The Landing Gear Case Study in Hybrid Event-B

Richard Banach

School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk

Abstract. A case study problem based on a set of aircraft landing gear is examined in Hybrid Event-B (an extension of Event-B that includes provision for continuously varying behaviour as well as the usual discrete changes of state). Although tool support for Hybrid Event-B is currently lacking, the complexity of the case study provides a valuable challenge for the expressivity and modelling capabilities of the formalism. The size of the case study, and in particular, the number of overtly independent subcomponents that the problem domain contains, both significantly exercise the multi-machine and coordination capabilities of Hybrid Event-B, requiring the use of novel coordination mechanisms.

1 Introduction

This paper reports on a treatment of the landing gear case study using Hybrid Event-B. Hybrid Event-B [4] is an extension of the well known Event-B framework, in which continuously varying state evolution, along with the usual discrete changes of state, is admitted. There is a *prima facie* case for attempting such an exercise using Hybrid Event-B, since aircraft systems are replete with interactions between physical law and the engineering artifacts that are intended to ensure appropriate aircraft behaviour. In the case of landing gear systems specifically, a good idea of the real complexity of such systems can be gained from Chapter 13 of [16].

Given that landing gear is predominantly controlled by hydraulic systems (see Chapter 12 of [16]), it might be imagined that the requirements for the present case study [6], would feature relevant physical properties quite extensively. Hybrid Event-B would be ideally suited to describe the interactions between these and the control system — for example on the basis of the theory and models detailed in [10,1,11]. However, it is clear that the requirements in [6] have been heavily slanted to remove such aspects almost completely, presumably because the overwhelming majority of tools in the verification field would not be capable of addressing the requisite continuous aspects. Instead, the relevant properties are reduced to constants (perhaps accompanied by margins of variability) that delimit the *duration* of various physical processes, these being relevant to a treatment centred on discrete control events. Such an approach reduces the modelling workload, but the penalty paid for it is the loss of the ability to *justify* the values of these constants during the verification process, whether this be on the basis of deeper theory or of values obtained from lower level phenomenological models.

Despite this reservation, a small number of simple continuous behaviours are left within the requirements in [6], these being confined to simple linear behaviours of some parts of the physical apparatus. Yet, these are enough to demonstrate many essential capabilities of the Hybrid Event-B formalism in dealing with continuous phenomena and their interaction with discrete events.

The reduced workload of the restricted requirements was in fact welcome, since the limited resources available for the present work meant that a treatment including all failure modes could not be included. However, the nominal regime study that is presented here is sufficient to bring out the main benefits of the approach, and some comments on the failure cases are included in the latter parts of this paper.

Since there is presently no specific tool support for Hybrid Event-B, our case study is primarily an exploration of modelling capabilities. As explained below, a major element of this is the challenge of modelling physically separate components in separate machines, and of interconnecting all these machines in ways appropriate to the domain, all supported by relevant invariants. This requires novel machine interconnection mechanisms, introduced for pure Event-B in [2]. The suitability of proposals for such mechanisms can only be tested convincingly in the context of independently conceived substantial case studies like this one, so it is gratifying that the mechanisms exercised here fare well in the face of the complexities of the given requirements.

The rest of this paper is as follows. Section 2 briefly overviews the landing gear requirements. Section 3 gives an overview of Hybrid Event-B, while Section 4 covers the case of multiple machines and our modelling strategy for complex systems. A description of our development appears in Section 5. Section 6 summarises the lessons learned from this exercise and concludes.

2 Landing Gear Overview

The landing gear case study is presented in [6]. Here we give the gist of it, focusing on features of most interest to the Hybrid Event-B treatment. Fig. 1, reproduced from [6], shows the architecture of the system.

The sole human input to the system is the pilot handle: when pulled up it instructs the gear to retract, and when pulled down it instructs the gear to extend. The signal from the handle is fed both to the (replicated) computer system and to the analogical switch, the latter being an analogue device that gatekeeps powerup to the hydraulic system, to prevent inappropriate gear movement even in the case of computer malfunction. In a full treatment, including faulty behaviour, there are further inputs from the sensors, which can behave in an autonomous manner to introduce faults. But in our purely nominal treatment, sensor behaviour is a deterministic consequence of other actions, so does not constitute independent influence from the environment. A further point concerns the shock absorber sensors, which are modelled using a guard rather than as inputs. The relevant issue is discussed at the beginning of Section 5.

The analogical switch passes a powerup command from the computers to the general electro-valve. [1] This pressurises the rest of the landing gear hydraulic system, ready for specific further commands to manipulate its various parts, these being the doors of the cabinets that contain the gear when retracted, and the gear extension and retraction mechanisms themselves. Beyond this, both the analogical switch and the output

[1] As a rule, commands from the two computers are ORed by the components that obey them.

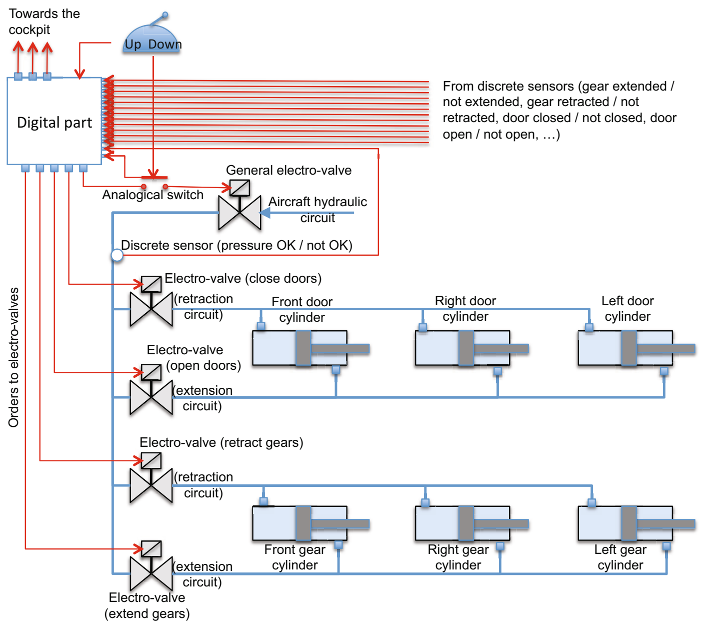
Towards the cockpit
Up Down
Digital part
From discrete sensors (gear extended / not extended, gear retracted / not retracted, door closed / not closed, door open / not open, …)
General electro-valve
Analogical switch
Aircraft hydraulic circuit
Discrete sensor (pressure OK / not OK)
Electro-valve (close doors)
(retraction circuit)
Front door cylinder
Right door cylinder
Left door cylinder
Orders to electro-valves
Electro-valve (open doors)
(extension circuit)
Electro-valve (retract gears)
(retraction circuit)
Front gear cylinder
Right gear cylinder
Left gear cylinder
(extension circuit)
Electro-valve (extend gears)

Fig. 1. Architectural overview of the landing gear system, reproduced from [6]

of the general electro-valve are monitored by (triplicated) sensors that feed back to the computer systems, as is discernible from Fig. 1[2].

What is particularly interesting about the system so far, is that the arrangement of these various interconnections between system components is evidently quite far from the kind of tree shape that facilitates clean system decomposition. Thus, the handle is connected to the computers, and the handle is connected to the analogical switch. But the analogical switch is also connected to the computers, so ‘dividing’ the computers from the analogical switch in the hope of ‘conquering’ structural complexity will not work, and obstructs the clean separation of proofs into independent subproofs concerning analogical switch and computers separately. This poses a major challenge for our modelling methodology, and gave rise to the need for new interconnection mechanisms, discussed in Section 4.

Beneath the level of the general electro-valve, it is a lot easier to see the system as comprised of the computers on the one hand, and the remaining hydraulic components on the other, connected together in ways that are tractable when the new interconnection mechanisms are available.

3 Hybrid Event-B, Single Machines

In this section we look at Hybrid Event-B for a single machine. In Fig. 2 we see a bare bones Hybrid Event-B machine, *HyEvBMch*. It starts with declarations of time

[2] A large number of other sensors also feed back to the computers, but this not relevant to the point we are making just now.

```
MACHINE $HyEvBMch$
TIME $t$
CLOCK $clk$
PLIANT $x, y$
VARIABLES $u$
INVARIANTS
  $x, y, u \in \mathbb{R}, \mathbb{R}, \mathbb{N}$
EVENTS
  $INITIALISATION$
    STATUS ordinary
    WHEN
      $t = 0$
    THEN
      $clk, x, y, u := 1, x_0, y_0, u_0$
    END
... ...
```

```
... ...
  $MoEv$
    STATUS ordinary
    ANY $i?, l, o!$
    WHERE
      $grd(x, y, u, i?, l, t, clk)$
    THEN
      $x, y, u, clk, o! :|$
        $BApred(x, y, u, i?, l, o!,$
        $t, clk, x', y', u', clk')$
    END
... ...
```

```
... ...
  $PliEv$
    STATUS pliant
    INIT $iv(x, y, t, clk)$
    WHERE $grd(u)$
    ANY $i?, l, o!$
    COMPLY
      $BDApred(x, y, u,$
        $i?, l, o!, t, clk)$
    SOLVE
      $\mathcal{D}\, x =$
        $\phi(x, y, u, i?, l, o!, t, clk)$
      $y, o! :=$
        $E(x, u, i?, l, t, clk)$
    END
END
```

Fig. 2. A schematic Hybrid Event-B machine

and of a clock. In Hybrid Event-B, time is a first class citizen in that all variables are functions of time, whether explicitly or implicitly. However time is special, being read-only. Clocks allow more flexibility, since they are assumed to increase like time, but may be set during mode events (see below). Variables are of two kinds. There are mode variables (like u) which take their values in discrete sets and change their values via discontinuous assignment in mode events. There are also pliant variables (such as x, y), declared in the PLIANT clause, which typically take their values in topologically dense sets (normally $\mathbb{R}$) and which are allowed to change continuously, such change being specified via pliant events (see below).

Next are the invariants. These resemble invariants in discrete Event-B, in that the types of the variables are asserted to be the sets from which the variables' values *at any given moment of time* are drawn. More complex invariants are similarly predicates that are required to hold *at all moments of time* during a run.

Then, the events. The *INITIALISATION* has a guard that synchronises time with the start of any run, while all other variables are assigned their initial values as usual.

Mode events are direct analogues of events in discrete Event-B. They can assign all machine variables (except time itself). In the schematic *MoEv* of Fig. 2, we see three parameters $i?, l, o!$, (an input, a local parameter, and an output respectively), and a guard *grd* which can depend on all the machine variables. We also see the generic after-value assignment specified by the before-after predicate *BApred*, which can specify how the after-values of all variables (except time, inputs and locals) are to be determined.

Pliant events are new. They specify the continuous evolution of the pliant variables over an interval of time. The schematic pliant event *PliEv* of Fig. 2 shows the structure. There are two guards: there is *iv*, for specifying enabling conditions on the pliant variables, clocks, and time; and there is *grd*, for specifying enabling conditions on the mode variables. The separation between the two is motivated by considerations connected with refinement.

The body of a pliant event contains three parameters $i?, l, o!$, (again an input, a local parameter, and an output) which are functions of time, defined over the duration of the pliant event. The behaviour of the event is defined by the COMPLY and SOLVE clauses. The SOLVE clause specifies behaviour fairly directly. For example the behaviour of pliant variable y and output $o!$ is given by a direct assignment to the (time dependent) value of the expression $E(\ldots)$. Alternatively, the behaviour of pliant variable x is given by the solution of the first order ordinary differential equation (ODE) $\mathcal{D}\,x = \phi(\ldots)$, where $\mathcal{D}$ indicates differentiation with respect to time. (In fact the semantics of the $y, o! = E$ case is given in terms of the ODE $\mathcal{D}\,y, \mathcal{D}\,o! = \mathcal{D}\,E$, so that x, y and $o!$ satisfy the same regularity properties.) The COMPLY clause can be used to express any additional constraints that are required to hold during the pliant event via its before-during-and-after predicate *BDApred*. Typically, constraints on the permitted range of values for the pliant variables, and similar restrictions, can be placed here.

The COMPLY clause has another purpose. When specifying at an abstract level, we do not necessarily want to be concerned with all the details of the dynamics — it is often sufficient to require some global constraints to hold which express the needed safety properties of the system. The COMPLY clauses of the machine's pliant events can house such constraints directly, leaving it to lower level refinements to add the necessary details of the dynamics.

Briefly, the semantics of a Hybrid Event-B machine is as follows. It consists of a set of *system traces*, each of which is a collection of functions of time, expressing the value of each machine variable over the duration of a system run. (In the case of *HyEvBMch*, in a given system trace, there would be functions for clk, x, y, u, each defined over the duration of the run.)

Time is modeled as an interval $\mathcal{T}$ of the reals. A run starts at some initial moment of time, t_0 say, and lasts either for a finite time, or indefinitely. The duration of the run $\mathcal{T}$, breaks up into a succession of left-closed right-open subintervals: $\mathcal{T} = [t_0 \ldots t_1), [t_1 \ldots t_2), [t_2 \ldots t_3), \ldots$. The idea is that mode events (with their discontinuous updates) take place at the isolated times corresponding to the common endpoints of these subintervals t_i, and in between, the mode variables are constant and the pliant events stipulate continuous change in the pliant variables.

Although pliant variables change continuously (except perhaps at the t_i), continuity alone still admits a wide range of mathematically pathological behaviours. To eliminate these, we insist that on every subinterval $[t_i \ldots t_{i+1})$ the behaviour is governed by a well posed initial value problem $\mathcal{D}\,xs = \phi(xs \ldots)$ (where xs is a relevant tuple of pliant variables and $\mathcal{D}$ is the time derivative). 'Well posed' means that $\phi(xs \ldots)$ has Lipschitz constants which are uniformly bounded over $[t_i \ldots t_{i+1})$ bounding its variation with respect to xs, and that $\phi(xs \ldots)$ is measurable in t. Moreover, the permitted discontinuities at the boundary points t_i enable an easy interpretation of mode events that happen at t_i.

The differentiability condition guarantees that from a specific starting point, t_i say, there is a maximal right open interval, specified by t_{MAX} say, such that a solution to the ODE system exists in $[t_i \ldots t_{\mathrm{MAX}})$. Within this interval, we seek the earliest time t_{i+1} at which a mode event becomes enabled, and this time becomes the preemption point beyond which the solution to the ODE system is abandoned, and the next solution is sought after the completion of the mode event.

In this manner, assuming that the *INITIALISATION* event has achieved a suitable initial assignment to variables, a system run is *well formed*, and thus belongs to the semantics of the machine, provided that at runtime:

- Every enabled mode event is feasible, i.e. has an after-state, and on its completion enables a pliant event (but does not enable any mode event).[3] (1)
- Every enabled pliant event is feasible, i.e. has a time-indexed family of after-states, and EITHER: (2)

  (i) During the run of the pliant event a mode event becomes enabled. It pre-empts the pliant event, defining its end. ORELSE
  (ii) During the run of the pliant event it becomes infeasible: finite termination. ORELSE
  (iii) The pliant event continues indefinitely: nontermination.

Thus in a well formed run mode events alternate with pliant events. The last event (if there is one) is a pliant event (whose duration may be finite or infinite). In reality, there are a number of semantic issues that we have glossed over in the framework just sketched. We refer to [4] for a more detailed presentation.

We point out that the presented framework is quite close to the modern formulation of hybrid systems. See eg. [15,12] for representative modern formulations, or [8] for a perspective stretching further back.

4 Top-Down Modelling of Complex Systems, and Multiple Cooperating Hybrid Event-B Machines

The principal objective in modelling complex systems in the B-Method is to start with small simple descriptions and to refine to richer, more detailed ones. This means that, at the highest levels of abstraction, the modelling must **abstract away from concurrency**. By contrast, at lower levels of abstraction, the events describing detailed individual behaviours of components become visible. In a purely discrete event framework, like conventional Event-B, there can be some leeway in deciding whether to hold all these low level events in a single machine or in multiple machines — because all events execute instantaneously, isolated from one another in time (in the usual interpretation).

In Hybrid Event-B the issue is more pressing. Because of the continuous behaviour that is represented, *all* components are always executing *some* event. Thus an integrated representation risks hitting the combinatorial explosion of needing to represent each possible combination of concurrent activities within a separate event, and there is a much stronger incentive to put each (relatively) independent component into its own machine, synchronised appropriately. Put another way, there is a very strong incentive to **not abstract away from concurrency**, an impulse that reflects the actual system architecture. In Hybrid Event-B, there is thus an even greater motivation than usual for the refinement methodology to make the step from monolithic to concurrent convincingly.

This is accomplished by using normal Hybrid Event-B refinement up to the point where a machine is large enough and detailed enough to merit being split up. Then, the key concept in the decomposition is the INTERFACE. This is adapted from the idea in [9] to include not only declarations of variables, but of the invariants that involve them,

[3] If a mode event has an input, the semantics assumes that its value only arrives at a time strictly later than the previous mode event, ensuring part of (1) automatically.

```
INTERFACE $Level7\_AnSw\_IF$
READS $Level7\_Comp\_IF$
REFERS $Level7\_Comp\_IF$
CLOCK $clk\_AnSw$
VARIABLES
  $AnSwClosed, sens\_AnSw_i$
PLIANT
  $answ2genev$
INVARIANTS
  $AnSwClosed \in$ BOOL
  $answ2genev \in$ BOOL
  $(AnSw\_CLOSED\_INIT < clk\_AnSw <$
    $AnSw\_CLOSED\_FIN) \Rightarrow AnSwClosed$
  $\neg(AnSw\_CLOSED\_INIT \leq clk\_AnSw \leq$
    $AnSw\_CLOSED\_FIN) \Rightarrow \neg AnSwClosed$
...   ...   ...   ...
```

```
...   ...   ...   ...
  $\bigwedge_i sens\_AnSw_i \in \{OPEN, CLOSED\}$
  $\bigwedge_i AnSwClosed \Rightarrow sens\_AnSw_i = CLOSED$
  $\bigwedge_i \neg AnSwClosed \Rightarrow sens\_AnSw_i = OPEN$
  $answ2genev \Rightarrow cmp2answ_1 \vee cmp2answ_2$
  $answ2genev \Rightarrow AnSwClosed$
INITIALISATION
  BEGIN
    $clk\_AnSw := BIGT$
    $AnSwClosed := FALSE$
    $||_i\ sens\_AnSw_i := OPEN$
    $answ2genev := FALSE$
  END
END
```

Fig. 3. Level 7 interface for the analogical switch, from the case study

and their initialisations. A community of machines may have access to the variables declared in an interface if each machine CONNECTS to the interface. All events in the machines must preserve all of the invariants in the interface, of course. An important point is that *all* invariants involving the interface's variables must be in the interface.

Well, not quite all; an exception is needed. Invariants of the form $U(u) \Rightarrow V(v)$, where variables u and v belong to different interfaces, are also allowed. Such *cross-cutting* invariants (which we call type 2 invariants, t2i s) are needed to express fundamental dependencies between subsystems which are coupled in a nontrivial manner (such couplings invariably arise in multicomponent systems). In a t2i, the u and v variables are called the local and remote variables respectively. By convention a t2i resides in the interface containing its local variables.

Fig. 3 shows an example of the preceding taken from the landing gear case study. It is an interface, $Level7\_AnSw\_IF$, primarily intended for some variables of the *An*alogical *Sw*itch. It contains some, by now, familiar ingredients, such as a clock clk_AnSw, and some mode and pliant variables, $AnSwClosed, sens\_AnSw_i, answ2genev$. These model the state of the analogical switch, the state of its sensors, and the signal from the switch to the general electro-valve. It also contains statements READS $Level7\_Comp\_IF$ and REFERS $Level7\_Comp\_IF$.

The first of these says that the interface contains a t2i (specifically $answ2genev \Rightarrow cmp2answ_1 \vee cmp2answ_2$) for which the local variables (i.e. $answ2genev$) are found in $Level7\_AnSw\_IF$, and the remote variables (i.e. $cmp2answ_1, cmp2answ_2$) are found in $Level7\_Comp\_IF$, which is another interface, predominantly concerned with variables (and their invariants) belonging to the computer systems.

The second expresses the converse idea, namely that there is a t2i in $Level7\_Comp\_IF$ for which the local variables are in $Level7\_Comp\_IF$, and the remote variables are in $Level7\_AnSw\_IF$.

By restricting to t2is as the only means of writing invariants that cross-cut across two interfaces (and, implicitly, across the machines that access them), we can systematise, and then mechanise, the verification of such invariants. Thus, for a t2i $U(u) \Rightarrow V(v)$ it is sufficient for events that update the u variables to preserve $\neg U$ (if it is true in the before-state) and for events that update the v variables to preserve V (if it is true in the before-state). A more comprehensive treatment of the notion of interface used here appears in [2].

As well as sharing variables via interfaces, multi-machine Hybrid Event-B systems need a synchronisation mechanism — one that is more convenient than creating such a thing *ab initio* from the semantics. For this the shared event paradigm [7,14] turns out to be the most convenient. In this scheme, identically named (mode) events in two (or more) machines of the system are deemed to be required to execute simultaneously. In practice, it means that for each such event, its guard is re-interpreted as the conjunction of the guards of all the identically named events. Below, in Section 6, we say rather more more about mode event synchronisation. In particular, we point out the need for a more flexible method of identifying events which are to be synchronised than pure name identity. (In fact, a more flexible mechanism has been implemented in the Rodin Tool [13] than is described in the literature. However, we stick, for simplicity and comparability with the published literature, to the simple and static identical name scheme.)

5 Model Development

Having discussed the technical preliminaries, in this section, we overview the development of the landing gear case study. To clarify some minor inconsistencies in the spec [6], we assume that the pilot controls the gear via a *handle* for which handle *UP* means gear up, and handle *DOWN* means gear down. We also assume that in the initial state the gear is down and locked, since the aircraft does not levitate when stationary on the ground, presumably. Connected with this requirements aspect is the absence of provision in [6] of what is to happen if the pilot tries to pull the handle up when the aircraft is not in flight. Presumably the aircraft should not belly-flop to the ground, so we just incorporate a suitable guard on the handle movement events, based on the value of the shock absorber sensors. This leaves open the question of what *actually* happens if the pilot pulls the handle up when the plane is on the ground. Does the handle resist the movement, or does gear movement remain pending until released by the state of the shock absorber sensors, or ...?

This issue, in turn, raises a further interesting question. Although the fact just pointed out causes no special problem for an event-by-event verification strategy like the B-Method, the absence of any explicit requirement that allows the shock absorber to change value, would be equivalent to the aircraft never leaving the ground, leading to the absence of nontrivial traces for a trace based verification strategy to work on (unless suitable additional events were introduced into the model, just for this purpose).

Pursuing the technical strategy discussed earlier, implies that in the final development, each component that is identifiable as a separate component in the architectural model, should correspond to a machine in its own right. Thus, at least, the pilot subsystem (handle and lights), the two computers, the analogical switch, the general electro-valve, and the individual movement electro-valves (and their associated hydraulic cylinders), should all correspond to separate machines at the lowest level of development. The nontrivial interdependencies between these subsystems give rise to enough cross-cutting type 2 invariants between the corresponding machines to thoroughly exercise the modelling capabilities of our formal framework.

A further technical goal in this development is, as far as possible, to use variables that correspond directly to quantities discussed in the requirements document. The aim is to strive for the closest possible correspondence between requirements and formal model, in the belief that this improves the engineering process. Allied to this is the fact that the present work is the most complex case study attempted in Hybrid Event-B to date, so, a certain amount of experimentation was carried out during the case study in order to evaluate different modelling approaches to various features found in [6]. Consequently, the same kind of situation is not always approached in the same way.

5.1 The Nominal Regime

With these remarks made, we turn to the development itself. This is too big to include in full here of course; the details can be found at [3]. In this section we summarise the essentials, pausing to discuss interesting issues as they arise.

We focus on the nominal regime. For the faulty regime, see below. Adhering to the vision of the B-Method, the development starts very simply, and proceeds to add detail via layers of refinement. As different parts of the system require different numbers of refinement steps in order to reach their final degree of detail, in [3], the various syntactic constructs are labeled with a level number, and the caption accompanying each construct states which constructs constitute the system at the current level of development.

Level 0 gives the simplest, pilot-level view of the system, and consists of just one machine: *Level0_PilotAndLightsNominal*. There are mode events for raising and lowering the handle, and for switching the green and orange lights on and off (the red light is ignored in the nominal regime). For example:

```
PilotGearUP
  ANY in?
  WHERE in? = pilotGearUP_X ∧ handle = DOWN
  THEN handle := UP
  END
```

This is identical to normal Event-B, aside from the input parameter *in*?, which is required to be *pilotGearUP_X*, and which is furthermore unused in the event. The explanation for this is that while in normal Event-B, events are assumed to execute *lazily*, i.e. *not* at the very instant they become enabled (according to the normal interpretation of how event occurrences map to real time), in Hybrid Event-B, mode events execute *eagerly*, i.e. as soon as they are enabled (in real time).

This is because physical law is similarly eager: if a classical physical system reaches a state in which some transition is enabled, it is overwhelmingly the case that energetics and thermodynamics force the transition to take place straight away. Hybrid Event-B, in being designed to model physical systems, must therefore conform to this. As a consequence, typical Event-B models, in which a new after-state immediately enables the next transition, would cause an avalanche of mode event occurrences if interpreted according to Hybrid Event-B semantics.

To avoid this, and yet to allow modelling convenience in Hybrid Event-B, the undesirable avalanche of mode event occurrences is avoided at runtime by building a delay into the semantics. The delay lasts as long as a required input parameter remains absent, and the semantics *assumes* that the input does not arrive until after some positive (but otherwise unspecified — unless more precisely constrained in the guard) period of time has elapsed.

There is also a default pliant event *PliTrue* to define behaviour between occurrences of the mode events. It merely stipulates COMPLY *INVARIANTS*.

Level 1 is a simple refinement of level 0, and just introduces some additional variables. Aside from minor details of syntax, it is just a discrete Event-B refinement of *Level0_PilotAndLightsNominal* to *Level1_PilotAndLightsNominal*.

Level 2 begins the process of splitting things into smaller components. The level 1 machine is split into *Level2_PilotNominal* and *Level2_CompNominal*. Each event of *Level1_PilotAndLightsNominal* is split into a pair of synchronised events in the two machines. The former reflects the pilot's view, in which the pilot is responsible for handle events (so the earlier $in? = pilotGearUP\_X$ goes into the *Pilot* machine), and the computer is responsible for the lights events (so the inputs for those events go into the *Comp* machine). The rationale for the latter is that the occurrences of the lights events depend on as yet absent *Comp* details, so at this level of abstraction, they just appear as spontaneously generated events from *Comp*'s environment, to be eventually refined to the deterministic behaviour of a more complete computing machine. The relationship between *Level2_PilotNominal* and *Level2_CompNominal* is mediated by an interface, *Level2_Comp_IF*, which contains all the variables shared by the two machines. The decomposition of the level 1 machine into the two level 2 machines plus their interface constitutes a 'textbook' example of doing decomposition according to the scheme described earlier.

The next few levels are concerned with reconciling the pilot's view of a singular computing system behaviour with the reality of the duplicated computing modules of the architecture of Fig. 1. Again, while the system description is still small, a 'textbook' approach to the issue is taken. What we mean by this is that there will be a machine depicting a singular computing system behaviour for the pilot, connected with two actual computing modules which will be successively refined to include further implementation detail. The textbook approach to this is to refine the *Level2_CompNominal* machine to a machine, *Level3_CompNominal* that: firstly, duplicates the computer initiated events (to model potential asynchrony of the two computing modules[4]); secondly, replicates the relevant variables so that each representative machine will have its own copy of each relevant variable. This situation is supported by an enriched interface *Level3_Comp_IF*. That done, at level 4, we can decompose *Level3_CompNominal* into *Level4_CompNominal* (expressing the pilot's view), and *Level4_Comp*$_1$*Nominal* and *Level4_Comp*$_2$*Nominal* (the two 'real' computing modules-to-be).

The main outcome of this approach is to convince us of its extreme verbosity as a way of modelling the 'OR' of the two computing modules' commands whenever they must send a command to any external component. In the remainder of the development, such verbosity is avoided by having the receiving component simply react to the OR of the received signals in the guards of its events, even though this is slightly inaccurate architecturally (since, in reality, the OR is calculated outside the relevant component).

The next step is to introduce the analogical switch, whose functioning takes time, for which the *Level5_AnalogicalSwitchNominal* machine introduces a clock, *clk_AnSw*. The analogical switch is open by default. When stimulated by a handle event, it takes

[4] We allow for potential asynchrony, even though in our idealised modelling sphere, both computing modules will follow exactly the same trajectory.

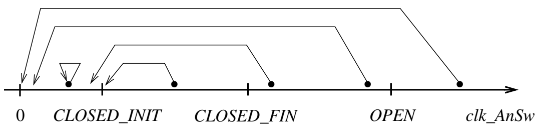

Fig. 4. The analogical switch machine's transitions when interrupted by a fresh handle event

some time to close (from 0 till *CLOSED_INIT*), then stays closed for a while (from *CLOSED_INIT* till *CLOSED_FIN*), then takes some time to open once more (from *CLOSED_FIN* till *OPEN*). Fig. 4 indicates what happens to the clock value when a fresh handle event occurs before the previous sequence has completed. The handle events that intiate these activities are synchronised with the pilot's handle events (in machine *Level5_PilotNominal*, which is a copy of *Level2_PilotNominal* but including these additional synchronisations). This in order to model the fact that —according to the architecture of Fig. 1— the pilot's handle events reach the analogical switch directly, and not via the computing modules.

Thus far, the development is relatively tree-shaped. Practically speaking, this means that there is no need for nontrivial invariants involving variables that are not declared in the same place. For a development of modest size, it is always possible to arrange things so that this holds. However, as the size of the development increases, the prescience needed to arrange the development so that this remains true, and the need to appropriately separate concerns, both render this desire unrealistic. We see this in concrete terms in our development at level 6, which is concerned with introducing the analogical switch sensors. For clarity, these are introduced in a separate step to that which introduces the analogical switch itself. Since the analogical switch is by now in a separate machine from the computing modules, any invariant involving the sensors and computing variables becomes a cross-cutting t2i. This applies to $\bigwedge_i \mathit{gearsMoving}_k \Rightarrow \mathit{sens\_AnSw}_i = \mathit{CLOSED}$ which states that various landing gears do not start moving until the the analogical switch is sensed to be closed. This t2i appears in the *Level6_Comp_IF* interface, using the t2i machinery discussed above. This necessitates a partitioning of *Level5_AnalogicalSwitchNominal* in that a new interface, *Level6_AnSw_IF* is needed to house some of the *Level5_AnalogicalSwitchNominal* variables, so as to conform to the syntactic conventions for t2is, yielding also machines *Level6_AnalogicalSwitchNominal* and *Level6_Comp*$_k$*Nominal*.

A similar process can be followed for introducing the general electro-valve. This is carried out at level 7, rather as for the analogical switch at level 5. What is interesting though, for the general electro-valve, is that the requirements [6] do specify some continuous behaviour for this component, albeit that this is simple linear behaviour. The opportunity is taken here to model this using nontrivial pliant events in machine *Level7_General_EV_Nominal*. For instance, the growth of pressure in the door and gear movement circuits is given by:

```
$PressureIncreasingOrHIGH$
  INIT $answ2genev$
  SOLVE
    $\mathcal{D}\ genEVoutput = PressureIncRate \times bool2real((genEVoutput < HIGH) \wedge answ2genev)$
  END
```

This says that the time derivative of *genEVoutput* is constant as long as *genEVoutput* does not exceed *HIGH* and the control signal *answ*2*genev* is true. Once *genEVoutput* $=$ *HIGH* is reached, the derivative drops to zero and so *genEVoutput* remains constant.

Level 8, which introduces the sensors for the general electro-valve, is as interesting as level 7. The general electro-valve sensors only signal *HIGH* when *genEVoutput* actually reaches *HIGH*. This leads to a multi-step refinement of the level 7 pliant event *PressureIncreasingOrHIGH*. A first pliant event models the increasing episode during which the derivative is nonzero, and a second pliant event models the constant episode during which *genEVoutput* remains at *HIGH*. The two pliant events are separated by a mode event *PressureHIGH_reached*, that turns the sensors to *HIGH*. A similar state of affairs holds for the pressure decreasing regime, when the *answ*2*genev* signal goes false.

Even more interesting is the fact that due to pilot initiated handle events, the analogical switch's behaviour may be restarted before a previous behaviour has completed, leading to two possible mode events in the general electro-valve that synchronise with the analogical switch closure event: one for the normal case when the general electro-valve is depressured *AnSw_CLOSED_INIT_reached_*1*_S*, and another for when it is already pressured-up *AnSw_CLOSED_INIT_reached_*2*_S*.

And even more interesting than that, is the fact that the timing of pilot initiated handle events may be such that mode event *AnSw_CLOSED_INIT_reached_*2*_S* is scheduled to occur at exactly the same moment as the mode event that naturally separates the increasing and *HIGH* episodes in the general electro-valve, *PressureHIGH_reached*. The guards and actions of the two mode events are identical, which would cause trouble with respect to the semantics, were it not for the fact that one of the mode events is a synchronised event and the other is not.

Normally, the unproductive complications of such coincidences in the semantics are avoided in Hybrid Event-B by assuming *in the semantics* that inputs do not arrive at times which clash with other mode events (see the earlier discussion in Section 4). But the case we are discussing is not like this since the coincidence occurs *as a consequence* of an earlier mode event that is quite innocent. Clearly such coincidences are not statically computable in general, so cannot be avoided by some kind of static definition in the semantics. Then, rather than complicate the modelling as we have done in the present case study, a possible way forward is as follows.

During design and development, we neglect the possible existence of these issues of undesired coincidence of mode events. In an environment with proper tool support for Hybrid Event-B, the potential coincidences will invariably generate some unprovable necessary conditions for semantic soundness. These conditions can then be added as further hypotheses in a domain theory, leading to closure of the previously open proofs. Provided such conditions only occupy a portion of the parameter space that is of zero measure, no harm would be done to any practical implementation, since no practical implementation that behaves in a stable way can hit a portion of the parameter space of zero measure.

We proceed to level 9. Now that the general electro-valve can be powered up and down, this level introduces the individual movement electro-valves, and implicitly, the hydraulic cylinders that they manipulate. Each of the four movement electro-valves and

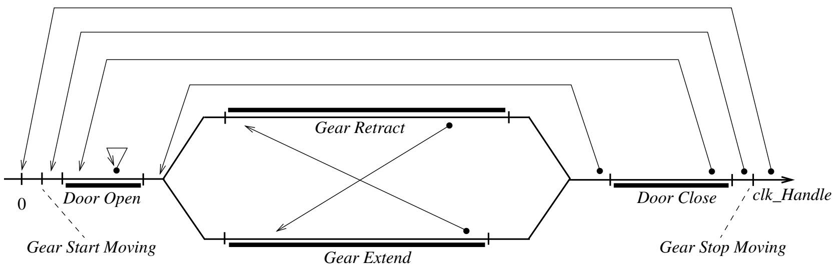

Fig. 5. The approximate timing diagram for the level 10 computing machine

cylinders gives rise to a new machine. Also there is *Level9_HydraulicCylinders_EV_IF*, a new interface that links them all to the computing modules. New synchronised events in the computing modules and electro-valve/cylinder machines command the initiation of the operation of the movement hydraulic cylinders, and timed events monitor the completion of the relevant operations via the relevant battery of sensors, given the variability in completion time described in [6]. All four operations are similar, so only one has been modelled in detail in [3]. The cross-cutting t2is that couple variables in *Level9_HydraulicCylinders_EV_IF* to those in the computing interface *Level9_Comp_IF* are handled in the by now familiar way.

Up to now, the impetus for executing any particular event that is potentially available in a machine has come from the environment, via the technique of using an external input that is created for that sole purpose. (Where there are synchronised families of events, one of them is allocated the external input and the rest are synchronised with it.) The final step in modelling the nominal regime is to remove this artifice, and replace it with explicit timing constraints. This is the job of level 10. Note that explicit timing information is already included in subsystems for which the description is relatively complete, such as the analogical switch, and the the general and movement electro-valves, so this development step only concerns the computing modules.

It was tempting to try to introduce the computing module timing constraints in a step by step fashion. However, it was soon realised that the complexity and interconnectedness of the constraints was such that a stepwise approach would need to allow guard *weakening* as well as guard strengthening. Since Event-B is not geared for guard weakening, the idea was abandoned in favour of a monolithic approach that introduced all of the timing machinery in one go.

Fig. 5 outlines the behaviour of the computing module's clock *clk_Handle*, when the handle is manipulated during the course of gear extending or retracting. Unlike Fig. 4 though, where the behaviour illustrated is close to what the model describes (since the analogical switch just responds to handle events in a self-contained way), Fig. 5 neglects important detail. For example, consider a *PilotGearUP_S* event while the gear is extending. Then, the retracting sequence has to be executed but only from the point that extending has reached. So first, *clk_Handle* is changed to stop the gear extending command. Then, *clk_Handle* is changed to a time sufficiently before the gear retracting command time that hydraulic hammer[5] has subsided. Once it is safe to activate the gear retracting command, the gear retracting command is activated, and then *clk_Handle* is changed again to advance the clock in proportion to the undone part of the gear extending activity. In effect, we use *clk_Handle* intervals as part of the state machine controlling the behaviour of the computing modules (along with additional internal variables). This proves especially convenient when the state transitions involved concern delays between commands that need to be enforced in order to assure mechanical safety (e.g. the hydraulic hammer case, just discussed). Such details are not visible in Fig. 5, but make the design of the level 10 events quite complicated. This completes our development of the nominal regime.

5.2 The Faulty Regime and the Imperative Closed Loop

Although we do not cover the faulty regime in detail in this study, we now indicate briefly how it would go in the context of a fuller Hybrid Event-B development. The structuring given by the nominal regime gives a good basis for considering the faulty regime. A great help here is the fact that the faults described in [6] are basically all *stuck_at* faults. To inject such faults into a nominal model is easy and systematic. For each potentially failing component we introduce a fault variable, and we additionally guard each preexisting event on the fault variable's falsehood. Furthermore, we introduce an event in the relevant machine to spontaneously make the fault variable true.

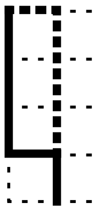
Fig. 6. The Tower Pattern

Having built up the nominal regime, the faulty regime would be constructed by retrenching the various nominal machines to include the needed faults in the manner just described. A great added benefit of this is that the suite of invariants built up for the nominal regime need not be changed in the face of *stuck_at* faults — retrenchment allows the invariants to be violated, after which further nominal behaviour ceases.

In a multistage development like the present one, the nominal and faulty versions would be related by *Tower Pattern* theorems such as can be seen in [5]. Fig. 6 shows the general scheme. The top-down nominal refinement-based development we have done appears as the bold left line, descending vertically through levels of abstraction as we have described. The faulty regime then takes a horizontal development step to the right, and builds up the analogous refinement chain bottom-up. This is indicated by the bold dashed line segments.

The ultimate product of an exercise like the present one, is to produce an iterative closed loop controller in a suitable imperative language, so that the control is reduced to the instructions of a suitable embedded processor. The modelling in this case study has not been carried that far, but we explain now why it would be easy to do.

We would just need a straightforward refinement. The reason for this is that the only continuous behaviour that is relevant to the case study is linear with respect to time

[5] Hydraulic hammer is the term for the collection of transient shock waves that propagate round the hydraulic system when relatively abrupt changes are inflicted on its control surfaces (i.e. the pistons in the various cylinders), and which are typically damped using a relatively elastic hydraulic accumulator somewhere in the hydraulic circuit in order to avoid damage to the hydraulic circuit components.

(whether this concerns a clock variable, or some other physical variable). Being linear, the behaviour becomes completely predictable over the duration of a sampling period. The needed refinement would thus need to simply replace the continuous behaviour of the pliant event that ran during the sampling period with a (continuous) skip, and augment the mode event that ran at the end of the sampling period with a discrete update that expressed the calculated changes in pliant variables over the sampling period just elapsed. The semantics of Hybrid Event-B would ensure that a straightforward retrieve relation was provable regarding this change of representation (see [4] for examples). This is indicated by the lowest vertical bold line segment in Fig. 6.

6 Review, Lessons Learned, and Conclusions

In the last few sections, we have overviewed the landing gear case study, and tackled the modeling challenges of capturing the resulting development using Hybrid Event-B. Although we restricted to the nominal regime, this provided a sufficient challenge to the modelling capabilities of Hybrid Event-B to reassure us of its suitability for this kind of system. In fact, with the nominal regime done, we were able to indicate that the faulty regime could be handled quite straightforwardly. A number of lessons emerged from this modelling exercise, which we summarise now.

[1] Doing an exercise like the present one by hand is *really* tricky. Almost every re-reading of some fragment of the development revealed another bug (although typically, such bugs would be easily picked up mechanically). Proper machine support is obviously vital when doing such a development in anger.

[2] Using a component's clock as an adjunct to its state machine proved very convenient in combination with conventional state variables. Modelling mechanical safety delays using pure state machine techniques would have made the state machines much more cumbersome. Simply adjusting the clock to allow a safety margin of time to elapse before the next required action was an elegant solution.

[3] The possibility of using t2is as a tool for breaking up complex architectures into more digestible components, while maintaining interdependencies, proved vital. This generic pattern showed itself to be both sufficiently expressive that needed dependencies could be captured, and sufficiently well structured that mechanisation across multiple machines and interfaces is feasible.

[4] Composition/decomposition mechanisms based on event name identity are inadequate to express the more dynamic synchronisations needed by complex system architectures. As noted already, the current Rodin Tool implementation of synchronised events goes beyond static event name identity, a need vividly illustrated in our case study.

[5] The tension between describing components as self-contained machines, utilising their own naming conventions as standalone entities, contrasts with the approach of regarding them *ab initio* as elements of the full system, adhering to system-wide naming conventions. In general, the synchronisation mechanisms referred to in **[4]** need to be combined with sufficiently flexible instantiation mechanisms to enable a proper component based approach to be pursued.

The need for the more flexible mechanisms mentioned in the last two points above is already apparent in some of the synchronisations used in the case study here, where it already proved impossible to do the needed job using purely static mechanisms. Such challenges, and others (for example, how to model edge-triggered behaviour in a formalism based primarily on states, or the more intensive use of input and output variables rather than shared variables), provide good inspiration for the further fine-tuning of the multi-machine version of the Hybrid Event-B formalism. Such insight will provide valuable guidance for subsequent tool building effort.

References

1. Akers, A., Gassman, M., Smith, R.: Hydraulic Power System Analysis. CRC Press (2010)
2. Banach, R.: Invariant Guided System Decomposition. These proceedings
3. Banach, R.: Landing Gear System Case Study in Hybrid Event-B Web Site (2013), `http://www.cs.man.ac.uk/~banach/some.pubs/ABZ2014LandingGearCaseStudy/LandingGearCaseStudy.html`
4. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core Hybrid Event-B: Adding Continuous Behaviour to Event-B (2012) (submitted)
5. Banach, R., Jeske, C.: Retrenchment and Refinement Interworking: the Tower Theorems. Math. Struct. Comp. Sci. (to appear)
6. Boniol, F., Wiels, V.: The Landing Gear System Case Study. In: ABZ 2014 Case Study Track. CCIS, vol. 433, pp. 1–18. Springer, Heidelberg (2014)
7. Butler, M.: Decomposition Structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
8. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and Tools for Hybrid Systems Design. Foundations and Trends in Electronic Design Automation 1, 1–193 (2006)
9. Hallerstede, S., Hoang, T.S.: Refinement by Interface Instantiation. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 223–237. Springer, Heidelberg (2012)
10. Ionel, I.: Pumps and Pumping. Elsevier (1986)
11. Manring, N.: Hydraulic Control Systems. John Wiley (2005)
12. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
13. RODIN Tool, `http://www.event-b.org/`, `http://www.rodintools.org/`, `http://sourceforge.net/projects/rodin-b-sharp/`
14. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition Tool for Event-B. Software Practice and Experience 41, 199–208 (2011)
15. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer (2009)
16. U.S. Department of Transportation, Federal Aviation Administration, Flight Standards Service: Aviation Maintenance Technician Handbook — Airframe (2012), `http://www.faa.gov/regulations_policies/handbooks_manuals/aircraft/amt_airframe_handbook/`