Modelling Hybrid Systems in Event-B and Hybrid Event-B: A Comparison of Water Tanks

Richard Banach[1] and Michael Butler[2]

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk
[2]School of Electronics and Computer Science, University of Southampton,
Highfield, Southampton, SO17 1BJ, U.K.
mjb@ecs.soton.ac.uk

Abstract. Hybrid and cyberphysical systems pose significant challenges for a formal development formalism based on pure discrete events. This paper compares the capabilities of (conventional) Event-B for modelling such systems with the corresponding capabilities of the Hybrid Event-B formalism, whose design was intended expressly for such systems. We do the comparison in the context of a simple water tank example, in which filling and emptying take place at different rates, necessitating a control strategy to ensure that the safety invariants are maintained. The comparative case study is followed by a general discussion of issues in which the two approaches reveal different strengths and weaknesses. It is seen that restricting to Event-B means handling many more things at the meta level, i.e. by the user, than is the case with its Hybrid counterpart.

1 Introduction

Hybrid [9] and cyberphysical [10] systems pose significant challenges for a formal development formalism based on discrete events. A number of compromises are needed in order to allow a discrete event formalism to relate to the important continuous aspects of the behaviour of such systems. Formalisms that are more purpose built address such concerns more easily. This paper compares the capabilities of (conventional) Event-B (**EB**) for modelling such systems with the capabilities of the more purposely designed Hybrid Event-B (**HEB**). We do the comparison in the context of a simple water tank example, in which filling and emptying take place at different rates, necessitating a control strategy to ensure that the required safety invariants are maintained. This familiar scenario makes the discussion easier to follow. The example was modelled using **EB** in [8] using facilities built in **EB** for expressing certain continuous features of behaviour.

The rest of this paper is as follows. Section 2 overviews the **HEB** framework, and shows how **EB** results from forgetting the novel elements of **HEB**. Section 3 briefly recalls the water tank problem. Then Section 4 overviews the development in [8], which is a detailed study of the water tank example in the **EB** framework. Section 5 looks at a comparative (though on-paper-only) study of the same problem in **HEB**. Section 6 then embarks on a general comparison of the pros and cons of the **EB** and **HEB** approaches. Section 7 concludes.

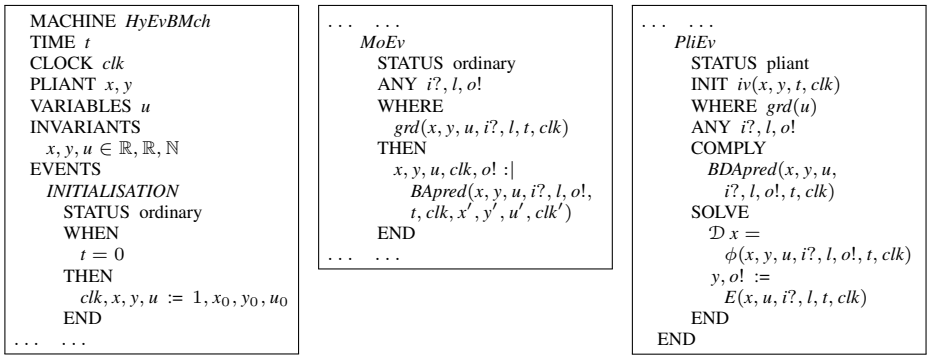

Fig. 1. A schematic Hybrid Event-B machine.

2 An Outline of Hybrid Event-B, and of Event-B

In this section we outline Event-B and Hybrid Event-B for a single machine. Because it is more complex, we describe Hybrid Event-B first via Fig. 1, and show how it reduces to Event-B (which of course came earlier) by erasing the more recently added elements.

Fig. 1 shows a schematic Hybrid Event-B machine. It starts with declarations of time and of a clock. Time is a first class citizen in that all variables are functions of time (which is read-only), explicitly or implicitly. Clocks are assumed to increase like time, but may be set during mode events. Variables are of two kinds. There are mode variables (like u) which take their values in discrete sets and change their values via discontinuous assignment in mode events. There are also pliant variables (such as x, y), declared in the PLIANT clause, which typically take their values in topologically dense sets (normally $\mathbb{R}$) and which are allowed to change continuously, such change being specified via pliant events.

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 similarly are predicates that are required to hold *at all moments of time* during a run.

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

Mode events are analogues of events in discrete Event-B. They can assign all machine variables (except time). The schematic *MoEv* of Fig. 1, has parameters $i?, l, o!$, (input, local, and an output), and a guard *grd*. It also has the after-value assignment specified by the before-after predicate *BApred*, which can specify the after-values of all variables (except time, inputs and locals).

Pliant events are new. They specify the continuous evolution of the pliant variables over an interval of time. Fig. 1 has a schematic pliant event *PliEv*. There are two guards: *iv*, for specifying enabling conditions on the pliant variables, clocks, and time; and *grd*, for specifying enabling conditions on the mode variables.

The body of a pliant event contains three parameters $i?, l, o!$, (input, local, and output, again) 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 contains direct assignments, e.g. of y and output $o!$ (to time dependent functions); and differential equations, e.g. specifying x via an ODE (with $\mathcal{D}$ as the time derivative).

The COMPLY clause can be used to express any additional constraints that are required to hold during the pliant event via the before-during-and-after predicate *BDApred*. Typically, constraints on the permitted ranges of the pliant variables, can be placed here. The COMPLY clause can also specify at an abstract level, e.g. stating safety properties for the event without going into detail.

Briefly, the semantics of a Hybrid Event-B machine 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.

Time is modeled as an interval $\mathcal{T}$ of the reals. A run starts at some initial moment of time, t_0 say, and lasts either for a finite time, or indefinitely. The duration of the run $\mathcal{T}$, breaks up into a succession of left-closed right-open subintervals: $\mathcal{T} = [t_0 \ldots t_1), [t_1 \ldots t_2), [t_2 \ldots t_3), \ldots$. 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.

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). 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).[1] (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 several semantic issues that we have glossed over in the framework just sketched. We refer to [5] for a more detailed presentation (and to [6] for the extension to multiple machines). The presentation just given is quite close to the modern formulation of hybrid systems. See e.g. [15, 13], or [9] for a perspective stretching further back.

[1] 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.

If, from Fig. 1, we erase time, clocks, pliant variables and pliant events, we arrive at a skeleton (conventional) Event-B machine. This simple erasure process illustrates (in reverse) the way that Hybrid Event-B has been designed as a clean extension of the original Event-B framework. The only difference of note is that now —at least according to the (conventional) way that Event-B is interpreted in the physical world— (the mode) events (left behind by the erasure) execute *lazily*, i.e. *not* at the instant they become enabled (which is, of course, the moment of execution of the previous event).[2]

3 The Water Tank Problem

The water tank problem is a familiar testing ground for approaches to control problems in event based frameworks like the B-Method. The purpose of the water tank controller is to maintain the water level in the tank between a low and a high level. There is a mechanism, assumed to act continually, by which water drains from the tank. To counteract this, there is a filling mechanism, acting faster than the draining mechanism, that can be activated at the behest of the controller to refill the tank when the water level has become too low — it is deactivated once the level has become high enough.

4 The Event-B Water Tank Development

In [8] there is a development of the water tank in **EB**. Since **EB** has no inbuilt continuous facilities, a considerable amount of continuous infrastructure had to be built behind the scenes using the theory plugin of the Rodin tool [3, 14]. A fragment of this, the **EB** pattern for a pliant event in the style used in [8], is shown in Fig. 2. This treats update to continuous behaviour monolithically (i.e. by adding the whole piece from *clk* to *t* in a single action).

EB_*PliEv*
ANY t, f
WHERE
$t \geq clk + \epsilon$
$f \in ctsF(clk, t)$
$f(clk) = m(clk)$
$P(f)$
THEN
$clk, m := t, m \cup f$
END

Fig. 2. The **EB** pattern for representing a pliant event.

In more detail, there is a clock *clk*, and the presumption is that the event describes what happens in a time interval following clock value *clk*. A parameter t is introduced, greater than *clk* by at least ϵ (to prevent Zeno behaviour, though Zeno behaviour would not be detectable, nor cause any upset, in an Event-B proof). Another parameter f, describes the graph of a *con*tinuous*F*unction on the interval $[clk \ldots t]$ by which the function m, defined hitherto only on the interval $[0 \ldots clk]$, is to be extended. Defining functions set theoretically by their graphs, the extension of the function m is just the union of its previous value and f. Of course, *clk* must also be updated to t, ready for the next increment. For the function m to be continuous, its preceding final value must match the initial value of the increment f, as stated in the guard $m(clk) = f(clk)$. Finally, $P(f)$ expresses any further properties that the increment f is required to satisfy.

[2] We observe however, that it is considerably easier to simulate lazy execution semantics using eager semantics (e.g. via guards that depend on nondeterministically/probabilistically set auxiliary variables), than to achieve eager behaviour using lazy semantics.

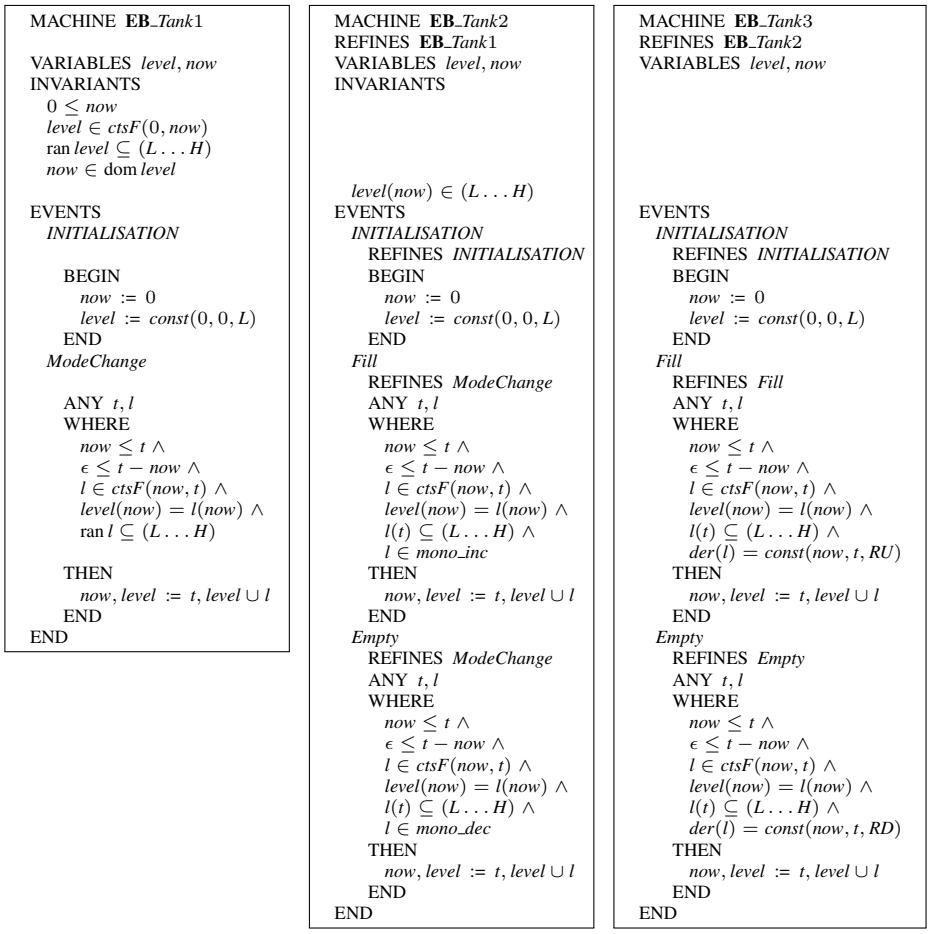

Fig. 3. Event-B machines for the water tank.

In Fig. 3 we see the main thread of the **EB** water tank, essentially as in [8]. Aside from what is shown, there are two contexts $c1$ and $c2$, which introduce various constants used in the development.

The Fig. 3 development starts with **EB**_*Tank*1. This introduces the water *level* variable, as well as the *now* variable (the analogue of *clk* in Fig. 2). Initialisation fixes *now* at 0 and *level* to the *const*ant function over the degenerate closed interval $[0 \ldots 0]$ with value L, the lower water level.

There is one event *ModeChange*, which illustrates how continuous behaviour is handled in the **EB** modelling style of [8]. As is clear, this is a simple instantiation of the pattern of Fig. 2.

From its name, one can infer that *ModeChange* is intended to model the transitions between filling and emptying episodes. However, there is nothing in its definition that forces this — the event merely extends the *level* function, defined by its graph, by some non-empty chunk into the future (that obeys the restriction on its range).

```
MACHINE EB_Tank30
REFINES EB_Tank2
VARIABLES
  level, now, step, slevel, mode
INVARIANTS
  step ∈ ℝ
  slevel ∈ ctsF(now, step)
  mode = UP ⇒
        slevel ∈ mono_inc
  mode = DOWN ⇒
        slevel ∈ mono_dec
  level(now) = slevel(now)
  slevel(step) = (L . . . H)
EVENTS
  INITIALISATION
    REFINES INITIALISATION
    BEGIN
      now := 0
      level := const(0, 0, L)
      step := 0
      slevel := const(0, 0, L)
      mode := UP
    END
  StepUp
    ANY l
    WHERE
      mode = UP ∧
      slevel(step) ≤ HT ∧
      l ∈ ctsF(step, step + P) ∧
      slevel(step) = l(step) ∧
      l ∈ mono_inc ∧
      l(step) ≤ l(step + P) ∧
      l(step + P) ≤
        l(step) + (RU × P) ∧
      l(step + P) ≤ H
    THEN
      step := step + P
      slevel := slevel ∪ l
    END
  EndFill
    REFINES Fill
    WHEN
      mode = UP ∧
      ¬ (slevel(step) ≤ HT)
    WITH
      l = slevel
      t = step
    THEN
      now := step
      level := level ∪ slevel
      mode := DOWN
      slevel := const(step, step,
                slevel(step))
    END
  StepDown  . . .  . . .
  EndEmpty  . . .  . . .
END
```

Fig. 4. The **EB**_*Tank*30 machine.

EB_*Tank*1 is refined to **EB**_*Tank*2. The variables are the same, and another invariant $level(now) \in (L \ldots H)$ is introduced to aid proof (of course, it follows mathematically from the earlier invariants $level \in ctsF(0, now)$ and $\mathrm{ran}\, level \subseteq (L \ldots H)$). The previous event *ModeChange*, is refined to two separate events, *Fill* and *Empty*. These events have additional contraints in their guards, $l \in$ *mono*tonically_*inc*reasing_functions for *Fill*, and $l \in$ *mono*tonically_*dec*reasing_functions for *Empty*. So each chunk that increments the *level* function is increasing or decreasing, but cannot oscillate.

Again, from their names, we might infer that *Fill* and *Empty* are intended to model the full filling and emptying episodes, which we expect to alternate. But there is no requirement that filling results in a *level* anywhere near H, nor analogously for emptying; also there is nothing to prevent successive filling, or successive emptying episodes.

EB_*Tank*2 is refined to **EB**_*Tank*3. The variables are the same, and there are no new invariants. The only change now is that monotonic behaviour is implemented by an axiomatic form of an ordinary differential equation. Thus, $l \in$ *mono_inc* in *Fill* is replaced by $der(l) = const(now, t, RU)$, which says that the derivative of l is a constant function over the interval $[now \ldots t]$, with value *R*ate*U*p. This, and the analogously modified *Empty*, covers what is shown in Fig. 3.

Aside from the machines in Fig. 3, there is a further machine, *Tank*30, in the development discussed in [8]. This is also a refinement of **EB**_*Tank*2, although a different one. This one models a putative implementation of **EB**_*Tank*2 using a time triggered loop. A new variable *step* is introduced, whose job, like that of *now*, is to model increments of time, but on this occasion small ones, whose duration is determined by a constant P. Another new variable *slevel* models the small increments or decrements to the water level accrued in each interval of length P. The events modelling these small increments or decrements also follow the pattern described earlier. Most of this machine is shown in Fig. 4 (the parts omitted are the details of events *StepDown* and *EndEmpty*, which are straightforward analogues of events *StepUp* and *EndFill*).

Unlike the models of Fig. 3, there is a variable $mode \in \{UP, DOWN\}$ to enforce filling or emptying behaviour until the boundary values are approached. And since, when using fixed time increments of length P, it is not realistic to expect filling and emptying to reach the limits H or L 'on the nose', thresholds HT and LT are introduced (respec-

tively less than and greater than H and L), upon reaching which, the mode changes. Technically, the 'intermediate' filling and emptying events, *StepUp* and *StepDown*, are 'new' events, refining a notional skip in **EB**_*Tank*2. The 'endpoint' events, *EndFill* and *EndEmpty*, refine *Fill* and *Empty* in **EB**_*Tank*2, determining the needed values of *now* and *level* to achieve refinement.[3]

Finally, we comment on the methodology used to arrive at these results. The properties of the reals, and of real functions, were axiomatised using the theory plugin of the Rodin tool [3, 14]. One aspect of this is that derivatives, expressed using axioms for *der*, are axiomatised as belonging to the continuous functions *ctsF*, for convenience (see [8]). If we then look at the way that these are used in **EB**_*Tank*3, we see that the derivatives specified are always constant functions. But filling episodes have a positive derivative of the l function, and emptying episodes give l a negative derivative. Joining two such episodes cannot yield a continuous derivative.

This apparent contradiction is resolved by noticing that each element of *ctsF* is only defined with respect to its domain. Thus, a function f_1 defined on $[t_1 \ldots t_2]$ may have one continuous derivative, and a different function f_2 defined on $[t_2 \ldots t_3]$ may have a different continuous derivative. Even if f_1 and f_2 can be joined at t_2, the exclusive use of closed intervals for domains of continuous behaviour (which happens quite commonly in formulations of hybrid systems, see e.g. [15, 13, 9]) does not enable us to deduce that their derivatives can be joined at t_2. While consistent, the consequence of this is that the joined $f_1 \cup f_2$ cannot be regarded as a differentiable function on $[t_1 \ldots t_3]$, and in fact, an attempt to regard it as such would lead to multiple values of the putative derivative at t_2. While relatively innocuous in the present example, it indicates a number of things. The first is that what is true can depend delicately on the axioms adopted. The second is that care needs to be taken in case the unexpected consequences of the axioms lead one astray. The third is a caution regarding the scalability of such an approach, as the number of counterintuitive cases proliferates.

5 The Hybrid Event-B Water Tank Development

In Fig. 5 we see a development of the water tank problem in **HEB**. It consists of three machines: **HEB**_*TankAbs*, an abstract formulation, which is refined by **HEB**_*TankMon* which includes the pump, and which is in turn refined by **HEB**_*TankODE*. These are relatively straightforward analogues of the machines **EB**_*Tank*1, **EB**_*Tank*2, **EB**_*Tank*3 in the last section. The main difference between the two treatments is that in **HEB**, functions of time are manipulated solely using expressions for their values at any individual instant, and not *en bloc*, as graphs over (some portion of) their domain. This aligns the way that pliant and mode updates can be regarded, and simplifies many less trivial matters. For ease of comparison, we keep the names of constants in the two treatments the same, but alter other names to aid distinguishability.

[3] The use of thresholds *HT* and *LT* rather than the precise limits H and L, correlates with the absence of guards to check reaching H or L in the corresponding **EB**_*Tank*1 and **EB**_*Tank*2 events. However, since the behaviour stipulated is *nondeterministic* monotonic, adding an extra constraint to demand that the behaviour exactly reached the required limit in events *EndFill* and *EndEmpty* would be perfectly feasible (mathematically, if perhaps not practically).

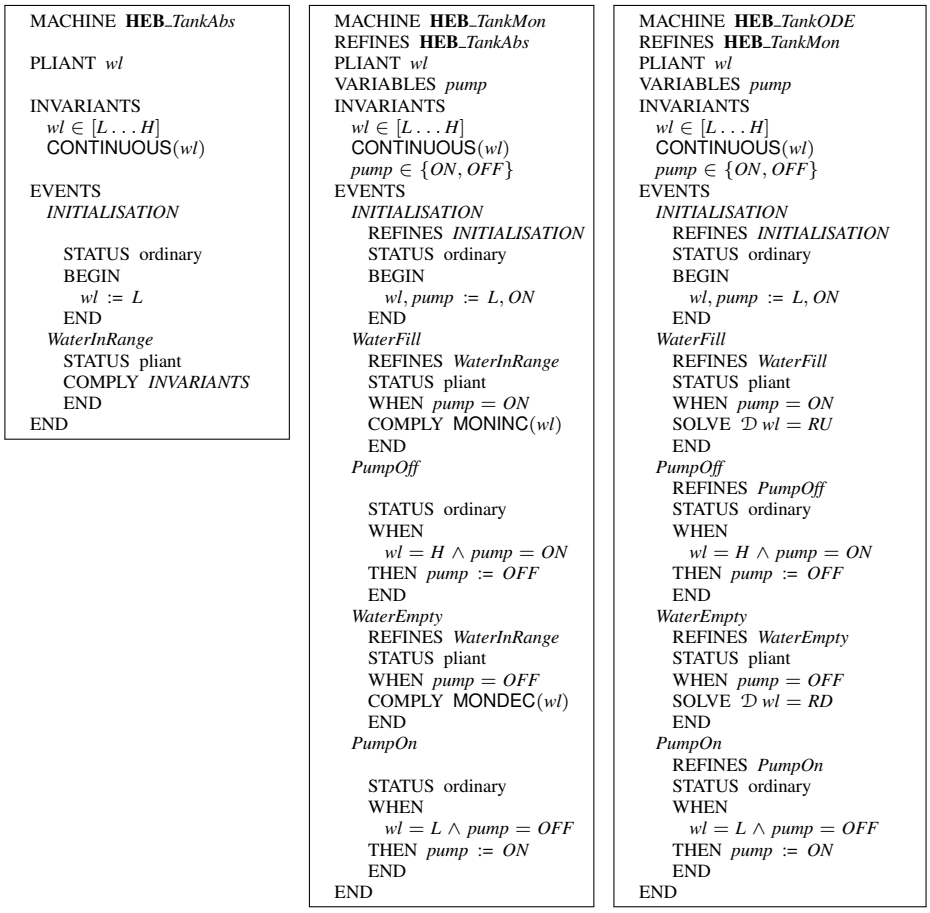

Fig. 5. Hybrid Event-B machines for the water tank.

HEB_*TankAbs* has only the water level variable *wl*, which is pliant, taking values in $\mathbb{R}$. The behaviour of *wl* is required to be CONTINUOUS (to prevent discontinuous jumps), and the nontrivial invariant $wl \in [L \ldots H]$ confines the water level to the real closed interval $[L \ldots H]$. In **HEB**, invariants are properties that have to hold at all times, so $wl \in [L \ldots H]$ is sufficient to express the safety property that *wl* is required to never leave $[L \ldots H]$. The only non-*INITIALISATION* event in **HEB**_*TankAbs* is the pliant event *WaterInRange*. This merely requires the behaviour to COMPLY (with the) *INVARIANTS*. So **HEB**_*TankAbs* specifies the required safety property and does not concern itself with how that safety property is to be maintained. The ability to do this properly in a hybrid/cyberphysical setting is an important feature of development in **HEB**. So **HEB**_*TankAbs* mirrors **EB**_*Tank*1 quite closely.

The next machine **HEB**_*TankMon*, starts to engage with how the key invariant is maintained. It introduces the **EB**-style mode variable $pump \in \{ON, OFF\}$. The pump is turned on and off by mode events *PumpOn* and *PumpOff*. These are like **EB** events aside from their eager behaviour — they execute as soon as their guards become true.

Again illustrating the ability to postpone implementation details, the behaviour of *wl* in the presence of the pump is merely specified to be MONotonically DECreasing when the pump is *OFF*, and to be MONotonically INCreasing when it is *ON*: in pliant events *WaterEmpty* and *WaterFill* respectively. Note that the *pump* variable, introduced earlier than in **EB**, prevents successive filling or successive emptying episodes (unless we had additional mode events to interleave them, to conform with (1) and (2)).

Importantly, **HEB**_*TankMon* is a formal refinement of **HEB**_*TankAbs* according to the detailed definition in [5], as we would wish. Both of *WaterEmpty* and *WaterFill* refine the abstract *WaterInRange*, in that monotonic continuous behaviour is a refinement of continuous behaviour. The relevant PO expresses this by saying the following. For all times t during an execution of a concrete event, *WaterFill* say, that started at some time $\mathbb{t}_{\mathrm{L}}$ say, if the value that *wl* reached at t due to executing *WaterFill* from its starting value $wl(\mathbb{t}_{\mathrm{L}})$ was $wl(t)$, then the same value can be reached by executing the abstract event *WaterInRange* from $\mathbb{t}_{\mathrm{L}}$ to t.

Mode events *PumpOn* and *PumpOff* are 'new' events in **EB** parlance, updating only the 'new' mode variable *pump*, so there is no change to abstract variable *wl* when they execute. However, there is no VARIANT that they decrease when they execute. The abstract event that they relinquish control to upon completion is the immediately succeeding pliant event, *WaterEmpty* for *PumpOff* or *WaterFill* for *PumpOn*. An auxiliary (pliant) variable could be introduced that was increased by these events and decreased by the mode events to create a variant, but this would clutter the model. Thus we see that **HEB**_*TankMon* mirrors **EB**_*Tank*2 quite closely, aside from the presence of *pump* and its controlling events, which fix the durations of the monotonic episodes to be maximal, and ensures that switching takes place at the extreme values of the range.

Machine **HEB**_*TankODE* refines **EB**_*TankMon*. This time the various events are refined 1-1, so there are no 'new' events to worry about. The monotonic continuous behaviour of *WaterEmpty* and *WaterFill* is further refined to be given by ODEs in which the derivative of the water level variable *wl* is *RD* for *WaterEmpty* and *RU* for *WaterFill*, as in **EB**. This appears in the SOLVE clauses of these events. Once more, **HEB**_*TankODE* mirrors **EB**_*Tank*3 quite closely, aside from issues concerning *pump*, which we have discussed already.

Supplementing the machines of Fig. 5, machines analogous to the **EB** *Tank*30 machine appear in Fig. 6. Machine **HEB**_*TankTTL* is a time triggered development of **HEB**_*TankMon*, and comparing it with *Tank*30 is instructive. Note that there are no new variables, just new behaviour of events. Thus *WaterFill* is refined to *WaterFillNormal* and to *WaterFillEnd*. The former of these is enabled when the water level is below the threshold *HT*. It demands increasing *wl* behaviour, but restricted to a filling rate no greater than *RU*. Occurrences of *WaterFillNormal* are interleaved by occurrences of mode event *WaterFillObs*, which runs at times that are multiples of *P*, provided the water level is not actually *H* itself. Since *WaterFillNormal* is increasing *wl*, *WaterFillObs* merely skips. Once above *HT*, *WaterFillEnd* runs. This is like *WaterFillNormal* except for an additional condition insisting that *wl* hits *H* at the end of the interval.[4] And once *wl* has reached *H*, *PumpOff* runs, as previously. While this design is unimpeachable mathematically, it is, of course, much more questionable from a practical perspective,

[4] The constraint is consistent provided the various constants are suitably related, of course.

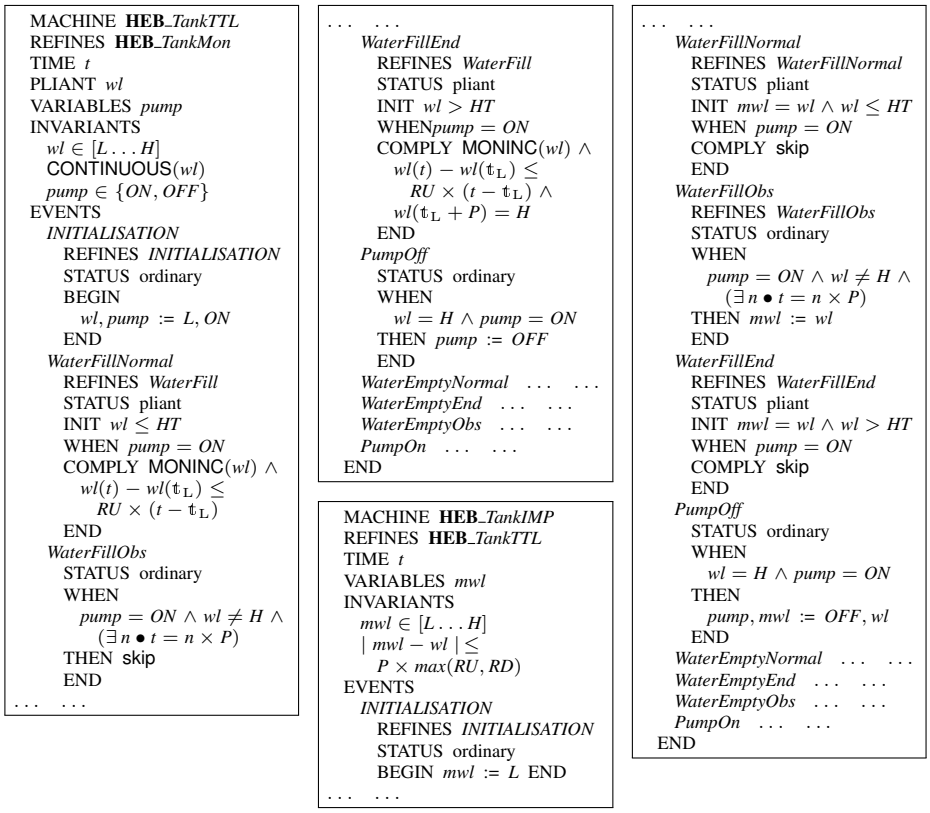

Fig. 6. The **HEB**_*TankTTL* and **HEB**_*TankIMP* machines.

as we pointed out in footnote 3. It does have the virtue though, of providing a straightforward refinement from **HEB**_*TankMon*. Machine **HEB**_*TankTTL* is completed by events *WaterEmptyNormal*, *WaterEmptyEnd*, *WaterEmptyObs*, *PumpOn*, which do the same as the preceding, but for the emptying phase.

Machine **HEB**_*TankTTL* is data refined to **HEB**_*TankIMP* on the right of Fig. 6. This 'implementation' machine illustrates the refinement of *pliant behaviour interleaved by mode* skips, to *pliant* skips *interleaved by mode updates* — a major aim of **HEB** is to allow such a passage from a high level continuous design to a discrete, digital implementation. A fresh variable *mwl* (monitored water level) is introduced, inc/decremented at each of the mode events. Observing *wl* and updating *mwl* at each multiple of *P* enables the invariant $\mid mwl - wl \mid \leq P \times max(RU, RD)$ to be maintained, attesting to the reasonableness of the digital implementation.

What has been achieved by formulating the development in the **HEB** way compared to the **EB** way? Firstly, there is a certain fluency in referring to continuous behaviour via expressions that denote instantaneous values rather than having to assemble and disassemble graphs of functions (but only in the continuous case). Secondly, there are issues of potential semantic subtlety. We saw an example in the discussion of the differential

properties of the *level* function in **EB**_*Tank*3: it was not formally differentiable globally, but consisted of differentiable monotonic pieces, leading to the join points having more than one derivative value, despite these being 'kink' points of the function. In **HEB** such matters are handled *ab inito* in the semantics, by the use of closed/open intervals and the Carathéodory formulation of differential equations and derivatives (which are only required to be defined almost everywhere). Thirdly, there is also the fluency of the passage from pliant behaviour interleaved by mode skips to pliant skips interleaved by mode updates. Discussion of further and more general matters appears in the next section.

6 Event-B versus Hybrid Event-B

Based on the previous **EB** and **HEB** developments, we can draw some comparisons between the two approaches for modelling and formally refining hybrid systems.

1. First and foremost, **EB** has a well developed existing tool, whereas for **HEB**, tool development is, as yet, an aspiration. Having an existing tool is of inestimable benefit when you need to get the job done.

2. In an **EB** development, real time has to be modelled as a normal state variable. This imposes a responsibility on the model writer to not abuse the capabilities this offers. In truth, time is (in physical parlance) an *independent variable* — whereas other state variables correspond (physically) to *dependent variables*. From a linguistic formalism point of view, staying faithful to the physical reality means that time has to be a read-only variable, and that all other variables have to be functions of time. In an **EB** context, it is down to the self-discipline of the model writer to reflect these properties properly. Clearly it is possible to transgress them and to write unphysical models. In **HEB** these realities are hardwired into the syntax and semantics, making it impossible for the model writer to violate them.

3. An analogue of point **2** concerns the mathematical equipment of **EB** and **HEB**. In **EB** all mathematical objects beyond those needed for discrete modelling need to be axiomatised, typically using the theory plugin of the Rodin tool [3, 14]. Although this framework is agnostic regarding the level of abstraction of the concepts being axiomatised, existing work emphasises a bottom up approach (as in the case study above). This potentially creates a lot of work before the level of abstraction needed for applications is reached, increasing risk.

The **HEB** perspective on this is to design the theoretical foundations of the semantics in a way that best suits the needs of applications engineering, giving system developers a mental model that is clear and easy to grasp, and, importantly, is free from unexpected surprises (such as the two-valued 'derivative' discussed earlier). The aim would be to internalise the world of continuous mathematics with the same level of care and consistency as the Rodin tool currently supplies for discrete mathematics and logic, and to supplement it via extensive imported support from external tools such as *Mathematica* [12] for calculational purposes. The facility for user designed rules and axiom schemes would be retained for specialised purposes, but would not be the default approach for continuous mathematics.

4. A specific example of the general remarks in the preceding point lies in the contrast between the explicit construction of functions as relations, manipulated via their graphs in Section 4 and their representation as expressions based on values of variables at a single (arbitrary) element of their time domain in Section 5.

5. Connected with the previous point is the observation that in **EB**, the discrete and continuous updates have to be handled by different means. Thus, discrete transitions are written down using (in effect, pairs of) state expressions, referred to via syntax such as $xs := E(xs)$, with the accepted conventions surrounding the syntactic machinery enabling the relevant expressions to be discerned. For continuous transitions though, because the **EB** framework offers no alternative syntax for update than that which is used for discrete transitions, updates to continuous behaviour have to be handled by updating the relation describing (the function of time that is) the continuous behaviour as a whole, in one action. Section 4 offers many examples. The discrete analogue of such an approach would be to update (in one action), for a discrete variable x, a non-trivial portion of its trace during an execution, i.e. to update say $\langle x_{i-1}, x_i \ldots x_{i+k} \rangle$, as a whole. (Aside from anything else, this would require the introduction into every model of an index variable (incremented at each event occurrence), as well as suitable history variables.)

By contrast, **HEB** provides special purpose syntactic machinery (via the COMPLY and SOLVE clauses) to specify continuous update incrementally and microscopically, rather than macroscopically, which is significant from an expressivity standpoint. As most physical models specify behaviour in a microscopic way (usually via differential equations etc.), being able to write these directly in the formal framework aids the ability to specify in a manner as close to application domain concerns as possible. Also, since the solutions to these microscopic specifications are macroscopic (describing properties of the solution over an extended portion of time/space), specifying in a microscopic way prevents forcing the move from microscopic to macroscopic from being done offline. In this way, discontinuous transitions and continuous transitions are handled in a consistent manner, via mode transitions and pliant transitions respectively, both of which are predominantly expression based ways of specifying updates.

6. Continuing from point **5**, when specifying the unavoidable handovers between continuous and discrete behaviours while using the macroscopic, relation based, way of specifying continuous behaviour, the endpoints of the periods of continuous behaviour need to be described within the relations themselves, so that the domain of the relevant relation can be specified. This is potentially an overhead for the model designer when the problem is complicated enough, since the handovers take place when prompted by physical law. In **HEB**, this job is taken over by a generic preemption mechanism, which is, in turn, much easier to handle in the expression based way of managing pliant behaviour, since all the details regarding the domain of applicability of the pliant behaviour do not need to be specified in advance.

7. Another consequence of point **5** concerns invariants. Invariants are normally expressions written in the state variables, that are expected to be true at all times. Now, when we only have the usual changes of discrete state, and we have the conventional interpretation of Event-B in the physical world in which discrete transitions occur at isolated times, then the state does not change in between these discrete transitions. Thus,

once true at some point of an execution (e.g. at initialisation time), if invariants are reestablished at each discrete transition, then the invariants hold throughout the duration of the execution. Note that this reasoning takes place largely outside of the formal **EB** framework.

When the discrete **EB** transitions are extended to encompass updates to lumps of continuous behaviour, the preceding argument no longer holds. Straightforward safety properties built out of natural problem entities no longer correspond to equivalent expressions built on state variables, but need to be extracted from the relations containing pieces of continuous behaviour, potentially making the proof of safety properties more difficult.

The observation particularly concerns refinement. In relatively benign cases where refinement amounts to 'reduction of nondeterminism', it may be possible relatively straightforwardly to argue that, say, a continuous monotonic function is continuous, and thus, that a chunk of continuous monotonic function refines a continuous specification. But the challenge can get much harder when 'data refinement' is involved. Then, the chunks have to be unpacked and the pointwise expressions compared (in fact reflecting the **HEB** process), before anything can be deduced.

By contrast, the **HEB** approach expresses all instantaneous state update, both mode and pliant, via expressions in the state variables, which usually correspond to the natural variables of the problem. This enables the invariants to be be built in the same straightforward way as in the purely discrete case. Refinement is rendered no harder than the discrete case, though the time parameter has to be carried around through the derivation (which, in the vast majority of cases, imposes no overhead).

While, in principle, any invariant written using the more transparent methods of **HEB** could, with effort, be translated into the more convoluted **EB** kind, as a general point, we should not underestimate the impact on those aspects of the application that are emphasised, made by the detailed formalism in which the models and properties of a given application are written. Thus: (a) properties in model based frameworks tend to be written as invariants on the state space, and behavioural properties remain implicit in the enabledness (or not) of events in the after-states of preceding events; (b) properties in behaviourally based frameworks tend to be written as temporal logic expressions, and say little or nothing about states or whether behaviours other than ones described are permissible; (c) the architectural structure of a system leads to an emphasis on the properties of the individual components, whether state based or behavioural, and properties of the system as a whole that depend on the correct execution of protocols by collections of components are downplayed (other than in approaches focused specifically on protocols), etc. So the difference between the **EB** and **HEB** approaches can lead to subtle bias in the safety properties that are written, and later checked during verification.

8. Although not a feature of the **EB** treatment here, a number of treatments of continuous phenomena using **EB**, describe continuous, time dependent phenomena via lambda expressions such as $\lambda\,\tau \bullet E(\tau)$. Extraction of a value is done via application of such an expression to a parameter. This technique makes even more distant (than in the **EB** technique used here) the connection between problem quantities and actual model variables, since there needs to be even more packing and unpacking of these

lambda expressions to get at the juice inside (than in the present case). From a formal point of view, a binder like λ typically binds its variable: moreover, the bound variable is formally alpha convertible [7, 11], which can change its name arbitrarily. If this is the case, the identification of the variable τ in the given expression with a problem domain quantity like the time, lies completely outside the formal framework — it becomes an application level convention. This contrasts with the practice in conventional descriptions of physical phenomena, of naming physical quantities using free variables, leading to the possibility of being able to correlate the mention of the same quantity at different places by simple lexical identity. Of course this practice is reflected in the design of **HEB**.

However, we have to be a little careful. In many similar formalisms, such as in the refinement calculus [4], alpha conversion is an intrinsic part of the machinery, leading precisely to the phenomenon being discussed. However in the logical language of **EB**, the *not-free-in* property is used instead when introducing binders. This is based on the idea that provided wise, non-clashing choices of bound variables are made at the point of introduction, those choices will never need to be overridden in the reasoning algorithms, precluding the need for formal alpha conversion. In the B-Book [1], the *not-free-in* property is explicitly correlated with the quantified variable in the predicate that specifies the lambda expression (B-Book p. 89, & *ff.*). In the **EB**-Book [2], the lambda variable is a *pattern*, and although the formalities of its role as bound variable are not explicitly discussed, similar properties may be inferred (**EB**-Book p. 331 & *ff.*). Thus, in the context of the *not-free-in* technique, *in theory*, it might be possible to use the free problem variables as lambda variables in sufficiently simple situations where this would cause no untoward clashes, but *in practice* this is not something that could be expected to be applicable with any generality.[5]

A genuine reconciliation of the issues just discussed would run as follows. A richer language of *type names* would be introduced. These names would be free identifiers. Complex (or built-in) types could be given a name, and name equivalence (rather than structural equivalence) would decide type equality and compatibility. That way, a type of time could be distinguished from a type of lengths, even though both are based on $\mathbb{R}$ under the bonnet. Alpha conversion would apply to lambda expressions etc. as usual, but not to the type name expressions that declared their types. We would have reinvented the free name convention of **HEB**, removed one level!

It is notable how most of the issues identified in the above list do not concern the details of the **EB** and **HEB** formalisms themselves, but engage with questions that surround how the formalism connects with the wider requirements and applications environment. This is another illustration of the observation that the more naturally a formal framework relates to the problem domain, the more useful its contribution to overall system dependability is likely to be.

[5] Strictly speaking, *not-free-in* means 'does not occur free —*but may occur bound*— in'. Thus, the possibilities for alpha conversion are latent in the B-Method, even if they are downplayed.

7 Conclusions

In the previous sections we reviewed Event-B and its hybrid extension, and then summarised the water tank development in the two formalisms. This provided the background for a more thorough comparison of the two ways of developing hybrid systems in Section 6. What this showed was that although many issues that were rather natural to express in Hybrid Event-B could be handled, with some effort, in Event-B, doing it that way placed more and more reliance on conventions that lay outside the formal Event-B framework. Obviously, the aim of having a formal framework is to open the possibility of having a system whereby properties directly relevant to the application can be checked mechanically, instead of relying on informal conventions verified by humans for their enforcement. Thus the pure Event-B approach to hybrid system design and development will inevitably struggle increasingly, as the scale of the problem being tackled grows.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. STTT 12, 447–466 (2010)
4. Back, R.J.R., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
5. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core Hybrid Event-B I: Single Hybrid Event-B Machines. Sci. Comp. Prog. 105, 92–123 (2015)
6. Banach, R., Butler, M., Qin, S., Zhu, H.: Core Hybrid Event-B II: Multiple Cooperating Hybrid Event-B Machines (2015), submitted.
7. Barendregt, H.: The Lambda Calculus Its Syntax and Semantics. Elsevier (1981)
8. Butler, M., Abrial, J.R., Banach, R.: Modelling and Refining Hybrid Systems in Event-B and Rodin. In: Petre, Sekerinski (eds.) From Action System to Distributed Systems: The Refinement Approach. Dedicated to Kaisa Sere. pp. 29–42. CRC Press, Taylor and Francis (2015)
9. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and Tools for Hybrid Systems Design. Foundations and Trends in Electronic Design Automation 1, 1–193 (2006)
10. Geisberger, E., Broy (eds.), M.: Living in a Networked World. Integrated Research Agenda Cyber-Physical Systems (agendaCPS) (2015), http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Publikationen/Projektberichte/acaetch_STUDIE_agendaCPS_eng_WEB.pdf
11. Hindley, R., Seldin, J.: Introduction to Combinators and λ-Calculus. Cambridge U.P. (1986)
12. Mathematica: http://www.wolfram.com
13. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
14. RODIN Tool: http://www.event-b.org/ http://sourceforge.net/projects/rodin-b-sharp/
15. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer (2009)