Int J Softw Tools Technol Transfer (2017) 19:205–228
DOI 10.1007/s10009-015-0409-7

CrossMark

ABZ 2014

The landing gear system in multi-machine Hybrid Event-B

Richard Banach[1]

Published online: 18 December 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract A system development 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 Hybrid Event-B 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 the modelling formalism. These aspects of the case study, vital in the development of realistic cyberphysical systems in general, have contributed significant improvements in the theoretical formulation of multi-machine Hybrid Event-B itself.

Keywords Hybrid Event-B · Landing gear case study · Refinement · Multicomponent systems

1 Introduction

This paper reports on a treatment of a landing gear system case study (see [17]) using Hybrid Event-B. Hybrid Event-B [9,10] 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. Since aircraft systems are replete with interactions between physical law and the engineering artefacts that are intended to ensure appropriate aircraft behaviour, they are prime examples of cyberphysical systems [15,19,26,31], especially when one takes into account the increasing use of remote monitoring of such systems via global communication networks. As such, there is a *prima facie* case for attempting the landing gear problem using Hybrid Event-B. 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 [33].

Given that landing gear is predominantly controlled by hydraulic systems (see Chapter 12 of [33]), it might be imagined that the requirements for the present case study [17], would feature relevant physical properties quite extensively. Of course Hybrid Event-B would be ideally suited to model and quantify the interactions between these physical properties and the control system—for example, on the basis of the theoretical models and practical heuristics detailed in references such as [1,22,25]. However, it is clear that the requirements in [17] 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. This perspective is appropriate to a treatment centred on system control via isolated discrete events, such events being used to mark the start and end of a physical process while quietly ignoring what might happen in the interior. While this approach certainly reduces the modelling workload, the penalty paid for it is the loss of the ability to *justify* the values of these constants during the verification activity, 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 [17].

✉ Richard Banach
banach@cs.man.ac.uk

[1] School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK

These are confined to simple linear behaviours of some parts of the physical apparatus. Although they are very simple, these linear behaviours are nevertheless enough to demonstrate many of the essential capabilities of the Hybrid Event-B formalism in dealing with continuous phenomena and their interaction with discrete events.

Besides the continuous behaviours that need to be addressed by any comprehensive formalism for cyberphysical systems, there are the issues of structure and architecture. Genuine cyberphysical systems are invariably composed of a(n often large) number of components, each with some degree of autonomy—while they nevertheless remain coupled to, and interact with, each other. The multi-machine version of Hybrid Event-B is intended to confront the challenges that this raises. Fortunately, the landing gear case study [17] is rich enough in structure to adequately exercise these aspects of the formalism.

The rest of this paper is as follows. Section 2 overviews the landing gear system, emphasising those elements that are of most interest to our development. Section 3 then gives a summary of single machine Hybrid Event-B. Section 4 extends this to cover multiple machines. The issues explored here lead to the hypergraph structured system architectures discussed in Sect. 5.

Section 6 then gives an architectural overview of the subsequent development, relating the general discussion of Sect. 5 to the specific architecture of the case study. Section 7 introduces the case study itself, indicating the most significant elements (from our point of view). Section 8 deals extensively with the nominal development via a series of refinements from a simple initial model.

We then move to the faulty regime. Regarding the introduction of faults, the present case study proves to be a fertile vehicle for exploring how the retrenchment framework [11,12,14] can handle the incompatibilities involved in extending from a nominal to a faulty regime, and the essential ideas are given in Sect. 9. Section 10 then applies these ideas to the faulty regime of the case study, again proceeding via a series of steps that breaks down the complexity of the task.

Section 11 shows how the retrenchment *Tower Pattern* can lead to further checking of the development. Section 12 discusses the development of a time-triggered implementation level model, as would arise in a genuine implementation. Section 13 looks back at the main lessons that emerged from the case study and summarises the most useful patterns that were developed. Issues that merit further attention are also discussed in some detail there. Section 14 concludes.

Comparison with the conference version. In the conference version of the case study [6], hereafter referred to as *Conf*, only the nominal regime of the case study was covered. Still, this proved sufficient to bring out the main benefits of the approach, and, through the complexity of the case study, highlighted several issues that needed to be handled better in the multi-machine context. Here, a more comprehensive development is covered. Various detailed differences from the earlier treatment are mentioned below, as they arise.

2 Landing gear overview

The definition of the landing gear case study together with its requirements is presented in [17]. Here, we give the gist of it, focusing on features of most interest to the Hybrid Event-B treatment. Figure 1, reproduced from [17], 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 to the computer systems from the sensors. These can behave in a relatively autonomous manner, to reveal faults to the computer(s) to which the system is required to be, to some extent, resilient. A further point concerns the shock absorber sensors. These are modelled using a guard rather than as inputs. The relevant issue is discussed at the beginning of Sect. 7.

The analogical switch passes a powerup command from the computers to the general electrovalve.[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 of the general electrovalve 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

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

[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.

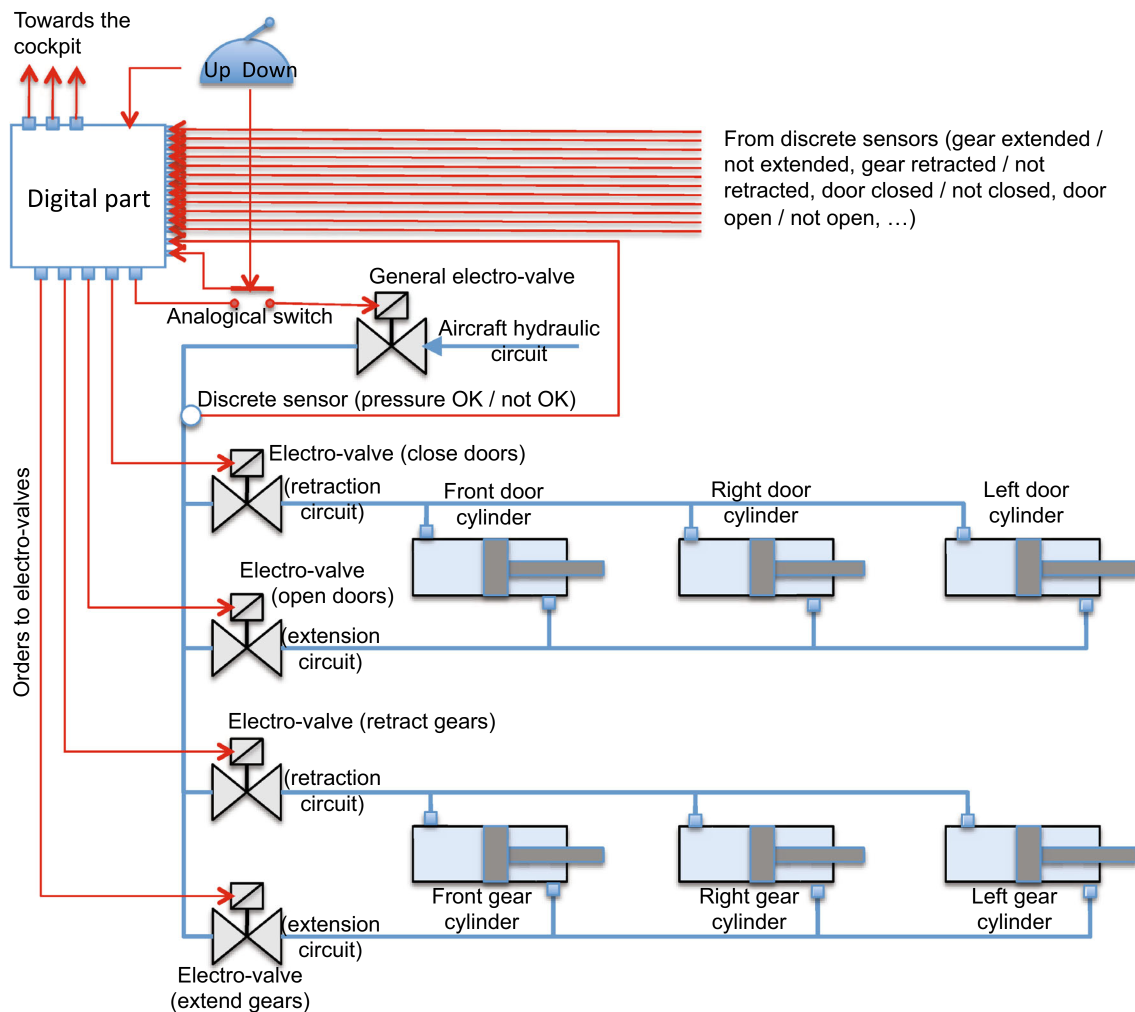

Fig. 1 Architectural overview of the landing gear system, reproduced from [17]

concerning analogical switch and computers separately. This poses a significant challenge for our modelling methodology, and gave rise to the need for new interconnection mechanisms (at least it did so in *Conf*), as discussed in Sects. 4 and 6.

Beneath the level of the general electrovalve, 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 rather more tractable by readily understood interconnection mechanisms.

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. Depending on the complexity of the interconnection network, this can require novel machine interconnection mechanisms, introduced for pure Event-B in [5]. 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 requirements of the case study.

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 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

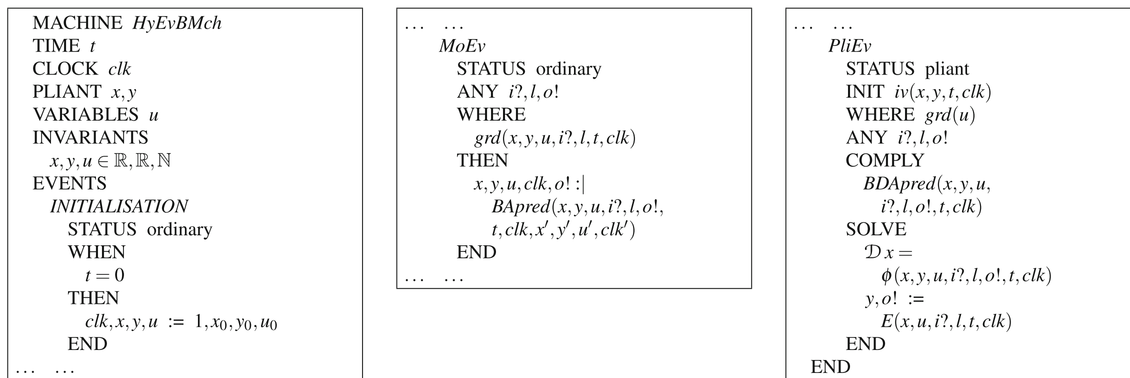

Fig. 2 A schematic Hybrid Event-B machine

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, we have 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. (For a more detailed discussion of issues such as this, see [9].)

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, aside from some small technical details, 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. (In fact, a major use to which we put the COMPLY capability in our case study is to demand that pliant variables, which would otherwise be unconstrained during a (n essentially default) pliant event, remain constant during it. This is mentioned in Sect. 10.)

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 modelled 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. (For example, speaking measure-theoretically, 'most' continuous functions are non-differentiable almost everywhere.) To eliminate such pathologies, 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 $[t_i \ldots t_{\text{MAX}})$. 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 preempts 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 [9] for a more detailed presentation.

We point out that the presented framework is quite close to the modern formulation of hybrid systems. See, e.g. [27,32] for representative modern formulations, or [19] 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), and in between, nothing changes.

4.1 Multi-machine systems via INTERFACEs

In Hybrid Event-B, the issue just mentioned 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. To put it another way, there is a very strong incentive to **not abstract away from concurrency**, an impulse that matches with 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 abstract and simple descriptions to more detailed and concrete concurrent descriptions, convincingly.

In our approach, this is accomplished using normal Hybrid Event-B refinement up to the point where a machine is large enough and detailed enough to merit being split up. After that, the key concept in the decomposition strategy is the INTERFACE construct. This is adapted from the similarly named idea in [21], to include not only declarations of variables (as in [21]), but of the invariants that involve them, and also their initialisations. (Thus, an interface becomes a kind of shell of a machine, except one without any specific events to change the variables' values, and thus permitting *any* change of value imposed by the events of a machine accessing the interface, provided it preserves the invariants.) A community of machines may have access to the variables declared in an

[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 provided each such machine CONNECTS to the interface. All events in such 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 recorded in the interface, which assists a (putative) tool to mechanically monitor whether all the needed proof obligations in the verification of a machine have been adequately discharged.

The way that this strategy is defined in [5] and in [10] means that provided the relevant combinatorial rules are followed regarding what is visible where, the ideas just described can serve equally well as a discipline for **composing** separate (and in principle, independently conceived) components in a component-based system construction discipline. This gives an alternative to the refinement and decomposition-based methodology just discussed. Of course, with suitable attention to the formal details, both approaches can coexist. A combination of refinement ideas and composition ideas can be used in the same development, since the addition of new components to a system can be viewed as a refinement of the system at the component level, analogously to the way that addition of new variables and their behaviours to an individual component is a refinement of the system at the level of variables. The paper [5] gives a brief description aimed at the (familiar, discrete-only) Event-B context, whereas [10] gives a more detailed discussion, taking all the additional considerations of Hybrid Event-B into account, and explaining how (de)composition and refinement all can be viewed as different sides of the same coin.

4.2 Type II invariants in multi-machine systems

Although we said above that an interface contains *all* of the invariants mentioning any of its variables, in practice this can be too restrictive. If a system architecture contains many components that are tightly coupled in complex ways, aggregating all the needed invariants together with the variables that they mention, when this involves many interconnected components, may result in a single interface that is too large and unwieldy for convenient and independent development. To cater for such an eventuality, the approach described in [5,10] permits also type II invariants (tIIi's), which are defined to be of the form: $U(u) \Rightarrow V(v)$, where variables u and v belong to different interfaces. This *pattern* for invariants is sufficient to express many kinds of dependency between components, without forcing variables u and v to belong to the same interface, aiding decomposition. In a tIIi, the u and v variables are called the local and remote variables, respectively. By convention, a tIIi resides in the interface containing its local variables, and the remote variables must also reside in an interface. Syntactically, each of these interfaces will contain a reference to the other.

By restricting to tIIi's 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 conveniently mechanise, the verification of such invariants. Thus, for a tIIi like $U(u) \Rightarrow V(v)$, 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). These observations are helpful in 'dividing and conquering' the verification task to promote separate working, while yet being sufficient to express a large fraction of the properties that may be required to hold between two different machines.

4.3 Synchronisation in multi-machine systems

As well as sharing variables via interfaces, multi-machine Hybrid Event-B systems need a mechanism to achieve synchronisation between machines—preferably, a mechanism that is more convenient than creating such a thing *ab initio* from the raw semantics. For this the shared event paradigm [18,30] turns out to be the most convenient. In this scheme, *mode event groups*, i.e. specified 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 has to be re-interpreted as the conjunction of the guards of all the events in the group. The restriction that only mode events are eligible for such synchronisation simplifies the theory of the synchronisaiton mechanism considerably. Moreover, it proves to be no real restriction at all. According to the semantics sketched in Sect. 3, to launch the synchronised execution of a family of pliant events spread across several machines, it would be sufficient to arrange (via the mode event synchronisation mechanism) the synchronised execution of a corresponding family of mode events whose primary purpose was to enable all the pliant events. But this is easy to program in general.

5 Hypergraph-based system architectures

The sections above described an armoury of techniques that can be applied to the problem of system specification and development. However, by itself, this gives no advice about *how* these techniques ought to be used in any particular case. Here, we give some guidance on that point.

Our principal recommendation is that wherever possible, **systems should have a hypergraph architecture**. In the light of the commentary above, this means that there should be a hypergraph in which the machines of the systems constitute the nodes, and the interfaces should form the hyperedges connecting families of nodes (i.e. machines).[4]

[4] An equivalent way of saying the same thing is the familiar recasting of hypergraphs as bipartite conventional graphs, with machine nodes

What this implies is that the variables that a machine needs to manipulate at various times typically split into a number of constituencies, each focused on a different concern. In turn, each such concern may involve a number of machines to achieve its goals. Provided the concerns do not overlap, the variables for each concern will be distinct from the variables for other concerns, and can thus be placed in an interface focused on that concern, to be manipulated by the machines that are involved with that concern as necessary. In this manner, we arrive at the *machine* = *node* and *interface* = *hyperedge* hypergraph structure.

This system architecture proposal is well validated in the landing gear case study. In the *Conf* version, this pattern was **not** followed (for reasons explained below). Instead, interfaces were associated with machines, each containing the variables typically written by that machine. Each *Conf* machine also needed access to interfaces containing the variables it read but did not update. Although this was a reasonable proposal architecturally, expressing dependencies between machines was made more cumbersome. A good deal of use was made of the tIIi mechanism discussed earlier to express needed dependencies, with the attendant cross-referencing between interfaces.

In the present development, a different tack was taken regarding a specific design decision (we give the specific details in Sect. 8). This made the hypergraph structure much more convenient. As a result of the different structure, *all uses of the tIIi mechanism were eliminated in the present work*. This was a byproduct of the hypergraph architecture that was quite unexpected (until the detailed work revealed it). This simplification of the architectural challenge via a hypergraph structure leads to our promoting it as a **generic good thing**. Of course, the tIIi mechanism remains available for situations in which even the hypergraph architecture implies a need for sufficiently complex cross-cutting interdependencies between machines and interfaces.

Of course, the issues we have discussed above need to be managed at the syntactic level. In [10], we describe in detail how this is done via the PROJECT file. At a given level of a development, the project file takes care of four issues. First, it lists the components (interfaces and machines) that constitute the system at that level. Second, it defines the synchronisations that need to hold between individual machine events. Third (though not relevant for the present development), it is the place where instantiation issues can be dealt with (in a component-based methodology using a component repository, or similar). Fourth, it can contain a reference to a file of global invariants; these would be overreaching invariants, derivable from the contents of all the interfaces of the project, to be used as required in the development process.

Footnote 4 continued
and interface nodes, and edges that connect a single machine to a single interface (Then, each interface node, with all its incident edges, constitutes a hyperedge of the previous description).

6 Case study architectural overview

In this section, we describe how the preceding ideas play out in the landing gear case study. Figure 3 shows the system architecture. Rectangles represent components of the system; rounded corners for machines and unrounded corners for interfaces. Figure 3 splits into three phases. The first, at the top, shows the very first model: *Level_00_PilotAndLights Nominal*. This is then refined and decomposed at level 01 into three machines: *Level_01_PilotNominal, Level_01_Comp*$_1$ *Nominal*, *Level_01_Comp*$_2$*Nominal*. The decomposition is represented by the diagonal dashed lines in the figure.

Below the higher horizontal dashed line is the remainder of the nominal development. Machine suffixes are '*Nominal*', interface suffixes are '*_IF*'. The nominal development itself consists of levels 00, 01, 02, 03, 04, 05, 06, 07. The series of numbers in each rectangle indicate the development levels at which components are introduced or changed. Thus, at level 04, the development consists of *Pilot, Central_IF, Comp*$_1$, *Comp*$_2$, *AnalogicalSwitch, General_EV*.

The solid lines represent the CONNECTS relationships between machines and interfaces. Since each such line indeed joins a machine to an interface, the bipartite graph representation of the hypergraph structure is clear from the structure of the middle layer of Fig. 3, so that all levels of the nominal development fit it. Thus, *Central_IF* contains the variables that embody the interaction between the computers, and the analogical switch and general electrovalve, while *HydraulicCylinders_EV_IF* contains the variables that embody the interaction between the computers and the hydraulic cylinders. The important point is that these sets of variables are *disjoint*, leading to a clean structure.

Below the second dashed line is the faulty development. Whereas the nominal development is accomplished entirely using refinement and refinement-compatible techniques, the faulty development entails departures from previously established behaviours. We deal with this using retrenchment, in two development levels: 10 and 11. The faulty level identifiers are separated from the nominal ones by writing two slashes in the level number series of each component. The shadows behind the interface components *Central_IF_Faulty* and *HydraulicCylinders_EV_IF_ Faulty* denote the *Nominal* versions that the faulty versions have been retrenched from, and which still play a role behind the scenes in the faulty development. All this is explained below. Clearly, the earlier hypergraph structure persists.

Regarding the nature of the refinements used in this case study, it is worthwhile pointing out that, overwhelmingly, the successive levels of the modelling refine their predecessor

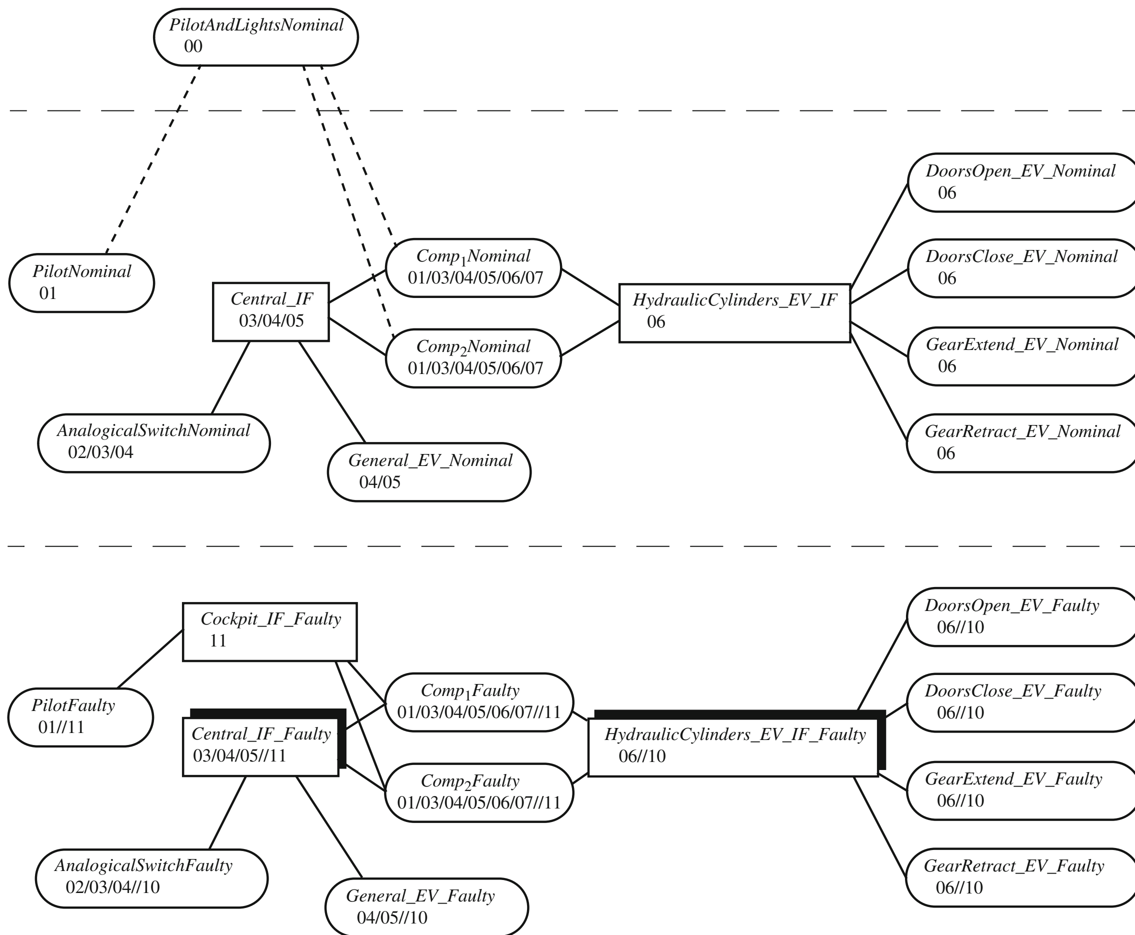

Fig. 3 Overview of the multi-machine Hybrid Event-B landing gear system development architecture. *Rectangles* depict interfaces (suffix '*_IF*' for nominal regime interfaces, '*_IF_Faulty*' for faulty regime interfaces). *Rounded rectangles* depict machines (suffix '*Nominal*' for nominal regime machines and '*Faulty*' for faulty regime machines). *Numbers* associated with components indicate the development levels (from 00,01,02,03,04,05,06,07 nominal, 10,11 faulty) at which components are introduced or undergo change

levels by the aggregation of fresh design detail. This makes the details of the refinement (and in particular the details of the verification of the refinement via the requisite proof obligations) relatively trivial. The refinement relation between two successive levels is just a projection from the concrete level to the abstract level that forgets the newly introduced detail. Although on the face of it this fails to exercise the data refinement capabilities of Hybrid Event-B to any significant degree, by contrast, the fact that real time is treated a just a parameter in Hybrid Event-B implies that any nontrivial data refinement will work by simply applying a discrete event style data refinement to the abstract and concrete variables in a parameterised way.

The exception to these remarks for this case study arises in the use of retrenchment, in which greater variance between abstract variables and their concrete counterparts is required than can be accommodated via refinement. However, it turns out that capturing these relationships between abstract and concrete can only be done using the additional machinery found in retrenchment, whereas the refining parts of the relationships remain as simple as before. This rather reinforces the preceding point.

7 Model development preliminaries

Having covered the architectural issues, we now look at the development in more detail. We start by clarifying our interpretation of some minor inconsistencies in the spec [17]. First, 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 [17] 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 would *actually* happen if the pilot pulled 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 any non-trivial gear manipulation traces for a trace-based verification strategy to work on. Thus, for a model checking approach of any kind to work, suitable additional events would have to be introduced into the model, for just the purpose of allowing the aircraft to leave the ground.

Pursuing the technical strategy advocated earlier, among other things 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, we should have separate machines for: the pilot subsystem (handle and lights), the two computers, the analogical switch, the general electrovalve, and the individual movement electrovalves (and their associated hydraulic cylinders). In concert with this was the desire to use variables that correspond directly to quantities discussed in the requirements document. The aim was to strive for the closest possible correspondence between requirements and formal model, in the belief that this improves the engineering process. It is possible that this perspective led to a granularity in the modelling which was not optimally efficient. However, the main priority in this study was to challenge the expressivity of the framework, rather than to test the efficiency of any putative tool implementation.

In addition to the above, the requirements concerned with the faulty regime in [17] mention the variables *normal_mode* and $anomaly_k$, without going into any explanation about their further purpose. This made it less clear how best to model these requirements. In the end, it was decided to create a fresh interface *Level_11_Cockpit_IF_Faulty* to contain those variables (and other quantities were conveniently included there too). This would in fact be needed if those variables were to be used by machines that modelled a cockpit display, for example (in contrast to the pilot's lights, which fall within the scope of the given requirements).

As mentioned earlier, both the *Conf* development and the present one *above all* constituted modelling challenges for the Hybrid Event-B formalism. The lessons that had already been learned from *Conf* were applied in the present development (see Sect. 8), which in turn generated further questions to be considered in future (see Sect. 13 for those). One notable element of this process was the change to the hypergraph architecture, which so dramatically eliminated the need for any type II invariants between interfaces.

8 The nominal regime

We now comment on the various levels of the nominal development, level by level. Along the way, we describe further notational conventions used in the development, as we did for the architectural overview already given. Adhering to the vision of the B-Method, the development starts very simply, and proceeds to add detail via layers of refinement and composition, with most of the steps of the development being quite small. Table 1 summarises the nominal development levels. The full details of the models in each level can be found at [3], with each level defined via the relevant PROJECT construct. The site [3] contains not only the present development (with all the levels aggregated into a single file) but also the previous *Conf* development.

Level 00. Level 00 starts the development. Because it is so small, we can quote a lot of the details, which will be helpful for other descriptions below. The PROJECT file for level 00 needs only to indicate the *PilotAndLightsNominal* machine, the sole construct at this level.

```
MACHINE Level_00_PilotAndLightsNominal
VARIABLES
  handle, green, orange
INVARIANTS
  handle ∈ {UP, DOWN}
  green, orange ∈ {ON, OFF}
EVENTS
  INITIALISATION
    STATUS ordinary
    BEGIN
      handle, green, orange := DOWN, ON, OFF
    END
  PliTrue
    STATUS pliant
    COMPLY INVARIANTS
    END
  PilotGearUP
    STATUS ordinary
    ANY in?
    WHERE in? = pilotGearUP_X ∧ handle = DOWN
    THEN handle := UP
    END
  GearStartMoving
    STATUS ordinary
    ANY in?
    WHERE in? = gearStartMoving_X
    THEN orange := ON
    END
...  ...  ...  ...
END
```

The *PilotAndLightsNominal* machine starts with its name, and the variables introduced for the handle and the green and orange lights. The invariants just state the values these are drawn from (i.e. their types). The initialisation is obvious

Table 1 Summary of the levels of the nominal development

Level	Feature
00	Pilot's view
01	Adds two computer machines and decomposes
02	Adds the analogical switch machine
03	Adds the triplicated analogical switch sensors
04	Adds the general electrovalve machine
05	Adds the triplicated general electrovalve sensors
06	Adds the movement electrovalves and hydraulic cylinders
07	Adds the timing automaton

(*green* is *ON* initially to indicate that the gear is down when the aircraft is in its initial state on the ground), after which there are the less trivial events.

PliTrue is the default pliant event—every Hybrid Event-B machine needs at least one pliant event because the machine must describe what is happening at *all times*, which cannot be done via mode events alone. Pliant event *PliTrue* just demands that the invariants are maintained, since the principal concern of the *PilotAndLightsNominal* machine is with the nontrivial mode events. These are *PilotGearUP* (and the analogous *PilotGearDOWN*) and *GearStartMoving* (and other mode events to switch the orange and green lights on and off).

PilotGearUP has a WHERE guard: *handle* = *DOWN*, which says the handle can only be pulled up if it is down to start with. The other part of the guard, *in?* = *pilotGearUP_X*, is the Hybrid Event-B metaphor for indicating that the event is stimulated from the environment and not from some other part of the system model. Thus *in?* denotes an input variable, and *pilotGearUP_X* is its required value when the input is supplied.[5] The body of the event, *handle* := *UP*, does not use the input, so the only role the input plays is to indicate the asynchronous timing of the event. The suffix '*_X*' on the input value is a naming convention used in this development to indicate the use of this metaphor.

The *GearStartMoving* event is very similar to the event *PilotGearUP* we just discussed, *except* that there is no term *orange* = *OFF* in the guard; we explain this shortly. This completes level 00.

Level 01. Level 01 REFINESandDECOMPOSES the level 00 machine *PilotAndLightsNominal*. In detail, it first introduces the first collection of variables needed for the two computing machines, $Comp_1$ and $Comp_2$, and then second, decomposes the result into the three separate level 01 machines: *PilotNominal*, *Comp₁Nominal*, *Comp₂Nominal*, furthermore doing all this in one step to save verbosity.

The relationship between these machines deserves comment. Each of the level 00 mode events has become, at level 01, an event that is synchronised between an event in *Pilot* and another in $Comp_1$ or $Comp_2$. As another notational convention, events which are synchronised are named with a *_S* suffix for visibility, although the actual definitions of the synchronisations are in the level 01 project file. The handle movement events are initiated by the pilot, so are modelled as previously, with, e.g. an *in?* = *pilotGearUP_X* guard to indicate the source of the external initiative for the event. The corresponding $Comp_1$ or $Comp_2$ events merely synchronise passively and update a corresponding variable. On the other hand, the lights switching on and off events are initiated by the computers, in response to (as yet undefined) behaviour in the rest of the system, so the *in?* = *gearStartMoving_X* guard moves from the pilot machine to the computer machines for these events, and this time, it is the pilot machine that synchronises passively.[6]

At this point, we hit the most significant difference between the *Conf* development and this one. Although it is not visible in Fig. 1, the communication from pilot to computers is a wired AND, and from computers to pilot is a wired OR. In *Conf*, some effort was made to model this faithfully, creating a machine to serve as the fictional single computing system presented to the pilot, interacting with the two real computers. This modelling style entailed the fictional computer synchronising with each of the two real computers, and proved to be extremely verbose—the authenticity was far outweighed by the obfuscating verbosity. Suffice it to say that the corresponding part of the present development occupies about a third of the text of the *Conf* development. It was this aspect, mainly, that made a hypergraph architecture excessively cumbersome in *Conf*, without the realisation (at the time), of how much modelling convenience was being given up by doing so.

[5] In discrete 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 must execute *eagerly*, i.e. as soon as they are enabled (in real time), to model physical phenomena.
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 the modelling convenience of permitting lazily executed mode events in Hybrid Event-B, the convention is that if a mode event has an input, at runtime the input value arrives at some time strictly later than the time of execution of the previous mode event, thus introducing a delay—if there is no input, execution is eager. The delay is nonzero, but undetermined unless more precisely constrained by restrictions in the event's guard. So having an input is the Hybrid Event-B metaphor for ensuring lazy execution, even if the input value is not used.

[6] Ultimately, the spontaneous occurrences of the *GearStartMoving* (and similar) events in the *Comp* machines will be refined to the more deterministic behaviour of more complete computing machines.

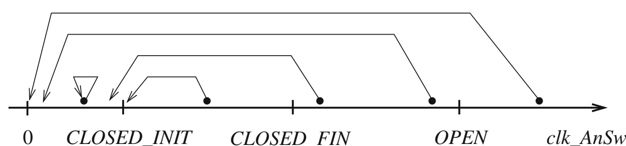

Fig. 4 The analogical switch machine's transitions when interrupted by a fresh handle event

In the present development, there is no fictional computer, and the OR is done in the pilot machine rather than outside—so the modelling is slightly less authentic. It turns out that this has consequences. If a single pilot event (such as the pilot's *GearStartMoving_S*) has to synchronise with each of the two computers' corresponding events (called also *GearStartMoving_S* in $Comp_1$ and $Comp_2$) then, assuming some asynchrony between $Comp_1$ and $Comp_2$, the pilot's *GearStartMoving_S* would be executed twice. This means that the pilot's event must be *idempotent*—executing it a second time needs to have no effect.[7]

Now, we see why there was no *orange* = *OFF* guard in *GearStartMoving_S* earlier. If there had been, the second occurrence of *GearStartMoving_S* would have been disabled in the pilot machine, causing problems.[8]

We have discussed this point with some care because the same issue arises every time the computers issue commands to any of the remaining equipment, e.g. to the analogical switch, or to the hydraulic apparatus. In all such cases, the synchronised event in the receiving machine must be idempotent.

Level 02. Level 02 introduces the analogical switch. The analogical switch is open by default. When a handle event occurs, the switch slowly closes (which takes from time *0* till time *CLOSED_INIT*), remains closed for a period (from time *CLOSED_INIT* till time *CLOSED_FIN*, allowing the onward transmission of commands from the computers to the general electrovalve), and then slowly opens again (from time *CLOSED_FIN* till time *OPEN*). If a handle event occurs part way through this process, Fig. 4 shows how the behaviour is affected: during closing, no effect; while closed, the closed period is restarted; during reopening, closing is restarted from a point proportional to the remainder of the reopening period.

A clock, *clk_AnSw* (*clk_xxx* being another naming convention, used for clocks), controls this activity. For this to work, the pilot's handle events are further synchronised with analogical switch events that reset *clk_AnSw* to the appropriate value, depending on its value at the occurrence of the handle event (N. B. The pilot's handle events reach the analogical switch directly, and not via the computing modules, this being part of the complex interaction between pilot, analogical switch, and general electrovalve).

Two further events (*AnSw_CLOSED_INIT_reached* and *AnSw_CLOSED_FIN_reached*) mark the transitions between episodes: from closing to closed, and from closed to reopening. Since these are 'new' events in an Event-B refinement, their STATUS is convergent, and a ($\mathbb{N}$-valued) VARIANT is included in the analogical switch machine, that is decreased on each occurrence of either of them. As usual, since $\mathbb{N}$ is well founded, the decrease of the variant implies that the new events cannot continue to occur indefinitely without the occurrence of 'old' events.

We dwelt on this last point a little since it introduces a useful pattern in the development that is reused a number of times below. Suppose an activity, engaged in by a number of actors, needs to progress through a series of tasks *t1,t2,t3,* circumscribed by a series of deadlines: $DL1 < DL2 < DL3$, all the deadlines being measured from a common starting point. Then, a useful variant that is decreased by events that mark the completion of the stages within the deadlines is:

No. of actors yet to complete t1 within DL1 +
No. of actors yet to complete t2 within DL2 +
No. of actors yet to complete t3 within DL3 +
0 × *No. of actors past DL3*

If the actors need to be in a specific state at each stage, etc. this can be built into the expressions occurring above. The idea is that as each actor completes the tasks and the deadlines expire in turn, the expressions above stop contributing to the variant, ensuring its decrease.

Level 03. Level 03 introduces the (triplicated) analogical switch sensors. The main reason for not doing this in the previous level is to exercise the composition and architectural features of the multi-machine formalism, to confirm that it is flexible enough to cope with this kind of gradual elaboration of components that are already present. In contrast to the *Conf* development, in this development, sensors respond asynchronously: each sensor is permitted to respond in its own right within a small time window following the closure or opening of the switch. This requires more 'new' events, and the variant that ensures their convergence is based once more on the pattern just described.[9]

[7] Dealing with this properly in the *Conf* development caused the majority of the excessive verbosity.

[8] It may be argued that the phenomenon being discussed is absent at level 00, so the guard could have been included there, and removed at level 01, but in Event-B refinement, guards are *strengthened*, so this would have prevented the 00 to 01 development step from being an Event-B refinement.

[9] In fact, the most compact way of writing the variant requires counting the *identities* of sensors satisfying the requisite properties. Strictly speaking this is outside the usual kind of B-Method type system, but it could be simulated by some more cumbersome programming. In our paper exercise, we sidestep this problem for the sake of clarity.

Here is the event that sets the ith analogical switch sensor to *CLOSED*. Note how temporal nondeterminism is dealt with by inputting a time which is loosely constrained within the event guard (In fact, the WHERE guard is equivalent to ($AnSw_CLOSED_INIT < clk_AnSw < AnSw_CLOSED_INIT + AnSw_DL) \wedge AnSwClosed$, but having an external input means that the event does not have execute eagerly, and thus does not have to disable itself).

```
AnSw_CLOSED_INIT_close_sens_AnSw_i
  STATUS convergent
  ANY time?
  WHERE clk_AnSw = time? ∧
    (AnSw_CLOSED_INIT < time? <
    AnSw_CLOSED_INIT + AnSw_DL) ∧
    AnSwClosed
  THEN
    sens_AnSw_i := CLOSED
  END
```

N. B. More naming conventions: $sens_xxx_i$ names a sensor and i always indexes over a triple of sensors (In the same vein, k always indexes over the two computers in the development). Suffix '*_DL*' denotes a deadline or delay.

The way that the sensors are handled embodies another pattern introduced in this development. This one concerns a framework for 'timed stimulus and response'. One machine (in this case, the pilot machine) executes a stimulus event (in this case, moving the handle) to which a timely response (in this case, the activation of the analogical switch's sensors) is anticipated. The stimulus event is synchronised in the responding machine with a clock value on a clock within the responding machine (in this case, the analogical switch's clock *clk_AnSw*). The synchronisation (in this case, the earlier handle event) opens a time window in the responding machine (in this case between *AnSw_CLOSED_INIT* and *AnSw_CLOSED_INIT* + *AnSw_DL*) within which the responding event takes place (the timing restriction being enforced via the responding event's guard). We see all this in the event $AnSw_CLOSED_INIT_close_sens_AnSw_i$ above.

Of course, the purpose of the sensors is to enable something else to happen elsewhere in the system. In this case, it is in the computers, which need to be aware of the sensors; their gear movement events (such as *GearStartMoving_S* earlier) need to be refined to acquire stronger guards, now with all three $sens_AnSw_i$ set to *CLOSED* conjoined, to enable them.

We observe at this point that our general purpose timed stimulus and response pattern could equally well have been used to model the handle sensors and their role in the control of the system. In fact, though, we did not model the handle sensors in this development, using event synchronisation to model the cooperation between handle and computers. This is equivalent to assuming that the handle never fails, since the synchronisation forces the computer event whenever there is a handle event.

The presence of the sensors permits quite a number of invariants connecting the behaviour of the analogical switch clock, the analogical switch state and the behaviour of the analogical switch sensors to be written. This and the additional involvement of the computers with the analogical switch sensors prompts the creation of an interface, the level 03 *Central_IF*. The analogical switch's clock, its variables and invariants are all moved there. The analogical switch machine and the two computing machines all CONNECT to this interface.

Level 04. Level 04 commences the introduction of the general electrovalve. The approach is similar to the introduction of the analogical switch, in that there is a new machine for the general electrovalve to model its behaviour. The general electrovalve machine is connected to the *Central_IF* interface because of the way the general electrovalve is connected to the remaining components.

It is clear from Fig. 1 and from the accompanying discussion, that the pilot, computers, analogical switch and general electrovalve are all interconnected in a quite complicated way (giving especial interest to this case study). Now, that all of these components are present in the development, the 'chain of command' between them can be better represented. Thus, level 04 introduces variables $comp2answ_k$ and *answ2genev* to represent the commands from the computers via the switch to the general electrovalve, and variable *genEVoutput* to represent the output of the general electrovalve to the rest of the hydraulics. All these variables reside in *Central_IF*.

Unlike for the other components of the landing gear system though, the description of the general electrovalve in [17] does specify some continuous behaviour, albeit that this is simple linear behaviour. We take the opportunity to model this using nontrivial Hybrid Event-B pliant events in the general electrovalve machine. For instance, the growth of pressure in the door and gear movement circuits is given by the following pliant event:

```
PressureIncreasingOrHIGH
  INIT answ2genev
  SOLVE
    𝒟 genEVoutput =
        PressureIncRate ×
        bool2real((genEVoutput < HIGH) ∧ answ2genev)
  END
```

In this event, The INIT clause only permits the behaviour described if *answ2genev* is true. In that case, $\mathcal{D}$, the time derivative symbol in Hybrid Event-B signals an ordinary differential equation (ODE) system in the SOLVE clause that has to be solved to define the behaviour. The right hand side (RHS) of this ODE contains a case analysis. If the pressure *genEVoutput* is less than *HIGH*, the its value increases linearly. As soon as it reaches *HIGH* though, the *bool2real* function makes the RHS zero, which maintains *genEVoutput*

at the high level. In the *Conf* development, a mode event separated these two phases because of the different synchronisation of sensors there. In this development, there is no intervening mode event, and the RHS represents a genuine case analysis.

There is a corresponding pliant event that governs the decrease of *genEVoutput* when *answ2genev* is false. It contains a similar case analysis that stops the linear decrease once the level drops to *LOW*. The increasing and decreasing episodes are separated by mode events that are synchronised with the analogical switch's mode events that move *answ2genev* between true and false. This is as required by (1) and (2) above.

We can take this argument further. In reality, a piece of equipment like the general electrovalve obeys a single physical law which would state that the rate of change of *genEVoutput* was governed by a particular physical property (most probably depending on the properties of a hydraulic accumulator elsewhere in the hydraulic circuit). As such, it would be fair to define it in Hybrid Event-B using a single ODE such as $\mathcal{D}\, genEVoutput = some_expression$, leading to a machine with a single plaint event and nothing else. Taking our linear modelling seriously, an explicit such event would resemble:

```
PressureLaw
  SOLVE
    𝒟 genEVoutput =
        PressureIncRate ×
        bool2real((genEVoutput < HIGH) ∧ answ2genev) +
        −PressureDecRate ×
        bool2real((genEVoutput > LOW) ∧ ¬answ2genev)
  END
```

In *PressureLaw*, the value of *answ2genev* in the RHS switches between increasing and decreasing episodes, so the RHS of the ODE now contains a four-way case analysis. In terms of modelling, it does not change much aside from the elimination of mode events. In terms of automated verification though, it increases the burden on any automated analysis system, since such a system would have to discover the four-way case analysis automatically. The range of possibilities considered, from the four pliant events of the *Conf* development, through the two pliant events of the present development, to the single pliant event *PressureLaw* above, are all connected to the issue of instantiation of a standard component into a specific development. The description in [10] gives a simple proposal, based on simple renaming of the elements of a standard component, but there is clearly scope for more elaborate schemes.

Level 05. Level 05 completes the development of the general electrovalve by incorporating its sensors. This is done in the same way that it was done for the analogical switch. New events are introduced in the general electrovalve machine to assign the sensors, governed by the same asynchronous setting pattern. Their convergence is confirmed by a variant built according to the same variant pattern we discussed earlier. Other events, primarily in the computer machines, are refined to take note of the sensors.

Of course, most of the preceding could have been accomplished in many fewer development steps than we expended. The main purpose in the more numerous small steps we took was to confirm that such increments of functionality could be handled by the multi-machine formalism without problems. Furthermore, small steps are much easier to handle for automated verification (looking forward to mechanical support for Hybrid Event-B).

Level 06. We proceed to level 06. Now, that the general electrovalve can be powered up and down, this level introduces the individual movement electrovalves, and implicitly, the hydraulic cylinders that they manipulate. Little further purpose is served by slicing the development into small increments, given that the patterns for doing this are well established by now, so we introduced all in one step, the four movement electrovalves, their sensors, and a new interface *HydraulicCylinders_EV_IF*.

Each of the four movement electrovalves and cylinders gives rise to a new machine: *DoorsOpen_EV*, *DoorsClose_EV*, *GearExtend_EV*, *GearRetract_EV*. These four machines are identical in structure, so only *DoorsOpen_EV* is written out in full. The *HydraulicCylinders_EV_IF* contains all the variables needed for this step, and they are coordinated using the same collection of by now familiar patterns.

The $Comp_k$ machines grow steadily larger due to the accumulating set of variables that they have to be sensitive to. They also display an interesting phenomenon. There are 'new' events to initiate the manipulation of the movement hydraulic cylinders, and to detect the completion of their movement tasks. Normally, new events need to decrease a variant. However, during the movement tasks, the handle may be manipulated an arbitrary number of times, which can even prolong the task indefinitely if it is a gear movement task. It is true that 'old' events (the handle manipulation events) interleave the new event occurrences, and thus it would be possible to invent state variables that could be used to create a suitable variant. But such variables would not address any system requirements, so they were not introduced. Consequently, the new events were defined with STATUS 'ordinary', not 'convergent'. This absolves them from the obligation of decreasing any variant (In a more realistic development, such variables would typically prove more useful in the faulty regime, but we did not introduce them here; further related comments appear below).

The situation just discussed illustrates a general point. The more an event relies on input or stimulus from the

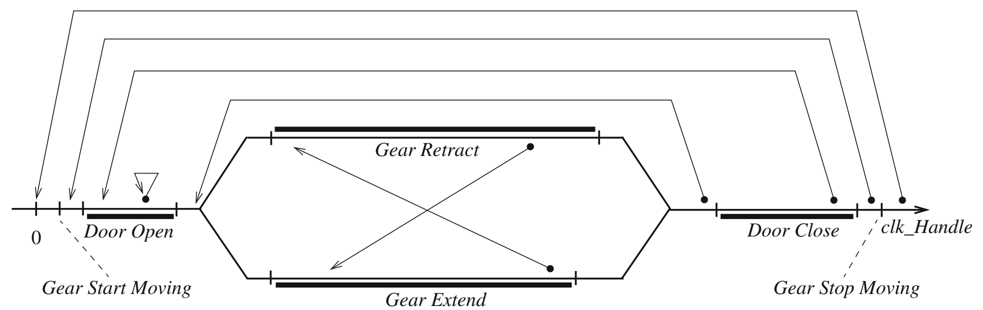

Fig. 5 The approximate timing diagram for the level 07 computing machine

environment, the weaker are the properties that we can expect to be able to prove regarding aspects that depend on this external influence—the potentially arbitrary intervention of handle movement events during landing gear operation being a case in point. Only when events depend on variables or stimuli that are fully internalised in the model, can we expect to prove relatively strong properties regarding them, since we can then identify and thence quantify *all* the influences that they might depend on.

This portion of the development also shows eloquently the benefits of the hypergraph system architecture. Although the computers have to converse with the handle, the analogical switch and the general electrovalve on the one hand, and with the movement cylinders on the other hand, these conversations involve separate sets of variables, so the variables can be conveniently partitioned into the *Central_IF* and the *HydraulicCylinders_EV_IF* interfaces, respectively, the separation benefiting separate development.

Level 07. We proceed to level 07. 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. Having reached a fairly complete level of detail in the nominal regime, the final step in modelling is to remove this artifice, and replace it with explicit timing constraints. This is the job of level 07. We observe that explicit timing information is already included in subsystems for which the description is relatively complete, such as the analogical switch, and the general and movement electrovalves, so the level 07 development step only concerns the computing modules.

In attempting to incorporate the timing information into the computing modules, it was tempting to try to introduce the timing constraints in a step-by-step fashion. For example, one could imagine having only non-overlapping raising and lowering episodes first, and then adding the extra complexities brought about by allowing raising and lowering episodes to overlap. However, it was soon realised that the complexity and interconnectedness of the timing constraints was such that a stepwise approach would need to allow guard *weakening* as well as guard strengthening in various events.[10] Since the Event-B notion of refinement is not designed for guard weakening (the goal of guard weakening being delegated to relative deadlock freedom POs), the idea was abandoned in favour of a monolithic approach that introduced all of the computing modules' timing machinery in one go.

Figure 5 outlines the behaviour of the computing modules' clock clk_Handle_k, 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 position that the extending sequence has reached. So first, clk_Handle_k is changed to stop the gear extending command. Then, the clk_Handle_k clock is changed to a time sufficiently before the gear retracting command time, that there is certainty that hydraulic hammer[11] has subsided. Once it is safe to activate the gear retracting command, the gear retracting command is activated, and then clk_Handle_k is changed again to advance the clock in proportion to the undone part of the gear extending activity. In effect, we use clk_Handle_k intervals as part of the state machine controlling the behaviour of the computing modules (along with some additional internal variables). This proves especially convenient when the state transitions involved concern delays between commands that need to be enforced to assure mechanical safety (e.g. as in the hydraulic hammer case, just discussed). Such

[10] For example, introducing conflicting handle events gives rise to the need to *extend* the time interval within which certain other events are required to occur, thus weakening the event guards that express such temporal constraints.

[11] 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 to avoid damage to the hydraulic circuit components.

fine detail is not visible in Fig. 5, but it makes the design of the level 07 events quite complicated. As a result, the size of the $Comp_k$ machines grows considerably at level 07; the movement events grow because of the extra detail, but, most particularly, the original handle events are refined into a large number of subevents, to cater for all the different things that need to happen depending on where in the sequence of activities an interruption by a fresh handle event occurs.

We can give an example of the preceding in the event *PilotGearDOWN_DoorsClose_Start_DoorsClose_End_S*. It defines what happens when the pilot pulls the handle down while the doors are closing at the end of a previous manoeuvre (which must have been a gear retraction manoeuvre, so that clk_Handle_k is between *DoorsClose_Start_TIME* and *DoorsClose_End_TIME*). Then, $Comp_k$'s clock is moved to just before *DoorsClose_End_TIME* so that door closure control can be tidied up, $doors_open_progress_time_k$ is set to the appropriate portion of the door opening period, and the $doors_open_mode_k$ variable is set to *INTERRUPTED*, so that when reopening is started, the opening event can react to $doors_open_progress_time_k$ instead of opening the doors from the beginning. By manipulating the clock value, it is easy to allow for the subsidence of hydraulic hammer before switching between extension and retraction of the gear.

```
PilotGearDOWN_DoorsClose_Start_DoorsClose_End_S
  STATUS ordinary
  REFINES PilotGearDOWN_k_S
  WHEN
    ⋀_{X,i} sens_ShockAbsorber_{X,i} = AIR ∧
    (DoorsClose_Start_TIME < clk_Handle_k <
      DoorsClose_End_TIME)
  THEN
    clk_Handle_k := DoorsClose_End_TIME − ε
    doors_open_restart_time_k :=
        DoorsOpen_Start_TIME − DoorsOpen_Hydraulic_DL
    doors_open_progress_time_k := DoorsOpen_Start_TIME +
        [ (DoorsClose_End_TIME − clk_Handle_k) / (DoorsClose_End_TIME − DoorsClose_Start_TIME) ] ×
        (DoorsOpen_End_TIME − DoorsOpen_Start_TIME)
    gear_retract_restart_time_k := Gear_Start_TIME
    gear_retract_progress_time_k := Gear_Start_TIME
    doors_open_mode_k := INTERRUPTED
    gear_movement_mode_k := NORMAL
    handlecmp_k := DOWN
    cmp2answ_k := TRUE
  END
```

9 Retrenchment and the introduction of faults

Our technique for incorporating faults into the development is based on retrenchment. This is an approach that dates back to [13], especially in the context of the B-Method. More up to date treatments include [11,12,14,29]. Applications to mechanised fault tree construction include [7,8].

Retrenchment is a technique that allows an event to be modified during a development step more drastically than permitted by refinement. This implies that the kinds of guarantee that refinement can typically offer are missing from development steps that are described via retrenchment. In that sense, retrenchment can be seen as a formally documented requirements engineering and requirements modification framework. Of course, the modification of a nominal model to incorporate and tolerate faults falls neatly within this remit.

Formally, and using Event-B terminology,[12] retrenchment relates pairs of events (rather as refinement does): an event in an 'abstract' level model and a corresponding event in a 'concrete' level model. Because the aim is to have a notion that can coexist smoothly with refinement (c.f. [11]), retrenchment is characterised by a PO similar in structure to a typical refinement PO, but populated with additional data that yields the more liberal notion. Besides the events that are related, there can be other events at both levels that are unrelated to one other by the retrenchment PO—no restrictions are placed on these.

We adapt a little the formal syntactic structure of retrenchment for Event-B given in [2] to fit the framework of Sect. 3 a bit better. If a concrete model retrenches an abstract model, the state variables of the two models are related via a RETRIEVES clause. This is essentially like a collection of joint invariants during refinement in conventional Event-B, except that we distinguish them syntactically for easier manipulation.[13] Such RETRIEVES clauses can occur in interfaces (to relate abstract and concrete interface variables) and in machines (to relate abstract and concrete machine variables). With this in place, the schematic structure of a retrenched event now appears as follows.

```
EventNameC
  STATUS ...
  RETRENCHES EventNameA
  ANY locals
  WHERE/WITH
    guards
  OUT
    output_clause
  CONC
    concedes_clause
  THEN
    actions
  END
```

[12] Retrenchment has been explored for Event-B specifically in [2].

[13] Thus far, we have not needed this kind of *data refinement* facility, since all refinement was expressed by the aggregation of new detail, and of further constraints on existing behaviour. But now, we need to *alter* some of the previously defined behaviour, so we need to distinguish the behaviours of earlier and new versions of variables.

The above shows the details for a mode event; for pliant events, the idea is essentially the same, as we describe. After the event name and status, the abstract event that is being retrenched is identified. Then, come the guards, which have just the same structure as for conventional mode or pliant events, as applicable. The WITH alternative is to express nontrivial relationships between abstract and concrete locals, when needed—this works just as for refinement in conventional Event-B. After that come the only visible differences from conventional events, the OUT and CONC clauses. The idea is that if the simulation of the concrete by the abstract event is able to reestablish the RETRIEVES relation and all the invariants, then any additional facts for that case can be accommodated in the OUT(put) clause. Alternatively, if the simulation of the concrete by the abstract event is *not* able to reestablish the RETRIEVES relation and the invariants, then any relevant facts for that case can be accommodated in the CONC(edes) clause. Both clauses offer maximum expressivity, in that abstract and concrete before- and after-values can be mentioned in them for mode events. For pliant events, the whole of these variables' behaviours during the pliant event are available—thus, they both act as (and therefore replace) the COMPLY clause, one for the retrieving case and the other for the conceding case. Given how expressive the OUT and CONC clauses are, it is often useful to have a trivial RETRIEVES relation, delegating all aspects of the relationship between abstract and concrete variables to the WHEN/WITH and OUT and CONC clauses. Below we describe how these facilities are used in building the faulty regime of the landing gear case study.

In the literature, various notions that do not conform to refinement have been proposed for incorporating faults into formal models. These permit the concrete model to depart from the abstract model in specific ways. In many cases, these ways are, in fact, very similar to what the retrenchment PO permits. In the cases of interest to us, once faults have occurred error recovery is modelled, and after it has completed, system behaviour is allowed to rejoin refining behaviour. In this category, we can cite, e.g. [23] and [24]. Their authors typically call their notions by names that are 'flavours of refinement', but given the explicit departure from refining behaviour that is permitted, we prefer the formulation used here.

10 The faulty regime

To model the landing gear faulty regime, we use the well-known fault injection technique. Each potentially failing element *xxx* has an associated boolean fault variable *ff_xxx* (another naming convention). To each potentially failing component, we add an fault injection event (parameterised by failing element, as required) $Inject_A_Fault_{xxx}$. This sets the relevant fault variable to *TRUE*, and sets the failing element to an arbitrary value. Additionally, we assume elements fail hard, so once set at the point of failure, neither the failing element nor its fault variable ever changes again. The elements that are allowed to fail this way are the sensors (since it is only the sensors that the $Comp_k$ machines can react to); also, the $Comp_k$ machines themselves can fail (using ff_comp_k), which disables all further computation by $Comp_k$. Regarding the sensors, the $Comp_k$ machines also maintain variables $OK_sens_xxx_k$ to record which sensors have (not) failed, to ensure that no more than one out of each triple of sensors has failed during normal operation. As noted earlier, in this study, for simplicity, we did not model the handle sensors or their failure; nor did we model failure of the shock absorber sensors. But we could have done it, by the same techniques described below. The construction of the faulty regime was approached in stages, as was the case for the nominal regime. Table 2 gives a summary.

Table 2 Summary of the levels of the faulty development

Level	Feature
10	Electromechanical component failures
11	Computer failures

Level 10. Level 10, the successor of level 07, is the first faulty regime level. This level models the easiest part of the faulty regime, the failures of the basic electromechanical components, namely: the analogical switch, the general electrovalve, and the four gear movement hydraulic cylinder sets. We capture the transition from level 07 to level 10 using a retrenchment from the level 07 project to the level 10 project.

In common with many development situations in which there is a considerable difference between behaviours at the two levels, the RETRIEVES relation for this retrenchment is taken to be *TRUE*. This does not constrain the concrete model in any way, and we can describe the deviation from the abstract to concrete behaviour quite adequately in the other data of the retrenchment, namely the WHEN/WITH guard relations (for the before-states) and the OUT and CONC relations (for the after-states, and their relationships with the before-states). This RETRIEVES relation appears in the two interfaces of the development, since all the electromechanical components' variables reside in these interfaces.

The change in an event of the electromechanical components in the presence of fault injection is illustrated in the following example, taken from the faulty analogical switch.

```
AnSw_CLOSED_FIN_open_sens_AnSw_i
  STATUS ordinary
  RETRENCHES AnSw_CLOSED_FIN_open_sens_AnSw_i
  ANY time?
  WHERE clk_AnSw = time? ∧
    (AnSw_CLOSED_FIN < time? <
    AnSw_CLOSED_FIN + AnSw_DL) ∧
    ¬AnSwClosed
  OUT
    ¬ff_sens_AnSw_i ∧ sens_AnSw'_i = sens_AnSw'_i^05
  CONC
    ff_sens_AnSw_i ∧ sens_AnSw'_i ∈ {OPEN, CLOSED}
  THEN
    sens_AnSw_i :|
      (¬ff_sens_AnSw_i ∧ sens_AnSw'_i = OPEN) ∨
      (ff_sens_AnSw_i ∧ sens_AnSw'_i = sens_AnSw_i)
  END
```

If the fault is not active, then the behaviour is as before: $sens_AnSw_i \;:|\; sens_AnSw'_i = OPEN$; while in the presence of a hard fail, we have: $sens_AnSw_i \;:|\; sens_AnSw'_i = sens_AnSw_i$ which says that the after-value of $sens_AnSw_i$ is the same as its before-value.

In the former case, the OUT clause is established, i.e. we have $sens_AnSw'_i = sens_AnSw'^{05}_i$, or that the level 10 after-value of $sens_AnSw_i$ is equal to its abstract counterpart, indicated via the 05 superscript. In the latter case, we establish merely that $sens_AnSw'_i \in \{OPEN, CLOSED\}$, since there is no control over the faulty value that $sens_AnSw_i$ is stuck at.

In this development step, all of the events of the electromechanical components are affected the exact same way. The $Comp_k$ machines are unaffected. Consequently, as soon as there is a failure in an electromechanical component, the $Comp_k$ machines become insensitive to it, since they demand unanimity from the sensors.

Note how the separation of concerns made explicit by having many different machines in this development is very beneficial in dividing and conquering the complexities of the faulty regime. Thus, we could focus on the failures of the electromechanical subsystem first, without needing to involve the computers. The fact that we are using retrenchment is helpful in not needing to define *all* the behaviours relevant to a specific failure scenario in one go (as would need to be the case if we attempted this via refinement).

Level 11. The failure behaviour of the $Comp_k$ machines is tackled at level 11. Referring to level 07 (which contains the final nominal $Comp_k$ machine), a typical $Comp_k$ event either reacts to a handle movement, or stimulates (or reacts to) some activity in the electromechanical system, or manipulates the lights. There are no other cases.

It is easy to integrate faulty behaviour into the handle movement events. We assume the handle does not fail, so the $Comp_k$ machines receive these events reliably. Whether anything happens depends on the state of the $Comp_k$ machines themselves. Hard fails of the $Comp_k$ machines are modelled using the ff_comp_k variables, whereas failures detected in the electromechanical components which are not to be compensated are recorded using the variables $anomaly_k$. All events thus include the term $\neg(ff_comp_k \vee anomaly_k)$ in their guards (Also, at the start of any sequence of activity, the landing gear shock absorber sensors are checked to indicate that the aircraft is in the *AIR*). Therefore, for the handle movement events, the inclusion of the additional guards is sufficient. Also, for the events that signal the pilot by manipulating the lights, inclusion of these additional guards is enough, since we assume that cockpit functionality never fails.

The movement events are trickier. Under nominal operation, initiation of a movement is done when clk_Handle_k reaches a suitable value (conditional on other variables, and on the relevant sensors being unanimous about a cylinder's condition)—termination is triggered by the last relevant sensor indicating success. In the faulty regime, two out of three trustworthy sensors indicating success is sufficient. But the first two trustworthy sensors will always indicate success regardless the behaviour of the last one, so the events for the '*2of3_OK*' cases must be fired on a timeout, to give the last sensor time to respond. Additionally, the movement events are already split into *NORMAL* and *INTERRUPTED* cases since their behaviour needs to differ depending on whether a previous sequence has been interrupted by a fresh handle event before completion. Addition of faulty behaviour thus generates a four-way split. But this only covers the cases compatible with eventually acceptable behaviour. There remain cases for each movement event which are not to be compensated, when at least two sensors out of three are misbehaving. So there must be *Anomaly_xxx* events at level 11 that catch these cases and set the $anomaly_k$ variable and unset *normal_mode*, and this last case covers the transmutation of the nominal regime into the faulty regime.

Here is an *Anomaly_xxx* example (for gear retracting). The test is that at least two sensors are not registering correctly *and* the timeout has expired. The latter means that at least two sensors on the same device failed somewhere. Otherwise, a '*2of3_OK*' case would have succeeded.

```
Anomaly_GearRetract_End_S
  STATUS ordinary
  WHEN ¬(ff_comp_k ∨ anomaly_k) ∧
  clk_Handle_k = GearRetract_End_TIME ∧
    (∃ (X,i) ≠ (Y,j) •
      sens_GearsRetracted_X,i = FALSE ∧
      sens_GearsRetracted_Y,j = FALSE)
  THEN
    anomaly_k := TRUE
    normal_mode := FALSE
  END
```

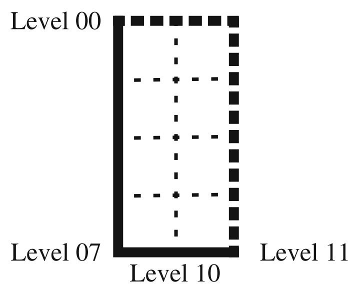

Fig. 6 The Tower Pattern, schematically illustrating the landing gear development. The nominal development, performed via series of descending *vertical* refinement steps appears on the *left*. Retrenchment steps, shown *horizontally*, construct the faulty regime in two stages, arriving at the Level 11 system at the *bottom right*

11 Fault handling and the Tower Pattern

The series of refinements and retrenchments we have described is a good example of the *Tower Pattern* [11]. This envisages a world of system models spread over a two-dimensional surface. Different models that lie on the same vertical line are assumed to be related by (a suitable notion of) refinement, and different models that lie on the same horizontal line are assumed to be related by retrenchment. In this scheme, the trajectory we have taken in our development is shown in the solid line of Fig. 6. The earliest model, level 00, is at the top left, and it is refined through seven refinement layers to level 07 to complete the nominal development.[14] Then, we embark on retrenchments towards the right to get the faulty regime at level 11 in the bottom right corner. The theorems of [11] guarantee that there are system models (consistent with the development) at all the grid points of the rectangle generated by the downward and rightward trajectories. With this picture, we can additionally do the following: from the bottom right corner, we can undo the various levels of refinement (in an essentially automatic way according to the theorems of [11]) to get an ultimate abstraction of the faulty regime model at a level corresponding to level 00. Since the refinements were all based on the aggregation of design detail, the corresponding abstractions will all be based on the forgetting of that detail in the faulty context. The end result will therefore be a level 00 analogue of the level 11 machine *Level_11_PilotFaulty*. The only serious difference from the level 00 machine is the inclusion of the red light, as befits a faulty retrenchment of it.[15]

We are not done yet though. The case study description in [17] lists a collection of requirements in its final section. The opportunity was taken to take these on board verbatim, within the multi-machine Hybrid Event-B methodology. The methodology permits the PROJECT file to contain a GLOBINVS declaration indicating a file of global invariants. These are supposed to be provable from the contents of all the interfaces in the project.

In [17], the requirements we have in mind were a mixture of kinds. Some of them were obviously expressible as state invariants, while others were much more like liveness constraints. While the latter might present problems for a purely discrete methodology without explicit liveness facilities, the presence of real time in Hybrid Event-B and of deadlines in the requirements permitted the translation of these into time- sensitive invariants,[16] thus falling within the remit of the basic Hybrid Event-B methodology. These translated requirements all appear in the global invariants construct *Level_11_LandingGearSystem_GI*. The conventions that are demanded for global invariants insist that they be derivable from the interfaces in the project, so in order that all of the variables needed by the global invariants were correctly declared, a number of the variables needed by the $Comp_k$ machines were moved into a new interface created at level 11 specially for the purpose: *Level_11_Cockpit_IF_Faulty*. This interface could also serve a wider objective, if modelling of the cockpit display were to be included in the development.

One further job was done at level 11. Whereas in previous development steps, the text highlighted only the changed parts (e.g. newly refined events), abbreviating unchanged parts using ellipses, for example, and copying unchanged components verbatim in their most recently updated form, at level 11, all these missing details have been filled in. This includes filling in all the details of the default pliant events that interleave the mode events in the majority of the machines. Where the modelling metaphor involved the use of pliant variables for signals that needed to change instantaneously in synchronised mode events, their constancy outside of these mode events is demanded using the CONST modality [4] in the relevant COMPLY clause of the default pliant event.

We are still not done yet though. The integration of the nominal and faulty regimes into one unified development encourages further cross-checking, beyond the global invariants just discussed, as follows. During the nominal development, as successive components are introduced, it is tempting to write additional, and quite strong invariants, knowing that these hold in the nominal regime. In fact, this was done during the development of the analogical switch and general electrovalve and similar components, and the additional invariants can be seen in the relevant interfaces (especially

[14] Figure 6 shows four layers instead of seven.

[15] Because of the simplicity of these two models, this retrenchment is a degenerate one, which defaults to a refinement.

[16] Above, we said that invariants had to be true *at all times*. There is no contradiction. Time- sensitive invariants that are always true can be written in the form *condition on time or on clocks* $\Rightarrow$ *property of variables*.

in *Central_IF*). In the faulty regime, these invariants no longer hold, and this fact necessitates the retrenchment of the *Central_IF* and *HydraulicCylinders_EV_IF* interfaces to their level 10/11 versions, *Central_IF_Faulty* and *Hydraulic-Cylinders_EV_IF_Faulty*. But as long as no fault has been activated, these stronger invariants will hold in a run of the system. This suggests using the *Tower Pattern* development architecture to close the verification loop in the following way.

It would be quite feasible to mechanise a process, to be applied to the faulty system as follows: (a) delete/disable all fault injection events from the development; (b) unify each interface's nominal and faulty versions and include the resulting interface in the development;[17] (c) then reverify everything. Particularly with a variable naming convention like the one we have used in this development, in which the same names are used at successive levels, by doing the preceding, the stronger invariants are made to apply within the faulty regime, and they will be expected to hold, since the faulty behaviour has been rendered unreachable.[18] In a realistic version of this process, the above steps would most likely need to be strengthened by additional invariants expressing the absence of faults, since their unreachability would be implicit from the point of view of an individual event, whose guard might still be sensitive to the possibility of faults.

12 The time-triggered loop

A practical implementation of a system like the present one will ultimately culminate in a time-triggered loop implementation. That is to say, the computing systems will have a timer interrupt, which will regularly wake them at a predetermined frequency, and upon being woken, the timer value, the internal state, and the sensors will all be examined to determine what, if anything, needs to happen.

In principle, the events of the level 11 system are fine-grained enough that we would want to see them directly implemented in the time-triggered loop system. Two issues immediately emerge. The first is relatively trivial, and concerns the observation that the time-triggered implementation is typically a sequential program, so that some reconfiguring of the event descriptions may be required to fit this paradigm better, in particular when the events of our 'ideal' event system occur sufficiently close in time that they have to be accommodated in the same polling routine. We regard this as a relatively minor refactoring issue. The second concerns the fact that, in a time-triggered loop, events are not executed at exactly the time that the level 11 system would execute them.[19] It is the job of this section to explore this.

One rather simple approach is to assume that all events happen at some exact multiple of the sampling period. Provided the sampling period is short enough, this is often good enough from an engineering viewpoint. Then, the behaviour of the implementation is just a subset the behaviours permitted by the abstract model, and the latter can simply be viewed as an over-generous safety property. Consequently, the relationship between abstract and implementation models becomes a straightforward refinement, based on an identity retrieve relation (and on the predictability of linearly evolving behaviours for those parts of the system expressed in continuous terms).

However, in reality, events will *not* all happen at a multiple of the sampling period, and those assumptions will not hold. The consequence of this is that the abstract model becomes, not a safety property, but a requirement, and the implementation must address all of the behaviours it permits.

Several observations now follow. The first is that only the computing systems are modified by the time-triggered implementation. The electromechanical components continue to behave as before. The second is that because we model handle stimuli via events synchronised with the computers (and not via sensors), we are forced to restrict handle events to occur at exact multiples of the sampling period.[20] The third is that regarding the electromechanical components, when they are stimulated they behave just as they did before, so no modelling change is needed there. The fourth is that when the electromechanical components attempt to stimulate the computer(s), the computer(s) can only react at predetermined times, so that the synchrony of the relevant events in our detailed modelling would need to be broken, a delay between a stimulus and its processing would need to be introduced, and the consequences of that delay would need to be captured in the formal relationship between abstract and concrete systems. In summary, a system run would now be broken up

[17] While the nominal version contains the stronger invariants, the faulty version contains the references to the abstract variable names, so both are needed. Furthermore, they are compatible, since the retrenchment step just removes the former and adds the latter.

[18] With different naming conventions, more/different refactoring might be needed to achieve this overall effect.

[19] This is based on the assumption that the specifications of these events remain as in the level 11 system, but that the times at which they occur changes—this also being connected with the issue of whether the polling frequency is high enough.

[20] As a digression, suppose we modelled handle events via sensors. Then, putting aside practicability considerations, in theory, there could be an arbitrary number of handle events within a single sampling period. But only the parity of this number would count. If there were an even number of events, the situation seen by the computer(s) at the end of the sampling period would be the same as before, and the preceding activity would continue undisturbed. If there were an odd number of events, it would be the same as a single handle event, and a new gear movement activity would be started, or the orientation of the previously ongoing one would be reversed. Interesting effects might occur if, due to asynchrony, one computer detected an even number of events and the other computer detected an odd number of events.

into some events in which the computer(s) stimulated the electromechanical components, where the timing would be as before, and other events in which the electromechanical components stimulated the computer(s), but where the reaction would be delayed by up to a sampling period.

One popular way of accommodating this kind of disparity in behaviours between abstract and concrete models is the *retiming* technique [16,20,28]. In this, the concrete model is given a more elastic notion of time, so that despite the asynchronies caused by time-triggering, corresponding events in the two models nevertheless occur 'at the same time'—time is 'wiggled a bit' in the jargon of [20]. Of course, a lot of detailed inequalities have to be proved, so that one can be sure that unrelated events do not collide with one another, or get put out of temporal or causal order.

In general, the present author is disinclined towards the retiming approach. A time like *3 o'clock* should refer to the same moment in all models in an engineering development. Otherwise, there is a risk of confusion between developers at the requirements level. In fact, the Hybrid Event-B formalism explicitly forbids retiming, by insisting that time progresses in exactly the same way in all models of a development chain. Of course, this viewpoint does not make the need for the complicated, detailed facts necessitated by the retiming approach, go away. Rather, a better place for them would be a more complex refinement retrieve relation, supplemented where needed by suitable retrenchment data.

If we apply this perspective to our development, we identify two different regimes. In the first, the relationship between the level 11 and time-triggered systems is a retrieve relation identity. This applies any time the computing systems can be said to 'have the initiative'. This covers the handle events (because, as we said above, handle events are synchronised with the computing systems), and those events in which the computing systems issue commands to the electromechanical components. In the other, the electromechanical components' sensors feed back to the computing systems, with the attendant delays before response. In the first regime, a retrieve relation that consists of clauses such as:

$$\begin{aligned}&\textit{condition on time and state variables}\\&\Rightarrow\ \mathrm{var}^{11} = \mathrm{var}^{t\text{-}t}\end{aligned}\tag{3}$$

captures what we have been saying above. In this, the *11* and *t-t* superscripts refer to the level 11 and time-triggered versions of the model variable var, and we need a conjunction of such terms to make the relationship between level 11 and the time-triggered system precise.

In the second regime, a retrieve relation that contains many clauses such as:

$$\begin{aligned}&\textit{condition on time and state variables}\\&\Rightarrow\ \overrightarrow{(\mathrm{var}^{11}(\lfloor t/\Delta\rfloor))} = \mathrm{var}^{t\text{-}t}(t)\end{aligned}\tag{4}$$

is needed. This says that at any time t, if the relevant conditions on the time and on various state variables are satisfied, then the value of the time-triggered variable $\mathrm{var}^{t\text{-}t}$ will equal the value of the level 11 variable var^{11}, at a time that corresponds to the last sample time $\lfloor t/\Delta\rfloor$, where Δ is the sampling interval. The optional overrightarrow selects, on a case by case basis, whether it is the actual value at $\lfloor t/\Delta\rfloor$ itself that is required, or its left limit.

Clauses like (4) cover variables for which the behaviour is piecewise constant, only. In our case study, we also have linearly increasing behaviours (for the general electrovalve). For that, the right hand side of (4) must be replaced by a relation between the actual and time-delayed values, according to the relevant rate of change. And that is not all. The linear behaviour of the general electrovalve gets cut off at the lowest and highest values, where it remains constant, till the opposite behaviour is required by the analogical switch. This requires clauses that instead of (4), describe the end effects of these periods of linear behaviour.

If enough effort were invested in identifying all the relevant cases, then a refinement could be proved between level 11 and the time-triggered system. This would depend on a collection of axioms that enforced relationships between the various durations occurring in our development that ensured that the different cases just discussed did not 'crash into each other', ruining the refinement. A less exacting collection of axioms might only enable a retrenchment to be proved, rather than a refinement, and a collection of even weaker axioms might not even permit that. Note that in all cases, there may be many complexities arising from the need to ensure that for each variable, there was always at least one clause in the retrieve relation that enforced the needed relationship between the two models.

What we have discussed is already complicated, and this was for a situation where piecewise constant behaviours predominated. Life gets even more complex when we contemplate situations in which sensors and actuators send values that can vary in more complex and unpredictable ways. The effects of these low-level issues can have a major impact on the stability and predictability of the behaviour of cyber-phyisical systems.

13 Review, modelling patterns, lessons, issues

In the last few sections, we have overviewed the landing gear case study, redone since the *Conf* treatment in the by now more fully developed multi-machine Hybrid Event-B formalism of [10]. The final system in the series of devel-

opment steps was the level 11 system, taking into account that we did not write out the time-triggered system in full. In the level 11 system documentation, we took the additional trouble to write out *all* the details, in contrast to previous levels, where we concentrated on writing out the changed parts of components while abbreviating unchanged parts, and simply copied unchanged components from the preceding level.

As the development progressed, and it became clear that certain techniques worked well to address certain modelling challenges, the amount of work per step increased. While this can be convenient for human comprehension of the development process, it can be the enemy of automatic verification, where small steps tend to be much more digestible for machines.

To exercise the architectural capabilities of multi-machine Hybrid Event-B, we placed each identifiable component in its own machine. This represents one kind of expressivity extreme—each machine was a simple as it could be, but it increased the challenge of keeping all these machines properly related, while striving to maximise independent working. At the opposite end of the expressivity spectrum lies the possibility of putting each computer into a machine, and putting all the physical components into another. Conceptually, this would be feasible for the following reason. Each physical component is designed to have self-contained physical behaviour. Thus, if its pliant behaviour were to be captured by a single, universally enabled pliant event (covering all the different behaviours that it might be required to exhibit in a given application), then all of these pliant events would be independent, and would thus be capable of being conjoined into a single pliant event.

To illustrate this, suppose the state of apparatus A was described by pliant variable x_A, and its behaviour was specified via the ODE $\mathcal{D}\, x_A = \phi_A(x_A \ldots)$. Likewise suppose apparatus B had state x_B, and behaviour given via the ODE $\mathcal{D}\, x_B = \phi_B(x_B \ldots)$. We assume that apparatus A and apparatus B are independent, so, since engineering in general is described using *local realistic* theories, ϕ_A will not depend on x_B and ϕ_B will not depend on x_A. In such a case, we can put ODE $\mathcal{D}\, x_A = \ldots$ in a machine for apparatus A and ODE $\mathcal{D}\, x_B = \ldots$ in a machine for apparatus B.

But equally easily, we could construct one big machine for both apparatus A and apparatus B, and describe the joint pliant behaviour via the vector ODE:

$$\begin{bmatrix} \mathcal{D}\, x_A \\ \mathcal{D}\, x_B \end{bmatrix} = \begin{bmatrix} \phi_B(x_A \ldots) \\ \phi_B(x_B \ldots) \end{bmatrix}$$

In fact, semantically, there is no difference. The semantics for the multiple machine formulation is (behind the scenes) formulated in a global manner, even if the syntactic appearance is partitioned into multiple machines for the convenience of designers. So a conjoined description is perfectly feasible.

Beyond this, we furthermore note that the mode events belonging to all the different physical components could also coexist in the same machine, easily enough. We saw some of the consequences of this in our discussion of the analogical switch above, where we stressed the consequences for automated verification.

13.1 Modelling patterns

In our step-by-step development, we identified a set of useful modelling patterns that appeared widely applicable. We summarise these now.

- **P1** The organisation of a development according to the hypergraph architecture proved to be a major advantage in terms of structural clarity. It permitted many complex invariants to be easily expressed. This gives the concept very wide applicability. The notions of type II invariants, and of global invariants in a project, supplements this by making different types of cross-cutting invariant available.
- **P2** The pattern $(\Sigma_j \text{ no. of actors in } (DL_j/stage_j))$ for variants that track progress through, and ensure termination of, some sub-procedure is very general and ought to be more widely known.
- **P3** Using a component's clock as an adjunct to its state machine proved to be *extremely* convenient in overcoming the complexities inherent in achieving a safe and transparent design, when safety-sensitive activities with extended duration needed to overlap due to too close proximity of external stimuli. Modelling mechanical safety delays using pure state machine techniques would have made the state machines much more cumbersome, and, importantly, made the design much more opaque. Simply adjusting the clock to allow a safety margin of time to elapse before the next required action proved to be an elegant solution.
- **P4** In Hybrid Event-B, stimulus and response normally need to be separated temporally. A useful general purpose pattern for modelling this has the stimulus event synchronised with starting a clock in the response event's machine. This allows the response event itself to be triggered by a timeout or the entry into a time period. This kind of modelling permits many different kinds of cooperation to be expressed.
- **P5** The introduction of faults in a principled way, starting bottom-up from faults in the lowest level components, and using retrenchment to step back from nominal behaviour in a stepwise manner, proved to be a useful

organisational technique for mastering the the complexity of faulty system behaviour.

P6 Once faults had been incorporated as in **P5**, the check back to nominal behaviour when faults were disabled via the retrenchment architecture, offered the possibility of useful additional confirmation of correct design.

13.2 Issues and questions

As well as these positive outcomes of the exercise, detailed examination of the issues thrown up by the case study revealed a number of areas for further development of the Hybrid Event-B framework itself. Among these we can give the following.

Q1 In contexts in which there is significant use of replicated components, such as the present one, the availability of expressive and flexible indexing techniques for defining collections of such components is important for expressivity and succinctness. In the present development, this was addressed by meta level indexing, but it would be needed in the formal framework in a mature formalism.

Q2 In contexts in which there is significant possibility of component failure, the rather hardwired synchronisation mechanisms in the present framework can prove inadequate. An example of this occurs when we expect a synchronisation under nominal conditions and yet accept a partial version if one of the machines has failed. It is fairly clear that the present development has not fully taken on board all the requirements that arise in this context, for example, when synchronisation is only needed when the *last* of a set of components that behave asynchronously has responded.

Q3 A possible contribution to **Q2** is to allow events to have more than one name, e.g. *EvName1* ALSO *EvName2* ALSO *EvName3* STATUS ..., etc. Then, the different names of the same event body could appear in different synchronisation schemes. This, by itself, is not enough though.

Q4 The present development used state variables exclusively. It neglected the possibilities made available using the I/O semantics of synchronised events. This may lead to certain advantages, and is a possibility that should be explored.

Q5 The use of clocks as part of the state machine (**P3**) is very convenient but embodies assumptions that only certain things happen within particular time ranges. Different tradeoffs between this approach and using more detailed state machines should be explored, and their pros and cons should be understood.[21]

Q6 Although type II invariants were available if needed, the hypergraph architecture rendered them unnecessary in this development. Is this a general truth or just a fortunate property of the present case study? In the opposite direction, is there a requirement for even more general, or more complex, cross-cutting invariant patterns in other problem domains?

Q7 Are there ways of further integrating the modelling of nominal and faulty behaviours, beyond what has been suggested by the disabling-of-faults technique?

Q8 The most appropriate handling of the implementation of complex event-based systems in terms of time-triggered systems raises many questions. Although the essential mathematical challenges are not novel, the most appropriate way of packaging the important elements so that they can be deployed with the least risk, and maximum engineering benefit, remains as a nontrivial challenge for the future. The simple predominantly piecewise constant behaviours of the present development already generate a number of complex detailed cases, and these do not even touch on the additional frequency domain questions that arise when sensors and actuators have to deal with rapidly varying signals. Reconciling the established frequency domain techniques with state-based refinement frameworks remains as a significant open question for future research.

Q9 Of course, doing an exercise like the present one by hand is *really* tricky—playing the role of a human IDE is extremely error prone. Almost every re-reading of some fragment of the development revealed another bug (although typically (and reassuringly), the overwhelming majority of such bugs were of a kind that would easily be picked up mechanically by surface syntactic checking). Proper machine support is obviously vital when doing such a development in a serious way.

We regard the above questions as outcomes of this study that are just as valuable as the positive results of the modelling exercise itself.

14 Conclusions

This paper has described the development of the landing gear case study in the multi-machine Hybrid Event-B framework, as it is defined in [10]. We deliberately approached the development in a way that would put the formalism under the

[21] As an explicit example, consider that in the level 11 model, the $anomaly_k$ cases were called on (in principle) quite short timeouts, in order not to allow clk_Handle_k to overrun into the next nominal event. More finegrained state modelling (as noted earlier) could obviously avoid this problem if knowledge of equipment behaviour justified waiting longer, in case some physical component responded rather late.

maximum stress, so as to identify what worked well, and what issues connected with the formalism might warrant further investigation. We surveyed the resulting good and bad points in the previous section.

In the earlier part of the work, in the nominal development where it was legitimate to rely on the perfect working of all the components, it was easy to specify the required behaviour in a very tightly controlled way. In the B-Method generally, this shows itself as the opportunity to write many precise invariants concerning the behaviour. However, when the faulty regime is contemplated, the vast majority of these stronger properties no longer hold, so the intriguing question arises as to how one can combine the two worlds in a way that might be most beneficial for the final delivered system. One of the opportunities offered by this case study was the chance to examine this question in a context that was relatively convincing regarding the complexities of a real application.

On this point, the author utilised his established work on retrenchment to try to reconcile the two worlds. Especially in the light of the convenient naming convention used in this development, it proved possible to reintroduce the stronger nominal invariants into a version of the faulty regime which had the fault injection mechanism switched off. A practical version of this might well necessitate the inclusion of additional axioms/invariants expressing the non-occurrence of faults, to facilitate the verification process.

To summarise, the landing gear case study has proved to be an excellent vehicle for significantly exercising the Hybrid Event-B framework. Since it was conceived independently of the framework, it does not fall into the trap of only including modelling issues anticipated to be convenient for the formalism, as a case study conceived by the framework's designers might have done, even despite their best efforts. All this bodes well for the development of tool support for the framework.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Akers, A., Gassman, M., Smith, R.: Hydraulic Power System Analysis. CRC Press, Boca Raton (2010)
2. Banach, R.: Retrenchment for Event-B: UseCase-wise development and Rodin integration. Formal Aspects Comput. **23**, 113–131 (2011)
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.: Pliant Modalities in Hybrid Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods, vol. 8051, pp. 37–53. Springer, Heidelberg (2013)
5. Banach, R.: Invariant guided system decomposition. In: Ait Ameur, Y., Schewe K.-D. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. LNCS, vol. 8477, pp. 271–276. Springer, Heidelberg (2014)
6. Banach, R.: The landing gear system case study in hybrid Event-B. In: Boniol, F., Weils, V., Ait Ameur, Y., Schewe K.-D. (eds.) ABZ 2014: The Landing Gear Case Study. CCIS, vol. 433, pp. 126–141. Springer (2014)
7. Banach, R., Bozzano, M.: The mechanical generation of fault trees for reactive systems via retrenchment I: combinational circuits. Formal Aspects Comput. **25**, 573–607 (2013)
8. Banach, R., Bozzano, M.: The mechanical generation of fault trees for reactive systems via retrenchment II: clocked and feedback circuits. Formal Aspects Comput. **25**, 609–657 (2013)
9. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core Hybrid Event-B I: single Hybrid Event-B machines. Sci. Comput. Program. **105**, 92–123 (2015)
10. Banach, R., Butler, M., Qin, S., Zhu, H.: Core Hybrid Event-B II: Multiple Cooperating Hybrid Event-B Machines (Submitted) (2014)
11. Banach, R., Jeske, C.: Retrenchment and refinement interworking: the Tower theorems. Math. Struct. Comput. Sci. **25**, 135–202 (2014)
12. Banach, R., Jeske, C., Poppleton, M.: Composition mechanisms for retrenchment. J. Log. Algebraic Program. **75**, 209–229 (2008)
13. Banach, R., Poppleton, M.: Retrenchment: an engineering variation on refinement. In: Bert, D. (ed.) B'98: Recent Advances in the Development and Use of the B Method. LNCS, vol. 1393, pp. 129–147. Springer, Heidelberg (1998)
14. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and theoretical underpinnings of retrenchment. Sci. Comput. Program. **67**, 301–329 (2007)
15. Barolli, L., Takizawa, M., Hussain, F.: Special issue on emerging trends in cyber-physical systems. J. Ambient Intell. Hum. Comput. **2**, 249–250 (2011)
16. Beauquier, D., Slissenko, A.: On Semantics of Algorithms with Continuous Time. Tech. Rep. TR-LACL-1997-15, LACL, University of Paris-12 (1997)
17. Boniol, F., Wiels, V.: The Landing Gear System Case Study. In: ABZ Case Study, Communications in Computer Information Science, vol. 433. Springer (2014)
18. Butler, M.: Decomposition structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) Integrated Formal Methods. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009)
19. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and tools for hybrid systems design. Found. Trends Electron. Des. Autom. **1**, 1–193 (2006)
20. Davoren, J.M.: Epsilon-tubes and generalized skorokhod metrics for hybrid paths spaces. In: Majumdar, R., Tabuada, P. (eds.) Hybrid Systems: Computation and Control. LNCS, vol. 5469, pp. 135–149. Springer, Heidelberg (2009)
21. Hallerstede, S., Hoang, T.: Refinement by interface instantiation. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. LNCS, vol. 7316, pp. 223–237. Springer, Heidelberg (2012)
22. Ionel, I.: Pumps and Pumping. Elsevier, Amsterdam (1986)
23. Jeffords, R., Heitmeyer, C., Archer, M., Leonard, E.: A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009: Formal Methods. LNCS, vol. 5850, pp. 173–189. Springer, Heidelberg (2009)
24. Le Guilly, T., Olsen, P., Ravn, A., Skou, A.: Component reliability analysis. In: Sekerinski (ed.) Proc. FASDS-14, dedicated to Kaisa Sere. Springer (2014, to appear)
25. Manring, N.: Hydraulic Control Systems. Wiley, New York (2005)

26. National Science and Technology Council: Trustworthy Cyberspace: Strategic plan for the Federal Cybersecurity Research and Development Program (2011). http://www.whitehouse.gov/sites/default/files/microsites/ostp/fed_cybersecurity_rd_strategic_plan_2011.pdf
27. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer, Berlin (2010)
28. Quesel, J.D., Fränzle, M., Damm, W.: Crossing the bridge between similar games. In: Fahrenberg, U., Tripakis, S. (eds.) Formal Modeling and Analysis of Timed Systems. LNCS, vol. 6919, pp. 160–176. Springer, Heidelberg (2011)
29. Retrenchment Homepage: http://www.cs.man.ac.uk/~banach/retrenchment
30. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition tool for Event-B. Softw. Pract. Experience **41**, 199–208 (2011)
31. Sztipanovits, J.: Model Integration and Cyber Physical Systems: A Semantics Perspective. In: Butler, Schulte (eds.) Proc. FM-11. Springer, LNCS 6664, p. 1 (2011). Invited talk, FM 2011. http://sites.lero.ie/download.aspx?f=Sztipanovits-Keynote.pdf
32. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer, Berlin (2009)
33. 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/