Shared-Variable Concurrency, Continuous Behaviour and Healthiness for Critical Cyberphysical Systems

Richard Banach[1]* and Huibiao Zhu[2]**

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
Email: banach@cs.man.ac.uk
[2]Shanghai Key Laboratory of Trustworthy Computing
MOE International Joint Laboratory of Trustworthy Software
International Research Center of Trustworthy Software
East China Normal University, Shanghai 200062, China
Email: hbzhu@sei.ecnu.edu.cn

Abstract. In the effort to develop critical cyberphysical systems, existing computing formalisms are extended to include continuous behaviour. This may happen in a way that neglects elements necessary for correct continuous properties and correct physical properties. A simple language is taken to illustrate this. Issues and risks latent in this kind of approach are identified and discussed under the umbrella of 'healthiness conditions'. Modifications to the language in the light of the conditions discussed are described. An example air conditioning system is used to illustrate the concepts presented, and is developed both in the original language and in the modified version.

1 Introduction

With the massive proliferation in computing systems that interact with the real world, spurred by the tumbling costs of processors, memory and sensor/actuator equipment, the need for reliable methods to construct such systems has never been greater, especially since so many of these systems have high consequence aspects if they fail to behave as intended. In the light of this drive, systematic methodologies from the discrete formalisms world are being adapted to incorporate the needs of the physical behaviours that are now intrinsic to these systems. While this is entirely appropriate as a broad objective, in reality, many such initiatives may turn out skewed in the execution, in that a great emphasis is placed on the discrete aspects of such an extended formalism, to the neglect of needs coming from the continuous aspects, especially regarding the more subtle of these pertaining to continuous behaviour, and to credible physical properties. The interplay between these worlds can also fail to get the attention it requires. The balance of emphasis perceptible in typical texts in this area such as [1, 2] gives a good indication of this situation.

* The work reported here was done while Richard Banach was a visiting researcher at E.C.N.U. The support of E.C.N.U. is gratefully acknowledged.

** Huibiao Zhu is supported by National Natural Science Foundation of China (Grant No.61361136002) and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (ZF1213).

In this paper we intend to address this perceived imbalance by examining an example language for concurrent discrete update and critically analysing the consequences that follow when continuous update facilities are added in a relatively naïve way. We describe this critical analysis as bringing some 'healthiness considerations' into play, by analogy with the terminology used in UTP [3]. Having brought these out, we show how to modify our original language to better take them into account within the syntax (where possible). We discuss how remaining points need to be addressed semantically. It is worth saying that our language is one that we would not necessarily use seriously for such applications, but actually, its very lack of obvious suitability serves to better highlight the points we make.

We illustrate the above by developing a simple case study concerning the steady state operation of an air conditioning system, this being a system where there is enough *a priori* physical behaviour to exemplify some of what we discuss abstractly. We give a development in the original language, and a revised version in the revised language.

The rest of the paper is as follows. In Section 2 we present our initial language, and our initial attempt at adding continuous behaviour, specified using differential equations (DEs). Discussing the semantics of this, even relatively informally, leads to a substantial detour regarding the possibilities available when DEs are involved. In Section 3 we give our initial AC system development. In Section 4 we turn to the healthiness considerations, enlarging the earlier semantic discussion to include further issues. Section 5 then modifies the initial language syntactically, where possible. Section 6 redevelops the AC system. Section 7 considers some related approaches. Section 8 concludes.

2 An Initial Concurrent Language

Here is the syntax of our initial language. It is a fairly conventional concurrent shared variable language, allowing delays of a specified number of time units.

Declarations:

$$Decl \ ::= \ [\, x : T \, [= x_0\,] \ ; \,]^*$$

Discrete behaviours:

$$Db \ ::= \ x := e \mid \{xs := es\} \mid @b \mid \#r$$

Constructs:

$$P0 \ ::= \ Db$$

Programs:

$$\begin{aligned} Pr0 \ ::= \ & P0 \mid Name \mid [\, Name \ =]\, Decl \ ; \ Pr0 \mid Pr0 \ ; \ Pr0 \\ & \mid \mathbf{if}\ b\ \mathbf{then}\ Pr0\ \mathbf{else}\ Pr0\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ Pr0\ \mathbf{od} \mid Pr0 \parallel Pr0 \end{aligned}$$

As well as this syntax, we use parentheses in the usual way. In connection with this definition we note the following:

(1) All variables used have to be declared with their types in a declaration block $Decl$ in whose scope (defined as usual) their uses occur.

(2) The discrete variable assignment, $x := e$, is *atomic*, so that no action can interleave the reading the variables of e and writing the result to x. The vacuous assignment is written $\mathbf{skip}$. Each variable has to be assigned an initial value (in terms of constants

and already assigned variables) before it can be used. Initialisation is optionally taken care of during declaration.

(3) The simultaneous assignment $\{xs := es\}$ merely defines a *package* of several atomic updates, which are effected at the same instant.

(4) The discrete event-guard, $@b$, is enabled when the guard b holds; otherwise it is disabled and waits; b is a Boolean condition. $\#r$ represents a delay of r time units.

(5) Program constructs are familiar. $\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q\ \mathbf{fi}$ is the conditional, and $\mathbf{while}\ b\ \mathbf{do}\ P$ is iteration. $P\ ;\ Q$ is sequential composition. Shared-variable concurrency is expressed via $P \parallel Q$, where P and Q can contain the behaviours outlined.

Semantically, if we momentarily disregard the delay $\#r$, everything is quite conventional and we do not need to repeat the details. A language like $Pr0$ expresses updates to variables, which are related to each other via the usual syntactically derived causality relation, but there is no indication about how these updates might relate to the real world. In practice, (real world counterparts of) the atomic updates are usually understood to occur at isolated moments of real time, but there is no absolute necessity for this, e.g. if we interpret according to the conventions of the duration calculus [4].[1]

When we now reconsider the delay $\#r$, things change. We are obliged to take note of real world time. Consequently we take the view that all (packages of) update execution instances have their own specific isolated points in time at which they execute.

The preceding sets the scene for introducing continuous variable update.

Continuous behaviours:

$$Cb\ ::=\ @g \mid [iv]\,\mathcal{D}\,\boldsymbol{x} = F(\boldsymbol{x}, \boldsymbol{y}, \tau)\ \mathbf{until}\ g$$

Constructs:

$$P1\ ::=\ Db \mid Cb$$

Programs:

$$Pr1\ ::=\ P1 \mid Name \mid [\,Name\ =]\ Decl\ ;\ Pr1 \mid Pr1\ ;\ Pr1 \mid\ \ldots\ \text{etc.}$$

Regarding the above we make the following further comments:

(6) Declarations may now include continuous variables as well as discrete variables.

(7) The command $@g$ waits for its guard g to be satisfied. It is like $@b$ except that g may now contain continuous variables.

(8) The differential equation (DE) command $[iv]\,\mathcal{D}\,\boldsymbol{x} = F(\boldsymbol{x}, \boldsymbol{y}, \tau)\ \mathbf{until}\ g$ first guards the entry point of executing the DE until the initial conditions on the variables of the DE system (expressed in $[iv]$) are satisfied (execution is delayed if they are not). Once $[iv]$ is satisfied, the current values of the variables being updated define the DE's initial values, and the behaviour specified by the DE continues ($\mathcal{D}$ denotes the time derivative), until the preempting guard g is satisfied or the DE itself becomes infeasible. The preempting guard g is a Boolean condition, like $@g$.

Semantically, the leeway we had in interpreting pure discrete events, evaporates when we add differential equations. At least it does so if we want a credible correspondence with the real world. While pure discrete event formalisms may, quite sensibly, be studied axiomatically, this is never the case for DEs.

[1] In this paper we wish to sidestep the race conditions that arise when two (packages of) updates which read each others' left hand side variables execute at exactly the same moment.

In conventional pure and applied mathematics, the ingredients of differential equations are always first interpreted with respect to a semantic domain that is stipulated in advance (albeit often implicitly in the case of applied mathematics). Different choices of such semantic domains are justified on grounds of the differing generality that they permit in the properties of the functions that are deemed to solve those differential equations, see e.g. [5]. Accordingly, to embed behaviours defined by differential equations into our language in a sound way, we must first pay some attention to matters of operational semantics for the whole language. We base our treatment here on fairly standard interpretations of state based discrete constructs and of DE systems.

Working bottom-up, the fundamental concept is the state σ, a mapping from each variable v to a value in its type: $v \mapsto \sigma(v)$. We also need clocks, written generically as τ. A clock is a continuous real variable whose time derivative is fixed at 1. The phrase 'a clock is started' means that a fresh clock, initialised to 0, starts to run from the beginning of the semantic interpretation of some non-atomic construct of interest.

The Db part of the language is unsurprising. The discrete atomic variable assignment, $x := e$, sends the state σ to $\sigma[\sigma(e)/x]$, which is identical to σ, except at x, which becomes $\sigma(e)$. Similarly for packaged atomic updates.

For $@b$, if b is true in the current state, then the program completes successfully. Otherwise a clock is started, and runs as long as it takes for the environment to make b true, at which point the program completes.

For $\#r$, if $r \leq 0$, then the program completes successfully. Otherwise a clock is started, and runs for $r > 0$ time units, at which point the program completes.

For the continuous behaviours, for $@g$, since g may contain continuous variables, the true-set of g must be **closed**. With this proviso, if g is true in the current state, then the program completes successfully. Otherwise a clock is started, and runs as long as it takes for the environment to make g true, at which point the program completes.

For the DE forms, we first mention some generalities.

If we write a general first order differential equation as $\Phi(\boldsymbol{v}, \mathcal{D}\boldsymbol{v}, t) = 0$, where $\boldsymbol{v}$ is some tuple of real variables, $\mathcal{D}\boldsymbol{v}$ is a corresponding tuple of real variables intended to denote the derivatives of $\boldsymbol{v}$, and Φ is an arbitrary real-valued function, then nothing can be said about whether any sensible interpretation of such an equation exists. See e.g. [5], or any other rigorous text on DEs, for a wealth of counterexamples that bear this out. Accordingly, rigorous results on differential equations that cover a reasonably wide spectrum of cases, are confined to DE forms that fit a restricted syntactic shape and satisfy specific semantic properties. The best known such class covers first order families that can be written in the form:

$$\mathcal{D}\,\boldsymbol{x} = \boldsymbol{F}(\boldsymbol{x}, \tau) \qquad \text{or} \qquad \mathcal{D}\,\boldsymbol{x} = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{y}, \tau)$$

Here, the left hand form refers to a closed system of variables $\boldsymbol{x}$, whereas the right hand form also permits the presence of additional external controls $\boldsymbol{y}$. As well this syntactic shape, conditions have to be demanded on the vector of funtions $\boldsymbol{F}$ and on the entry conditions of the behaviour to be defined by these definitions.

For simplicity, we assume that the vector of functions $\boldsymbol{F}$ is defined on a closed rectangular region, where for each $\boldsymbol{x}$ component index i we have a Cartesian component $x_i \in [x_{iL} \dots x_{iU}]$, and for each $\boldsymbol{y}$ component index j we have a Cartesian component

$y_j \in [y_{jL} \dots y_{jU}]$, and where the time dependence of $\boldsymbol{F}$ has been normalised to a clock $\tau \in [0 \dots \tau_f]$, with τ_f maximal, which starts when the DE system starts.

For each x_i component, x_{iL} is either $-\infty$ or a finite real number, and x_{iU} is either $+\infty$ or a finite real number, and if both are finite, then $x_{iL} < x_{iU}$. Similarly for the y_j components. We denote this region by $XY \times T$, where XY refers to all the $\boldsymbol{x}, \boldsymbol{y}$ components, and T refers to clock time. We write X for just the $\boldsymbol{x}$ components and Y for just the $\boldsymbol{y}$ components, so that $XY = X \times Y$.

To guarantee existence of a solution the vector $\boldsymbol{F}$ must satisfy a Lipschitz condition:

$$\exists K \bullet K \in \mathbb{R} \wedge \forall \boldsymbol{x}_1, \boldsymbol{y}_1, \boldsymbol{x}_2, \boldsymbol{y}_2, \tau \bullet (\boldsymbol{x}_1, \boldsymbol{y}_1) \in XY \wedge (\boldsymbol{x}_2, \boldsymbol{y}_2) \in XY \wedge \tau \in T \Rightarrow \\ ||\boldsymbol{F}(\boldsymbol{x}_1, \boldsymbol{y}_1, \tau) - \boldsymbol{F}(\boldsymbol{x}_2, \boldsymbol{y}_2, \tau)||_\infty \leq K||(\boldsymbol{x}_1, \boldsymbol{y}_1) - (\boldsymbol{x}_2, \boldsymbol{y}_2)||_\infty$$

Here, we have used the supremum norm $|| \cdot ||_\infty$ since it composes best under logical operations. For finite dimensional systems, any norm is just as good; see [6, 7]. Additionally, we require that $\boldsymbol{F}$ is continuous in time for all $\boldsymbol{y}(\tau) \in Y$.

With the above in place, if $\boldsymbol{x}_0$ is an initial value for $\boldsymbol{x}$ such that $\boldsymbol{x}_0 \in X$, then the standard theory for existence and uniqueness of solutions to DE systems guarantees us a solution $\boldsymbol{x}(\tau)$ for $\tau \in [0 \dots \tau_{\boldsymbol{x}_0}]$, where $\tau_{\boldsymbol{x}_0} \leq \tau_f$, with $\boldsymbol{x}(\tau)$ differentiable in the interval $[0 \dots \tau_{\boldsymbol{x}_0}]$ and satisfying the DE system, and such that we have $\forall \tau \bullet \tau \in [0 \dots \tau_{\boldsymbol{x}_0}] \Rightarrow \boldsymbol{x}(\tau) \in X$. See [5] for details.

Let us abbreviate $[iv]\, \mathfrak{D}\, \boldsymbol{x} = F(\boldsymbol{x}, \boldsymbol{y}, \tau)$ **until** g, to $[iv]\, \mathcal{DE}$ **until** g below. For soundness, we assume all the properties above regarding F hold, but it is impractical to include in the syntax all the data needed to establish them. Even including such data would still leave the problem of proving the properties needed — not trivial in general. So our view is that the presence of F in the language construct is accompanied, behind the scenes, by the needed data, together with proofs that the requisite properties hold.

Along with the properties of F, we need to know that on entry to $\mathcal{DE}$, the iv properties hold. This means that $[\boldsymbol{x}_0 \in X \wedge P(\boldsymbol{x}_0)]$, where $P(\boldsymbol{x}_0)$ denotes any properties needed beyond the domain requirement $\boldsymbol{x}_0 \in X$. The semantics of iv is as for any other guard. If iv holds, then the guard succeeds immediately, and execution of $\mathcal{DE}$ commences. If iv fails, then the process pauses, a clock is started, and it runs until the environment makes iv true, at which point the guard succeeds.

Assuming the guard has succeeded, a fresh clock is started to monitor the progress of the solution to $\mathcal{DE}$ — this clock is the one that is referred to as τ in the expression $F(\boldsymbol{x}, \boldsymbol{y}, \tau)$. We are guaranteed that the solution exists for some period of time.[2]

There remains the preemption guard g. As for $@g$, for the preemption moment to be well defined, we demand that the true-set of g is **closed**. If during the period $[0 \dots \tau_{\boldsymbol{x}_0}]$ for which we have a solution, g becomes true, execution of the solution is stopped and the execution of the whole construct $[iv]\, \mathcal{DE}$ **until** g succeeds. If during the period $[0 \dots \tau_{\boldsymbol{x}_0}]$, g never becomes true, then as in other cases, the execution of $[iv]\, \mathcal{DE}$ **until** g stops once $\tau_{\boldsymbol{x}_0}$ is reached. This completes the operational semantics of the DE construct.

Thus far we have covered the semantics of individual constructs in terms of their individual durations. DEs, positive delays, and unsatisfied guards have all acquired non-

[2] The period of time during which the solution exists may be very short indeed. If $\boldsymbol{x}_0$ is right at the boundary of X and F is directed towards the exterior of XY, then $\tau_{\boldsymbol{x}_0}$ may equal 0, and the initial value may be all that there is. This makes the $\mathcal{DE}$ execution equivalent to **skip**.

zero durations. Non-positive delays and immediately satisfied guards are instantaneous, but since they do not change the state, we can allow them to complete immediately.

Atomic updates *do* change the state though. And to ensure that (packages of) atomic changes of state take place at isolated points in time, to execute an update, we start a clock which runs for a finite, unspecified, (but typically short) time, during which a non-clashing time point is chosen and the update is done. Non-clashing means that the update is separated from time points specifying other semantic events.

The remaining outer level constructors offer few surprises. Sequential composition, $P_1\ ;\ P_2$, starts by executing P_1, and if it terminates after a finite time, then P_2 is started. The conditional $\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\ \mathbf{fi}$ is familiar. Depending on the (instantaneous) truth value of b, the execution of either P_1 or P_2 is started, and the other is forgotten. For iteration, $\mathbf{while}\ b\ \mathbf{do}\ P$, if b is false, the construct terminates. If b is true, the execution of P is started. If it completes in finite time, the whole process is repeated. The parallel construct $P_1 \parallel P_2$ denotes programs P_1 and P_2 running concurrently.

With the above, we can describe the runs of a program, having characteristics that are consistent with the physical picture we would want in a formalism that includes DEs, by giving, for each variable, a function of time that gives its value at each moment. For discrete variables, such a function is piecewise constant, being constant on left-closed right-open intervals, with an atomic update at t_α say, taking the left-limit value at t_α to the actual value at t_α. For continuous updates running till τ_g, we remove the final value of an interval $[0 \ldots \tau_g]$, getting a left-closed right-open interval again, and interpreting the guard g as the left-limit value at τ_g.

3 Example: An Air Conditioning System

We illustrate how the language $Pr1$ works via a simplified air conditioning example. Although failures in AC systems are typically not critical, the kind of modelling needed, and the issues to be taken into account regarding the modelling, are common to systems of much higher consequence, making the simple example useful.

The AC system is controlled by a $User$. The user can switch it on or off, using the boolean $runAC$. The user can also increase or decrease the target temperature by setting booleans $tempUp$ and $tempDown$. Since $Pr1$ does not have pure events as primitives, the AC system reacts on the rising edges of $tempUp$ and $tempDown$, resetting these values itself (whereas it reacts to both the rising and falling edges of $runAC$).

Here then is the $User$ program. In the following, we assume available a function rnd, that returns a random non-negative integer value. Note that $runAC$, $tempUp$ and $tempDown$ are not declared here since they need to be declared in an outer scope.

```
$User =$
  while true
  do $\#(\mathsf{rnd})\ ;\ runAC := \mathbf{true}\ ;\ cnt : \mathbb{N} = \mathsf{rnd}\ ;$
     while $cnt > 0$
     do $\#(\mathsf{rnd})\ ;\ \mathbf{if}\ \mathsf{rnd}\ \%\ 2\ \mathbf{then}\ tempUp := \mathbf{true}\ \mathbf{else}\ tempDown := \mathbf{true}\ \mathbf{fi}$
     od ;
     $runAC := \mathbf{false}$
  od
```

The above models the nondeterministic behaviour of the user by using random waits between user events, and random counts of temperature modification commands. This is evidently a bit clumsy, but is adequate for purposes of illustration.

The AC apparatus consists of a room unit and an external unit. It operates on a Carnot cycle, in which a compressible fluid (passed between the two units via insulated piping) is alternately compressed and expanded. The fluid is compressed in the external unit to raise its temperature higher than the surroundings, where it is cooled by forced ventilation to (close to) the temperature of the surroundings. The fluid is then expanded, cooling it, so that, in the room unit, it is cooler than the room, and forced ventilation with the room's air warms it again, thus cooling the room. The cycle runs continuously. The inefficient thermodynamics of the Carnot cycle means this process cannot work without a constant input of energy, making AC systems expensive to run.

Our simplified model of AC operation depends on a number of temperature variables, reflecting the structure of the Carnot cycle: θ_S is the room temperature set by the user; θ_R is the current room temperature; θ_X is the temperature of the external unit's surroundings; θ_{FH} is the temperature of the fluid when compressed; θ_{FL} is the temperature of the fluid when expanded. All of these are real valued.

When an AC system is started, each part will be at the temperature of its own surroundings, and there will be a transient phase during which the AC system reaches its operating conditions. For simplicity we ignore this, and our model starts in a state in which all components are initialised to their operating conditions. Consequently θ_{FH}, θ_{FL} and θ_X are assumed constant, so do not require their own dynamical equations.

For simplicity we further assume that θ_{FH} is independent of other quantities, and that θ_{FL} is lower than θ_{FH} by an amount proportional to $\theta_{FH0} - \theta_{X0}$. We also assume that when operating, the AC system cools the room air according to a linear law.

```
$ACapparatus =$
  $\theta_S : \mathbb{N} \cap [S_L \dots S_H] = \theta_{S0}$ ; $\theta_R : \mathbb{R} \cap [R_L \dots R_H] = \theta_{R0}$ ;
  [ $\theta_X : \mathbb{R} \cap [X_L \dots X_H] = \theta_{X0}$ ;
   $\theta_{FH} : \mathbb{R} = \theta_{FH0}$ ; $\theta_{FL} : \mathbb{R} = \theta_{FL0} = \theta_{FH0} - K_X(\theta_{FH0} - \theta_{X0})$ ; ]
  **while true**
  **do** $@(runAC = \textbf{true})$ ;
    **while** $runAC = \textbf{true} \wedge \theta_R > \theta_S$
    **do** [ $\theta_R \in [R_L \dots R_H]$ ]
      $\mathcal{D}\,\theta_R = -K_R(\theta_R - \theta_{FL})$ **until**
      $(\theta_R = \theta_S \vee tempUp = \textbf{true} \vee tempDown = \textbf{true} \vee runAC = \textbf{false})$ ;
      **if** $tempUp = \textbf{true}$
      **then** $\{tempUp, \theta_S := \textbf{false}, \min(\theta_S + 1, S_H)\}$
      **elsif** $tempDown = \textbf{true}$
      **then** $\{tempDown, \theta_S := \textbf{false}, \max(\theta_S - 1, S_L)\}$
      **elsif** $\theta_R = \theta_S$
      **then** $@(\theta_R = \theta_S + 1)$
      **else skip**
      **fi** ;
    **od** ;
    $@(\theta_R \geq \theta_S + 1)$
  **od**
```

Putting *User* and *ACapparatus* together gives us the complete system.

$$\begin{array}{l} ACsystem = \\ \quad runAC : \mathbb{B} = \textbf{false} \,;\, tempUp : \mathbb{B} = \textbf{false} \,;\, tempDown : \mathbb{B} = \textbf{false} \,; \\ \quad (User \mid\mid ACapparatus) \end{array}$$

Note that in the above, while *runAC* works as a toggle, *tempUp* and *tempDown* are reset by the apparatus. Finally, we recognise that for a sensibly behaved system, we would need a considerable number of relations to hold between all the constants that implicitly define the static structure of the system.

4 Healthiness Considerations

At this point we step back from the detailed discussion of the example to cover a number of general considerations that arise when physical systems interact with computing formalisms.

[1] Allowing all variables of interest to be considered as functions of time yields a convenient uniformity between isolated discrete updates and continuous updates. Treating the two kinds in different ways can lead to a certain amount of technical awkwardness, at the very least.

[2] When variables are functions of time, values at individual points in time have no physical significance. Only values aggregated over an interval of time make sense physically, and for these to be well defined, the functions of time in question have to be well behaved enough (e.g. ‘continuous’, although ‘integrable’ would actually suffice).

[3] In dealing with CPS systems we must take into account the consequences of using differential equations. In a sense we have already fallen into covering this quite extensively in discussing the semantics of our prototypical language in Section 2. The existence of solutions to arbitrary DEs cannot be taken for granted without the imposition of appropriate sufficient conditions. An easy way to ensure this is to impose strict syntactic restrictions on the permitted DEs, e.g. by insisting that they are linear.

[4] Physics is relentlessly *eager*. In conventional discrete system formalisms, assuming that the discrete events in question are intended to correspond with real world events, the precise details of the correspondence with moments of time is seldom critical (other than for explicitly timed systems), and more than one interpretation is permissible, provided the causal order of events remains the same. As soon as physical behaviour enters the scene though, this choice disappears. If one physical behaviour stops, another must take over immediately, as the universe does not ‘go on hold’ until some new favourable state of affairs arises.

[5] Point **[4]** places quite strong restrictions on the semantics of languages intended for the integrated descriptions of computing and physical behaviour, since many of the options available for discrete systems simply disappear. Although it is perfectly possible to design languages that ignore this consideration and integrate continuous behaviour and discrete behaviour in an arbitrary fashion, even though they may be perfectly consistent mathematically, unless they take due consideration of the requirements of the physical world, they are irrelevant for the description of real world systems.

[6] Points **[4]** and **[5]** boil down to a requirement that decriptions of physical behaviour must be guaranteed to be *total* over time. Languages intended for CPS and critical systems should not permit gaps in time during which the behaviour of some physical component is undefined.

[7] The requirements of the last few points can be addressed by having separate formalisms for the discrete and continuous behaviours of the whole system and having a well thought out framework for their interworking. However, in cases of multiple cooperating formalisms, it is always the cracks between the formalisms that make the most hospitable hiding places for bugs, so particular vigilance is needed to prevent that.

[8] The impact of the preceding points may be partly addressed by careful syntactic design — we demonstrate this to a degree in Section 5. However, most aspects are firmly rooted in the semantics. In this regard, a language framework that puts such semantic criteria to the fore is highly beneficial. The semantic character of most of the issues discussed implies that an approach restricted to syntactic aspects can only achieve a very limited amount.

[9] The implications of the heavily semantic nature of most of the issues discussed above further implies the necessity of having runtime abortion as an ingredient of the operational semantics of any language suitable for the purposes we contemplate. Although this is seldom an issue *per se* for practical languages, which must include facilities for division, hence for division by zero at runtime, it is nevertheless perfectly possible to contemplate languages in which all primitive expression building operations are total, and hence to dispense with runtime abortion, even if such languages are of largely theoretical interest.

The overwhelmingly semantic nature of the preceding discussion motivates our referring to the matters raised as 'healthiness conditions'. (The nomenclature is borrowed from UTP [3], where appropriate structural conditions that play a similar role are baptised thus.) Checking that the necessary conditions hold for a given system, compels checking that the relevant criteria, formulated as suits the language in question, hold for the system at runtime (for the entire duration of the execution). Depending on the language and how it is structured, this may turn out to be more convenient or less convenient.

5 An Improved Concurrent Language

Taking on board the discussion in Section 4, we redesign our language as follows.

$$\begin{aligned}
&\mathit{Decl} \ ::= \ [\, x : T \; [= x_0\,] \; ; \,]^* \\
&\mathit{Db} \ ::= \ x := e \mid \{xs := es\} \mid @b \mid \#r \\
&\mathit{Pr}0 \ ::= \ \mathit{Db} \mid \mathit{Decl} \; ; \; \mathit{Pr}0 \mid \mathit{Pr}0 \; ; \; \mathit{Pr}0 \\
&\qquad\quad \mid \mathbf{if}\ b\ \mathbf{then}\ \mathit{Pr}0\ \mathbf{else}\ \mathit{Pr}0\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ \mathit{Pr}0\ \mathbf{od} \mid \mathit{Pr}0 \parallel \mathit{Pr}0 \\
&\mathit{CbE} \ ::= \ [iv]\, \mathcal{D}\, \boldsymbol{x} = F(\boldsymbol{x}, \boldsymbol{y}, \tau)\ \mathbf{until}\ g \mid \mathbf{obey}\ \mathit{Rstr}\ \mathbf{until}\ g \\
&\mathit{Pr}2 \ ::= \ \mathit{CbE} \mid \mathit{Pr}2 \; ; \; \mathit{Pr}2 \\
&\qquad\quad \mid \mathbf{if}\ b\ \mathbf{then}\ \mathit{Pr}2\ \mathbf{else}\ \mathit{Pr}2\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ \mathit{Pr}2\ \mathbf{od} \mid \mathit{Pr}2 \parallel \mathit{Pr}2 \\
&\mathit{PrSys} \ ::= \ \mathit{Name} \mid [\, \mathit{Name} \ =]\ \mathit{Decl} \; ; \; \mathit{PrSys} \mid \mathit{Pr}0 \mid \mathit{Pr}2 \mid \mathit{PrSys} \parallel \mathit{PrSys}
\end{aligned}$$

In the above grammar, the healthiness considerations that can be addressed via the syntax have been incorporated. Thus, there is a visible separation between the previous discrete program design $Pr0$ (which remains unchanged), and the provisions made for describing physical behaviour $Pr2$, which have been restructured.

Specifically, there are now no facilities for $Pr2$ processes to wait. Furthermore, they can only be combined with discrete processes at top level, precluding their sudden appearance part way through a system run. This also means that they must be declared at top level, reflected in the design of the $PrSys$ syntax.

Note the additional $\mathbf{obey}$ clause for physical behaviour. This permits relatively loosely defined behaviour to be specified in cases where more prescriptive behavour via a DE is not desired or is impossible due to lack of knowledge, etc. This replaces use of waiting clauses in the earlier grammar. Note that DE behaviour and $\mathbf{obey}$ behaviour are the *only* permitted ways of describing continuous behaviour at the bottom level.

Although we have ensured that $Pr2$ processes cannot wait for syntactic reasons, we have to ensure that they can't wait for semantic reasons either. Thus we must stipulate what happens in the DE and $\mathbf{obey}$ cases when one or other of their syntactic components fails. Taking the DE case first, if iv does not evaluate to **true**,[3] then the whole top level $PrSys$ process must abort, that is to say, execution terminates abruptly in a failing state. If F fails to satisfy the conditions for existence of a DE solution, then the top level $PrSys$ process aborts. If g does not evaluate to **true** at some moment in the DE solution, in case that the duration of the DE solution τ_f is finite, then when τ_f is reached, the top level $PrSys$ process aborts. Turning to the $\mathbf{obey}$ case, if $Rstr$ does not evaluate to **true** in a left closed right open time interval starting from the moment the $\mathbf{obey}$ construct is encountered (or amounts to $\mathbf{skip}$ at that moment), then top level $PrSys$ process aborts. If g does not evaluate to **true** at some moment during the **true** interval of $Rstr$, in case that the duration of the **true** interval of $Rstr$, say τ_f, is finite, then when τ_f is reached, the top level $PrSys$ process aborts.

Having defined the improved language, we can check over how it addresses the healthiness conditions described earlier. Re. **[1]**, we have already stipulated that all variables depend on time in our description of the semantics, so **[1]** is covered. Re. **[2]**, this is again implicit in our semantics. Likewise, **[3]** is also covered by our relatively detailed discussion of DEs. Re. **[4]**, we have designed the syntax to prohibit explicit lazy behaviour in the continuous domain, and this is backed up by the semantics which disallows lazy behaviour arising from runtime conditions — this justification extends to point **[5]**, and this, combined with the fact that DE behaviour and $\mathbf{obey}$ behaviour are the only permitted ways of describing continuous behaviour at the bottom level guarantee totality over time *provided* the behaviour described by the syntax is well defined semantically, covering point **[6]**. Points **[7]** and **[8]** are things that can be achieved syntactically, and our design does so. Point **[9]** indicates the necessity of having runtime aborts in the semantics, this being forced by the eagerness of physical behaviour. The need for this also arose in our remarks regarding point **[6]**.

The heavy dependence on semantics of this discussion raises the question of how we can be sure that any system that is written down defines a sensible behaviour. In

[3] That is to say, it evaluates to **false**, or fails to evaluate at all.

purely discrete languages, there is a well trodden route from the syntactic structure of a system description to verification conditions that confirm the absence of runtime errors.

The same approach extends to languages containing continuous update, such as ours. The syntactic structure of such a language can be analysed to elicit all the dependencies between different syntactic elements that can arise at runtime, and these dependencies can be used to create template verification conditions. Given a specific model, the generic template verification conditions can be instantiated to the elements of the model to provide sufficient (although not necessarily necessary) conditions for runtime well definedness. Still, it has to be conceded that such conditions can be more challenging than in the discrete case because of the more subtle nature of aspects of continuous mathematics.

Although we do not give a comprehensive account of the verification templates for our (improved) language (it has, after all, been constructed just for illustrative purposes), we can give an indication of a couple of them.

Thus, if the flow of control reaches an DE construct $[iv]\mathcal{D}\boldsymbol{x} = F(\boldsymbol{x}, \boldsymbol{y}, \tau)$ $\mathbf{until}$ g we need to know the initial value guard will succeed. We can ensure statically that this will be the case if the DE construct occurs in a case analysis whose collection of guards covers all values that could be generated.

Similarly, once a DE construct has been preempted by its preemption guard becoming true, we need to ensure that there is a viable continuous successor behaviour for the physical process to engage in. This is helped in our case by the syntax, and can be supported by a proof that the truth of the preemption guard enables some syntactically available successor option.

In the discrete part of the language, the success of an $\mathbf{if}$ statement can be assured provided there is a default $\mathbf{else}$ clause to capture any exceptional cases. And so on.

Still, achieving full static assurance of freedom from runtime errors may require fully simulating the system, which will usually be impractical. Much depends on the language design. To help the process, languages may be designed in which all expression forming constructs are guaranteed to denote (e.g. *in extremis* by not having division in the language). Such languages may help in the verified design of critical systems.

6 The Running Example, Improved

In the light of the preceding discussions, we return to our running example and restructure it for the improved language. For simplicity we will omit the bracketed constant declarations that appeared in the earlier *ACapparatus*. We also keep the definition of the *User* the same, as that conforms to the syntax of the improved language. Regarding the *ACapparatus*, it requires some significant restructuring.

Firstly, the previous design mixed discrete and continuous update in a fairly uncritical manner. Thus the DE $\mathcal{D}\,\theta_R = -K_R(\theta_R - \theta_{FL})$, describing the fluid behaviour, is mixed with discrete updates to θ_S, done at the behest of the *User*. Worse, when the DE is preempted, no physical behaviour is defined for the fluid — the *ACapparatus* just hangs around waiting for the next opportunity to do some cooling. This is not really acceptable: the fluid does not stop being a physical system, subject to the laws of nature, just because, with our focus on the *ACapparatus* design, we have no great interest in its behaviour during a particular period.

Our restructured design separates the physical from the discrete aspects. The earlier *ACapparatus* is split into an *ACcontroller* process, looking after the discrete updates, and a *ACfluid* process, which describes the physical behaviour of the fluid.

Normally, the *User* would communicate with the *ACcontroller*, which would then control the *ACfluid*, but we are a bit sloppy, and allow the *User*'s *runAC* variable to also directly control the *ACfluid*, thus sharing the fluid control between the *User* and the *ACcontroller*. The latter therefore just controls the θ_S value while *runAC* is **true**.

The *ACfluid* process, now constrained by the restricted syntax for physical processes, describes the fluid's properties at all times. At times when the DE behaviour is not relevant, an **obey** clause defines default behaviour, amounting to θ_R remaining within the expected range. The separation of control and fluid allows us to make the fluid responsible for detecting temperature and to only initiate the DE behaviour when the temperature is at least a degree above the set point θ_S. Of course this is rather unrealistic, and a more credible (and detailed) design would involve sensors under the control of the *ACcontroller* to manage this aspect.

```
$ACcontroller =$
  while true
  do $@(runAC = \textbf{true})$ ;
    while $runAC = \textbf{true} \wedge \theta_R > \theta_S$
    do $@(tempUp = \textbf{true} \vee tempDown = \textbf{true} \vee runAC = \textbf{false})$ ;
      if $tempUp = \textbf{true}$
      then $\{tempUp, \theta_S := \textbf{false}, \min(\theta_S + 1, S_H)\}$
      elsif $tempDown = \textbf{true}$
      then $\{tempDown, \theta_S := \textbf{false}, \max(\theta_S - 1, S_L)\}$
      fi
    od
  od

$ACfluid =$
  while true
  do obey $\theta_R \in [R_L \ldots R_H]$ until $runAC = \textbf{true}$ ;
    if $\theta_R \geq \theta_S + 1$
    then $[\,\theta_R \in [R_L \ldots R_H]\,]\ \mathfrak{D}\,\theta_R = -K_R(\theta_R - \theta_{FL})$
         until $(\theta_R = \theta_S \vee runAC = \textbf{false})$
    else obey $\theta_R \in [R_L \ldots R_H]$ until $\theta_R \geq \theta_S + 1 \vee runAC = \textbf{false}$
    fi
  od
```

Putting all three components together gives us the complete system.

```
$ACsystem =$
  $runAC : \mathbb{B} = \textbf{false}$ ; $tempUp : \mathbb{B} = \textbf{false}$ ; $tempDown : \mathbb{B} = \textbf{false}$ ;
  $(\,User \mid\mid (\,\theta_S : \mathbb{N} \cap [S_L \ldots S_H] = \theta_{S0}$ ; $\theta_R : \mathbb{R} \cap [R_L \ldots R_H] = \theta_{R0}$ ;
           $ACcontroller \mid\mid ACfluid\,)\,)$
```

7 Related Approaches

It is fair to say that the critical systems industry is rather conservative — advocating radical new ways of doing things that do not enjoy the highest levels of trust risks major disasters in the field. Even the newer standards in key fields, such as DO-178C (for avionics [8]), ISO 26262 (for automotive systems [9]), IEC 62304 (for medical devices [10]), or CENELEC EN 50128 (for railway systems [11]), are still heavily weighted in favour of mandating specific testing strategies, and other practices heavily rooted in traditional development techniques. Thus the entry of formal techniques into the standardised critical systems development portfolio is rather cautious, despite the strong evidence in niche quarters about the dependability that can be gained by appropriate use of formal development, suitably integrated into the wider system engineering process. This is as much because entrenched industrial practice cannot move as nimbly as one might hope, even when the evidence for attempting to do so is relatively strong.

Here, we briefly comment on some approaches that compare with our exercise to realign a candidate language for utility in the cyberphysical and critical systems arena.

In the cyberphysical systems area [12–14], we can point to the extensive survey [15], which covers a wide spectrum of research into cyberphysical systems, and the tools and techniques used in that sphere. As we might expect, despite the relative newness of the cyberphysical systems area, formal approaches are somewhat overshadowed by more traditional and simulation based techniques. Again, this is due to the fact that cyberphysical systems still have to be built, and this falls back on traditional approaches.

The older survey [16] is more linguistically based and covers a large spectrum of languages and tools for hybrid systems. One is struck by the typically low expressivity in the continuous sphere of many of the systems discussed there, motivated, of course, by the desire for decidability of the resulting languages and systems. For decidability reasons, most of these are based on variations of the hybrid automaton concept [17–19]. In fact, for simple linear behaviours, e.g. $\mathcal{D}\,x = K$, with K constant, there is very little difference between using a DE as just quoted, and using an expression $x' = x + K\varDelta T$ where $\varDelta T$ is the duration of the behaviour.

Neverthless, many of the formalisms in these sources are focused on the single goal of hybrid or continuous behaviour, to the exclusion of more general computing concerns. This leads to the 'bugs in between formalisms' risk noted earlier, when multiple formalisms need to be combined.

Closer to our perspective is the work of Platzer [20], supported by the KeyMaera tool [21]. This supports the kind of modelling exemplified in this paper, with a strong focus on verification. Alternatively there is the Hybrid Event-B formalism [22, 23]. This is an extension of the pure discrete event formalism Event-B [24], building on the earlier classical B-Method [25], (which is still actively used in critical applications in the urban rail sector [26]). The extension is expressly designed to avoid the kind of traps regarding continuous behaviour and verification that we illustrated earlier in this paper.

Thus far our discussion has avoided mentioning noise or randomness. This is legitimate when the physical considerations imply that it is negligible. But if sources of uncertainty are significant, then probabilistic techniques need to be taken on board. These add nontrivial complication to the semantics of any language. An indication of the issues that can arise can be found in [27, 28].

8 Conclusion

Motivated by the current dramatic proliferation in critical and cyberphysical systems, especially in urbanised areas all over the world, in the preceding sections, we examined the problem of extending typical existing, more conventional formalisms for programming, to allow them to incorporate the needed physical behaviour that is a vital ingredient of these systems. Such integrated formalisms can come into their own if we contemplate the *integrated* verification of critical cyberphysical systems, in which we seek to avoid the possibilities of there being bugs that hide in the semantic cracks between separate formalisms that are used to check separate parts of the behaviour.

Rather than being comprehensive, our approach in this paper has been to illustrate the range of issues to be considered, by taking a somewhat prototypical shared variable language for concurrent sequential programming, and extending it in a relatively naïve way to incorporate continuous behaviour. We then critically examined the consequences of this, and identified a number of issues that are not always taken sufficient account of when embarking on such an extension exercise. For want of a pithy name, we termed these 'healthiness considerations', by analogy with the nomenclature used in UTP. This done, we showed how the earlier naïve syntax could be improved to partially address some of these issues, the remainder being the responsibility of the semantics.

We illustrated our particular solution with a simplified air conditioning system, giving the core steady state behaviour in both the original and improved formulations.

It is important to emphasise that we do not claim that the details of our solution (even in the case of our specific language) are unique. One could resolve the same issues in a number of ways that differed in the low level detail. Nevertheless, the broad sweep of the things needing to be considered would remain similar.

We also do not claim that our language (and its improved version) are to be particularly recommended for critical cyberphysical system development. In many ways, the issues we have striven to highlight are brought our more clearly in a language which one would rather *not* choose to use.

We can liken the urge to match the surface syntactic features of the language as closely as possible to what is needed by the semantics of the physical considerations, with the longstanding process whereby machine code was superseded by assembly language, which was superseded by higher level languages, etc., in each case the desire being to raise the level of abstraction in such a way as to preclude as many user level errors as possible by making them syntactically illegal (or simply impossible to express), and backing this up semantically.

It is to be hoped that the insights from an exercise like the one we have undertaken can help to improve the broader awareness of the issues lurking under the bonnet when formalisms for critical and cyberphysical systems are designed in future.

References

1. Alur, R.: Principles of Cyberphysical Systems. MIT Press (2015)
2. Lee, E., Shesha, S.: Introduction to Embedded Systems: A Cyberphysical Systems Approach. 2nd. edn. LeeShesha.org (2015)

3. Hoare, T., , He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
4. Zhou, C., Hoare, T., Ravn, A.: A Calculus of Durations. Inf. Proc. Lett. **40** (1991) 269–276
5. Walter, W.: Ordinary Differential Equations. Springer (1998)
6. Horn, R., Johnson, C.: Matrix Analysis. Cambridge University Press (1985)
7. Horn, R., Johnson, C.: Topics in Matrix Analysis. Cambridge University Press (1991)
8. DO-178C: http://www.rtca.org.
9. ISO 26262: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=54591.
10. IEC 62304: https://webstore.iec.ch/preview/info_iec62304{ed1.0}en_d.pdf.
11. CENELEC EN 50128: https://www.cenelec.eu/dyn/www/f?p=104:105.
12. 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.
13. Willems, J.: Open Dynamical Systems: Their Aims and their Origins. Ruberti Lecture, Rome (2007) http://homes.esat.kuleuven.be/~jwillems/Lectures/2007/Rubertilecture.pdf.
14. National Science and Technology Council: Trustworthy Cyberspace: Strategic plan for the Federal Cybersecurity Research and Development Program (2011) http://www.whitehouse.gov/sites/default/files/microsites/ostp/fed_cybersecurity_rd_strategic_plan_2011.pdf.
15. 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.
16. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and Tools for Hybrid Systems Design. Foundations and Trends in Electronic Design Automation **1** (2006) 1–193
17. Henzinger, T.: The Theory of Hybrid Automata. In: Proc. IEEE LICS-96, IEEE (1996) 278–292 Also http://mtc.epfl.ch/~tah/ Publications/the_theory_of_hybrid_automata.pdf.
18. Alur, R., Courcoubetis, C., Henzinger, T., Ho, P.H.: Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In: Proc. Workshop on Theory of Hybrid Systems. Volume 736 of LNCS., Springer (1993) 209–229
19. Alur, R., Dill, D.: A Theory of Timed Automata. Theor. Comp. Sci. **126** (1994) 183–235
20. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
21. Symbolaris: http://www.symbolaris.org.
22. Banach, R., Butler, M., Qin, S., Verma, N., Zhu, H.: Core Hybrid Event-B I: Single Hybrid Event-B Machines. Sci. Comp. Prog. **105** (2015) 92–123
23. Banach, R., Butler, M., Qin, S., Zhu, H.: Core Hybrid Event-B II: Multiple Cooperating Hybrid Event-B Machines. Sci. Comp. Prog. (2017) to appear.
24. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
25. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
26. Clearsy: http://www.clearsy.com/en/.
27. Zhu, H., Qin, S., He, J., Bowen, J.: PTSC: Probability, Time and Shared-Variable Concurrency. Innov. Syst. Softw. Eng. **5** (2009) 271–284
28. Zhu, H., Yang, F., He, J., Bowen, J., Sanders, J., Qin, S.: Linking Operational Semantics and Algebraic Semantics for a Probabilistic Timed Shared-Variable Language. J. Log. Alg. Prog. **81** (2012) 2–25