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 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.

![Figure 6.1: Different types of Switches: divergent and convergent](image)

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 (\textit{MetroSystem\textsubscript{M0}} to \textit{MetroSystem\textsubscript{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 \url{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.

![Diagram](image)

**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 \to CDV
  axm2 cdvfn = CDV \to CDV
  axm3 tcl ∈ cdvrel → cdvrel
  axm4 ∀r.(cdvrel ⇒ r ∈ tcl(r)) // r included in tcl(r)
  axm5 ∀r.(cdvrel ⇒ r \cdot tcl(r) ⊆ tcl(r)) // unfolding included in tcl(r)
  axm6 ∀r,t.(cdvrel ∧ r \cdot t ⇒ tcl(r \cdot t)) \cdot tcl(r) is least
  theorem @thm1 cdvfn ⊆ cdvrel
  theorem @thm2 ∀r.(cdvrel ⇒ tcl(r) = r \cup \{r; tcl(r)\}) // tcl(r) is a fixed point
  theorem @thm3 ∀t;cdvrel(t ∩ s \cdot \text{not} s \cdot s = s) ⇒ tcl(t \cdot n(CDV < id)) = \varnothing
  theorem @thm4 tcl(\varnothing) = \varnothing
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 ∈ CDV \leftrightarrow CDV, the transitive closure of r is the least x satisfying x = r ∪ 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 → 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
for net is expressed by axiom axm11. Theorems thm1 states that net preserves transitive closure.

- Switches (aiguillages in French) are sections (axm3) that cannot be connected to each others (axm6). 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 (axm10).

- Non-switches have at most one successor and at most one predecessor (axm9).

Besides the global property described before defined by invariant inv13 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 \rightarrow CDV. next is a subset of net (inv1) preserving the transitive closure property as described by theorem thm1,thm2 and does not have loops (thm3). Sections occupied by trains are represented by variable occp. These sections also preserve the transitive closure property as seen by thm4.

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 (inv11).
The system proceeds as follows: trains modelled in the system circulate by entering and leaving sections (events \textit{enterCDV} and \textit{leaveCDV} in Fig. 6.7(b)), ensuring that the next section is not occupied (\textit{grd9} in \textit{enterCDV}) and updating all the sections occupied by the train (\textit{act1} and \textit{act2} in both events). At this abstract level, event \textit{modifyTrain} modifies a train defining the set of occupied sections for a train \( t \). A train changes speed, brakes or stops braking in events \textit{changeSpeed}, \textit{brake} and \textit{stopBraking}. When event \textit{brake} occurs, train \( t \) is added to a set of braking trains (variable \textit{braking}). Variable \textit{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 \textit{switchChangeDiv} and \textit{switchChangeCnv} as seen in Fig. 6.7(b).

6.4 First Refinement: \textit{MetroSystem_M1}

\textit{MetroSystem_M1} refines \textit{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 \textit{stopping distance downstream}.

The messages are represented by variables \textit{tmsgs} that stores the messages (buffer) sent from the tracks and \textit{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
Figure 6.7: Variables, invariant and events of \textit{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 grd3 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 grd3 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](image)

(a) Variables and invariants in MetroSystem_M1

![Events brake, sendTrainMsg, recvTrainMsg, stopBraking, toggleEmergencyButton](image)

(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 ($grd11\_leaveCDV$). Adding/removing train doors is modelled in events $addDoor\_Train$ and $removeDoor\_Train$ 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 $grd3\_openDoor$ 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.
Chapter 6 Case Study

machine MetroSystem_M2 refines MetroSystem_M1 sees MetroSystem_C1

variables next trns occpA occpZ braking speed teqs permit
door door_state emergency_button

invariants
@inv1 door_state ∈ DOOR ⇒ DOOR_STATE
@inv2 door e trns ⇒ P(DOOR)
@inv3 ∀t, t' : ∀e : done(door) ∧ t < done(door) ⇒ t + t' = e
door(t) ∧ door(t') = e
@inv4 ∀t : ∀e = done(door) ⇒ 3d🏼door(t) ∧ door_state[d] = OPEN
@inv5 ∀t : ∀e = done(door) ⇒ speed(t) > 0
door(t) ∧ door_state = CLOSED
@inv6 ∀t : ∀e = done(door) ∧ e = door(t) ∧ PLATFORM ∧ occpA = TRUE
@inv7 ∀t : ∀e = done(door) ∧ e = door(t) ∧ door_state = OPEN
platform ∧ occpA = TRUE ∧ emergency_button

context MetroSystem_C1 extends MetroSystem_C0
constants OPEN CLOSED PLATFORM
sets DOOR_STATE DOOR
axioms
@axm1 @grd2 @act1 @grd5 @grd4 @grd2 @grd1
@axm3 @grd3 @act1 @grd1
@axm4 @grd1

@act1 @grd1 @grd3 @grd2 @grd1

context MetroSystem_C1 extends MetroSystem_C0
constants OPEN CLOSED PLATFORM
sets DOOR_STATE DOOR
axioms
@axm1 @grd2 @act1 @grd5 @grd4 @grd2 @grd1
@axm3 @grd3 @act1 @grd1
@axm4 @grd1

(a) Context MetroSystem_C1

(b) Variables, invariants in MetroSystem_M2

(c) Some events of MetroSystem_M2

Figure 6.9: Excerpt of MetroSystem_M2
Some guards need to be rewritten in the refined events. For instance, guard $grd_{10}$ 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:

\[
\forall tt: tt \in trns \land card((occp \cup \{c2 \rightarrow t1\})^{-1}[[tt]]) > 1 \Rightarrow (occpZ \uplus \{t1 \rightarrow c2\})(tt) \neq occpA(tt)
\]

to:

\[
\forall tt: tt \in dom(occpZ) \land card((occp \cup \{c2 \rightarrow t1\})^{-1}[[tt]]) > 1 \Rightarrow (occpZ \uplus \{t1 \rightarrow c2\})(tt) \neq occpA(tt) \text{ (Fig. 6.11)}.
\]

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 $grd_3$ in $toggleEmergencyButton$ is rewritten to separate variables $occp$ and $door$. In this case, an additional parameter $occpTrns$ representing the variable $occp$ is added ($grd_4$). 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 $grd_3$. Similarly guard $grd_4$ 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
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 ∈ CDV ↔ trns (inv4 in Fig. 6.7(a))`; in `Track` is `occp ∈ CDV ↔ TRAIN` (which is the same as theorem `typing_occp : occp ∈ P(CDV × 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. `inv3` in Fig. 6.7(a)). Variables `occpA` and `occpZ` are subject to the same procedure where the original invariant is a total function `trns → CDV` and in the sub-component both become `P(TRAIN × 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

![Figure 6.11: Preparation step before decomposition of MetroSystem_M3](image-url)
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 is 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
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.
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 latter occurs when the emergency button of a train is active.
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.
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: \textit{SIDE} corresponding to which side a carriage door or a platform is located (constants \textit{LEFT} or \textit{RIGHT}) and \textit{TRAI\textit{N} STATE} that defines the state of a train (\textit{MAINTENANCE}, \textit{MANUAL} or \textit{AUTOMATIC}). There are some new constants added as well: \textit{CABIN CARRIAGE} defined as a sub-
set of \textit{CARRIAGE}, \textit{NUMBER\_CABIN\_CARRIAGE} defining the number of cabin carriages allowed per train, \textit{DOOR\_SIDE} defined as a total function between \textit{DOOR} and \textit{SIDE} representing which side a door is located, \textit{MAX\_SPEED} defining the upper speed limit for running a train before the activation of the emergency brake and \textit{PLATFORM\_SIDE} defining the side of a platform.

Figure 6.17 shows \textit{Train\_M2} where several new variables are introduced: \textit{leader\_carriage} defining the leader carriage for a train (inv6), \textit{trns\_state} defining the state of a train (inv8), \textit{emergency\_brake} that defines which trains have the emergency brake activated (inv11) and \textit{carriage\_door\_state} defining the state of the carriage doors (inv15). Moreover \textit{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 \textit{door} that disappears in this refinement level, plus some gluing invariants: inv1, inv5 and theorem \textit{thm2} state that the range of \textit{door} for a train \textit{t} is the same as the range of \textit{door\_train\_carriage} as long as \textit{t} has doors.

Property 1 is expressed by \textit{inv9}. Property 2 is expressed by \textit{inv10} and property 3 by \textit{inv12}. \textit{inv13} and \textit{inv14} state that the doors in the domain of \textit{door\_state} are the same as the ones in carriage\_door\_state and therefore their state must match. Theorem \textit{thm1} relates the carriages doors with variables \textit{door\_train\_carriage} and \textit{train\_carriage}. Theorem \textit{thm3} states that the domain of carriage\_door\_state is a subset of the domain of \textit{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 \textit{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 deallocatte the leader carriage in event \textit{deallocateLeaderCabinCarriageTrain}, the train must be in maintenance. A train change state in event \textit{modifyTrain} to change to \textit{MAINTENANCE}, the train must be stopped (grd2); for the other states, the number of cabin carriages must be \textit{NUMBER\_CABIN\_CARRIAGE} and a leading carriage have to be allocated already (grd3). Abstract event \textit{changeSpeed} is refined by four events: two to increase the speed (\textit{increaseSpeed} and \textit{increaseMaxSpeed} in Fig. 6.17(c)) and two to reduce the speed (\textit{reduceSpeed} and \textit{reduceMaxSpeed}). If the speed of a train is increasing in a way that is superior to \textit{MAX\_SPEED}, event \textit{increaseMaxSpeed} is enabled and if it occurs, the emergency\_brake is activated. If the current speed of a train is superior to \textit{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 \textit{reduceMaxSpeed}).
Chapter 6 Case Study

Figure 6.17: Excerpt of machine Train_M2
6.9 Third Refinement of \textit{Train}: \textit{Train\_M3}

As a continuation of the refinement of the train doors by carriage, we data refine variable \textit{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 \textit{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 (\textit{inv2}) and the leader carriage belongs to the set of carriage of that train (\textit{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 \textit{door\_state} disappears being refined by \textit{door\_carriage\_state} and gluing invariants \textit{inv1} and \textit{thm2}. Theorem \textit{thm1} is added to help with the proofs: the carriage doors of a train \textit{t} are the same as the doors defined by the constant \textit{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 \textit{inv2}, event \textit{allocateLeaderCabinCarriageTrain} needs to be strengthened by adding guard \textit{grd8}: this event is only enabled if the number of carriages for that train is equal to \textit{NUMBER\_CABIN\_CARRIAGE}. Also events \textit{allocateCarriageTrain} and \textit{removeCarriageTrain} require an additional guard (\textit{grd4} and \textit{grd11} respectively) stating that the events are only enabled if train \textit{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 \textit{openDoors} is strengthened with the inclusion of guard \textit{grd8}: the set of carriage doors \textit{ds} that are opened are located in the same side as the \textit{platform}.

6.10 Fourth Refinement of \textit{Train} and Second Decomposition: \textit{Train\_M4}

The fourth refinement of \textit{Train} corresponds to the preparation step before the decomposition. Context \textit{Train\_C4}, illustrated in Fig. 6.19(a), introduces an enumerated carrier set \textit{TRAIN\_MOVING\_STATE} defining the moving state of a train: \textit{MOVING}, \textit{NOT\_READY} (not ready to move) and \textit{NEUTRAL} (not moving but ready to move). We use additional control variables to help in the separation of aspects resulting in adding variables \textit{ready\_train} and \textit{train\_doors\_closed}. Both are total functions between \textit{trns} and \textit{BOOL} (\textit{inv1} and \textit{inv2} in Fig. 6.19(b)). \textit{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 (\textit{inv3})); \textit{train\_doors\_closed} defines trains that have all their doors closed (\textit{inv4}). These variables are somehow redundant and are mainly added as a preparation for the shared event decomposition: they will be allocated to \textit{LeaderCarriage} and represent a combination of states defined by \textit{Carriage} variables. They also simplify
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) = 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 glueing 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
  @act2
  @act1
  @grd4
  @grd3
  @grd2
  @grd1
  @guard
  @typing_t
  @act12
  @act11
  @act8
  @act7
  @act2
  @act1
  @grd1
  @grd8
  @grd7
  @grd5
  @grd4

(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
  @inv2
  @inv3
  @inv4
  @inv5
  @inv6

(b) Variables and invariants
```

```
event increaseMaxSpeed refines increaseMaxSpeed
  any \( t \in \mathbb{N} \)
  ggrd1: \( t \in \mathbb{N} \)
  ggrd2: \( t \in \mathbb{N} \)
  ggrd3: \( t \in \mathbb{N} \)
  ggrd4: trns.state(t) = MAINTENANCE
  ggrd5: \( t \in \mathbb{N} \)
  ggrd6: speed(t) = MAX_SPEED
  ggrd7: \( t \in \mathbb{N} \)
  ggrd8: speed(t) = MAX_SPEED
  ggrd9: train_doors_closed(t) = TRUE
  ggrd10: permit(t) = TRUE
  ggrd11: speed(t) = MAX_SPEED
  ggrd12: ready_train(t) = TRUE
  then
  gact1: speed(t) = \( t \in \mathbb{N} \)
  gact2: emergency_brake = emergency_brake \( (t) \)

(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):

<table>
<thead>
<tr>
<th>Variables</th>
<th>LeaderCarriage</th>
<th>Carriage</th>
</tr>
</thead>
<tbody>
<tr>
<td>trns, permit, braking, emergency_button</td>
<td>carriage_alarm, leader_carriage</td>
<td></td>
</tr>
<tr>
<td>trns_state, speed, emergency_brake</td>
<td>carriage_door_state, door_train_carriage</td>
<td></td>
</tr>
<tr>
<td>ready_train, train_doors_closed</td>
<td>train_carriage</td>
<td></td>
</tr>
</tbody>
</table>

Events
- openDoors, closeDoors
- activateEmergencyCarriageButton
- deactivateEmergencyCarriageButton
- allocateLeaderCabinCarriageTrain
- deallocateLeaderCabinCarriageTrain
- allocateCarriageTrain
- deallocateCarriageTrain
- modifyTrain, removeCarriageTrain
- increaseSpeed, increaseMaxSpeed
- reduceSpeed, reduceMaxSpeed
- recvTrainMsg, brake, stopBraking
- addTrain, enterCDV, leaveCDV

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
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_doors 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-
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 6.10. The resulting sub-events can be seen in Fig. 6.22.
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 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.
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
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 `inv6` to `inv9`). Property 2 is defined by invariant `inv2` (if a door is locked, then the door is not opened) and events `lockDoor/unlockDoor`. Property 3 is defined by variables `door_opening_device`, `inv3` and `inv11` (if a door is opened automatically, then a command has been issued to do so). Property 4 is defined by variable `obstructed_door`, `inv5` 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 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:
Chapter 6 Case Study

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)).
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 GDoor_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
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

RENAMES /*rename variables, events and params*/
VARIABLES generic_doors := carriage_doors generic_door_state := carriage_ds
EVENTS addDoor := allocateCarriageTrain removeDoor := removeCarriageTrain

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 grd13.

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
This case study was carried out under the following conditions:  

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

instantiation (in our case study, machine \emph{CarriageDoors}) is refined. The resulting refinement machine (\emph{CarriageDoorsInst}_M0) refines the first instantiation machine (i.e. \emph{CarriageDoors} \sqsupset \emph{CarriageDoorsInst}_M0 \sqsupset \emph{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 instantiation. Nevertheless a valid refinement is enough to apply the instantiation.

An instance machine \emph{EmergencyDoor}_M2 (Fig. 6.31) is similar to \emph{GCDoor}_M2 apart from the replacements and renaming applied in \emph{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.
In some stages of the development, all the proof obligations were

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
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 into 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.