The original publication is available at http://dx.doi.org/10.1016/j.scico.2016.04.010
In Science of Computer Programming, online since May 2016
© Elsevier

Large-scale System Development Using Abstract Data Types and Refinement

Andreas Fürst[a], Thai Son Hoang[b,1,*], David Basin[a], Naoto Sato[c], Kunihiko Miyazaki[c]

[a] *Institute of Information Security, ETH Zurich, Switzerland*
[b] *ECS, University of Southampton, U.K.*
[c] *Yokohama Research Laboratory, Hitachi Ltd., Japan*

Abstract

We present a formal modelling approach using Abstract Data Types (ADTs) for large-scale system development 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 just their necessary properties, and we postpone the introduction of their concrete definitions to later development steps. As the ADTs are incrementally instantiated and become more concrete, behavioural details of systems are expanded via refinement in a manner consistent with the ADTs' transformation. We evaluate this 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: instantiation, refinement, Event-B, 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 *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 the abstraction of machines, and since abstract machines contain fewer details than concrete ones, they are usually easier to verify.

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

*Corresponding authors
[1] The first version of this paper was written when Thai Son Hoang was a visiting researcher at YRL, Hitachi Ltd., Japan.

the system's collision-freeness and non-derailment properties, we needed to provide detailed models, for example formalising the network on which the trains operate and the trains' position 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 model to reduce the complexity and increase the automation of the resulting proofs.

Approach. To increase system abstraction and reuse during modelling, we introduce *Abstract Data Types* (ADTs) [2] in Event-B. 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 contexts to formalise ADTs, we can subsequently utilise the ADTs to model the system's dynamic behaviour in the machines. We use generic instantiation [3] to further concretise and thereby implement the ADTs. As the ADTs evolve, the machines are also refined accordingly.

We evaluate our approach by using it in a pilot project to formalise a part of a CBTC software system developed by Hitachi Ltd., and measuring how the use of ADTs reduces the size and increases the automation of the developments. 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 [4] or ETCS Level 3 [5]. 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 to introduce ADTs into a refinement formalism and show how they can be used to manage development complexity. In this paper, we use Event-B [1] to illustrate our approach, but other refinement formalisms such as ASMs [6], Action Systems [7], or TLA [8] could be used instead. We show that reasoning using ADTs can be done purely based on the properties of the ADTs' operations, regardless of how they are 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, while the machine merely "calls" the ADTs' operations. 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 is novel as this technique has so far only been applied to reuse developments, for example in [9]. Using generic instantiation, we effectively perform data refinement for the ADTs within contexts. In contrast, we use generic instantiation to gradually introduce details into the formal models similar to refinement.

The way we use ADTs allows them to be used alongside 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 more flexible and thereby appears better suited for developing large-scale industrial systems. More details on related work can be found in Section 5.

Structure. The rest of this 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 it 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

We use Event-B [1] for our developments. Event-B represents a further evolution of the classical-B method [10], which has been simplified and focused around the 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 [1, 11]. Instead, we will describe those modelling concepts that are important for our subsequent presentation.

Event-B models are organised in terms of two 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 behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, and *events*. Variables v define the state of a machine.[2] 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

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

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 event's action is composed of one or more *assignments* of the form $x := E(t, v)$, where x is a variable in v. Assignments 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 [1]. A machine **CM** can refine another machine **AM**. We call **AM** the *abstract* machine and **CM** the *concrete* machine. The abstract machine's states are related to the concrete machine's states by a *gluing invariant* $J(v, w)$, 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)

[2] When referring to variables v, parameters t, carrier sets s, or constants c, we usually allow for multiple variables, parameters, carrier sets, or constants, and we abuse the notation and treat them as sets of the corresponding objects.

is when v is kept in the refinement, *i.e.*, $v \subseteq w$. Intuitively, any behaviour of **CM** can be simulated by a behaviour of **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 invariant $J(v, w)$ establish a simulation of f by e (*simulation*).

Instantiation. Instantiation is a technique for reusing models by providing values for their parameters. Since an Event-B model is parameterised by the carrier sets and constants, instantiation in Event-B [3, 9] 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 [3] and [9]. All instantiation steps described in this paper were performed using the generic instantiation plug-in [12] for the Rodin platform [13]. The plug-in is jointly developed by Hitachi and ETH Zurich. An example of generic instantiation can be found in Section 3.2 for implementing ADTs.

3. Abstract Data Types in Event-B

We now 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 the operations' effect. Let us start with the standard example: a *stack* is a last-in first-out data type that contains a collection of elements and 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. Three important constraints on the stack operations are: Given a stack S and an element e, $push(S, e) \neq empty$, $pop(push(S, e)) = S$, and $top(push(S, e)) = e$.

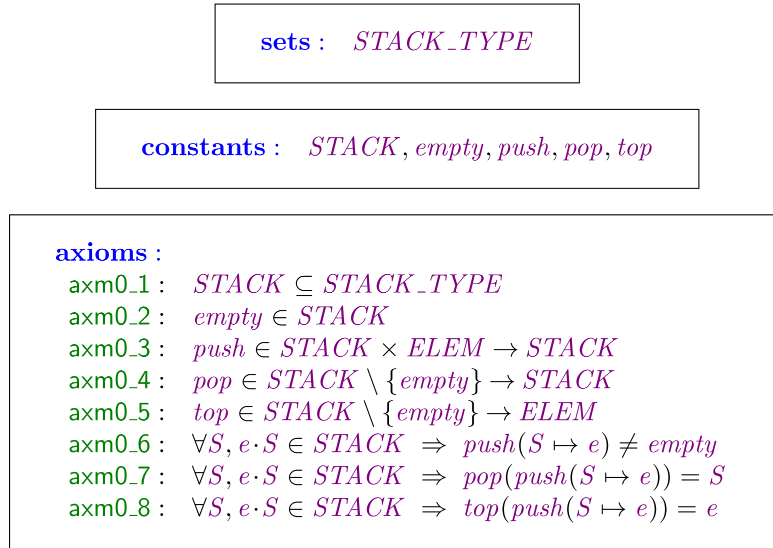
ADTs and their operations can be modelled using carrier sets, constants, and axioms in Event-B. Instantiation can be used to "implement" an ADT using other ADTs. The instantiation proofs ensure that the implementations satisfy their specification.

3.1. Specifying ADTs in Event-B

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

- A carrier set $\mathcal{A}_TYPE$ defines the type of the **A** objects.
- An associated set constant $\mathcal{A} \subseteq \mathcal{A}_TYPE$ represents all valid **A** objects.[3]
- Each operation is modelled as 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.



sets : $STACK_TYPE$

constants : $STACK, empty, push, pop, top$

axioms :

axm0_1 : $STACK \subseteq STACK_TYPE$

axm0_2 : $empty \in STACK$

axm0_3 : $push \in STACK \times ELEM \rightarrow STACK$

axm0_4 : $pop \in STACK \setminus \{empty\} \rightarrow STACK$

axm0_5 : $top \in STACK \setminus \{empty\} \rightarrow ELEM$

axm0_6 : $\forall S, e \cdot S \in STACK \;\Rightarrow\; push(S \mapsto e) \neq empty$

axm0_7 : $\forall S, e \cdot S \in STACK \;\Rightarrow\; pop(push(S \mapsto e)) = S$

axm0_8 : $\forall S, e \cdot S \in STACK \;\Rightarrow\; top(push(S \mapsto e)) = e$

In the above, the notation $S \times T$ corresponds to the *Cartesian product* of S and T, $f \in S \rightarrow T$ specifies that f is a *total function* from S to T, and $a \mapsto b$ denotes the *ordered pair* (a, b). The axioms axm0_6–axm0_8 specify the relationship between the pop, top, and $push$ operations and the constant stack $empty$. Note that there is no need to fully specify an ADT. For example, additional operations and axioms are required to support inductive reasoning about stacks. In subsequent examples, we define as many axioms as are needed to prove the stated properties.

3.2. Implementing ADTs by Instantiation

A possible implementation of the stack ADT is one where a stack is represented as an array, where arrays are defined by an *array* ADT. Specifically, 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. Operations of the array ADT are as follows:

[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.

- *append*: takes an array A and an element e and returns a new array where e is appended to the end of A.
- *front*: takes an array A and returns a new array where A's last element is removed.
- *last*: takes an array A and returns A's last element.

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

constants : $ARRAY, append, front, last$

axioms :

axm1_1 : $ARRAY = \{f \mapsto n \mid n \in \mathbb{N} \land f \in 0\,..\,n-1 \rightarrow ELEM\}$

axm1_2 : $append = (\lambda\ (f \mapsto n) \mapsto e \cdot f \mapsto n \in ARRAY \land e \in ELEM \mid (f \mathbin{⩤\!\!\!\!+} \{n \mapsto e\}) \mapsto n+1)$

axm1_3 : $front = (\lambda\ f \mapsto n \cdot f \mapsto n \in ARRAY \land n \neq 0 \mid ((\{n-1\} \mathbin{⩤} f) \mapsto n-1))$

axm1_4 : $last = (\lambda\ f \mapsto n \cdot f \mapsto n \in ARRAY \land n \neq 0 \mid f(n-1))$

In the above, $f \mathbin{⩤\!\!\!\!+} \{x \mapsto E\}$ denotes the function identical to f except the entry at x is assigned E, and $\{x\} \mathbin{⩤} f$ denotes the function where the entry for x is removed from f. Notice that at this point all the constants are concretely defined using lambda abstractions. Note that, $(\lambda X \cdot P \mid E)$ denotes a lambda abstraction where the pattern expression X specifies the domain of the function. More specifically, $(\lambda X \cdot P \mid E)$ is defined to be the set comprehension $\{x \cdot P \mid X \mapsto E\}$, where x is the list of variables that appears in X.

To prove that the array datatype implements the stack ADT, we instantiate the stack ADT as follows.

$$\begin{aligned}
STACK_TYPE &\longleftarrow \mathbb{P}(\mathbb{Z} \times ELEM) \times \mathbb{Z} \\
STACK &\longleftarrow ARRAY \\
push &\longleftarrow append \\
pop &\longleftarrow front \\
top &\longleftarrow last \\
empty &\longleftarrow \varnothing \mapsto 0
\end{aligned}$$

We must prove that the instantiated abstract axioms axm0_1–axm0_8 are derivable from the concrete axioms axm1_1–axm1_4. The following theorems *thm0_1*–*thm0_8* are the instantiated versions of the corresponding abstract axioms, where the carrier sets and constants are syntactically replaced according to the specified instantiation.

$$
\begin{array}{ll}
\mathit{thm0_1}: & \mathit{ARRAY} \subseteq \mathbb{P}(\mathbb{Z} \times \mathit{ELEM}) \times \mathbb{Z} \\
\mathit{thm0_2}: & \varnothing \mapsto 0 \in \mathit{ARRAY} \\
\mathit{thm0_3}: & \mathit{append} \in \mathit{ARRAY} \times \mathit{ELEM} \to \mathit{ARRAY} \\
\mathit{thm0_4}: & \mathit{front} \in \mathit{ARRAY} \setminus \{\varnothing \mapsto 0\} \to \mathit{ARRAY} \\
\mathit{thm0_5}: & \mathit{last} \in \mathit{ARRAY} \setminus \{\varnothing \mapsto 0\} \to \mathit{ELEM} \\
\mathit{thm0_6}: & \forall S, e \cdot S \in \mathit{ARRAY} \;\Rightarrow\; \mathit{append}(S \mapsto e) \neq \varnothing \mapsto 0 \\
\mathit{thm0_7}: & \forall S, e \cdot S \in \mathit{ARRAY} \;\Rightarrow\; \mathit{front}(\mathit{append}(S \mapsto e)) = S \\
\mathit{thm0_8}: & \forall S, e \cdot S \in \mathit{ARRAY} \;\Rightarrow\; \mathit{last}(\mathit{append}(S \mapsto e)) = e
\end{array}
$$

The proofs that *thm0_1*–*thm0_8* are derivable from axm1_1–axm1_4 can be constructed by expanding the definitions of the concrete constants, *i.e.*, *ARRAY*, *append*, *front*, and *last*, accordingly. For instance, the proof of *thm0_2* gives rise the following sub-goals:

$$
\begin{array}{l}
0 \in \mathbb{N}, \text{ and} \\
\varnothing \in 0 \,..\, -1 \to \mathit{ELEM} \;,
\end{array}
$$

which are trivial to prove. The proof of *thm0_6* eventually requires proving that $n+1 \neq 0$ for any natural number n, which is again trivial. The Rodin platform archive of the development can be found online.[4]

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 in developing such a complex system without ADTs. We then present part of the development where we used ADTs. Finally, we evaluate our approach by giving an overview of the entire development together with statistics that document our approach's effectiveness.

4.1. System Description

In our case study, we develop part of a modern train control system. The system is intended to keep all trains in a railway network at a safe distance apart to prevent collisions. The network consists of tracks divided into sections, and of points connecting these tracks. An interlocking system switches the points to connect different tracks together, and results in a dynamically changing track layout. Instead of light signals, the train control system uses radio communication to send the trains the permission to move or stop.

Many train control systems use trackside hardware to detect whether a section is occupied by a train. By contrast, our system determines this information from the trains' position and length. An overview of the interacting system components is given in Figure 1. The trains themselves determine their positions and send them to the train control system by radio. Based on information about which parts of the network are occupied, the controller calculates for each train the area in which it can safely move without collisions. This area is called the *Movement Authority* (MA) and represents the

[4]URL: `http://sourceforge.net/projects/gen-inst/files/Examples/`

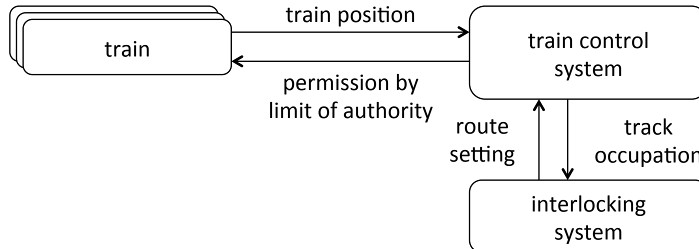


Figure 1: Train control system with the interlocking system as its environment

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, called the *Limit of Authority* (LoA). To prevent driving over the LoA, the onboard unit continuously determines a speed limit and applies the train's emergency brakes if necessary.

The most important properties of the train system are collision-freeness and non-derailment.

REQ 1	There must be no collision between any two (different) trains in the system.

REQ 2	Every train in the system must stay on the tracks of the network.

Collision-freeness between trains, *i.e.*, (REQ 1), is guaranteed by the overall system and relies on two conditions: (C1) The trains are always within their assigned movement authorities, and (C2) the controller ensures that the MAs issued to the trains do not overlap. In fact, (C1) is implementable only if the MAs issued by the controller are never reduced at the front of the trains. Non-derailment, *i.e.*, (REQ 2), is ensured by condition (C1) mentioned before, and condition (C3) stating that the controller only grants MAs over the active network.

4.2. The Need for Abstraction

In this section, we review our initial failed attempt to model the train control system using Event-B refinement without ADTs. This motivates the development of our approach using ADTs, described in Section 4.3. Our first challenge in developing the train control system is formalising the trains in the network. Figure 2 depicts a train occupying a part of the network. It illustrates a sequence of sections with fully occupied sections in the middle and partially occupied sections at each end of the train.

In our first attempt at modelling this system, we used different variables to denote how 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,

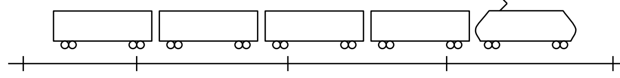


Figure 2: A train occupying a sequence of sections

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.

variables : $ids, head, rear, middle, connection, \ldots$

invariants :

inv0_1 : $head \in ids \rightarrow SECTION$
inv0_2 : $rear \in ids \rightarrow SECTION$
inv0_3 : $middle \in ids \rightarrow \mathbb{P}(SECTION)$
inv0_4 : $connection \in ids \rightarrow (SECTION \rightarrowtail\!\!\!\!\!\!\!+\;\; SECTION)$
inv0_5 : $\forall t \cdot t \in ids \;\Rightarrow\; head(t) \notin middle(t)$
inv0_6 : $\forall t \cdot t \in ids \;\Rightarrow\; rear(t) \notin middle(t)$
inv0_7 : $\forall t \cdot t \in ids \land connection(t) = \varnothing \;\Rightarrow\; head(t) = rear(t)$
inv0_8 : $\forall t, s \cdot t \in ids \land s \in \mathrm{ran}(connection(t)) \;\Rightarrow\; head(t) \mapsto s \in cl(connection(t))$
inv0_9 : $\forall t, s \cdot t \in ids \land s \in \mathrm{dom}(connection(t)) \;\Rightarrow\; s \mapsto rear(t) \in cl(connection(t))$

In the above, $f \in S \rightarrowtail\!\!\!\!\!\!\!+\;\; T$ specifies that f is a *partial injective function* from S to T, $\mathrm{dom}(r)$ and $\mathrm{ran}(r)$ represent the domain and range of a relation r, respectively, and cl denotes the irreflexive transitive closure as defined in [1]. The invariants inv0_5–inv0_9 specify several properties of trains. For example, inv0_7 specifies that if a train occupies only a single section, its head and rear are in the same section. Furthermore, inv0_8 and inv0_9 state that every train is connected from its head and its rear.

To motivate the need for additional abstraction during modelling, we focus on the events train_extend and train_reduce.

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

```
train_reduce :
any  t  where
  t ∈ ids
  rear(t) ∈ ran(connection(t))
then
  rear(t) := connection(t)^{-1}(rear(t))
  connection(t) := connection(t) \ {connection(t)^{-1}(rear(t)) ↦ rear(t)}
end
```

In the above, r^{-1} denotes the inverse relation of r.

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

The event train_reduce reduces a train t by removing its rear. The event's guard guarantees that the train spans at least two sections. Its action updates t's rear and connection accordingly.

Proving that train_extend and train_reduce maintain the invariants, *e.g.*, inv0_5 and inv0_6, requires additional invariants that relate the different aspects of trains, *i.e.*, head, rear, middle, and connection. Two examples of this are the following invariants, inv0_10 and inv0_11.

inv0_10 : $\forall t \cdot middle(t) = \mathrm{dom}(connection(t)) \setminus \{head(t)\}$
inv0_11 : $\forall t \cdot middle(t) = \mathrm{ran}(connection(t)) \setminus \{rear(t)\}$

Table 1 shows the proof statistics for the incomplete development using Rodin 2.3 (the latest version available at the time of development) with the Atelier-B provers. After 14 refinement steps and 45 difficult manual proofs, we stopped our development with numerous undischarged proof obligations remaining due to missing invariants. We would have needed additional invariants that are complex to express and would 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 concluded that continuing this way would be infeasible. Hence it was necessary to adapt our development strategy and incorporate additional abstraction techniques to simplify the proofs. In what follows,

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

Table 1: Statistics from incomplete development without ADTs (Rodin 2.3, with the Atelier-B provers)

we discuss several problems with the current train system model and indicate how further modelling abstractions using ADTs could help to overcome these difficulties.

Encapsulation. The invariants inv0_5–inv0_11 describe that the trains' layouts 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 their proofs, which substantially increases their complexity. This suggests that some encapsulation for the trains' models will help simplify our proofs.

High-level Properties of Low-level Details. An attempt to specify and prove properties such as collision-freeness (REQ 1) 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 suggests that we should start modelling the system at an even more abstract level by omitting the detailed aspects of the sequences.

Modelling Using Transitive Closure. Establishing invariants involving transitive closure, *i.e.*, inv0_8 and inv0_9, is extremely complicated. In the end, we did not manage to prove all of the resulting proof obligations. One reason for this is that additional invariants were missing. However, this suggests that the decision to model using transitive closure was not a good one. In particular, if we can model at a more abstract level, we can delay the decision of which data type is used to model trains until we have more complete requirements for the data type. In fact, the formal model described in Section 4.3 does not use transitive closure.

Reuse. In addition to the above problems, 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 can be modelled using the same ADT, which allows reuse of proofs.

4.3. Development using Abstract Data Types

In our development of the train control system, we develop the ADTs used for modelling the trains and MAs, as follow.

- We start with an ADT that represents *regions* within the network.
- We subsequently instantiate the region ADT with an ADT representing *sequences* of sections.
- Finally, we instantiate the sequence ADT with the ADT corresponding to *arbitrary-based arrays*.

Another entity of the system that we model using ADTs is the active network that is controlled by the interlocking system.

- We start with an ADT representing the *network*.
- We subsequently instantiate the network ADT with an ADT representing *graphs*, where nodes are sections and edges correspond to connections between sections.

These ADTs allow us to encapsulate the concepts useful for modelling, such as a region within a network. Moreover, we used the same ADT to model different aspects of the system. For example, both trains and MAs are represented abstractly by the *Region* ADT. Finally, important high-level properties of the system can be specified and reasoned about easily using the ADTs.

In the following, we describe the ADTs with their usage in our formal model in the order that we defined them.

4.3.1. The Network ADT

The *network* ADT abstractly represents the interlocking system, which controls the active network on which the trains operate. The network ADT has two operations.

- *enlarge*: takes a network N, a relation between sections g (which must be a partial injective function), a set of sections s, and returns a new network where g and s become active, *i.e.*, trains can operate on these connections and sections.
- *contract*: takes a network N, a relation between sections g (which must be a partial injective function), a set of sections s, and returns a new network where g and s become inactive.

In Event-B, the network ADT is modelled as follows.

sets : $NETWORK_TYPE$

constants : $NETWORK, enlarge, contract$

axioms :

axm0_1 : $NETWORK \subseteq NETWORK_TYPE$

axm0_2 : $enlarge \in \begin{pmatrix} & NETWORK \\ \times & (SECTION \rightarrowtail\!\!\!\!\!\!+\; SECTION) \\ \times & \mathbb{P}(SECTION) \end{pmatrix} \nrightarrow NETWORK$

axm0_3 : $contract \in \begin{pmatrix} & NETWORK \\ \times & (SECTION \rightarrowtail\!\!\!\!\!\!+\; SECTION) \\ \times & \mathbb{P}(SECTION) \end{pmatrix} \nrightarrow NETWORK$

The axioms axm0_1–axm0_3 specify "typing" information of the operations. Additional axioms stating the consistency of the network ADT's operations are related to the *region* ADT introduced in Section 4.3.2.

Using this network ADT, the interlocking system can be abstractly modelled as follows, where *network* represents the current active network.

variables : $network$

invariants :

inv0_1 : $network \in NETWORK$

```
network_enlarge :
any   g, s   where
   network ↦ g ↦ s ∈ dom(enlarge)
then
   network := enlarge(network ↦ g ↦ s)
end
```

```
network_contract :
any   g, s   where
   network ↦ g ↦ s ∈ dom(contract)
then
   network := contract(network ↦ g ↦ s)
end
```

4.3.2. The Region ADT

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

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

Note that there are other operations of the region ADT that we omit for clarity. The region ADT is formalised as follows.

sets : $REGION_TYPE$

constants : $REGION, contained, disjoint, extend$

axioms :

axm1_1 : $REGION \subseteq REGION_TYPE$
axm1_2 : $contained \in REGION \leftrightarrow REGION$
axm1_3 : $disjoint \in REGION \leftrightarrow REGION$
axm1_4 : $extend \in REGION \times SECTION \pfun REGION$

Constraints on the operations of the region ADT are modelled as axioms. For example, *contained* is transitive (axm1_5), *disjoint* is symmetric (axm1_6), and *extend* is strengthening with respect to *contained* (axm1_7). Note that in the following, we use $R_1 \Subset R_2$ to denote $R_1 \mapsto R_2 \in contained$, and $R_1 \not\pitchfork R_2$ to denote $R_1 \mapsto R_2 \in disjoint$.

axm1_5 : $\forall R_1, R_2, R_3 \cdot R_1 \Subset R_2 \land R_2 \Subset R_3 \;\Rightarrow\; R_1 \Subset R_3$
axm1_6 : $\forall R_1, R_2 \cdot R_1 \not\pitchfork R_2 \;\Rightarrow\; R_2 \not\pitchfork R_1$
axm1_7 : $\forall R, s \cdot R \mapsto s \in \mathrm{dom}(extend) \;\Rightarrow\; R \Subset extend(R \mapsto s)$
axm1_8 : $\forall R_1, R_2, R_3 \cdot R_1 \Subset R_2 \land R_2 \not\pitchfork R_3 \;\Rightarrow\; R_1 \not\pitchfork R_3$

Note that our axiomatisation simplifies the construction of correctness proofs for the system, where the role of the axioms is similar to that of lemmas in a complex proof. During generic instantiation, these axioms become theorem statements that must be proven.

To express the non-derailment property (REQ 2), we must link the region ADT with the network ADT defined in Section 4.3.1. The following "inside" operation captures the relationship between them.

- *inside*: a binary relation associating a region R with a network N if R is inside N. In the following, $R \sqsubseteq N$ denotes $R \mapsto N \in inside$.

constants : *inside*

axm1_9 : $inside \in REGION \leftrightarrow NETWORK$
axm1_10 : $\forall R_1, R_2 \cdot R_1 \Subset R_2 \land R_2 \sqsubseteq N \;\Rightarrow\; R_1 \sqsubseteq N$
axm1_11 : $\forall R, N, g, s \cdot R \sqsubseteq N \land N \mapsto g \mapsto s \in \mathrm{dom}(enlarge) \;\Rightarrow$
$R \sqsubseteq enlarge(N \mapsto g \mapsto s)$

The current states of the active trains and their associated movement authorities are represented by two mappings (*train* and *ma*) from trains' identifiers *ids* to the set of possible regions *REGION*. The invariant inv1_3 states that the trains always stay within their movement authorities. The invariant inv1_4 states that the movement authorities of any two trains are disjoint. The invariant inv1_5 states that the movement authorities are always inside the active network.

variables : *ids*, *train*, *ma*

invariants :
inv1_1 : $train \in ids \rightarrow REGION$
inv1_2 : $ma \in ids \rightarrow REGION$
inv1_3 : $\forall t \cdot t \in ids \;\Rightarrow\; train(t) \Subset ma(t)$
inv1_4 : $\forall t_1, t_2 \cdot t_1 \in ids \land t_2 \in ids \land t_1 \neq t_2 \;\Rightarrow\; ma(t_1) \not\pitchfork ma(t_2)$
inv1_5 : $\forall t \cdot t \in ids \;\Rightarrow\; ma(t) \sqsubseteq network$

Properties such as collision-freeness (REQ 1) and non-derailment (REQ 2) are theorems derivable from the invariants inv1_3, inv1_4, and inv1_5, and the property relating $\Subset$, $\not\pitchfork$, and $\sqsubseteq$, *e.g.*, axm1_6, axm1_8, and axm1_10.

$$
\begin{array}{ll}
\textit{thm1_1}: & \forall t_1, t_2 \cdot t_1 \in \textit{ids} \land t_2 \in \textit{ids} \land t_1 \neq t_2 \ \Rightarrow\ \textit{train}(t_1) \not\pitchfork \textit{train}(t_2) \\
\textit{thm1_2}: & \forall t \cdot t \in \textit{ids} \ \Rightarrow\ \textit{train}(t) \sqsubseteq \textit{network}
\end{array}
$$

For example, the proof of *thm1_1* is as follows. For all t_1 and t_2, we prove that if $t_1 \in \textit{ids} \land t_2 \in \textit{ids} \land t_1 \neq t_2$ then $\textit{train}(t_1) \not\pitchfork \textit{train}(t_2)$.

$$
\begin{array}{ll}
 & \textit{train}(t_1) \not\pitchfork \textit{train}(t_2) \\
\Leftarrow & \quad \{\text{axm1_8}\} \\
 & \textit{train}(t_1) \Subset \textit{ma}(t_1) \ \land\ \textit{ma}(t_1) \not\pitchfork \textit{train}(t_2) \\
\Leftarrow & \quad \{\text{inv1_3}\} \\
 & \textit{ma}(t_1) \not\pitchfork \textit{train}(t_2) \\
\Leftarrow & \quad \{\text{axm1_6}\} \\
 & \textit{train}(t_2) \not\pitchfork \textit{ma}(t_1) \\
\Leftarrow & \quad \{\text{axm1_8}\} \\
 & \textit{train}(t_2) \Subset \textit{ma}(t_2) \ \land\ \textit{ma}(t_2) \not\pitchfork \textit{ma}(t_1) \\
\Leftarrow & \quad \{\text{inv1_3 and inv1_4}\} \\
 & \top
\end{array}
$$

This proof corresponds to the informal reasoning mentioned in Section 4.1. It illustrates that by introducing the region ADT we can abstractly state and prove central system properties such as collision-freeness and non-derailment early in the development. This leads to simple and understandable proofs.

The event network_enlarge maintains inv1_5 trivially (with axm1_11). For the event network_contract, we must strengthen the guard accordingly.

network_contract :
any g, s **where**
$\quad \textit{network} \mapsto g \mapsto s \in \text{dom}(\textit{contract})$
$\quad \forall t \cdot t \in \textit{ids} \ \Rightarrow\ \textit{ma}(t) \sqsubseteq \textit{contract}(\textit{network} \mapsto g \mapsto s)$
then
$\quad \textit{network} := \textit{contract}(\textit{network} \mapsto g \mapsto s)$
end

The event train_extend can be specified abstractly as follows. The last predicate of its guard ensures that the extended train never exceeds its assigned movement authority, and hence maintains inv1_3.

$$
\begin{array}{l}
\textsf{train_extend}: \\
\textbf{any} \quad t, s \quad \textbf{where} \\
\quad t \in \mathrm{dom}(train) \\
\quad train(t) \mapsto s \in \mathrm{dom}(extend) \\
\quad extend(train(t) \mapsto s) \Subset ma(t) \\
\textbf{then} \\
\quad train(t) := extend(train(t) \mapsto s) \\
\textbf{end}
\end{array}
$$

4.3.3. 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 and network ADTs, which are not fully "implemented". For (2), we utilise generic instantiation to introduce more details on how the region and network ADTs and their operations are realised. Similar to refinement, this realisation 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.
- $link$: takes a sequence S and returns the link between sections from the $head$ to the $rear$ of S.

$$\textbf{sets}: \quad SEQUENCE_TYPE$$

$$\textbf{constants}: \quad SEQUENCE, prepend, head, rear, middle, link$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{axioms}:} \\
\textsf{axm2_1}: & SEQUENCE \subseteq SEQUENCE_TYPE \\
\textsf{axm2_2}: & prepend \in SEQUENCE \times SECTION \rightarrow\!\!\!\!+ SEQUENCE \\
\textsf{axm2_3}: & head \in SEQUENCE \rightarrow SECTION \\
\textsf{axm2_4}: & rear \in SEQUENCE \rightarrow SECTION \\
\textsf{axm2_5}: & middle \in SEQUENCE \rightarrow \mathbb{P}(SECTION) \\
\textsf{axm2_6}: & link \in SEQUENCE \rightarrow (SECTION \rightarrowtail\!\!\!\!+ SECTION) \\
\textsf{axm2_7}: & \forall S \cdot S \in SEQUENCE \;\Rightarrow\; head(S) \notin middle(S) \\
\textsf{axm2_8}: & \forall S \cdot S \in SEQUENCE \;\Rightarrow\; rear(S) \notin middle(S)
\end{array}
$$

Note that our sequence ADT formalizes some domain-specific aspects, such as the axioms axm2_7 and axm2_8; as such, it differs from the sequence data type, for example, in [10]. As mentioned before, we have omitted additional operations, such as the head and rear positions within a section, and their corresponding axioms, for brevity.

We prove that the sequence ADT is a valid representation of the region ADT with the following instantiation.

$$\begin{aligned} \mathit{REGION_TYPE} &\longleftarrow \mathit{SEQUENCE_TYPE} \\ \mathit{REGION} &\longleftarrow \mathit{SEQUENCE} \\ \mathit{extend} &\longleftarrow \mathit{prepend} \end{aligned}$$

We replace (instantiate) the operations *contained* and *disjoint* using *head*, *rear*, *middle*, and *link*. For example, *contained* is instantiated as

$$\mathit{contained} = \left\{ S_1 \mapsto S_2 \mid \begin{array}{l} S_1 \in \mathit{SEQUENCE} \land S_2 \in \mathit{SEQUENCE} \land \\ \mathit{link}(S_1) \subseteq \mathit{link}(S_2) \land \\ \mathit{members}(S_1) \subseteq \mathit{members}(S_2) \land \\ \ldots \end{array} \right\},$$

where $\mathit{members}(S) = \{\mathit{head}(S)\} \cup \mathit{middle}(S) \cup \{\mathit{rear}(S)\}$. 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 have not given the exact definition for sequences and we still rely on operators such as *head*, *rear*, *middle*, and *link* and the relationships between them.

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

```
train_extend :
any   t, s   where
   t ∈ dom(train)
   head(train(t)) ≠ head(ma(t))
   ... // other guards related to head/rear positions
then
   train(t) := prepend(train(t) ↦ s)
end
```

4.3.4. The Graph ADT

Similar to how the region ADT is replaced with the sequence ADT, we also transform the network ADT into the graph ADT, where sections represent nodes and connections between sections are modelled as edges. The graph ADT has the following operations:

- *edges*: take a graph G and returns its edges (a partial injective function between sections).
- *nodes*: take a graph G and returns its nodes (a set of $\mathit{SEQUENCE}$).

- *add*: takes a network G, a relation between sections g (a partial injective function), a set of sections s, and returns a new graph where G and s are added.
- *remove*: takes a network G, a relation between sections g (a partial injective function), a set of sections s, and returns a new graph where g and s are removed.

The formalisation of the graph ADT is as follows. The axioms axm3_6 and axm3_7 specify that a graph's edges connect only the nodes of the graph. The axioms axm3_8–axm3_11 express how graphs' edges and nodes change with the operations *add* and *remove*.

sets : $\mathit{GRAPH_TYPE}$

constants : $\mathit{GRAPH}, \mathit{edges}, \mathit{nodes}, \mathit{add}, \mathit{remove}$

axioms :

axm3_1 : $\mathit{GRAPH} \subseteq \mathit{GRAPH_TYPE}$

axm3_2 : $\mathit{edges} \in \mathit{GRAPH} \to (\mathit{SECTION} \rightarrowtail\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!+ \ \mathit{SECTION})$

axm3_3 : $\mathit{nodes} \in \mathit{GRAPH} \to \mathbb{P}(\mathit{SECTION})$

axm3_4 : $\mathit{add} \in \begin{pmatrix} & \mathit{GRAPH} \\ \times & (\mathit{SECTION} \rightarrowtail\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!+ \ \mathit{SECTION}) \\ \times & \mathbb{P}(\mathit{SECTION}) \end{pmatrix} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{GRAPH}$

axm3_5 : $\mathit{remove} \in \begin{pmatrix} & \mathit{GRAPH} \\ \times & (\mathit{SECTION} \rightarrowtail\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!+ \ \mathit{SECTION}) \\ \times & \mathbb{P}(\mathit{SECTION}) \end{pmatrix} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{GRAPH}$

axm3_6 : $\forall G \cdot G \in \mathit{GRAPH} \Rightarrow \mathrm{dom}(\mathit{edges}(G)) \subseteq \mathit{nodes}(G)$

axm3_7 : $\forall G \cdot G \in \mathit{GRAPH} \Rightarrow \mathrm{ran}(\mathit{edges}(G)) \subseteq \mathit{nodes}(G)$

axm3_8 : $\forall G, g, s \cdot G \mapsto g \mapsto s \in \mathrm{dom}(\mathit{add}) \Rightarrow \mathit{edges}(\mathit{add}(G \mapsto g \mapsto s)) = \mathit{edges}(G) \cup g$

axm3_9 : $\forall G, g, s \cdot G \mapsto g \mapsto s \in \mathrm{dom}(\mathit{add}) \Rightarrow \mathit{nodes}(\mathit{add}(G \mapsto g \mapsto s)) = \mathit{nodes}(G) \cup s$

axm3_10 : $\forall G, g, s \cdot G \mapsto g \mapsto s \in \mathrm{dom}(\mathit{remove}) \Rightarrow \mathit{edges}(\mathit{remove}(G \mapsto g \mapsto s)) = \mathit{edges}(G) \setminus g$

axm3_11 : $\forall G, g, s \cdot G \mapsto g \mapsto s \in \mathrm{dom}(\mathit{remove}) \Rightarrow \mathit{nodes}(\mathit{remove}(G \mapsto g \mapsto s)) = \mathit{nodes}(G) \setminus s$

The graph ADT is a trivial realisation of the network ADT, where the operations *add* and *remove* implement *enlarge* and *contract*.

$$\begin{aligned} \mathit{NETWORK_TYPE} &\longleftarrow \mathit{GRAPH_TYPE} \\ \mathit{NETWORK} &\longleftarrow \mathit{GRAPH} \\ \mathit{enlarge} &\longleftarrow \mathit{add} \\ \mathit{contract} &\longleftarrow \mathit{remove} \end{aligned}$$

The operation *inside* that connects the network ADT and region ADT is realised by linking the graph ADT and the sequence ADT's operations as follows:

$$inside = \left\{ S \mapsto G \mid \begin{array}{l} S \in SEQUENCE \land G \in GRAPH \;\land \\ link(S) \subseteq edges(G) \;\land \\ members(S) \subseteq nodes(G) \end{array} \right\}.$$

A sequence S is within a graph G if S's links are edges of G, and the members (head, middle, and rear) of S are nodes of G. This realisation of *inside* leads to the following refinement of the event network_contract.

```
network_contract :
any   g, s   where
   network ↦ g ↦ s ∈ dom(contract)
   ∀t·t ∈ ids ⇒ link(ma(t)) ∩ g = ∅
   ∀t·t ∈ ids ⇒ members(ma(t)) ∩ s = ∅
then
   network := contract(network ↦ g ↦ s)
end
```

The event network_contract specifies that when the interlocking system disables a part of the active network, represented by (g, s), there is no overlap between this part and the MAs of any active train.

4.3.5. The Arbitrarily-based Array Data Type

The model based on the sequence ADT is abstract. To ensure that the model of the software controller is implementable, we must give a representation for the sequence ADT. In our development, we use an arbitrarily-based array data type to implement 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 $a \mapsto b \mapsto f$, 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 *link* 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 standard (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.

$$prepend = \left(\begin{array}{l} \lambda\ (a \mapsto b \mapsto f) \mapsto s \cdot \\ \quad a \mapsto b \mapsto f \in ARRAY\ \wedge \\ \quad s \in SECTION \wedge \\ \quad \ldots \\ \mid \\ \quad (a-1) \mapsto b \mapsto (f \Leftarrow \{a-1 \mapsto s\}) \end{array} \right)$$

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

4.4. Development Summary and Comparison

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 incrementally transformed in small steps. This also serves to decompose the proof of correctness of the systems into small, simple instantiation and refinement steps.

Our development contains five different stages (numbered 0–4), connected by instantiation relationships, where Stages 1–4 start as an instantiation of the previous stage. Each stage contains several refinement steps, 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 and network ADTs to the sequence and graph ADTs in three different instantiations. In **Stage 1** we instantiate the *contained* operation (other operations such as *inside* and *disjoint* are axiomatically defined as before). In **Stage 2** we instantiate the *inside* operation between the region ADT and the network ADT. Finally, in **Stage 3** 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 system's design details, 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.

Overall, our model using ADTs contains 67 refinements. The last machine contains 25 variables (6 that model the environment, 15 that model the state of the controller, and 4 that model the communication channels); 30 events (11 that model the behaviour of the environment, 13 that specify how the controller calculate MAs, and 6 that model the communication between the trains and the controller). There are 107 invariants and 56 theorems within these 67 refinements.

Table 2 shows the proof statistics for our development using ADTs (for Rodin 3.0, with the Atelier-B provers). 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, proofs for 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 further increase the amount of proof automation. Overall, the instantiation proofs have a better automation rate (82%) than the refinement proofs (62%).

		Obligations	Auto.	Manual
Stage 0	8 Refinements	267	267	0
Stage 1	Instantiation	34	24	10
	14 Refinements	632	493	139
Stage 2	Instantiation	165	161	4
	1 Refinement	57	48	9
Stage 3	Instantiation	175	172	3
	16 Refinements	761	441	320
Stage 4	Instantiation	174	90	84
	28 Refinements	1590	794	796
Total		3855	2490 (65%)	1365 (35%)
	Instantiation	548 (14%)	447 (82%)	101 (18%)
	Refinement	3307 (86%)	2043 (62%)	1264 (38%)

Table 2: Development with ADTs (Rodin 3.0 with the Atelier-B provers)

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, the level of detail in our models, which stems from realistic requirements, supports our claims about the relevance of our approach for large and complex real-world systems.

5. Related Work

5.1. Instantiation and Data Types

In our approach to introducing ADTs into formal development, generic instantiation [3] is the key technique for realising the ADTs. This differs from [9] where instantiation provides a means to reuse formal models in combination with a composition technique. In particular, to guarantee the correctness of an instantiated model, carrier sets (which are assumed to be non-empty and maximal) must be instantiated by type expressions. This has been overlooked in [3] and [9].

Part of our approach was previously published in [14]. The current paper is an extended version of our conference paper [15]. In [14], 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, complex systems. We not only use ADTs to abstract away implementation details for the domain expert, we also use them as an integral part of our development from the start to abstractly specify the system's properties and to simplify the proofs.

The development of the *Theory Plug-in* [16] for Rodin allows users to extend the mathematical language of Event-B, for example, by defining new data types, including polymorphic ones. Theorems about new data types can be stated and later used by a dedicated tactic associated with the 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 complete 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, and 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 experimented with both implementations and decided to use arbitrarily-based arrays as they resulted in simpler proofs.

A recent release of the Theory plug-in includes the capability to define types and operators *axiomatically*. However, polymorphism is not yet supported. As a result, it is similar to our approach of using contexts to define ADTs. A notable difference is that operators defined axiomatically require the users to specify their *well-definedness* conditions when the operators are partial. Specifying these conditions for an operator at the abstract level can be challenging since not all necessary details about the data type are available. In our approach, an operator is defined by a partial function and the operator's well-definedness condition is abstracted by the function's domain. We can delay the exact specification of the well-definedness condition until the data type is concrete enough for that purpose. Using the Theory plug-in, additional operations must be added to capture the "abstract" well-definedness conditions of the abstract operators.

Our approach of using ADTs in Event-B is similar to work on algebraic specifications [17]. In this domain, a specification contains a collection of *sorts*, *operations*, and *axioms* constraining the operations. Specifications can be *enriched* by additional sorts, operations, and axioms. Furthermore, to develop programs from specifications, the specifications are transformed by 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 operations' code 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 [18], where the entire functionality of a system is modelled as ADTs (in the form of many-sorted algebras) [17], 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 [19] 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. We discuss the most relevant approaches below.

The development of Metro line 14 in Paris [20, 21] is one of the better known industrial applications of formal methods. In particular, the safety critical part of the software was developed using the (classical) B Method [10]. The formal reasoning there was only at the software-level, *i.e.*, reasoning about the correctness of the software in isolation. In contrast, we model not only the train control system, but also its environment, including 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 [22].

In [23], James et. al. present an approach using the CSP || B formalism to model and verify railway interlockings. In particular, they consider the tracking of train lengths for trains spanning over several section. This is similar to our model of the head and rear positions of the trains (which is omitted in this paper for brevity). However, since they ultimately rely on model checkers as their verification tools, their model is already "concrete". Properties such as collision-freeness are checked at this concrete level. In our development, we initially abstracted this information away using ADTs when proving these system properties. Subsequently, when we instantiated our ADTs with more concrete representations, the system properties are automatically preserved. Moreover, in [23], the properties are verified for two concrete network layouts, whereas we make no assumptions regarding the actual network.

Platzer and Quesel verify parts of a similar train control system in [24] using their own verification tool KeYmaera. While we developed the functionality of the controller, their development focuses on 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 using ADTs and refinement. The ADTs allow us to hide details that are unimportant for proving abstract properties. One can therefore focus on building and refining abstract models of 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 used ADTs to overcome this problem. In the subsequent work [25], we took the current development and generated code for the train controller. Furthermore, we developed a framework for simulating the generated code with a given network topology. For this purpose, we instantiated the network with tracks and points in a realistic way.

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 handle this, we would need to extend the semantics of Event-B contexts and adapt the generic instantiation technique accordingly. We are also investigating the possibility of incorporating the notion of data type instantiation within the Theory plug-in.

References

[1] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.

[2] B. Liskov, S. Zilles, Programming with Abstract Data Types, in: Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, ACM, New York, NY, USA, 1974, pp. 50–59, http://doi.acm.org/10.1145/800233.807045.

[3] J.-R. Abrial, S. Hallerstede, Refinement, decomposition, and instantiation of discrete models: Application to Event-B, Fundam. Inform. 77 (1–2) (2007) 1–28.

[4] IEEE Std 1474.1-2004, IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements, IEEE, New York, USA, 2005.

[5] European Railway Agency, ERTMS/ETCS Functional Requirements Specification, European Railway Agency, Valencinnes, France, 2007.

[6] E. Börger, Why use evolving algebras for hardware and software engineering?, in: M. Bartosek, J. Staudek, J. Wiedermann (Eds.), SOFSEM '95, 22nd Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 23 - December 1, 1995, Proceedings, Vol. 1012 of Lecture Notes in Computer Science, Springer, 1995, pp. 236–271. doi:10.1007/3-540-60609-2_12.
URL http://dx.doi.org/10.1007/3-540-60609-2_12

[7] R. Back, Refinement calculus, part II: parallel and reactive programs, in: J. W. de Bakker, W. P. de Roever, G. Rozenberg (Eds.), Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings, Vol. 430 of Lecture Notes in Computer Science, Springer, 1989, pp. 67–93. doi:10.1007/3-540-52559-9_61.
URL http://dx.doi.org/10.1007/3-540-52559-9_61

[8] L. Lamport, The temporal logic of actions, Transactions on Programming Languages and Systems (TOPLAS) 16 (3) (1994) 872–923.

[9] R. Silva, M. Butler, Supporting reuse of Event-B developments through generic instantiation, in: Breitman and Cavalcanti [26], pp. 466–484, http://dx.doi.org/10.1007/978-3-642-10373-5_24.

[10] J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, 1996.

[11] T. S. Hoang, An introduction to the Event-B modelling method, in: A. Romanovsky, M. Thomas (Eds.), Industrial Deployment of System Engineering Methods, Springer-Verlag, 2013, pp. 211–236, http://www.springer.com/computer/swe/book/978-3-642-33169-5.

[12] A. Fürst, K. Desai, T. S. Hoang, N. Sato, Generic instantiation plug-in for the Rodin platform, http://sourceforge.net/projects/gen-inst/ (2014).

[13] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: An open toolset for modelling and reasoning in Event-B, Software Tools for Technology Transfer 12 (6) (2010) 447–466.

[14] D. Basin, A. Fürst, T. S. Hoang, K. Miyazaki, N. Sato, Abstract data types in Event-B - an application of generic instantiation, CoRRhttp://arxiv.org/abs/1210.7283.

[15] A. Fürst, T. S. Hoang, D. Basin, N. Sato, K. Miyazaki, Formal system modelling using abstract data types in Event-B, in: Y. A. Ameur, K.-D. Schewe (Eds.), ABZ, Vol. 8477 of Lecture Notes in Computer Science, Springer-Verlag, Toulouse, France, 2014, pp. 222–237, http://dx.doi.org/10.1007/978-3-662-43652-3_20.

[16] M. Butler, I. Maamria, Practical theory extension in Event-B, in: Z. Liu, J. Woodcock, H. Zhu (Eds.), Theories of Programming and Formal Methods, Vol. 8051 of Lecture Notes in Computer Science, Springer, 2013, pp. 67–81, http://dx.doi.org/10.1007/978-3-642-39698-4_5.

[17] D. Sannella, A. Tarlecki, Essential concepts of algebraic specification and program development, Formal Asp. Comput. 9 (3) (1997) 229–269, http://dx.doi.org/10.1007/BF01211084.

[18] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, Vol. 6 of EATCS Monographs on Theoretical Computer Science, Springer, 1985, http://dx.doi.org/10.1007/978-3-642-69962-7.

[19] D. Bjørner, New results and trends in formal techniques & tools for the development of software for transportation systems, in: FORMS, 2003.

[20] P. Behm, P. Benoit, A. Faivre, J.-M. Meynadier, Météor: A successful application of B in a large project, in: J. M. Wing, J. Woodcock, J. Davies (Eds.), World Congress on Formal Methods, Vol. 1708 of Lecture Notes in Computer Science, Springer, Toulouse, France, 1999, pp. 369–387, http://dx.doi.org/10.1007/3-540-48119-2_22.

[21] J.-R. Abrial, Formal methods in industry: Achievements, problems, future, in: L. Osterweil, H. Rombach, M. Soffa (Eds.), ICSE, ACM, Shanghai, China, 2006, pp. 761–768, http://doi.acm.org/10.1145/1134406.

[22] J.-R. Abrial, From Z to B and then Event-B: Assigning proofs to meaningful programs, in: E. Johnsen, L. Petre (Eds.), IFM, Vol. 7940 of Lecture Notes in Computer Science, Springer, 2013, pp. 1–15, http://dx.doi.org/10.1007/978-3-642-38613-8_1.

[23] P. James, F. Moller, H. N. Nguyen, M. Roggenbach, S. A. Schneider, H. Treharne, On modelling and verifying railway interlockings: Tracking train lengths, Sci. Comput. Program. 96 (2014) 315–336. doi:10.1016/j.scico.2014.04.005.
URL http://dx.doi.org/10.1016/j.scico.2014.04.005

[24] A. Platzer, J.-D. Quesel, European train control system: A case study in formal verification, in: Breitman and Cavalcanti [26], pp. 246–265, http://dx.doi.org/10.1007/978-3-642-10373-5_13.

[25] A. Fürst, T. S. Hoang, D. Basin, K. Desai, N. Sato, K. Miyazaki, Code generation for Event-B, in: E. Albert, E. Sekerinski (Eds.), IFM, Vol. 8739 of Lecture Notes in Computer Science, Springer, Bertinoro, Italy, 2014, pp. 323–338.
URL http://dx.doi.org/10.1007/978-3-319-10181-1

[26] K. Breitman, A. Cavalcanti (Eds.), Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Vol. 5885 of Lecture Notes in Computer Science, Springer, Rio de Janeiro, Brazil, 2009, http://dx.doi.org/10.1007/978-3-642-10373-5.