Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 280 (2011) 57–68

www.elsevier.com/locate/entcs

Development of Control Systems Guided by Models of their Environment

Simon Hudon[1] Thai Son Hoang[2,3]

Department of Computer Science
Swiss Federal Institute of Technology Zurich (ETH-Zurich)
Zurich, Switzerland

Abstract

Event-B is a formal method that allows one to develop various kinds of systems including discrete control systems. However, it is lacking a systematic approach for developing this type of systems and it hinders the applicability of Event-B. Our contribution is such an approach and it is presented in this paper. Our proposed method focuses on a set of elements that should be captured by the formal model and prescribes an order in which they should be introduced. The key aspect of our approach is to first model the required behaviour of the environment, and then to introduce the controller to appropriately influence the environment. It has the advantage that every step of such a development is dictated by the information available so far, including the requirements. We argue that having a clear development strategy early in the design process will assist the developers in producing high-quality models of the future software systems.

Keywords: Event-B, formal modelling, refinement, development strategy, system development.

1 Introduction

Event-B [2] is a modelling method for discrete transition systems which are correct-by-construction. Its applications range from sequential programs, concurrent programs to distributed systems. In particular, Event-B is one of the few modelling methods having control systems within its scope. More importantly, the development of such systems in Event-B includes the model of the environment which is a necessity for the assurance about the correctness of the future systems.

As a result, developing systems in Event-B is a complex task involving the management of several aspects of the systems, including the environment. A central aspect of Event-B is the use of step-wise refinement to reduce the complexity of system modelling. Abrial suggested in [2] that in practice, before engaging in the

[1] Email: shudon@student.ethz.ch

[2] Email: htson@inf.ethz.ch

[3] This author is sponsored by the DEPLOY project (http://www.deploy-project.eu).

doi:10.1016/j.entcs.2011.11.018

actual modelling task, developers should design a *refinement strategy* specifying for each refinement step which details will be introduced into the model. However, coming up with a good and helpful refinement strategy, which aids the system development, is a challenging task. Guidelines are needed in order to design such a refinement strategy.

For developing control systems, Butler has proposed modelling guidelines in what is known as the *cookbook* [3]. An application of the cookbook for developing a cruise control system is reported in [7]. Our approach and the cookbook's differ mainly in the order in which various ingredients are introduced during developments. More comparisons are in Sect. 5.

In the present paper, we propose our *development strategy* which differs from that of the cookbook in some key aspects (see Sect. 5). We start in Sect. 2 by offering a summary of the Event-B notation; in Sect. 3, we explain our strategy and apply it to a control problem in Sect. 4. Finally, we discuss our results in Sect. 5.

2 The Event-B Modelling Method

Event-B is supported by a specialized notation for abstract machines, the central object of the development method. It supports both the formulation of formal specifications and their refinement. We give a brief overview of some essential aspects of Event-B in this Section. For a full details of Event-B, we refer our readers to [2].

Specification

In the Event-B notation, a machine is characterized by its state space modelled by some variables v and its transitions modelled by some events. The state variables v are constrained by some invariant $I(v)$. An event evt has the following form: evt $\widehat{=}$ **any** p **where** $G(p,v)$ **then** $S(p,v,v')$ **end** , where p is a list of parameters, $G(p,v)$ specifies the enabled condition, and $S(p,v,v')$ is the action. A dedicated event without parameters and guards is used as the initialisation.

Action $S(p,v,v')$ contains several assignments that are supposed to happen simultaneously. Each assignment can take one of the three forms: $v := E(p,v)$, $v :\in E(p,v)$, or $v :| P(p,v,v')$. While the first form deterministically assigns the value of expression $E(p,v)$ to v, the second form non-deterministically assigns to v some value from $E(p,v)$. The last assignment form is the most general. It assigns to v some value satisfying the before-after predicate $P(p,v,v')$.

A machine is consistent if its invariant holds at any given time. In practice, this is guaranteed by proving that the invariant is established by the initialisation and maintained by all its events.

Refinement

Refinement is a well-known technique for reducing the complexity of developing formal models. One starts with an abstract machine capturing some central aspect of the system, and subsequently refines the machine by adding more concrete details

to the model. When refining a machine, it is possible to introduce new variables and new events.

Consistency has to be proved between a concrete machine and its abstract machine. In practice, this is done on a per event basis. An event of the concrete machine is a refinement of an abstract event if the guard is *strengthened* and the action of the concrete event can be *simulated* by the action of the abstract event.

Tool Support

Developing in Event-B is supported by the Rodin platform [1]. This is an industrial tool-set including supports for constructing Event-B models, proving their consistency, and animating them.

3 Development Strategy

Despite being a powerful modelling method, Event-B lacks a systematic approach for developing different types of systems. We suggest here some guidelines, which we call a *development strategy*, for developing control systems together with a model of their environment. The environment and the controller communicate in a bi-directional fashion: the controller receives input from the environment via various *sensors*; reciprocally, the controller produces output to change the environment via various *actuators*. This interaction is illustrated in Figure 1.

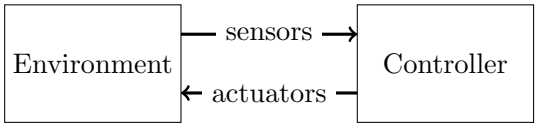

Fig. 1. Interaction between the Environment and the Controller

Our development strategy contains four different stages. Note that each stage can be developed through several refinements.

Stage 1 To model the environments as it should behave.

Stage 2 To model the actuators to command the changes in the environment.

Stage 3 To model the sensors together with the controller.

Stage 4 To model some appropriate scheduler for the controller.

Stage 1 aims at describing an environment with the desirable properties, based on the requirements document. At this stage, we omit the controller completely, focus on global safety properties and how the physical components should work together to achieve such properties.

In Stage 2, actuators are introduced as means by which the controller will affect the environment, such that the physical components interact correctly with each other. This, in turns, puts some constraints on how the actuators can be set.

Up until Stage 2, all the control of the environment via actuators is done with perfect information of the whole system. Since this is unrealistic, Stage 3 aims at

interposing sensors between observed components and the controller. This enforces an appropriate specification for the controller.

At Stage 4, we can introduce a scheduling strategy to the controller. The purpose is to optimise its efficiency.

Benefits of the Approach

Desirable safety properties of the systems are modelled earlier in the development in Stage 1. We can rely on refinement for the preservation of these properties during the development.

The safety properties captured in Stage 1 serve as guidelines to introduce as needed the actuators (Stage 2) and, in turn, the sensors and the controller (Stage 3). This way, the controller and its interface are introduced as a *solution* to the *problem* of maintaining safety in the system.

The role of the actuator is therefore to force the environment to behave as we modelled it. Shortly after, the introduction of the sensors will answer the question: "on the basis of what information are the actuators acting?" Introducing the sensors before the actuators then seems upside down: the actuators are the reason we need information about the state of the environment. It seems reasonable to find out what information it needs before we arrange for gathering said information.

By deciding to introduce scheduling at the end of the development, we facilitate the design of the controller: the models are not poluted by scheduling details. We can have separate models corresponding to different scheduling algorithms.

4 A Signal Control System

In this section, we first present a requirements document of a signal control systems, then subsequently describe our formal development, applying our proposed development strategy [4] .

4.1 Requirements Document

Our aim is to develop a signal control system at a particular train station. An overview of the system can be seen in Fig. 2. In the following, we give a set of plausible requirements for the management of a train station. They have been tailored to let us solve some interesting problems. Some realism has been abandoned for the sake of simplicity. We are trying to solve only a few problems in this paper.

Our first set of requirements concern the trains and the topology of the network.

ENV0 The station contains a number of *platforms* in between an *entry block* and an *exit block*.

ENV1 A train occupies *no more than one block*.

[4] The model is developed using the Rodin platform and is available on-line at http://deploy-eprints.ecs.soton.ac.uk/308/.

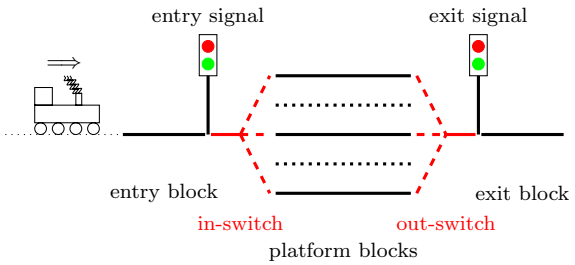

Fig. 2. A signal control system

ENV2 The track is *one-way*, i.e. the train enters the station via the entry block and leaves the station via the exit block.

The next requirements concern the switches located at the two ends of the stations.

ENV3 There are two switches connecting the entry and exit block to some platforms, called in-switch and out-switch accordingly.

ENV4 A train at entry block can only enter some platform block if the in-switch is set to that particular block. Similarly for the out-switch.

We make an assumption that the switch changes its position instantaneously.

The most important property of the system concerns safety: the system must guarantee that trains never collide. This is ensured by precluding the simultaneous presence of two trains on the same block.

SAF5 Two trains cannot be on the same block.

In this simplified example, we are not interested in proving that trains will not derail. Doing so would complicate the development and divert the attention from the illustration of our approach.

Two (light) signals are installed at the two ends of the station, called entry signal and exit signal respectively.

ENV6 There are two signals which are either red or green.

ENV7 Trains are assumed to stop at red signals.

The controller receives input from various sensors and output its commands via actuators.

ENV8 There are sensors detecting whether a block is occupied.

ENV9 There are sensors detecting the status of the signals.

ENV10 The sensors reflect the current status of the corresponding components [5]
.

We design a controller for changing the switch positions connecting to a certain platform, and changing the signal from red to green. The signals automatically change from green to red when a train passes by.

[5] This assumption does not effect the applicability of our approach.

ENV11 For each signal, there is an actuator for the controller to command the signal to turn from red to green.

ENV12 The signals change from green to red when a train passes by.

ENV13 For each switch, there is an actuator for the controller to command the switch to change to a specific platform.

4.2 Stage 1. The Model of the Environment

In the first stage, we build a model of the environment. We proceed step-by-step by introducing the details of the system in the following sequence: the blocks, the switches, the signals and the trains. We adopt the following conventions for our Event-B models to have a clear distinction between different modelling elements.

- The environment variables and events are in capital letters.
- The controller variables and events are in lower cases prefixed by *ctr*.
- The sensor variables are in lower cases prefixed by *snsr*.
- The actuator variables are in lower cases prefixed by *act*.

The Blocks. We specify the set of blocks ($BLOCK$) containing an entry (ENT), an exit (EXT) and a set of platform blocks ($PLFS$). This corresponds to our requirement **ENV0**. In the initial model, variable OCC is used to record the set of occupied blocks. Initially, OCC is assigned $\varnothing$, the empty set. There are four different events, namely ARRIVE, MOVE_IN, MOVE_OUT, and LEAVE, to model different cases on how the status of a block can be changed.

ARRIVE : ENT becomes occupied (because of an arriving train).

MOVE_IN : ENT becomes unoccupied and a platform becomes occupied (because of a train moving into the station).

MOVE_OUT : EXT becomes occupied and a platform becomes unoccupied (because of a train moving out of the station).

LEAVE : EXT becomes unoccupied (because of a leaving train).

Because of the symmetry between these events, we only present the events MOVE_IN and ARRIVE, and their subsequent refinements.

ARRIVE
when
$ENT \notin OCC$
then
$OCC := OCC \cup \{ENT\}$
end

MOVE_IN
any p **where**
$p \notin OCC$
$ENT \in OCC$
then
$OCC := (OCC \cup \{p\}) \setminus \{ENT\}$
end

The Switches. In this refinement step, we introduce the variables IN_SW and OUT_SW to model the two switches located at the two end of the station. The status each switch represents which platform block they are connected to. This corresponds to our requirement **ENV3**. Initially, the switches are set arbitrarily to any platform.

We only need to adjust event MOVE_IN. Its parameter p is instantiated with the platform the in-switch is connected to. This reflects requirement **ENV4**.

MOVE_IN
when
$IN\_SW \in PLFS \setminus OCC$
$ENT \in OCC$
then
$OCC := (OCC \cup \{IN\_SW\}) \setminus \{ENT\}$
end

The step is finalised by providing a means to change the switches. Events TURN_IN_SW and TURN_OUT_SW are introduced and we leave the choice arbitrary. We focus on TURN_IN_SW and its refinements.

TURN_IN_SW
begin
$IN\_SW :\in PLFS$
end

The Signals. In this refinement step, we introduce the signals ENT_SGN and EXT_SGN located at the two ends of the station. The signals are either RED (meaning that passage is *forbidden*) or GRN (meaning that passage is *allowed*). This corresponds to requirement **ENV6**. Initially, both signals are RED.

We refine MOVE_IN event accordingly, by refining its guards using ENT_SGN instead of referring directly to the status of ENT block. This also reflects the requirement that trains obey the signals (**ENV7**).

However, this guard substitution is only valid if it constitutes a strengthening which we ensure by introducing the following invariant.

invariants:
inv2_0 : $ENT\_SGN = GRN \Rightarrow IN\_SW \notin OCC$

To preserve **inv2_0**, we make sure that the signal becomes red as soon as the platform designated by the in-switch becomes occupied (**ENV12**). Also for the sake of preserving **inv2_0**, we strengthen the guard of TURN_IN_SW accordingly.

MOVE_IN
when
$ENT\_SGN = GRN$
$ENT \in OCC$
then
$OCC := (OCC \cup \{IN\_SW\}) \setminus \{ENT\}$
$ENT\_SGN := RED$
end

TURN_IN_SW
when
$ENT\_SGN = RED$
then
$IN\_SW :\in PLFS$
end

There are two new events to change the status of the signals, namely ALLOW_ENTRY and ALLOW_EXIT. We show here ALLOW_ENTRY only, taking into account **inv2_0**.

ALLOW_ENTRY
when
$IN\_SW \notin OCC$
then
$ENT\_SGN := GRN$
end

The Trains. In the last refinement of the environment model, we introduce the trains into the system. The safety properties concerning the trains all concern their position so this is a good candidate for a new variable. POS is thus introduced to map each train to the only block where it is located (as stated by **inv3_0**), consistently with **ENV1**. To rule out the possibility of collisions, i.e. to enforce **SAF5**, we can now introduce **inv3_1** which states that each train is alone on its block [6]. Finally, for the sake of consistency with the variable OCC, we introduce **inv3_2** so that only trains can occupy a block [7].

invariants:
inv3_0 : $POS \in TRAIN \nrightarrow BLOCK$
inv3_1 : $\forall t_1, t_2 \cdot t_1 \in \mathrm{dom}(POS) \wedge t_2 \in \mathrm{dom}(POS) \ \wedge \ t_1 \neq t_2 \ \Rightarrow \ POS(t_1) \neq POS(t_2)$
inv3_2 : $\mathrm{ran}(POS) = OCC$

In this model, events such as TURN_IN_SW and ALLOW_ENTRY stay unchanged since it does not directly interact with train positions. We refine events MOVE_IN and ARRIVE accordingly to include how the train position are updated.

MOVE_IN
any t **where**
$ENT\_SGN = GRN$
$t \in \mathrm{dom}(POS)$
$POS(t) = ENT$
then
$OCC := (OCC \cup \{IN\_SW\}) \setminus \{ENT\}$
$ENT\_SGN := RED$
$POS(t) := IN\_SW$
end

ARRIVE
any t **where**
$ENT \notin OCC$
$t \notin \mathrm{dom}(POS)$
then
$OCC := OCC \cup \{ENT\}$
$POS(t) := ENT$
end

Finally, **inv3_2** enables us to rewrite the guard of MOVE_IN as shown [8].

4.3 Stage 2. The Actuators

At the end of the first stage we have an idealised model of the environment specifying how physical components should be working together. We introduce some actuators, i.e. output of the future controller, to commands the adaptation of the state of the environment component, in such a way that the normal behavior of the environment is coerced into the modelled behavior.

The *switch actuators* are used to send commands to the switches to change to

[6] Combining **inv3_0** and **inv3_1**, POS is an injective function, i.e. $POS \in TRAIN \nrightarrow BLOCK$.

[7] In principle, we can eliminate OCC from the model. However, we refrain from doing so, since OCC captures an useful abstraction that we can still make use of throughout the development.

[8] Note that the last two guards of MOVE_IN can be rewritten as $t \mapsto ENT \in POS$. However, we prefer to use two separated clauses specifying that (1) t is a monitored train in the system and (2) t is at position ENT.

a specific platform (**ENV13**). We focus on the actuator of the in-switch. Two new variables *act_in_sw* and *act_in_sw_plf* are used to model the actuator: the former is a boolean to indicate whether there is a pending command for the device, the latter specifies which platform the switch should change to.

Similarly, the *signal actuators* are used for sending commands to set the signals to *GRN* (**ENV11**). The signal actuators are modelled as one boolean each, *act_ent_sgn* for the entry signal and *act_ext_sgn* for the exit signal, respectively.

Event TURN_IN_SW is refined accordingly using the command from the actuator. The concrete guard specifies that there is a command from the controller for changing the in-switch. As a result, the in-switch is set to the specific platform as commanded. The actuator is reset after the switch changes.

(abs)TURN_IN_SW
when
$ENT\_SGN = RED$
then
$IN\_SW :\in PLFS$
end

(cnr)TURN_IN_SW
when
$act\_in\_sw = \text{TRUE}$
then
$IN\_SW := act\_in\_sw\_plf$
$act\_in\_sw := \text{FALSE}$
end

Event ALLOW_ENTRY is refined similarly. We now introduce invariants to make sure that the substitution of guards is indeed a strengthening.

invariants:
inv4_0 : $act\_in\_sw = \text{TRUE} \Rightarrow ENT\_SGN = RED$
inv4_1 : $act\_ent\_sgn = \text{TRUE} \Rightarrow IN\_SW \notin OCC$

Finally, we create new controller events responsible for sending commands via the actuators to the in-switch (ctrl_trigger_in_sw) and to the entry signal (ctr_chg_ent_sgn).

ctrl_trigger_in_sw
any p **where**
$act\_in\_sw = \text{FALSE}$
$ENT\_SGN = RED$
$act\_ent\_sgn = \text{FALSE}$
$p \in PLFS$
then
$act\_in\_sw := \text{TRUE}$
$act\_in\_sw\_plf := p$
end

ctr_chg_ent_sgn
when
$act\_ent\_sgn = \text{FALSE}$
$IN\_SW \notin OCC$
$act\_in\_sw = \text{FALSE}$
then
$act\_ent\_sgn := \text{TRUE}$
end

Event ctrl_trigger_in_sw specifies that the actuator *act_in_sw* can be set to instruct the switch to change to any platform p, when the entry signal is *RED* and both actuators *act_in_sw* and *act_ent_sgn* are unset. Event ctr_chg_ent_sgn models the fact that the actuator *act_ent_sgn* can be set to command the entry signal to change to *GRN*, when the in-switch point to unoccupied platform and both actuators *act_in_sw* and *act_ent_sgn* are unset. Notice that the guards of these events guarantee that the newly introduced invariants are maintained.

4.4 Stage 3. The Sensors and the Controller

We are now ready to introduce the sensors together with the assumption that they reflect the status of the actual physical components. This is straightforward and we add variables for the sensors: *snsr_occ*, *snsr_ent_sgn* and *snsr_ext_sgn* corresponding respectively to the sensors of the blocks, the entry signal and the exit signal (**ENV8**, **ENV9**). These new variables are glued with the old variables using invariants, such as $snsr\_occ = OCC$, corresponding to requirement **ENV10**.

Within the controller events such as ctrl_trigger_out_sw references to the status of a physical component such as OCC are replaced by the corresponding sensor, in this case *snsr_occ*.

Finally, since IN_SW is used in the guard of ctr_chg_ent_sgn, the controller needs to know the status of the in-switch when sending the command for changing the entry signal. The controller keeps a copy of status of the in-switch with its variable *ctrl_in_sw*. Note that variable *ctrl_in_sw* does not necessarily reflect the current value of IN_SW. Indeed, we only need them to be the same when there is no actuator command for the in-switch. *ctrl_in_sw* is updated when the controller commands the corresponding switch to change with event ctrl_trigger_in_sw.

4.5 Stage 4. Scheduling

At the end of Stage 3, we have a model of the signal control system including its working environment which guarantees to satisfy our safety requirement. We can then impose extra scheduling algorithm for our controller for optimising its execution. In our Event-B model, it is done by merely strengthening guards of the controller events. As an example, we show here the optimisation for event ctrl_trigger_in_sw so that the in-switch

- changes only to a new free platform, i.e. $p \notin snsr\_occ \wedge p \neq ctrl\_in\_sw$, and
- only when the entry block is occupied, i.e. $ENT \in snsr\_occ$.

For more complex scheduling algorithms, one can adopt a strategy by going through an iteration: environment – actuators – sensors and controller.

(i) To describe the scheduling algorithm in terms of the environment.

(ii) To specify how such algorithm can be achieved in terms of actuators.

(iii) To design the sensors and controller to realise the algorithm.

5 Conclusion

We have presented our development strategy for developing control systems together with a model of their environment. Our strategy starts with the modelling of the environment, followed by the introduction of the actuators, before the controller and sensors are modelled. Finally, further scheduling details are imposed on the controller as an optimisation step for the system. Applying our development strategy reduces the difficulty in modelling this type of systems, results in models which

are easy to understand and verify. We illustrate our approach by developing a simplified signal control system. Even though there are not yet any code generators for Event-B, the controller variables and events in our final model are concrete and clear enough, and can be used as a software low level design.

Our development strategy is initially inspired by the development of an elevator system by Laurent Voisin, which has been used as a student project for a course on Event-B at ETH Zurich. We have applied the approach to several systems of this type, including a re-development of "Cars on a bridge" example from Abrial [2, Chapter 2]. Our approach is fundamentally different from the *inside-out* approach taken by Abrial. In contrast to our approach, Abrial starts by first modelling the controller and the environment is introduced after. Even though both approaches are possible for developing this type of systems, our *outside-in* approach is more constructive: instead of defining a controller and then proving that it fits the environment, we use the requirements to deduce constraints that the controller must fulfill and we go on to build it accordingly.

Our development strategy is similar to Butler's [3] in that it focuses initially on a model of the environment. The two approaches differ mainly by the order of the introduction of the actuators and the sensors; in our approach the actuators come before the sensors. In our opinion, this points to a correct design more clearly. This is influenced by our *backward* reasoning: we want to deduce the design of our controller and its input from constraint imposed on its output. We believe that this approach is simpler and gives stronger guidance for the design, similar to the reasoning using weakest-precondition [4].

The validation of control systems have been studied using other formal methods. Hansen validated a railway interlocking model using VDM [5]. However, the paper only establishes a model of the environment without the controller. Haxthausen and Peleska presented an approach using RAISE for developing a distributed railway control system [6]. Their approach consists of two stages. In their first stage, the model of the environment and controllers are developed globally together. Their second stage focuses on the design of a distributed controller corresponding to the model in the first stage. Our development strategy can be seen as a guideline for developing the model in their first stage.

One aspect that has not been captured in our example is the assumptions. Typically, they concern the speed of communication and response of the controller. It can be shown that using our development strategy, these assumptions arise naturally during the formal developments which otherwise will be difficult to find *a priori*. Furthermore, we have focused on the development of a system with some critical safety properties. Developing systems satisfying liveness properties, e.g. all trains must eventually leave the station, would require additional modelling guidelines.

References

[1] J-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.

[3] Michael Butler. Towards a Cookbook for Modelling and Refinement of Control Problems. Working paper, http://deploy-eprints.ecs.soton.ac.uk/108/, May 2009.

[4] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[5] Kirsten Mark Hansen. Validation of a railway interlocking model. In Maurice Naftalin, B. Tim Denvir, and Miquel Bertran, editors, *FME*, volume 873 of *LNCS*, pages 582–601. Springer, 1994.

[6] Anne Elisabeth Haxthausen and Jan Peleska. Formal development and verification of a distributed railway control system. *IEEE Trans. Software Eng.*, 26(8):687–701, 2000.

[7] S. Yeganefard, M. Butler, and A. Rezazadeh. Evaluation of a guideline by formal modelling of cruise control system in Event-B. In *Proceedings of NFM 2010*, pages 182–191, 2010.