Chapter 7

Event-B and Rodin

7.1. Event-B

7.1.1. *The Event-B definition*

Event-B [ABR 08] is a formal method for specifying, modeling, and reasoning about systems based on set theory and predicate logic. Event-B evolved from Classical B [ABR 96] and Action Systems [BAC 89]. On the one hand, Event-B is a simplification as well as an evolution of the B-Methods; on the other hand, Event-B is influenced by the action systems approach. It has the same structure as an action system, which describes the behavior of a reactive system in terms of the guarded actions that can take place during its execution.

Event-B is different from the B-Method in some aspects. The B-Method is organized in a way that is suitable for the development of non-concurrent programs, whereas Event-B is geared toward the development of systems including reactive and concurrent systems. Building a model in Event-B starts with a very abstract level, and continues in different abstraction levels by use of refinement, which will be explained in section 7.1.3. Event-B uses mathematical proof to verify consistency between refinement levels. Association of proof obligations in Event-B permits us to reason about it; see section 7.1.4. Rodin is a tool platform for modeling and proving in Event-B. It will be outlined in section 7.2. Section 7.4 describes the development of a metro system case study in Event-B.

Chapter written by Michael BUTLER, Asieh Salehi FATHABADI and Renato SILVA.

7.1.2. *Event-B structure and notation*

A model in Event-B [ABR 08] consists of *contexts* and *machines*. Contexts contain the static part (types and constants) of a model while machines contain the dynamic part (variables and events). Contexts provide axiomatic properties of an Event-B model, whereas machines provide behavioral properties of an Event-B model. Items of machines and contexts presented in this section are called modeling elements. There are various relationships between contexts and machines. A context can be “extended” by other contexts and “seen” by machines. A machine can be “refined” by other machines and refer to contexts as its static part. Refinement is described more in section 7.1.3. The relationship between machine and context is illustrated in Figure 7.1.

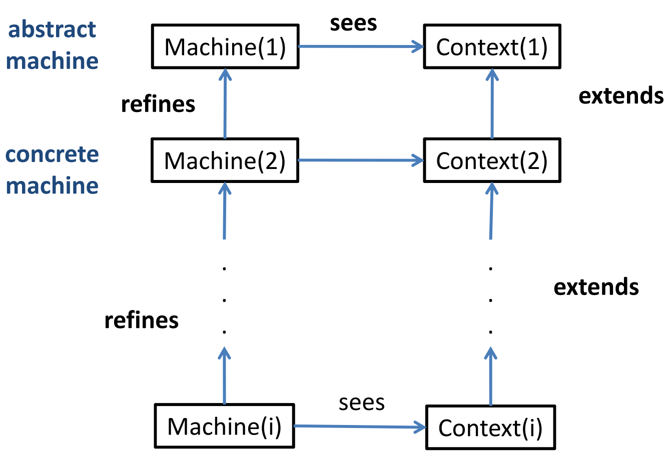


Figure 7.1. *Machine and context relationships*

From a given machine, $Machine1$ in this case, a new machine, $Machine2$, can be built as a refinement of $Machine1$. In this case, $Machine1$ is called an abstraction of $Machine2$, and $Machine2$ is said to be a concrete version of $Machine1$.

7.1.2.1. *Context structure*

Modeling elements of a context [ABR 08] are of four forms: sets, constants, axioms, theorems. This is illustrated in Figure 7.2. Axioms are predicates that describe the properties of sets and constants. Theorems are properties that should follow from the axioms. A context may extend one or more other contexts. And can also be seen by several machines in a direct or indirect way.

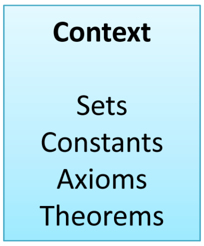


Figure 7.2. *Structure of a context*

7.1.2.2. *Machine Structure*

A machine [ABR 08] consists of variables, invariants, events, theorems, and variants, illustrated in Figure 7.3. Variables, v, define the states of a model. Invariants, $I(v)$, constrain variables, and are supposed to be hold whenever variables are changed by an event. New events can be defined in a concrete machine. These will be described more in section 7.1.3. To prove that they do not take control forever, a new event must decrease a natural number expression called a variant.

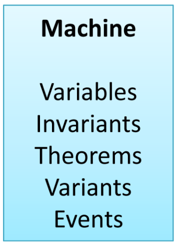


Figure 7.3. *Structure of a machine*

7.1.2.3. *Events*

In Event-B, state of a model is changed by means of event execution. Each event is composed of a name, a set of guards $G(t, v)$, and some actions $S(t, v)$, where t are parameters of the event and v is state of the system, which is defined by variables. All events are atomic and can be executed only when their guards hold. When the guards of several events hold at the same time, then only one of those events is chosen non-deterministically to be executed. An event can appear in three forms presented in Table 7.1. In the simplest terms, an event contains only some actions, in the second form it can be composed of guards and actions without parameters, and finally in the third form, an event has guards, actions, and some parameters.

The action of an event can have one of several forms of assignment, illustrated in Table 7.2. Here x is a variable, $E(t, v)$ is an expression, and $P(t, v, x)$ is a predicate. The first assignment form is deterministic. In the second row, the assignment is non-deterministic (for instance, assign a value within a non-empty set). The third row assigns a value to x according to the predicate defined, and it is also considered non-deterministic.

Three Possible Forms of an Event
E = begin S(v) end
E = when G(v) then S(v) end
E = any t when G(t,v) then S(t,v) end

Table 7.1. *Event forms*

Type	**Generalized Substitution**
Deterministic	x := E(t, v)
Non-deterministic	$x :\in E(t, v)$
Non-deterministic	$x :\mid P(t, v, x')$

Table 7.2. *Action forms*

7.1.3. *Refinement in Event-B*

In an Event-B development, rather than having a single large model, we are encouraged to construct the system in a series of successive layers, starting with an abstract representation of the system. The abstract model should provide a simple view of the system, focusing on the main purpose and key features of the system. The details of how the purpose is achieved are ignored in the abstraction. Details of functionality of the system are added gradually to the abstract model in a stepwise manner. This process is called refinement.

In Event-B modeling, we use proof to verify the consistency of a refinement. The definitions of some refinement proof obligations are described in section 7.1.4.

Refining an Event-B model can consist of context extension and machine refinement. Considering context extension, it is possible to add new sets, constants, and properties while retaining the old ones.

Refinement in Event-B has different views or classification. From Event-B notation point of view, refinement of a machine consists of:

1. Refining existing events:

(a) Adding new parameters, guards and actions to the existing abstract event: in this case the resulting concrete event is labeled as $extended$. In an $extended$ event, the existing parameters, guards and actions cannot be modified.

(b) Modifying parameters, guards, and actions of the existing abstract event: in this case, the resulting concrete event is labeled as $non - extended$ ($refine$). Adding new parameters, guards, and actions are allowed, too.

In both types, the guards of the concrete event must be proved to be stronger than its abstraction (guard strengthening).

2. Adding new events

The new event refines a dummy event in the abstraction, which does nothing (*skip*).

The new event does not diverge. It means that it should not take control forever. The new event can be labeled as:

- Convergent: Each convergent event requires a variant to ensure non-divergence.
- Anticipated: Events that will be introduced in a future refinement but are declared in anticipation.
- Ordinary: None of the others and the most commonly used.

3. Add new variables and invariants:

Introducing new variables usually results in (2) or (1.a) types of refinement. Sometimes, abstract variables can be replaced by new concrete variables. In this case, the refinement can result in (1.b). Variable replacement is called data refinement.

A gluing invariant connects the abstract variables to the concrete variables. In other words, it glues the state of the concrete model to that of its abstraction. The invariant of the concrete model including gluing invariants should be preserved for every event.

Each abstract event should be refined by at least one concrete event. One abstract event can be refined by more than one concrete event. This is called event splitting. Also, one concrete event can refine more than one abstract event. This is called event merging.

Refinement is the process of enriching or modifying the abstract model to introduce new functionality or add details of the current functionality. From another view, there are two forms of refinement:

– Vertical Refinement (Structural Refinement): In this form, design details of current functionalities are added. This form of refinement may involve data refinement (3) and modifying abstract events (1.b). In the refinement level, the modified events are labeled as non-extended events.

– Horizontal Refinement (Superposition Refinement or Feature Augmentation): New functionalities of the system, which are not addressed in the abstract level, are introduced. Usually, it can be achieved by introducing new events (2), new variables

(3) or extending abstract events (1.a). In the refinement level, these concrete events are labeled as extended events.

7.1.4. *Proof obligations*

There are different proof obligations, which are generated by the Event-B tool, Rodin, during the development of a system using Event-B [ABR 08]. Here, we describe some of those that are most important. Considering Figure 7.4, machine $M2$ refines machine $M1$. Both of them see context Ctx. $M2$ contains two events, $evt3$ as a new event and $evt2$ as a refining event. It also contains some gluing invariants.

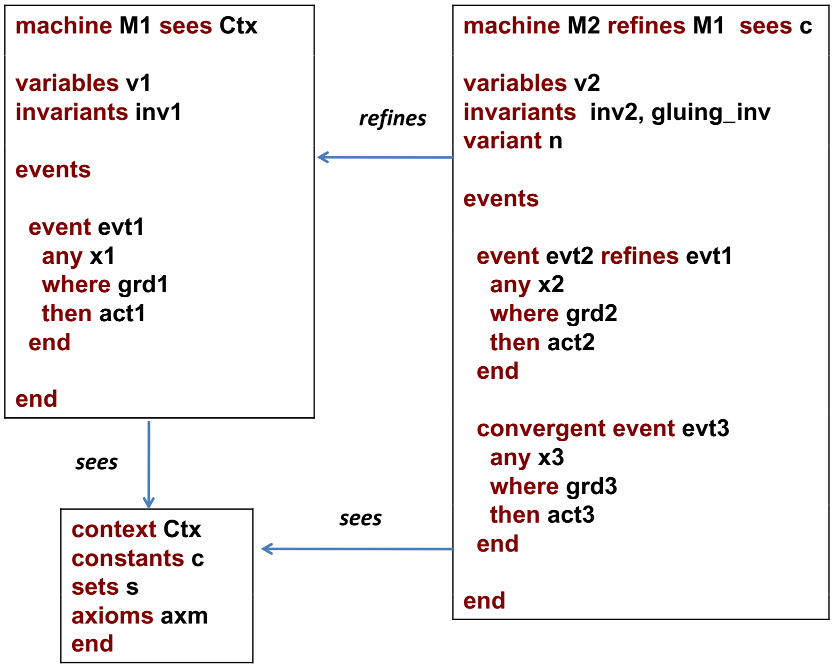


Figure 7.4. *An Event-B model (context Ctx, abstract machine $M1$, concrete machine $M2$)*

Table 7.3 contains a list of important proof obligation in Event-B modeling.

Here is an explanation for each of these proof obligations:

– **Well-definedness (WD)**: Ensure that an axiom, theorem, invariant, guard, action, variant is well-defined. When using the cardinality of a set, $card(S)$, it should be proved that the set, S, is finite.

– **Invariant Preservation (INT)**: Ensure that every invariant is preserved by each event. For instance, in Figure 7.4, one of the generated proof obligation is $evt1$/$inv1$/INV, ensuring that $inv1$ is preserved by event $evt1$ in machine $M1$.

– **Feasibility (FIS)**: Ensure that each non-deterministic action is feasible. In Figure 7.4, for event $evt1$ in machine $M1$, this proof obligation is given: $evt1$ / $act1$ / FIS; this means there should exist values for variable $v1$ such that the assignment $act1$ is feasible.

– **Guard Strengthening (GRD)**: Ensure that each abstract guard is no stronger than the concrete ones in the refining event. As a result, when a concrete event is enabled, the corresponding abstract one is also enabled. For instance, for the model in Figure 7.4, $evt2$ / $grd1$ / GRD ensure that the abstract guard $grd1$ is weaker than the guards of the concrete event $evt2$.

– **Simulation (SIM)**: Ensure that each action in a concrete event simulates the corresponding abstract action. When a concrete event executes, the corresponding abstract event is not contradicted. In Figure 7.4 the simulation proof is $evt2$ / $act1$ / SIM.

– **Numeric Variant (NAT)**: Ensures that under the guards of each convergent event a proposed numeric variant is indeed a natural number. $evt3$ / NAT is the proof obligation generated for the model of Figure 7.4.

– **Decreasing of Variant (VAR)**: Ensures that each convergent event decreases the proposed numeric variant. As a consequence the new event does not take control forever. $evt3$ / VAR in Figure 7.4 ensures that event $evt3$ does not take control forever.

Well-definedness	x / WD	x is the name of axiom, theorem, invariant, guard, action, variant
Invariant Preservation	evt / inv / INV	evt is the event name, inv is the invariant name
Feasibility of a non-deterministic event action	evt / act / FIS	evt is the event name, act is the action name
Guard Strengthening	evt / grd / GRD	evt is the concrete event name, grd is the abstract guard name
Action Simulation	evt / act / SIM	evt is the concrete event name, act is the abstract action name
Natural number for a numeric Variant	evt / NAT	evt is the new event name
Decreasing of Variant	evt / VAR	evt is the new event name

Table 7.3. *Proof obligations in Event-B*

7.1.5. *A comparison between Event-B and other formal methods*

Classical B, Z, and VDM have a one-to-one operation refinement, meaning that one abstract operation is refined by only one concrete operation. There is no facility for introducing new events in refinements in these formal methods. Event-B is more flexible as it bases its refinement on action systems. Also, event merging and event splitting are provided in Event-B refinement. Although Event-B is an extension of Classical B, there are some differences between them:

– The model structure is different. In Event-B, the context as the static part of the system and the machine as the dynamic part of the system are explicitly separated. In the B-Method, a machine contains both parts.

– In the B-Method, operations are called by other operations while in Event-B, the enabled events are continually executed in a non-deterministic manner. Since in Event-B, we are modeling reactive systems, the events are not called and the model controls its behavior by non-deterministically choosing the enabled events.

– A B-Method operation contains pre-conditions, which express formally what is to be proved when the operation is invoked. The caller of an operation is responsible for ensuring that pre-conditions of the called operation are satisfied before calling it. The called operation can assume that its pre-conditions are satisfied, and it does not need to check its pre-conditions. In contrast, an Event-B event contains guards. An event can be executed only when its guards hold. In Event-B, enabled events are non-deterministically chosen to execute.

– Refinement is more general in Event-B. Introducing new events is an important ability in Event-B refinement.

7.2. Rodin as an Event-B tool

Rodin [ABR 10, EVE] is an open source software tool for formal modeling and proving in Event-B. Rodin has an open platform and is an extensible and adaptable modeling tool. The ProB animator [WIK 02, LEU 08], UML-B [WIK 03, SNO 06], B2LaTeX [WIK 01] and model decomposition [SIL 11] are good examples of plug-in developments; ProB is a model checker, which checks the consistency of B machines; UML-B maps a graphical formal modeling notation to the Event-B language; B2LaTex is used for translating Event-B models into LaTeX documents; and model decomposition which decomposition of a model into sub-models. Decomposition will be explained in section 7.3.

Like programming tools, Rodin carries out many tasks automatically, and provides fast feedback in the case of changes in a model text. While a programming tool provides feedback to the programmer by compiling and executing a program, Rodin provides feedback to modellers by generating proof obligations and verifying these using automated provers.

Rodin is an integration between modeling and proving. As described in previous sections, proving is an essential part of modeling. The proof obligations define what is to be proved for an Event-B model. Discharging all proof obligations of a model shows that all model properties are consistent. Sometimes, a model can be changed using proof errors. When a proof obligation cannot be charged, it shows that there is an inconsistency in the model. This leads us to learn more about the system to change the model in an inconsistent way. Therefore, during modeling, we can learn about the system and eliminate misunderstandings. We can also learn new requirements by proving the failed proof obligations.

7.3. Event-B model decomposition

7.3.1. *Overview*

Model decomposition predated Event-B and is found in action systems [BAC 89]. In developing a model in Event-B, one of the key features is introducing new events and new state variables during refinement. As a consequence, it usually ends up with many events and many variables in the last refinement level. Dealing with a large number of events and variables can be complex; particularly when we need to refine just a few variables and events and so other variables and events play no role in the refinement.

Model decomposition in Event-B [SIL 11] is intended to decrease the complexity and increase the modularity of a large Event-B model, especially after several layers of refinements. The idea of model decomposition is cutting a huge model into smaller pieces called sub-models, which we can more easily deal with than the first model, and each of them can be refined separately.

Distribution of proof obligations into several sub-models is one of the major results of model decomposition, which is expected to be easier to discharge. The further refinements of independent sub-models in parallel is a benefit of model decomposition. Moreover, the possibility of team development after model decomposition seems useful in developing a big system.

An overview of the model decomposition in Event-B is illustrated in Figure 7.5. As presented, the model becomes bigger during refinement layers and with decomposition, it is split into smaller sub-models. Then, each sub-model can be refined independently.

7.3.2. *Decomposition styles*

There are two ways of decomposing an Event-B model, *shared variable* and *shared event* [SIL 11]. The shared event approach seems particularly suitable for message-passing distributed programs, whereas the shared variable approach seems more suitable for concurrent programs [BUT 97]. In shared event model decomposition, variables are partitioned among the sub-models, whereas in the shared variable approach, events are partitioned among the sub-models. Details are explained in the next section. A model decomposition plug-in [SIL 11] in the Rodin platform provides tool support for both styles of model decomposition.

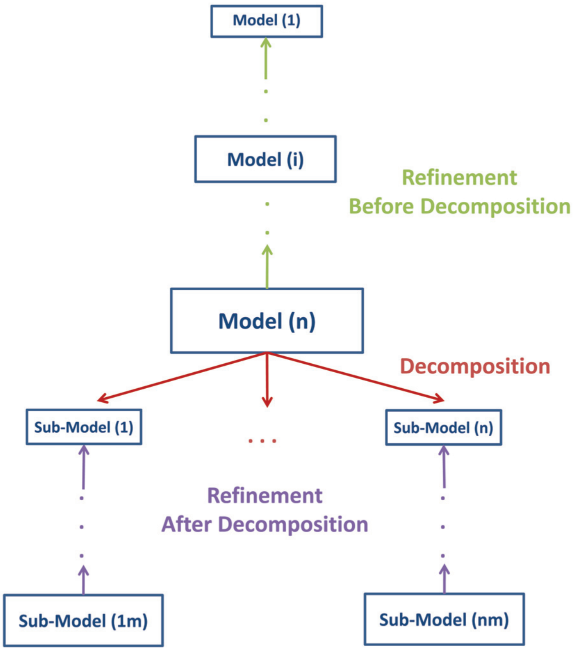


Figure 7.5. *Model decomposition in Event-B*

7.3.2.1. *Shared variable style*

Shared variable decomposition, illustrated in Figure 7.6, is proposed by Abrial and Hallerstede [ABR 07]. Machine *M* is decomposed into machine *M1* and *M2*. The solid lines show relationships between events and variables in each machine.

The shared variable decomposition does not permit event sharing and a variable can be split into different sub-models. This variable is called a shared variable. First, the events of *M* are partitioned among *M1* and *M2*. Then, the variables of *M* are distributed according to the event partition. Variables *v1* and *v3* are private variables since they are accessed by events of only one sub-model, *e1* in *M1* and *e4* in *M2*, respectively. Variable *v2* is a shared variable, which is accessed by event *e2* in *M1* and *e3* in *M2*. External event of *e2_ext* is built in *M2*, since *e2* modifies the shared variable *v2* in *M1*. The invariant distribution is done according to variable distribution. An invariant belongs to a sub-model if all variables used in that invariant belong to that sub-model.

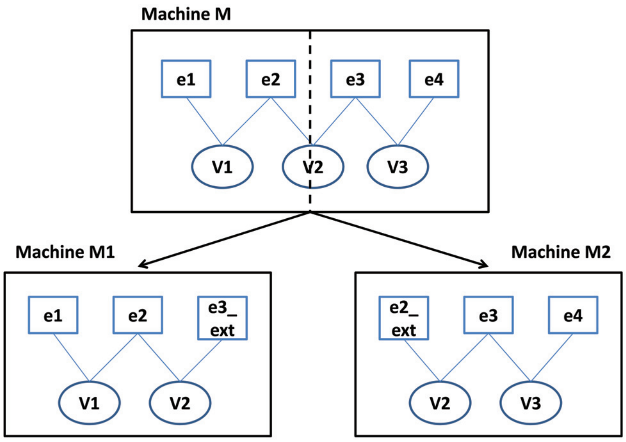


Figure 7.6. *Shared variable decomposition*

7.3.2.2. *Shared event style*

Figure 7.7 illustrates shared event decomposition proposed by Butler [BUT 09]. Variables of the machine *M* are partitioned among the sub-models, *M1* and *M2*. After the variable partition, it is necessary to split the events according to the variable partition. Events using variables allocated to different sub-models, *e2* using *v1* from *M1* and *v2* from *M2*, are called shared events and must be split. Part of the shared event, which is corresponding to each variable, *e2_1* and *e2_2*, is used to build sub-model events. Invariant distribution is similar to shared variable decomposition.

7.4. Case study: metro system

This section describes how the modeling, refinement, and decomposition techniques presented in the previous sections can be applied in practice. We aim to develop a system that becomes more complex with each refinement step, preserves its properties (requirements) and re-uses existing developments and proofs as much as possible. A safety-critical metro system case study is developed. This version is a simplified version of a real system but tackles points where the techniques outlined in the previous sections become relevant: stepwise incrementation of the complexity of the system being modeled, sub-components communication, stepwise addition of requirements at each refinement level, refinement of decomposed sub-components. Although this system is initially modeled as a single component, it can be seen as a distributed system where the initial model is split into smaller sub-components that communicate via shared events. The split is achieved through a shared event decomposition and the sub-components are further refined independently. After several refinements, we reach a refinement that fits an existing generic development of metro doors. Using that development as a pattern, two models are instantiated accordingly.

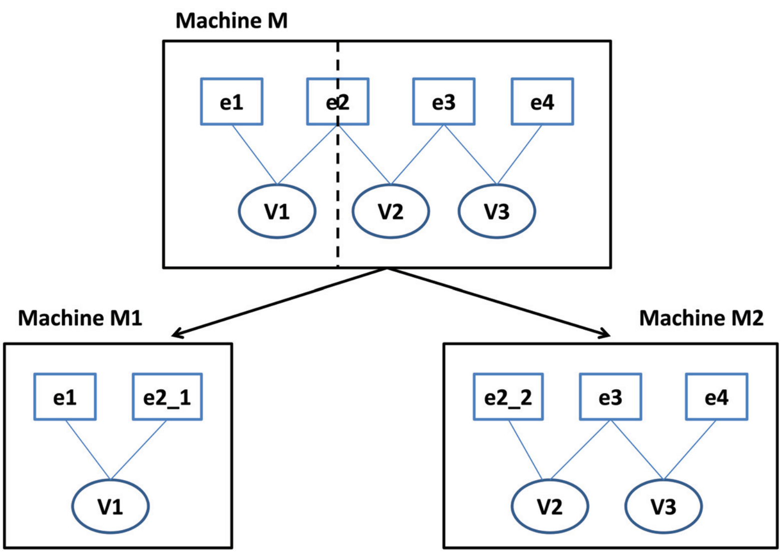


Figure 7.7. *Shared event decomposition*

7.4.1. *Overview of the safety-critical metro system*

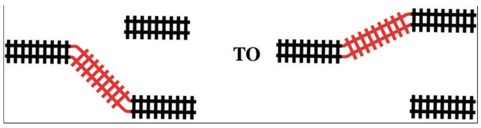
The safety-critical metro system case study describes a formal approach for the development of embedded controllers for a metro system[1]. Butler [BUT 02] makes a description of embedded controllers for a railway using classical B. The railway system is based on the French train system. Our starting 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 characterized 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

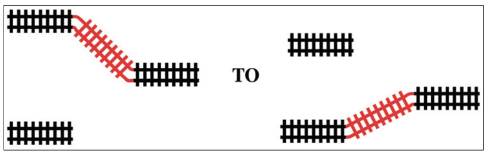
1 A version of this model is available online at http://eprints.ecs.soton.ac.uk/23135/.

enters or leaves a section, a permission notification must be received. In case of a hazardous 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 Figure 7.8), and sending signaling messages to the trains.

Figure 7.9 shows a schematic representation of the metro system decomposed into three sub-components. Initially, the metro system is modeled as a whole. Global properties are introduced and proved to be preserved throughout refinement steps. The abstract model is refined in three levels (*MetroSystem_M0* to *MetroSystem_M3*) before we apply the first decomposition. We follow a general top-down guideline to apply decomposition:



(a) Divergent Switch



(b) Convergent Switch

Figure 7.8. *Different types of switches: divergent and convergent*

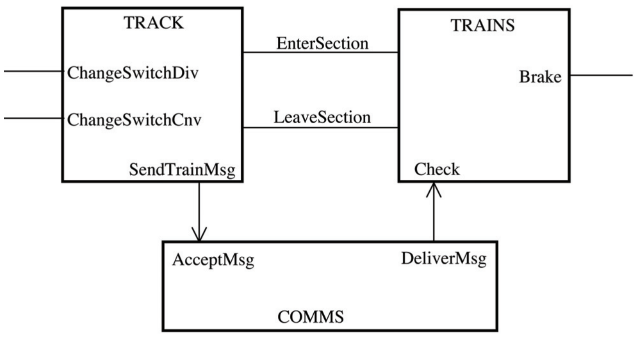


Figure 7.9. *Components of metro system*

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.

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.

An overview of the development can be seen in Figure 7.10. 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 to introduce doors requirements. These requirements are extracted from real requirements for metro carriage doors.

7.4.2. *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 might collide).

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

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

– There are no loops in the route sections: sections cannot be connected to themselves 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 [ABR 08] and used to model a tree-structured file system in Event-B [DAM 08]: 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 easier to prove. Context $TransitiveClosureCtx$ containing the transitive closure property can be seen in Figure 7.11.

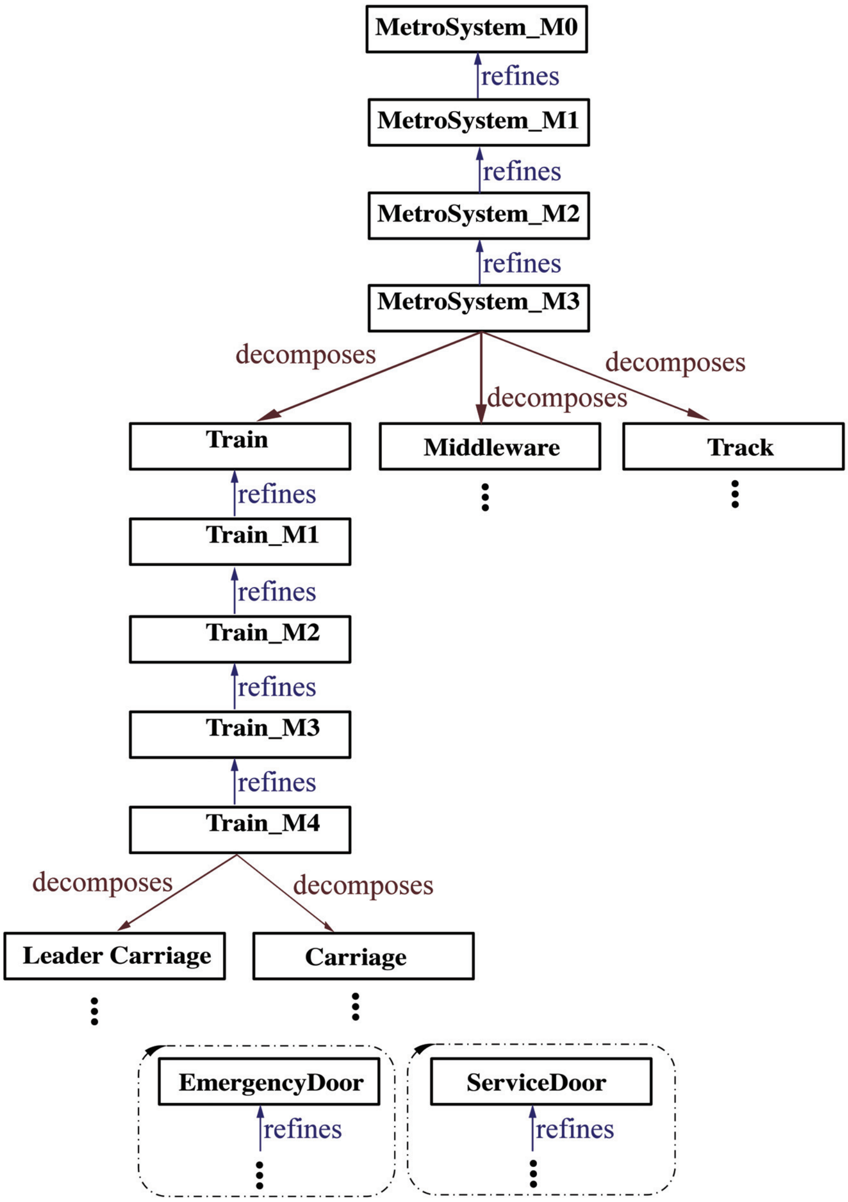


Figure 7.10. *Overall view of the safety-critical metro system development*

```
context TransitiveClosureCtx

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

sets CDV // Track Sections

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

Figure 7.11. *Context TransitiveClosureCtx*

Set CDV represents all the track sections in our model. Constant tcl, which is a transitive closure, is defined as a total function mapped from $CDV \leftrightarrow CDV$ to $CDV \leftrightarrow CDV$. Giving $r \in CDV \leftrightarrow CDV$, the transitive closure of r is the least x satisfying $x = r \cup r; x$ [DAM 08]. Difficult transitive closure proofs in machines are avoided by using Theorems, such as theorem $thm3$ shown in Figure 7.11: for $s \subseteq CDV$ and t as a partial function $CDV \nrightarrow CDV$, $s \subseteq t^{-1}[s]$ means that s contains a loop in the t relationship. Hence, this states that the only such set that can exist is the empty set and thus the t structure cannot have loops. This theorem has been proved using the interactive prover of Rodin. The strategy to prove this theorem is to use proof by contradiction [DAM 08].

We define the environment of the case study (static part) with context *MetroSystem_C0* that extends *TransitiveClosureCtx* as seen in Figure 7.12. 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 $thm1$ state that net preserves transitive closure.

– Switches (*aiguillages* in French) are sections ($axm3$) that cannot be connected to each other ($axm6$). 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 ($axm9$).

```
context MetroSystem_C0 extends TransitiveClosureCtx

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

sets TRAIN

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

Figure 7.12. *Context MetroSystem_C0*

In addition to the global property defined by invariant $inv13$ in Figure 7.13a, the following system properties are added to the Event-B model:

1. The trains (variable $trns$) circulate in tracks. The current route based on current positions of switches is defined by $next$: a partial injection $CDV \rightarrowtail\!\!\!\!\!\!\!+ \; CDV$. $next$ is a subset of net ($inv1$) 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 modeled in the system circulate by entering and leaving sections (events $enterCDV$ and $leaveCDV$ in Figure 7.13b), ensuring that the next section is not occupied ($grd9$ in $enterCDV$) and updating all the sections occupied by the train ($act1$ and $act2$ in both events). At this abstract level, event $modifyTrain$ modifies a train defining the set of occupied sections for a train t. A train changes speed, brakes, or stops braking in events $changeSpeed$, $brake$, and $stopBraking$. When event $brake$ occurs, train t is added to a set of braking trains (variable $braking$). Variable $next$ represents the current connectivity of the trail based on the positions of switches. The current connectivity can be updated by changing

```
machine MetroSystem_M0 sees MetroSystem_C0

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

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

(a) Variables, invariants in *MetroSystem_M0*

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

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

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

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

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

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

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

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

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

(b) Events of *MetroSystem_M0*

Figure 7.13. *Variables, invariant and events of MetroSystem_M0*

convergent/divergent switches in events $switchChangeDiv$ and $switchChangeCnv$ as seen in Figure 7.13b.

7.4.3. *First refinement: MetroSystem_M1*

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

1. Messages are exchanged between trains and tracks. If a train intends to move to an occupied section, the 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 entering and leaving the 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 behavior (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 a built-in way to detect the required space to break. For instance, in Communication-Based Train Control (CBTC [TSD 12, FAL 09]) systems, that is called the *stopping distance downstream*.

The messages are represented by variables $tmsgs$ that store the messages (buffer) sent from the tracks and $permits$ that receive the message in the train, expressing property 1. At this level, the messages are just Boolean values assessing whether a train can move to the 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 modeled 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 Figure 7.14b). 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 The event $stop$ $Braking$: a train can only stop braking if the emergency button is not enabled.

```
machine MetroSystem_M1 refines MetroSystem_M0  sees MetroSystem_C0

variables next trns occp occpA occpZ
          braking speed
          tmsgs permit emergency_button

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

(a) Variables and invariants in *MetroSystem_M1*

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

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

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

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

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

(b) Some events of *MetroSystem_M1*

Figure 7.14. *Excerpt of MetroSystem_M1*

7.4.4. *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 as follows:

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 on 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 on a platform if one of the occupied sections corresponds to a platform. Doors are introduced as illustrated in Figure 7.15a 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 Figure 7.15b: all trains have allocated a subset of doors ($inv2$). Several invariants are introduced to preserve the required properties: property 1 is defined by invariants $inv4$ and $inv5$; property 2 is defined by invariant $inv7$; property

```
context MetroSystem_C1 extends MetroSystem_C0

constants OPEN CLOSED PLATFORM

sets DOOR_STATE DOOR

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

(a) Context *MetroSystem_C1*

```
machine MetroSystem_M2 refines MetroSystem_M1  sees MetroSystem_C1

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

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

(b) Variables, invariants in *MetroSystem_M2*

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

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

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

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

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

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

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

(c) Some events of *MetroSystem_M2*

Figure 7.15. *Excerpt of MetroSystem_M2*

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 Figure 7.14b) are strengthened by $grd4$ ensuring that whilst the train is moving, the train doors are closed. Also, events that model entering and leaving sections are affected, with the introduction of a similar guard ($grd11$ in $leaveCDV$). Adding/removing train doors is modeled in events $addDoorTrain$ and $removeDoorTrain$, respectively: to add/remove a door, the respective train must be stopped. If the train is stopped and either one of the occupied sections corresponds to a platform or the emergency button is activated (guard $grd3$), doors can be opened as seen in event $openDoor$. For safety reasons, event $toggleEmergencyButton$ is strengthened by guard $grd3$ to activate the emergency button whenever doors are open and the train is not on a platform.

7.4.5. *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 section 7.3.2.2. Variables are distributed according to Figure 7.16. To avoid constraints during the decomposition process, predicates and assignments containing variables that belong to different sub-components are re-arranged in this refinement step.

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

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

to:

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

(Figure 7.17).

Both predicates represent the same property since $trns$ corresponds to the domain of variable $occpZ$ (see $inv7$ in Figure 7.13a). In Figure 7.17, the original guard $grd3$

in $toggleEmergencyButton$ is rewritten to separate variables $occp$ and $door$. In this case, an additional parameter $occpTrns$ representing the variable $occp$ is added ($grd4$). This additional parameter will represent the value passing between the resulting decomposed events: parameter $occpTrns$ is written the value of $occp$ and afterward it is read in guard $grd3$. Similarly, guard $grd4$ in event $openDoor$ must not include variables $occp$ and $emergency\_button$ and consequently parameter $occpTrns$ is added.

Sub-components $Train$, $Track$, and $Middleware$ are described in the following sections.

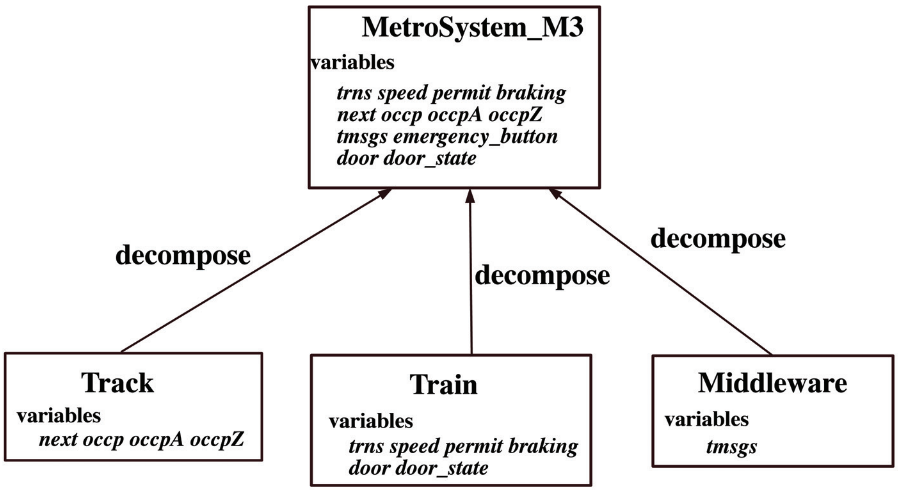


Figure 7.16. *MetroSystem_M3 (shared event) decomposed into Track, Train and Middleware*

7.4.6. *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 Figure 7.18. The original events $toggleEmergency$ $Button$ and $openDoor$ require $occp$ in their guards. Consequently, a part of these original events are included in this sub-component.

Note that the invariants defining the variables may change: in *MetroSystem_M1* variable $occp$ is defined as $occp \in CDV \leftrightarrow trns$ (*inv4* in Figure 7.13a); in *Track* is $occp \in CDV \leftrightarrow TRAIN$ (which is the same as theorem $typing\_occp$: $occp \in \mathbb{P}(CDV \times TRAIN)$ in Figure 7.18). This is a consequence of the variable partition

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

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

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

Figure 7.17. *Preparation step before decomposition of MetroSystem_M3*

```
machine Track sees MetroSystem_C1

variables next occp occpA occpZ

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

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

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

   event openDoor
     any t occpTrns ds
     where
       @typing_t t ∈ TRAIN
       @typing_occpTrns occpTrns ∈ ℙ(CDV)
       @typing_ds ds ∈ ℙ(DOOR)
       @grd3 occpTrns = occp∼[{t}]
       @grd7 ds≠∅
   end
   event leaveCDV
     any t1 c1 c2
     where
       @typing_t1 t1 ∈ TRAIN
       @grd2 c1 ∈ CDV
       @grd3 c2 ∈ CDV
       @grd5 c1∈dom(next)
       @grd6 c1=occpA(t1)
       @grd7 c2=next(c1)
       @grd8 occpA(t1)≠occpZ(t1)
       @grd9 c2 ∈ (occp\{c1↦t1})∼[{t1}]
       @grd10 ∀tt·tt∈dom(occpZ)
              ∧ card(((occp \ {c1 ↦ t1}))∼[{tt}])>1
              ⇒ (occpA⩤{t1 ↦ c2})(tt)≠occpZ(tt)
     then
       @act1 occpA(t1)≔c2
       @act2 occp ≔ (occp\{c1↦t1})
   end
  event toggleEmergencyButton
   any t value occpTrns
   where
     @typing_t t ∈ TRAIN
     @typing_occpTrns occpTrns ∈ ℙ(CDV)
     @grd2 value ∈ BOOL
     @grd4 occpTrns = occp∼[{t}]
  end
```

Figure 7.18. *Excerpt of Track*

since *trns* is not part of *Track* and therefore, the *occp* relation is updated with *trns*'s type: $TRAIN$ (see $inv3$ in Figure 7.13a). Variables *occpA* and *occpZ* are subject to the same procedure where the original invariant is a total function $trns \rightarrow CDV$ and in the sub-component, both become $\mathbb{P}(TRAIN \times CDV)$. The sub-components invariants are derived from the different initial abstract models (see their labels in Figure 7.18). Invariants that only restrain the sub-component variables are automatically included although additional ones can be added manually.

7.4.7. *Machine Train*

Machine *Train* models the trains in the metro system. Trains entering/leaving a section, modeled by events $enterCDV$ and $leaveCDV$, are part of this sub-component, (see Figure 7.19b). The interaction with sub-component *Track* occurs through parameters $t1$, $c1$ and $c2$ (see events $Track.leaveCDV$ in Figure 7.18). 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 $recvTrain Msg$ receives messages sent to the 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 Figure 7.19.

7.4.8. *Machine Middleware*

Finally, the communication layer is modeled by *Middleware* as seen in Figure 7.20. *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 Figure 7.10.

7.4.9. *Refinement of Train: Train_M1*

In Train_M1, carriages are introduced as parts of a train. Each carriage has an individual alarm, which when activated, triggers the train alarm (enables the emergency button of the train). Each train has a limited number of carriages. Each carriage has

```
machine Train sees MetroSystem_C1

variables trns speed permit braking emergency_button door_state door

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

(a) Variables and invariants in *Train*

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

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

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

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

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

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

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

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

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

(b) Some events of *Train*

Figure 7.19. *Excerpt of Train*

a set of doors and the sum of carriage doors corresponds to the doors of a train. The properties to be preserved are as follows:

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.

```
machine Middleware sees MetroSystem_C1

variables tmsgs

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

events
  event INITIALISATION
    then
      @act1 tmsgs ≔ ∅
  end

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

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

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

Figure 7.20. *Machine Middleware*

The definition of these requirements need the introduction of some static elements, such as a carrier set *CARRIAGE*, constants *MAX_NUMBER_CARRIAGE*, and *DOOR_CARRIAGE* (function between *DOOR* and *CARRIAGE*). The latter is defined as a constant because the number of doors in a carriage does not change. Context *Train_C2* is depicted in Figure 7.21a. 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 Figure 7.21b. 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 Figure 7.21b. Events *activateEmergencyCarriageButton* and *deactivateEmergency TrainButton* 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*

```
context Train_C1 extends MetroSystem_C1

constants MAX_NUMBER_CARRIAGE
          DOOR_CARRIAGE

sets CARRIAGE

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

(a) Context *Train_C1*

```
machine Train_M1 refines Train  sees Train_C1

variables trns speed permit braking door_state door emergency_button
          train_carriage carriage_alarm

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

(b) Variables and Invariants of *Train_M1*

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

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

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

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

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

(c) Some events of *Train_M1*

Figure 7.21. *Excerpt of machine Train_M1*

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: $d = DOOR\_CARRIAGE^{-1}[\{c\}]$.

7.4.10. *Further development*

Details of some remaining refinement and decomposition steps may be found in [SIL 12]. *Carriage* is refined and decomposed until it fits in a generic model *GCDoor* corresponding to a *Generic Carriage Door* development as seen in Figure 7.22. A generic model *GCDoor* is instantiated into two instances: *EmergencyDoors* and *ServiceDoors* benefiting from the refinements in the pattern.

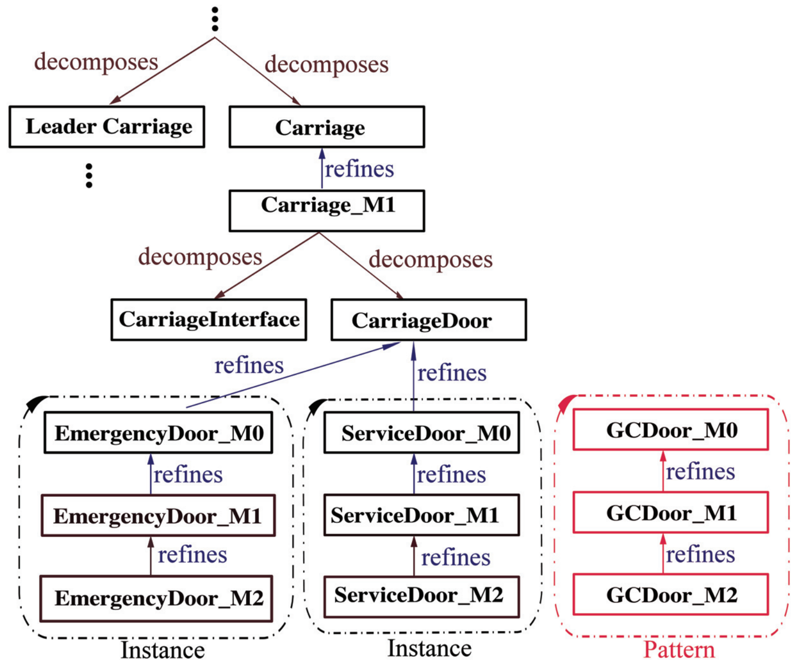


Figure 7.22. *Carriage Refinement Diagram and Door Instantiation*

7.4.11. *Conclusion*

We modeled a metro system case study, starting by proving its global properties through several refinement steps. Afterward, due to an architectural decision and to alleviate the problem of modeling and handling a large system in one single machine, the system is decomposed in three sub-components. We further refine one of the resulting sub-components (*Train*), introducing several details in four refinements levels.

The derivation of the distributed rail system illustrates a formal design approach for embedded controllers that takes into account models of the physical behavior as well as required control behavior. Traditionally, formal methods are used to verify correctness of computer systems with respect to a specification. Here, we are using formal methods to model and reason about a system as a whole, both the physical system and the required control behavior.

Specifying the system-level model does require skill in deciding on the appropriate abstractions, what aspects of behavior need to be modeled and what aspects can be left out of the model. The benefit of the system-level model is that it is easier to understand and reason about the behavior of the system as a whole.

The complexity of the entities and the relationships between them is handled through the use of refinement, which allows complexity to be introduced and reasoned about in steps. We made use of refinement for two main purposes, to introduce communications mechanisms leading to system partition and to replace abstract structures by more concrete realizations (such as replacing $next$ by pos).

Although we are mainly interested in safety properties, the model checker ProB [WIK 02] 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 is free from deadlocks.

7.5. Acknowledgments

The authors of this chapter would like to acknowledge funding from the FP7 DEPLOY Project (www.deploy-project.eu).

7.6. Bibliography

[ABR 96] ABRIAL J.-R., *The B-book: Assigning Programs to Meanings*, Cambridge University Press, New York, USA, 1996.

[ABR 07] ABRIAL J.-R., HALLERSTEDE S., "Refinement, decomposition, and instantiation of discrete models: application to Event-B", *Fundamenta Informaticae*, vol. 77, no. 1-2, p. 1-28, 2007.

[ABR 08] ABRIAL J.-R., *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, 2008.

[ABR 10] ABRIAL J.-R., BUTLER M., HALLERSTEDE S., HOANG T.S., MEHTA F., VOISIN L., “Rodin: an open toolset for modelling and reasoning in Event-B”, *STTT*, vol. 12, no. 6, p. 447-466, 2010.

[BAC 89] BACK R.-J., “Refinement calculus, part II: parallel and reactive programs”, *REX Workshop*, p. 67-93, 1989.

[BUT 97] BUTLER M.J., “An approach to the design of distributed systems with B AMN”, *ZUM*, p. 223-241, 1997.

[BUT 02] BUTLER M., “A system-based approach to the formal development of embedded controllers for a railway”, *Design Automation for Embedded Systems*, vol. 6, p. 355-366, 2002.

[BUT 09] BUTLER M., “Decomposition structures for Event-B”, *Integrated Formal Methods iFM2009, Springer LNCS 5423*, February 2009.

[DAM 08] DAMCHOOM K., BUTLER M., ABRIAL J.-R., “Modelling and proof of a tree-structured file system in Event-B and Rodin”, *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM ’08, p. 25-44, Springer-Verlag, Berlin, Heidelberg, 2008.

[EVE] Event-B and Rodin Website [Online]. http://www.event-b.org/.

[FAL 09] FALAMPIN J., BUTLER M., FITZGERALD J., Deploy deliverable d16 d2.1 pilot deployment in transportation (wp2). http://www.deploy-project.eu/pdf/D16_final6, September 2009.

[LEU 08] LEUSCHEL M., BUTLER M., “ProB: an automated analysis toolset for the B Method”, *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 10, p. 185-203, 2008.

[SIL 11] SILVA R., PASCAL C., HOANG T.S., BUTLER M., “Decomposition tool for Event-B”, *Software, Practice and Experience*, vol. 41, no. 2, p. 199-208, 2011.

[SIL 12] SILVA R., *Supporting Development of Event-B Models*, Draft PhD Thesis, University of Southampton, 2012.

[SNO 06] SNOOK C.F., BUTLER M.J., “UML-B: formal modeling and design aided by UML”, *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, p. 92-122, 2006.

[TSD 12] Transportation Systems Design Inc, Communications based train control, http://www.tsd.org/cbtc/, January 2012.

[WIK 01] Wiki. B2Latex [Online]. http://wiki.event-b.org/index.php/B2Latex.

[WIK 02] Wiki. ProB [Online]. http://wiki.event-b.org/index.php/ProB.

[WIK 03] Wiki. UML-B [Online]. http://wiki.event-b.org/index.php/UML-B.