Chapter 6

Case Study

A case study involving the specification and refinement of an Event-B model is presented. This chapter describes how the techniques presented in the previous chapters may be used in practice. Throughout the case study, some design rules for Event-B are presented. These rules are specialisations of Event-B techniques already presented. These rules were suggested by the needs of the case study, but are general enough to be useful in other cases.

6.1 Introduction

Case studies can be described as a process or record of research in which detailed consideration is given to the development of a particular matter over a period of time. They have two main purposes: the explanation and description of the application of a particular technique (illustration purposes) and to validate the usefulness of the technique in a variety of systems (validation purpose). The described case study fulfils the first purpose: modelling a complex system from an abstraction to a more concrete model. Consequently the number of events, variables and proof obligations increase in a way that the model starts becoming hard to manage. Therefore a suitable solution at this stage is to use our decomposition technique. This procedure is repeatedly applied to the rest of the refinements. The application of decomposition in simple, abstract cases has very little or no real advantage. As aforementioned in Section. 4.4, the point of decomposition (correct abstraction level) is important, since if it is done too early, the sub-component might be too abstract and will not be able to be refined (without knowing more about the other sub-systems); if the system is decomposed too late, it will not benefit from the approach anymore. Therefore the application of decomposition only occurs after several refinements as expected.

The second purpose of case studies is usually achieved through the development of different models that represent different kind of systems. Their application allows the assessment of techniques, their suitability, advantages and disadvantages when applied in different manners. Besides the case study in this chapter, the presented techniques have already been used for different systems:

- Flash System Development [62, 60]: use of shared event composition and decomposition.
- Decomposition of a Spacecraft System [73]: use of shared event decomposition.
- Development of a Cruise Control System [190]: use of shared event composition and decomposition.
- Development of a Pipeline System [56, 12]: use of shared event composition and decomposition.
- Development of Parallel Programs [90]: use of shared variable decomposition over shared data accessed by different components.
- Development of a Multi-directional Communication Channel [163]: use of generic instantiation.

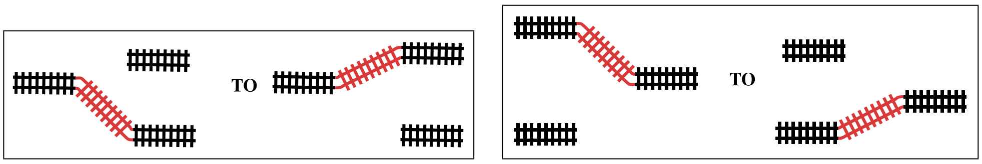
Here, a safety-critical metro system case study is developed. This version is a simplified version of a real system but tackles points where there the model becomes complex and where the presented techniques are suitable: stepwise incrementation of the complexity of the system being modelled, sub-components communication, stepwise addition of requirements at each refinement level, refinement of decomposed sub-components. We develop a metro system model introducing several details including notion of tracks, switches, several safety measures and doors functionality among others. If the presented techniques were not used, the metro system model would be extremely complex and hard to manage after the inclusion of all the requirements due to the high number of variables, events, properties to be added and proof obligations to be discharged. Decomposition and generic instantiation alleviate that issue by introducing modularity and reusing existing sub-components allowing further manageable refinements to be reached.

The metro doors requirements are based on real requirements. The case study is developed in the Rodin platform using the developed tools whenever possible. We use the shared event composition/decomposition and generic instantiation. The metro system can be seen as a distributed system. Nevertheless the modelling style suggested can be applied to a more general use.

6.2 Overview of the safety-critical metro system

The safety-critical metro system case study describes a formal approach for the development of embedded controllers for a metro system[1]. Butler [44] makes a description of embedded controllers for a railway using classical B. The railway system is based on the french train system and it was subject of study as part of the european project MATISSE [121]. Our starting point is based on that work but applied to a metro system. That work goes as far as our first decomposition originating three sub-components. We augment that work by refining each sub-component, introducing further details and more requirements to the model. Moreover in the end we instantiate emergency and service doors for the metro system.

The metro system is characterised by trains, tracks circuits (also called sections or CDV:*Circuit De Voie*, in French) and a communication entity that allows the interaction between trains and tracks. The trains circulate in sections and before a train enters or leaves a section, a permission notification must be received. In case of a hazard situation, trains receive a notification to brake. The track is responsible for controlling the sections, changing switch directions (switch is a special track that can be divergent or convergent as seen in Fig. 6.1) and sending signalling messages to the trains.



(a) Divergent Switch (b) Convergent Switch

FIGURE 6.1: Different types of Switches: divergent and convergent

Figure 6.2[2] shows a schematic representation of the metro system decomposed into three sub-components. Initially the metro system is modelled as a whole. Global properties are introduced and proved to be preserved throughout refinement steps. The abstract model is refined in three levels (*MetroSystem_M0* to *MetroSystem_M3*) before we apply the first decomposition. We follow a general top-down guideline to apply decomposition:

Stage 1 : Model system abstractly, expressing all the relevant global system properties.

Stage 2 : Refine the abstract model to fit the decomposition (preparation step).

Stage 3 : Apply decomposition.

Stage 4 : Develop independently the decomposed parts.

[1]A version of this model is available online at http://eprints.ecs.soton.ac.uk/23135/
[2]Image extracted from [44]

For instance, **Stage 1** is expressed by refinements *MetroSystem_M0* to *MetroSystem_M3*. *MetroSystem_M3* is also used as the preparation step before the decomposition corresponding to **Stage 2**. The model is decomposed into three parts: *Track*, *Train* and *Middleware* as described in **Stage 3**. This step allows further refinements of the individual sub-components corresponding to **Stage 4**. The following decompositions follow a similar pattern.

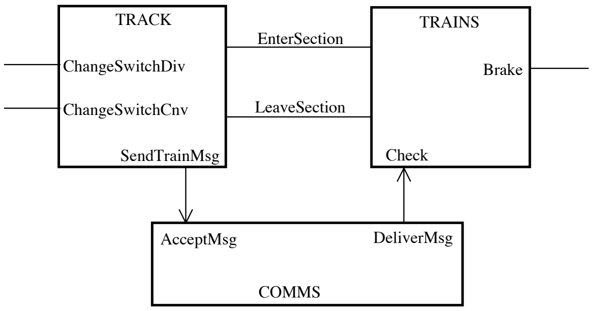


FIGURE 6.2: Components of metro system

An overview of the entire development can be seen in Fig. 6.3. After the first decomposition, sub-components can be further refined. Train global properties are introduced in *Train* leading to several refinements until *Train_M4* is reached. *Train_M4* is decomposed into *LeaderCarriage* and *Carriage*. We are interested in refining the sub-component corresponding to carriages in order to introduce doors requirements. These requirements are extracted from real requirements for metro carriage doors.*Carriage* is refined and decomposed until it fits in a generic model *GCDoor* corresponding to a *Generic Carriage Door* development as seen in Fig. 6.4. We then instantiate *GCDoor* into two instances: *EmergencyDoors* and *ServiceDoors* benefiting from the refinements in the pattern. We describe in more detail each of the development steps in the following sections.

6.3 Abstract Model: *MetroSystem_M0*

We model a system constituted by trains that circulate in tracks. The tracks are divided into smaller parts called sections. The most important (safety) global property introduced at this stage states that two trains cannot be in the same section at the same time (which would mean that the trains had clashed).

We need to ensure some properties regarding the routes (set of track sections):

- Route sections are all connected: sections should be all connect and cannot have empty spaces between them.

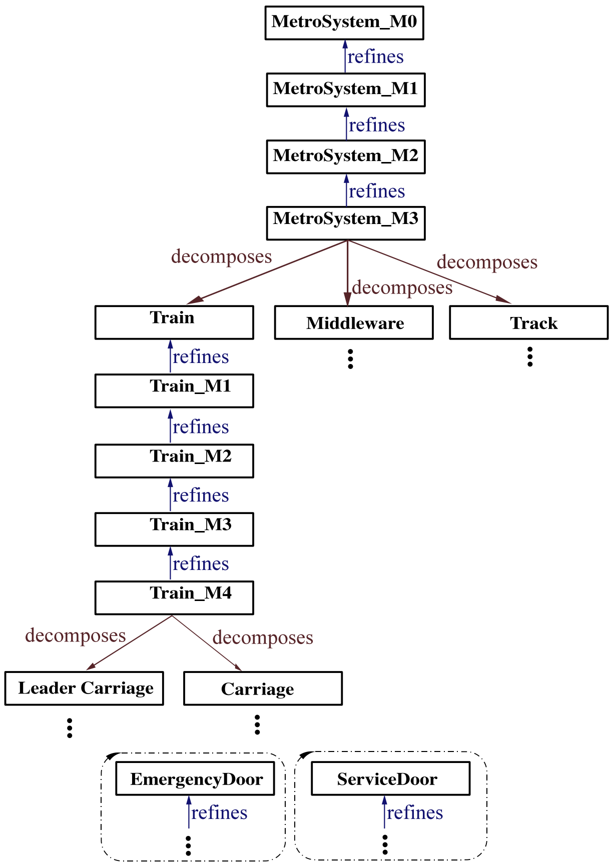


FIGURE 6.3: Overall view of the safety-critical metro system development

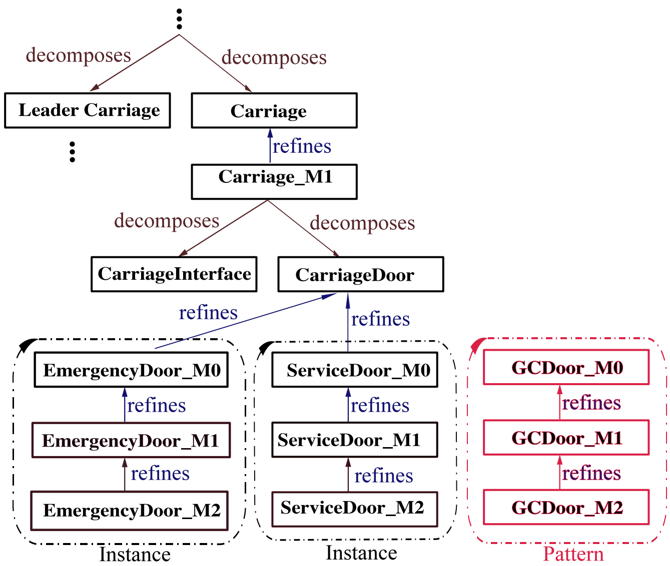


FIGURE 6.4: Carriage Refinement Diagram and Door Instantiation

- There are no loops in the route sections: sections cannot be connected to each other and cannot introduce loops.

These properties can be preserved if we represent the routes as a transitive closure relation. We use the no-loop property proposed by Abrial [9] applied to model a tree structured file system in Event-B [61]: a context is defined and this property is proved over track section relations and functions. The reason we choose this formulation, instead of transitive closure which is generally used is to make the model simpler and easier to prove. Context $TransitiveClosureCtx$ containing the transitive closure property can be seen in Fig. 6.5.

```
context TransitiveClosureCtx

constants cdvrel // type of relation on sections
        tcl // transitive closure of an cdvrel
        cdvfn // type of function on sections */

sets CDV // Track Sections

axioms
  @axm1 cdvrel = CDV ↔ CDV
  @axm2 cdvfn = CDV ⇸ CDV
  @axm3 tcl ∈ cdvrel → cdvrel
  @axm4 ∀r·(r∈cdvrel ⇒ r ⊆ tcl(r)) // r included in tcl(r)
  @axm5 ∀r·(r∈cdvrel ⇒r;tcl(r) ⊆ tcl(r)) // unfolding included in tcl(r)
  @axm6 ∀r,t·(r∈cdvrel ∧ r⊆t ∧ r;t⊆t ⇒ tcl(r)⊆t) // tcl(r) is least
  theorem @thm1 cdvfn ⊆ cdvrel
  theorem @thm2 ∀r·r∈cdvrel ⇒ tcl(r) = r ∪ (r;tcl(r)) // tcl(r) is a fixed
point
  theorem @thm3 ∀t·t∈cdvfn∧(∀s·s⊆t~[s]⇒s=∅)⇒tcl(t)∩(CDV ◁ id)=∅
theorem @thm4 tcl(∅) = ∅
end
```

FIGURE 6.5: Context *TransitiveClosureCtx*

Set CDV represents all the track sections in our model. Constant *tcl* which is a transitive closure, it is defined as a total function mapped from $CDV \leftrightarrow CDV$ to $CDV \leftrightarrow CDV$. Giving $r \in CDV \leftrightarrow CDV$, the transitive closure of r is the least x satisfying $x = r \cup r; x$ [61]. Difficult transitive closure proofs in machines are avoided by using theorems such as theorem $thm3$ shown in Fig. 6.5: for $s \subseteq CDV$ and t as a partial function $CDV \nrightarrow CDV$, $s \subseteq t^{-1}[s]$ means that s contains a loop in the t relationship. Hence, this states that the only such set that can exist is the empty set and thus the t structure cannot have loops. This theorem has been proved using the interactive prover of Rodin. The strategy to prove this theorem is to use proof by contradiction [61].

We define the environment of the case study (static part) with context *MetroSystem_C0* that extends *TransitiveClosureCtx* as seen in Fig. 6.6. Set $TRAIN$ represent all the trains in our model. Several track properties are described in the axioms:

- The constant *net* represents the total possible connectivity of sections (all possible routes subject to the switches positions) defined as relation $CDV \leftrightarrow CDV$ ($axm1$). No circularity is allowed as described by $axm2$. Moreover, the no loop property

```
context MetroSystem_C0 extends TransitiveClosureCtx

constants aig_cdv // Switches
          net // Total connectivity of sections */
          div_aig_cdv // divergent switches 1->2
          cnv_aig_cdv // convergent switches 2->1
          next0

sets TRAIN

axioms
  @axm1 net ∈ CDV ↔ CDV // net represents the connectivity between track sections /*
  @axm2 net ∩(CDV ◁ id)=∅ // no cdv is connected to itself
  @axm3 aig_cdv ⊆ CDV // aig_cdv is a subset of CDV representing cdv that are switches
  @axm4 div_aig_cdv  ⊆ aig_cdv // div_aig_cdv ⊆ aig_cdv
  @axm5 cnv_aig_cdv ⊆ aig_cdv
  @axm6 div_aig_cdv ∩ cnv_aig_cdv = ∅
  @axm7 finite(net) // explicite declaration to simplify the proving
  @axm8 (aig_cdv × aig_cdv) ∩ net = ∅ // switches are not directly connected
  @axm9 ∀cc·(cc ∈ (CDV\aig_cdv) ⇒ card(net[{cc}]) ≤1 ∧ card(net∼[{cc}])≤1) // non
switch cdv has at most one successor and at most one predecessor
  @axm10 ∀cc·( cc ∈ aig_cdv ⇒ ( (card(net[{cc}])≤2 ∧ card(net∼[{cc}])≤1) ∨ (
card(net[{cc}]) ≤1 ∧ card( net∼[{cc}])≤2 ))) // switch cdv has at most two predecessors
and one successor or one predecessor and two successors
  @axm11 tcl(net)∩id=∅ // No-loop property
   theorem @thm1 tcl(net) = net ∪ (net;tcl(net))// the transitive closure of net is
equal to net ∪ net;tcl(net)
end
```

FIGURE 6.6: Context *MetroSystem_C0*

for *net* is expressed by axiom *axm*11. Theorems *thm*1 states that *net* preserves transitive closure.

- Switches (*aiguillages* in French) are sections (*axm*3) that cannot be connected to each others (*axm*6). They are represented by *aig_cdv* divided into two kinds: *div_aig_cdv* for divergence switches and *cnv_aig_cdv* for convergent switches. Moreover switches have at most two predecessors and one successor or one predecessor and two successors (*axm*10).
- Non-switches have at most one successor and at most one predecessor (*axm*9).

Besides the global property described before defined by invariant *inv*13 in Fig. 6.7(a), some other properties of the system are added:

1. The trains (variable *trns*) circulate in tracks. The current route based on current positions of switches is defined by *next*: a partial injection $CDV \rightarrowtail\!\!\!\!\!\!\!+\; CDV$. *next* is a subset of *net* (*inv*1) preserving the transitive closure property as described by theorem *thm*1,*thm*2 and does not have loops (*thm*3). Sections occupied by trains are represented by variable *occp*. These sections also preserve the transitive closure property as seen by *thm*4.
2. A train occupies at least one section and the section corresponding to the beginning and end of the train is represented by variables *occpA* and *occpZ* respectively. Note that *next* does not indicate the direction that a train is moving in: the direction can be *occpA* to *occpZ* or *occpZ* to *occpA*. These two variables point to the same section if the train only occupies one section (*inv*11).

The system proceeds as follows: trains modelled in the system circulate by entering and leaving sections (events *enterCDV* and *leaveCDV* in Fig. 6.7(b)), ensuring that the next section is not occupied (*grd*9 in *enterCDV*) and updating all the sections occupied by the train (*act*1 and *act*2 in both events). At this abstract level, event *modifyTrain* modifies a train defining the set of occupied sections for a train t. A train changes speed, brakes or stops braking in events *changeSpeed*, *brake* and *stopBraking*. When event *brake* occurs, train t is added to a set of braking trains (variable *braking*). Variable *next* represents the current connectivity of the trail based on the positions of switches. The current connectivity can be updated by changing convergent/divergent switches in events *switchChangeDiv* and *switchChangeCnv* as seen in Fig. 6.7(b).

6.4 First Refinement: *MetroSystem_M1*

MetroSystem_M1 refines *MetroSystem_M0*, incorporating the communication layer and an emergency button for each train. The communication work as follows: a message is sent from the tracks, stored in a buffer and read in the recipient train. The properties to be preserved for this refinement are:

1. Messages are exchanged between trains and tracks. If a train intends to move to an occupied section, track sends a message negating the access to that section and the train should brake.
2. As part of the safety requirements, all trains have an emergency button.
3. While the emergency button is enabled, the train continues braking and cannot speed up.

Now the system proceeds as follows: trains that enter and leave sections must take into account the messages sent by the tracks. Therefore events corresponding to enter and leaving section need to be strengthened to preserve this property. The requirement concerning the space required for the train to halt is a simplification of a real metro system and could require adjustments to replicate the real behaviour (for instance the occupied sections of a train could be defined as the sum of the sections directly occupied by the train and the sections indirectly occupied by the same train that correspond to the sections required for the train to halt). Nevertheless in real systems, trains can have in-built a way to detect the required space to break. For instance in Communication Based Train Control (CBTC [97, 72]) systems, that is called the *stopping distance downstream*.

The messages are represented by variables *tmsgs* that stores the messages (buffer) sent from the tracks and *permit* that receives the message in the train, expressing property 1. At this level, the messages are just boolean values assessing if a train can move to the

```
machine MetroSystem_M0 sees MetroSystem_C0

variables next // Currrent connectivity based on switch positions
          trns // Set of trains on network
          occp // Occupancy function for section
          occpA // Initial cdv occupied by train
          occpZ // Final   cdv occupied by train
          braking speed

invariants
  @inv1 next ⊆ net
  @inv2 next ∈ CDV ⤔ CDV
  @inv3 trns ⊆ TRAIN
  @inv4 occp ∈ CDV ↔ trns
  @inv5 occpA ∈ trns → CDV
  @inv6 ∀tt·(tt∈trns ⇒ occpA(tt) ∈ occp∼[{tt}])
  @inv7 occpZ ∈ trns → CDV
  @inv8 ∀tt·(tt∈trns ⇒ occpZ(tt) ∈ occp∼[{tt}])
  @inv9 braking ⊆ trns
  @inv10 speed ∈ trns → ℕ
  @inv11 ∀tt·tt∈trns ∧ card(occp∼[{tt}])>1 ⇒ occpA(tt) ≠ occpZ(tt)
  @inv12 finite(occp∼)
  @inv13 ∀t1,t2·t1∈trns ∧ t2∈trns ∧ t1≠t2 ⇒ occp∼[{t1}]∩occp∼[{t2}]=∅
  theorem @thm1 next ∈ cdvfn
  theorem @thm2 tcl(next) = next ∪ (next;tcl(next)) // tcl(next) is a fixed
point
  theorem @thm3 (∀s·s⊆next∼[s]⇒s=∅)⇒tcl(next)∩(CDV ◁ id)=∅ // next has no
loops
  theorem @thm4 ∀tt,s·tt∈trns ∧ s ⊆ next▷occp∼[{tt}] ⇒ tcl(s) = s ∪
(s;tcl(s))
```

(a) Variables, invariants in *MetroSystem_M0*

```
event enterCDV
  any t1 c1 c2
  where
    @grd1 t1 ∈ trns
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd5 c1 = occpZ(t1)
    @grd6 c1∈dom(next)
    @grd7 c2 = next(occpZ(t1))
    @grd8 ∀tt·tt∈trns ∧ card((occp ∪ {c2 ↦ t1})∼[{tt}])>1
        ⇒ (occpZ⩤{t1 ↦ c2})(tt) ≠ occpA(tt)
    @grd9 c2 ∉ dom(occp)
  then
    @act1 occpZ(t1) ≔ c2
    @act2 occp≔occp ∪ { c2 ↦ t1}
end

event leaveCDV
  any t1 c1 c2
  where
    @grd1 t1 ∈ trns
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd5 c1∈dom(next)
    @grd6 c1=occpA(t1)
    @grd7 c2=next(c1)
    @grd8 occpA(t1)≠occpZ(t1)
    @grd9 c2 ∈ (occp\{c1↦t1})∼[{t1}]
    @grd10 ∀tt·tt∈trns ∧ card(((occp \ {c1 ↦ t1}))∼[{tt}])>1
         ⇒ (occpA⩤{t1 ↦ c2})(tt)≠occpZ(tt)
  then
    @act1 occpA(t1)≔c2
    @act2 occp ≔ occp\{c1↦t1}
end

event changeSpeed
  any t1 s1
  where
    @grd1 t1 ∈ trns
    @grd2 s1 ∈ ℕ
    @grd3 t1∈ braking ⇒ s1<speed(t1)
  then
    @act1 speed(t1) ≔ s1
end

event brake
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈trns\braking
  then
    @act1 braking≔braking∪ {t1}
end

event stopBraking
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈braking
  then
    @act1 braking≔braking\{t1}
end

event switchChangeDiv
  any ac c1 c2
  where
    @grd1 ac ∈ div_aig_cdv
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd8 c2 ∉ ran (next)
    @grd4 (ac ↦ c1) ∈ next
    @grd5 (ac ↦ c2) ∈ net
    @grd6 c1 ≠ c2
    @grd7 ac ∉ dom(occp)
  then
    @act1 next ≔ next ⩤ {ac ↦ c2}
end

event switchChangeCnv
  any ac c1 c2
  where
    @grd1 ac ∈ cnv_aig_cdv
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd8 c2 ∉ dom (next)
    @grd4 (c1 ↦ ac) ∈ next
    @grd5 (c2 ↦ ac) ∈ net
    @grd6 c1 ≠ c2
    @grd7 ac ∉ dom (occp)
  then
    @act1 next ≔ ({c1}⩤next) ∪ {c2 ↦ ac}
end

event addTrain
  any t oc
  where
    @grd1 t ∈ TRAIN\trns
    @grd2 oc ∈ CDV
    @grd3 oc ∉ dom(occp)
  then
    @act1 trns≔trns ∪{t}
    @act2 speed(t)≔0
    @act3 occpA(t) ≔ oc
    @act4 occpZ(t) ≔ oc
    @act5 occp ≔ occp ∪ {oc↦t}
end

event modifyTrain
  any t ocA oc
  where
    @grd1 ocA∈dom(next)
    @grd2 t ∈ trns
    @grd3 oc ⊆ CDV
    @grd4 ocA ∈ oc
    @grd5 oc ∩ dom(occp)=∅
    @grd6 finite(oc)
    @grd7 occpZ(t)∈dom(next)
    @grd8 card(oc)=0 ⇒ocA = occpZ(t)
    @grd9 card(oc)≥1
       ⇒ occpZ(t) ≠ ocA ∧ next(occpZ(t))∈oc
    @grd10 next(ocA)∉oc
  then
    @act1 occpA(t) ≔ ocA
    @act2 occp ≔ occp ∪ (oc×{t})
 end
```

(b) Events of *MetroSystem_M0*

FIGURE 6.7: Variables, invariant and events of *MetroSystem_M0*

following section (check if the section is free): if TRUE the train can move; if FALSE the next section is occupied and the train should brake. New event *sendTrainMsg* models the message sending. The reception of messages is modelled in event *recvTrainMsg* where the message is stored in *permit* before *tmsgs* is reset. The guards of event *brake* are strengthened to allow a train to brake when $permit(t)$ = FALSE or when the emergency button is activated (guard *grd*3 in Fig. 6.8(b)). Property 2 is expressed by adding variable *emergency_button*. The activation/deactivation of the emergency button occurs in the new event *toggleEmergencyButton*. Property 3 is expressed by guard *grd*3 in event *stopBraking*: a train can only stop braking if the emergency button is not enabled.

```
machine MetroSystem_M1 refines MetroSystem_M0  sees MetroSystem_C0

variables next trns occp occpA occpZ
          braking speed
          tmsgs permit emergency_button

invariants
  @inv1 tmsgs ∈ trns → ℙ(BOOL)
  @inv2 permit ∈ trns → BOOL
  @inv3 emergency_button ∈ trns → BOOL
```

(a) Variables and invariants in *MetroSystem_M1*

```
event brake refines brake
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈trns\braking
    @grd3 permit(t1) = FALSE
          ∨ emergency_button(t1)=TRUE
  then
    @act1 braking≔braking ∪ {t1}
end

event stopBraking refines stopBraking
  any t1
  where
    @grd1 t1 ∈ TRAIN
    @grd2 t1∈braking
    @grd3 emergency_button(t1) = FALSE
  then
    @act1 braking≔braking\{t1}
end
```

```
event sendTrainMsg
    any t1
    where
      @grd1 t1 ∈ trns
      @grd2 tmsgs(t1) = ∅
    then
      @act1 tmsgs(t1)≔ {bool(
            occpZ(t1)∈dom(next)
            ∧next(occpZ(t1)) ∉ dom(occp))}
  end

event recvTrainMsg
    any t1 bb
    where
      @grd1 t1 ∈ trns
      @grd2 bb ∈ tmsgs(t1)
    then
      @act1 permit(t1) ≔ bb
      @act2 tmsgs(t1) ≔ ∅
  end
```

```
event toggleEmergencyButton
  any t value
  where
    @guard t ∈ trns
    @guard1 value ∈ BOOL
  then
    @act1 emergency_button(t)≔ value
end
```

(b) Some events of *MetroSystem_M1*

FIGURE 6.8: Excerpt of *MetroSystem_M1*

6.5 Second Refinement: *MetroSystem_M2*

In this refinement, we introduce train doors and platforms where the trains can stop to load/unload. When stopped, a train can open its doors. The properties to be preserved are:

1. If a train door is opened, then the train is stopped. In contrast, if the train is moving, then its doors are closed.

2. If a train door is opened, that either means that the train is in a platform or there was an emergency and the train had to stop suddenly.

3. A train door cannot be allocated to different trains.

We consider that platforms are represented by single sections. A train is in a platform if one of the occupied sections correspond to a platform. Doors are introduced as illustrated in Fig. 6.9(a) by sets $DOOR$ and their states are represented by $DOOR\_STATE$. Variables *door* and *door_state* represent the train doors and their current states as seen in Fig. 6.9(b): all trains have allocated a subset of doors ($inv2$). Several invariants are introduced to preserve the desired properties: property 1 is defined by invariants $inv4$ and $inv5$; property 2 is defined by invariant $inv7$; property 3 is stated by $inv3$; theorem $thm1$ is used for proving purposes (if no doors are open, then all doors are closed).

To preserve $inv5$, the guards of *changeSpeed* (in Fig. 6.8(b)) are strengthened by $grd4$ ensuring that whilst the train is moving, the train doors are closed. Also events that model entering and leaving sections are affected, with the introduction of a similar guard ($grd11$ in $leaveCDV$). Adding/removing train doors is modelled in events $addDoorTrain$ and $removeDoorTrain$ respectively: to add/remove a door, the respective train must be stopped. If the train is stopped and either one of the occupied sections corresponds to a platform or the emergency button is activated (guard $grd3$), doors can be opened as seen in event $openDoor$. For safety reasons, event $toggleEmergencyButton$ is strengthened by guard $grd3$ to activate the emergency button whenever doors are open and the train is not in a platform.

6.6 Third Refinement and First Decomposition: *MetroSystem_M3*

This refinement does not introduce new details to the model. It corresponds to the preparation step before the decomposition. We want to implement a three way shared event decomposition and therefore we need to separate the variables that will be allocated to each sub-component. In particular for exchanged messages between the sub-components, the protocol will work as follows: messages are sent from *Track* and stored in the *Middleware*. After receiving the message, the *Middleware* forwards it to the corresponding *Train*. *Train* reads the message and processes it according to the content. This protocol allows a separation between *Train* and *Track* with the *Middleware* working as a bridge between these two sub-components.

The decomposition follows the steps described in Sect. 5.5. Variables are distributed according to Fig. 6.10. To avoid constraints during the decomposition process, predicates and assignments containing variables that belong to different sub-components are rearranged in this refinement step.

```
context MetroSystem_C1 extends MetroSystem_C0

constants OPEN CLOSED PLATFORM

sets DOOR_STATE DOOR

axioms
  @axm1 partition(DOOR_STATE, {OPEN}, {CLOSED})
  @axm2 PLATFORM ⊆ CDV
end
```

(a) Context *MetroSystem_C1*

```
machine MetroSystem_M2 refines MetroSystem_M1  sees MetroSystem_C1

variables next trns occp occpA occpZ
          braking speed tmsgs permit
          door door_state emergency_button

invariants
  @inv1 door_state ∈ DOOR → DOOR_STATE
  @inv2 door ∈ trns → ℙ(DOOR)
  @inv3 ∀t1,t2·t1 ∈ dom(door) ∧ t2 ∈ dom(door) ∧ t1 ≠t2
        ⇒ door(t1) ∩ door(t2) = ∅
  @inv4 ∀t·t ∈ dom(door) ⇒(∃d·d⊆door(t) ∧ door_state[d]={OPEN}
        ⇒ speed(t)=0)
  @inv5 ∀t·t ∈ dom(door) ∧ speed(t) > 0
        ⇒ door(t) ⊆ door_state~[{CLOSED}]
  @inv6 ∀t,d·t ∈ dom(door) ∧ d ∈ door(t) ∧ PLATFORM ∩ occp~[{t}]≠∅
        ⇒ door_state(d) ∈ {OPEN, CLOSED}
  @inv7 ∀t·t ∈ dom(door) ∧ door(t) ∩ door_state~[{OPEN}] ≠ ∅
        ⇒ PLATFORM ∩ occp~[{t}]≠∅ ∨ emergency_button(t) = TRUE
  theorem @thm1 ∀t·t ∈ dom(door) ∧ door(t) ∩ door_state~[{OPEN}] =∅
        ⇒ door(t)⊆door_state~[{CLOSED}]
```

(b) Variables, invariants in *MetroSystem_M2*

```
event toggleEmergencyButton
refines toggleEmergencyButton
  any t value
  where
    @grd1 t ∈ dom(door)
    @grd2 value ∈ BOOL
    @grd3 door(t) ∩ door_state~[{OPEN}] ≠ ∅
          ∧ PLATFORM ∩ occp~[{t}]=∅
          ⇒ value = TRUE
  then
    @act1 emergency_button(t)≔ value
end

event openDoor
  any t ds
  where
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd3 occp~[{t}] ∩ PLATFORM ≠ ∅
          ∨ emergency_button(t) = TRUE
    @grd4 ds ⊆ door(t)
    @grd5 ∃d·d∈ds⇒door_state(d)=CLOSED
    @grd6 ds≠∅
  then
    @act1 door_state≔ door_state  (ds×{OPEN})
end

event closeDoor
  any t ds
  where
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd3 ds ⊆ door(t)
    @grd4 door_state[ds]={OPEN}
    @grd5 ds≠∅
  then
    @act1 door_state≔ door_state  (ds×{CLOSED})
end

event addDoorTrain
  any t d
  where
    @grd1 t ∈ trns
    @grd2 d ⊆ DOOR
    @grd3 ∀tr·tr∈dom(door) ∧ tr≠t
          ∧ door(tr)≠∅⇒d∩door(tr)=∅
    @grd5 speed(t)=0
    @grd7 d∩door(t)=∅
  then
    @act1 door(t)≔door(t)∪d
    @act2 door_state≔
            door_state(d×{CLOSED})
end

event removeDoorTrain
  any t d
  where
    @grd1 t ∈ dom(door)
    @grd2 d ⊆ DOOR
    @grd3 d ⊆ door(t)
    @grd4 door_state[d]={CLOSED}
    @grd5 speed(t)=0
  then
    @act1 door(t) ≔ door(t)\d
end

event leaveCDV refines leaveCDV
  any t1 c1 c2
  where
    @grd1 t1 ∈ dom(door)
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd5 c1∈dom(next)
    @grd6 c1=occpA(t1)
    @grd7 c2=next(c1)
    @grd8 occpA(t1)≠occpZ(t1)
    @grd9 c2 ∈ (occp\{c1↦t1})~[{t1}]
    @grd10 ∀tt·tt∈trns
    ∧ card(((occp \ {c1 ↦ t1}))~[{tt}])>1
    ⇒ (occpA{t1 ↦ c2})(tt)≠occpZ(tt)
    @grd11 door(t1)∩door_state~[{OPEN}]=∅
    @grd12 permit(t1)=TRUE
  then
    @act1 occpA(t1)≔c2
    @act2 occp ≔ (occp\{c1↦t1})
end

event changeSpeed refines changeSpeed
  any t1 s1
  where
    @grd1 t1 ∈ dom(door)
    @grd2 s1 ∈ ℕ
    @grd3 t1∈ braking ⇒ s1<speed(t1)
    @grd4 door(t1)∩door_state~[{OPEN}]=∅
  then
    @act1 speed(t1) ≔ s1
end
```

(c) Some events of *MetroSystem_M2*

Figure 6.9: Excerpt of *MetroSystem_M2*

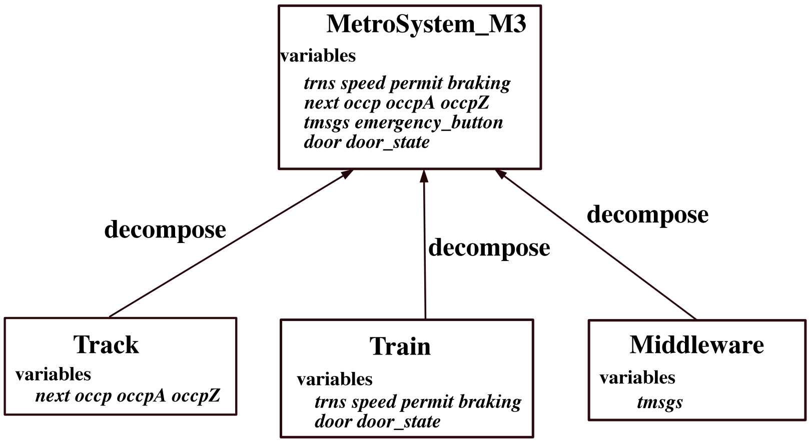


FIGURE 6.10: *MetroSystem_M3* (shared event) decomposed into *Track*, *Train* and *Middleware*

Some guards need to be rewritten in the refined events. For instance, guard $grd10$ in event $leaveCDV$ needs to be rewritten in order not to include both variables $trns$ (sub-component $Train$) and $occp$ (sub-component $Track$). Therefore it is changed from:

$$\forall tt\cdot tt \in \mathbf{trns} \wedge card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \lhd\!\!- \{t1 \mapsto c2\})(tt) \neq occpA(tt)$$

to:

$$\forall tt\cdot tt \in \mathbf{dom(occpZ)} \wedge card((occp \cup \{c2 \mapsto t1\})^{-1}[\{tt\}]) > 1 \Rightarrow (occpZ \lhd\!\!- \{t1 \mapsto c2\})(tt) \neq occpA(tt) \text{ (Fig. 6.11).}$$

Both predicates represent the same property since $trns$ corresponds to the domain of variable $occpZ$ (see $inv7$ in Fig. 6.7(a)). In Fig. 6.11, the original guard $grd3$ in $toggleEmergencyButton$ is rewritten to separate variables $occp$ and $door$. In this case, an additional parameter $occpTrns$ representing the variable $occp$ is added ($grd4$). This additional parameter will represent the value passing between the resulting decomposed events: parameter $occpTrns$ is written the value of $occp$ and afterwards it is read in guard $grd3$. Similarly guard $grd4$ in event $openDoor$ must not include variables $occp$ and $emergency\_button$ and consequently parameter $occpTrns$ is added.

Sub-components $Train$, $Track$ and $Middleware$ are described in the following sections. The composed machine corresponding to the defined decomposition can be seen in Fig. 6.12 where it is illustrated how the original events are decomposed.

6.6.1 Machine *Track*

Machine *Track* contains the properties concerning the sections in the metro system. Events corresponding to entering, leaving tracks and changing switch positions are part of this sub-component resulting from the variables allocation for this sub-component: $next$, $occp$, $occpA$ and $occpZ$. Event $sendTrainMsg$ is also added since the messages are

```
event toggleEmergencyButton
refines toggleEmergencyButton
   any t value occpTrns
   where
     @grd1 t ∈ dom(door)
     @grd2 value ∈ BOOL
     @grd3 door(t) ∩ door_state∼[{OPEN}] ≠ ∅
           ∧ PLATFORM ∩ occpTrns=∅
           ⇒ value = TRUE
     @grd4 occpTrns = occp∼[{t}]
   then
     @act1 emergency_button(t)≔ value
 end

 event openDoor refines openDoor
   any t occpTrns ds
   where
     @grd1 t ∈ dom(door)
     @grd2 speed(t) = 0
     @grd3 occpTrns = occp∼[{t}]
     @grd4 occpTrns ∩ PLATFORM ≠ ∅
           ∨ emergency_button(t) = TRUE
     @grd5 ds ⊆ door(t)
     @grd6 ∃d·d∈ds⇒door_state(d)=CLOSED
     @grd7 ds≠∅
   then
     @act1 door_state≔ door_state ⩤ (ds×{OPEN})
 end
```

```
event leaveCDV refines leaveCDV
  any t1 c1 c2
  where
    @grd1 t1 ∈ dom(door)
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd5 c1∈dom(next)
    @grd6 c1=occpA(t1)
    @grd7 c2=next(c1)
    @grd8 occpA(t1)≠occpZ(t1)
    @grd9 c2 ∈ (occp\{c1↦t1})∼[{t1}]
    @grd10 ∀tt·tt∈dom(occpZ)
           ∧ card(((occp \ {c1 ↦ t1}))∼[{tt}])>1
           ⇒ (occpA⩤{t1 ↦ c2})(tt)≠occpZ(tt)
    @grd11 door(t1)∩door_state∼[{OPEN}]=∅
    @grd13 permit(t1)=TRUE
  then
    @act1 occpA(t1)≔c2
    @act2 occp ≔ (occp\{c1↦t1})
 end
```

FIGURE 6.11: Preparation step before decomposition of *MetroSystem_M3*

sent from the tracks as seen in Fig. 6.13. The original events *toggleEmergencyButton* and *openDoor* require *occp* in their guards. Consequently part of these original events are included in this sub-component.

Note that the invariants defining the variables may change: in *MetroSystem_M1* variable *occp* is defined as $occp \in CDV \leftrightarrow trns$ (*inv4* in Fig. 6.7(a)); in *Track* is $occp \in CDV \leftrightarrow TRAIN$ (which is the same as theorem *typing_occp* : $occp \in \mathbb{P}(CDV \times TRAIN)$ in Fig. 6.13). This is a consequence of the variable partition since *trns* is not part of *Track* and therefore the *occp* relation is updated with *trns*'s type: $TRAIN$ (cf. *inv*3 in Fig. 6.7(a)). Variables *occpA* and *occpZ* are subject to the same procedure where the original invariant is a total function $trns \rightarrow CDV$ and in the sub-component both become $\mathbb{P}(TRAIN \times CDV)$. The sub-components invariants are derived from the different initial abstract models (cf. their labels in Fig. 6.13). Invariants that only restrain the sub-component variables are automatically included although additional ones can be added manually.

6.6.2 Machine *Train*

Machine *Train* models the trains in the metro system. Trains entering/leaving a section, modelled by events *enterCDV* and *leaveCDV* are part of this sub-component, in spite of the decomposed events do not execute any actions (see Fig. 6.14(b)). The interaction with sub-component *Track* occurs through parameters $t1$, $c1$ and $c2$ (see events *Track.leaveCDV* in Fig. 6.13). Variables *door* and *door_state* are part of this sub-component and consequently the events that modify these variables: *openDoor* and *closeDoor*. Moreover, since the emergency button is part of a train, the respective variable *emergencyButton* (and the modification event *toggleEmergencyButton*) is also included in this sub-component. Event *recvTrainMsg* receives messages sent to the

```
COMPOSED MACHINE MetroSystem_M3_cmp
REFINES MetroSystem_M3
INCLUDES
    Track   Train   Middleware
EVENTS
    addTrain refines addTrain
        Combines Events Train.addTrain || Middleware.addTrain ||Track.addTrain
    modifyTrain refines modifyTrain
        Combines Events Train.modifyTrain ||Track.modifyTrain
    sendTrainMsg refines sendTrainMsg
        Combines Events Track.sendTrainMsg || Middleware.sendTrainMsg
    recvTrainMsg refines recvTrainMsg
        Combines Events Train.recvTrainMsg || Middleware.recvTrainMsg
    changeSpeed refines changeSpeed
        Combines Events Train. changeSpeed
    brake refines brake
        Combines Events Train.brake
    stopBraking refines stopBraking
        Combines Events Train.stopBraking
    enterCDV refines enterCDV
        Combines Events Train.enterCDV || Track.enterCDV
    leaveCDV refines leaveCDV
        Combines Events Train.leaveCDV || Track.leaveCDV
    openDoor refines openDoor
        Combines Events Train.openDoor || Track.openDoor
    closeDoor refines closeDoor
        Combines Events Train.closeDoor
    toggleEmergencyButton refines toggleEmergencyButton
        Combines Events Train.toggleEmergencyButton || Track.toggleEmergencyButton
    addDoorTrain refines addDoorTrain
        Combines Events Train.addDoorTrain
    removeDoorTrain refines removeDoorTrain
        Combines Events Train.removeDoorTrain
    switchChangeDiv refines switchChangeDiv
        Combines Events Track.switchChangeDiv
    switchChangeCnv refines switchChangeCnv
        Combines Events Track.switchChangeCnv
END
```

FIGURE 6.12: Composed machine tool view corresponding to *MetroSystem_M3* decomposition

trains and the content is stored in the variable *permit*. Although variable *permit* is set based on the content of the messages exchanged between *Train* and *Track*, that variable is read by trains. This is the reason why it is allocated to this sub-component. The events that change the speed of the train are also included in this sub-component: *brake*, *stopBraking*, *changeSpeed* due to variables *speed* and *braking* as depicted in Fig. 6.14.

6.6.3 Machine *Middleware*

Finally the communication layer in modelled by *Middleware* as seen in Fig. 6.15. *Middleware* bridges *Track* and *Trains*, by receiving messages (*sendTrainMsg*) from the tracks and delivering to the trains (*recvTrainMsg*). Variable *tmsgs* is used as a buffer.

Benefiting from the monotonicity of the shared event approach, the resulting sub-components can be further refined. Following Fig. 6.3, *Train* is refined as described

```
machine Track sees MetroSystem_C1

variables next occp occpA occpZ

invariants
  theorem @typing_occpZ occpZ ∈ ℙ(TRAIN × CDV)
  theorem @typing_occp occp ∈ ℙ(CDV × TRAIN)
  theorem @typing_next next ∈ ℙ(CDV × CDV)
  theorem @typing_occpA occpA ∈ ℙ(TRAIN × CDV)
  @MetroSystem_M0_inv1 next ⊆ net
  @MetroSystem_M0_inv2 next ∈ CDV ⤔ CDV
  @MetroSystem_M0_inv12 finite(occp~)

  event sendTrainMsg
    any t1 bb
    where
      @typing_t1 t1 ∈ TRAIN
      @typing_bb bb ∈ BOOL
      @grd3 bb = bool (occpZ(t1)∈dom(next)
            ∧ next(occpZ(t1))∉dom(occp) )
  end

   event enterCDV
     any t1 c1 c2
     where
       @typing_t1 t1 ∈ TRAIN
       @grd2 c1 ∈ CDV
       @grd3 c2 ∈ CDV
       @grd5 c1 = occpZ(t1)
       @grd6 c1∈dom(next)
       @grd7 c2 = next(occpZ(t1))
       @grd8 ∀tt·tt∈dom(occpZ)
             ∧ card((occp ∪ {c2 ↦ t1})~[{tt}])>1
             ⇒ (occpZ⩤{t1 ↦ c2})(tt) ≠ occpA(tt)
       @grd9 c2∉dom(occp)
     then
       @act1 occpZ(t1) ≔ c2
       @act2 occp≔occp ∪ { c2 ↦ t1}
   end

event openDoor
    any t occpTrns ds
    where
      @typing_t t ∈ TRAIN
      @typing_occpTrns occpTrns ∈ ℙ(CDV)
      @typing_ds ds ∈ ℙ(DOOR)
      @grd3 occpTrns = occp~[{t}]
      @grd7 ds≠∅
  end

 event leaveCDV
   any t1 c1 c2
   where
     @typing_t1 t1 ∈ TRAIN
     @grd2 c1 ∈ CDV
     @grd3 c2 ∈ CDV
     @grd5 c1∈dom(next)
     @grd6 c1=occpA(t1)
     @grd7 c2=next(c1)
     @grd8 occpA(t1)≠occpZ(t1)
     @grd9 c2 ∈ (occp\{c1↦t1})~[{t1}]
     @grd10 ∀tt·tt∈dom(occpZ)
            ∧ card(((occp \ {c1 ↦ t1}))~[{tt}])>1
            ⇒ (occpA⩤{t1 ↦ c2})(tt)≠occpZ(tt)
   then
     @act1 occpA(t1)≔c2
     @act2 occp ≔ (occp\{c1↦t1})
 end

event toggleEmergencyButton
  any t value occpTrns
  where
    @typing_t t ∈ TRAIN
    @typing_occpTrns occpTrns ∈ ℙ(CDV)
    @grd2 value ∈ BOOL
    @grd4 occpTrns = occp~[{t}]
end
```

FIGURE 6.13: Excerpt of *Track*

in the following section.

6.7 Refinement of *Train*: *Train_M1*

In Train_M1, carriages are introduced as parts of a train. Each carriage has an individual alarm that when activated, triggers the train alarm (enables the emergency button of the train). Each train has a limited number of carriages. Each carriage has a set of doors and the sum of carriage doors corresponds to the doors of a train. The properties to be preserved are:

1. There is a limit to the number ($MAX\_NUMBER\_CARRIAGE$) of carriages per train.
2. Whenever a carriage alarm is activated, then the emergency button of that same train is activated.
3. The sum of carriage doors corresponds to the doors of a train.

The definition of these requirements require the introduction of some static elements like a carrier set *CARRIAGE*, constants *MAX_NUMBER_CARRIAGE* and *DOOR_CARRIAGE* (function between *DOOR* and *CARRIAGE*). The latter is defined as a constant because the number of doors in a carriage does not change. Context

```
machine Train sees MetroSystem_C1

variables trns speed permit braking emergency_button door_state door

invariants
  theorem @typing_trns trns ∈ ℙ(TRAIN)
  theorem @typing_door_state door_state ∈ ℙ(DOOR × DOOR_STATE)
  theorem @typing_braking braking ∈ ℙ(TRAIN)
  theorem @typing_speed speed ∈ ℙ(TRAIN × ℤ)
  theorem @typing_permit permit ∈ ℙ(TRAIN × BOOL)
  theorem @typing_door door ∈ ℙ(TRAIN × ℙ(DOOR))
  theorem @typing_emergency_button emergency_button ∈ ℙ(TRAIN × BOOL)
  @MetroSystem_M0_inv3 trns ⊆ TRAIN
  @MetroSystem_M0_inv9 braking ⊆ trns
  @MetroSystem_M0_inv10 speed ∈ trns ⟶ ℕ
  @MetroSystem_M1_inv2 permit ∈ trns ⟶ BOOL
  @MetroSystem_M1_inv7 emergency_button ∈ trns ⟶ BOOL
  @MetroSystem_M2_inv1 door_state ∈ DOOR ⟶ DOOR_STATE
  @MetroSystem_M2_inv2 door ∈ trns ⟶ ℙ(DOOR)
  @MetroSystem_M2_inv3 ∀t1,t2·t1 ∈ dom(door) ∧ t2 ∈ dom(door) ∧ t1 ≠t2 ⇒ door(t1) ∩ door(t2) = ∅
  @MetroSystem_M2_inv4 ∀t·t ∈ dom(door) ⇒(∃d·d⊆door(t) ∧ door_state[d]={OPEN} ⇒ speed(t)=0)
  @MetroSystem_M2_inv5 ∀t·t ∈ dom(door) ∧ speed(t) > 0 ⇒ door(t) ⊆ door_state~[{CLOSED}]
  theorem @MetroSystem_M2_thm1 ∀t·t ∈ dom(door) ∧ door(t) ∩ door_state~[{OPEN}] =∅
                               ⇒ door(t)⊆door_state~[{CLOSED}]
```

(a) Variables and invariants in *Train*

```
event recvTrainMsg
  any t1 bb
  where
    @typing_t1 t1 ∈ TRAIN
    @typing_bb bb ∈ BOOL
  then
    @act2 permit(t1)≔bb
end

event changeSpeed
  any t1 s1
  where
    @typing_t1 t1 ∈ TRAIN
    @typing_s1 s1 ∈ ℤ
    @grd1 s1 ∈ ℕ
    @grd2 t1 ∈ dom(door)
    @grd3 t1 ∈ braking ⇒ s1 < speed (t1)
    @grd4 door(t1) ∩ door_state~[{OPEN}] =∅
  then
    @act1 speed (t1) ≔ s1
end

event brake
  any t1
  where
    @typing_t1 t1 ∈ TRAIN
    @grd1 t1 ∈ trns\braking
    @grd2 t1 ∈ dom(emergency_button)
    @grd3 permit(t1) = FALSE
          ∨ emergency_button(t1)=TRUE
  then
    @act1 braking ≔ braking ∪ {t1}
end

event openDoor
  any t occpTrns ds
  where
    @typing_t t ∈ TRAIN
    @typing_occpTrns occpTrns ∈ ℙ(CDV)
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd4 occpTrns ∩ PLATFORM ≠ ∅
          ∨ emergency_button(t) = TRUE
    @grd5 ds ⊆ door(t)
    @grd6 ∃d·d∈ds⇒door_state(d)=CLOSED
    @grd7 ds≠∅
  then
    @act1 door_state≔ door_state ⩤ (ds×{OPEN})
end

event closeDoor
  any t ds
  where
    @typing_t t ∈ TRAIN
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ dom(door)
    @grd2 speed(t) = 0
    @grd3 ds ⊆ door(t)
    @grd4 door_state[ds]={OPEN}
    @grd5 ds≠∅
  then
    @act1 door_state≔ door_state ⩤ (ds×{CLOSED})
end

event toggleEmergencyButton
  any t value occpTrns
  where
    @typing_t t ∈ TRAIN
    @typing_occpTrns occpTrns ∈ ℙ(CDV)
    @grd1 t ∈ dom(door)
    @grd2 value ∈ BOOL
    @grd3 door(t) ∩ door_state~[{OPEN}] ≠ ∅
          ∧ PLATFORM ∩ occpTrns=∅
          ⇒ value = TRUE
  then
    @act1 emergency_button(t)≔ value
end

event addDoorTrain
  any t d
  where
    @typing_d d ∈ ℙ(DOOR)
    @typing_t t ∈ TRAIN
    @grd1 t ∈ trns
    @grd2 d ⊆ DOOR
    @grd3 ∀tr·tr∈dom(door) ∧ tr≠t
          ∧ door(tr)≠∅ ⇒ d∩door(tr)=∅
    @grd5 speed(t)=0
    @grd7 d∩door(t)=∅
  then
    @act1 door(t)≔door(t)∪d
    @act2 door_state≔door_state⩤(d×{CLOSED})
end

event removeDoorTrain
  any t d
  where
    @typing_d d ∈ ℙ(DOOR)
    @typing_t t ∈ TRAIN
    @grd1 t ∈ dom(door)
    @grd2 d ⊆ DOOR
    @grd3 d ⊆ door(t)
    @grd4 door_state[d]={CLOSED}
    @grd5 speed(t)=0
  then
    @act1 door(t) ≔ door(t)\d
end

event leaveCDV
  any t1 c1 c2
  where
    @typing_t1 t1 ∈ TRAIN
    @grd1 t1 ∈ dom(door)
    @grd2 c1 ∈ CDV
    @grd3 c2 ∈ CDV
    @grd4 speed(t1)>0
    @grd11 door(t1)∩door_state~[{OPEN}]=∅
    @grd12 permit(t1)=TRUE
end
```

(b) Some events of *Train*

FIGURE 6.14: Excerpt of *Train*

Train_C2 is depicted in Fig. 6.16(a). Several variables are added such as *train_carriage* relating carriages with trains and *carriage_alarm* that is a total function between *CARRIAGE* and BOOL, illustrated in Fig. 6.16(b). Property 1 is expressed by invariant *inv6* stating that trains have a maximum of *MAX_NUMBER_CARRIAGE* carriages. Property 2 is defined in *inv7* as seen in Fig. 6.16(b). Events *activateEmergencyCarriageButton* and *deactivateEmergencyTrainButton* refine abstract event *toggleEmergencyButton*: the first event enables a carriage alarm and consequently enables the emergency button of the train; the later occurs when the emergency button of a train is active

```
machine Middleware sees MetroSystem_C1

variables tmsgs

invariants
  theorem @typing_tmsgs tmsgs ∈ ℙ(TRAIN × ℙ(BOOL))

events
  event INITIALISATION
    then
      @act1 tmsgs ≔ ∅
  end

  event sendTrainMsg
    any t1 bb
    where
      @typing_t1 t1 ∈ TRAIN
      @typing_bb bb ∈ BOOL
      @grd1 t1 ∈ dom(tmsgs)
      @grd2 tmsgs(t1)=∅
    then
      @act1 tmsgs(t1) ≔ {bb}
  end

event recvTrainMsg
  any t1 bb
  where
    @typing_t1 t1 ∈ TRAIN
    @typing_bb bb ∈ BOOL
    @grd1 t1 ∈ dom(tmsgs)
    @grd2 bb ∈ tmsgs(t1)
  then
    @act1 tmsgs(t1)≔∅
end

event addTrain
  any t oc
  where
    @typing_t t ∈ TRAIN
    @grd1 oc ∈ CDV
  then
    @act6 tmsgs(t)≔∅
end
```

FIGURE 6.15: Machine *Middleware*

and corresponds to the deactivation of the last enabled carriage alarm which results in deactivating the emergency button; a new event *deactivateEmergencyCarriageButton* is added to model the deactivation of a carriage alarm when there is still another alarm enabled for the same train (guards $grd4$ and $grd5$). The allocation and removal of carriages (events *allocateCarriageTrain* and *removeCarriageTrain*) refine *addDoorTrain* and *removeDoorTrain* respectively. In these two events, the parameter d representing a set of doors, is replaced in the witness section by the doors of the added/removed carriage: $d = DOOR\_CARRIAGE^{-1}[\{c\}]$. We continue the refinement of *Train* in the following section.

6.8 Second Refinement of *Train*: *Train_M2*

In this refinement of *Train*, carriages requirements are added. We specify *carriage doors* instead of the more abstract *train doors*. As a consequence, variable *doors* is data refined and disappears. Each train contains two cabin carriages (type A) and two ordinary carriages (type B) allocated as follows: A+B+B+A. Only one of the two cabin carriages is set to be the *leader carriage* controlling the set of carriages and the moving direction. Trains have states defining if they are in *maintenance* or if they are being driven *manually* or *automatically*. More safety requirements are introduced: if the speed of a train exceeds the safety maximum speed, the emergency brake for that train must be activated. The abstract event representing the change of speed is refined by several concrete events and includes the behaviour of the system when a train is above the maximum speed. The properties to be preserved in this refinement are:

1. If a train is not in maintenance, then it must have the correct number of carriages and the leader carriage must be defined already. Consequently, this is a condition to be verified before the train can change speed.

```
context Train_C1 extends MetroSystem_C1

constants MAX_NUMBER_CARRIAGE
          DOOR_CARRIAGE

sets CARRIAGE

axioms
  @axm1 MAX_NUMBER_CARRIAGE ∈ ℕ1
  @axm2 DOOR_CARRIAGE ∈ DOOR→CARRIAGE
  @axm3 ∀c·c∈ran(DOOR_CARRIAGE)
                ⇒DOOR_CARRIAGE∼[{c}]≠∅
end
```

(a) Context *Train_C1*

```
machine Train_M1 refines Train  sees Train_C1

variables trns speed permit braking door_state door emergency_button
          train_carriage carriage_alarm

invariants
  @inv1 finite(trns)
  @inv2 carriage_alarm ∈ CARRIAGE → BOOL
  @inv3 train_carriage ∈ CARRIAGE ⇸ trns
  @inv4 finite(train_carriage)
  @inv5 finite(dom(train_carriage))
  @inv6 ∀t·t ∈ trns ⇒ card(train_carriage∼[{t}])≤MAX_NUMBER_CARRIAGE
  @inv7 ∃c·(c ∈ dom(train_carriage) ∧ carriage_alarm(c) = TRUE
  ⇔ c ∈ dom(train_carriage) ∧ emergency_button(train_carriage(c))= TRUE)
  @inv8 ∀t·t∈dom(door) ⇒ door(t)=DOOR_CARRIAGE∼[train_carriage∼[{t}]]
```

(b) Variables and Invariants of *Train_M1*

```
event activateEmergencyCarriageButton
refines toggleEmergencyButton
  any c occpTrns
  where
    @grd1 occpTrns ∈ ℙ(CDV)
    @grd2 c ∈ dom(train_carriage)
    @grd3 carriage_alarm(c) = FALSE
  with
    @value value = TRUE
    @t t = train_carriage(c)
  then
    @act1 carriage_alarm(c) ≔ TRUE
    @act2 emergency_button(train_carriage(c)) ≔ TRUE
end

event deactivateEmergencyCarriageButton
  any c
  where
    @grd1 c ∈ dom(train_carriage)
    @grd2 emergency_button(train_carriage(c)) = TRUE
    @grd3 carriage_alarm(c) = TRUE
    @grd4 {c} ≠ (dom(carriage_alarm ▷ {TRUE})
                ∩ train_carriage∼[{train_carriage(c)}])
    @grd5 card(train_carriage▷{train_carriage(c)})>1
  then
    @act1 carriage_alarm(c)≔ FALSE
end

event deactivateEmergencyTrainButton
refines toggleEmergencyButton
  any c occpTrns
  where
    @grd1 occpTrns ∈ ℙ(CDV)
    @grd2 c ∈ dom(train_carriage)
    @grd3 emergency_button(train_carriage(c)) = TRUE
    @grd4 carriage_alarm(c) = TRUE
    @grd5 {c} = (dom(carriage_alarm ▷ {TRUE})
                 ∩ train_carriage∼[{train_carriage(c)}])
    @grd6 door(train_carriage(c))∩door_state∼[{OPEN}] = ∅
  with
    @value value = FALSE
    @t t = train_carriage(c)
  then
    @act1 carriage_alarm(c)≔ FALSE
    @act2 emergency_button(train_carriage(c)) ≔ FALSE
end

event alocateCarriageTrain refines addDoorTrain
  any c t
  where
    @grd1 c ∈ CARRIAGE\dom(train_carriage)
    @grd2 carriage_alarm[{c}]= {FALSE}
    @grd3 ∀tr·tr∈dom(door) ∧ tr≠t ∧ door(tr)≠∅
          ⇒ DOOR_CARRIAGE∼[{c}]∩door(tr)=∅
    @grd4 t ∈ trns
    @grd5 emergency_button(t) = FALSE
    @grd6 finite(train_carriage∼[{t}])
    @grd7 card(dom(train_carriage ▷ {t}))
                          <MAX_NUMBER_CARRIAGE
    @grd8 speed(t)=0
    @grd9 DOOR_CARRIAGE∼[{c}] ∩ door(t)=∅
  with
    @d d=(DOOR_CARRIAGE∼[{c}])
  then
    @act1 train_carriage(c)≔ t
    @act2 door(t)≔door(t) ∪ DOOR_CARRIAGE∼[{c}]
    @act3 door_state≔
      door_state⩤(DOOR_CARRIAGE∼[{c}]×{CLOSED})
end

event removeCarriageTrain refines removeDoorTrain
  any c t
  where
    @grd1 t ∈ dom(door)
    @grd2 c↦t ∈ train_carriage
    @grd3 carriage_alarm(c) = FALSE
    @grd4 emergency_button(t) = FALSE
    @grd5 speed(t)=0
    @grd6 DOOR_CARRIAGE∼[{c}]⊆door(t)
    @grd7 DOOR_CARRIAGE∼[{c}]≠∅
    @grd8 door_state[DOOR_CARRIAGE∼[{c}]]={CLOSED}
  with
    @d d = (DOOR_CARRIAGE∼[{c}])
  then
    @act1 train_carriage ≔ {c}⩤train_carriage
    @act2 door(t)≔door(t)\DOOR_CARRIAGE∼[{c}]
end
```

(c) Some events of *Train_M1*

FIGURE 6.16: Excerpt of machine *Train_M1*

2. If a train is in maintenance, then it must be stopped.

3. If the speed of a train exceeds the maximum speed, the emergency brake must be activated.

Figure 6.17(a) illustrates two new carrier sets: $SIDE$ corresponding to which side a carriage door or a platform is located (constants $LEFT$ or $RIGHT$) and $TRAIN\_STATE$ that defines the state of a train ($MAINTENANCE$, $MANUAL$ or $AUTOMATIC$). There are some new constants added as well: $CABIN\_CARRIAGE$ defined as a subset of $CARRIAGE$, $NUMBER\_CABIN\_CARRIAGE$ defining the number of cabin carriages allowed per train, $DOOR\_SIDE$ defined as a total function between $DOOR$ and $SIDE$ representing which side a door is located, MAX_SPEED defining the upper speed limit for running a train before the activation of the emergency brake and $PLATFORM\_SIDE$ defining the side of a platform.

Figure 6.17 shows *Train_M2* where several new variables are introduced: $leader\_carriage$ defining the leader carriage for a train ($inv6$), $trns\_state$ defining the state of a train ($inv8$), $emergency\_brake$ that defines which trains have the emergency brake activated ($inv11$) and $carriage\_door\_state$ defining the state of the carriage doors ($inv15$). Moreover $door\_train\_carriage$ defines the train doors based on the carriages ($inv2$, $inv3$ and $inv4$) and each door belongs to at most one train ($inv4$) although a train can have several doors ($inv2$). This variable refines $door$ that disappears in this refinement level, plus some gluing invariants: $inv1$, $inv5$ and theorem $thm2$ state that the range of $door$ for a train t is the same as the range of $door\_train\_carriage$ as long as t has doors.

Property 1 is expressed by $inv9$. Property 2 is expressed by $inv10$ and property 3 by $inv12$. $inv13$ and $inv14$ state that the doors in the domain of $door\_state$ are the same as the ones in carriage_door_state and therefore their state must match. Theorem $thm1$ relates the carriages doors with variables $door\_train\_carriage$ and $train\_carriage$. Theorem $thm3$ states that the domain of $carriage\_door\_state$ is a subset of the domain of $door\_state$ since both variables refer to the same set of doors.

New events are added defining the allocating of a leader carriage to a train (event $allocateLeaderCabinCarriageTrain$ in Fig. 6.17(c)). This event is enabled only if the train is in maintenance ($grd5$), already has the required number of carriages ($grd6$) but does not have a leader carriage yet ($grd7$). To deallocate the leader carriage in event $deallocateLeaderCabinCarriageTrain$, the train must be in maintenance. A train change state in event $modifyTrain$: to change to $MAINTENANCE$, the train must be stopped ($grd2$); for the other states, the number of cabin carriages must be $NUMBER\_CABIN\_CARRIAGE$ and a leading carriage have to be allocated already ($grd3$). Abstract event $changeSpeed$ is refined by four events: two to increase the speed ($increaseSpeed$ and $increaseMaxSpeed$ in Fig. 6.17(c)) and two to reduce the speed ($reduceSpeed$ and $reduceMaxSpeed$). If the speed of a train is increasing in a way that is superior to MAX_SPEED, event $increaseMaxSpeed$ is enabled and if it occurs, the emergency_brake is activated. If the current speed of a train is superior to MAX_SPEED but the new speed is decreasing in a way that is inferior to the maximum speed then the $emergency\_brake$ can be deactivated (event $reduceMaxSpeed$).

```
context Train_C2 extends Train_C1

constants CABIN_CARRIAGE NUMBER_CABIN_CARRIAGE
          LEFT RIGHT DOOR_SIDE PLATFORM_SIDE
          MAINTENANCE MANUAL AUTOMATIC MAX_SPEED

sets SIDE TRAIN_STATE

axioms
  @axm1 CABIN_CARRIAGE ⊆ CARRIAGE
  @axm2 NUMBER_CABIN_CARRIAGE ∈ ℕ1
  @axm3 DOOR_SIDE ∈ DOOR → SIDE
  @axm4 partition(SIDE, {LEFT}, {RIGHT})
  @axm5 partition(TRAIN_STATE, {MAINTENANCE},
        {MANUAL},{AUTOMATIC})
  @axm6 MAX_SPEED ∈ ℕ1
  @axm7 PLATFORM_SIDE ∈ PLATFORM → SIDE
  @axm8 finite(CABIN_CARRIAGE)
  @axm9 PLATFORM ≠∅
  @axm10 CABIN_CARRIAGE≠∅
  @axm11 CABIN_CARRIAGE⊆ ran(DOOR_CARRIAGE)
end
```

(a) Context *Train_C2*

```
variables trns speed permit braking door_state emergency_button train_carriage carriage_alarm
leader_carriage trns_state emergency_brake carriage_door_state door_train_carriage

invariants
  @inv1 ∀t·t∈ dom(door_train_carriage) ⇒ t ∈ dom(door) ∧ door(t) = door_train_carriage[{t}] ∧ door(t)≠∅
  @inv2 door_train_carriage ∈ trns ↔ DOOR
  @inv3 door_train_carriage = (DOOR_CARRIAGE;train_carriage)∼
  @inv4 door_train_carriage∼∈ DOOR ⇸ trns
  @inv5 ∀t·t∈ dom(door) ∧ door(t)≠∅ ⇒ door(t) = door_train_carriage[{t}]
  @inv6 leader_carriage ∈ trns ⤔ CABIN_CARRIAGE
  @inv7 finite(leader_carriage)
  @inv8 trns_state ∈ trns → TRAIN_STATE
  @inv9 ∀t,c· t∈ran(train_carriage) ∧ trns_state(t)≠MAINTENANCE ∧ c = train_carriage∼[{t}]
        ∧ finite(CABIN_CARRIAGE) ∧ t ∈ dom(leader_carriage)
        ⇒ card(c∩CABIN_CARRIAGE)=NUMBER_CABIN_CARRIAGE ∧ leader_carriage(t) ∈ c
  @inv10 ∀t· t∈trns ∧ trns_state(t)=MAINTENANCE ⇒ speed(t)=0
  @inv11 emergency_brake ⊆trns
  @inv12 ∀t·((t∈trns ∧ speed(t)>MAX_SPEED) ⇒ t ∈ emergency_brake)
  @inv13 carriage_door_state ∈ DOOR_CARRIAGE → DOOR_STATE
  @inv14 ∀d·d ∈ dom(door_state) ∧ door_state(d)=OPEN ⇒ carriage_door_state(d↦DOOR_CARRIAGE(d))=OPEN
  @inv15 ∀d·d∈dom(door_state)∧door_state(d)=CLOSED ⇒ carriage_door_state(d↦DOOR_CARRIAGE(d))=CLOSED
  theorem @thm1 ∀c·c∈ran(DOOR_CARRIAGE) ∧ c∈dom(train_carriage)
          ⇒ DOOR_CARRIAGE∼[{c}]⊆door_train_carriage[{train_carriage(c)}]
  theorem @thm2 ∀c·c ∈ dom(train_carriage) ∧ door(train_carriage(c)) ∩ door_state∼[{OPEN}]=∅
          ∧ door(train_carriage(c))≠∅ ⇒ DOOR_CARRIAGE∼[{c}]⊆door(train_carriage(c))
          ∧ DOOR_CARRIAGE∼[{c}] ∩ door_state∼[{OPEN}]=∅
  theorem @thm3 dom(dom(carriage_door_state)) ⊆ dom(door_state)
```

(b) Variables and Invariants

```
event increaseMaxSpeed refines changeSpeed
  any t1 s1
  where
    @grd1 s1 ∈ ℕ
    @grd2 t1 ∈ dom(door_train_carriage)\braking
    @grd3 trns_state(t1) ≠ MAINTENANCE
    @grd4 s1 > MAX_SPEED
    @grd5 speed(t1)<s1
    @grd6 t1 ∉ emergency_brake
    @grd7 speed(t1)≤ MAX_SPEED
    @grd8 door_train_carriage[{t1}]
          ∩ door_state∼[{OPEN}] =∅
    @grd9 door_train_carriage[{t1}]≠∅
    @grd10 permit(t1)=TRUE
  then
    @act1 speed (t1) ≔ s1
    @act2 emergency_brake ≔emergency_brake ∪ {t1}
end

event modifyTrain refines modifyTrain
  any t state
  where
    @grd1 t ∈ trns
    @grd2 state = MAINTENANCE ⇒ speed(t)=0
    @grd3 card(train_carriage∼[{t}]∩CABIN_CARRIAGE)
           =NUMBER_CABIN_CARRIAGE
           ∧ t ∈ dom(leader_carriage)
          ∧ leader_carriage(t) ∈ train_carriage∼[{t}]
    @grd4 state ∈ TRAIN_STATE
    @grd5 state ≠ trns_state(t)
  then
    @act1 trns_state(t)≔state
end

event allocateLeaderCabinCarriageTrain
  any c
  where
    @grd1 c ∈ dom(train_carriage)
    @grd2 finite(train_carriage∼[{train_carriage(c)}])
    @grd3 c ∈ CABIN_CARRIAGE
    @grd4 c ∈ dom(train_carriage ▷ {train_carriage(c)})
    @grd5 trns_state(train_carriage(c))=MAINTENANCE
    @grd6 card(dom(train_carriage ▷ {train_carriage(c)}))
          =MAX_NUMBER_CARRIAGE
    @grd7 train_carriage(c) ∉ dom(leader_carriage)
  then
    @act1 leader_carriage(train_carriage(c)) ≔ c
end

event deallocateLeaderCabinCarriageTrain
  any t
  where
    @grd1 t ∈ dom(leader_carriage)
    @grd2 finite(train_carriage∼[{t}])
    @grd3 trns_state(t)=MAINTENANCE
    @grd4 card(dom(train_carriage ▷ {t}))
          =MAX_NUMBER_CARRIAGE
  then
    @act1 leader_carriage ≔ {t}⩤leader_carriage
end
```

(c) Some events of *Train_M2*

FIGURE 6.17: Excerpt of machine *Train_M2*

6.9 Third Refinement of *Train*: *Train_M3*

As a continuation of the refinement of the train doors by carriage, we data refine variable *door_state*. The opening doors event needs to be strengthened to specify which doors to open when a train is stopped in a platform. Figure 6.18 shows an excerpt of *Train_M3*. Some additional properties related to the allocation of the leader carriage are defined: when a train has already allocated a leader carriage, then it has the correct number of carriages ($inv2$) and the leader carriage belongs to the set of carriage of that train ($inv3$). These two invariants could have been included in the previous refinement. Nevertheless due to the high number of proof obligations already existing in the previous refinement, they were added later. Variable *door_state* disappears being refined by *door_carriage_state* and gluing invariants $inv1$ and $thm2$. Theorem $thm1$ is added to help with the proofs: the carriage doors of a train t are the same as the doors defined by the constant $DOOR\_CARRIAGE$ restricted to the carriages. Some existing events are strengthened in this refinement to be consistent with the invariants as illustrated in Fig. 6.18(b). Due to $inv2$, event $allocateLeaderCabinCarriageTrain$ needs to be strengthened by adding guard $grd8$: this event is only enabled if the number of carriages for that train is equal to $NUMBER\_CABIN\_CARRIAGE$. Also events $allocateCarriageTrain$ and $removeCarriageTrain$ require an additional guard ($grd4$ and $grd11$ respectively) stating that the events are only enabled if train t does not have a leader carriage yet. Therefore we reinforce some ordering in the events: first carriages are allocated/removed; after the leader carriage can be allocated. Refined event $openDoors$ is strengthened with the inclusion of guard $grd8$: the set of carriage doors ds that are opened are located in the same side as the $platform$.

6.10 Fourth Refinement of *Train* and Second Decompositon: *Train_M4*

The fourth refinement of *Train* corresponds to the preparation step before the decomposition. Context *Train_C4*, illustrated in Fig. 6.19(a), introduces an enumerated carrier set $TRAIN\_MOVING\_STATE$ defining the moving state of a train: $MOVING$, NOT_READY (not ready to move) and $NEUTRAL$ (not moving but ready to move). We use additional control variables to help in the separation of aspects resulting in adding variables *ready_train* and *train_doors_closed*. Both are total functions between $trns$ and BOOL ($inv1$ and $inv2$ in Fig. 6.19(b)). *ready_train* defines trains that are ready to move or moving (which therefore have a leader carriage and the correct number of carriages to move ($inv3$)); *train_doors_closed* defines trains that have all their doors closed ($inv4$). These variables are somehow redundant and are mainly added as a preparation for the shared event decomposition: they will be allocated to *LeaderCarriage* and represent a combination of states defined by *Carriage* variables. They also simplify

```
machine Train_M3 refines Train_M2  sees Train_C2

variables trns speed permit braking emergency_button train_carriage carriage_alarm leader_carriage
          trns_state emergency_brake carriage_door_state door_train_carriage

invariants
  @inv1 ∀d,ds·d ∈ dom(door_state) ∧ ds ∈ DOOR_STATE ∧ carriage_door_state(d↦DOOR_CARRIAGE(d))=ds ⇔ door_state(d)=ds
  @inv2 ∀t·t∈trns ∧ t ∈ dom(leader_carriage) ∧ card(train_carriage∼[{t}])=MAX_NUMBER_CARRIAGE
        ∧ card(train_carriage∼[{t}]∩CABIN_CARRIAGE)=NUMBER_CABIN_CARRIAGE
  @inv3 ∀t·t∈trns ∧ t ∈ dom(leader_carriage) ⇒ leader_carriage(t) ∈ train_carriage∼[{t}]
  theorem @thm1 ∀t·t∈dom(door_train_carriage) ⇒ door_train_carriage[{t}]=DOOR_CARRIAGE∼[train_carriage∼[{t}]]
  theorem @thm2 ∀d,ds·d ⊆ dom(door_state) ∧ ds ∈ DOOR_STATE ∧ carriage_door_state[d×DOOR_CARRIAGE[d]]={ds}
                ⇔door_state[d]={ds}
```

(a) Variables and invariants

```
event allocateLeaderCabinCarriageTrain
refines allocateLeaderCabinCarriageTrain
  any c
  where
    @grd1 c ∈ dom(train_carriage)
    @grd2 finite(train_carriage∼[{train_carriage(c)}])
    @grd3 c ∈ CABIN_CARRIAGE
    @grd4 c ∈ dom(train_carriage ▷ {train_carriage(c)})
    @grd5 trns_state(train_carriage(c))=MAINTENANCE
    @grd6 card(train_carriage∼[{train_carriage(c)}])
          =MAX_NUMBER_CARRIAGE
    @grd7 train_carriage(c) ∉ dom(leader_carriage)
    @grd8 card(train_carriage∼[{train_carriage(c)}]∩CABIN_CARRIAGE)
          =NUMBER_CABIN_CARRIAGE
  then
    @act1 leader_carriage(train_carriage(c)) ≔ c
end

event allocateCarriageTrain refines allocateCarriageTrain
  any c t
  where
    @grd1 c ∈ CARRIAGE\dom(train_carriage)
    @grd2 carriage_alarm[{c}]= {FALSE}
    @grd3 ∀tr·tr ∈ dom(door_train_carriage) ∧ tr≠t
          ⇒ DOOR_CARRIAGE∼[{c}]∩door_train_carriage[{tr}]=∅
    @grd4 t ∈ trns\dom(leader_carriage)
    @grd5 emergency_button(t) = FALSE
    @grd6 finite(train_carriage∼[{t}])
    @grd7 card(dom(train_carriage ▷ {t}))<MAX_NUMBER_CARRIAGE
    @grd8 speed(t)=0
    @grd9 DOOR_CARRIAGE∼[{c}] ∩ door_train_carriage[{t}]=∅
    @grd10 trns_state(t)=MAINTENANCE
  then
    @act1 train_carriage(c)≔ t
    @act2 door_train_carriage ≔ door_train_carriage
                                ∪ ({t} × DOOR_CARRIAGE∼[{c}])
    @act3 carriage_door_state≔ carriage_door_state
                              ⩤ ((DOOR_CARRIAGE▷{c})×{CLOSED})
end

event openDoors refines openDoors
  any t occpTrns platform ds
  where
    @grd1 t ∈ TRAIN
    @grd2 occpTrns ∈ ℙ(CDV)
    @grd3 platform ∈ PLATFORM
    @grd4 platform ∈ (occpTrns ∩ PLATFORM)
    @grd5 t ∈ dom((DOOR_CARRIAGE;train_carriage)∼)
    @grd6 speed(t) = 0
    @grd7 ({platform} ≠ ∅) ∨ emergency_button(t) = TRUE
    @grd8 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
    @grd9 ds ⊆ DOOR_CARRIAGE∼[train_carriage∼[{t}]]
    @grd10 ∀d·d∈ds
           ⇒carriage_door_state[{d}◁DOOR_CARRIAGE]={CLOSED}
    @grd11 ds≠∅
  then
    @act1 carriage_door_state≔ carriage_door_state
                              ⩤ ((ds◁DOOR_CARRIAGE)×{OPEN})
end

event removeCarriageTrain refines removeCarriageTrain
  any c t
  where
    @grd1 t ∈ dom(door_train_carriage)
    @grd2 c↦t ∈ train_carriage
    @grd3 carriage_alarm(c) = FALSE
    @grd4 emergency_button(t) = FALSE
    @grd5 trns_state(t)=MAINTENANCE
    @grd6 speed(t)=0
    @grd7 carriage_door_state[DOOR_CARRIAGE▷{c}]={CLOSED}
    @grd8 ∀d·d∈DOOR_CARRIAGE∼[{c}]
          ⇒ t = door_train_carriage∼(d)
    @grd9 c ∈ ran(DOOR_CARRIAGE)
    @grd10 DOOR_CARRIAGE∼[{c}]⊆door_train_carriage[{t}]
    @grd11 t ∉ dom(leader_carriage)
  then
    @act1 train_carriage ≔ {c}⩤train_carriage
    @act2 door_train_carriage ≔ door_train_carriage
                               ⩥DOOR_CARRIAGE∼[{c}]
```

(b) Refinement of some events in *Train_M3*

FIGURE 6.18: Excerpt of machine *Train_M3*

the event splitting by replacing predicates that contain variables related to carriages. For instance, in Fig. 6.19(c) guard $grd8$ of event $increaseMaxSpeed$ replaces guard $grd8$ in the abstract event (Fig. 6.17(c)): this event does not need to refer to variable $door\_train\_carriage$ since it is only required to ensure that all the train doors are closed when a train increases its speed ($train\_doors\_closed(t1) = \text{TRUE}$). The consequence of adding these variables is that they need to be consistent throughout the events. For instance, $act2$ needs to be added to the actions of $deallocateLeaderCabinCarriageTrain$ when a leader carriage is deallocated from a train which implies that the train is no longer ready to move (Fig. 6.19(c)). Therefore these control variables should be added with care in particular when it is intended to further refine the resulting sub-events after an event decomposition. Invariants $inv5$ and $inv6$ are gluing invariants resulting from the added variables: the first states that if a train has its doors opened, then the train must be stopped; the second states that if a train is ready, then the set of carriages for that train is not empty. All other events are updated reflecting the introduction of the new variables.

```
context Train_C4 extends Train_C2

constants MOVING NOT_READY NEUTRAL

sets TRAIN_MOVING_STATE

axioms
  @axm1 partition(TRAIN_MOVING_STATE, {MOVING}, {NOT_READY}, {NEUTRAL})
end
```

(a) Context *Train_C4*

```
machine Train_M4 refines Train_M3  sees Train_C4

variables trns speed permit braking emergency_button train_carriage
carriage_alarm leader_carriage trns_state emergency_brake
carriage_door_state door_train_carriage ready_train train_doors_closed

invariants
  @inv1 ready_train ∈ trns → BOOL
  @inv2 train_doors_closed ∈ trns → BOOL
  @inv3 ∀t· t∈dom(ready_train) ∧ ready_train(t) = TRUE ⇒ t∈trns
        ∧ card(train_carriage∼[{t}])=MAX_NUMBER_CARRIAGE
        ∧ card(train_carriage∼[{t}]∩CABIN_CARRIAGE)
          =NUMBER_CABIN_CARRIAGE
        ∧ t ∈ dom(leader_carriage)
  @inv4 ∀t· t∈dom(train_doors_closed)
        ∧ train_doors_closed(t) = TRUE
        ⇒ (∀d·d ∈door_train_carriage[{t}]
            ⇒ carriage_door_state(d↦DOOR_CARRIAGE(d))≠OPEN)
  @inv5 ∀t· t∈dom(train_doors_closed)
        ∧ train_doors_closed(t) = FALSE ⇒ speed(t) = 0
  @inv6 ∀t· t∈dom(ready_train) ∧ ready_train(t) = TRUE
        ⇒ DOOR_CARRIAGE▷train_carriage∼[{t}]≠∅
```

(b) Variables and invariants

```
event increaseMaxSpeed refines increaseMaxSpeed
  any t1 s1
  where
    @grd1 s1 ∈ N
    @grd2 t1 ∈ trns
    @grd3 t1 ∉ braking
    @grd4 trns_state(t1) ≠ MAINTENANCE
    @grd5 s1 > MAX_SPEED
    @grd6 speed(t1)<s1
    @grd7 t1 ∉ emergency_brake
    @grd8 speed(t1)≤ MAX_SPEED
    @grd9 train_doors_closed(t1) = TRUE
    @grd10 permit(t1)=TRUE
    @grd11 speed(t1)>0
    @grd12 ready_train(t1) = TRUE
  then
    @act1 speed (t1) ≔ s1
    @act2 emergency_brake ≔ emergency_brake ∪ {t1}
 end
```

```
event deallocateLeaderCabinCarriageTrain
refines deallocateLeaderCabinCarriageTrain
  any t lc
  where
    @grd1 t ∈ dom(leader_carriage)
    @grd2 finite(train_carriage∼[{t}])
    @grd3 trns_state(t)=MAINTENANCE
    @grd4 card(dom(train_carriage ▷ {t}))=MAX_NUMBER_CARRIAGE
    @grd5 lc = leader_carriage
  then
    @act1 leader_carriage ≔ {t}⩤leader_carriage
    @act2 ready_train(t) ≔ FALSE
end
```

(c) Refinement of some events in *Train_M4*

FIGURE 6.19: Excerpt of machine *Train_M4*

Now we are ready to proceed to the next decomposition as described in Fig. 6.3. We want to separate the aspects related to carriages from the aspects related to leader carriages:

Leader Carriage: Allocates the leader carriage, controls the speed of the train, modifies the state of the train, receives the messages sent from the central, handles the emergency button of the train.

Carriage: Add and removes carriages, opens and closes carriage doors, handles the carriage alarm.

The decomposition is summarised in Table 6.1 (equivalent to view of Fig. 6.12 with the addition of the variable partition):

	LeaderCarriage	Carriage
Variables	$trns, permit, braking, emergency\_button$ $trns\_state, speed, emergency\_brake$ $ready\_train, train\_doors\_closed$	$carriage\_alarm, leader\_carriage$ $carriage\_door\_state, door\_train\_carriage$ $train\_carriage$
Events	$openDoors, closeDoors$ $activateEmergencyCarriageButton$ $deactivateEmergencyCarriageButton$ $deactivateEmergencyTrainButton$ $allocateLeaderCabinCarriageTrain$ $deallocateLeaderCabinCarriageTrain$ $allocateCarriageTrain$ $modifyTrain, removeCarriageTrain$ $increaseSpeed, increaseMaxSpeed$ $reduceSpeed, reduceMaxSpeed$ $recvTrainMsg, brake, stopBraking$ $addTrain, enterCDV, leaveCDV$	$openDoors, closeDoors$ $activateEmergencyCarriageButton$ $deactivateEmergencyCarriageButton$ $deactivateEmergencyTrainButton$ $allocateLeaderCabinCarriageTrain$ $deallocateLeaderCabinCarriageTrain$ $allocateCarriageTrain$ $modifyTrain, removeCarriageTrain$

TABLE 6.1: Decomposition summary of *Train_M4*

6.10.1 Machine *LeaderCarriage*

Machine *LeaderCarriage* contains the variables that are not related to the carriages (Fig. 6.20(a)). Some events are only included in this sub-component: events dealing with the speed changes, entering and leaving sections, receiving messages and adding trains. All the other events are shared between the two sub-components.

6.10.2 Machine *Carriage*

The variables related to carriages are included in sub-component *Carriage* (Fig. 6.20(b)). All the events of *Carriage* result from splitting the original events as described in Table. 6.1. We are interested in adding more details about the carriage doors, therefore we further refine *Carriage*.

6.10.3 Refinement of *Carriage* and Decomposition: *Carriage_M1*

This refinement is a preparation step before the next decomposition. We intend to use an existing generic development of carriage doors as a pattern and apply a *generic instantiation* to our model. We use the shared event decomposition to adjust our current model to fit the first machine of the pattern. *Carriage_M1* refines *Carriage* and after is decomposed in a way that one of the resulting sub-components fits the generic model of carriage doors. The generic model is described in Sect. 6.11.

Two variables are introduced in this refinement, representing the carriage doors (*carriage_door*) and their respective state (*carriage_ds*) as seen in Fig. 6.21(a). The last variable is used

```
machine LeaderCarriage sees LeaderCarriage_C0

variables trns speed permit braking emergency_button trns_state
          emergency_brake ready_train train_doors_closed

invariants
  theorem @typing_train_doors_closed train_doors_closed ∈ ℙ(TRAIN × BOOL)
  @Train_MetroSystem_M0_inv3 trns ⊆ TRAIN
  @Train_MetroSystem_M0_inv9 braking ⊆ trns
  @Train_MetroSystem_M0_inv10 speed ∈ trns → ℕ
  @Train_MetroSystem_M1_inv2 permit ∈ trns → BOOL
  @Train_MetroSystem_M1_inv3 emergency_button ∈ trns → BOOL
  @Train_M1_inv1 finite(trns)
  @Train_M2_inv8 trns_state ∈ trns → TRAIN_STATE
  @Train_M2_inv10 ∀t·t∈trns ∧ trns_state(t)=MAINTENANCE ⇒ speed(t)=0
  @Train_M2_inv11 emergency_brake ⊆trns
  @Train_M2_inv12 ∀t·((t∈trns ∧ speed(t)>MAX_SPEED) ⇒ t ∈ emergency_brake)
  @Train_M4_inv14 ready_train ∈ trns → BOOL
  @Train_M4_inv16 train_doors_closed ∈ trns → BOOL
  @Train_M4_inv18 ∀t·t∈dom(train_doors_closed) ∧ train_doors_closed(t) = FALSE
                  ⇒ speed(t) = 0
  theorem @WD_Train_M4_inv6 ∀t·t∈dom(ready_train)⇒ready_train∈TRAIN ⇸ BOOL
```

(a) sub-component *LeaderCarriage*

```
machine Carriage sees Carriage_C0

variables train_carriage carriage_alarm leader_carriage carriage_door_state
          door_train_carriage

invariants
  theorem @typing_leader_carriage leader_carriage ∈ ℙ(TRAIN × CARRIAGE)
  theorem @typing_door_train_carriage door_train_carriage ∈ ℙ(TRAIN × DOOR)
  theorem @typing_train_carriage train_carriage ∈ ℙ(CARRIAGE × TRAIN)
  theorem @typing_carriage_alarm carriage_alarm ∈ ℙ(CARRIAGE × BOOL)
  @Train_M1_inv2 carriage_alarm ∈ CARRIAGE → BOOL
  @Train_M1_inv4 finite(train_carriage)
  @Train_M1_inv5 finite(dom(train_carriage))
  @Train_M2_inv3 door_train_carriage = (DOOR_CARRIAGE;train_carriage)∼
  @Train_M2_inv7 finite(leader_carriage)
  @Train_M2_inv13 carriage_door_state ∈ DOOR_CARRIAGE → DOOR_STATE
  theorem @Train_M2_thm1 ∀c·c∈ran(DOOR_CARRIAGE) ∧ c∈dom(train_carriage)
          ⇒ DOOR_CARRIAGE∼[{c}]⊆door_train_carriage[{train_carriage(c)}]
  theorem @Train_M3_thm1 ∀t·t∈dom(door_train_carriage)
          ⇒ door_train_carriage[{t}]=DOOR_CARRIAGE∼[train_carriage∼[{t}]]
```

(b) sub-component *Carriage*

FIGURE 6.20: Variables and invariants of *LeaderCarriage* and *Carriage*

to data refine *carriage_door_state* that disappears. The gluing invariant for this data refinement is expressed by $inv4$: the state of all the doors in *carriage_ds* match the state of the same door in *carriage_door_state*. As a result, some events need to be refined to fit the new variables. For instance, in Fig. 6.21(b), $act1$ in event $openDoors$ updates variable *carriage_ds* instead of the abstract variable *carriage_door_state*. Also when carriage doors are allocated, both new variables are assigned as seen in actions $act3$ and $act4$ of event $allocateCarriageTrain$ (similar for $removeCarriageTrain$).

Comparing with the generic model of carriage doors, the relevant events to fit the instantiation are $openDoors$, $closeDoors$, $allocateCarriageTrain$ and $removeCarriageTrain$. Not by coincidence, these events manipulate variables *carriage_ds* and *carriage_door* that will instantiate generic variables *generic_door_state* and *generic_door* respectively. The decomposition summary is described in Table 6.2.

6.10.4 Machine *CarriageInterface*

Machine *CarriageInterface* contains the variables that are not related to the carriage doors. This machine handles the activation/deactivation of the carriage alarm, the deac-

```
machine Carriage_M1 refines Carriage sees Carriage_C0

variables carriage_alarm leader_carriage train_carriage carriage_door carriage_ds door_train_carriage

invariants
  @inv1 carriage_door ⊆ DOOR
  @inv2 carriage_ds ∈ carriage_door → DOOR_STATE
  @inv3 ∀c·c∈dom(train_carriage) ⇒ DOOR_CARRIAGE~[{c}]⊆carriage_door
  @inv4 ∀d,c·d↦c∈dom(carriage_door_state) ∧ d ∈ dom(carriage_ds) ∧ d∈ran(door_train_carriage)
        ⇒carriage_ds(d)=carriage_door_state(d↦c)
  @inv5 door_train_carriage~∈DOOR ⇸ TRAIN
  @inv6 ∀d·d∈ran(door_train_carriage) ⇒ d ∈ carriage_door
```

(a) Variables and invariants

```
event openDoors refines openDoors
  any t occpTrns platform ds
  where
    @typing_platform platform ∈ CDV
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ TRAIN
    @grd2 occpTrns ∈ ℙ(CDV)
    @grd3 platform ∈ PLATFORM
    @grd4 platform ∈ (occpTrns ∩ PLATFORM)
    @grd5 t ∈ dom((DOOR_CARRIAGE;train_carriage)~)
    @grd6 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
    @grd7 ds ⊆ DOOR_CARRIAGE~[train_carriage~[{t}]]
    @grd8 ds ⊆ dom(carriage_ds)
    @grd9 carriage_ds[ds]={CLOSED}
  then
    @act1 carriage_ds≔carriage_ds⊲ (ds×{OPEN})
end

event closeDoors refines closeDoors
  any t ds closed cds
  where
    @typing_closed closed ∈ BOOL
    @typing_ds ds ∈ ℙ(DOOR)
    @grd1 t ∈ TRAIN
    @grd2 t ∈ dom(((train_carriage~);(DOOR_CARRIAGE~)))
    @grd3 ds ⊆ ((train_carriage~);(DOOR_CARRIAGE~))[{t}]
    @grd4 cds = carriage_ds
    @grd5 (∃d·d∈DOOR_CARRIAGE~[train_carriage~[{t}]]\ds
           ∧ cds(d)≠CLOSED) ⇔ closed = FALSE
    @grd6 ds ⊆ dom(carriage_ds)
    @grd7 carriage_ds[ds]={OPEN}
  then
    @act2 carriage_ds≔carriage_ds ⊲ (ds×{CLOSED})
end

event allocateCarriageTrain refines allocateCarriageTrain
  any c t ds
  where
    @typing_t t ∈ TRAIN
    @typing_c c ∈ CARRIAGE
    @grd1 c ∈ CARRIAGE\dom(train_carriage)
    @grd2 carriage_alarm[{c}]= {FALSE}
    @grd3 t ∈ dom(door_train_carriage)
    @grd4 ∀tr·tr ∈ dom(door_train_carriage) ∧ tr≠t
          ⇒ DOOR_CARRIAGE~[{c}]∩door_train_carriage[{tr}]=∅
    @grd5 finite(train_carriage~[{t}])
    @grd6 card(dom(train_carriage ▷ {t}))<MAX_NUMBER_CARRIAGE
    @grd7 DOOR_CARRIAGE~[{c}] ∩ door_train_carriage[{t}]=∅
    @grd8 t∉dom(leader_carriage)
    @grd9 ds = DOOR_CARRIAGE~[{c}]
    @grd10 ds∩dom(carriage_ds)=∅
  then
    @act1 train_carriage(c)≔ t
    @act2 door_train_carriage ≔ door_train_carriage
          ∪ ({t} × DOOR_CARRIAGE~[{c}])
    @act3 carriage_door ≔ carriage_door ∪ ds
    @act4 carriage_ds ≔ carriage_ds ∪ (ds×{CLOSED})
end

event removeCarriageTrain refines removeCarriageTrain
  any c t ds
  where
    @typing_t t ∈ TRAIN
    @typing_c c ∈ CARRIAGE
    @grd1 t ∈ dom(door_train_carriage)
    @grd2 c↦t ∈ train_carriage
    @grd3 carriage_alarm(c) = FALSE
    @grd16 t ∈ dom(door_train_carriage)
    @grd10 ∀d·d∈DOOR_CARRIAGE~[{c}]
           ⇒ t = door_train_carriage~(d)
    @grd11 c ∈ ran(DOOR_CARRIAGE)
    @grd12 t ∉ dom(leader_carriage)
    @grd13 ds = DOOR_CARRIAGE~[{c}]
    @grd14 ds⊆carriage_door
    @grd15 carriage_ds[DOOR_CARRIAGE~[{c}]] = {CLOSED}
  then
    @act1 train_carriage ≔ {c}⩤train_carriage
    @act2 door_train_carriage ≔
              door_train_carriage ⩥DOOR_CARRIAGE~[{c}]
    @act3 carriage_door ≔ carriage_door \ ds
    @act4 carriage_ds ≔ ds⩤carriage_ds
end
```

(b) Refinement of some events in *Carriage_M1*

FIGURE 6.21: Excerpt of machine *Carriage_M1*

tivation of the emergency button and the allocation/deallocation of the leader cabin carriage. Events *openDoors*, *closeDoors*, *allocateCarriageTrain* and *removeCarriageTrain* are shared with *CarriageDoor*.

6.10.5 Machine *CarriageDoor*

CarriageDoors contains the variables related to carriage doors and the events resulting from splitting the original events as described in Table 6.2. The resulting sub-events can be seen in Fig. 6.22.

	CarriageInterface	CarriageDoor
Variables	*carriage_alarm, leader_carriage* *train_carriage, door_train_carriage*	*carriage_doors, carriage_ds*
Events	*openDoors, closeDoors* *allocateCarriageTrain* *removeCarriageTrain* *activateEmergencyCarriageButton* *deactivateEmergencyCarriageButton* *deactivateEmergencyTrainButton* *allocateLeaderCabinCarriageTrain* *deallocateLeaderCabinCarriageTrain* *modifyTrain*	*openDoors, closeDoors* *allocateCarriageTrain* *removeCarriageTrain*

TABLE 6.2: Decomposition summary of *Carriage_M1*

```
event openDoors
 any t occpTrns platform ds
 where
  @typing_platform platform ∈ CDV
  @typing_ds ds ∈ ℙ(DOOR)
  @grd1 t ∈ TRAIN
  @grd2 occpTrns ∈ ℙ(CDV)
  @grd3 platform ∈ PLATFORM
  @grd4 platform ∈ (occpTrns ∩ PLATFORM)
  @grd7 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
  @grd11 ds ⊆ dom(carriage_ds)
  @grd12 carriage_ds[ds]={CLOSED}
 then
  @act2 carriage_ds≔carriage_ds   (ds×{OPEN})
end

event closeDoors
 any t ds closed cds
 where
  @typing_cds cds ∈ ℙ(DOOR × DOOR_STATE)
  @typing_closed closed ∈ BOOL
  @typing_ds ds ∈ ℙ(DOOR)
  @grd1 t ∈ TRAIN
  @gd13 cds = carriage_ds
  @grd11 ds ⊆ dom(carriage_ds)
  @grd12 carriage_ds[ds]={OPEN}
 then
  @act2 carriage_ds≔carriage_ds   (ds×{CLOSED})
end

event allocateCarriageTrain
 any c t ds
 where
  @typing_ds ds ∈ ℙ(DOOR)
  @typing_t t ∈ TRAIN
  @typing_c c ∈ CARRIAGE
  @grd14 ds = DOOR_CARRIAGE∼[{c}]
  @grd15 ds∩dom(carriage_ds)=∅
 then
  @act3 carriage_door ≔ carriage_door ∪ ds
  @act4 carriage_ds ≔ carriage_ds ∪ (ds×{CLOSED})
end

event removeCarriageTrain
 any c t ds
 where
  @typing_ds ds ∈ ℙ(DOOR)
  @typing_t t ∈ TRAIN
  @typing_c c ∈ CARRIAGE
  @grd11 c ∈ ran(DOOR_CARRIAGE)
  @grd13 ds = DOOR_CARRIAGE∼[{c}]
  @grd14 ds⊆carriage_door
  @grd15 carriage_ds[DOOR_CARRIAGE∼[{c}]] = {CLOSED}
 then
  @act3 carriage_door ≔ carriage_door \ ds
  @act4 carriage_ds ≔ ds⩤carriage_ds
end
```

FIGURE 6.22: Events of sub-component *CarriageDoors*

There are two kind of carriage doors: *emergency doors* and *service doors*. We intend to instantiate twice the generic doors development, one per kind of door (the developments are similar for both kind of doors). Specific details for each kind of door are added as additional refinements later on. We describe the generic model and afterwards the instantiation.

6.11 Generic Model: *GCDoor*

The generic model for the carriage doors is based in three refinements: *GCDoor_M0*, *GCDoor_M1* and *GCDoor_M2*. In each refinement step, more requirements and details are introduced.

6.11.1 Abstract machine *GCDoor_M0*

We start by adding the carriage doors and respective states. Four events model carriage doors. The properties to be preserved are:

1. Doors can be added or removed.
2. Doors can be in an opening or closing state. Doors can only be open if the train is in a platform.
3. When adding/removing doors, they are closed by default for safety reasons.

The static part of the generic development is initially divided in two parts: context *GCDoor_C0* for the doors and context *GCTrack_C0* for the tracks as seen in Fig. 6.23.

```
context GCDoor_C0 extends GCTrack_C0

constants GEN_DOOR_CARRIAGE
          DOOR_SIDE
          OPEN CLOSED

sets DOOR DOOR_STATE
     GEN_CARRIAGE

axioms
  @axm1 partition(DOOR_STATE, {OPEN}, {CLOSED})
  @axm2 GEN_DOOR_CARRIAGE ∈ DOOR → GEN_CARRIAGE
  @axm3 DOOR_SIDE ∈ DOOR → SIDE
end
```

(a) Context *GCDoor_C0*

```
context GCTrack_C0

constants RIGHT PLATFORM LEFT PLATFORM_SIDE

sets SIDE TRACK

axioms
  @axm1 PLATFORM ⊆ TRACK
  @axm2 partition(SIDE, {LEFT}, {RIGHT})
  @axm3 PLATFORM_SIDE ∈ PLATFORM → SIDE
end
```

(b) Context *GCTrack_C0*

FIGURE 6.23: Generic contexts

Context *GCDoor_C0* contains sets $DOOR$, $DOOR\_STATE$ and $GEN\_DOOR\_CARRIAGE$, representing carriage doors, defining if a door is opened or closed and defining the carriages to which a door belongs to, respectively. Context *GCTrack_C0* contains sets $SIDE$ and $TRACK$, defining the side ($LEFT$ or $RIGHT$) of a door or platform and each section of the track, respectively. Machine *GCDoor_M0* contains variables *generic_door* and *generic_door_state*. The invariants of this abstraction are quite weak since we just add the type variables as can be seen in Fig. 6.24(a).

Property 1 is expressed by events *addDoor* and *removeDoor*. Property 2 is expressed by variable *generic_door_state* and events *openDoors* and *closeDoors*. Event *openDoors* is only enabled if the set of doors *ds* is closed and if the parameter $occpTrns$, corresponding to the sections occupied by the carriage, intersects a platform. Doors are removed in event *removeDoor*, if they are $CLOSED$ confirming property 3. Next section describes the refinement of this machine.

```
machine GCDoor_M0 sees GCDoor_C0

variables generic_door generic_door_state

invariants
  @inv1 generic_door ⊆ DOOR
  @inv2 generic_door_state ∈ generic_door → DOOR_STATE

event openDoors
    any ds platform occpTrns
    where
      @grd ds ⊆ DOOR
      @grd1 ds ⊆ dom(generic_door_state)
      @grd2 generic_door_state[ds]={CLOSED}
      @grd3 platform ∈ PLATFORM
      @grd4 platform ∈ (occpTrns ∩ PLATFORM)
      @grd5 ds ≠∅
      @grd6 DOOR_SIDE[ds]={PLATFORM_SIDE(platform)}
    then
      @act1 generic_door_state≔generic_door_state  (ds×{OPEN})
  end
```

(a) Variables, invariants and event *openDoors*

```
event closeDoors
  any ds
  where
    @grd ds ⊆ DOOR
    @grd1 ds ⊆ dom(generic_door_state)
    @grd2 generic_door_state[ds]={OPEN}
    @grd3 ds ≠∅
  then
    @act1 generic_door_state≔generic_door_state
                                   (ds×{CLOSED})
end

event addDoor
  any ds c
  where
    @grd1 ds ∩ generic_door = ∅
    @grd2 ds ≠ ∅
    @grd3 ds = GEN_DOOR_CARRIAGE~[{c}]
  then
    @act1 generic_door ≔ generic_door ∪ ds
    @act2 generic_door_state ≔ generic_door_state
                                   ∪ (ds×{CLOSED})
end

event removeDoor
  any ds c
  where
    @grd1 ds ⊆ generic_door
    @grd2 ds ≠ ∅
    @grd3 generic_door_state[ds]={CLOSED}
    @grd4 ds = GEN_DOOR_CARRIAGE~[{c}]
  then
    @act1 generic_door ≔ generic_door \ ds
    @act2 generic_door_state ≔
                         ds⩤generic_door_state
end
```

(b) Some events in *GCDoors_M0*

FIGURE 6.24: Machine *GCDoors_M0*

6.11.2 Second refinement of *GCDoor*: *GCDoor_M1*

In this refinement more details are introduced about the possible behaviour of the doors. The properties to be preserved are:

1. The actions involving the doors may result from *commands* sent from the central door control. These commands have a type ($OPEN\_RIGHT\_DOORS$, $OPEN\_LEFT\_DOORS$, $CLOSE\_RIGHT\_DOORS$, $CLOSE\_LEFT\_DOORS$, $ISOLATE\_DOORS$, $REMOVE\_ISOLATION\_DOORS$), a state ($START$, $FAIL$, $SUCCESS$ and $EXECUTED$) and a target (set of doors).

2. After the doors are closed, they must be locked for the train to move.

3. If a door is open, then an opening device was used: $MANUAL\_PLATFORM$ if opened manually in a platform, $MANUAL\_INTERNAL$ if opened inside the carriage manually and $AUTOMATIC\_CENTRAL\_DOOR$ if opened automatically from the central control.

4. Doors can get obstructed when closed automatically (people/object obstruction). If an obstruction is detected then it should be tried to close the doors again.

The context used in this refinement ($GCDoor\_C1$) extends the existing one as seen in Fig. 6.25(a). Abstract events are refined to include the properties defined above. Some

```
context GCDoor_C1 extends GCDoor_C0

constants MANUAL_PLATFORM MANUAL_INTERNAL AUTOMATIC_CENTRAL_DOOR START FAIL SUCCESS EXECUTED
OPEN_RIGHT_DOORS OPEN_LEFT_DOORS CLOSE_RIGHT_DOORS CLOSE_LEFT_DOORS ISOLATE_DOORS REMOVE_ISOLATION_DOORS

sets OPENING_DEVICE COMMAND_STATE COMMAND_TYPE COMMAND

axioms
  @axm1 partition(OPENING_DEVICE, {MANUAL_PLATFORM}, {MANUAL_INTERNAL}, {AUTOMATIC_CENTRAL_DOOR})
  @axm2 partition(COMMAND_STATE, {START}, {FAIL}, {SUCCESS},{EXECUTED})
  @axm3 partition(COMMAND_TYPE, {OPEN_RIGHT_DOORS}, {OPEN_LEFT_DOORS}, {CLOSE_RIGHT_DOORS},
                              {CLOSE_LEFT_DOORS}, {ISOLATE_DOORS}, {REMOVE_ISOLATION_DOORS})
end
```

(a) Context *GCDoors_C1*

```
machine GCDoor_M1 refines GCDoor_M0  sees GCDoor_C1

variables generic_door generic_door_state locked_doors door_opening_device obstructed_door command
          command_doors command_type command_state

invariants
  @inv1 locked_doors ⊆ DOOR
  @inv2 ∀d·d∈locked_doors ∧ d ∈ dom(generic_door_state) ⇒ generic_door_state(d)∉{OPEN}
  @inv3 door_opening_device ∈ generic_door ⇸  OPENING_DEVICE
  @inv4 ∀d·d∈generic_door ∧ generic_door_state(d)=OPEN ⇒d∈dom(door_opening_device)
  @inv5 obstructed_door ⊆ dom(generic_door_state)
  @inv6 command ⊆ COMMAND
  @inv7 command_type ∈ command → COMMAND_TYPE
  @inv8 command_state ∈ command → COMMAND_STATE
  @inv9 command_doors ∈ command → ℙ(generic_door)
  @inv10 ∀dos·dos∈ran(command_doors) ⇒ dos ≠∅
  @inv11 ∀d,opDev·d ∈ generic_door ∧ opDev ∈ OPENING_DEVICE ∧ (d↦opDev)∈door_opening_device
         ∧ opDev=AUTOMATIC_CENTRAL_DOOR (∃cmd·cmd∈command ∧ d ∈ command_doors(cmd))
```

(b) Variables, invariants

FIGURE 6.25: Excerpt of machine *GCDoors_M1*

new invariants are added as seen in Fig. 6.25(b). Property 1 is defined by new variables *command*, *command_type*, *command_state* and *command_doors* (see invariants *inv*6 to *inv*9). Property 2 is defined by invariant *inv*2 (if a door is locked, then the door is not opened) and events *lockDoor*/*unlockDoor*. Property 3 is defined by variables *door_opening_device*, *inv*3 and *inv*11 (if a door is opened automatically, then a command has been issued to do so). Property 4 is defined by variable *obstructed_door*, *inv*5 and events *doorIsObstructed* and *closeObstructedDoor*. The system works as follows: doors can be opened/closed manually or automatically. To open/close a door automatically, a command must be issued from the central door control defining which doors are affected (for instance, to open a door automatically, event *commandOpenDoors* needs to occur). A command starts with state *START* which can lead to a successful result (*SUCCESS*) or failure (*FAIL*). Either way, it finishes with state *EXECUTED*. Abstract event *otherCommandDoors* refers to commands not defined in this refinement. If a door gets obstructed when being closed automatically (event *doorIsObstructed*) then event *closeObstructedDoor* models a successful attempt to close an obstructed door. Otherwise, it needs to be closed manually.

The system works as follows: doors can be opened/closed manually or automatically. If it is done automatically, a command sent from the central door control is issued defining which doors are affected (for instance, event *commandOpenDoors*, illustrated in Fig. 6.26, issues a command to open a set of doors automatically). Event *otherCommandDoors* is left abstract the enough in order to refer to commands not defined in this refinement. If a door gets obstructed when closing automatically (event *doorIsObstructed*) then

```
event commandOpenDoors
   any doors cmd cmd_type
   where
     @grd doors ⊆ generic_door
     @grd1 generic_door_state[doors]={CLOSED}
     @grd2 cmd_type
         ∈ {OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
     @grd3 cmd ∈ COMMAND\command
     @grd4 doors ≠∅
   then
     @act1 command_state(cmd)≔START
     @act2 command_doors(cmd)≔doors
     @act3 command ≔ command ∪ {cmd}
     @act4 command_type(cmd)≔cmd_type
 end

 event otherCommandDoors
   any doors cmd cmd_type
   where
     @grd doors ⊆ generic_door
     @grd1 cmd_type ∈ COMMAND_TYPE
     @grd3 cmd ∈ COMMAND\command
     @grd4 doors ≠∅
   then
     @act1 command_state(cmd)≔START
     @act2 command_doors(cmd)≔doors
     @act3 command ≔ command ∪ {cmd}
     @act4 command_type(cmd)≔cmd_type
 end

event doorIsObstructed
   any ds cmd
   where
     @grd ds ⊆  DOOR\(locked_doors ∪ obstructed_door)
     @grd1 ds ⊆ dom(generic_door_state)
     @grd2 cmd ∈ command
     @grd3 command_type(cmd)
         ∈ {CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
     @grd4 command_state(cmd)∈{START,FAIL}
     @grd5 ds ⊆ command_doors(cmd)
     @grd6 ds ≠∅
     @grd7 generic_door_state[ds]={OPEN}
   then
     @act1 obstructed_door ≔ obstructed_door ∪ ds
     @act2 command_state(cmd)≔FAIL
 end

event openDoorAutomatically
refines openDoors
 any ds cmd
 where
  @grd ds ⊆ generic_door\locked_doors
  @grd1 ds ⊆ dom(generic_door_state)
  @grd2 generic_door_state[ds]={CLOSED}
  @grd3 cmd ∈ command
  @grd4 command_type(cmd) ∈
     {OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
  @grd5 command_state(cmd)=START
  @grd6 ds ⊆ command_doors(cmd)
  @grd7 ds ≠∅
 then
  @act1 generic_door_state≔
      generic_door_state  (ds×{OPEN})
  @act2 door_opening_device ≔ door_opening_device
                  (ds×{AUTOMATIC_CENTRAL_DOOR})
 end
event lockDoor
  any d
  where
   @grd d ∈ generic_door\locked_doors
   @grd1 generic_door_state(d)=CLOSED
  then
   @act1 locked_doors≔locked_doors ∪ {d}
end

event unlockDoor
  any d
  where
   @grd1 d ∈ generic_door
   @grd2 d ∈ locked_doors
  then
   @act1 locked_doors≔locked_doors \ {d}
end

event closeObstructedDoor
refines closeDoors
 any ds cmd st
 where
   @grd ds ⊆ obstructed_door
   @grd1 ds ⊆ dom(generic_door_state)
   @grd2 cmd ∈ command
   @grd3 command_type(cmd)∈
   {CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
   @grd4 command_state(cmd)=FAIL
   @grd5 ds ⊆ command_doors(cmd)
   @grd6 ds ≠∅
   @grd7 generic_door_state[ds]={OPEN}
   @grd8 st ∈ {SUCCESS,FAIL}
   @grd9 st = SUCCESS ⇔ command_doors(cmd)\ds=∅
       ∨ generic_door_state[command_doors(cmd)\ds]
       ={CLOSED}
  then
   @act1 generic_door_state≔
          generic_door_state(ds×{CLOSED})
   @act2 obstructed_door ≔ obstructed_door \ ds
   @act3 command_state(cmd)≔st
end
```

FIGURE 6.26: Some events in *GCDoors_M1*

event *closeObstructedDoor* models a successful attempt to close an obstructed door. Otherwise, it needs to be closed manually.

6.12 Third refinement of *GCDoor*: *GCDoor_M2*

In the third refinement, malfunctioning doors can be isolated and in that case, they ignore the commands issued by the central command. Isolated doors can be either opened or closed. After the execution of a command, the corresponding state is updated according to the success/failure of the command. The properties to be preserved are:

1. Doors can be isolated (independently of the respective door state) in case of malfunction or safety reasons.

2. If a command is successful, it means that the command already occurred.

3. Two commands cannot have the same door as target except if the command has already been executed.

4. If a door is obstructed, then it must be in a state corresponding to $OPEN$.

The properties to be preserved are mainly defined as invariants. Property 1 is defined by new variable *isolated_door*, $inv1$, $inv6$ and events $commandIsolationDoors$, *isolateDoor* and $removeIsolatedDoor$ as seen in Fig. 6.27(b). Property 2 is defined by several invariants depending on the command: $inv2$ for opening doors, $inv3$ for closing doors, $inv4$ to isolate doors, $inv5$ to lift the isolation from a door. Property 3 is defined by $inv7$ and the last property by $inv8$.

An excerpt of *GCDoors_M2* is depicted in Fig. 6.27. New event $commandIsolationDoors$ models a command to add/remove doors from isolation refining the abstract event $otherCommandDoors$. After this command is issued, the actual execution (or not) of the command dictates the command state at refined event $updateIsolationCmdState$. A command log is created corresponding to the end of the command's task in event *executeLogCmdState*. Other commands could be added in a similar manner but we restrict to these commands for now. The state update of other commands (opening and closing doors) follows the same behaviour as the isolation one.

This model has three refinement layers with all the proof obligations discharged. We instantiate this model, benefiting from the discharged proof obligations and refinements to model emergency and service doors.

6.13 Instantiation of Generic Carriage Door

We use the *GCDoor* development as a pattern to model emergency and service doors. The instantiation is similar for both kind of doors: specific details for each type of door are added later. We abstract ourselves from these details and focus in the instantiation of one of the doors: *emergency doors*.

The pattern context is defined by contexts $GCDoor\_C0$ (and context $GCTrack\_C0$) in Fig. 6.23 and $GCDoor\_C1$ in Fig. 6.25(a). The parameterisation context seen by the instance results from the context seen by the abstract machine $CarriageDoors$ as illustrated in Fig. 6.28(a). *CarriageDoors_C0* does not contain all the sets and constants that need to be instantiated. Therefore *CarriageDoors_C1* is created based on the pattern context *GCDoor_C1* (Fig. 6.28(b)).

```
machine GCDoor_M2 refines GCDoor_M1  sees GCDoor_C1

variables generic_door generic_door_state isolated_door locked_doors door_opening_device obstructed_door
          command command_doors command_type command_state

invariants
  @inv1 isolated_door ⊆ DOOR
  @inv2 ∀cmd,d·cmd ∈ command ∧ command_type(cmd)∈{OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
        ∧d ∈ DOOR∧d ∈ command_doors(cmd)∧command_state(cmd)=SUCCESS ∧ d ∉ isolated_door⇒ generic_door_state(d)=OPEN
  @inv3 ∀cmd,d·cmd ∈ command ∧ command_type(cmd)∈{CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
        ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd)∧command_state(cmd)=SUCCESS∧d ∉ isolated_door⇒ generic_door_state(d)=CLOSED
  @inv4 ∀cmd,d·cmd ∈ command ∧ command_type(cmd)=ISOLATE_DOORS ∧ d ∈ DOOR
        ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∈ isolated_door
  @inv5 ∀cmd,d·cmd ∈ command ∧ command_type(cmd)=REMOVE_ISOLATION_DOORS
        ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∉ isolated_door
  @inv6 ∀d·d∈isolated_door ∧ d ∈ dom(generic_door_state)⇒  generic_door_state(d)∈{OPEN, CLOSED}
  @inv7 ∀cmd1,cmd2·cmd1∈command ∧ cmd2∈command ∧ cmd1≠cmd2
        ∧ command_state(cmd1)≠EXECUTED ∧ command_state(cmd2)≠EXECUTED ⇒command_doors(cmd1)∩command_doors(cmd2)=∅
  @inv8 ∀d·d∈obstructed_door ⇒ generic_door_state(d)=OPEN
```

(a) Variables, invariants

```
event commandIsolationDoors refines otherCommandDoors
  any doors cmd cmd_type
  where
    @grd doors ⊆ generic_door
    @grd1 cmd_type ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
    @grd2 cmd ∈ COMMAND\command
    @grd3 ∀cmd1· cmd1∈command
            ∧ command_state(cmd1)≠EXECUTED
            ⇒doors∩command_doors(cmd1)=∅
    @grd4 doors ≠∅
    @grd5 cmd_type = ISOLATE_DOORS ⇔ (doors∩isolated_door = ∅ )
    @grd6 cmd_type = REMOVE_ISOLATION_DOORS ⇔ isolated_door≠∅
          ∧ doors∩isolated_door≠∅
  then
    @act1 command_state(cmd)≔START
    @act2 command_doors(cmd)≔doors
    @act3 command ≔ command ∪ {cmd}
    @act4 command_type(cmd)≔cmd_type
end

event updateIsolationCmdState refines updateCmdState
  any state cmd
  where
    @grd cmd ∈ command
    @grd1 state ∈ COMMAND_STATE\{START,EXECUTED}
    @grd2 command_state(cmd)=START
    @grd3 command_type(cmd)
            ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
    @grd4 (command_type(cmd) = ISOLATE_DOORS
          ∧ (∃d·d∈command_doors(cmd) ∧ d ∉isolated_door))
          ∨ (command_type(cmd) = REMOVE_ISOLATION_DOORS
          ∧ (∃d·d∈command_doors(cmd) ∧ d ∈isolated_door))
          ⇔ state = FAIL
  then
    @act1 command_state(cmd)≔state
end

event executedLogCmdState refines updateCmdState
  any cmd
  where
    @guard3 cmd ∈ command
    @guard1 command_state(cmd)∈{FAIL,SUCCESS}
  with
    @state state = EXECUTED
  then
    @act1 command_state(cmd)≔EXECUTED
end

event isolateDoor
  any d cmd
  where
    @grd d ∈ generic_door\isolated_door
    @grd1 cmd ∈ command
    @grd2 command_state(cmd)=START
    @grd3 d ∈ command_doors(cmd)
    @grd4 command_type(cmd) = ISOLATE_DOORS
    @grd5 generic_door_state(d)∈{OPEN, CLOSED}
  then
    @act1 isolated_door≔ isolated_door ∪ {d}
end

event removeIsolatedDoor
  any d cmd
  where
    @grd d ∈ isolated_door
    @grd1 cmd ∈ command
    @grd3 d ∈ command_doors(cmd)
    @grd4 command_type(cmd) = REMOVE_ISOLATION_DOORS
    @grd2 command_state(cmd)=START
    @grd5 generic_door_state(d)∈{OPEN, CLOSED}
  then
    @act1 isolated_door≔ isolated_door \ {d}
end
```

(b) Some events in *GCDoor_M2*

FIGURE 6.27: Excerpt of machine *GCDoor_M2*

Following the steps suggested in Sect. 3.5.2, we create the instantiation refinement for emergency carriage doors as seen in Fig. 6.29. As expected, the generic sets and constants are replaced by the instance sets existing in contexts *CarriageDoors_C0* and *CarriageDoors_C1*. Moreover, generic variables are renamed to fit the instance and be a refinement of abstract machine *CarriageDoors*. The same happens to generic events *addDoor* and *removeDoor*.

Comparing the abstract machine of the pattern *GCDoor_M0* and the last refinement of our initial development *CarriageDoors*, we realise that they are similar but not a perfect match. *CarriageDoors* events contains some additional parameters and guards resulting from the previous refinements. For instance, event *closeDoors* in *CarriageDoors* (Fig. 6.30(b)) contains an additional parameter *cds* compared to event *closeDoors* in

```
context CarriageDoor_C0

constants PLATFORM DOOR_SIDE PLATFORM_SIDE CLOSED OPEN
          DOOR_CARRIAGE

sets DOOR DOOR_STATE CDV SIDE CARRIAGE

axioms
  @MetroSystem_C1_axm1 partition(DOOR_STATE, {OPEN}, {CLOSED})
  @MetroSystem_C1_axm2 PLATFORM ⊆ CDV
  @Train_C1_axm2 DOOR_CARRIAGE ∈ DOOR → CARRIAGE
  @Train_C1_axm3 ∀c·c∈ran(DOOR_CARRIAGE)⇒DOOR_CARRIAGE∼[{c}]≠∅
  @Train_C2_axm4 DOOR_SIDE ∈ DOOR → SIDE
  @Train_C2_axm5 PLATFORM_SIDE ∈ PLATFORM → SIDE
  @Train_C2_axm6 PLATFORM ≠∅
end
```

(a) Context *CarriageDoors_C0*

```
context CarriageDoor_C1 extends CarriageDoor_C0

constants MANUAL_PLATFORM MANUAL_INTERNAL AUTOMATIC_CENTRAL_DOOR
START FAIL SUCCESS EXECUTED OPEN_RIGHT_DOORS OPEN_LEFT_DOORS
CLOSE_RIGHT_DOORS CLOSE_LEFT_DOORS ISOLATE_DOORS REMOVE_ISOLATION_DOORS

sets OPEN_DEV COMD_ST COMD_TYPE COMD

axioms
 @axm1 partition(OPEN_DEV, {MANUAL_PLATFORM}, {MANUAL_INTERNAL},
                                   {AUTOMATIC_CENTRAL_DOOR})
  @axm2 partition(COMD_ST, {START}, {FAIL}, {SUCCESS},{EXECUTED})
  @axm3 partition(COMD_TYPE,{OPEN_RIGHT_DOORS},{OPEN_LEFT_DOORS},
                {CLOSE_RIGHT_DOORS}, {CLOSE_LEFT_DOORS},{ISOLATE_DOORS},
               {REMOVE_ISOLATION_DOORS})
end
```

(b) Context *CarriageDoors_C1*

FIGURE 6.28: Parameterisation context *CarriageDoors_C0* plus additional context *CarriageDoors_C1*

INSTANTIATED REFINEMENT IEmergencyDoor_M2
INSTANTIATES *GCDoors_M2* **VIA** *GCDoor_C0 GCDoor_C1*
REFINES *CarriageDoors* `/* abstract machine */`
SEES *CarriageDoors_C0 CarriageDoors_C1* `/* instance contexts */`
REPLACE
  SETS $GEN\_CARRIAGE := CARRIAGE$ $DOOR := DOOR$
    $DOOR\_STATE := DOOR\_STATE$ $SIDE := SIDE$
    $OPENING\_DEVICE := OPEN\_DEV$ $COMMAND\_STATE := COMD\_ST$
    $COMMAND := COMD$ $COMMAND\_TYPE := COMD\_TYPE$
  CONSTANTS $GEN\_DOOR\_CARRIAGE := DOOR\_CARRIAGE$
    $OPEN := OPEN$ $PLATFORM := PLATFORM$
    $CLOSED := CLOSED$ $PLATFORM\_SIDE := PLATFORM\_SIDE$
    ...
RENAME `/*rename variables, events and params*/`
  VARIABLES $generic\_doors := carriage\_doors$ $generic\_door\_state := carriage\_ds$
  EVENTS $addDoor := allocateCarriageTrain$ $removeDoor := removeCarriageTrain$
END

FIGURE 6.29: Instantiated Refinement *IEmergencyDoor_M2*

GCDoor_M0 (Fig. 6.30(a)). Some customisation is tolerable in the generic event to ensure that the instantiation of *GCDoor_M0.closeDoors* refines *CarriageDoors.closeDoors* by adding a parameter that match *cds* and respective guard *grd*13.

The customisation can be realised by a (shared event) composition of event *GCDoor_M0.closeDoors* with another event that introduces the additional parameter *cds* and guard $cds = carriage\_ds$. The monotonicity of the shared event composition allows the composed pattern to be instantiated as initially desired. Another option is to introduce an additional step: the last machine of the refinement chain before the

```
event closeDoors
  any ds
  where
    @grd ds ⊆ DOOR
    @grd1 ds ⊆ dom(generic_door_state)
    @grd2 generic_door_state[ds]={OPEN}
    @grd3 ds ≠∅
  then
    @act1 generic_door_state≔generic_door_state
                                     ⊲ (ds×{CLOSED})
end
```

(a) Event *GCDoor_M0.closeDoors*

```
event closeDoors
  any ds cds
  where
    @typing_cds cds ∈ ℙ(DOOR × DOOR_STATE)
    @typing_ds ds ∈ ℙ(DOOR)
    @grd11 ds ⊆ dom(carriage_ds)
    @grd12 carriage_ds[ds]={OPEN}
    @grd13 cds = carriage_ds
  then
    @act2 carriage_ds≔carriage_ds ⊲ (ds×{CLOSED})
end
```

(b) Event *CarriageDoors.closeDoors*

```
event closeDoors refines closeDoors
  any ds
  where
    @typing_ds ds ∈ ℙ(DOOR)
    @grd11 ds ⊆ dom(carriage_ds)
    @grd12 carriage_ds[ds]={OPEN}
  with
    @cds cds = carriage_ds
    @t t ∈ TRAIN
    @closed closed ∈ BOOL
  then
    @act2 carriage_ds≔carriage_ds ⊲ (ds×{CLOSED})
end
```

(c) Event *CarriageDoorsInst_M0.closeDoors*

FIGURE 6.30: Event *closeDoors* in the pattern and instance; they differ in the parameters, guards and witnesses

instantiation (in our case study, machine *CarriageDoors*) is refined. The resulting refinement machine (*CarriageDoorsInst_M0*) refines the first instantiation machine (i.e. $CarriageDoors \sqsubseteq CarriageDoorsInst\_M0 \sqsubseteq EmergencyDoors\_M0$) "customising" the instantiation. Therefore the additional parameters (and respective guards) can disappear by means of witnesses as can be seen in Fig. 6.30(c). Ideally we aim to have a syntactic match (after instantiation) between the pattern and the initial instantiantion. Nevertheless a valid refinement is enough to apply the instantiation.

An instance machine *EmergencyDoor_M2* (Fig. 6.31) is similar to *GCDoor_M2* apart from the replacements and renaming applied in *IEmergencyDoor_M2* (cf. Figs. 6.27, Fig. 6.29 and Fig. 6.31). That machine can be further refined (and decomposed) introducing the specific details related to emergency doors. The instantiation of the service doors follows the same steps.

Statistics: In Table 6.3, we describe the statistics of the development in terms of variables, events and proof obligations (and how many POs were automatically discharged by the theorem prover of the Rodin platform) for each refinement step. Almost 3/4 of the proof obligations are automatically discharged.

This case study was carried out under the following conditions:

- Rodin v2.1
- Shared Event Composition plug-in v1.3.1

```
machine EmergencyDoors_M2 refines EmergencyDoors_M1  sees CarriageDoors_C1

variables carriage_door carriage_ds isolated_door locked_doors door_opening_device obstructed_door
command command_doors command_type command_state

invariants
 @inv1 isolated_door ⊆ DOOR
 @inv2 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)∈{OPEN_RIGHT_DOORS,OPEN_LEFT_DOORS}
        ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS
        ∧ d ∉ isolated_door⇒ carriage_ds(d)=OPEN
 @inv3 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)∈{CLOSE_RIGHT_DOORS,CLOSE_LEFT_DOORS}
        ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS
        ∧ d ∉ isolated_door⇒ carriage_ds(d)=CLOSED
 @inv4 ∀d· d∈isolated_door ∧ d ∈ dom(carriage_ds)⇒  carriage_ds(d)∈{OPEN, CLOSED}
 @inv5 ∀cmd1,cmd2· cmd1∈command ∧ cmd2∈command ∧ cmd1≠cmd2
         ∧ command_state(cmd1)≠EXECUTED
         ∧ command_state(cmd2)≠EXECUTED ⇒command_doors(cmd1)∩command_doors(cmd2)=∅
 @inv6 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)=ISOLATE_DOORS ∧ d ∈ DOOR
         ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∈ isolated_door
 @inv7 ∀cmd,d· cmd ∈ command ∧ command_type(cmd)=REMOVE_ISOLATION_DOORS
         ∧ d ∈ DOOR ∧ d ∈ command_doors(cmd) ∧ command_state(cmd)=SUCCESS ⇒ d∉ isolated_door
 @inv8 ∀d· d∈obstructed_door ⇒ carriage_ds(d)=OPEN
```

(a) Variables, invariants

```
event commandIsolationDoors refines otherCommandDoors
 any doors cmd cmd_type
 where
  @guard doors ⊆ carriage_door
  @guard1 cmd_type
       ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
  @guard3 cmd ∈ COMD\command
  @guard4 ∀cmd1 · cmd1∈command
           ∧ command_state(cmd1)≠EXECUTED
           ⇒doors∩command_doors(cmd1)=∅
  @grd4 doors ≠∅
  @grd5 cmd_type = ISOLATE_DOORS ⇔ (doors∩isolated_door = ∅ )
  @grd6 cmd_type = REMOVE_ISOLATION_DOORS ⇔ isolated_door≠∅
         ∧ doors∩isolated_door≠∅
 then
  @act1 command_state(cmd)≔START
  @act2 command_doors(cmd)≔doors
  @act3 command ≔ command ∪ {cmd}
  @act4 command_type(cmd)≔cmd_type
end

event updateIsolationCmdState refines updateCmdState
 any state cmd
 where
  @guard3 cmd ∈ command
  @guard state ∈ COMD_ST\{START,EXECUTED}
  @guard1 command_state(cmd)=START
  @guard5 command_type(cmd)
             ∈ {ISOLATE_DOORS,REMOVE_ISOLATION_DOORS}
  @grd3 (command_type(cmd) = ISOLATE_DOORS
        ∧ (∃d · d∈command_doors(cmd) ∧ d ∉isolated_door))
        ∨ (command_type(cmd) = REMOVE_ISOLATION_DOORS
        ∧ (∃d · d∈command_doors(cmd) ∧ d ∈isolated_door))
        ⇔ state = FAIL
 then
  @act1 command_state(cmd)≔state
end

event executedLogCmdState refines updateCmdState
 any cmd
 where
  @guard3 cmd ∈ command
  @guard1 command_state(cmd)∈{FAIL,SUCCESS}
 with
  @state state = EXECUTED
 then
  @act1 command_state(cmd)≔EXECUTED
end

event isolateDoor
 any d cmd
 where
  @guard d ∈ carriage_door\isolated_door
  @guard1 cmd ∈ command
  @guard2 command_state(cmd)=START
  @guard3 d ∈ command_doors(cmd)
  @guard4 command_type(cmd) = ISOLATE_DOORS
  @guard5 carriage_ds(d)∈{OPEN, CLOSED}
 then
  @act1 isolated_door≔ isolated_door ∪ {d}
end

event removeIsolatedDoor
 any d cmd
 where
  @guard d ∈ isolated_door
  @guard1 cmd ∈ command
  @guard3 d ∈ command_doors(cmd)
  @guard4 command_type(cmd) = REMOVE_ISOLATION_DOORS
  @guard2 command_state(cmd)=START
  @guard5 carriage_ds(d)∈{OPEN, CLOSED}
 then
  @act1 isolated_door≔ isolated_door \ {d}
end
```

(b) Some events in *EmergencyDoor_M2*

FIGURE 6.31: Excerpt of instantiated machine *EmergencyDoor_M2*

- Model Decomposition plug-in v1.2.1
- Instantiation was done manually (currently tool support is not available).
- ProB v2.1.2
- Camille Text Editor 2.0.1

Although we are interested mainly interested in safety properties, the model checker ProB [141] proved to be very useful as a complementary tool during the development of this case study. In some stages of the development, all the proof obligations were

	Variables	Events	ProofObligations/Auto
TransitiveClosureCtx	–	–	10/10
MetroSystem_C0	–	–	5/3
MetroSystem_C1	–	–	0/0
MetroSystem_M0	7	10	75/64
MetroSystem_M1	10	13	17/17
MetroSystem_M2	12	17	78/57
MetroSystem_M3	12	17	24/22
Track	4	10	0/0
Train	7	14	0/0
Middleware	1	4	0/0
Train_M1	9	16	74/52
Train_M2	13	21	155/79
Train_M3	12	21	65/24
Train_M4	14	21	119/89
LeaderCarriage	9	21	0/0
Carriage	5	11	0/0
Carriage_M1	6	11	28/21
CarriageInterface	4	11	0/0
CarriageDoors	2	5	0/0
CarriageDoorsInst_M0	2	5	2/1
GCDoor_M0	2	5	6/6
GCDoor_M1	9	15	81/80
GCDoor_M2	10	22	170/153
Total			909/678(74.6%)

Table 6.3: Statistics of the metro system case study

discharged but with ProB we discovered that the system was deadlocked due to some missing detail. In large developments, these situations possibly occur more frequently. Therefore we suggest discharging the proof obligations to ensure the safety properties are preserved and run the ProB model checker to confirm that the system actually is free from deadlocks.

6.14 Discussion: Conclusions and Lessons Learned

We modelled a metro system case study, starting by proving its global properties through several refinement steps. Afterwards, due to an architectural decision and to alleviate the problem of modelling and handling a large system in one single machine, the system is decomposed in three sub-components. We further refine one of the resulting sub-components (*Train*), introducing several details in four refinements levels. Then again, due to the number of proof obligations, to achieve separation of aspects and to ease the further developments, we decompose it into two sub-components: *LeaderCarriage* and *Carriage*. Since we are interested in modelling carriage doors, sub-component *Carriage* is refined and afterwards decomposed originating sub-component *CarriageDoors*. Benefiting from an existing generic development for carriage doors *GCDoor*, we consider this development as a pattern and instantiate two kind of carriage doors: *service* and *emergency* doors. Although the instantiation is similar for both types of doors, the resulting instances can be further refined independently. Using generic instantiation, we avoid having to prove the proof obligations regarding the pattern *GCDoor*: *GCDoor_M0*, *GCDoor_M1* and *GCDoor_M2* (in the overall 257 POs). This figure only considers the instantiation of emergency doors (the instantiation of service doors would imply twice the number of POs).

From the experience of other developments involving a large number of refinements levels or refinements with large models, the development tools reach a point where it is not possible to edit the model due to the high amount of resources required to do it (or it is done very slowly). The decomposition is a possible solution that alleviates this issue by splitting the model into more tool manageable dimensions. Following a top-down approach, developed models become more complex in each refinement step. Nevertheless by applying decomposition, we alleviate the consequences of such complexity by separating concerns (architecture approach), decreasing the number of events and variables per sub-component which results in models that are more manageable from a tool point of view. Moreover, for each refinement, the properties (added as requirements) are preserved. Using generic instantiation, we avoid proving the pattern proof obligations *GCDoor*. Therefore we reach our goal of reusing existing developments as much as possible and discharge as little proof obligations as possible. Even the interactive proofs were relatively easy to discharge once the correct tactic was discovered. This task would be more difficult without the decomposition due to the elevated number of hypotheses to considered for each PO. Nevertheless we believe that the effort of discharging proof obligations could be minimised by having a way to reuse tactics. In particular when the same steps are followed to discharge similar POs.

In a combination of refinement and instantiation, we learned that the abstract machine and the abstract pattern do not necessarily match perfectly. In particular, some extra guards and parameters may exist resulting from previous refinements in the instance. Nevertheless the generic model can still be reused. We can (shared event) compose the pattern with another machine in a way that the resulting events include the additional parameters and guards to guarantee a valid refinement. Another interesting conclusion is that throughout an instantiation, it is possible not to use all the generic events. A subset of generic events can be instantiated in opposition to instantiate all. This a consequence of the event refinements that only depend on abstract and concrete events. Nevertheless this only applies for safety properties. If we are interested in liveness properties, the exclusion of a generic event may result in a system deadlock.

With this case study we aim to illustrate the application of decomposition and generic instantiation as techniques to help the development of formal models. Following these techniques, the development is structured in a way that simplifies the model by separating concerns and aspects and decreases the number of proof obligations to be discharged. Although we use Event-B, these techniques are generic enough to suit other formal notations and other case studies. Formal methods has been widely used to validate requirements of real systems. The systems are formally described and properties are checked to be preserved whenever a system transition occurs. Usually this result in complex models with several properties to be preserved, therefore structuring and reusability are pursued to facilitate the development. Lutz [114] describes the reuse of formal methods when analysing the requirements and designing the software between two spacecrafts' formal models. Stepney *et al.* [177, 178] propose patterns to be applied to formal methods in system engineering. Using the Z notation, several patterns (and anti-patterns) are identified and catalogued to fit particular kind of models. These patterns introduce structure to the models and aim to aid formal model developers to choose the best approach to model a system, using some examples. Although the patterns are expressed for Z, they are generic enough to be applied to other notations. Comparing with the development of our case study, the instantiation of service and emergency doors corresponds to the Z promotion, where a global system is specified in terms of multiple instances of local states and operations. Although there is not an explicit separation of local and global states in our case study, service and emergency doors states are connected to the state of $CarriageDoor$ and we even use decomposition, instantiation and refactoring (called meaning preservation refactoring steps in Z promotion) to fit into a specific pattern. [177] suggests template support and architecture patterns to be supported by tools, something that currently does not happen. We have a similar viewpoint and we would like to address this issue in the future. Templates could be customised according to the modeller's needs and selected from an existing list, perhaps categorised as suggested in [177].

Butler [44] uses the shared event approach in classical B to decompose a railway system into three sub-components: Train, Track and Communication. The system is modelled and reasoned as a whole in an event-based approach, both the physical system and the desired control behaviour. Our case study follows a similar methodology applied to a metro system following the same shared event style. Moreover we introduce more requirements regarding the trains and the carriage doors, expressed through the use of decomposition and generic instantiation.