A System-based Approach to the Formal Development of Embedded Controllers for a Railway

Michael Butler*
Department of Electronics and Computer Science,
University of Southampton,
Highfield, Southampton SO17 IBJ, United Kingdom

4 January 2002

Abstract

We describe a formal approach to the development of embedded controllers for a railway. The approach starts with a system-level specification modelling the system under control and the desired control behaviour. Correctness-preserving refinement is then used to add more and more implementation detail to the models and to decompose the models into sub-systems to arrive at models of individual controllers. The B Method is used as the formal notation and methodology.

Keywords: Verification, Refinement, Embedded Control Systems, B Method.

1 Introduction

Formal methods provide powerful analytical ways of validating the safety and correctness of systems containing embedded controllers. This helps to ensure a higher level of dependability, which is essential when embedded controllers are used in safety critical systems. Many different formalisms have been applied to control system development. For example, the volume [3] presents over twenty case studies involving the application of various formal methods to the specification and analysis of a steam boiler controller.

In this paper, we describe a system-level approach to the development of embedded controllers for a railway. It is a system-level approach in the sense that we start with an abstract formal model of the overall system to be controlled, including relevant control behaviour. We then refine this high-level model, adding detail about interactions between sub-components of the system, so that formal models may be decomposed into the system under control and the embedded controllers. Thus, rather than starting with formal models of the embedded controllers and verifying that they interact with the system under control, we start with a model of the desirable behaviour of the overall system and use refinement to derive models of the embedded controllers. The refinement is typically performed in several steps, and at each step we verify that any behaviour of the refined model is allowed by the previous model, thus ensuring that the final detailed model is correct with respect to the original system-level model. The advantage of our approach is that is makes it easier to deal with the complexity of the designs. We start with a simple system-level model and introduce extra complexity one step at a time. The approach we take is particularly suited to systems which consist of multiple distributed controllers. In the system level model, we abstract away from any distribution and only introduce it in refinement steps.

The railway case study presented here is a simplified version of work that we are undertaking as part of the MATISSE project with input from one of the MATISSE partners, Siemens

*This work is part of the EU-funded MATISSE project (IST-1999-11435).

Transportation Systems. Our system-level model describes features of the railway such as connections between track sections, occupancy of track sections by trains, basic movement of trains and changing of track connections. The model represents safety features only and does not represent mission-oriented features such as scheduling and routing. We outline a refinement which allows the system to be decomposed into the track behaviour and the train behaviour, with communication between the track and the trains.

The formal method we use is the B Method developed by Abrial [2]. B is based on set theory and predicate logic and has strong notions of modularity and refinement. B is used in several critical industries and has commercial tools [8, 4] which support refinement and code generation. Traditionally B has not been used for the development of distributed systems. More recently, B has been extended to deal with the development of distributed systems and this extended form is known as *Event B* [1]. Our approach uses Event B.

In this paper, we give a very brief introduciton to the B notation. We then describe the system-level model. We introduce the B notion of refinement and present some of the refinement steps in the development.

2 Introducing the B Notation

In B, a specification consists of a hierarchically structured collection of *machines*. Each machine consists of the some state variables, representing the state space of the system, an initialisation for the system and a set of operations on the variables of the system. The variables are typically structured sets, such as relations and functions.

An operation definition has the form

$$a(x) \mathrel{\hat{=}} S,$$

where a is the action name, x represents some parameter(s), and S is a statement in the language which we briefly outline now. Changes to the state variables are specified using assignment statements ($v := E$). A simultaneous assignment of the form $x := E \parallel y := F$ is equivalent to $x, y := E, F$. A guarded operation is written as a statement of the form:

SELECT G THEN S END,

where the guard G is some condition on the state variables and S is some statement. Such an operation is enabled only when G holds.

B machines are state-based since they use an explicit model of the state of a system. But an event-based view can also be taken of a B machine. In the event-based view, execution of an operation a with parameter value e models occurrence of the event $a(e)$. Occurrence of the event is only possible when the operation is enabled.

3 System-level Model of Railway

Table 1 describes an abstraction of the railway states in B notation. (Here, and in other tables, parts of the B secifications are omitted.) *SECTION* and *TRAIN* are types used to identify track sections and trains respectively (1). A railway switch has two incoming and one outgoing sections (convergent) or vice versa (divergent). Railway switches are regarded as special cases of track sections. The constant *switch* (2) represents the subset of sections that are switches, while the constant *net* is a graph representing the static connectivity between sections (i.e., the connectivity regardless of the switch positions). In the *properties* clause, *switch* is declared to be a subset of *SECTION* (3), while *net* is declared to be a relation between *SECTIONS* (4). Here, we have omitted other properties of *net* such as the fact that a section cannot be connected to itself.

The variables of the machine (5) are typed in the *invariant* clause. Variable *next* represents the dynamic connectivity between sections based on the positions of switches. Note that we have

$$
\begin{array}{lr}
\text{MACHINE } \mathit{Railway1} & \\
\text{SETS } \mathit{SECTION}, \mathit{TRAIN} & (1) \\
\text{CONSTANTS } \mathit{switch}, \mathit{net} & (2) \\
\text{PROPERTIES} & \\
\quad \mathit{switch} \subseteq \mathit{SECTION} \ \wedge & (3) \\
\quad \mathit{net} \in \mathit{SECTION} \leftrightarrow \mathit{SECTION} & (4) \\
\text{VARIABLES } \mathit{next}, \mathit{occp}, \mathit{front}, \mathit{back}, \mathit{braking}, \mathit{checked} & (5) \\
\text{INVARIANT} & \\
\quad \mathit{next} \subseteq \mathit{net} \ \wedge & (6) \\
\quad \mathit{next} \in \mathit{SECTION} \rightarrowtail \mathit{SECTION} \ \wedge & (7) \\
\quad \mathit{occp} \in \mathit{TRAIN} \leftrightarrow \mathit{SECTION} \ \wedge & (8) \\
\quad \mathit{front} \in \mathit{TRAIN} \rightarrow \mathit{SECTION} \ \wedge \ \mathit{front} \subseteq \mathit{occp} \ \wedge & (9) \\
\quad \mathit{back} \in \mathit{TRAIN} \rightarrow \mathit{SECTION} \ \wedge \ \mathit{back} \subseteq \mathit{occp} \ \wedge & (10) \\
\quad \mathit{braking} \in \mathit{TRAIN} \rightarrow \mathit{Bool} \ \wedge & (11) \\
\quad \mathit{checked} \in \mathit{TRAIN} \rightarrow \mathit{Bool} \ \wedge & (12) \\
\quad \forall\, t1, t2.(t1 \in \mathit{TRAIN} \ \wedge \ t2 \in \mathit{TRAIN} \ \wedge & \\
\qquad \mathit{front}(t1) = \mathit{back}(t2) \ \Rightarrow \ \mathit{braking}(t1) = \mathit{TRUE} & (13)
\end{array}
$$

Table 1: Variables and invariant of abstract railway.

abstracted away from the actual positions of switches and concentrated on connectivity. This makes it easier to specify and analyse the system-level model. Variable *next* will always be a subgraph of *net* (6) and be an injective function (7) meaning a section can have at most one predecessor and at most one successor under *next*.

Each train may occupy several sections and this is modelled by the relation *occp* (8). Note that more than one train may be related to the same section by *occp* which is a potential hazard since trains occupying the same section may collide. Although the railway strives to avoid multiple occupancy of the same section, we feel that a pure model which absolutely prevented it would be unrealistic. The invariant does have a weaker safety constraint saying that if the front of $t1$ is on the same section as the back of $t2$, then $t2$ should be braking (13). This may be insufficient to avoid collisions since it doesn't take account of stopping distance, but it gives a flavour of a safety constraint for the purposes of this paper. The functions *front* and *back* map each train to the section occupied by its front and back respectively (9),(10). The function *braking* maps each train to a boolean representing whether or not it is braking.

Table 2 describes some operations of the system-level model. The *Check* operation checks whether the brakes need to be applied to a train t. The brakes are applied if the front of t has no successor or its successor is already occupied[1]. The brakes are applied by setting $\mathit{braking}(t)$ to *TRUE*. Regardless of whether the brakes need to be applied, the *Check* operation sets the $\mathit{checked}(t)$ flag to true.

The *EnterSection* operation models the front of train t entering section s. Train t can enter section s provided t has been checked (since it last entered a section) and t has a successor section under *next*. The effect of the operation is to change the front section of the train to be s, add a mapping from t to s to the relation *occp* and reset the $\mathit{checked}(t)$ flag.

Use of the *checked* flag means that the check operation must be applied to each train t at least once in between each *EnterSection* for t. In the real system, the train cannot be directly prevented from entering a new section and the only way of satisfying this ordering constraint is to ensure that the *Check* operation is performed before the train enters a new section. So this represents a hard real-time constraint.

The *SwitchChangeDiv* operation represents a switch change for a divergent switch and is modelled by the effect the switch change has on the *next* function. Parameter *sw* must be a switch

[1] $t \notin \mathit{dom}(\mathit{next})$ means that $\mathit{next}(t)$ is undefined. $\mathit{ran}(\mathit{occp})$ is the set of sections that are occupied.

$$
\begin{array}{l}
\mathit{Check}(t : \mathit{TRAIN}) = \\
\quad \mathit{braking}(t) := \mathit{bool}(\mathit{front}(t) \notin \mathit{dom}(\mathit{next}) \;\vee\; \mathit{next}(\mathit{front}(t)) \in \mathit{ran}(\mathit{occp})) \qquad || \\
\quad \mathit{checked}(t) := \mathit{TRUE} \\
\\
\mathit{EnterSection}(t : \mathit{TRAIN},\; s : \mathit{SECTION}) = \\
\quad \mathrm{SELECT} \\
\qquad \mathit{checked}(t) \;\wedge\; \mathit{front}(t) \in \mathit{dom}(\mathit{next}) \;\wedge\; s = \mathit{next}(\mathit{front}(t)) \\
\quad \mathrm{THEN} \\
\qquad \mathit{front}(t) := s \;\;||\;\; \mathit{occp} := \mathit{occp} \cup \{t \mapsto s\} \;\;||\;\; \mathit{checked}(t) := \mathit{FALSE} \\
\quad \mathrm{END} \\
\\
\mathit{LeaveSection}(t : \mathit{TRAIN},\; s : \mathit{SECTION}) = \cdots \\
\\
\mathit{SwitchChangeDiv}(sw, s1, s2 : \mathit{SECTION}) = \\
\quad \mathrm{SELECT} \\
\qquad sw \in \mathit{switch} \;\wedge\; sw \notin \mathit{ran}(\mathit{occp}) \;\wedge \\
\qquad (sw \mapsto s1) \in \mathit{next} \;\wedge\; (sw \mapsto s2) \in \mathit{net} \;\wedge\; s1 \neq s2 \\
\quad \mathrm{THEN} \\
\qquad \mathit{next}(sw) := s2 \\
\quad \mathrm{END} \\
\\
\mathit{SwitchChangeCnv}(sw, s1, s2 : \mathit{SECTION}) = \cdots
\end{array}
$$

Table 2: Some operations of abstract railway.

which is currently connected to successor section $s1$. The effect of the operation is to update the *next* function so that *sw* is connected to $s2$ instead. There is a further constraint on the occurrence of the switch change saying that the *sw* should not be currently occupied since changing an occupied switch has the potential to cause a derailment.

The tool support available for B is used to verify that the operations of a model preserve the invariant. The tool will generate a list of proof obligations and attempt to prove them automatically. Obligations that are not automatically proved are proved semi-automatically where the user guides the proof.

In our system-level model we have defined some observable behaviour of the overall system, but we haven't yet put a boundary around individual components nor defined the interface between the components. Ultimately, the aim is to use this system-level model to derive models for the individual brake controllers for each train and the trackside controllers which should control the switches and send signals to the trains.

4 Refinement

An abstract machine is refined by applying the standard technique of data refinement to its state: an abstraction invariant is used to relate the state variables of the abstract system to those of the refined system and data refinement should hold between correspondingly-named operations in the abstract and refined systems (see, e.g., [2, 5]). If S is a statement that acts on abstract variables, T is a statement that acts on concrete variables, and AI is an abstraction invariant, then we write

$$S \sqsubseteq_{AI} T$$

for 'S is data-refined by T under abstraction invariant AI'. The formulation of $S \sqsubseteq_{AI} T$ is omitted here. Details may be found in [2].

A system M is refined by system N under abstraction invariant AI if N has an operation $N.a$ corresponding to each operation $M.a$, and for each such a:

$$M.a \sqsubseteq_{AI} N.a.$$

As well as having an operation corresponding to each of the abstract operations, a refined system may also have some auxiliary 'internal' operations. These are operations that can change the state of the refined machine but whose occurrence is not observable at the more abstract level. Viewed in terms of the abstract state, such operations should have no effect, that is, they should be data-refinements of the *skip* statement on the abstract state. Each auxiliary operation $N.h$ of refined machine N must satisfy:

$$skip \sqsubseteq_{AI} N.h.$$

The tool support available for B can be used to verify refinement between two models. Proof obligations are generated based on the abstraction invariant and these are discharged automatically and semi-automatically.

First Refinement of Railway

The *Check* operation of the system-level model (Table 2) reads the values of *next* and *occp* directly to determine whether to apply the brakes. An individual train cannot read these values directly but instead receives signals from the trackside controllers. Our first refinement introduces this signalling by introducing an auxiliary operation *SendTrainMsg* (Table 3). The variables of the refined model are all of the variables of the abstract model plus an extra variable *tmsgs* mapping each train to a boolean signal being sent from a trackside controller to the train. $BOOL_\perp$ represents $BOOL$ augmented with the value $\perp$, where $\perp$ models the absence of a signal. When sending a signal to train t, the *SendTrainMsg* operation sets the value of the signal to $TRUE$ if it is ok for the train to continue into the next section and $FALSE$ otherwise. The refined *Check* operation (also shown in Table 3) reads the signal when it is available and sets the brake accordingly. The abstraction invariant used to verify that this model is a valid refinement of the original model is shown in the invariant clause of Table 3.

5 Decomposition to Parallel Sub-systems

[6] describes a technique for composing B machines in a way that corresponds to parallel composition. The interaction between the parallel machines is based on synchronisation (with value-passing) between corresponding operations of the respective machines. There are no state variables shared by the machines. Here, we apply this composition in reverse to break our model into several subsystems. The decomposition structure is illustrated in Figure 1. In this figure, each of the boxes represents a separate machine. The three machines are $TRACK$, representing the trackside behaviour, *TRAINS* representing the train behaviour, and *COMMS* representing the communications layer used to send signals between the trackside and the trains. A line joining two boxes represents a synchronisation between machine operations. For example the *Check* operation of *TRAINS* is synchronised with the *DeliverMsg* operation of *COMMS*. An arrowhead indicates directed value-passing. The lines attached to just one box represent operations that are not involved in synchronisation.

Use of the communications layer models asynchronous signalling between the trackside and the trains. The trackside and the trains also synchronise directly through the *EnterSection* and *LeaveSection* operations. This models the fact that the trackside constrains train behaviour by determining precisely which section a train enters because the connectivity between sections is contained in $TRACK$. The synchronisation also models the fact that the trackside detects a train entering or leaving a section immediately when it happens. The guard of the composite *EnterSection* operation is the conjunction of the guards of the *EnterSection* operation of $TRACK$

REFINEMENT $Railway2$ REFINES $Railway1$
INVARIANT

$tmsgs \in TRAIN \rightarrow BOOL_{\perp} \ \wedge$

$\forall\, t.(t : TRAIN \ \wedge \ tmsg(t) \neq \perp \quad \Rightarrow$

$\quad tmsg(t) = bool(front(t) \in dom(next) \ \wedge \ next(front(t)) \notin ran(occp))$

OPERATIONS

$SendTrainMsg(t : TRAIN) =$
SELECT
$\quad tmsg(t) = \perp$
THEN
$\quad tmsg(t) := bool(front(t) \in dom(next) \ \wedge \ next(front(t)) \notin ran(occp))$
END

$Check(t : TRAIN) =$
SELECT
$\quad tmsg(t) \neq \perp$
THEN
$\quad braking(t) := \neg\, tmsg(t) \ \ || \ \ tmsg(t) := \perp \ \ || \ \ checked(t) := TRUE$
END

Table 3: Parts of first refinement of railway.

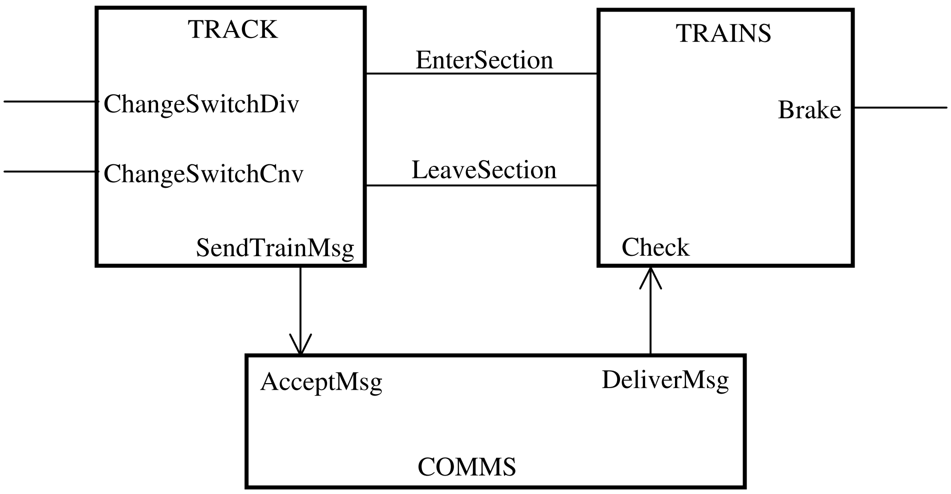

Figure 1: Decomposition Structure.

and the *EnterSection* operation of *TRAINS*, so that the composite operation is constrained by the operation in both machines. Occurrence of *EnterSection* in the *TRAINS* machine models a train actually entering a section, while occurrence of *EnterSection* in the *TRACK* machine models the detection of this by the trackside.

The variables of the three machines are formed by partitioning the variables of the previous refinement. For example, the *tmsg* variable is placed in the *COMMS* machine, while the *braking* variable is placed in the *TRAINS* machine. A copy of the occupancy variables is placed in both *TRAINS* and *TRACKS*. These are not shared variables but instead they are separate variables which will always contain the same value since they are updated simultaneously with the same values by the shared *EnterSection* and *LeaveSection* operations. The values of the occupancy variables in the *TRAINS* machine models the actual positions of trains, while the values in the *TRACK* machines models the trackside's awareness of the positions of the trains.

The decomposition of the railway is modelled in B by defining a B machine which glues together the three machines *TRACK*, *TRAINS* and *COMMS*. This gluing machine is declared to be a refinement of the previous model. This is partly shown in Table 4 where the B *includes* clause is used to include each of the three machines in refinement *Railway*3. Table 4 shows how the composite *Check* operation is formed by sequentially composing a call to the *DeliverMsg* operation of *COMMS* with the *Check* operation of *TRAINS*. The call to *DeliverMsg* returns a boolean value which is assigned to local variable b, and b is then passed on as an input parameter for the call to *Check* in *TRAINS*. The semantics of the B notation mean that the guard of the composite *Check* operation is the conjunction of the guard of *comms.AcceptMsg* and *trains.Check*, so, as well as modelling value passing, this construction also models synchronisation.

REFINEMENT $\mathit{Railway}3$ REFINES $\mathit{Railway}2$
INCLUDES $\mathit{track}.\mathit{TRACK}, \mathit{trains}.\mathit{TRAINS}, \mathit{comms}.\mathit{COMMS}$
OPERATIONS

$\mathit{Check}(t : \mathit{TRAIN}) =$
  VAR $b \in \mathit{BOOL}$ WHERE
    $b \longleftarrow \mathit{comms}.\mathit{DeliverMsg}(t);$
    $\mathit{trains}.\mathit{Check}(t, b)$
  END

Table 4: Gluing the components together.

An important feature of this form of parallel composition is that it is monotonic with respect to refinement, i.e., the sub-systems can be refined independently and the composition of the resulting refinements is a refinement of the original model. The validity of the composition in shown in [6].

Fault Tolerance

The development presented above assumes that the communication mechanisms between trains and trackside is fault-free. In the real system, signals can fail and fault-tolerant techniques need to be used to deal with this. Faults and fault-tolerance also need to be modelled in the formal refinement process. For the communications mechanism, faults can be introduced already at the first refinement step. The body of the *SendTrainMsg* (Table 3) could be specified so that it nondeterministically sends a signal or fails to send a signal as follows:

$$\mathit{tmsg}(t) := \mathit{bool}(\ \cdots\ )\ \ [\!]\ \ \mathit{tmsg}(t) := \perp .$$

Here, $[\!]$ is the nondeterministic choice operator of B.

Potential loss of a signal is dealt with by having a timeout mechanism on a train that applies the brakes if no signal arrives within a certain time. This can be modelled by adding a *Timeout* operation that applies the brake if there is no signal. Rather than explicitly modelling the timing delay, it is sufficient at this level to constrain the model so that either the *Check* operation or the *Timeout* operation must occur before the *EnterSection* operation occurs. An example of incorporating fault tolerance into formal refinement may be found in [7] which develops a controller for a steam boiler. In this, a safe mode at the system-level is refined to a normal mode or a degraded mode where deals with sensor failure.

6 Introducing Switch Positions

In our original system-level model, switch positions were abstracted into the *next* function. While this makes modelling and reasoning easier, it does not model the physical switches in sufficient detail to be able to model controllers for them. One of the first steps in refining the *TRACK* machine is to introduce explicit switch positions. Some of the formal details for this refinement are shown in Table 5. We include it because it represents a good example of introducing complexity in a refinement step in order to represent the real system more closely. Rather than explaining all of the B notation used in Table 5 in detail, we will explain the refinement informally.

REFINEMENT $TRACK2$ REFINES $TRACK$
SETS $POS = \{dir\_pos, dev\_pos\}$
CONSTANTS dir, dev
PROPERTIES $dir \in switch \to SECTION \;\wedge\; dev \in switch \to SECTION$
VARIABLES pos
INVARIANT

$$
\begin{aligned}
& pos \in switch \to POS \;\wedge \\
& next = (div\_switch \lhd net \rhd cnv\_switch) \\
& \quad \cup \{\, sw \mapsto s \mid sw \in div\_switch \;\wedge\; pos(sw) = dir\_pos \;\wedge\; dir(sw) = s \,\} \\
& \quad \cup \{\, sw \mapsto s \mid sw \in div\_switch \;\wedge\; pos(sw) = dev\_pos \;\wedge\; dev(sw) = s \,\} \\
& \quad \cup \{\, s \mapsto sw \mid sw \in cnv\_switch \;\wedge\; pos(sw) = dir\_pos \;\wedge\; dir(sw) = s \,\} \\
& \quad \cup \{\, s \mapsto sw \mid sw \in cnv\_switch \;\wedge\; pos(sw) = dev\_pos \;\wedge\; dev(sw) = s \,\}
\end{aligned}
$$

$SwitchChangeDiv(sw, s1, s2) =$
PRE $sw \in SECTION \;\wedge\; s1 \in SECTION \;\wedge\; s2 \in SECTION$ THEN
SELECT
$sw \in switch \;\wedge\; sw \notin ran(occp) \;\wedge$
$(sw \mapsto s1) \in dir \;\wedge\; (sw \mapsto s2) \in dev \;\wedge\; pos(sw) = dir\_pos$
THEN
$pos(sw) := dev\_pos$
WHEN
$sw \in switch \;\wedge\; sw \notin ran(occp) \;\wedge$
$(sw \mapsto s1) \in dev \;\wedge\; (sw \mapsto s2) \in dir \;\wedge\; pos(sw) = dev\_pos$
THEN
$pos(sw) := dir\_pos$
END
END

Table 5: Refined switch change operation.

A switch will be in one of two positions, *direct* or *deviate*. In the case of a divergent switch, one of the switch's successors will be linked to the direct position while the other will be linked to the deviate position. In Table 5, this is modelled by the constant functions *dir* and *dev* which map each switch to its direct and deviate successor respectively. The *pos* variable maps each switch to its current position. The abstract variable *next* is replaced by *pos* and the invariant clause of Table 5 describes the relationship between *next* and *pos*. The invariant specifies that *next* is formed by taking the static connectivity *net* and replacing the connectivity of switches in *net* by mappings derived from *pos*. There are four cases of switch mappings to be dealt with: whether a switch is convergent (*cnv_switch*) or divergent (*div_switch*) and whether the current position is direct or deviate. The refined *SwitchChangeDiv* operation is also shown in Table 5. Here a single change to the abstract *next* function is replaced by two cases of updating the *pos* variable depending on whether the switch is currently in the direct or deviate position.

Note that we have simplified the switch position refinement for this paper. In the real development the switch change operations are refined to be non-atomic, and a switch has a third position representing the transition form direct to deviate. This is achieved by introducing separate operations modelling initiation of switch changes. When a switch is in transition it is not available to be occupied.

7 Further Refinements

We briefly describe the further refinement steps involved in the development. The track sections are partitioned into sectors and each sector has its own trackside controller. Each sector needs to communicate occupancy information about its sections to its neighbouring sectors. A communications mechanism is introduced in a refinement step to model this, similar to the way in which we introduced the communication between the trains and the trackside. Again, a timeout mechanism should be used to help tolerate communications failures. The refined *TRACKS* machine is partitioned into a machine representing a collection of sectors and a machine representing the communications mechanism between sectors. Each sector is in turn partitioned into models of the physical switches and models of the controllers for each sector. A sector controller sends signals to trains, communicates with other sectors and controls individual switches.

The *TRAINS* machine is also refined so that it is partitioned into models of the physical occupancy and braking status of the trains and models of the on-board controller for trains. An on-board controller receives signals from sectors and decides when the brakes should be applied.

8 Concluding

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

Specifying the system-level model does require skill in deciding on the appropriate abstractions, what aspects of behaviour need to be modelled and what aspects can be left out of the model. The benefit of the system-level model is that it is easier to understand and reason about the behaviour of the system as a whole. Providing guidelines on constructing system models is one of the aims of this on-going work.

The complexity of the entities and the relationships between them is handled through the use of refinement, which allows the complexity to be introduced and reasoned about in steps. We made use of refinement for two main purposes, to introduce communications mechanisms leading to system partition and to replace abstract structures by more concrete realisations (such as replacing *next* by *pos*). Deriving abstraction invariants for refinement requires skill to ensure it is sufficiently strong, but also amenable to verification.

The idea of incorporating behaviour of the system under control is also found in [7] where a single controller for a steam boiler is derived through refinement from a system-level model incorporating behaviour of the boiler. In the railway development we have developed multiple distributed controllers. The formal model developed in [9] of a railway system using the Raise method is also a system-level model not unlike our system-level model. Here we have presented a more comprehensive way of refining the model to derive models for the individual controllers than found in [9].

The way in which we deal with real-time constraints in the abstract models is to prevent physical events from occurring unless some control event has occurred (e.g., the *EnterSection* operation in the system-level model is guarded by the $checked(t)$ flag). This needs to be realised by ensuring that the occurrence of the control event is timely. So far, we have not introduced explicit reasoning about time to the formal development. We are guided by wanting a pragmatic approach that fits in easily with the B Method and can feasibly be used in an industrial setting.

Acknowledgements

Thanks to Jerome Falampin of Siemens Transportation Systems for explaining important aspects of railway control to me.

References

[1] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *Second International B Conference*, April 1998.

[2] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[3] J.R. Abrial, E. Börger, and H. Langmaack. *Formal Methods for Industrial Applications – Specifying and Programming the Steam Boiler Control*, volume LNCS 1165. Springer, 1996.

[4] B-Core. B Toolkit. www.b-core.com.

[5] M.J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, September 1996.

[6] M.J. Butler. An approach to the design of distributed systems with B AMN. In J.P. Bowen and M.G. Hinchey, editors, *10th International Conference of Z Users (ZUM'97)*, volume LNCS 1212, pages 223–241. Springer–Verlag, 1997.

[7] M.J. Butler, E. Sekerinski, and K. Sere. An action system approach to the steam boiler problem. In *[3]*, 1996.

[8] ClearSy. AtelierB. www.atelierb.societe.com.

[9] 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, *FM'99–Formal Methods, Volume II*, volume LNCS 1709. Springer, September 1999.