Application of Decomposition and Generic Instantiation to a Metro System in Event-B[*]

Renato Silva[**]

School of Electronics and Computer Science
University of Southampton, UK
ras07r@ecs.soton.ac.uk

Abstract. It is believed that reusability in formal development should reduce the time and cost of formal modelling within a production environment. Event-B is a formal method that allows modelling and refinement of systems. Generic instantiation and decomposition are techniques that simplify formal developments by reusing existing models and avoiding re-proofs. We apply these techniques in Event-B for the development of a metro system case study based on safety properties. This work aims to give some guidelines of a practical way to develop large systems by instantiating generic models and (shared event) decomposing components into smaller sub-components.

Key words: Formal Methods, Event-B, Reusability, Generic Instantiation, Decomposition

1 Introduction

Reusability has always been sought in several areas as a way to reduce time, cost and improve the productivity of developments [1]. Examples can be found in areas like software, mathematics and even formal methods. The formal development of specifications in a "top-down" style starts with an abstract model of the envisaged system. Throughout refinements the initial model becomes less abstract and more concrete, closer to an implementation. As a consequence, there is a better view of the system as a whole and design decisions can be taken. Nonetheless refinements of a system bring complexity and tractability problems when the model augments in a way that becomes cumbersome to manage [2]. *Decomposition* [3] is precisely the process by which a single model can be split into various sub-components in a systematic fashion. The complexity of the whole model is decreased by studying, and thus refining, each part independently of the others [2]. Consequently the independent sub-components can be developed in parallel which is attractive in an industrial environment. *Generic Instantiation* [4] is another technique that can be seen as a way of reusing components

[*] Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (http://www.deploy-project.eu).

[**] R. Silva is sponsored by a Fundação Ciência e Tecnologia (FCT-Portugal) scholarship.

and solving difficulties raised by the construction of large and complex models [2,5]. The goal is to reuse generic developments (single model or a chain of refinements) and create components with similar properties instead of starting from scratch. Reusability is applied through the use of a *pattern* as the basic structure and afterwards each new component is generated through parameterisation.

These two techniques have been studied within the Event-B [6] formalism where tool support is available in the Rodin platform [7]. In this document, we share our experiences by applying these techniques during the development of a metro system focusing in safety properties. We combine shared event decomposition (where sub-components interact via synchronised shared events and shared states are not allow), generic instantiation and refinement to model particular aspects of the system. The requirements of the system are based on real requirements for carriage doors of a metro system The case study is developed in the Rodin platform using the available tools. We mainly use shared event 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.

A brief overview of the Event-B Language is given in Section 2. We briefly introduce decomposition and generic instantiation in Section 3 and Section 4 respectively. The metro system development is described in Section 5. We finish with conclusions and related work in Section 6.

2 Background

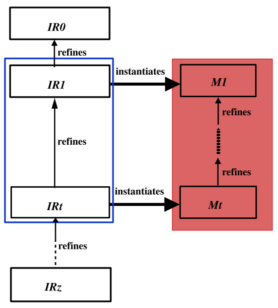
Event-B is a formal modelling method for developing *correct-by-construction* hardware and software systems. An Event-B specification is divided into two parts: a static part called *context* and a dynamic part called *machine*. A machine *SEES* as many contexts as desired. The context consists of sets, constants and assumptions (axioms) of the system. Sets in the context can be seen as a collection of elements or a type definition. An Event-B model is a state transition system where the state corresponds to *variables* v and transitions are represented by a collection of *events* evt in machines. The most general form of an event is: evt $\widehat{=}$ **any** t **where** $G(t,v)$ **then** $S(t,v,v')$ **end** , where t is a set of parameters, $G(t,v)$ is the enabling condition (called guard) and $S(t,v,v')$ is a before-after predicate computing after state v'. Essential to Event-B is the formulation of *invariants* $I(v)$: safety conditions to be preserved at all times.

To facilitate the construction of large-scale models, Event-B advocates the use of *refinement*: the process of gradually adding details to a model. An Event-B development is a sequence of models linked by refinement relations. It is said that a concrete model refines an abstract one. Abstract variables v are linked to concrete variables w by a *gluing invariant* $J(v,w)$. Any behaviour of the concrete model must be *simulated* by some behaviour of the abstract model, with respect to the gluing invariant $J(v,w)$. Rodin is an industrial-strength toolset supporting Event-B. Rodin provides an integrated modelling environment with a range of editors, modelling assistants, automatic generator of verification conditions and a set of automated provers tasked to discharge verification conditions.

3 Generic Instantiation

The generic instantiation approach for Event-B is applied by instantiating machines. The instances inherit properties from the generic development (pattern) and afterwards are *parameterised* by renaming/replacing those properties to more specific names according to the instance. Proofs obligations are generated to ensure that assumptions used in the pattern are satisfied in the instantiation. In that sense this approach avoids re-proof pattern proof obligations in the instantiation.

Consider a pattern that consists of a chain of refinements *M1, M2,... Mt* as seen in Fig. 1 (the shadowed part). *M1... Mt* are instantiated, originating *IR1... IRt* as long as the instance *IR1* refines *IR0*. The elements of the context (sets and constants) seen by the *M1... Mt* are in the *pattern context* Ctx_t. They are replaced by instance elements that must already exist in a context seen by the instantiated machine D_0 (*parameterisation context*). We create a generic Instantiated Refinement *IR* as seen in Fig. 1. *IR* instantiates the refinement



```
INSTANTIATED REFINEMENT IR
INSTANTIATES $M_t$ VIA $Ctx_t$
REFINES $IR_0$      /* abstract machine */
SEES $D_0$          /* context containing instance properties */
REPLACE             /* replace parameters in context $Ctx_t$ */
      SETS $S_1 := DS_1, \ldots, S_m := DS_m$ /* Sets or Constants
*/
      CONSTANTS $C_1 := DC_1, \ldots, C_n := DC_n$
RENAME      /*rename vars, events and params in $M_1/M_t$*/
      VARIABLES $v_1 := nv_1, \ldots, v_q := nv_q$
      EVENTS $ev_1 := nev_1$      /* optional */
                  $p_1 := np_1, \ldots, p_s := np_s$
                  ...
                  $ev_r := nev_r$
END
```

Fig. 1. An Instantiated Refinement

M_t via the parameterisation context Ctx_t. *IR* refines an abstract machine IR_0 and sees the context D_0 containing the instance properties. Generic sets and constants ($S_1, \ldots, S_m$ and $C_1, \ldots, C_n$) are replaced by instance ones existing in D_0 ($DS_1, \ldots, DS_m$ and $DC_1, \ldots, DC_n$). Variables, event names and parameters are renamed to fit the abstract machine *IR0*. During the creation of instances validity checks are required:

1. A static validation of replaced elements is required, e.g., a type must be replaced with a type, or a constant set and a constant with a constant.
2. All sets and constants should be replaced, i.e., no uninstantiated parameters.
3. Renaming the elements must be injective (not introducing name clashes) in order to reuse all the existing proof obligations.
4. Replacing sets does not have to be injective. Different sets in the instance can be replaced by the same generic set.

5. Only given sets (defined by the user) can be replaced. Built-in types such as integer numbers $\mathbb{Z}$ and boolean BOOL cannot be replaced.

The instantiation of refinements reuses the pattern proof obligations in the sense that the instantiation renames and replaces elements in the model but does not change the model itself (nor the respective properties). The correctness of the refinement instantiation relies on reusing the pattern proof obligations and ensuring the assumptions in the context parameterisation are satisfied in the instance.

4 Decomposition

In Event-B, decomposition of a component corresponds to distributing events and variables among sub-components. Shared event decomposition does not permit variable sharing and an event can be split into different sub-components. The sub-components can be further refined independently according to the monotonicity property of decomposition [8].

Figure 2 shows the shared event decomposition of machine S into machines T and W: variable $v1$ is allocated to machine T and variables $v2, v3$ are allocated to machine W). Event $e2$ is shared since uses both variables $v1$ and $v2$. Therefore during the decomposition, it must be decomposed into $e2'$ (containing only guards and actions related to $v1$) and $e2''$ (containing only guards and actions related to $v2$). Besides alleviating problems when dealing with complex specifications, decomposition also partition the proof obligations which are expected to be easier to be discharged in the sub-components. We follow a general

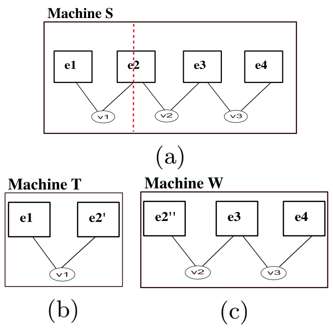


Fig. 2. Shared Event Decomposition of machine S into machines T and W with shared event *e2*

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 given decomposition technique (preparation step).
Stage 3 Apply decomposition.
Stage 4 Develop independently the decomposed parts.

5 Case Study: Metro System

The metro system case study describes a formal approach for the development of trains that circulate in a metro system. Trains circulate whenever they have permission. When stopping to load/unload, the doors can be opened/closed. As part of the safety requirements, all trains have an emergency button enabling its emergency brake. Throughout refinement steps, we introduce these requirements until we have enough information to split the model into smaller sub-components.

An overview of the entire development can be seen in Fig. 3 following the top-down guideline suggested in the previous section. **Stage 1** is expressed by refinements *Train* to *Train_M4* where global properties are introduced. *Train_M4* is also used as the preparation step before the decomposition corresponding to **Stage 2**. The model is decomposed into two parts: *LeaderCarriage*, and *Carriage* 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.

We are interested in refining the sub-component corresponding to carriages in order to introduce doors requirements. These requirements were 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. 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 steps of the development in the following sections.

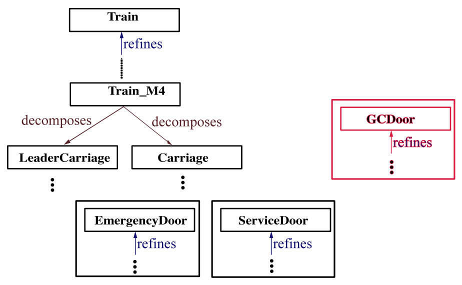


Fig. 3. Overall view of the safety metro system development

5.1 Refinements of *Train*

Train is refined several times before the decomposition. The properties for each refinement step are summarised below (due to the lack of space we do not describe them in detail[1]). Instead we focus on the resulting sub-component that is further refined and after decomposed.

[1] The model is available online at `http://eprints.ecs.soton.ac.uk/22195/`

Abstract Model: The model starts with the introduction of trains that can change speed, brake, open and close doors. A central control handles the circulation of trains by granting permissions. A train only moves if the central control grants permission. If a train receives a message disallowing the circulation, the train must brake.

First Refinement of Train: 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.

Second Refinement of Train: In this refinement of *Train*, carriages and doors requirements are added. We want to 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. 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 is superior to the 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.

Third Refinement of Train: Some additional properties related to the allocation of the leader carriage are defined: when a train already has a leader carriage, then it has the correct number of carriages and the leader carriage belongs to the set of carriages of that train. These two properties could have been included in the previous refinement but it was chosen to be added later due to the high number of proof obligations already existing in that refinement.

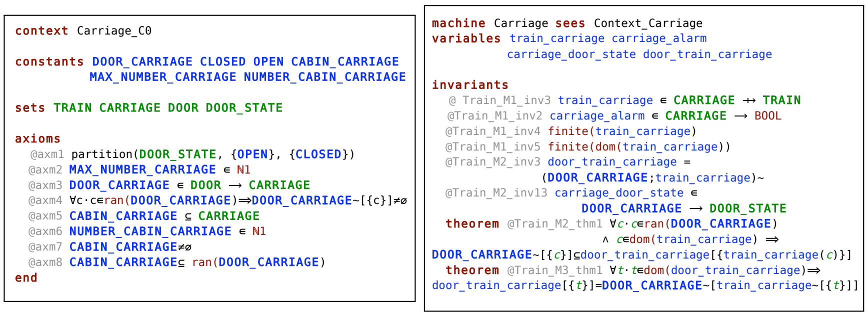
Fourth Refinement of Train: The four refinement of *Train* corresponds to the preparation step before the decomposition. We want to separate the aspects related to carriages from the aspects related to leader carriages:

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 (events $allocateCarriageTrain/$ $removeCarriageTrain$) , opens and closes carriage doors ($openDoors/closeDoors$), handles the carriage alarm ($activateEmergencyCarriageButton$, $deactivateEmergencyCarriageButton$, $deactivateEmergencyTrainButton$).

5.2 Machine *Carriage*

Carrier set *CARRIAGE* represents carriages, constant *MAX_NUMBER_CARRIAGE* defines the maximum number of carriages per train and *DOOR_CARRIAGE* (function between *DOOR* and *CARRIAGE*) relates doors and respective carriages. The latter is defined as a constant because the number of doors in a carriage does not change. Cabin carriages are a subset of carriages as described by axiom *axm*5 in Fig. 4. The variables related to carriages are allocated to sub-component *Carriage* (see Fig. 4): *train_carriage* defines which carriages belong to a train; *carriage_alarm* defines if the alarm in the carriage is enabled (TRUE) or not (FALSE); *carriage_door_state* defines if a carriage door is opened or closed; *door_train_carriage* relates trains, carriages and respective doors. We are interested in adding more details about the carriage doors, therefore we further refine *Carriage*.



```
context Carriage_C0

constants DOOR_CARRIAGE CLOSED OPEN CABIN_CARRIAGE
         MAX_NUMBER_CARRIAGE NUMBER_CABIN_CARRIAGE

sets TRAIN CARRIAGE DOOR DOOR_STATE

axioms
  @axm1 partition(DOOR_STATE, {OPEN}, {CLOSED})
  @axm2 MAX_NUMBER_CARRIAGE ∈ ℕ1
  @axm3 DOOR_CARRIAGE ∈ DOOR → CARRIAGE
  @axm4 ∀c·c∈ran(DOOR_CARRIAGE)⇒DOOR_CARRIAGE~[{c}]≠∅
  @axm5 CABIN_CARRIAGE ⊆ CARRIAGE
  @axm6 NUMBER_CABIN_CARRIAGE ∈ ℕ1
  @axm7 CABIN_CARRIAGE≠∅
  @axm8 CABIN_CARRIAGE⊆ ran(DOOR_CARRIAGE)
end
```

```
machine Carriage sees Context_Carriage
variables train_carriage carriage_alarm
          carriage_door_state door_train_carriage

invariants
   @ Train_M1_inv3 train_carriage ∈ CARRIAGE ⇸ TRAIN
  @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_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}]]
```

Fig. 4. Context *Carriage_C0*, variables and invariants of *Carriage*

5.3 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. 5.6.

Two variables are introduced in this refinement, representing the carriage doors (*carriage_door*) and their respective state (*carriage_ds*) as seen in Fig. 5. The last variable is used to data refine *carriage_door_state* that disappears. The gluing invariants for this data refinement is expressed by invariant *inv*4: 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. 5, *act*1 in event *openDoors* updates variable

```
variables carriage_alarm 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
```

```
event openDoors refines openDoors
  any t ds
  where
    @grd1 t ∈ TRAIN
    @grd2 t ∈ dom((DOOR_CARRIAGE;train_carriage)~)
    @grd3 ds ⊆ DOOR_CARRIAGE~[train_carriage~[{t}]]
    @grd4 ds ⊆ dom(carriage_ds)
    @grd5 carriage_ds[ds]={CLOSED}
  then
    @act1 carriage_ds≔carriage_ds⩤ (ds×{OPEN})
end

event closeDoors refines closeDoors
  any t ds closed cds
  where
    @grd1 t ∈ TRAIN
    @grd2 t ∈ dom(((train_carriage~);(DOOR_CARRIAGE~)))
    @grd4 ds ⊆((train_carriage~);(DOOR_CARRIAGE~))[{t}]
    @gd13 cds = carriage_ds
    @grd7 (∃d·d∈DOOR_CARRIAGE~[train_carriage~[{t}]]\ds
           ∧ cds(d)≠CLOSED) ⇔ closed = FALSE
    @grd11 ds ⊆ dom(carriage_ds)
    @grd12 carriage_ds[ds]={OPEN}
  then
    @act2 carriage_ds≔carriage_ds ⩤ (ds×{CLOSED})
end
```

```
event allocateCarriageTrain
refines allocateCarriageTrain
  any c t ds
  where
    @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 ds = DOOR_CARRIAGE~[{c}]
    @grd9 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
```

Fig. 5. Excerpt of machine *Carriage_M1*

carriage_ds instead of the abstract variable *carriage_door_state*. Also when carriage doors are allocated, both the new variables are assigned as seen in actions *act*3 and *act*4 for event *allocateCarriageTrain* (similar for *removeCarriageTrain*).

Comparing with the generic model of the 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 the Table 1.

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

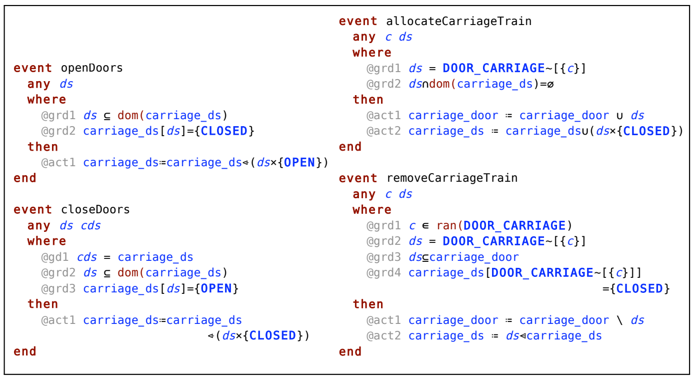
Table 1. *Carriage_M1* decomposition summary

5.4 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 deactivation of the emergency button and the allocation/de-allocation of the leader cabin carriage. Events *openDoors*, *closeDoors*, *allocateCarriageTrain* and *removeCarriageTrain* are shared.

5.5 Machine *CarriageDoor*

Sub-component *CarriageDoors* contains the variables related to carriage doors and the events result from splitting the original events as described in Table 1. The resulting sub-events can be seen in Fig. 6.



```
event openDoors
  any ds
  where
    @grd1 ds ⊆ dom(carriage_ds)
    @grd2 carriage_ds[ds]={CLOSED}
  then
    @act1 carriage_ds≔carriage_ds(ds×{OPEN})
end

event closeDoors
  any ds cds
  where
    @gd1 cds = carriage_ds
    @grd2 ds ⊆ dom(carriage_ds)
    @grd3 carriage_ds[ds]={OPEN}
  then
    @act1 carriage_ds≔carriage_ds
                              (ds×{CLOSED})
end
```

```
event allocateCarriageTrain
  any c ds
  where
    @grd1 ds = DOOR_CARRIAGE~[{c}]
    @grd2 ds∩dom(carriage_ds)=∅
  then
    @act1 carriage_door ≔ carriage_door ∪ ds
    @act2 carriage_ds ≔ carriage_ds∪(ds×{CLOSED})
end

event removeCarriageTrain
  any c ds
  where
    @grd1 c ∈ ran(DOOR_CARRIAGE)
    @grd2 ds = DOOR_CARRIAGE~[{c}]
    @grd3 ds⊆carriage_door
    @grd4 carriage_ds[DOOR_CARRIAGE~[{c}]]
                                   ={CLOSED}
  then
    @act1 carriage_door ≔ carriage_door \ ds
    @act2 carriage_ds ≔ ds⩤carriage_ds
end
```

Fig. 6. 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.

5.6 Generic Model: GCDoor

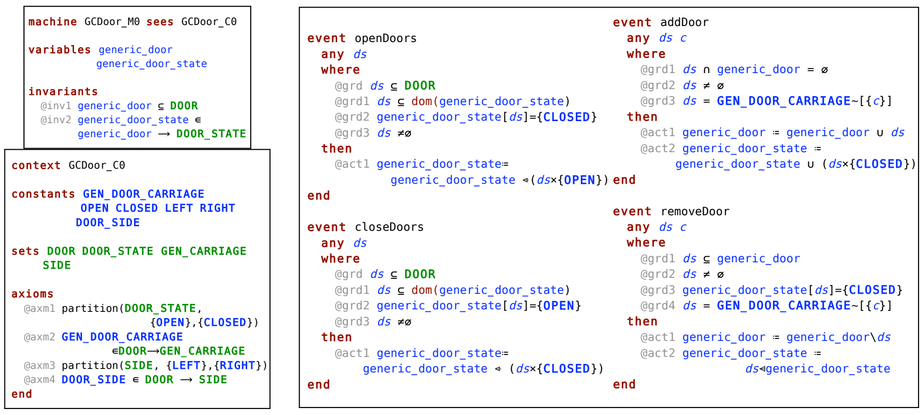
The generic model for the carriage doors is based in two refinements: *GCDoor_M0* and *GCDoor_M1*. In each refinement step, more requirements are introduced.

5.7 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 the opening or closing state.
3. When adding/removing doors, they are closed by default for security reasons.

The static part of the generic development is defined in context *GCDoor_C0* as seen in Fig. 7. It contains sets $DOOR$, $DOOR\_STATE$, $GEN\_CARRIAGE$ and $SIDE$, representing carriage doors, defining if a door is opened or closed, defining the carriages and defining the side of a door respectively. Constant $GEN\_DOOR\_CARRIAGE$ defines the relation between doors and carriages ($axm2$). Machine *GCDoor_M0* contains variables *generic_door* and *generic_door_state*. The invariants of this abstraction are very weak since we just add the type variables as can be seen in Fig. 7. Property 1 is expressed by events *addDoor* and



```
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
```

```
context GCDoor_C0

constants GEN_DOOR_CARRIAGE
          OPEN CLOSED LEFT RIGHT
         DOOR_SIDE

sets DOOR DOOR_STATE GEN_CARRIAGE
     SIDE

axioms
  @axm1 partition(DOOR_STATE,
                  {OPEN},{CLOSED})
  @axm2 GEN_DOOR_CARRIAGE
              ∈DOOR→GEN_CARRIAGE
  @axm3 partition(SIDE, {LEFT},{RIGHT})
  @axm4 DOOR_SIDE ∈ DOOR → SIDE
end
```

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

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
```

Fig. 7. Machine *GCDoors_M0*

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. Doors are removed in event *removeDoor*, if their state is $CLOSED$ confirming property 3. Next section describes the refinement of this machine.

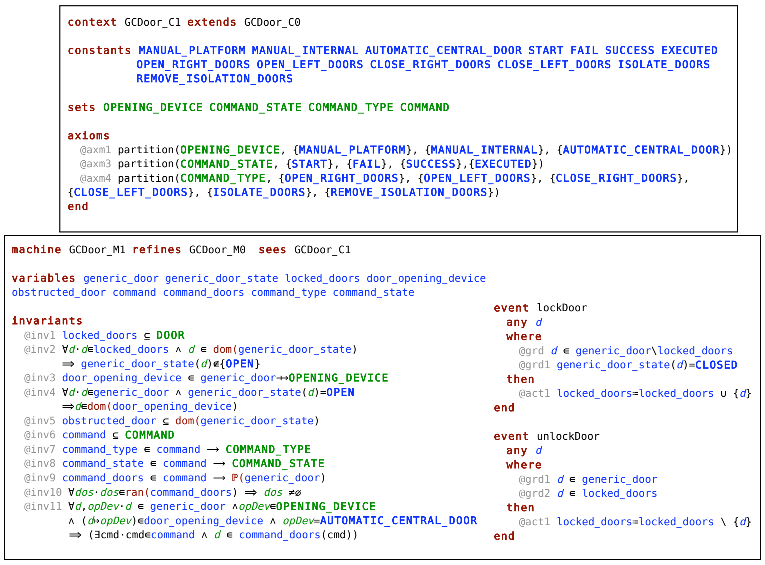
5.8 Second refinement of GCDoor: GCDoor_M1

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.

The context used in this refinement (*GCDoor_C1*) extends the existing one as seen in Fig. 8. Abstract events are refined to include the properties defined above. Some new invariants are added as seen in Fig. 8. Property 1 is defined by new



```
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})
  @axm3 partition(COMMAND_STATE, {START}, {FAIL}, {SUCCESS},{EXECUTED})
  @axm4 partition(COMMAND_TYPE, {OPEN_RIGHT_DOORS}, {OPEN_LEFT_DOORS}, {CLOSE_RIGHT_DOORS},
{CLOSE_LEFT_DOORS}, {ISOLATE_DOORS}, {REMOVE_ISOLATION_DOORS})
end
```

```
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))
```

```
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
```

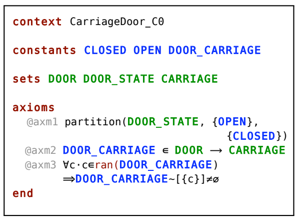
Fig. 8. Excerpt of machine *GCDoors_M1*

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 specific 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.

5.9 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 $GCDoor\_C0$ in Fig. 7 and $GCDoor\_C1$ in Fig. 8. The parameterisation context used by the instance results from the context seen by the abstract machine *CarriageDoors* as illustrated in Fig. 9. *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*.

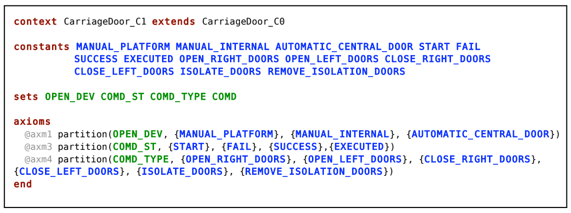


```
context CarriageDoor_C0

constants CLOSED OPEN DOOR_CARRIAGE

sets DOOR DOOR_STATE CARRIAGE

axioms
  @axm1 partition(DOOR_STATE, {OPEN},
                              {CLOSED})
  @axm2 DOOR_CARRIAGE ∈ DOOR → CARRIAGE
  @axm3 ∀c·c∈ran(DOOR_CARRIAGE)
        ⇒DOOR_CARRIAGE~[{c}]≠∅
end
```



```
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})
  @axm3 partition(COMD_ST, {START}, {FAIL}, {SUCCESS},{EXECUTED})
  @axm4 partition(COMD_TYPE, {OPEN_RIGHT_DOORS}, {OPEN_LEFT_DOORS}, {CLOSE_RIGHT_DOORS},
{CLOSE_LEFT_DOORS}, {ISOLATE_DOORS}, {REMOVE_ISOLATION_DOORS})
end
```

Fig. 9. Parameterisation context *CarriageDoors_C0* plus additional context *CarriageDoors_C1*

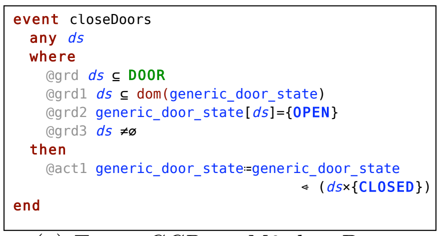
Following the steps suggested in Sect. 3, we create the instantiation refinement for emergency carriage doors as seen in Fig. 10. As expected, the generic sets and constants are replaced by the instance sets existing in contexts *CarriageDoors_C0* and *CarriageDoors_C1*. Moreover, generic variables are 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* contains some additional parameters and guards in some events resulting from the previous refinements. For instance, event *closeDoors* in *CarriageDoors* (Fig. 11(b)) contains an additional parameter *cds* compared to event *closeDoors* in *GCDoor_M0* (Fig. 11(a)). Some customisation is required in the generic event to ensure that the instantiation of *GCDoor_M0.closeDoors* refines *CarriageDoors.closeDoors* by adding an a parameter that match *cds* and respective guard $grd13$.

The customisation can be realised after the instantiation by adding the required elements to ensure a valid refinement. In our case, we would need to add

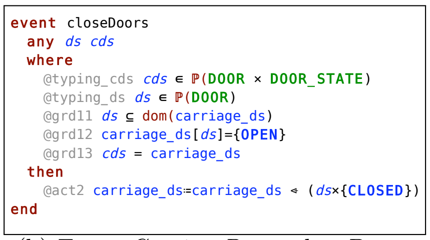
```
INSTANTIATED REFINEMENT IEmergencyDoor_M1
INSTANTIATES GCDoors_M1 VIA GCDoor_C0 GCDoor_C1
REFINES CarriageDoors      /* abstract machine */
SEES CarriageDoors_C0 CarriageDoors_C1      /* instance contexts */
REPLACE
    SETS GEN_CARRIAGE := CARRIAGE   DOOR := DOOR
         DOOR_STATE := DOOR_STATE   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
              ...
RENAME    /*rename variables, events and params*/
    VARIABLES generic_doors := carriage_doors   generic_door_state := carriage_ds
    EVENTS addDoor := allocateCarriageTrain   removeDoor := removeCarriageTrain
END
```

Fig. 10. Instantiated Refinement *IEmergencyDoor_M1*



```
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*

Fig. 11. Events *CarriageDoors.closeDoors* and

the additional parameter *cds* and guard $cds = carriage\_ds$. This is possible since the refinement verification is local to the event and not global to the machine. An instance machine *EmergencyDoor_M1* is similar to *GCDoor_M1* apart from the replacements and renaming applied in *IEmergencyDoor_M1*. That machine can be further refined by introducing the specific details related to emergency doors. The instantiation of the service doors follows the same steps.

Statistics: In Table 2, we describe the statistics of the development in terms of variables, events and proof obligations (and how many were automatically discharged) for each refinement step. This case study was carried out under the

	Variables	Events	ProofObligations/Auto
Train	7	12	60/59
Train_M1	9	14	71/54
Train_M2	13	19	144/80
Train_M3	12	19	63/26
Train_M4	14	19	113/84
Carriage_M1	5	10	29/22
GCDoor_M0	2	4	5/5
GCDoor_M1	9	15	79/78

Table 2. Statistics of the metro system case study

following conditions:

– Rodin v2.1
– Plug-ins: Model Decomposition plug-in v1.2.1, ProB v2.1.2
– The instantiation was done manually (tool to be developed).

Although we were interested mainly interested in safety properties, the model checker ProB [9] 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 runs and does what it should do.

6 Conclusions

We model a metro system case study, starting by proving its global properties through several refinement steps. Afterwards, the system is decomposed in two sub-components that can be further refined independently: *LeaderCarriage* and *Carriage*. Since we were 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. Our contribution is the definition of a methodology to develop large formal methods using refinement, decomposition and generic instantiation and respective application to a distributed system case study. We share our experience and suggest some guidelines on how to develop our case studies following our approach. Although we use Event-B, this 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. Sabatier [10] discusses the reuse of formal models as a detailed component specification or as a high level requirement and presents some real project examples. Lutz [11] describes the reuse of formal methods when analysing the requirements and designing the software between two spacecrafts formal models. Blazy *et al* [12] reuse Gang of Four (GoF) design pattern adapted to formal specifications for classical B. Several reuse mechanisms are suggested like instantiation, composition and extension. Proof obligations are also reused when the patterns are applied. Focusing on the instantiation, this is achieved by renaming sets (machine parameters), variables and operations. Unlike our work, this approach only defines patterns as single abstract machine whereas we define the parameterisation to a chain of refinements. Butler [13] 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 but with additional use of generic instantiation for the carriage doors.

In a combination of refinement and instantiation, the abstract machine and the abstract pattern do not necessarily match perfectly. In particular, some extra guards and parameters may exist in the abstract events resulting from previous refinements. It is still possible to reuse the generic model. We can customise the instance after the instantiation to ensure a valid refinement and proceed the development (part of the requirements for our future work of developing an instantiation tool). Although we did not use here, another interesting conclusion is that throughout an instantiation, only a subset of generic events can be used in opposition to use the entire set. If a refinement pattern, the subset of events are still refined since the event refinement only depends on the abstract and concrete events. Nevetheless this only applies for safety properties. If we are interested in liveness and enabledness properties, the exclusion of a generic event may result in deadlock. We intend to study these consequences in the future.

References

1. Standish, T.A.: An Essay on Software Reuse. IEEE Trans. Software Eng. **10**(5) (1984) 494–497
2. Métayer, C., Abrial, J.R., Voisin, L.: Event-B Language. Technical report, Deliverable 3.2, EU Project IST-511599 - RODIN (May 2005)
3. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition Tool for Event-B. Software: Practice and Experience **41**(2) (February 2011) 199–208
4. Silva, R., Butler, M.: Supporting Reuse of Event-B Developments through Generic Instantiation. In Breitman, K., Cavalcanti, A., eds.: Formal Methods and Software Engineering. Volume 5885 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Rio de Janeiro, Brazil (December 2009) 466–484
5. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundam. Inf. **77**(1-2) (2007) 1–28
6. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
7. Rodin: RODIN project Homepage. `http://rodin.cs.ncl.ac.uk` (September 2008) Online; accessed 27-July-2010.
8. Butler, M.: Synchronisation-Based Decomposition for Event-B. In: RODIN Deliverable D19 Intermediate report on methodology. (2006) 47–57
9. ProB: ProB. `http://www.stups.uni-duesseldorf.de/ProB/overview.php` (September 2008) Online; accessed 27-July-2010.
10. Sabatier, D.: Reusing Formal Models. In: IFIP Congress Topical Sessions. (2004) 613–620
11. Lutz, R.R.: Reuse of a Formal Model for Requirements Validation. In: In Fourth NASA Langley Formal Methods Workshop. NASA. (1997)
12. Blazy, S., Gervais, F., Laleau, R.: Reuse of Specification Patterns with the B Method. In Springer-Verlag SEP, ed.: ZB 2003: Formal Specification and Development in Z and B Lecture Notes in Computer Science. Volume 2651 of Lecture Notes in Computer Science., Turku, Finland (June 2003) 40–57
13. Butler, M.: A System-based Approach to the Formal Development of Embedded Controllers for a Railway. Design Automation for Embedded Systems **6** (2002) 355–366