelling control systems in Event-B.

5.5 PC Controller-based

In the controller-based PC modelling, functional requirements for modelling the behaviour of each controller of the production cell were grouped together as a feature. So, the requirements specification (given in Appendix A) was decomposed into various controller features. We also generalised these requirements for each of the controller so that we could model generic controllers; which could then be specialised and reused for modelling various controllers of different physical components of the PC. Hence, the controller-based modelling of PC was a result of decomposition plus generalisation. Figure 5.26 shows Event-B refinement architecture for controller-based PC modelling.

Table 5.7 shows part of the requirements specification for the *table* feature of component-based PC (column 1) and the *movement* feature for controller-based PC (column 2). This

shows how we can define the feature in terms of requirements for two styles of modelling the PC, while making the features more reusable. The compositional requirements are implemented while actually composing various components; this may include topological information and how components are connected together. Similarly, Table 5.8 shows part of the requirements specification for the *crane* feature of component-based PC (column 1) which can be modelled by specialising the requirements of the generic *magnet* feature for controller-based PC (column 2). The controller-based PC models consisted of *loader, movement, rotation* and *magnet* controllers. A member of PC product line could be modelled by instantiating and composing these controller-based reusable features. These features were then refined independently. We only discuss the refinement of *magnet* and *movement* features below where we introduced sensors and actuators in various refinement steps using the pattern for refining control systems as suggested in [51].

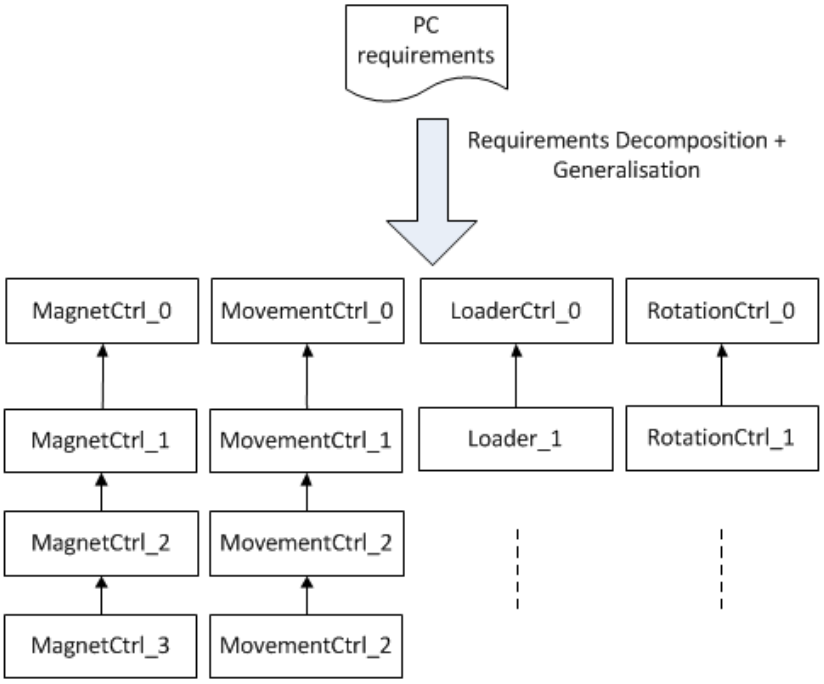


Figure 5.26: Controller-based PC modelling

5.5.1 Magnet Controller

At the abstract level, we have events for picking and dropping of blanks by a component. A component which has not already picked a blank can do so and a component which has picked a blank can drop it as shown below:

Table 5.7: Requirements description for *Table* and *Movement* features of PC

Table Component	**Generic Movement Controller**
Req#6 - Move table upwards/downwards Req#6 - Rotate table clockwise/anti-clockwise Req#9 - The table must not rotate clockwise if it is in a position to deliver blanks (unloading position) Req#10 - Table must not rotate when its at low position Req#11 - Table must not move down if it is rotated (rotate backward first and then then move down) or if it is already not elevated Compositional Requirements Req#7 - Drop blank on table from feed belt when it is in the loading position (not elevated and not rotated) Req#8 - Robot picks blank from table when it is in unloading position (elevated and rotated)	- Move a component from position A to position B and vice-versa Instantiation Requirements - Extend/Retract Arm1 - Extend/Retract Arm2 - Move Feed Belt/Deposit Belt - Move the Table upwards/ downwards - Move Crane To and from Feed Belt/Deposit Belt Compositional Requirements - Extend/Retract Arm1 if robot is facing table or facing press while press is in middle position - Extend/Retract Arm2 if robot is facing deposit or facing press while press is in lower position Table must not move down if its rotated (rotate backward first and then move down) or if it is already not elevated - Crane should only move towards feed belt if it is positioned on deposit belt and vice-versa

Table 5.8: Requirements description for *Crane* and *Magnet* features of PC

Crane Component	**Generic Magnet Controller**
Req#35 - A travelling crane is used to bring the unforged blanks back from the deposit belt to the feed belt Req#36 - The crane has an electromagnet gripper to pick and drop the blank Compositional Requirements Req#39 - The crane picks a blank from deposit belt if there is one at the end of the belt Req#40 - The crane drops a blank on to the feed belt if it is not already full	- Pick and drop a blank Instantiation Requirements - Pick and drop functionality for Arm1 - Pick and drop functionality for Arm2 - Pick and drop functionality for Crane Compositional Requirements - Arm1 picks a blank from table and drops on the press when the press is in middle position - Arm2 picks a blank from press and drops on the deposit belt - Crane picks a blank from the deposit belt and drops on the feed belt

Event $XcompXPickBlank \mathrel{\widehat{=}}$
any
b
where
grd1 : $XcompX \notin ran(position)$
grd2 : $b \in dom(position)$
then
act1 : $position(b) := XcompX$
end

Event $XcompXDropBlank \mathrel{\widehat{=}}$
any
b
where
grd1 : $XcompX \in ran(position)$
grd2 : $b \in dom(position)$
then
act1 : $position := position \setminus \{b \mapsto XcompX\}$
end

The notation '`XcompX`' means a generic placeholder for a component which must be filled with appropriate component during specialisation. So, this feature will be instantiated to a specific component such as a *crane* or a *robot arm*. The model is quite abstract and the details were added later in the refinements and during specialisation. Figure 5.27 shows how the two events of this abstract model are refined along with the introduction of new events in three refinements steps.

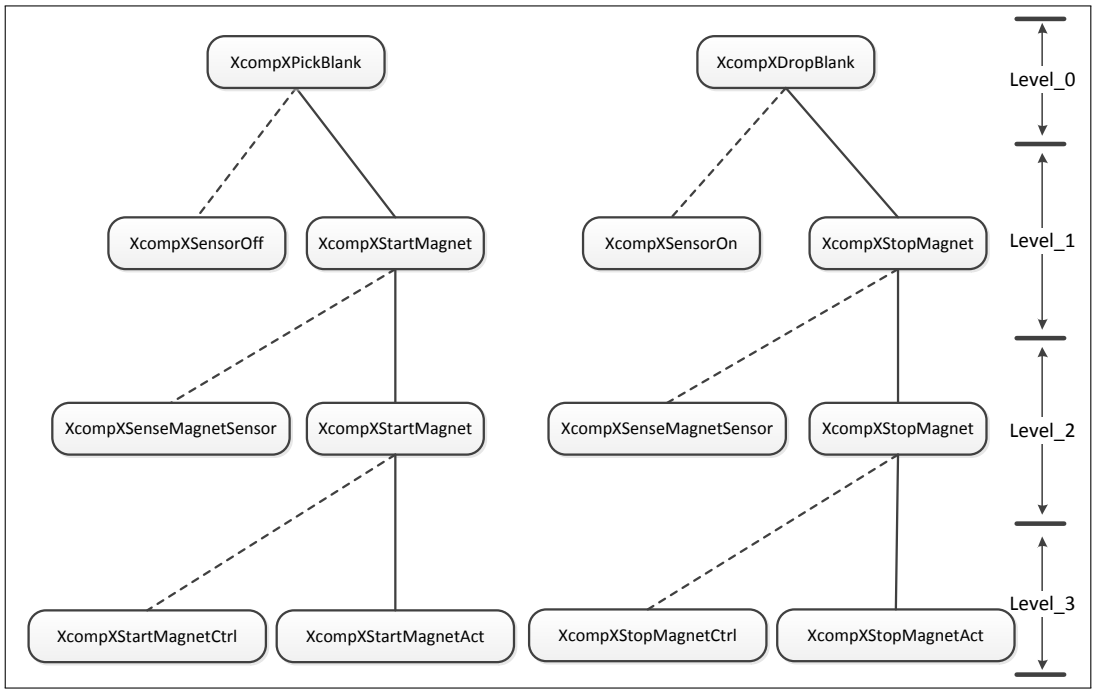


Figure 5.27: Atomicity refinement of magnet controller

In the first refinement, we added sensor functionality for magnet which informs the controller whether a blank has been picked up or dropped off. An electromagnet switch acts as an actuator for the magnet which performs the pick and drop of blanks. Two variables were introduced as following:

$$\texttt{inv1}: \; XcompXMagnetSensor \in SENSOR$$
$$\texttt{inv2}: \; XcompXMagnet \in ELECMAGNET$$

We also added events for starting and stopping the magnet and switching the sensor on and off as shown below:

Event *XcompXstartMagnet* $\widehat{=}$
refines *XcompXPickBlank*
any
b
where
grd1 : XcompX $\notin$ ran(position)
grd2 : b $\in$ dom(position)
grd3 : *XcompXMagnet* = *drop*
grd4 : *XcompXMagnetSensor* = *off*
then
act1 : position(b) := XcompX
act2 : *XcompXMagnet* := *pick*
end
Event *XcompXsensorOn* $\widehat{=}$
when
grd1 : *XcompXMagnetSensor* = *off*
grd2 : *XcompXMagnet* = *pick*
then
act1 : *XcompXMagnetSensor* := *on*
end

Event *XcompXstopMagnet* $\widehat{=}$
refines *XcompXDropBlank*
any
b
where
grd1 : XcompX $\in$ ran(position)
grd2 : b $\in$ dom(position)
grd3 : *XcompXMagnet* = *pick*
grd4 : *XcompXMagnetSensor* = *on*
then
act1 : position := position $\setminus$ {b $\mapsto$ XcompX}
act2 : *XcompXMagnet* := *drop*
end
Event *XcompXsensorOff* $\widehat{=}$
when
grd1 : *XcompXMagnetSensor* = *on*
grd2 : *XcompXMagnet* = *drop*
then
act1 : *XcompXMagnetSensor* := *off*
end

In the second refinement, we differentiate between the actual and sensed values of the sensors. So, a boolean variable `XcompXMagSensorFlag` (shown below) was introduced to ensure that the magnet's start event takes place when both the sensed and actual values are same.

inv4 : *XcompXMagSensorFlag* = *TRUE* $\Rightarrow$ *XcompXMagSensedVal* = *XcompXMagnetSensor*

Similarly, in the third refinement, we refined the actuation mechanism where controller sets the actuation of the motor before the motor actually starts moving. Here we split the actuation events into two, i.e., a new event for setting the actuation of magnet by the controller and a refined event for magnet to actuate accordingly. Figure 5.28 shows invariants and events for the third refinement model of magnet controller. For example, ***XcompXStartMagnetCtrl*** is a new event which tells the plant to start magnet for picking a blank by turning on its flag and is then followed by the event ***XcompXStartMagnetAct*** which actually picks the blanks. Similarly, ***XcompXStopMagnetCtrl*** is a new event to stop magnet for dropping a blank by turning on its flag which then leads to the event ***XcompXStopMagnetAct*** for actually dropping the blank. The sensor events occur after a start or stop magnet event takes place. The refinement style is similar to that of the press refinement discussed earlier, since both of these use the same refinement pattern of [51].

INVARIANTS

inv1 : $XcompXMagActStartFlag \in BOOL$

inv2 : $XcompXMagActStopFlag \in BOOL$

inv3 : $XcompXMagnetCtrl \in ELECMAGNET$

inv4 : $XcompXMagActStartFlag = TRUE \Rightarrow (XcompX \notin ran(position) \land XcompXMagnet = drop \land XcompXMagSensedVal = off \land XcompXMagSensorFlag = TRUE)$

inv5 : $XcompXMagActStopFlag = TRUE \Rightarrow (XcompX \in ran(position) \land XcompXMagnet = pick \land XcompXMagSensedVal = on \land XcompXMagSensorFlag = TRUE)$

Event $XcompXStartMagnetCtrl \mathrel{\widehat{=}}$
any
b
where
grd1 : $XcompX \notin ran(position)$
grd2 : $b \in dom(position)$
grd3 : $XcompXMagnet = drop$
grd4 : $XcompXMagSensedVal = off$
grd5 : $XcompXMagSensorFlag = TRUE$
grd6 : $XcompXMagActStartFlag = FALSE$
grd7 : $XcompXMagActStopFlag = FALSE$
then
act1 : $XcompXMagnetCtrl := pick$
act3 : $XcompXMagActStartFlag := TRUE$
end

Event $XcompXStartMagnetAct \mathrel{\widehat{=}}$
refines $XcompXstartMagnet$
any
b
where
grd2 : $b \in dom(position)$
grd6 : $XcompXMagActStartFlag = TRUE$
grd7 : $XcompXMagnetCtrl = pick$
then
act1 : $position(b) := XcompX$
act2 : $XcompXMagnet := XcompXMagnetCtrl$
act3 : $XcompXMagSensorFlag := FALSE$
act4 : $XcompXMagActStartFlag := FALSE$
end

Event $XcompXStopMagnetCtrl \mathrel{\widehat{=}}$
when
grd1 : $XcompX \in ran(position)$
grd3 : $XcompXMagnet = pick$
grd4 : $XcompXMagSensedVal = on$
grd5 : $XcompXMagSensorFlag = TRUE$
grd6 : $XcompXMagActStopFlag = FALSE$
grd7 : $XcompXMagActStartFlag = FALSE$
then
act2 : $XcompXMagnetCtrl := drop$
act3 : $XcompXMagActStopFlag := TRUE$
end

Event $XcompXStopMagnetAct \mathrel{\widehat{=}}$
refines $XcompXstopMagnet$
any
b
where
grd2 : $b \in dom(position)$
grd6 : $XcompXMagActStopFlag = TRUE$
grd7 : $XcompXMagnetCtrl = drop$
then
act1 : $position := position \setminus \{b \mapsto XcompX\}$
act2 : $XcompXMagnet := XcompXMagnetCtrl$
act3 : $XcompXMagSensorFlag := FALSE$
act4 : $XcompXMagActStopFlag := FALSE$
end

Event $XcompXsensorOn \mathrel{\widehat{=}}$
extends $XcompXsensorOn$
when
grd1 : `XcompXMagnetSensor` = `off`
grd2 : `XcompXMagnet` = `pick`
grd3 : `XcompXMagSensorFlag` = `FALSE`
then
act1 : `XcompXMagnetSensor` := `on`
end

Event $XcompXsensorOff \mathrel{\widehat{=}}$
extends $XcompXsensorOff$
when
grd1 : `XcompXMagnetSensor` = `on`
grd2 : `XcompXMagnet` = `drop`
grd3 : `XcompXMagSensorFlag` = `FALSE`
then
act1 : `XcompXMagnetSensor` := `off`
end

Event $XcompXSenseMagnetSensor \mathrel{\widehat{=}}$
extends $XcompXSenseMagnetSensor$
when
grd1 : `XcompXMagSensorFlag` = `FALSE`
then
act1 : `XcompXMagSensedVal` := `XcompXMagnetSensor`
act2 : `XcompXMagSensorFlag` := `TRUE`
end

Figure 5.28: Magnet controller third refinement model

5.5.2 Movement Controller

The development structure of movement controller is similar to that of the magnet controller. At the abstract level, we have events for moving a physical component forward and backward between two positions (i.e., $posA$, $posB$), as shown below:

Event $XcompXMoveToXposBX \mathrel{\widehat{=}}$
when
grd1 : $XPosVarX = XposAX$
then
act1 : $XPosVarX := XposBX$
end

Event $XcompXMoveToXposAX \mathrel{\widehat{=}}$
when
grd1 : $XPosVarX = XposBX$
then
act1 : $XPosVarX := XposAX$
end

INVARIANTS

inv1 : $XcompXmotor \in MOTOR$
inv2 : $XcompXSensorPosA \in SENSOR$
inv3 : $XcompXSensorPosB \in SENSOR$
inv4 : $XcompXSensorPosA = on \Rightarrow XcompXSensorPosB = off$
inv5 : $XcompXSensorPosB = on \Rightarrow XcompXSensorPosA = off$
inv6 : $XcompXmotor = stop \wedge XPosVarX = XposAX \Rightarrow XcompXSensorPosA = on$
inv7 : $XcompXmotor = stop \wedge XPosVarX = XposBX \Rightarrow XcompXSensorPosB = on$

EVENTS

Event $XcompXStartMotorFwd \widehat{=}$
when
grd1 : $XcompXSensorPosA = on$
grd2 : $XcompXmotor = stop$
then
act1 : $XcompXmotor := fwd$
end

Event $XcompXStopMotorFwd \widehat{=}$
extends $XcompXMoveToXpos2X$
when
grd1 : $XPosVarX = XposAX$
grd2 : $XcompXmotor = fwd$
grd3 : $XcompXSensorPosB = on$
then
act1 : $XPosVarX := XposBX$
act2 : $XcompXmotor := stop$
end

Event $XcompXStartMotorBwd \widehat{=}$
when
grd1 : $XcompXSensorPosB = on$
grd2 : $XcompXmotor = stop$
then
act1 : $XcompXmotor := bwd$
end

Event $XcompXStopMotorBwd \widehat{=}$
extends $XcompXMoveToXpos1X$
when
grd1 : $XPosVarX = XposBX$
grd2 : $XcompXmotor = bwd$
grd3 : $XcompXSensorPosA = on$
then
act1 : $XPosVarX := XposAX$
act2 : $XcompXmotor := stop$
end

Event $XcompXSensorPosBOn \widehat{=}$
when
grd1 : $XcompXSensorPosB = off$
grd2 : $XcompXSensorPosA = off$
then
act1 : $XcompXSensorPosB := on$
end

Event $XcompXSensorPosBOff \widehat{=}$
when
grd1 : $XcompXSensorPosB = on$
grd2 : $XcompXSensorPosA = off$
grd3 : $XcompXmotor = bwd$
then
act1 : $XcompXSensorPosB := off$
end

Event $XcompXSensorPosAOn \widehat{=}$
when
grd1 : $XcompXSensorPosA = off$
grd2 : $XcompXSensorPosB = off$
then
act1 : $XcompXSensorPosA := on$
end

Event $XcompXSensorPosAOff \widehat{=}$
when
grd1 : $XcompXSensorPosA = on$
grd2 : $XcompXSensorPosB = off$
grd3 : $XcompXmotor = fwd$
then
act1 : $XcompXSensorPosA := off$
end

END

Figure 5.29: Movement controller events and invariants for first refinement

This feature will be instantiated to a specific component such as a *crane* or a *table*. In case of crane feature, *posA* can become *deposit belt* and *posB* can be instantiated as *feed belt*. During the first refinement, we added sensors for the two positions and a motor for moving the component backward and forward. Events were added for starting and stopping the motor at different positions and switching the sensors on and off. Figure 5.29 shows events and invariant of the first refinement level. Here, we have events for starting and stopping the motor in forward and backward direction and turning the sensors on and off at positions A and B. In the second refinement, we differentiate between the actual and sensed values of the sensors as discussed earlier. Using the same refinement style of magnet controller, at third level of refinement, we differentiate between setting the motor's actuation by the controller from its actual movement.

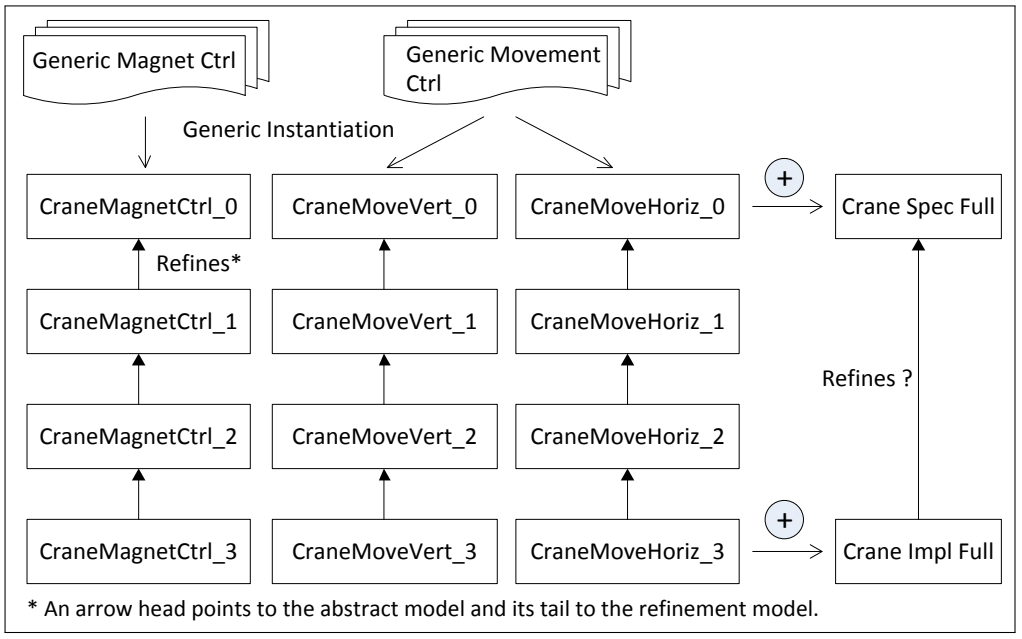


Figure 5.30: Crane instantiation

5.5.3 Instantiation & Composition

The *magnet* and *movement* controllers provide us refinement chains of generic Event-B models for the two features. In order to model any component of the PC, we need to instantiate and compose these chains of models. The composition is done in two phases. In the first phase, we specialise and compose generic models to build a component (e.g., *crane* or *robot*) and in the second phase, we compose all the components while providing topological information. For example, if we want to model the *crane* component, we have to specialise one instance of the *magnet* controller to pick and drop blanks and two instances of the *movement* controllers for moving the *crane* horizontally and vertically, as shown in Figure 5.30. In this example, we have three refinements in each development which align well during the composition. This alignment issue (also discussed in previous chapter) needs to be explored further to address the composition of Event-B developments having different number of refinements. Figure 5.31 shows a simple example where event ***PickBlank*** of *magnet* controller is specialised for the *crane* component. Here the generic model parameter `XcompX` is replaced by `crane`, provided both of these are of the same type. For now, we use `X...X` as a syntactic convention to model a generic parameter, given that the current Rodin tool does not support generics. This instantiation has no proof burden and the instantiated model will be correct by construction. Figure 5.32 shows another example where event ***XcompXMoveToXposBX*** of *movement* controller is specialised as ***craneMoveToPosFB*** event of the *crane* component, for moving it from deposit belt to feed belt.

The composition of abstract level models from each refinement chain would give us an abstract specification for the *crane*. We also had to do some guard strengthening and add

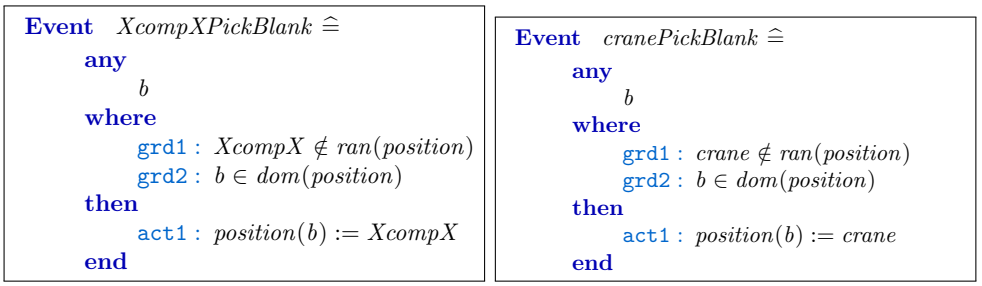


Figure 5.31: Event specialisation of magnet controller for crane

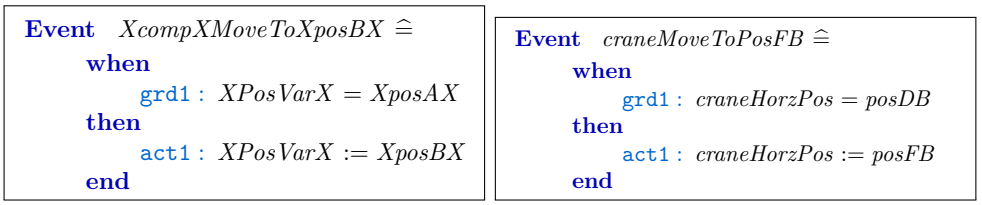


Figure 5.32: Event specialisation of movement controller for crane

some invariants during the composition. This would generate addition POs and some of the exiting POs may also change that would require reproving. The composition of implementation level models for each refinement would provide us with the implementation of the *crane*. Again extra guards for events and invariants were needed. Figure 5.33 shows the events ***cranePickBlank*** and ***craneMoveToPosFB*** (of Figures 5.31 & 5.32) with extra guards added during the composition. For example, `grd3` of ***cranePickBlank*** event specifies the topological information that the *crane* can only pick a blank when it is positioned on the deposit belt. Similarly, `grd2` of ***craneMoveToPosFB*** event in Figure 5.33 specifies that the *crane* can only move towards the *feed belt* if it has picked up a blank. The guard `grd2` of ***cranePickBlank*** event means it can pick any blank in the system. Table 5.9 shows POs for three crane components and the crane model resulting from their composition.

When we finally compose all the components to model the entire PC in the second phase of composition, we will need to strengthen this guard to say that the *crane* can only pick a blank from the *deposit belt*. Here we would need to give topological information of PC in terms of how different components are connected to each other as discussed in the components-based PC example earlier. Again by adding extra guards and invariants in this second phase of composition may generate additional POs to be reproved.

We call this style of composition 'feature composition' where additional information can be added during the composition. As of yet, this style of composition does not guarantee refinement preservation between the composed abstract and implementation models (see 'refines?' in Figure 5.30). In order to deal with this kind of composition, we need the support for proof reuse. By this we mean to find a way of automatically discharging composite POs with the help of already discharged POs of the components being composed. This requires further work. Although, we will discuss an alternative

Event *cranePickBlank* $\widehat{=}$
any
b
where
grd1 : $crane \notin ran(position)$
grd2 : $b \in dom(position)$
grd3 : $craneHorzPos = posDB$
then
act1 : $position(b) := crane$
end

Event *craneMoveToPosFB* $\widehat{=}$
when
grd1 : $craneHorzPos = posDB$
grd2 : $crane \in ran(position)$
then
act1 : $craneHorzPos := posFB$
end

Figure 5.33: Guard strengthening of events during composition

Event *cranePickBlank* $\widehat{=}$
any
b
where
grd1 : $crane \notin ran(position)$
grd2 : $b \in dom(position)$
grd3 : $craneHorzPos = posDB$
then
act1 : $position(b) := crane$
end

Event *craneMoveToPosFB* $\widehat{=}$
when
grd1 : $craneHorzPos = posDB$
grd2 : $crane \in ran(position)$
then
act1 : $craneHorzPos := posFB$
end

Figure 5.34: Guard strengthening of events during composition

Table 5.9: Proof obligations statistics for crane development

Ref. level	Crane Magnet	Crane Movement Vertical	Crane Movement Horizontal	Crane Full
L-0	3	-	-	**3**
L-1	-	24	24	**48**
L-2	8	16	16	**37**
L-3	20	67	67	**140**

approach in the next chapter where we can avoid reproof by following a particular modelling pattern, that approach would not be feasible for modelling generic components of PC due to its blank passing topological nature.

We had to use feature composition because the SED could not be applied here due to the shared variables between the components being composed. Similarly, the SVD approach is too constraining and could only be used here if we start with an abstract model containing the functionality of both the *magnet* and *movement* features. We could then decompose these into two developments (both having external events), refine each of these, instantiate for the crane and compose to build the crane model. This would require all the additional components being to be modelled in the same manner which would limit the reuse and befit from genericity. The ATM case-study discussed in the next chapter further explores these issues and suggests the modelling style through

which we could use existing techniques of Event-B to achieve partial reuse of existing specifications, when modelling variants of a product line.

5.5.4 Evaluation

The controller-based PC modelling discussed above showed how we can improve reusability by modelling a system as a set of generic features. These feature could then be specialised for modelling various components of the system. These components when composed together would result in a particular product of a product line. In comparison to the component-based approach discussed earlier, this style of modelling SPLs in Event-B seems more appropriate for product line modelling because it provides more reuse opportunities. This approach is modelled at a different abstraction level compared to the component-based approach where reuse only lies in terms of components and their connectivity. Here, we model and reuse fine-grained features as compared to coarse-grained features of component-based PC.

For this approach, we had to generalise requirements and group these together to model as features. We showed how generic features for movement and magnet controllers could be instantiated and composed to model crane component of PC. These components along with others could be instantiated and composed to model a robot and similarly rest of the PC components. Once we have an integral model of the PC, this could again be decomposed using any of the two decomposition techniques of Event-B (i.e., SVD/SED), depending on how the system is to be implemented. This experiment requires us to reprove the instantiated product since this middle-in loose-structured composition - which we call feature composition - does not guarantee refinement preservation. We will address this issue in the next chapter by suggesting a different approach when modelling another case-study. We will also discuss how this approach could be generalised in our suggested guidelines for future users in Chapter 7.

5.6 PC – Domain-specific SPL modelling through contexts

We also modelled a generic component-based PC which supports the two topologies mentioned earlier. The variability is provided through the context which means the machine for both the topologies will remain same and we could have a different topology by just switching the contexts. Figure 5.35 shows this development architecture where a generic product is specialised by switching the contexts to build specialised products. We modelled the topology of PC in the context, i.e., we specify the physical components and how these could be connected to each other. The machine is modelled in a generic way as shown in Figure 5.36. The event *passBlankBtwCpts* models how a blank is passed from one component to another and the event *moveCpt* models the movement of a component

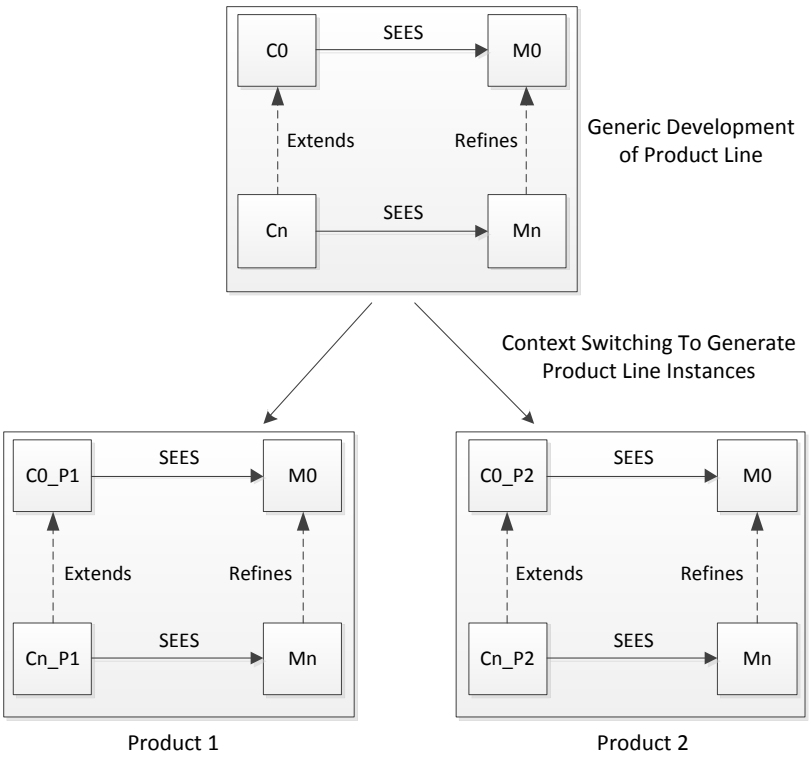


Figure 5.35: Development architecture for SPL modelling through context switching

itself as the components of PC can have different positions (e.g., *press* moves between three positions and *table* moves between two positions). The invariant '`inv1`' defines position of blank which can be at any one component at a time. The invariant '`inv2`' specifies the position of a component which can be any one of its possible positions, e.g., *press* can be in low, mid or high position and *table* can be in up or down position.

The topological information is modelled in the contextual part at the bottom of the figure where `cptGraph` defines the components graph, i.e., valid ways of connecting different components. This is then used in '`grd4`' of the ***`passBlankBtwCpts`*** event to make sure that the blank is passed between connectable components. If we like to model a different topology (e.g., *topology2* as discussed earlier) with two robots and two presses, we would simply need to modify the context, i.e., include these extra components to the set `CPT` and update the component connectivity information accordingly in the `cptGraph`. We can refine this model further to introduce sub-component functionality of a component. For example, a robot has two sub-components for its arms. This could then allow us to model a robot having three arms by modifying the context and using the same machine.

5.6.1 Evaluation

The advantage of modelling in this way is that we will not need to reprove a variant of PC resulting from static variability through context switching. The disadvantage is that

the modeller may not visualise various events of the machine for a particular topology by looking at the model unless the machine is animated. For example, in component-based approach we have events for each component while passing a blank to another component whereas in this approach we have only event reprsenting the same functionality. This could be a useful domain modelling activity for exploring variability of a product line in a distributed environment. This development could also be considered as a base for a product line and all of its products must be derived from this base development and then refined. Further work in this direction is required to explore this concept with different case-studies.

Another possibility is to refine this generic model down to a level where we introduce all the sensors and actuators; and in order to model a particular topology of PC, we could then instantiate events as required and switch contexts accordingly. This again would not need reproof effort unless additional information is required when instantiating various events of the development.

In order to model

5.7 Conclusion

By modelling production cell in three different ways, we have explored to what extent we can use existing Event-B tools and techniques for feature-based product line modelling. This also enabled us to figure out the requirements for future tooling and techniques (discussed in Chapter 7) that can further facilitate such development approach to benefit from reuse of existing models and their proofs.

The first style of modelling – component-based – is a natural approach of modelling in Event-B which used both types of decomposition techniques to reduce the complexity of modelling and proving by decomposing large models into smaller sub-models. We started with an integral model of PC and refined this up to four refinements. We then decomposed this model into various physical components of PC. Each of these components could then be refined separately. We only refined the *press* component of PC where we introduced actuators and sensors to model it closer to implementation. In order to explore variability, we modelled a variant of PC having two of the *press* and *robot* components each. This provided us with another topology of PC. So, we could specify the commonality and variability in terms of the components and the topology that can be used to connect these components respectively. Further variants of PC could be modelled by reusing existing models and altering the topology as required. We have shown reuse statistics in terms of specification and proofs when modelling two products of PC having different topologies. This clearly shows that we can significantly reduce modelling and proof effort through reuse. The downside of this approach is that the variability only exists in the number of components and their connections, i.e.,

INVARIANTS

inv1 : $blankPosition \in BLANKS \nrightarrow CPT$

inv2 : $cptPosition \in CPT \rightarrow POS$

EVENTS

Event $passBlankBtwCpts \mathrel{\widehat{=}}$

any

$b, cpt1, cpt2$

where

grd1 : $b \in BLANKS$

grd2 : $cpt1 \in CPT$

grd3 : $cpt2 \in CPT$

grd4 : $cpt1 \mapsto cpt2 \in cptGraph$

grd5 : $b \in dom(blankPosition)$

grd6 : $blankPosition(b) = cpt1$

grd7 : $cpt2 \notin ran(blankPosition)$

then

act1 : $blankPosition(b) := cpt2$

end

Event $moveCpt \mathrel{\widehat{=}}$

any

$pos1, pos2, cpt$

where

grd1 : $pos1 \in POS$

grd2 : $pos2 \in POS$

grd3 : $cptPosition(cpt) = pos1$

then

act1 : $cptPosition(cpt) := pos2$

end

CONSTANTS

`fb`, `tbl`, `arm1`, `arm2`, `pr`, `db`, `cr`

`cptGraph`

AXIOMS

axm1 : $partition(CPT, \{fb\}, \{tbl\}, \{arm1\}, \{arm2\}, \{pr\}, \{db\}, \{cr\})$

axm2 : $cptGraph \in CPT \nrightarrow CPT$

axm3 : $cptGraph = \{fb \mapsto tbl, tbl \mapsto arm1, arm1 \mapsto pr, pr \mapsto arm2, arm2 \mapsto db, db \mapsto cr\}$

axm4 : $partition(POS, \{up\}, \{down\}, \{low\}, \{mid\}, \{high\})$

Figure 5.36: Partial Event-B model of domain-specific modelling through context switching

a variation of PC could be modelled through reuse by replicating the functionality of existing components and how these would be connected to each other. In order to add a new component, we will have to model the PC again from abstract specification and doing the proof effort again. This approach is only feasible when we duplicate small number of components, otherwise it becomes very difficult to manage the development with the current tool support. This experiment helped us in exploring requirements for instantiation and refactoring tool support that could be useful in automating this approach.

The second approach of controller-based modelling is more feature-oriented as we have

modelled generic reusable features that could be instantiated and composed in different ways to model different PC components and hence benefit from their reuse. We generalised the requirements of PC and grouped these into several features. These generic features could then be modelled and refined independently. We could then specialise and compose these generic features to model a product of PC product line. We call the composition required in this modelling style as feature composition. This is a loose-structured cut-and-paste composition type which suits the feature-based modelling. Specialisation of generic features does not require reproof effort but this composition does not guarantee refinement preservation as we have to provide additional information during composition. This approach could be very useful if we can figure out proof reuse mechanism, i.e., how to discharge POs of composite model by analysing and reusing POs of the models being composed? However, we can avoid this proof reuse problem by using a modelling pattern suggested in the next chapter. Note that the existing composition techniques of Event-B could not be applied in this style of modelling.

The third approach of modelling static variability through context switching allows us to evaluate the scope of a product line and without doing the proof effort upfront. This is another way of modelling component-based PC and suits the product line development approach as we can figure out the common base for different variants (topologies) of the PC, and the configuration or the variability can be embedded in the context. This could be useful to foresee how a product line would evolve for a particular domain and later on could be modelled in one of the two styles mentioned to build a database of reusable features.

This work suggests that we can use existing tools and techniques to some extent for feature-based development using Event-B. But there are certain restrictions of the existing (de)composition techniques which must be followed and that restricts the feature-based development in terms of reuse. We have highlighted another form of composition – feature composition – which provides a less restrictive and more suitable form of composition for feature-based development. It does not support proof reuse as of yet and that requires further research work in future. In order to support our findings of the PC case-study, we present another case-study in the next chapter. This will enable use to explore any modelling patterns using existing (de)composition techniques of Event-B, to generalise our feature-oriented modelling frameworks and suggest guidelines for SPL modelling in Event-B for future users.