Cruise Control in Hybrid Event-B

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. A case study on automotive cruise control originally done in (conventional, discrete) Event-B is reexamined 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). A significant case study such as this has various benefits. It can confirm that the Hybrid Event-B design allows appropriately fluent application level modelling (as is needed for serious industrial use). It also permits a critical comparison to be made between purely discrete and genuinely hybrid modelling. The latter enables application requirements to be covered in a more natural way. It also enables some inconvenient modelling metaphors to be eliminated.

1 Introduction

With the ever decreasing size and cost of computing devices today, there is a strong incentive to embed digital processors in all sorts of devices and systems, in order to improve design flexibility, performance and production cost. This has two readily discernable consequences. Firstly, since many of these systems interact directly with humans, such designs rapidly acquire a safety-critical dimension that most computing systems in the past did not have. Secondly, the profusion of such systems, their interactions with the environment and with each other, dramatically increases design complexity beyond the bounds where traditional development techniques can reliably deliver the needed level of dependability.

It is by now well accepted that formal techniques, appropriately deployed, can offer significant help with both of these issues. However, in the main, these techniques are strongly focused on purely discrete reasoning, and deal poorly with the continuous behaviours, that of necessity, are forced into the blend by the intimate coupling of computing devices to real world systems. The *hybrid* and *cyberphysical* systems we speak of (see, e.g. [20, 23, 2, 22, 6]) are poorly served by conventional formal techniques. Although they do have approaches of their own (see, e.g. [8]), most of these techniques are either limited in their expressivity, or lack rigour by comparison with most discrete techniques. An exception is KeYmaera (see [1, 16]), a system that combines formal proof (of a quality commensurate with contemporary formal techniques) with continuous behaviour (as needed in the description of genuine physical systems).

The need for similar capabilities in systems to which the discrete Event-B methodology [3] has been applied in recent years, prompted the development of an extension, Hybrid Event-B [5], that treats discrete and continuous behaviours equally. In this paper, we apply this formalism to a case study previously done in discrete Event-B: the modelling of a cruise control system, first investigated as a component of the DEPLOY Project [9]. The motivation for doing this is: firstly, to judge the expressivity and fluency of the Hybrid Event-B formalism regarding the description of scenarios such as this (especially with a view to practical engineering use); and secondly, to readdress some of the methodological deficiencies that the original case study identified as caused by purely discrete modelling. In contrast to KeYmaera, there is at present no dedicated tool support for Hybrid Event-B. In light of this, a further benefit of the present study is to confirm that Hybrid Event-B contains the right collection of ingredients for industrial scale modelling, before more serious investment in extending the RODIN Tool for discrete Event-B [17] is made.

The rest of this paper is as follows. Section 2 overviews the cruise control system. Section 3 discusses the methodological issues raised by the previous discrete case study, and how Hybrid Event-B can address them. Section 4 overviews Hybrid Event-B itself. Sections 5, 6, 7 then take the cruise control system from a pure mode based model, through a model where continuous properties are constrained but not defined explicitly, to a model where both modes and continuous behaviour are fully defined. These models are related to one another using a sequence of refinements. Section 8 concludes.

2 Cruise Control System Overview

A cruise control system (CCS) is a software system which automatically controls the speed of a car. The CCS is part of the engine control software which controls actuators of the engine, based on the values of specific sensors. Since the CCS automatically controls the speed of the car there are some safety aspects to be considered and it needs to fulfil a number of safety properties. For example, the cruise control system must be deactivated upon request of the driver (or in case of a system fault).

The part of the CCS focused on in the DEPLOY Project, which we follow here, was the signal evaluation subsystem. For economy of space we simplify a bit from the full case study tackled in DEPLOY [12], but we take care to retain all the elements where we can demonstrate the methodological improvements discussed in Section 3 and show the advantages of our approach.

We broadly follow the description in [25, 24]. In Fig. 1 we see the state transition diagram for a simplified CCS at an intermediate level of description. The CCS starts in the *OFF* state, from where it can be *Sw*itched*On* to put it into the *ON* state.

In the *ON* state several things can happen. One option for the driver is to *Sw*itch*Off* the CCS. Alternatively, the driver can *Set*the*Speed* for the CCS, which will set the target speed for the car, to be maintained by the engine control system under the guidance of the CCS. While the speed is under the control of the CCS, the speed can be *Tip*ped*Up* by the driver to increase it a little, or *Tip*ped*Down* to decrease it a little. If the driver chooses to *Dep*ress*Br*ake*or*Cl*utch while the CCS is on, then the CCS is designed to switch off since it is assumed that a hazardous condition may have been encountered.

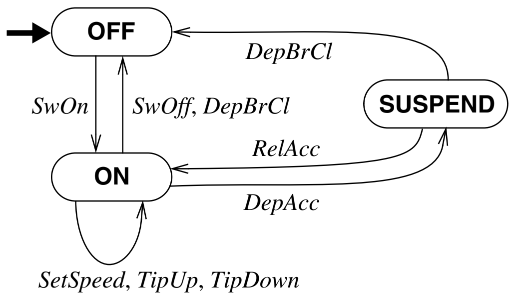

Fig. 1. The state transition diagram for a simplified cruise control system.

However, if the driver chooses to *Dep*ress*Acc*elerator while the CCS is on, then it is assumed that conditions are safe, and the CCS is merely put into the *SUSPEND* state. In this state the driver controls the speed via the accelerator pedal. If in this state the driver subsequently *Rel*eases*Acc*elerator, the previous CCS state is resumed. However, use of the brake or clutch in this state switches the CCS off, in line with the assumptions mentioned earlier.

Below, we will develop a series of Hybrid Event-B machines to capture this design. Before that though, we recap some methodological issues that arose in the context of the earlier, purely discrete development, in order to focus the reader's attention on how these are handled differently in the fully hybrid formalism later.

3 Methodological Considerations

In the original discrete Event-B development of the CCS [12] the formal techniques contemplated were based round existing design practices. These produce, for any proposed application: firstly, a set of functional requirements generated by a requirements engineering process; secondly, a set of safety requirements generated by a hazard anaylsis. In relation to Event-B, the former are transformed into events, and the latter are transformed into invariants of the eventual Event-B model(s).

Typical systems in the automotive industry are embedded real time applications which contain a closed loop controller as an essential part. Closed loop controller development is done by control engineers, while their verification requires reasoning about continuous behaviour. Discrete Event-B does not support continuous behaviour, so the application of discrete Event-B to the CCS case study in [12] had to avoid its direct inclusion. Since the presence of continuous behaviour cannot be avoided in CCS, whenever such behaviour was needed in an Event-B model of [12], the modelling incorporated a function that interfaced between the continuous behaviour and the rest of the model. The function itself though, was not (and because Event-B is discrete, could not be) specified within Event-B.

The extension of discrete Event-B to Hybrid Event-B permits this deficiency to be addressed. Most closed loop controller design takes place within the frequency domain

[15, 10, 11, 4]. This is seemingly a long way away from the state based approach of techniques like Event-B, but the state based formulation of control theory (increasingly popular today, especially when supported by tools such as SIMULINK [13]), enables a direct connection with the conceptual framework of Hybrid Event-B to be made.

In our reworking of the CCS case study, we are able to incorporate the modelling of a closed loop controller as an essential element. This inclusion of the closed loop controller constitutes the first major point of departure from the earlier account.

Another issue concerns the communication of values between subsytems at different levels of a system hierarchy, especially when real time aspects are paramount. Examples include the transmission of values registered by system sensors, handled by an Event-B sensor machine, which need to be communicated to the core machine that consumes them and decides future behaviour. A corresponding situation concerns values determined by the core machine that need to be communicated to an actuator machine.

Events in the machines concerned update relevant variables with the required values. However, the fact that enabledness of events in discrete Event-B merely *permits* them to execute but does not *force* them to do so, means that when such values need to be transmitted in a timely manner, the semantics does not guarantee that this will happen. To address this, flags are introduced to prevent later events from executing before earlier events that they depend on have completed. Such techniques, essentially handshake mechanisms, are discussed in [7, 12, 25, 24].

Handshake mechanisms are eloquent in modelling communication protocols at a high level of abstraction (see e.g. the examples in [3]). However, when the abstract level communication is continuous, such as in the coupling of a continuous controller to its plant, the low level mechanics do rather obscure the essentials of what is going on. In Hybrid Event-B, continuous behaviour is intrinsic, so the instantaneous communication of continuously changing values can be modelled directly. The capacity to directly model the communication of continuously varying values constitutes the second major point of departure from the earlier account.

4 Hybrid Event-B, a Sketch

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, never being assigned, since time cannot be controlled by any human-designed engineering process. Clocks allow a bit more flexibility, since they are assumed to increase their value at the same rate that time does, but may be set during mode events (see below). Variables are of two kinds. There are mode variables (like u, declared as usual) 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 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*

```
MACHINE HyEvBMch
TIME t
CLOCK clk
PLIANT x, y
VARIABLES u
INVARIANTS
  x ∈ ℝ
  y ∈ ℝ
  u ∈ ℕ
EVENTS
  INITIALISATION
    STATUS ordinary
    WHEN
      t = 0
    THEN
      clk := 1
      x := x0
      y := y0
      u := u0
    END
...  ...
```

```
...  ...
  MoEv
    STATUS ordinary
    ANY i?, l, o!
    WHERE grd(x, y, u, i?, l, t, clk)
    THEN
      x, y, u, clk, o! :| BApred(x, y, u,
        i?, l, o!, t, clk, x′, y′, u′, clk′)
    END
  PliEv
    STATUS pliant
    INIT iv(x, y, t, clk)
    WHERE grd(u)
    ANY i?, l, o!
    COMPLY
      BDApred(x, y, u, i?, l, o!, t, clk)
    SOLVE
      𝒟 x = φ(x, y, u, i?, l, o!, t, clk)
      y, o! := E(x, u, i?, l, t, clk)
    END
END
```

Fig. 2. A schematic Hybrid Event-B machine.

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 get to 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 in the usual way. As hinted above, in Hybrid Event-B, there are two kinds of event: mode events and pliant events.

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

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

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

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

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

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

Although pliant variables change continuously (except perhaps at the t_i), continuity alone still allows for a wide range of mathematically pathological behaviours. To eliminate these, we make the following restrictions which apply individually to every subinterval $[t_i \ldots t_{i+1})$:

I Zeno: there is a constant δ_{Zeno}, such that for all i needed, $t_{i+1} - t_i \geq \delta_{\mathsf{Zeno}}$.

II Limits: for every variable x, and for every time $t \in \mathcal{T}$, the left limit $\lim_{\delta \to 0} x(t - \delta)$ written $\overrightarrow{x(t)}$ and right limit $\lim_{\delta \to 0} x(t + \delta)$, written $\overleftarrow{x(t)}$ (with $\delta > 0$) exist, and for every t, $x(t) = \overleftarrow{x(t)}$. [N. B. At the endpoint(s) of $\mathcal{T}$, any missing limit is defined to equal its counterpart.]

III Differentiability: The behaviour of every pliant variable x in the interval $[t_i \ldots t_{i+1})$ is given by the solution of 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.

Regarding the above, the Zeno condition is certainly a sensible restriction to demand of any acceptable system, but in general, its truth or falsehood can depend on the system's full reachability relation, and is thus very frequently undecidable.

The stipulation on limits, with the left limit value at a time t_i being not necessarily the same as the right limit at t_i, makes for an easy interpretation of mode events that happen at t_i. For such mode events, the before-values are interpreted as the left limit values, and the after-values are interpreted as the right limit values.

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 intial 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 a number of semantic issues that we have glossed over in the framework just sketched. We refer to [5] for a more detailed presentation.

We note that this framework is quite close to the modern formulation of hybrid systems. (See eg. [21, 16] for representative formulations, or the large literature in the *Hybrid Systems: Computation and Control* series of international conferences, and the further literature cited therein.)

5 Cruise Control — Top Level Mode Oriented Model

In this section we begin the development of the cruise control system by introducing the top level, mode oriented model of the CCS, *CruiseControl_0*. At this level, we just model the state transition diagram given in Fig. 1, i.e. we just focus on the high level user view modes of operation of the system. Regarding a more realistic engineering development, such a model would probably be agreed on first, before the details of the various submodel behaviours were determined.

Regarding the CCS model itself, we see that we model the structure of Fig. 1 using two mode variables: *mode* and *sm* (submode). The former models whether the CCS

[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. This used in Fig. 3.

```
MACHINE CruiseControl_0
VARIABLES mode, sm
INVARIANTS
  mode ∈ {OFF, ON, SUSP}
  sm ∈ {NIL, SET}
  sm = SET ⇒ mode ∈ {ON, SUSP}
EVENTS
  INITIALISATION
    STATUS ordinary
    BEGIN
      mode := OFF
      sm := NIL
    END
  SwOn
    STATUS ordinary
    ANY in?
    WHERE in? = swOn ∧ mode = OFF
    THEN mode := ON
         sm := NIL
    END
  SwOff
    STATUS ordinary
    ANY in?
    WHERE in? = swOff ∧ mode = ON
    THEN mode := OFF
         sm := NIL
    END
  SetSpeed
    STATUS ordinary
    ANY in?
    WHERE in? = setSpeed ∧
      mode = ON ∧ sm = NIL
    THEN sm := SET
    END
  TipUp
    STATUS ordinary
    ANY in?
    WHERE in? = tipUp ∧
      mode = ON ∧ sm = SET
    THEN skip
    END
...  ...
...  ...
  TipDown
    STATUS ordinary
    ANY in?
    WHERE in? = tipDown ∧
      mode = ON ∧ sm = SET
    THEN skip
    END
  DepAcc
    STATUS ordinary
    ANY in?
    WHERE in? = depAcc ∧
      mode = ON ∧ sm = SET
    THEN mode := SUSP
    END
  RelAcc
    STATUS ordinary
    ANY in?
    WHERE in? = relAcc ∧
      mode = SUSP ∧ sm = SET
    THEN mode := ON
    END
  DepBrCl
    STATUS ordinary
    ANY in?
    WHERE in? = depBrCl ∧
      mode ∈ {ON, SUSP}
    THEN mode := OFF
         sm := NIL
    END
  PliTrue
    STATUS pliant
    COMPLY INVARIANTS
    END
END
```

Fig. 3. Mode level description of cruise control operation.

is *ON*, *OFF* or *SUSP*ended, while the latter models whether the speed has been *SET*, otherwise it is *NIL*. It is not hard to check that Fig. 3 gives a translation of Fig. 1 into the framework of Hybrid Event-B. Aside from typing invariants, we have an invariant that only allows the *sm* to be *SET* when the CCS is active (i.e. either *ON* or *SUSP*).

One aspect of both Fig. 1 and Fig. 3 that we should comment on, is that in the real world, the pressing of any of the pedals, or of the CCS control buttons, is not restricted to when the CCS deems it permissible to do so. The car driver has these user interface elements at his disposal at all times, and can operate them whenever he wishes. Thus, we have clearly designed Fig. 1 and Fig. 3 *aggressively*, assuming events take place only when their guards are true. This implies that there is a *defensive* layer above it, deflecting inappropriately commanded events away from the core CCS functionality.

Aside from a few details, Fig. 3 is almost identical to comparable models written in discrete Event-B as described in [12] or [25, 24]. The main difference between the earlier treatments and ours, is that these other treatments developed their mode level descriptions incrementally, adding a feature or two at a time via refinement, to ease automated verification. By contrast, we have presented our mode level model monolithically, so as to save space (and the reader's attention span) for the richer modelling of the continuous system behaviour between mode changes that is the main contribution of this paper.

Regarding the technical structure of Fig. 3, it differs from a discrete Event-B machine in only a couple of details. First is that each of the mode events (indicated by the 'STATUS ordinary' designation) has an input parameter whose value is (almost) the name of the event. Considering the actions of these various mode events, such parameters would be unnecessary in discrete Event-B. In Hybrid Event-B though, time is an essential feature of the modelling framework, so the timing of occurrences of mode events is an issue. The semantics of Hybrid Event-B stipulates that mode events with input parameters only become enabled when the input values become available from the environment, and it is *assumed* that they only become available at times that do not clash with other mode events. Thus the appearance of the inputs from the environment acts to schedule mode event occurrences in a way compatible with the usual interpretation of their occurrence in discrete Event-B. The only other difference from discrete Event-B that is visible in Fig. 3, is the pliant event *PliTrue*. This has a vacuous guard, and (essentially) vacuous semantics that merely insist that the INVARIANTS are maintained. Its job is simply to formally allow time to pass (according to the semantics of Hybrid Event-B) until the next mode event occurrence takes place, prompted by the appearance of the relevant parameter from the environment.

From the above, it is easy to see that a standard discrete Event-B machine, giving a mode level description of the behaviour of some desired system, could be mechanically and routinely translated to a machine of the form of Fig. 3, allowing an original discrete Event-B machine to be refined ultimately by a Hybrid Event-B machine. Alternatively, the formal semantics of UML-B [18, 14, 19] would enable the same job to be done starting from a more diagrammatic representation. This would enable a development process that started by focusing on just a conventional discrete Event-B mode level behaviour of the system, to be enriched with real time properties further along the development, within an integrated development activity.

6 Cruise Control — Abstract Continuous Behaviour

In this section we enhance the pure mode oriented model of Section 5 with a specification of the desired continuous behaviour in the periods between occurrences of the mode events.

The requirements that are intended to be addressed by this behaviour are relatively easy to formulate at a user level. Thus, once the CCS is in control of the car, we require that the actual speed of the car differs from the target speed that has been set by the driver by at worst a margin that is determined by the CCS design.

The car's actual behaviour may drift away from it's target value for many reasons. The target speed is set when the driver engages the CCS, and is translated into commands for the car to maintain it, but the actual behaviour is affected by many additional environmental factors. These include factors such as road slope, wind resistance, road surface characteristics, total car weight, fuel energy output, and so on. These add considerable uncertainty and complexity to the real world situation.

Control engineers cope with the vast range of environmental uncertainty by using feedback. The deviation between the actual and desired behaviour is monitored, and the difference is used to impel the controlled system towards the desired behaviour.

The low level design of a real CCS deals with the many factors that affect the car's performance, as indicated. In this paper we will restrict our attention to a simple control design addressing the user level requirements stated earlier. This illustrates how a control system design may be integrated with the modelling capabilities of Hybrid Event-B. More realistic designs will follow the same general principles as our example, and will merely exhibit increased complexity.

Our enhanced treatment of the CC system is to be found in Fig. 4, completed in Fig. 5. After the machine and refinement declarations, there is a declaration of a pliant variable v, representing the velocity of the car. In our simple approach to CCS, this single pliant variable will be sufficient.

Next come the (mode) variables. Of these, *mode* and *sm* are familiar from Fig. 3. Of the remainder, *setv* is the target velocity set by the driver of the car, while the new variable *rn* records whether a ramp up/down episode is needed after use of the accelerator prior to resuming cruising velocity. All other identifiers, occurring but not declared in Figs. 4 or 5, are constants of the system, as if they were in a Hybrid Event-B CONTEXT not included in the paper. We return to them at the end of this section.

Next come the invariants. For the real valued variables, discernable as such because of the invariants that restrict them to a real valued closed interval, e.g. $v \in [0 \dots V_{\max}]$, the restriction to the interval is mostly the only property they have to satisfy. Aside from v, these real valued variables are mode variables, so are piecewise constant during pliant transitions, despite being real valued.

The remaining invariant is $\mathsf{CONTINUOUS}(v)$, featuring the '$\mathsf{CONTINUOUS}$' *pliant modality*. Now, the semantics of Hybrid Event-B guarantees that in between mode transitions, the behaviour of all pliant variables must be absolutely continuous. Nevertheless, pliant variables may suffer discontinuities during mode transitions. The $\mathsf{CONTINUOUS}$ modality stipulates that this must not happen to v, and a simple static check on the mode events is enough to guarantee this. The global continuity of v is of course intended to contribute to the 'comfortable behaviour' requirement.

MACHINE $CruiseControl\_1$
REFINES $CruiseControl\_0$
PLIANT v
VARIABLES $mode, sm, setv, rn$
INVARIANTS
$v \in [0 \ldots V_{\max}]$
$\mathsf{CONTINUOUS}(v)$
$mode \in \{OFF, ON, SUSP\}$
$sm \in \{NIL, SET\}$
$sm = SET \Rightarrow mode \in \{ON, SUSP\}$
$setv \in [VCC_{\min} \ldots VCC_{\max}]$
$rn \in \mathrm{BOOL}$
EVENTS
INITIALISATION
STATUS ordinary
REFINES *INITIALISATION*
BEGIN
$v :\in [0 \ldots V_{\max}]$
$setv :\in [VCC_{\min} \ldots VCC_{\max}]$
$mode := OFF$
$sm := NIL$
$rn := \mathrm{FALSE}$
END
PliDefault
STATUS pliant
REFINES *PliTrue*
WHEN $mode \in \{OFF, SUSP\} \lor$
$(mode = ON \land sm = NIL)$
COMPLY *INVARIANTS*
END
SwOn
STATUS ordinary
REFINES *SwOn*
ANY $in?$
WHERE $in? = swOn \land mode = OFF$
THEN $mode := ON$
$sm := NIL$
END
SwOff
STATUS ordinary
REFINES *SwOff*
ANY $in?$
WHERE $in? = swOff \land mode = ON$
THEN $mode := OFF$
$sm := NIL$
END
... ...

... ...
SetSpeed
STATUS ordinary
REFINES *SetSpeed*
ANY $in?$
WHERE $in? = setSpeed \land$
$v \in [VCC_{\min} \ldots VCC_{\max}] \land$
$mode = ON \land sm = NIL$
THEN $sm := SET$
$setv := v$
END
Cruise
STATUS pliant
REFINES *PliTrue*
INIT $|v - setv| \leq \Delta_{\mathrm{Cruise}}$
WHERE $mode = ON \land sm = SET$
COMPLY $|v - setv| \leq \Delta_{\mathrm{Cruise}} \land$
$|\mathcal{D}v| \leq \Delta_{\mathrm{MCA}}$
END
RampUp
STATUS pliant
REFINES *PliTrue*
INIT $v - setv < -\Delta_{\mathrm{Cruise}}$
WHERE $mode = ON \land sm = SET$
COMPLY $|\mathcal{D}v - RUA| \leq \Delta_{\mathrm{RUD}}$
END
RampDown
STATUS pliant
REFINES *PliTrue*
INIT $v - setv > \Delta_{\mathrm{Cruise}}$
WHERE $mode = ON \land sm = SET$
COMPLY $|\mathcal{D}v + RDA| \leq \Delta_{\mathrm{RUD}}$
END
ResumeCruise
STATUS convergent
WHEN $|v - setv| \leq \Delta_{\mathrm{Cruise}} \land$
$mode = ON \land sm = SET \land rn$
THEN $rn := \mathrm{FALSE}$
END

VARIANT rn
... ...

Fig. 4. Cruise control operation with abstract continuous behaviour, first part.

```
...  ...
  TipUp
    STATUS ordinary
    REFINES TipUp
    ANY in?
    WHERE in? = tipUp ∧
      mode = ON ∧ sm = SET
    THEN
      setv := min{setv + TUD, VCC_max}
    END
  TipDown
    STATUS ordinary
    REFINES TipDown
    ANY in?
    WHERE in? = tipDown ∧
      mode = ON ∧ sm = SET
      setv − TUD ≥ VCC_min
    THEN
      setv := max{setv − TUD, VCC_min}
    END
  DepAcc
    STATUS ordinary
    REFINES DepAcc
    ANY in?
    WHERE in? = depAcc ∧
      mode = ON ∧ sm = SET
    THEN mode := SUSP
    END
...  ...
```

```
...  ...
  RelAccCruise
    STATUS ordinary
    REFINES RelAcc
    ANY in?
    WHERE in? = relAcc ∧
      mode = SUSP ∧ sm = SET ∧
      |v − setv| ≤ Δ_Cruise
    THEN mode := ON
         rn := FALSE
    END
  RelAccRamp
    STATUS ordinary
    REFINES RelAcc
    ANY in?
    WHERE in? = relAcc ∧
      mode = SUSP ∧ sm = SET ∧
      |v − setv| > Δ_Cruise
    THEN mode := ON
         rn := TRUE
    END
  DepBrCl
    STATUS ordinary
    REFINES DepBrCl
    ANY in?
    WHERE in? = depBrCl ∧
      mode ∈ {ON, SUSP}
    THEN mode := OFF
         sm := NIL
    END
END
```

Fig. 5. Cruise control operation with abstract continuous behaviour, second part.

The heart of the model consists of the events, the first of which is *INITIALISATION*. This intialises *mode* and *sm* as before, and sets all the real valued variables to arbitrary values in their permitted range. We examine the remaining events one by one.

PliDefault is a pliant event that refines *PliTrue* of Fig. 3. It allows the variables to vary arbitrarily via the 'COMPLY *INVARIANTS*' clause, although the invariants must be maintained. Note that the guard of *PliDefault* is stronger than that of *PliTrue* — the unconstrained behaviour is only permitted under conditions where the CCS would *not* be expected to be in control.

The events *SwOn* and *SwOff* are identical to their Fig. 3 precursors.

Event *SetSpeed* acquires new functionality, in that it now also sets the value of the demanded speed *setv* to be the car's current speed v.

The continuous control itself is handled by the next three pliant events, *Cruise*, *RampUp*, *RampDown*. We start with *Cruise*. On entry to *Cruise*, if the car's actual speed v is within a suitable margin (given by the constant $\mathit{\Delta}_{\mathrm{Cruise}}$) of the desired speed *setv*, then the event is enabled, as defined by the INIT clause $|\, v - \mathit{setv} \,| \leq \mathit{\Delta}_{\mathrm{Cruise}}$. In this case, at the present level of abstraction, the behaviour is not precisely defined, but the *Cruise* event demands that the speed remains within a suitable margin of *setv*, bounded by $\mathit{\Delta}_{\mathrm{Cruise}}$ again. A further requirement is once more related to 'comfort', in that the rate of change of v should not exceed a *M*aximum*C*ruise*A*cceleration, $\mathit{\Delta}_{\mathrm{MCA}}$. These stipulations are housed in the COMPLY $|\, v - \mathit{setv} \,| \leq \mathit{\Delta}_{\mathrm{Cruise}} \land |\, \mathcal{D}v \,| \leq \mathit{\Delta}_{\mathrm{MCA}}$ clause. Note that this represents a genuine *specification*, in that the COMPLY clause gives no indication of how a behaviour with the required properties is to be achieved. It also represents behaviour that trivially refines *PliTrue*, in that the latter accepts all behaviours obeying the invariants.

Similar considerations apply to *RampUp* and *RampDown*. Taking *RampUp*, it caters for the cases when, following use of the accelerator to casue some temporary variation in the car's speed, the car's actual speed v is less than the desired speed *setv* by an amount greater than $\mathit{\Delta}_{\mathrm{Cruise}}$.[2] In such a case, it is deemed that a(n approximately) constant acceleration towards the desired speed *setv* is an appropriate handling of the 'comfort' requirement. So we have a clause COMPLY $|\, \mathcal{D}v - \mathit{RUA} \,| \leq \mathit{\Delta}_{\mathrm{RUD}}$. This demands that the acceleration $\mathcal{D}v$ does not differ from the constant *RUA*, i.e. *R*amp*U*p*A*cceleration, by more than the deviation $\mathit{\Delta}_{\mathrm{RUD}}$. Again this is specification, pure and simple. No indication is given about how to achieve the behaviour described.

Event *RampDown* is very similar to *RampUp*. It fires when, following use of the accelerator, the car's actual speed is greater than *setv* by an amount exceeding the constant $\mathit{\Delta}_{\mathrm{Cruise}}$. Now the car is required to decelerate at the (approximately) constant acceleration $-\mathit{RDA}$ (with the same margin as before). Again, the COMPLY clause amounts to pure specification. No indication is given about how to achieve the behaviour described.

A number of additional remarks are in order regarding *Cruise*, *RampUp*, *RampDown*. Firstly, the constants occurring in the events' INIT guards must be chosen so that the disjunction of the INIT guards can cover all permissible car speeds in the car's permitted range $[0 \ldots V_{\max}]$. Otherwise, when $\mathit{mode} = \mathit{ON} \land \mathit{sm} = \mathit{SET}$, the relative deadlock freedom property of refinement will fail since all three events refine the unconstrained behaviour of *PliTrue*. It is clear that *Cruise*, *RampUp*, *RampDown*, as defined, meet this constraint.

Secondly, if say *RampUp* runs, then if left to continue in an unhindered manner, it will eventually cause the $v \in [0 \ldots V_{\max}]$ invariant to fail, since a constant positive acceleration will eventually cause *any* upper speed limit to be exceeded. To prevent this, we have introduced a new mode event *ResumeCruise*, which runs when the car's velocity, previously differing from the set speed by more than $\mathit{\Delta}_{\mathrm{Cruise}}$, eventually gets within $\mathit{\Delta}_{\mathrm{Cruise}}$ of the set speed. The main job of this mode event is to cause a reschedule, so that *RampUp* is preempted, and *Cruise* is able to run.

We only want *ResumeCruise* to only run once per resumption-of-cruise-control. In order that *ResumeCruise* disables itself upon completion, we use the new *rn* variable in its guard, and falsify it in the action of *ResumeCruise*. This causes *ResumeCruise* to

[2] The initially puzzling possibility that the car might need to *speed up* following use of the *accelerator* is explained by considering driving up a steep hill.

decrease the VARIANT *rn* (with which we interrupt the presentation of events, in order to show it at the most opportune place).

The remainder of the *CruiseControl_1* machine is in Fig. 5. Now, since *setv* is a new feature of the *CruiseControl_1* machine, and since *TipUp* and *TipDown* are intended to manipulate it, these events must be refined nontrivially in order to achieve this. The refinements therefore add or subtract the constant *TUD* from *setv*, although they must do it in a way that prevents the range of permissible cruise control speeds $[VCC_{\min} \ldots VCC_{\max}]$ from being overstepped.

Among the remaining events of *CruiseControl_1* (all mode events), *DepAcc* is as previously. Event *RelAcc* has been split in two though, depending on whether the car's speed is within the margin $\varDelta_{\text{Cruise}}$ when the accelerator pedal is released. If $|\, v - setv \,| \leq \varDelta_{\text{Cruise}}$ holds, i.e. the car is near enough its cruise speed, then *Cruise* can be entered directly, *ResumeCruise* will not be needed, and so *RelAccCruise* sets *nr* to FALSE. If $|\, v - setv \,| \leq \varDelta_{\text{Cruise}}$ is false though, then a ramp up or down episode is needed, so *RelAccRamp* sets *nr* to TRUE so that *ResumeCruise* will eventually be enabled.

Finally, *DepBrCl* is as in *CruiseControl_0*.

Having covered the whole system model, we are in a position to reconsider the constants, as promised earlier. While it is natural in high level modelling to introduce, at will, constants that constrain system behaviour in desirable ways, these constants will not normally be independent, and will need to satisfy a number of properties to ensure soundness. The safest way to ensure that all needed constraints have been considered, is to attempt mechanical verification — a mechanical prover will remorselessly uncover any missing constraints, which will show up by generating unprovable subgoals.

Despite lack of dedicated tool support for Hybrid Event-B at present, the simplicity of our model here, means that a large portion of this work can be done using discrete Event-B and the existing RODIN tool. The fact that, aside from properties involving continuity and differentiability, we only have uninstantiated constants, and only use properties of reals that are also true of the integers, means that unprovability in the integers is a strong indication of falsity in the reals. Thus, regarding the details of our models, we would obviously need $0 < VCC_{\min} < VCC_{\max} < V_{\max}$. Beyond that, the mode events can be treated directly, as noted earlier. This leaves the pliant events, *Cruise*, *RampUp*, *RampDown*.

For our purposes, we can treat *Cruise* as a mode event that skips, for the following collection of reasons: it maintains its guard; $0 < \varDelta_{Cruise}$ is a constant that is just used to partition the set of velocities; $\mathcal{D}v$ is a variable independent of v at any given time (and which is never tested in any guard); and $0 < \varDelta_{\text{MCA}}$ is a constant that occurs nowhere else. For *RampUp* and *RampDown*, aside from the obvious $0 < min\{RUA, RDA, \varDelta_{\text{RUD}}\}$, all of *RUA*, *RDA*, $\varDelta_{\text{RUD}}$ are again constants that occur nowhere else, that only concern $\mathcal{D}v$, and thus are not further constrained. Beyond that, the behaviour of *RampUp*, *RampDown* is intended to achieve $|\, v - setv \,| \leq \varDelta_{\text{Cruise}}$, so for our purposes, we can replace them by mode events with action $v :|\ |\, v' - setv \,| \leq \varDelta_{\text{Cruise}}$. In this manner, with the help of some admittedly informal reasoning regarding continuity and differentiability, we can go quite a long way towards replicating the reachability relation of the *CruiseControl_1* machine (expressed in terms of sequences of event names that are executed and the before-/after-values of the events' variables), using a discrete Event-B machine with the same constants obeying the same constraints. (In fact, the authors used this approach on an earlier version of the models, and uncovered a typo concerning inconsistent assumptions about the sign of *RDA*. *RDA* can be a negative constant, or alternatively, a positive constant that is negated when necessary at the point of use; but you must be consistent.)

7 Cruise Control — Continuous Behaviour Defined

In the previous section we specified the continuous behaviour of the CCS in terms of some safety properties captured in the invariants and COMPLY clauses. A real CCS though, would have to realise these properties in a specific design. In this section, we enhance *CruiseControl_1* with such a design.

Fig. 6 contains the enhancement, machine *CruiseControl_2*. This is a refinement of *CruiseControl_1* in which the vast majority of *CruiseControl_1* remains unchanged. The variable declarations show that we only introduce more refined behaviour in this machine, and even then, only in events *Cruise*, *RampUp*, *RampDown*.

We start with *Cruise*. Assuming INIT is satisfied, on entry to *Cruise*, the actual speed v may differ from *setv* by some margin since *Cruise* may have been preceded by *RampUp* or *RampDown*. And while *CruiseControl_1* tolerated a bounded deviation between these indefinitely, in *CruiseControl_2* we replace this by a more specific control law. Since *setv* is the desired speed, we drive the actual speed towards *setv* using negative feedback. The earlier *CruiseControl_1* behaviour is refined to a control law described in the SOLVE clause of the *CruiseControl_2* event. The control law sets the acceleration $\mathcal{D}v$ to be proportional to minus the excess of v over *setv*. Thus, if $v - setv$ is positive, the acceleration is negative, tending to diminish v towards *setv*, and if $v - setv$ is negative, the acceleration is positive, tending to increase v towards *setv*.

The preceding constitutes an extremely simple example of closed loop negative feedback linear control, expressed in the state space picture. The control law in the SOLVE clause, $\mathcal{D}v = -C(v - setv)$, is a simple linear ODE, and can be solved exactly, yielding $v(t) = setv + (v(\mathbb{t}_{\mathrm{L}}) - setv)\, e^{-C(t-\mathbb{t}_{\mathrm{L}})}$, where $\mathbb{t}_{\mathrm{L}}$ is the symbol used in Hybrid Event-B to refer generically to the start time of any time interval during which a pliant event runs. It is trivial to verify that with a suitable C, this refines the behaviour permitted by the *CruiseControl_1* model, since the maximum values of both $|\, v(t) - setv\,|$ and of $|\,\mathcal{D}v\,|$ occur precisely at $t = \mathbb{t}_{\mathrm{L}}$, and henceforth reduce.

In more realistic control scenarios, the overall objectives, namely to design a dynamics that behaves in an acceptable way in the face of the requirements, remains the same, but the technical details get considerably more complicated. To a large extent, frequency-based techniques using Laplace and Fourier transforms cast the reasoning into the algebraic domain, and the picture is further complicated by the use of varying criteria (often based on the properties of these frequency-based techniques) to evaluate design quality. Often, use of these techniques does not blend well with the reasoning found in state machine based formalisms like Event-B and its relatives. For this reason, resticting to state space control design techniques is recommended to achieve the optimal integration between approaches.

```
MACHINE CruiseControl_2
REFINES CruiseControl_1
PLIANT v
VARIABLES mode, sm, setv, rn
INVARIANTS
    ... ...
EVENTS
  INITIALISATION ... ...
  PliDefault ... ...
  SwOn ... ...
  SwOff ... ...
  SetSpeed ... ...
  ResumeCruise ... ...
  TipUp ... ...
  TipDown ... ...
  DepAcc ... ...
  RelAccCruise ... ...
  RelAccRamp ... ...
  DepBrCl ... ...
... ...
```

```
... ...
    Cruise
      STATUS pliant
      REFINES Cruise
      INIT |v − setv| ≤ Δ_Cruise
      WHERE mode = ON ∧ sm = SET
      SOLVE Dv = −C(v − setv)
      END
    RampUp
      STATUS pliant
      REFINES RampUp
      INIT v − setv < −Δ_Cruise
      WHERE mode = ON ∧ sm = SET
      SOLVE Dv = RUA
      END
    RampDown
      STATUS pliant
      REFINES RampDown
      INIT v − setv > Δ_Cruise
      WHERE mode = ON ∧ sm = SET
      SOLVE Dv = −RDA
      END
  END
```

Fig. 6. Cruise control operation with continuous control.

We turn to *RampUp* and *RampDown*. Here, the approximately linear and nondeterministic variation in speed of machine *CruiseControl_1* is replaced by a precise, deterministic linear law for the velocity, specified by a constant acceleration in the SOLVE clauses: $\mathcal{D}v = RUA$ for *RampUp* and $\mathcal{D}v = -RDA$ for *RampDown*.

In writing these deterministic dynamical laws, it is presumed that acceleration is something that can be commanded accurately by the engine management system, based on the properties of the engine, the fuel, the environmental conditions, etc., as discussed in Section 6. In truth, this is something of an exageration. In reality, there is too much uncertainty in all these environmental elements to enable the acceleration to be predicted (and therefore commanded) with complete precision. Aside form anything else, the car's sensors are severely limited regarding the type of information about the environment that they can obtain. So there will be some deviation between the acceleration that the engine management system predicts, and that which is actually achieved. On this basis we would expected to see some difference between our treatments of *Cruise*, and of *RampUp* and *RampDown*.

In the case of *Cruise*, a misjudgement of the precise acceleration that will be achieved is compensated for by the presence of negative feedback. If the car's velocity does not reduce quite as rapidly as anticipated by the engine management system, then the negative feedback will work that much harder to bring the velocity into line. The precise details of the control law can be adjusted to make allowance for such potential imprecision, without disturbing the overall structure of the behaviour. In this sense, the negative feedback makes the *Cruise* design robust against a margin of imprecision.

In the case of *RampUp* and *RampDown*, there is no feedback included in the control law. For these two events, it is the *acceleration* that might be awry, and that would need to be brought into line. There are a number of reasons why we did not include this in our models. Firstly, it would need the introduction of at least one other pliant variable into the models (to distinguish measured acceleration from commanded acceleration). Secondly, the resulting feedback law would make the control system higher order, adding unnecessary complexity. Thirdly we would lose the opportunity to illustrate the contrast between closed loop control (as in *Cruise*) and open loop control (as here, for *RampUp* and *RampDown*) in the context of Hybrid Event-B. Fourthly, if our earlier design is appropriate, then any deviation from cruise speed caused by use of the accelerator pedal will be temporary, and thus *RampUp* and *RampDown* describe *transients* of the system. The small imprecisions that may affect their behaviour will not significantly affect the quality of the CCS at the relatively simple level that we model it in this paper.

8 Conclusions

In the preceding sections, we overviewed the cruise control model examined within the DEPLOY project, and we commented on the deficiencies when formal modelling and verification are based purely on discrete Event-B, as was employed in DEPLOY. We then commented on the anticipated improvements expected when the more expressive Hybrid Event-B formalism is used instead. We continued by outlining the essential elements of Hybrid Event-B, sufficient to cater for the modelling to be done later.

We then developed a simple version of the CCS in Hybrid Event-B, through a number of relatively large scale refinements, using these refinements to illustrate the major modelling steps. Thus, we started with a pure mode oriented model, very similar to what DEPLOY achieved for CCS. The hybrid aspects of Hybrid Event-B were almost completely disregarded here by allowing the continuous behaviour to be arbitrary.

The first refinement then introduced additional structure and restrictions on the continuous behaviour. These, though nondeterministic, were deemed sufficient to express the system requirements. The next refinement then introduced specific control laws that modelled in a simple way how a real CCS might implement the continuous control.

Of course, a real system would be much more complicated than what we presented, but it would consist of a larger collection of ingredients of a similar nature to those in our design. For expository purposes then, we can claim that our presentation met the goals described in Section 3. Specifically, we showed that we could incorporate provision for closed loop controller designs unproblematically (including a brief discussion of open loop control too). Additionally, the smoothness with which our development proceeded, bore eloquent testimony to the fluency of the Hybrid Event-B formalism in tackling developments of this kind. This gives strong encouragement for the development of mechanical support for the Hybrid Event-B framework in the future.

Acknowledgement: Michael Butler is partly funded by the FP7 ADVANCE Project (http://www.advance-ict.eu).

References

1. KeYmaera, http://symbolaris.com
2. Report: Cyber-Physical Systems (2008), http://iccps2012.cse.wustl.edu/_doc/CPS_Summit_Report.pdf
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
4. Antsaklis, P., Michel, A.: Linear Systems. Birkhauser (2006)
5. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core Hybrid Event-B: Adding Continuous Behaviour to Event-B (2012), submitted.
6. Barolli, L., Takizawa, M., Hussain, F.: Special Issue on Emerging Trends in Cyber-Physical Systems. J. Amb. Intel. Hum. Comp. 2, 249–250 (2011)
7. Butler, M.: Towards a Cookbook for Modelling and Refinement of Control Problems (2009), http://deploy-eprints.ecs.soton.ac.uk/108/1/cookbook.pdf
8. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and Tools for Hybrid Systems Design. Foundations and Trends in Electronic Design Automation 1, 1–193 (2006)
9. DEPLOY: European Project DEPLOY IST-511599 http://www.deploy-project.eu/
10. Dorf, R., Bishop, R.: Modern Control Systems. Pearson (2010)
11. Dutton, K., Thompson, S., Barraclough, B.: The Art of Control Engineering. Addison Wesley (1997)
12. Loesch, F., Gmehlich, R., Grau, K., Mazzara, M., Jones, C.: Project DEPLOY, Deliverable D19: Pilot Deployment in the Automotive Sector (2010), http://www.deploy-project.eu/pdf/D19-pilot-deployment-in-the-automotive-sector.pdf
13. MATLAB and SIMULINK: http://www.mathworks.com
14. Mermet, J.: UML-B: Specification for Proven Embedded Systems Design. Springer (2004)
15. Ogata, K.: Modern Control Engineering. Pearson (2008)
16. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
17. RODIN: European Project RODIN (Rigorous Open Development for Complex Systems) IST-511599 http://rodin.cs.ncl.ac.uk/
18. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. TOSEM 15, 92–122 (2006)
19. Snook, C., Oliver, I., Butler, M.: The UML-B Profile for Formal Systems Modelling in UML. UML-B Specification for Proven Embedded Systems Design (2004)
20. Sztipanovits, J.: Model Integration and Cyber Physical Systems: A Semantics Perspective. In: Butler, Schulte (eds.) Proc. FM-11. Springer, LNCS 6664, p.1, http://sites.lero.ie/download.aspx?f=Sztipanovits-Keynote.pdf (2011), Invited talk, FM 2011, Limerick, Ireland
21. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer (2009)
22. White, J., Clarke, S., Groba, C., Dougherty, B., Thompson, C., Schmidt, D.: R&D Challenges and Solutions for Mobile Cyber-Physical Applications and Supporting Internet Services. J. Internet Serv. Appl. 1, 45–56 (2010)
23. Willems, J.: Open Dynamical Systems: Their Aims and their Origins. Ruberti Lecture, Rome (2007), http://homes.esat.kuleuven.be/~jwillems/Lectures/2007/Rubertilecture.pdf
24. Yeganefard, S., Butler, M.: Control Systems: Phenomena and Structuring Functional Requirement Documents. In: Proc. ICECCS-12. pp. 39–48. IEEE (2012)
25. Yeganefard, S., Butler, M., Rezazadeh, A.: Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B. In: Proc. 2nd NFM, NASA/CP-2010-216215. pp. 182–191. NASA (2010)