Formal System Modelling Using Abstract Data Types in Event-B

Andreas Fürst[1], Thai Son Hoang[1], David Basin[1], Naoto Sato[2], and Kunihiko Miyazaki[2]

[1] Institute of Information Security, ETH-Zurich, Switzerland
fuersta,htson,basin@inf.ethz.ch
[2] Yokohama Research Lab, Hitachi Ltd., Japan
naoto.sato.je,kunihiko.miyazaki.zt@hitachi.com

Abstract. We present a formal modelling approach using *Abstract Data Types* (ADTs) for developing large-scale systems in Event-B. The novelty of our approach is the combination of refinement and instantiation techniques to manage the complexity of systems under development. With ADTs, we model system components on an abstract level, specifying only the necessary properties of the components. At the same time, we postpone the introduction of their concrete definitions to later development steps. We evaluate our approach using a large-scale case study in train control systems. The results show that our approach helps reduce system details during early development stages and leads to simpler and more automated proofs.

Keywords: Event-B, refinement, abstract data types.

1 Introduction

Event-B [3] is a formalism for developing systems whose components can be modelled as discrete transition systems. An Event-B model contains two parts: a dynamic part (called *machines*) modelled by a transition system and a static part (called *contexts*) capturing the model's parameters and assumptions about them. Event-B's main technique to cope with system complexity is stepwise *refinement*, where design details are gradually introduced into the formal models. Refinement enables abstraction of machines, and since abstract machines contain fewer details than concrete ones, they are usually easier to verify.

However, when developing large, complex systems, refinement alone is often insufficient. Machines containing sufficient details to state and prove relevant safety properties may lead to proofs of unmanageable complexity. We observed this limitation while developing a large-scale train control system by refinement in Event-B. To specify and reason about collision-freeness properties, we needed to model the trains in detail, for example formalising their layout and movement. As a consequence, we had to state numerous complex invariants which resulted in many complicated manual proofs. This motivated an alternative approach to abstract away additional details from the system's model to reduce the complexity and increase the automation of the resulting proofs.

Approach. To model a system at a more abstract level, we introduce the notion of *Abstract Data Types* (ADTs) [16] in Event-B. An ADT is a mathematical model of a class of data structures. It is typically defined in terms of a set of operations that can be performed on the ADT, along with a specification of their effect. By using Event-B contexts to formalise ADTs and their operations, we can subsequently utilise the ADTs to model the system's dynamic behaviour in the machines. We use generic instantiation [5] as a means to further concretise and thereby implement the ADTs. As the ADTs evolve, the machines are also refined accordingly.

We evaluate our approach by developing a substantial industrial case study in the railway domain. Given an informal specification of a train control system, we incrementally develop a formal model of the overall system. This includes modelling the trains, the interlocking system, and the train controller. The complexity of the case study is comparable with that of real train control systems such as CBTC [15] or ETCS Level 3 [6]. We develop the controller all the way to a concrete implementation that runs on specialised hardware. To our knowledge, this is the first published development of a train control system on the system level, *i.e.*, modelling the train controller together with its environment, that is correct-by-construction.

Contribution. Our contribution is the introduction of ADTs in Event-B. We show that reasoning using ADTs can be done purely based on the properties of the ADTs' operations, regardless of how the ADTs will be implemented. As a result, systems specified with ADTs are more abstract and hence easier to verify than systems developed directly without them. In fact, ADTs encapsulate part of the system's dynamic behaviour in the static context of Event-B. This is novel as traditionally Event-B contexts are only used to specify static parameters of a system's model and all dynamic behaviour is modelled as a transition system in the Event-B machines. Furthermore, our use of generic instantiation in Event-B is novel as this technique has until now only been applied to reuse developments, for example in [19]. In contrast, we use generic instantiation as a mechanism to gradually introduce details into the formal models similar to refinement.

The way we introduce ADTs in Event-B allows ADTs to be used alongside Event-B refinement. Hence, one can combine these two different abstraction techniques during development and apply whichever fits better at a particular development stage and results in simpler proofs. In contrast to development strategies that use refinement or ADTs exclusively, our approach is better suited for developing large-scale industrial systems.

Structure. The rest of our paper is structured as follows. In Section 2, we briefly review Event-B, including refinement and instantiation techniques. We motivate and present our approach in Section 3. We evaluate our approach on an industrial case study in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 The Event-B Modelling Method

Event-B [3] represents a further evolution of the classical B-method [1], which has been simplified and focused around the general notion of *events*. Event-B has a semantics based on transition systems and simulation between such systems. We will not describe in detail Event-B's semantics here; full details are provided in [3]. Instead, we will describe some Event-B modelling concepts that are important for the later presentation.

Event-B models are organized in terms of the two basic constructs: *contexts* and *machines*.

Contexts. Contexts specify the static part of a model and may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types. Axioms constrain carrier sets and constants, whereas theorems express properties derivable from axioms. The role of a context is to isolate the parameters of a formal model (carrier sets and constants) and their properties, which are intended to hold for all instances.

Machines. *Machines* specify behavioral properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, and *events*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Theorems are properties derivable from the invariants. Possible state changes are described by events. An event e can be represented by the term

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } G(t, v) \textbf{ then } S(t, v) \textbf{ end} \ ,$$

where t is the event's *parameters*, $G(t, v)$ is the event's *guard* (the conjunction of one or more predicates), and $S(t, v)$ is the event's *action*. The guard states the condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. The action of an event is composed of one or more *assignments* of the form $x := E(t, v)$, where x is a variable in v. Assignments in Event-B may also be nondeterministic, but we omit this additional complexity here as it is not used in this paper. All assignments of an action $S(t, v)$ occur simultaneously. A dedicated event without any parameters or guard is used for *initialisation*.

Refinement. *Refinement* provides a means to gradually introduce details about the system's dynamic behaviour into formal models [3]. A machine $\mathbf{CM}$ can refine another machine $\mathbf{AM}$. We call $\mathbf{AM}$ the *abstract* machine and $\mathbf{CM}$ the *concrete* machine. The states of the abstract machine are related to the states of the concrete machine by *gluing invariants* $J(v, w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine. A special case of refinement (called superposition refinement) is when v is kept in the refinement, i.e. $v \subseteq w$. Intuitively, any behaviour of $\mathbf{CM}$ can be simulated by a behaviour of $\mathbf{AM}$ with respect to the gluing invariant $J(v, w)$.

Refinement can be reasoned about on a per-event basis. Each event e of the abstract machine is *refined* by one or more concrete events f. Simplifying somewhat, we can say that f refines e if f's guard is stronger than e's guard (*guard strengthening*), and the gluing invariants $J(v, w)$ establish a simulation of f by e (*simulation*).

Instantiation. *Instantiation* is a common technique for reusing models by providing concrete values for abstract model parameters. Since an Event-B model is parameterised by the carrier sets and constants, instantiation in Event-B [5,19] amounts to instantiating the contexts.

Suppose we have a generic development with machines $\mathbf{M}_1, \ldots, \mathbf{M}_n$ building a chain of refinements with carrier sets s and constants c, constrained by axioms $\mathsf{A}(s, c)$. Suppose too that we want to reuse the development within another context, specified by (concrete) carrier sets t and constants d, constrained by axioms $\mathsf{B}(t, d)$. Let $T(t)$, which must be an Event-B type expression, and $E(t, d)$ be the instantiated values for s and c respectively. Given that the instantiation is correct, *i.e.*, $\mathsf{B}(t, d) \Rightarrow \mathsf{A}(T(t), E(t, d))$, the instantiated development where s and c are replaced by their corresponding instantiated values is correct-by-construction.

For more details on instantiation in Event-B and its tool support see [5] and [19]. All instantiation steps described in this paper were performed using the generic instantiation plug-in developed by Hitachi and ETH Zurich [13].

3 Abstract Data Types in Event-B

In this section, we describe how to specify and implement ADTs in Event-B. Our approach is based on refinement and generic instantiation. An ADT is typically defined in terms of a set of operations that can be performed on the ADT, along with a specification of their effect. Let us start with the standard example: the *stack* ADT is a last in first out (LIFO) data type that contains a collection of elements.

A stack is characterised by three operations:

- $push$: takes a stack S and an item e, and returns a new stack where e is added to the top of S.
- pop: takes a (non-empty) stack S and returns a new stack where S's top element is removed.
- top: takes a (non-empty) stack S and returns S's top element.

A special stack is the $empty$ stack that contains no elements. Some important constraints for the operations of the stack ADT are as follows. Given a stack S and an element e, $push(S, e) \neq empty$, $pop(push(S, e)) = S$, and $top(push(S, e)) = e$.

Specifying ADTs in Event-B. ADTs and their operations can be modelled using carrier sets, constants and axioms in Event-B. Instantiation can then be used to "implement" the ADTs. The instantiation proofs ensure that the ADTs' implementations satisfy their specifications.

Each ADT **A** is modelled as follows:

- A carrier set $\mathcal{A}\_TYPE$ defining the type of the **A** objects along with an associated
- set constant $\mathcal{A} \subseteq \mathcal{A}\_TYPE$ representing all valid **A** objects. [3]

[3] Note that we do not currently support the definition of parameterised ADTs, which would allow one to specify a generic *stack* ADT independent of its elements' type.

– Each operation is modelled using a constant.
– The constraints on the operations are specified using axioms.

Consider the stack ADT for elements of type $ELEM$. It can be modelled in Event-B as follows.

$$
\begin{array}{ll}
\textbf{sets}: \quad STACK\_TYPE & \textbf{constants}: \quad STACK, empty, push, pop, top \\
\textbf{axioms}: & \\
\mathsf{axm0\_1}: & STACK \subseteq STACK\_TYPE \\
\mathsf{axm0\_2}: & empty \in STACK \\
\mathsf{axm0\_3}: & push \in STACK \times ELEM \to STACK \\
\mathsf{axm0\_4}: & pop \in STACK \setminus \{empty\} \to STACK \\
\mathsf{axm0\_5}: & top \in STACK \setminus \{empty\} \to ELEM \\
\mathsf{axm0\_6}: & \forall S, e \cdot S \in STACK \;\Rightarrow\; push(S \mapsto e) \neq empty \\
\mathsf{axm0\_7}: & \forall S, e \cdot S \in STACK \;\Rightarrow\; pop(push(S \mapsto e)) = S \\
\mathsf{axm0\_8}: & \forall S, e \cdot S \in STACK \;\Rightarrow\; top(push(S \mapsto e)) = e
\end{array}
$$

Axioms axm0_7 and axm0_8 specify the relationship between the pop, top, and $push$ operations. Notice that there is no need to fully specify an ADT. In subsequent examples, we will only define as many axioms as needed to prove the stated properties.

Instantiating ADTs. A possible implementation of the stack ADT is one where a stack is represented as an array. More formally, a stack is represented by a pair (f, n), where n is the stack's size and f is an array of size n representing its content. In other words, we intend to implement the stack ADT by the array datatype. Operations of the array datatype are as follows:

– $append$: takes an array and an element, and returns a new array where the element is appended to the end of the input array.
– $front$: takes an array and returns a new array where the last element of the input array is removed.
– $last$: takes an array and returns the last element of the input array.

The array datatype is specified in Event-B as follows.

$$
\begin{array}{ll}
\textbf{constants}: & ARRAY, append, front, last \\
\textbf{axioms}: & \\
\mathsf{axm1\_1}: & ARRAY = \{f \mapsto n \mid n \in \mathbb{N} \land f \in 0 \mathinner{..} n-1 \to ELEM\} \\
\mathsf{axm1\_2}: & append = (\lambda\, (f \mapsto n) \mapsto e \cdot f \mapsto n \in ARRAY \land e \in ELEM \\
 & \qquad\qquad\qquad\qquad \mid (f \mathbin{\lhd\!\!-} \{n \mapsto e\}) \mapsto n+1) \\
\mathsf{axm1\_3}: & front = (\lambda\, f \mapsto n \cdot f \mapsto n \in ARRAY \land n \neq 0 \\
 & \qquad\qquad\qquad\qquad \mid ((\{n-1\} \mathbin{⩤} f) \mapsto n-1)) \\
\mathsf{axm1\_4}: & last = (\lambda\, f \mapsto n \cdot f \mapsto n \in ARRAY \land n \neq 0 \mid f(n-1))
\end{array}
$$

Notice that at this point all the constants are concretely defined by lambda expressions.

To prove that the array datatype implements the stack ADT, we instantiate $STACK\_TYPE$ with $\mathbb{P}(\mathbb{Z} \times ELEM) \times \mathbb{Z}$, $STACK$ with $ARRAY$, and the operations $push$, pop, and top

with $append$, $front$, and $last$, respectively. Constant $empty$ is instantiated with $\varnothing \mapsto 0$. We must prove that the instantiated abstract axioms, *i.e.*, axm0_1–axm0_8, are derivable from the concrete axioms, *i.e.*, axm1_1–axm1_4. The proofs can be constructed by expanding the definitions of the concrete constants accordingly.

For more information on how to implement ADTs in Event-B using generic instantiation, we refer the reader to [7].

4 Developing a Train Control System Using ADTs

In this section, we illustrate our approach on an industrial case study. We first briefly describe the system and explain the difficulties when developing such a complex system without ADTs. We then present part of the development where we applied our approach using ADTs. Finally, we evaluate our approach by giving an overview of the entire development of this case study together with some statistics to justify our approach's effectiveness.

4.1 System Description

The scope of our case study is the development of a modern train control system. The main goal of the system is to keep all trains in the railway network a safe distance apart to prevent collisions. The network consists of tracks (divided into sections) and points connecting these tracks. An interlocking system switches the points to connect different tracks with each other, and results in a track layout that dynamically changes. Instead of light signals, the train control system uses radio communication to send the trains the permission to move or stop.

While classic train control systems use trackside hardware to detect whether a section is occupied by a train, our system determines this information from the trains' position and length. The trains themselves determine their positions and send them to the train control system by radio. Based on information on what part of the network is occupied, the controller calculates for every train the area in which it can safely move without collisions. This area is called the *Movement Authority* (MA) and represents the permission for a train to move as long as it does not leave this area. The calculated MAs are then directly sent to the trains where an onboard unit interprets them to calculate the location where the permission to drive ends (*Limit of Authority*, LoA). To prevent driving over the LoA, the onboard unit continuously determines a speed limit and applies the emergency brakes if necessary. An overview of the interacting system components is given in Figure 1.

Collision-freeness between trains is guaranteed by the overall system and relies on two conditions: (1) The trains are always within their assigned movement authorities, and (2) the controller ensures that the MAs issued to the trains do not overlap. In fact, (1) is implementable only if the MAs issued by the controller are never reduced at the front of the trains.

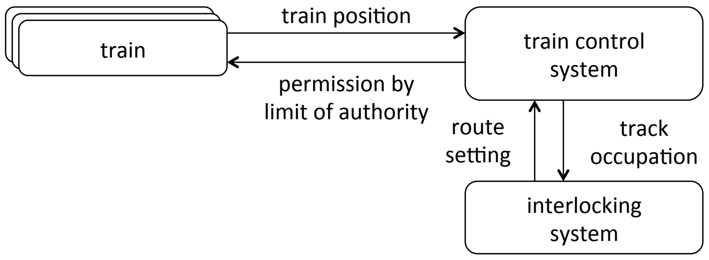

Fig. 1: Train control system with the interlocking system as its environment.

4.2 The Need for Abstraction

Our first challenge in developing the train control system is formalising the trains in the network. Figure 2 depicts a train occupying some part of the network. It illustrates a sequence of sections with fully occupied ones in the middle and partially occupied ones at each end of the train.

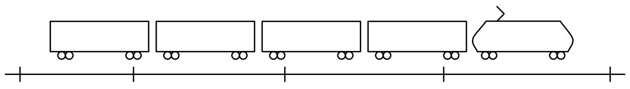

Fig. 2: A train occupying a sequence of sections.

In our first attempt at modelling the train control system, we used different variables to denote how the trains occupy the network. Let ids be the set of active trains in the network. We modelled the different aspects of the trains, such as their head, rear, middle, connections, etc., by total functions as follows. For clarity, we omit from the presentation other aspects of the trains, such as the head- and rear-position within a section.

$$\begin{array}{ll}
\textbf{variables}: & ids, head, rear, middle, connection, \ldots \\
\textbf{invariants}: & \\
\mathsf{inv0\_1}: & head \in ids \rightarrow SECTION \\
\mathsf{inv0\_2}: & rear \in ids \rightarrow SECTION \\
\mathsf{inv0\_3}: & middle \in ids \rightarrow \mathbb{P}(SECTION) \\
\mathsf{inv0\_4}: & connection \in ids \rightarrow (SECTION \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\;\;\;\;\rightarrow SECTION) \\
\mathsf{inv0\_5}: & \forall t \cdot t \in ids \;\Rightarrow\; head(t) \notin middle(t) \\
\mathsf{inv0\_6}: & \forall t \cdot t \in ids \;\Rightarrow\; rear(t) \notin middle(t) \\
\mathsf{inv0\_7}: & \forall t \cdot t \in ids \land connection(t) = \varnothing \;\Rightarrow\; head(t) = rear(t) \\
\ldots &
\end{array}$$

Invariants inv0_5–inv0_7 specify several important properties of trains. For example, inv0_7 specifies that if a train occupies only one single section, its head and rear are in the same section. Note that due to the lack of space, we omit other invariants that ensure that trains are connected and do not contain loops.

To motivate the need for additional abstraction in Event-B, we focus on the event train_extend. Its purpose is to extend the train, denoted by t, to a section, denoted by s. Namely, train_extend prepends s to the head of the train and s becomes the new head. This event is used whenever the train reaches the end of the current head section and moves to the beginning of the next section in front of it.

```
train_extend :
  any   $t, s$   where
    $t \in ids$
    $s \notin \mathrm{dom}(connection(t))$
    $head(t) \notin \mathrm{ran}(connection)$
  then
    $head(t) := s$
    $middle(t) := (middle(t) \cup \{head(t)\}) \setminus \{rear(t)\}$
    $connection(t) := connection(t) \cup \{s \mapsto head(t)\}$
  end
```

The event's guard ensures that the connection of t remains a partial injective function (inv0_4). When updating $middle(t)$, we remove $rear(t)$ to guarantee that in the case where the train occupies only one section (*i.e.*, $connection(t) = \varnothing$ and hence $head(t) = rear(t)$, according to inv0_7), the train's middle is still empty afterwards.

Proving that train_extend maintains the invariants, in particular inv0_5, requires more invariants, which we omit for clarity. All additional invariants are universally quantified, *i.e.*, of the form "$\forall t \cdot t \in ids \Rightarrow \ldots$" and they express the relationship between different aspects of a train.

Encapsulation. The invariants above describe the trains' layouts that change independently of each other. As a result, the preservation of the invariants should be proven on a per train basis and by hiding the rest of the model. In Event-B, however, invariants are global and all other parts of the system are taken into account during the proof, which increases their complexity. This indicates that some encapsulation for the models of trains will be useful for our proofs.

High-level Properties of Low-level Details. An attempt to specify and prove properties such as collision-freeness at a concrete level like that described above leads to complicated models and difficult proofs. In particular, expressing relationships between sequences, such as "containment" (*e.g.*, a train is always within its movement authority) and "being disjoint" (*e.g.*, the movement authorities of two different trains do not overlap) using information about the sequences' head, rear, middle and connections, is far from trivial. This indicates that we should start modelling the system at an even more abstract level by omitting the detailed aspects of the sequences.

Reuse. In addition to the above mentioned difficulties, another motivation for using ADTs in our development is that modelling the trains' movement authorities is similar to modelling the trains. In fact, both trains and their MAs should be modelled using the same ADT.

4.3 Development using Abstract Data Types

The Region ADT. Abstracting away the details of sequences, such as head, rear, middle, and connections, we start our modelling with an ADT corresponding to *regions* on a network, focusing on relationships between regions such as "contained" and "disjoint". The region ADT includes the following operations:

- extend: takes a region R and a section s, and returns a new region where s is added to R.
- $\mathit{contained}$: binary relation associating a region R_1 with every region R_2 that contains R_1.
- $\mathit{disjoint}$: binary relation associating two regions R_1 and R_2 with each other if they do not overlap.

Note that there are other operations of the region ADT that we omit for clarity.

In Event-B, this ADT is modelled as follows. Constants $\mathit{contained}$, $\mathit{disjoint}$, and extend correspond to the operations mentioned above.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{sets}: \quad \mathit{REGION\_TYPE}} \\
\multicolumn{2}{l}{\textbf{constants}: \quad \mathit{REGION}, \mathit{contained}, \mathit{disjoint}, \mathit{extend}} \\
\multicolumn{2}{l}{\textbf{axioms}:} \\
\mathsf{axm0\_1}: & \mathit{REGION} \subseteq \mathit{REGION\_TYPE} \\
\mathsf{axm0\_2}: & \mathit{contained} \in \mathit{REGION} \leftrightarrow \mathit{REGION} \\
\mathsf{axm0\_3}: & \mathit{disjoint} \in \mathit{REGION} \leftrightarrow \mathit{REGION} \\
\mathsf{axm0\_4}: & \mathit{extend} \in \mathit{REGION} \times \mathit{SECTION} \mathrel{+\!\!\!\!\to} \mathit{REGION}
\end{array}
$$

Constraints on the operations of the region ADT are modelled as axioms. For example, $\mathit{contained}$ is transitive, $\mathit{disjoint}$ is symmetric, extend is strengthening with respect to $\mathit{contained}$. Note that in the following, we use $R_1 \Subset R_2$ to denote $R_1 \mapsto R_2 \in \mathit{contained}$, and $R_1 \not\Cap R_2$ to denote $R_1 \mapsto R_2 \in \mathit{disjoint}$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{axioms}:} \\
\mathsf{axm0\_5}: & \forall t_1, t_2, t_3 \cdot t_1 \Subset t_2 \land t_2 \Subset t_3 \ \Rightarrow\ t_1 \Subset t_3 \\
\mathsf{axm0\_6}: & \forall t_1, t_2 \cdot t_1 \not\Cap t_2 \ \Rightarrow\ t_2 \not\Cap t_1 \\
\mathsf{axm0\_7}: & \forall t, s \cdot t \mapsto s \in \mathrm{dom}(\mathit{extend}) \Rightarrow t \Subset \mathit{extend}(t \mapsto s) \\
\mathsf{axm0\_8}: & \forall t_1, t_2, t_3 \cdot t_1 \Subset t_2 \land t_2 \not\Cap t_3 \ \Rightarrow\ t_1 \not\Cap t_3
\end{array}
$$

The current states of the active trains and their associated movement authorities are represented by a mapping from trains to the set of all possible regions (train) and a mapping from movement authorities to the set of all possible regions (ma). Invariant inv0_3 states that the trains always stay within their movement authorities. Invariant inv0_4 states that the movement authorities of any two trains are disjoint.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{variables}: \quad \mathit{ids}, \mathit{train}, \mathit{ma}} \\
\multicolumn{2}{l}{\textbf{invariants}:} \\
\mathsf{inv0\_1}: & \mathit{train} \in \mathit{ids} \to \mathit{REGION} \\
\mathsf{inv0\_2}: & \mathit{ma} \in \mathit{ids} \to \mathit{REGION} \\
\mathsf{inv0\_3}: & \forall t \cdot t \in \mathit{ids} \ \Rightarrow\ \mathit{train}(t) \Subset \mathit{ma}(t) \\
\mathsf{inv0\_4}: & \forall t_1, t_2 \cdot t_1 \in \mathit{ids} \land t_2 \in \mathit{ids} \land t_1 \neq t_2 \ \Rightarrow\ \mathit{ma}(t_1) \not\Cap \mathit{ma}(t_2)
\end{array}
$$

Importantly, the collision-freeness property, *i.e.*,

$$\forall t_1, t_2 \cdot t_1 \in ids \land t_2 \in ids \land t_1 \neq t_2 \;\Rightarrow\; train(t_1) \not\pitchfork train(t_2)\ ,$$

is derivable (as a theorem) from the invariants inv0_3, inv0_4 and the property relating *contained* and *disjoint*, *i.e.*, axm0_8.

The event train_extend can be specified abstractly as follows. Its last guard ensures that the extended train cannot exceed its assigned movement authority.

```
train_extend :
  any  $t, s$  where
    $t \in \mathrm{dom}(train)$
    $train(t) \mapsto s \in \mathrm{dom}(extend)$
    $extend(train(t) \mapsto s) \Subset ma(t)$
  then
    $train(t) := extend(train(t) \mapsto s)$
  end
```

The Sequence ADT. The model at this stage is abstract in two ways: (1) its dynamic behaviour is not fully described by the machine and (2) it uses the region ADT which is not fully "implemented". For (2), we utilise generic instantiation to introduce more details on how the region ADT and its operations are realised. Similar to refinement, this realisation of ADTs can be split into multiple instantiation steps.

In our development, we first replace the region ADT by the sequence ADT. The sequence ADT includes the following operations:

- *prepend*: takes a sequence S and a section s, and returns a new sequence where s is added to the head of S.
- *head*: takes a sequence S and returns the head section of S.
- *rear*: takes a sequence S and returns the rear section of S.
- *middle*: takes a sequence S and returns the middle sections of S.
- *connection*: takes a sequence S and returns the connection between sections of S.

sets : $SEQUENCE\_TYPE$
constants : $SEQUENCE, prepend, head, rear, middle, connection$
axioms :
axm1_1 : $SEQUENCE \subseteq SEQUENCE\_TYPE$
axm1_2 : $prepend \in SEQUENCE \times SECTION \pto SEQUENCE$
axm1_3 : $head \in SEQUENCE \to SECTION$
axm1_4 : $rear \in SEQUENCE \to SECTION$
axm1_5 : $middle \in SEQUENCE \to \mathbb{P}(SECTION)$
axm1_6 : $connection \in SEQUENCE \to (SECTION \pinj SECTION)$
axm1_7 : $\forall S \cdot S \in SEQUENCE \;\Rightarrow\; head(S) \notin middle(S)$
axm1_8 : $\forall S \cdot S \in SEQUENCE \;\Rightarrow\; rear(S) \notin middle(S)$

We prove that the sequence ADT is a valid representation of the region ADT with the instantiation of the set $REGION\_TYPE$ by $SEQUENCE\_TYPE$, the constants

REGION by *SEQUENCE*, *extend* by *prepend*, etc. We replace (instantiate) the operations *contained* and *disjoint* using *head*, *rear*, *middle*, and *connection*. For example, *contained* is instantiated as follows.

$$contained = \left\{ S_1 \mapsto S_2 \mid \begin{array}{l} S_1 \in SEQUENCE \land S_2 \in SEQUENCE \land \\ connection(S_1) \subseteq connection(S_2) \land \\ middle(S_1) \subseteq middle(S_2) \land \\ head(S_1) \in \{head(S_2)\} \cup middle(S_2) \cup \{rear(S_2)\} \\ rear(S_1) \in \{head(S_2)\} \cup middle(S_2) \cup \{rear(S_2)\} \\ \ldots \end{array} \right\}$$

Note that we omit from our presentation additional conditions related to the exact position of the head and rear within the section.

At this point the sequence ADT is still abstract. In particular, we do not give the exact definition for sequences and we still rely on the operators such as *head*, *rear*, *middle*, and *connection* and the relationships between them.

Given the instantiation, we subsequently refine the dynamic behaviour of the system (*i.e.*, the machines). For event train_extend, the refinement removes the reference to *contained* in the guard.

```
train_extend :
  any   $t, s$   where
    $t \in \mathrm{dom}(train)$
    $head(train(t)) \neq head(ma(t))$
    ... // other guards related to head/rear positions
  then
    $train(t) := prepend(train(t) \mapsto s)$
  end
```

The Arbitrarily-based Array Data Type. The model based on the sequence ADT is abstract. To ensure that the model is implementable, we must give a representation for the sequence ADT. In our development, we use an arbitrarily-based array data type as the implementation for the sequence ADT. An arbitrarily-based array is an array that starts from an arbitrary index, in contrast to the common zero-based array that always starts from 0. More formally, each arbitrarily-based array can be represented by a tuple (a, b, f), where a and b are the starting and ending indices and f represents the array's content. The operations of the arbitrarily-based array such as *head*, *rear*, *middle*, and *connection* are defined accordingly. For example, the *head* operation is defined as follows.

$$head = (\lambda\ a \mapsto b \mapsto f \cdot a \mapsto b \mapsto f \in ARRAY \mid f(a))$$

The advantage of using arbitrarily-based arrays compared to normal (zero-based) arrays is that there is no need to shift indices when extending or reducing the arrays. For example, the *prepend* operation is defined as follows.

$$\begin{array}{l} prepend = (\lambda\ (a \mapsto b \mapsto f) \mapsto s \cdot a \mapsto b \mapsto f \in ARRAY\ \land\ s \in SECTION \land \ldots \\ \qquad\qquad\qquad\qquad \mid\ (a-1) \mapsto b \mapsto (f \mathbin{\lhd\!\!\!-} \{a-1 \mapsto s\})) \end{array}$$

This simplifies the proof that the sequence ADT is correctly implemented by the array data type.

4.4 Development Summary

In our development of the train control system, the transformation of the region ADT into the sequence ADT is carried out in several instantiation steps. The benefit of having steps with small changes in the ADTs is that the machines that are specified using ADTs can also be gradually transformed in small steps. This also serves to decompose the proof of correctness of the systems into small instantiation and refinement steps.

Our development contains five different stages (numbered 0–4), connected by instantiation relationships, where a subsequent stage starts as an instantiation of the previous stage. Each stage contains several refinement steps for developing the system's main functionality.

Stage 0: We formalise the system at the most abstract, generic level, using the region ADT and the network ADT. In the refinement steps, we gradually introduce the active network, the active trains, the trains' movement authorities, the movement authorities calculated by the controller, and the relationships between them.

Stage 1–3: We carry out the transformation from the region ADT to the sequence ADT in three different instantiations. First, we instantiate the *contained* operation. Second, we instantiate the operation "part-of" between the region ADT and the network ADT (stating whether or not a region is *part of* a network). Finally, we instantiate the *disjoint* operation. The refinement steps in these stages have two purposes: (1) they transform the events to use the new data types, and (2) they introduce the design details of the system, including notions like *train ahead*, *train behind*, and *last train within a section*.

Stage 4: We instantiate the sequence ADT by the arbitrarily-based array data type. We also incrementally introduce details on the calculation of the trains' MAs.

Statistics and Comparison. We present statistics for our development in Table 1 and compare the development of the train control system with and without ADTs. Table 1a shows the proof statistics for our first attempt where we did not use ADTs. After 14 refinement steps and 45 difficult manual proofs, we stopped our development with numerous remaining undischarged proof obligations, due to missing invariants. We would have needed additional invariants that are complex to express and lead to even more complex proofs. Considering the proof effort needed up to this point, and the additional effort anticipated to complete the development, we were forced to adapt our development strategy and find additional abstraction techniques to simplify the proofs.

Table 1b shows the proof statistics of the development using ADTs. We distinguish between proofs related to instantiation and proofs related to refinement. Overall, 14% of the proofs are related to instantiation, and the other 86% are related to refinement. As expected, the machines at the more abstract and generic levels are more automated. Most of the manual proofs originating from instantiation (in particular of Stage 4) have a similar structure that includes manually expanding the instantiation definitions. These proof steps could be automated with a dedicated proof strategy, which would increase the amount of proof automation. Overall, the instantiation proofs have a better automation rate (82%) compared to the refinement proofs (58%).

The number of refinement steps as well as the total number of discharged proof obligations indicate that the size and complexity of our case study is significantly higher than typical academic examples. Moreover, given the level of detail in our model, stemming from realistic requirements, this supports our claims about the relevance of our approach for large and complex systems.

	Obligations	Auto.	Manual	Undischarged
14 Refinements	666	497 (75%)	45 (7%)	124 (18%)

(a) Development without ADTs

		Obligations	Auto.	Manual
Stage 0	8 Refinements	267	267	0
Stage 1	Instantiation	34	24	10
	14 Refinements	632	477	155
Stage 2	Instantiation	165	161	4
	1 Refinement	52	44	8
Stage 3	Instantiation	175	172	3
	16 Refinements	765	314	451
Stage 4	Instantiation	174	90	84
	18 Refinements	1748	891	857
Total		4012	2440 (61%)	1572 (39%)
	Instantiation	548 (14%)	447 (82%)	101 (18%)
	Refinement	3464 (86%)	1993 (58%)	1471 (42%)

(b) Development using ADTs

Table 1: Statistics

5 Related Work

5.1 Instantiation and Data Types

In our approach to introducing ADTs into formal development, generic instantiation [5] is the key technique for realising the ADTs. This differs from [19] where instantiation provides a means to reuse formal models in combination with a composition technique. In particular, to guarantee the correctness of the instantiated model, carrier sets (which are assumed to be non-empty and maximal) must be instantiated by type expressions. This has been overlooked in [5] and [19].

Part of our approach was previously published in [7]. There, our main motivation for using ADTs was to encapsulate data and to split the development process into two parts that can be handled by a domain expert and a formal methods expert, respectively. In this paper, we focus more on the need for alternative forms of abstraction when developing large and complex systems in Event-B. We not only use ADTs to abstract away implementation details for the domain expert, but we use them as an integral part from the beginning of our development to simplify the proofs. We describe relations between different ADTs to abstractly specify the system's properties.

The development of the *Theory Plug-in* [11] for Rodin allows users to extend the mathematical language of Event-B, for example, by including new data types. Theorems about new data types can be stated and later used by a dedicated tactic associated with the Theory Plug-in. There is also a clear distinction between the theory modules (capturing data structures and their properties) and the Event-B models using the newly defined data structures. The main difference between the Theory Plug-in and our approach is that the data types in the Theory Plug-in are "concrete". One must give the definitions for the data types and prove theorems about them before using these data types for modelling. This bottom-up approach is in contrast with our top-down approach where the choice of implementations for ADTs can be delayed. More specifically, we can have different implementations for the ADTs. For example, instead of implementing the sequence ADT using arbitrarily-based arrays, we can use standard, zero-based arrays for the same purpose. In fact, we did experiment with both implementations and decided to use arbitrarily-based arrays due to the simpler proofs for the systems.

Our approach of using ADTs in Event-B is similar to work on algebraic specifications [18]. In this domain, a specification contains a collection of *sorts*, *operations*, and *axioms* constraining the operations. Specifications can be *enriched* by additional sorts, operations, or axioms. Furthermore, to develop programs from specifications, the specifications are transformed via a sequence of small "refinement" steps. During these steps, the operations are "coded" until the specification becomes a concrete description of a program. For each such refinement step, one must prove that the code of the operations satisfies the axioms constraining them. An algebraic specification therefore corresponds to an Event-B context, while refinement in algebraic specifications is similar to generic instantiation in Event-B. In contrast to algebraic specifications [12], where the entire functionality of a system is modelled as ADTs (in the form of many-sorted algebras) [18], we use ADTs to abstract only part of our system's functionality. Modelling every aspect of a complex system like our example as an algebraic specification would be very challenging. In addition to the data types, the transition systems must also be encoded as ADTs in the specification. This would require a large number of axioms to describe the transitions.

5.2 Formal Development of Railway Systems

Bjørner gives in [9] a comprehensive overview of formal techniques and tools used for developing software for transportation systems. Beside techniques like model checking and model-based test case generation, he mentions approaches using refinement. The following approaches are of special interest for us.

The development of Metro line 14 in Paris [8,2] is one of the better known industrial application of formal methods. In particular, the safety critical part of the software was developed using the (classical) B Method [1]. The formal reasoning there was only at the software-level, *i.e.*, reasoning about the correctness of the software in isolation. In contrast, in our work we not only model the train control system, but also its environment such as the trains and their movement behaviour. Hence, we can reason on the *system-level* covering the overall structure of the system, its components, and their relationship [4].

In [14], Haxthausen and Peleska present the formal development and verification of a distributed railway control system using the RAISE formal method. Their approach is similar to our work as they also use stepwise refinement and ADTs to cope with the complexity of their system. However, their system is overly simplified at some points which reduced the development challenges that we found to be the most difficult in our work. First, they only consider simple network topologies without loops. Second, they develop a system where sections are either fully occupied or free. Third, their trains can occupy at most two sections. Although they claim that the system can be easily adapted for trains occupying more than two sections, from our experience, this generalisation is a challenging task. Moreover, in their proof they require that if any two events are enabled in a valid state, executing one of the events and therefore changing the state cannot disable the other event's guard. This is a strong property that is cumbersome to verify as one must prove it for all pairs of events. Our model does not require this property in order to guarantee the system's safety.

Platzer and Quesel verify parts of a similar train control system in [17] using their own verification tool KeYmaera. While we developed the functionality of the controller, their work focuses on developing the onboard unit. In their development, the controller belongs to the environment of the onboard unit and they assume that the controller does not issue MAs that are physically impossible for the trains. Our development fulfils this assumption by guaranteeing that the MAs are never reduced.

6 Conclusion

In this paper we presented an approach to building formal models in Event-B using ADTs. ADTs allow us to hide irrelevant details that are unimportant for proving abstract properties. On an abstract level, one can therefore focus on modelling the system's core functionality.

The way we introduce ADTs in our approach allows us to utilise generic instantiation. This handles both the instantiation of an ADT by the chosen data structure as well as the generation of the required proof obligations to guarantee that the chosen structure is a valid instance of the ADT. As a large scale case study we have successfully applied our approach to the development of a realistic train control system. We identified the limitations of only using refinement for this system and showed how we overcome these limitations using ADTs.

As future work we would like to overcome some of the current limitations of our work. As previously mentioned, we cannot presently specify parameterised ADTs. To overcome this limitation, we need to extend the semantics of Event-B contexts and adapt the generic instantiation technique accordingly. The Theory Plug-in might be useful to specify parameterised ADTs.

References

1. J-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. J-R. Abrial. Formal Methods in Industry: Achievements, Problems, Future. In L.J. Osterweil, H.D. Rombach, and M.L. Soffa, editors, *ICSE*, pages 761–768. ACM, 2006.
3. J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
4. J-R. Abrial. From Z to B and then Event-B: Assigning Proofs to Meaningful Programs. In E.B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.
5. J-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
6. European Railway Agency. *ERTMS/ETCS Functional Requirements Specification*. European Railway Agency, Valencinnes, France, 2007.
7. D. Basin, A. Fürst, T.S. Hoang, K. Miyazaki, and N. Sato. Abstract Data Types in Event-B - An Application of Generic Instantiation. *CoRR*, 2012.
8. P. Behm, P. Benoit, A. Faivre, and J-M. Meynadier. Météor: A Successful Application of B in a Large Project. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.
9. D. Bjørner. New Results and Trends in Formal Techniques & Tools for the Development of Software for Transportation Systems. In *FORMS*, 2003.
10. K. Breitman and A. Cavalcanti, editors. *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*. Springer, 2009.
11. M. Butler and I. Maamria. Practical theory extension in Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013.
12. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
13. A. Fürst, K. Desai, T.S. Hoang, and N. Sato. Generic Instantiation Plug-in. `http://sourceforge.net/projects/gen-inst/`.
14. A.E. Haxthausen and J. Peleska. Formal Development and Verification of a Distributed Railway Control System. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1546–1563. Springer, 1999.
15. IEEE Std 1474.1-2004. *IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements*. IEEE, New York, USA, 2005.
16. 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.
17. A. Platzer and J-D. Quesel. European Train Control System: A Case Study in Formal Verification. In Breitman and Cavalcanti [10], pages 246–265.
18. D. Sannella and A. Tarlecki. Essential Concepts of Algebraic Specification and Program Development. *Formal Asp. Comput.*, 9(3):229–269, 1997.
19. R. Silva and M. Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In Breitman and Cavalcanti [10], pages 466–484.