The original publication is available at https://doi.org/10.1007/978-3-319-67729-3_7
In Proceedings of ICTAC 2017: International Colloquium on Theoretical Aspects of Computing

Class-diagrams for Abstract Data Types

Thai Son Hoang, Colin Snook, Dana Dghaym, and Michael Butler

ECS, University of Southampton, U.K.
{t.s.hoang,cfs,dd4g12,mjb}@ecs.soton.ac.uk

Abstract. We propose to extend iUML-B class-diagrams to elaborate Abstract Data Types (ADTs) specified using Event-B theories. Classes are linked to data types, while attributes and associations correspond to operators of the data types. Axioms about the data types and operators are specified as constraints on the class. We illustrate our approach on a development of a control system in the railway domain.

Keywords: Event-B, iUML-B, Class-diagrams, Theory, Abstract Data Types (ADTs)

1 Introduction

Event-B [1] is a well-established formalism for developing systems whose components can be modelled as discrete transition systems. An Event-B model contains two parts: a dynamic part (called *machine*) modelled by a transition system and a static part (called *context*) capturing the model's parameters and assumptions about them. The main technique in Event-B to cope with system complexity is stepwise *refinement*, where design details are gradually introduced into the formal models. Refinement enables the abstraction of machines, and since abstract machines contain fewer details than concrete ones, they are usually easier to validate and verify.

To enhance the user experience with developing models, Event-B and its supporting *Rodin platform* (Rodin) is extensible. One of the extensions is iUML-B which includes state-machines and class-diagrams [9,10,11]. While state-machines give a visualisation of the system's dynamic state and the transitions between them, class-diagrams provide a visualisation of the model data and relationships. Another extension is the Theory plug-in [3] for extending the mathematical language of Event-B and supporting reasoning about these additional concepts. In particular, we can use Event-B theories to formalise *Abstract Data Types* (ADTs) [7] and subsequently utilise the ADTs to model the system's dynamic behaviour in the machines.

Our motivation is to provide a diagrammatic visualisation for the ADTs specified using Event-B theories. In particular, we propose to extend iUML-B class-diagrams with new and adapted diagrammatic elements, linking them to the data types and operators in the theories. The extension helps the design of the ADTs and provides a better understanding of the data types and the relationships between them.

Our contribution therefore is a proposal for extending iUML-B class-diagrams. Classes are linked to data types specified using theories. Attributes and associations elaborate operators of the data types. Axioms about the data types and operators are specified as class constraints. We illustrate our approach on a development of the RailGround case study [8] provided by Thales Austria GmbH.

The rest of the paper is structured as follows. Section 2 gives some background information about the Event-B method and the extensions such as iUML-B and the Theory plug-in. We present our proposal for extending iUML-B class-diagrams for Event-B theories in Section 3. We illustrate our approach using the Rail Ground case study in Section 4. We give a summary of our development in Section 5 and some conclusion of our work in Section 6.

2 Background

2.1 Event-B

Event-B [1] is a formal method for system development. Main features of Event-B include the use of *refinement* to introduce system details gradually into the formal model. An Event-B model contains two parts: *contexts* and *machines*. Contexts contain *carrier sets*, *constants*, and *axioms* that constrain the carrier sets and constants. Machines contain *variables* v, *invariants* $I(v)$ that constrain the variables, and *events*. An event comprises a guard denoting its enabling-condition and an action describing how the variables are modified when the event is executed. In general, an event `e` has the following form, where t are the event parameters, $G(t, v)$ is the guard of the event, and $v := E(t, v)$ is the action of the event[1].

```
e == any t where G(t,v) then v := E(t,v) end
```

A machine in Event-B corresponds to a transition system where *variables* represent the states and *events* specify the transitions. Contexts can be *extended* by adding new carrier sets, constants, axioms, and theorems. Machine **M** can be *refined* by machine **N** (we call **M** the abstract machine and **N** the concrete machine). The state of **M** and **N** are related by a gluing invariant $J(v, w)$ where v, w are variables of **M** and **N**, respectively. Intuitively, any "behaviour" exhibited by **N** can be simulated by **M**, with respect to the gluing invariant J. Refinement in Event-B is reasoned event-wise. Consider an abstract event `e` and the corresponding concrete event `f`. Somewhat simplifying, we say that `e` is refined by `f` if `f`'s guard is stronger than that of `e` and `f`'s action can be simulated by `e`'s action, taking into account the gluing invariant J. More information about Event-B can be found in [6]. Event-B is supported by Rodin [2], an extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

[1] Actions in Event-B are, in the most general cases, non-deterministic [6].

2.2 iUML-B

iUML-B [9,10,11] provides a diagrammatic modelling notation for Event-B in the form of state-machines and class-diagrams. The diagrammatic elements are contained within an Event-B model and generate or contribute to parts of it. For example a state-machine will automatically generate the Event-B data elements (sets, constants, axioms, variables, and invariants) to implement the states, and contribute additional guards and actions to existing events. Class diagrams provide a way to visually model data relationships. Classes, attributes and associations are linked to Event-B data elements (carrier sets, constants, or variables) and generate constraints on those elements. In this paper, we focus on extending class-diagrams for visualising abstract data types specified using theories.

2.3 Theory Plug-in

The Theory plug-in [3] enables developers to define new (polymorphic) data types and operators upon those data types. These additional modelling concepts might be defined directly (including inductive definitions) or axiomatically.

An (inductive) datatype can be directly defined using several constructors. Each constructor can have zero or more destructors. A datatype without any definition is axiomatically defined. We focus on axiomatic data types in this paper. By convention, an axiomatic datatype satisfies the *non-emptiness* and *maximality* properties, i.e., for an axiomatic type S, we have $S \neq \varnothing$ and $\forall\ e \cdot e \in S$. As an example, an axiomatic type for *stacks* is declared as follows.

```
theory Stack(T)
types STACK(T)
end
```

Operators can be defined *directly*, *inductively* (on inductive data types) or *axiomatically*. An operator defined without any definition will be defined axiomatically. Operator notation is `prefix` by default. Operators with two argument can be `infix`. Further properties can be declared for operators including *associativity* and *commutativity*.

In the following, we show the declaration for some stack operators: `emptyStack`, `top`, `pop`, and `push`.

```
operators
  emptyStack: "STACK(T)"
  top(st : "STACK(T)"): "T"
  pop(st : "STACK(T)"): "STACK"
    for "st ≠ emptyStack"
  push(st : "STACK(T)", e : "T"): "STACK"
  ≺ (e : "T", st : "STACK(T)") infix
axioms
  @axm1 "∀ st, e · e ∈ T ⇒ push(st, e) ≠ emptyStack"
  @axm2 "∀ st, e · e ∈ T ⇒ pop(push(st, e)) = st"
  @axm3 "∀ st, e · e ∈ T ⇒ top(push(st, e)) = e"
  @axm4 "∀ st · st ∈ STACK(T) ∧ st ≠ emptyStack ⇒ top(st) ≺ st"
  @thm1 "∀ st, e·  e ∈ T ⇒ e ≺ push(st, e)" theorem
```

An additional `infix` operator $\prec$ defines a predicate (without any returning type) specifying whether an element `e` is in the stack `st` or not. The axioms are the assumptions about these operators that can be used to define proof rules. Note that `@thm1` is a theorem which is derivable from the axioms defined previously. We omit the presentation of proof rules in this paper.

Finally, theories can be constructed in hierarchical manner: a theory can *extend* other theories by adding more data types, operators, and axioms.

3 Class-diagrams for Abstract Data Types

An ADT is a mathematical model of a class of data structures. It is typically defined by a set of operations that can be performed on the ADT, along with a specification of their effect. By using Event-B theories to formalise ADTs, we can subsequently utilise the ADTs to model the system's dynamic behaviour in the machines. An ADT can be specified straightforwardly using Event-B theories with axiomatic data type and operators, e.g., the `STACK` data type in Section 2.3.

In order to aid the design of ADTs, we propose to extend class diagrams to ADTs that are specified using theories. In particular, data types are represented using classes and operators are modelled using attributes or associations. We illustrate our idea using the `STACK` data type example. The class-diagram for the `STACK` data type is shown in Figure 1. In the diagram, there are two classes,

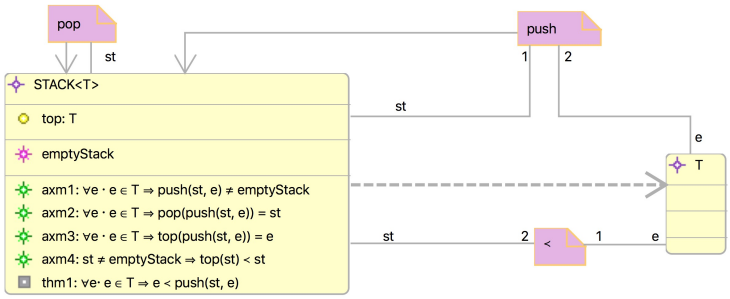

Fig. 1: A class-diagram for Stack ADT

namely `STACK` and `T`. The dashed arrow from `STACK` to `T` indicates that `STACK` is polymorphic and `T` is the type parameter of `STACK`. This is also denoted by the label, i.e., `STACK<T>`, of the `STACK` class. Since class `T` represents a formal type parameter it cannot own any child features such as associations or constraints.

For now, we use the existing class diagram features to illustrate the proposed approach. Our intention is to add features to iUML-B to represent the ADT features with new diagram elements including a new class container for adding class constant instances, a new arrow/label feature for expressing class type pa-

rameters, a new diagram node for multi-source associations, and a new diagram node for representing abstract formal type parameters such as T.

Operators are represented by the special associations between classes. Each association operator can have one or more inputs and zero or one outputs. An operator without any output, e.g., $\prec$, indicates a *predicate*. The inputs to operators are labelled to indicate their formal parameters. If an operator has two or more inputs, e.g., push or $\prec$, each input is numbered (e.g., $1, 2, \ldots$) specifying their order.

A "query" operator, i.e., those with one input which is an instance of the data type and one output (e.g., top), can be specified as attributes of the class. An operator without any input and return an instance of the data type, i.e., a constant of the data type (e.g., emptyStack), is specified using a "constant" of the class. Finally, the axioms and theorems about the data type and its operators are specified as constraints on the class. The constraints are lifted automatically to all instances of the data type. Let st be the instance name for the STACK data type, @axm1 becomes

$$\forall\ \texttt{st} \cdot \texttt{st} \in \texttt{STACK} \Rightarrow (\forall\ \texttt{e} \cdot \texttt{e} \in \texttt{T} \Rightarrow \texttt{push(st, e)} \neq \texttt{emptyStack}) .$$

Note that in general, the class-diagrams and their corresponding theories for ADTs are developed gradually through several steps. In each step, additional data types, operators, and constraints can be added.

4 Example. An Interlocking System

The example used in this paper is based on a formal model of a railway interlocking system, which was developed by Thales Austria GmbH. This is a simplified version of interlocking systems, built specifically for research on formal validation and verification of railway systems [8]. This example is used as part of the rail use case of the European project *Enable-S3* [4].

4.1 Requirements

Railway systems, in general, aim at providing a timely, efficient and most importantly a safe train service. This requires a reliable command and control system that ensures a train can safely enter its specified route. In the system under consideration, the railway topology consists of a set of connected elements, which are controlled by signals passing information to the trains. The safety of a train is ensured by allowing its route to be set, only if it does not conflict with the current available routes. The following requirements are extracted and simplified from [8]. For illustration, we will consider the network topology with one track and two points as in Figure 2.

Rail Elements The railway topology is formed by a set of rail elements. A *Rail Element* is a unit which provides a physical running path for the trains, i.e. rails

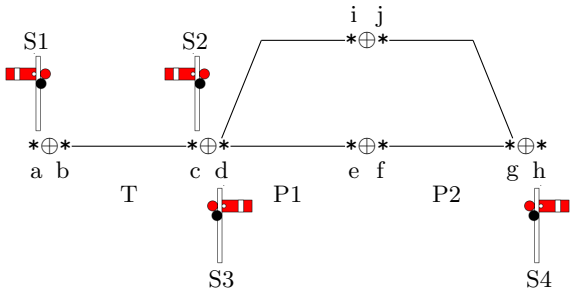

Fig. 2: An example railway topology [8]

(e.g. track, points, crossing). Typically, a rail element is made up of one or more *segments*. The sets of segments belong to each rail element are disjoint.

REQ 1 The network topology is a set of rail elements.

REQ 2 A rail element contains one or more segments

In Figure 2, the segments are {bc,cb,di,id,de,ed,jg,gj,fg,gf}. There are three rail elements, namely T (a track), P1, P2 (points). The relationship between the rail elements and the segments are as follows:

$$\texttt{T} \mapsto \{\texttt{bc},\texttt{cb}\},\ \texttt{P1} \mapsto \{\texttt{di},\texttt{id},\texttt{de},\texttt{ed}\},\ \texttt{P2} \mapsto \{\texttt{jg},\texttt{gj},\texttt{fg},\texttt{gf}\}.$$

Element Positions For each rail element, a *Rail Element Position* is a distinct situation of that rail element. Furthermore, each element position defines the set of possible element connections (defined by segments) for that particular rail element.

REQ 3 For each rail element, there is a set of possible element positions

REQ 4 Each rail element and position correspond to a set of rail segments

For example, a points has three possible position POS_X (in transition), POS_L (left), POS_R (right). Consider points P1, position POS_X corresponds to an emptyset of segments, POS_L corresponds to segments {di, id}, and POS_R corresponds to segments {de, ed}.

Paths A path is a sequence of rail segments, with the constraint that two rail segments of the same rail element are not allowed within one path. A path can be activated so that trains are allowed to be on that path.

REQ 5 A path is a sequence of rail segments.

REQ 6 Two rail segments belonging to the same element are not allowed within one path.

Consider the example in Figure 2, a path could be the following sequence of segments [bc,di,jg], or [gf,ed,cb]. Note that any sub-sequence of a path is also a path, e.g., [di, jg] is also a path.

Route Life-Cycle A set of routes are defined. Each route correspond to a pre-defined path in the network. Before becoming active, a route must be requested. As soon as all conditions for the route (e.g., rail elements must be in the required position to establish its path), a requested route can be activated. A path corresponds to an active route is called active path. As a train moves along a route, rail elements that are no longer in use can be released. An active route can be removed only after all its rail elements are released. A rail element position can only be changed if the rail element is not part of an active path.

REQ 7 A requested route can become an active route when all conditions for that route are met

REQ 8 An active route can be removed only after all its rail elements are released.

REQ 9 A rail element position can only be changed if it is not part of an active path.

In the example network topology, we can have the following routes R1–R4, with the following associations: R1 $\mapsto$ [bc, de, fg], R2 $\mapsto$ [bc, di, jg], R3 $\mapsto$ [gf, ed, cb], R4 $\mapsto$ [gj, id, cb].

Vacancy Detection To simplify, we assume that each rail element corresponds to exactly one *Track Vacancy Detection* (TVD) section. The state of the TVD section is either vacant or occupied. A TVD section is occupied if there is some train on some segment belonging to the rail element.

REQ 10 Each rail element corresponds to exactly one TVD section.

REQ 11 A TVD section can be either in vacant or occupied state.

Signals A signal is an entity capable of passing information to trains. A signal is associated with a rail element for a particular traversal direction. A signal aspect is an (abstract) information conveyed by a signal. Signal Default is a predefined aspect of signals. Trains are assumed to obey the signals, in particular, stop at a signal containing default aspect.

REQ 12 A signal is associated with a rail element.

REQ 13 A signal may be set to an aspect other than default, only if there is an active element after this signal.

In Figure 2, we have 4 signals, S1–S4. Note that both S1 and S3 associated with T, but they protect the rail element in different traversal directions.

Safety Properties Safety in this model is ensured by the paths which are active. The paths can only be set if all its elements are in the right positions. Safety is ensured by preventing paths to be requested if there are other paths requiring the same elements.

REQ 14 Two active paths cannot overlap

REQ 15 An active path must have all its elements in the right positions

REQ 16 A route can be requested if it is disjoint from other active or requested routes.

4.2 Development

For this paper, we omit the presentation of the proof rules associated with the theories. Most of them are directly inferred from the axioms constraining the data types. For the example, we abstract from rail segments. Details about rail segments (e.g., **REQ 2**, **REQ 4**, **REQ 5**, **REQ 6**) can be introduced later via refinement. The development is available online at `http://doi.org/10.5258/SOTON/D0162` including instructions on Rodin configuration.

Refinement strategy We adopt the following refinement strategy for developing a model of the system. The requirements taken into account at each refinement level is also listed.

- **M0**: To abstractly specify active routes in the system, focusing on collision-free properties (**REQ 14**).
- **M1**: To introduce the life-cycle of routes by specifying requested routes (**REQ 7**, **REQ 16**).
- **M2**: To formalise the rail elements and the link between rail elements and paths (**REQ 1**, **REQ 8**).
- **M3**: To specify the element positions and their association with the rail elements (**REQ 3**, **REQ 15**, **REQ 9**).
- **M4**: To introduce the track vacancy detection mechanism (**REQ 10**, **REQ 11**).
- **M5**: To introduce the signals controlling the trains' movement (**REQ 12**, **REQ 13**).

M0. Paths In the initial model, we focus on the notion of paths and the relationships between them (abstractly). In particular, our model of the dynamic of the system centres around the main safety property of the system, i.e., collision-free (**REQ 14**). For this, we want to specify that there are no overlaps between currently active paths. The diagram for the initial theory of the `PATH` data type can be seen in Figure 3. Two operators, namely $\oplus$ and $\sqsubseteq$, are introduced to specify *disjointness* and *sub-path* relationships between two paths `p1` and `p2`. Properties of the operators are specified by constraints `@axm1` and `@axm2`. Constraint `@axm1` states that $\oplus$ is symmetric and `@axm2` states that disjointness is

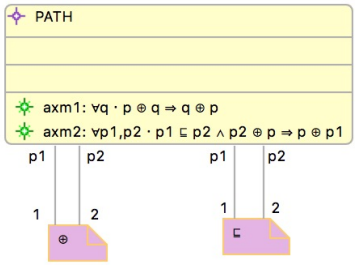

Fig. 3: Class-diagrams in M0

preserved by the sub-path relationship. The corresponding theory can be seen as follows. Note that the constraints are lifted to be universally quantified over all instance p of the PATH data type.

```
theory Paths_01
types
  PATH
operators
  ⊕ (p1: "PATH", p2 : "PATH") infix
  ⊑ (p1: "PATH", p2 : "PATH") infix
axioms
  @axm1 "∀ p· p ∈ PATH ⇒ (∀ q · p ⊕ q ⇒ q ⊕ p)"
  @axm2 "∀ p· p ∈ PATH ⇒ (∀ p1,p2 · p1 ⊑ p2 ∧ p2 ⊕ p ⇒ p ⊕ p1)"
end
```

We can use the PATH data type to specify our dynamic system as follows. Context C0_RG_Paths declares a carrier set ROUTE denoting a set of pre-defined routes. Constant path links the routes with its initial paths (specified by the PATH data type).

```
context C0_RG_Paths
sets ROUTE
constants path
axioms
  @axm1: "path ∈ ROUTE → PATH"
end
```

In machine M0_RG_Paths, variable path_curr is introduced to capture the active routes. Invariant @inv1 associates each active route with some path. Invariant @inv2 specifies the collision-free property: two different active routes must be disjoint.

```
machine M0_RG_Paths
sees C0_RG_Paths
variables path_curr
invariants
  @inv1: "path_curr ∈ ROUTE ⇸ PATH"
  @inv2: "∀ pth1, pth2 ·
     pth1 ∈ dom(path_curr) ∧ pth2 ∈ dom(path_curr) ∧ pth1 ≠ pth2 ⇒
                 path_curr(pth1) ⊕ path_curr(pth2)"
INITIALISATION == begin @act1: "path_curr := ∅ " end
```

Three events are modelled at this specification level for adding, modifying, and removing routes. In addRoute, a new route pe, where the corresponding path (i.e., path(pe)) does not conflict with any existing routes (addRoute's @grd2), is activated. The initial path associated with pe is path(pe). Event modifyRoute updates the path corresponding to the route pe with the new path pth. Guard @grd2 of modifyRoute specifies that the new path pth must be a sub-path of the current path associated with pe (a route can only be updated by releasing rail elements which no longer in use). Finally, event removeRoute removes an active route specified by route pe from the set of active routes.

```
events
  addRoute ==
  any pe where
    @grd1: "pe ∈ ROUTE \ dom(path_curr)"
    @grd2: "∀ p · p ∈ dom(path_curr) ⇒ path(pe) ⊕ path_curr(p)"
  then
    @act1: "path_curr(pe) := path(pe)"
  end

  modifyRoute ==
  any pe pth where
    @grd1: "pe ∈ dom(path_curr)"
    @grd2: "pth ⊑ path_curr(pe)"
  then
    @act1: "path_curr(pe) := pth"
  end

  removeRoute ==
  any pe where
    @grd1: "pe ∈ dom(path_curr)"
  then
    @act2: "path_curr := {pe} ⩤ path_curr"
  end
end
```

M1. Route Life-Cycle In the first refinement, we model the life-cycle of routes by introducing the notion of requested routes. In this refinement, there are no changes for the PATH data type. Variable path_req captures the set of requested routes (i.e., a subset of ROUTE) which must be disjoint from the set of current routes (@inv2).

```
invariants
  @inv1: "path_req ⊆ ROUTE"
  @inv2: "path_req ∩ dom(path_curr) = ∅ "
```

We refine event addRoute as follows, i.e., a requested route pe becomes an active route.

```
addRoute
refines addRoute
any pe where
  @grd1: "pe ∈ path_req"
then
  @act1: "path_curr(pe) := path(pe)"
  @act2: "path_req := path_req \ {pe}"
```

```
8 end
```

In order to prove the refinement of event addRoute, we need additional invariants linking path_req and path_curr.

```
1 @inv3: "∀ pth1, pth2 · pth1 ∈ path_req ∧ pth2 ∈ dom(path_curr) ⇒ path(pth1) ⊕ path_curr
      (pth2)"
2 @inv4: "∀ pth1, pth2 · pth1 ∈ path_req ∧ pth2 ∈ path_req ∧ pth1 ≠ pth2 ⇒ path(pth1) ⊕
      path(pth2)"
```

Two new events requestRoute and removeRequest are introduced to create a new request for a path and remove an existing request. Notice the guards of requestRoute ensure the maintenance of invariants @inv3 and @inv4.

```
1  requestRoute ==
2  any pe where
3    @grd1: "pe ∈ ROUTE \ path_req"
4    @grd2: "pe ∉ dom(path_curr)"
5    @grd3: "∀ p · p ∈ dom(path_curr) ⇒ path(pe) ⊕ path_curr(p)"
6    @grd4: "∀ p · p ∈ path_req ⇒ path(pe) ⊕ path(p)"
7  then
8    @act1: "path_req := path_req ∪ {pe}"
9  end
10
11 removeRequest ==
12 any pth where
13   @grd1: "pth ∈ path_req"
14 then
15   @act1: "path_req := path_req \ {pth}"
16 end
```

M2. Rail Elements In this refinement, we introduce the rail elements into the formal models. A new data type RAIL_ELEMENT is introduced. We extend the PATH data type with a new operator rail_elements returning the set of rail elements associated with each path (see Figure 4). Another operator ≪ specifying whether a rail element re belongs to some path p or not is defined using the direct definition, i.e. re ≪ p == re ∈ rail_elements(p). Finally, we introduce an operator shrink for removing a rail element re from the path p. The shrink operator is only defined for the rail element re belonging to the path p. Axiom @axm1 defines the disjointness between paths p and q as the disjointness of their rail elements. Axioms @axm2 and @axm3 specify the properties of shrink operator: it makes the path p smaller and removes the element re from the path's rail elements. The corresponding theory is as follows.

```
1  theory Paths_02
2  imports Paths_01 RailElement_01
3  operators
4    rail_elements(p: "PATH"): "(RAIL_ELEMENT)"
5    ≪ (re: "RAIL_ELEMENT", p : "PATH") infix =
6      "re ∈ rail_elements(p)"
7    shrink(p : "PATH", re : "RAIL_ELEMENT"): "PATH"
8      for "re ≪ p"
9  axioms
10   @axm1 "∀ p · p ∈ PATH ⇒ (∀ q · p ⊕ q ⇔ rail_elements(p) ∩ rail_elements(q) = ∅ )"
```

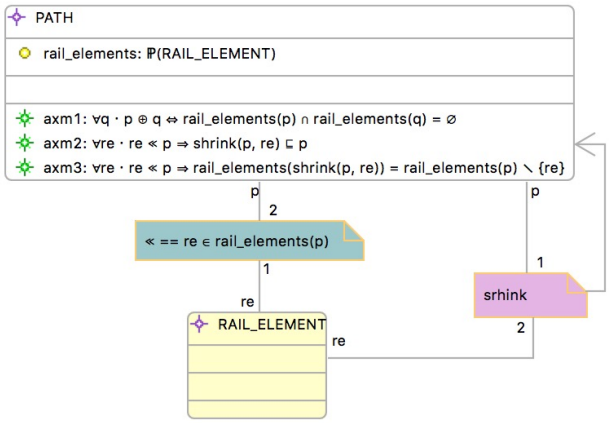

Fig. 4: Class-diagrams in M2

```
  @axm2 "∀ p · p ∈ PATH ⇒ (∀ re · re ≪ p ⇒ shrink(p, re) ⊑ p)"
  @axm3 "∀ p · p ∈ PATH ⇒ (∀ re · re ≪ p ⇒ rail_elements(shrink(p, re)) =
       rail_elements(p) \ {re})"
end
```

For the dynamic system, a variable `rail_element_path` is introduced to keep the relationship between the rail elements and the current active route. Each rail element is associated with at most one active route (`@inv1`). Invariant `@inv2` states the consistency between `rail_element_path` and the set of rail elements associated with some active route `p`.

```
@inv1: "rail_element_path ∈ RAIL_ELEMENT ⇸ dom(path_curr)"
@inv2: "∀ p· p ∈ dom(path_curr) ⇒ rail_elements(path_curr(p)) = rail_element_path∼[{p}]"
```

We focus on the refinement of `modifyRoute` in this level. The changes to the other events are trivial. With the introduction of the `shrink` operator, we can now be more precise about how an active route is modified, i.e., it can be done by releasing some no longer used rail element `re`.

```
modifyRoute
refines modifyRoute
any pe re where
  @grd1: "pe ∈ dom(path_curr)"
  @grd2: "re ≪ path_curr(pe)"
with
  @pth: "pth = shrink(path_curr(pe), re)"
then
  @act1: "path_curr(pe) := shrink(path_curr(pe), re)"
  @act2: "rail_element_path := {re} ⩤ rail_element_path"
end
```

The witness for `pth` (using the `with` clause) specifies the value of the removed abstract parameter `pth`. The rail element `re` is removed from the domain of `rail_element_path`: it is no longer associated with any active path.

M3. Element Positions In this refinement, we introduce the positions for rail elements. We introduce a new ADT, namely RAIL_POSITION (see Figure 5). A new operator $\sqsubset$ is added to the RAIL_ELEMENT ADT. For an element position rp and a rail element re, ep $\sqsubset$ re states that rp is a valid position for re. An additional operator Default for RAIL_ELEMENT which returns the default position for each rail element. Axiom @axm1 states that the default position for a rail element re is always a valid one for that rail element. Finally, an operator Path_Element_Pos is added to the PATH ADT which returns a (partial) function relating the rail elements (belonging to the path) with the element position. Axiom @axm2 states that the position defined for a rail element re of a path p must be a valid position for re. Axiom @axm3 gives the relationship between rail_elements and Path_Element_Pos as expected.

We introduce a variable rail_positions to capture the current position of every rail element (@inv1 below). Invariant @inv2 states that the position of every rail element re must be a valid one for re. Invariant @inv3 specifies the important safety property for each current active route: the position of the rail elements that belong to the active route must be the correct position.

```
invariants
  @inv1: "rail_positions ∈ RAIL_ELEMENT → RAIL_POSITION"
  @inv2: "∀ re · rail_positions(re) ⊏ re"
  @inv3: "∀ p, re · p ∈ dom(path_curr) ∧ re ≪ path_curr(p) ⇒ rail_positions(re) =
       Path_Element_Pos(path_curr(p))(re)"
```

An additional guard @grd2 is added to addRoute event as follows.

```
@grd2: "∀ re · re ⊏ path(pe) ⇒ rail_positions(re) = Path_Element_Pos(path(pe))(re)"
```

The guard ensures that only when every rail element re that belongs to a requested route pe is in the correct position, can this route pe can be turned into a current route. Two new events are added for setting the position of a rail element: setRailElementPos and setRailElementPath.

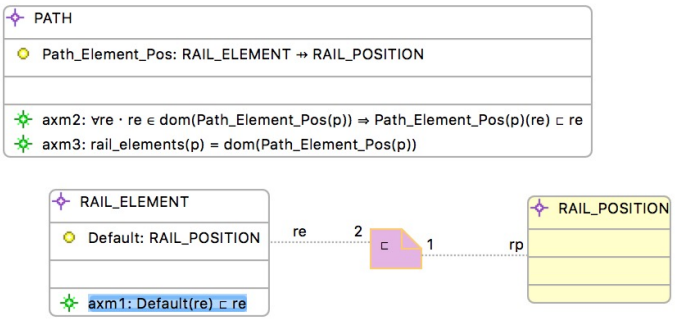

Fig. 5: Class-diagrams in M3

```
setRailElementPos
any re pos where
  @grd1: "re ∉ dom(rail_element_path)"
  @grd2: "∀ p · p ∈ path_req ⇒  re ≪ path(p)"
  @grd3: "pos ⊏ re" // @pos is valid
  @grd4: "pos ≠ rail_positions(re)" // @pos is new
then
  @act: "rail_positions(re) := pos"
end

setRailElementPath
any p re where
  @grd1: "p ∈ path_req" // @p is a requested path.
  @grd2: "re ≪ path(p)" // @re is a rail element of @p
  @grd3: "Path_Element_Pos(path(p))(re) ≠ rail_positions(re)"
then
  @act: "rail_positions(re) := Path_Element_Pos(path(p))(re)"
end
```

Event `setRailElementPos` sets the new position `pos` for a rail element `re` which does not belong to any active path (`@grd1`) and does not belong to any requested route (`@grd2`). Event `setRailElementPath` sets the position for a rail element `re` belonging to a requested route `p`. The new position of the element `re` is the position required for path `p` as specified by the operator `Path_Element_Pos`.

M4. Vacancy Detection In this refinement, we introduce the track vacancy detection. Each TVD section corresponds to a rail element. As a result, we introduce a new data type `TVD_SECTION` with an operator `TVD_Element` as in Figure 6. Axioms `@axm1` and `@axm2` ensure the one-to-one relationship between

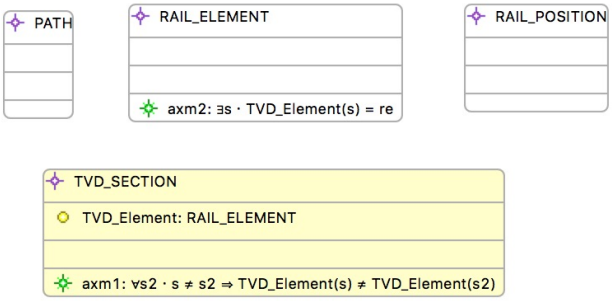

Fig. 6: Class-diagrams in M4

`TVD_SECTION` and `RAIL_ELEMENT`.

We introduce a new variable `TVD_status` to capture the current vacancy status of the TVD sections. The invariants for this refinement level are as follows,

```
@inv1: "TVD_status ∈ TVD_SECTION → TVD_STATE_ENUM"
@inv2: "∀ s · TVD_status(s) = TVD_STATE_OCCUPIED ⇒ TVD_Element(s) ∈ dom(
      rail_element_path)"
```

where TVD_STATE_ENUM is a data type with two elements: TVD_STATE_OCCUPIED and TVD_STATE_VACANT.

Invariant @inv2 states that if a TVD section s is occupied then the corresponding rail element must be a part of an active path. This corresponds to the assumption that trains cannot go out of the current active paths.

Event modifyRoute is extended as follows.

```
modifyRoute extended
refines modifyRoute
any s where
  @grd2: "TVD_status(s) = TVD_STATE_OCCUPIED"
  @grd3: "re = TVD_Element(s)"
then
  @act3: "TVD_status(s) := TVD_STATE_VACANT"
end
```

The additional parameter s denotes the TVD section corresponding to the rail element re (@grd3). The status of s is changed from occupied to vacant in this modifyRoute event. Essentially, this event models the situation where a train departs from the rail element re (hence the TVD status changed from occupied to vacant) and the rail element re is released.

A new event setTVDStatus is introduced for changing the status of a TVD section from vacant to occupied.

```
setTVDStatus
any s where
  @grd1: "TVD_status(s) = TVD_STATE_VACANT"
  @grd2: "TVD_Element(s) ∈ dom(rail_element_path)"
then
  @act1: "TVD_status(s) := TVD_STATE_OCCUPIED"
end
```

Guard @grd2 ensures that the rail element is currently within some active path.

M5. Signal In this refinement, we introduce the signals and signal aspects. Two new ADTs are introduced: SIGNAL and SIGNAL_ASPECT_ENUM (Figure 7). The SIGNAL data type has one operator, namely Signal_Element, returning the rail element that the signal protects. The SIGNAL_ASPECT_ENUM has a constant, namely SIGNAL_ASPECT_DEFAULT, representing the default aspect of the signals. No additional assumptions are made about SIGNAL and SIGNAL_ASPECT_ENUM.

We introduce a variable signal_status to model the status of all the signals.

```
@inv1: "signal_status ∈ SIGNAL → SIGNAL_ASPECT_ENUM"
```

Event setTVDStatus is refined by two events according to whether or not the rail element is protected by a signal. Event setTVDStatusPath captures the case where the rail element corresponding to the TVD section s is not protected by a signal. This reflects the situation where a train is moving along an existing path. Event setTVDStatusSignal corresponds to the case where the rail element is protected by a signal. Note that the signal is turned automatically to SIGNAL_ASPECT_DEFAULT when the train occupied the element.

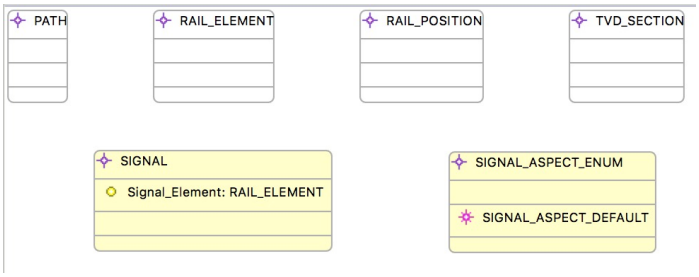

Fig. 7: Class-diagrams in M5

```
setTVDStatusPath refines setTVDStatus
any s where
  @grd1: "TVD_status(s) = TVD_STATE_VACANT"
  @grd2: "TVD_Element(s) ∈ dom(rail_element_path)"
  @grd3: "∀ sg · TVD_Element(s) ≠ Signal_Element(sg)"
then
  @act1: "TVD_status(s) := TVD_STATE_OCCUPIED"
end

setTVDStatusSignal refines setTVDStatus
any s sg where
  @grd1: "TVD_status(s) = TVD_STATE_VACANT"
  @grd2: "signal_status(sg) ≠ SIGNAL_ASPECT_DEFAULT"
  @grd3: "TVD_Element(s) = Signal_Element(sg)"
then
  @act1: "TVD_status(s) := TVD_STATE_OCCUPIED"
  @act2: "signal_status(sg) := SIGNAL_ASPECT_DEFAULT"
end
```

In order to prove the correctness of the refinement of `setTVDStatus` by `setTVDStatusSignal`, we need the following invariants. Invariants `@inv2` and `@inv3` state that if the signal status for `sg` is not `SIGNAL_ASPECT_DEFAULT` then (1) the rail element corresponding to the signal must belong to some active path and (2) the rail element must be vacant as detected by the TVD section. Invariant `@inv4` states that if two signals `sg1` and `sg2` protecting the same rail element and `sg1` is not `SIGNAL_ASPECT_DEFAULT` then `sg2` must have the default aspect.

```
@inv2: "∀ sg · signal_status(sg) ≠ SIGNAL_ASPECT_DEFAULT ⇒ Signal_Element(sg) ∈ dom(
    rail_element_path)"
@inv3: "∀ sg, s · signal_status(sg) ≠ SIGNAL_ASPECT_DEFAULT ∧ TVD_Element(s) =
    Signal_Element(sg) ⇒ TVD_status(s) = TVD_STATE_VACANT"
@inv4: "∀ sg1, sg2 · signal_status(sg1) ≠ SIGNAL_ASPECT_DEFAULT ∧ Signal_Element(sg1) =
    Signal_Element(sg2) ∧ sg1 ≠ sg2 ⇒ signal_status(sg2) = SIGNAL_ASPECT_DEFAULT"
```

A new event to set the signal aspect to proceed (i.e., not the default aspect) as follows, taking into account the above invariants.

```
setSignalAspectProceed
any sg asp s where
  @grd1: "signal_status(sg) = SIGNAL_ASPECT_DEFAULT"
  @grd2: "asp ≠ SIGNAL_ASPECT_DEFAULT"
  @grd3: "Signal_Element(sg) ∈ dom(rail_element_path)"
```

```
6   @grd4: "TVD_Element(s) = Signal_Element(sg)"
7   @grd5: "TVD_status(s) = TVD_STATE_VACANT"
8   @grd6: "∀ sg1 · Signal_Element(sg) = Signal_Element(sg1) ∧ sg ≠ sg1 ⇒ signal_status(
        sg1) = SIGNAL_ASPECT_DEFAULT"
9  then
10  @act1: "signal_status(sg) := asp"
11 end
```

5 Summary

Our RailGround development using theories contains 6 machines, i.e., M0–M5 forming a refinement-chain. Out of the total 147 proof obligations, 95% (139) are discharged automatically. This high perchantage of automatic proofs is due to the carefully constructed ADTs with appropriate axioms and proof rules supporting the reasoning.

Typically we develop Event-B models to express important (safety) properties at a very abstract level and then make a series of refinements to gradually introduce the details of a design mechanism that maintains this property. The RailGround model is atypical in that it begins by modelling the established principals of interlocking systems without modelling the safety properties that those systems are designed to achieve. The reason for this is that the principles of interlocking are a proven design mechanism for controlling trains in a safe way. The model focusses instead on providing a precise and accurate specification of the interlocking product-line. Nevertheless, the model provides a good case study to illustrate the use of our diagrammatic representation of ADTs linked to Event-B theories including sufficient properties concerning the lack of conflicts in paths.

6 Conclusion

In this paper, we propose an extension to class-diagrams elaborating ADTs specified using Event-B theories. Classes are linked to data types, while attributes and associations correspond to operators of the data types. Axioms about the data types and operators are specified as constraints on the class. We illustrate our approach on a development of RailGround case study provided by Thales Austria GmbH. The diagrammatic visualisation helps us to design appropriate theories supporting the system development. Moreover, the diagrams and their corresponding theories can be developed gradually and integrated seemlessly with the refinement development process of Event-B.

In the future, we plan to implement our proposal by extending iUML-B. Furthermore, we plan to incorporate other techniques such as instantiation [5] to support the development of theories. Currently, during the development, we extending our class-diagrams with new data types, operators and axioms. This result in data type with several operators and constraints. A possibility for ADT is that they contain contradict axioms. An alternative to data type extension

is instantation where one or more operators is "replaced" by new ones. For example, when we introduce the `rail_elements` operator for paths, we can instantiate $\oplus$ (i.e., define it) using `rail_elements` and prove the axioms about $\oplus$ can be derived from the properties of `rail_elements`. Compare to extension, instatiation will result in more concrete and smaller data types.

Acknowledgement

This work has been conducted within the ENABLE-S3 project that has received funding from the ECSEL Joint Undertaking under Grant Agreement no. 692455. This Joint Undertaking receives support from the European Union's HORIZON 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.
3. Michael J. Butler and Issam Maamria. Practical theory extension in event-b. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013.
4. The Enable-S3 Consortium. Enable-S3 European project, 2016. `enable-s3.eu`.
5. Andreas Fürst, Thai Son Hoang, David A. Basin, Naoto Sato, and Kunihiko Miyazaki. Large-scale system development using abstract data types and refinement. *Sci. Comput. Program.*, 131:59–75, 2016.
6. Thai Son Hoang. An introduction to the Event-B modelling method. In *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, 2013.
7. B. Liskov and S. Zilles. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM. `http://doi.acm.org/10.1145/800233.807045`.
8. Klaus Reichl. Railground Model on github, 2016. `https://github.com/klar42/railground/` (Accessed 20/04/2017).
9. Mar Yah Said, Michael Butler, and Colin Snook. A method of refinement in UML-B. *Softw. Syst. Model.*, 14(4):1557–1580, October 2015.
10. Colin Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, pages 29–30, Toulouse, France, 2014. `http://eprints.soton.ac.uk/365301/`.
11. Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.