Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Language Evolution and Healthiness for Critical Cyber-Physical Systems†

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

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

[2]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Zhongshan Bei Lu, Shanghai 200062, China

Correspondence

*Richard Banach, Department of Computer Science, University of Manchester. Email: richard.banach@manchester.ac.uk

Abstract

In the effort to develop critical cyber-physical systems, it is tempting to extend existing computing formalisms to include continuous behaviour. This may happen in a way that neglects elements necessary for correctly expressing continuous properties of the mathematics and correct physical properties of the real world physical system. A simple language is taken to illustrate these possibilities. 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 elaborated, resulting in the language CDPPP (Combined Discrete and Physical Programs in Parallel). An example air conditioning system is used to illustrate the concepts presented, and it is developed both in the original 'unhealthy' language and in the modified 'healthier' CDPPP. The formal semantics of the improved language is explored.

KEYWORDS:
language features, healthiness criteria, cyber-physical systems.

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.[1]

In this paper we intend to illuminate the imbalance that we perceive 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]. Briefly, the main points of this can be indicated as follows.

- Physical behaviour extends over real time, and computing metaphors that interact intimately with it must be consistent with that.
- Differential equations are unavoidable, and their properties (and limitations) must be properly integrated into the expressive framework.
- Physics is eager. Any description of a physical process must be total over the whole time interval of interest.

†The work reported here was partly 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 Key Research and Development Program of China (Grant No. 2018YFB2101300), National Natural Science Foundation of China (Grant No. 61872145), and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (Grant No.ZF1213).

[1]Having said that, much verification work, as presented in, e.g. the HSCC series, proves to be soundly based, due to conservative design choices. See further discussion in Section 8 .

Some of the ramifications of these observations can be quite subtle, and act at a deep semantic level, for example the precise formulation of conditions that govern how the semantics of differential equations interfaces with the semantics of the remainder of the language. We discuss these in due course. Other aspects are more immediately perceptible at the syntactic level. For example, the last point precludes inserting continuous dynamics facilities into the language in an arbitrary way, and this flies in the face of the impulse towards syntactic orthogonality and the arbitrary nesting of syntactic constructs encouraged by BNF-like language definition mechanisms, particularly when the trends set by the Algol family (and especially its more elaborate variants like Algol-68[4,5]) are followed. Our example language development illuminates some of this in a pedagogical manner.

Having brought these issues into the light, we explore how to modify our original language to better take them into account. This can partly be done by reshaping the syntax, which is presented in its modified form, and which we call CDPPP (**C**ombined **D**iscrete and **P**hysical **P**rograms in **P**arallel). Remaining aspects of healthiness reside within the semantics, and we discuss their impact. It is worth saying that our language is not one that we would necessarily use seriously for such applications, but actually, its very lack of obvious suitability serves to better highlight the points we make, somewhat like a caricature attracts an increased amount of attention to certain aspects of its subject (without necessarily imputing anything negative thereby). A more wide ranging discussion of the context for languages and approaches for cyber-physical systems can be found in the related work section near the end of the paper.

We illustrate the above train of thought 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 in more abstract terms. We give a development in the original language, and an improved version in CDPPP.

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 the extension, even relatively informally, implies a substantial technical detour regarding the possibilities available when DEs are involved. For readability, the details of this are relegated to Appendix A and Appendix B. In Section 3 we give our initial AC system development. Given that this appears to work out successfully, in Section 4 we ask the question of what might be missing from the treatment thus far, and in doing so we identify the healthiness considerations mentioned above, considerably enlarging the earlier semantic discussion thereby. A large number of the points raised come from contemplating the physical world (as anticipated in the comments above), and the constraints imposed by physical theory. Such considerations are often neglected when computing science perspectives are to the fore. Section 5 then modifies the initial language syntactically, where possible, resulting in CDPPP. Section 6 redevelops the AC system in CDPPP, alleviating some of the problems noted earlier. Section 7 is more technical, and examines the formal operational semantics of CDPPP in detail, pointing out how the earlier healthiness considerations interact with the technical details. Section 8 considers related approaches. Section 9 concludes. Finally, Appendix A and Appendix B deal in more technical depth with differential equations, and with closure properties of the various guard constructs that arise in some of our language constructs.

This paper is an extended version of[6]. Aside from a much less terse style of discussion and a reorganised presentation of the material for greater readability, many additional relevant facts are included, and there is an extended related work section. The principal additional technical enhancement beyond[6] is the discussion of formal semantics in Section 7.

2 | AN INITIAL CONCURRENT LANGUAGE AND ITS CONTINUOUS EXTENSION

In this section we introduce a simple concurrent language and a naïve attempt to enhance it with facilities for continuous update. We discuss the semantics fairly informally, focusing on issues that are often neglected in such exercises. This implies a significant detour regarding properties of differential equations, most details of which are relegated to an appendix.

2.1 | The Initial Concurrent Language

We present the syntax of our initial language. This is a relatively conventional language featuring: parallel processes, shared variables, the usual structuring constructs, and allowing delays of a specified (ntegral) number of time units. Besides the syntax given, we use parentheses in the usual way for scoping, nesting and disambiguation. An explanation of the various constructs follows below.

Declarations:

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

Discrete behaviours:

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

Constructs:

P0 ::= Db

Programs:

Pr0 ::= P0 | Name | [Name =] Decl ; Pr0 | Pr0 ; Pr0 | **if** b **then** Pr0 **else** Pr0 **fi** | **while** b **do** Pr0 **od** | Pr0 || Pr0

In connection with this definition we make the following amplifying comments:

(1) All variables used, e.g. x have to be declared (and optionally, initialised, e.g. x_0) with their types T in a declaration block Decl in whose scope (defined, as usual, to be the immediately following maximal Pr0 block in the specification Decl ; Pr0) their uses occur. A Pr0 block may be given a Name to allow reuse in more than one place in a program. Obviously, each such Name should have a unique declaration somewhere in the program.

(2) The discrete variable assignment, conventionally written x := e, is *atomic*, so that no action can interleave the reading the variables needed for e, the calculation of the value of e, and the writing of the result to x. The vacuous assignment (the assignment which changes nothing) is written **skip**. Each variable has to be assigned an initial value (in terms of constants and already assigned variables) before it can be used. As noted, initialisation is optionally taken care of during declaration.

(3) The simultaneous assignment {xs := es} merely defines a *package* of several atomic updates, all of which are effected indivisibly at the same instant. In particular, no other action can interject between the reading of all the variable values needed for es and the updates to xs.

(4) The discrete event-guard, @b, where b is a Boolean condition, is enabled when the guard b holds; otherwise it is disabled. In the former case @b is equivalent to **skip**, so that nothing happens and execution of the program proceeds to the next construct. In the latter case the process executing @b is suspended and waits until b becomes true, at which point execution proceeds to the next construct. The delay guard #r represents an unconditional delay of r time units, the expression r being evaluated with respect to the valuation of variables at the moment that the #r construct is encountered during the execution of the program.

(5) The usual outer level program constructs are familiar. **if** b **then** P **else** Q **fi** is the conditional, choosing branch P or branch Q according to the value of Boolean condition b, and **while** b **do** P **od** is iteration, repeating P until the Boolean condition b becomes false. P ; Q is sequential composition: P is executed first, and provided it has terminated successfully, execution moves on to Q. Shared-variable concurrency is expressed via P || Q, where P and Q can contain the behaviours outlined. When P || Q is encountered, the original process executing the program splits, and then one process starts to execute P and the other process starts to execute Q.

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. A familiar alternative interpretation for appropriately structured families of updates arises in the duration calculus, in which sequences of updates can be associated with the same 'real time' instant [7].[2]

When we now reconsider the delay #r, things change. We are obliged to take note of real world time. Consequently we now stipulate that all (packages of) update execution instances have their own specific isolated points in time at which they execute. In effect, this partially restricts the semantics of our simple language, compared with possibilities allowed elsewhere.

We observe that a number of things are left vague in the preceding discussion. For instance, we have not defined what we mean by a process. For such things we can either rely on the reader's experience and intuition to furnish a workable definition, or we can suggest that the reader 'borrows' the approach to the relevant issue from Section 7, where there is much more precision about all semantic matters.

The preceding sets the scene for introducing continuous variable updates. As we do so, the character of the language changes subtly. The discrete event language given so far is a conventional imperative language, albeit a rather abstract one, since the guard commands @b and #r, as well as concurrent composition ||, are handled via abstract combinators rather than the library calls one would expect in a true implementation level language. When we add continuous update via differential equations (our preferred technique for the time being) the language acquires much greater specificational characteristics. A differential equation is a specification mechanism *par excellence* — it declares a property that one or more variables should satisfy without describing how that property is to be realised. The language thus becomes, much more, a modelling language. Using it, the intention is that we can describe how a system should behave, with certain aspects given explicitly, and other aspects more implicitly. We discuss the issues left open by this in Section 4.

[2] 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. We take suitable semantic precautions for this below.

2.2 | A Continuous Extension

We now extend the initial discrete concurrent language by adding the following syntactic facilities. Further discussion follows.

Continuous behaviours:

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

Constructs:

P1 ::= Db | Cb

Programs:

Pr1 ::= P1 | Name | [Name =] Decl ; Pr1 | Pr1 ; Pr1 | **if** b **then** Pr1 **else** Pr1 **fi** | **while** b **do** Pr1 **od** | Pr1 ‖ Pr1

Regarding the above we make the following additional comments:

(6) Declarations may now include continuous variables as well as discrete variables. The same scope rules that apply to the discrete variables are extended to also apply to the continuous case.

(7) The command @g waits for its guard g to be satisfied. It is like @b except that g may now contain continuous variables. This implies that the model of time that is used must be a continuous time model (whereas a discrete time model would be sufficient for Pr0 programs if all types T were themselves discrete).

(8) The differential equation (DE) command [iv] $\mathcal{D}\mathbf{x} = \mathsf{F}(\mathbf{x}, \mathbf{y}, \tau)$ **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. If iv is not satisfied, execution is delayed, just as for @b or @g. (The presence of DEs to model physical quantities again implies the use of physical continuous time, there no longer being any choice in the matter). 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 ensues ($\mathcal{D}$ denotes the time derivative, operating on **x**). The behaviour specified by the DE continues until the preempting guard g is satisfied or the DE itself becomes infeasible. The preempting guard g is a Boolean condition, like @g, and as soon as it becomes true the behaviour specified by the DE stops. If, however, the DE behaviour becomes badly defined before g becomes true, the behaviour specified by the DE also stops.

2.3 | Semantic Considerations

The above outline serves well for intuition. However, if we need to be precise, more details need to be carefully defined. Thus, while we have some leeway in interpreting pure discrete events semantically (for instance, in the difference between allowing updates at the same moment of time, or not, as discussed above), this evaporates when we add differential equations. At least it does so if we want a credible correspondence with the real world. Thus, while pure discrete event formalisms may, quite sensibly, be studied axiomatically, allowing *a priori* decisions to be made about how syntactically specified updates correspond to semantic notions, this is never the case for DEs.

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.[8] for a discussion of a few of the many options. In Appendix A we give a more incisive discussion of the issues surrounding differential equations — that account being one among a number of possbilities. Here, so as not to impede the exposition by a lengthy digression, we can simply assume that all the DE cases we need are covered by a range of syntactically recognisable patterns, whose solutions are known in the mathematical literature, and which consist of functions of (intervals of) time. With the preceding understood, we give an outline of the various elements of the semantics of the language Pr1 in the following terms.

Working bottom-up from the most basic ideas, the fundamental semantic concept is the state σ, a mapping from each variable v to a value in its type: $\mathsf{v} \mapsto \sigma(\mathsf{v})$. Obviously, the state changes at various points of the execution of a program written in our language.

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 usual purpose of this is to ensure different semantic events (most importantly, particular updates to the state) are separated in time.

The Db part of the language is unsurprising. The discrete atomic variable assignment, $\mathsf{x} := \mathsf{e}$, sends the state σ to the state $\sigma[\sigma(\mathsf{e})/\mathsf{x}]$, which is identical to σ, except at x, the value of $\sigma(\mathsf{x})$ becoming $\sigma(\mathsf{e})$. Similarly for packaged atomic updates $\{\mathsf{xs} := \mathsf{es}\}$, in which the state σ changes to the state $\sigma[\sigma(\mathsf{es})/\mathsf{xs}]$, implying simultaneous update of the sequence xs to the sequence of values $\sigma(\mathsf{es})$, all of which, are derived from σ, as usual.

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 evolution of the program state to make b true, at which point the program completes.

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

```
1   User =
2     while true
3     do #(rnd) ; runAC := true ; cnt : ℕ = rnd ;
4       while cnt > 0
5       do #(rnd) ; 6if rnd % 2 then tempUp := true else tempDown := true fi
7       od ;
8       runAC := false
9     od
```

FIGURE 1 The $User$ program in Pr1.

Regarding the continuous behaviours, @g, as already noted, may contain continuous variables, and thus may wait for an arbitrary (i.e possibly non-integral) number of time units, but otherwise is like @b.

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-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 executed at a moment in time which is chosen to be separated from any moment in time at which any other semantic event that updates the state in a discrete manner executes. This, perhaps, is a somewhat unusual semantics for atomic update, but it is justified on physical grounds, given that we cannot avoid dealing with real time, now that DEs are present in our language.

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. If P_1 takes forever, we never start P_2. The conditional **if** b **then** P_1 **else** P_2 **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, **while** b **do** P **od**, 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. and if any of the iterates of P takes forever, then nothing further takes place. The parallel construct $P_1 \parallel P_2$ denotes programs P_1 and P_2 running concurrently, and the discussion about process splitting that we gave for Pr0 applies here too. The parallel construct completes when both of its subprograms have completed.

With the above, we can describe the runs of a Pr1 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, and for precision, we stipulate that the intervals of time on which it is constant are left-closed right-open intervals.[3] By convention, an atomic update taking place at time t_α say, takes the left-limit value at t_α to the actual value at t_α.

For a continuous update (specified by a DE with a preemption guard g for example), that runs till a time τ_g say —at which point g preempts it— we remove the final value of the closed interval $[0 \ldots \tau_g]$, getting a left-closed right-open interval again, and interpreting the guard g as its left-limit value at τ_g.

Looking ahead a little, the main problem with Pr1 that is visible on the surface is that it mixes conventional computing discrete update, and continuous physical evolution, at arbitrarily deeply nested points of a program's syntax. This leads to a number of potential difficulties. Other issues lie deeper in the semantics and we discuss them in due course. In Section 7 we give a more detailed formal operational semantics of a modified version of the Pr1 language, derived from Pr1 by taking a number of further considerations, inspired from the physical domain, into account, and we discuss how these changes address the difficulties we identified.

3 | EXAMPLE: AN AIR CONDITIONING SYSTEM

We illustrate how the Pr1 language works via a simplified air conditioning example. Although failures in AC systems are typically not safety 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.

[3]Note that any left-closed right-open interval must be of strictly positive dutation, and that such intervals are closed under the *append* operation.

```
1   ACapparatus =
2     θS : ℕ ∩ [SL ... SH] = θS0 ; θR : ℝ ∩ [RL ... RH] = θR0 ;
3     [ θX : ℝ ∩ [XL ... XH] = θX0 ;
4       θH : ℝ = θH0 ; θL : ℝ = θL0 = θH0 − KX(θH0 − θX0) ; ]
5     while true
6     do @(runAC = true) ;
7         while runAC = true ∧ θR > θS
8         do [ θR ∈ [RL ... RH] ]
9             D θR = −KR(θR − θL) until
10            (θR = θS ∨ tempUp = true ∨ tempDown = true ∨ runAC = false) ;
11            if tempUp = true
12            then {tempUp, θS := false, min(θS + 1, SH)}
13            elsif tempDown = true
14            then {tempDown, θS := false, max(θS − 1, SL)}
15            elsif θR = θS
16            then @(θR = θS + 1)
17            else skip
18            fi ;
19        od ;
20        @(θR ≥ θS + 1)
21    od
```

FIGURE 2 The $ACapparatus$ in Pr1.

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 $runAC$).

Fig. 1 contains 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 in the $User$ program since they need to be declared in an outer scope. Fig. 1 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 it is adequate for purposes of illustration.

The $User$ controls an AC apparatus, simplified compared with reality for convenience, the Pr1 code for which appears in Fig. 2. 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, simultaneously 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, which is what makes AC systems rather 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, initialised to θ_{S0}, a value within a range of natural numbers defined by two constants, S_L and S_H; θ_R is the current room temperature, initialised to θ_{R0}, a value within a range of real numbers defined by two constants, R_L and R_H. The remaining declarations in the square brackets on lines 3 and 4 contain detail that may be dispensed with if one is prepared to accept the constants introduced at face value. Thus: θ_X is the temperature of the external unit's surroundings, initialised to θ_{X0}, a value within a range of reals defined by constants X_L and X_H; θ_H is the temperature of the fluid when compressed; θ_L is the temperature of the fluid when expanded, all of these being real valued. The fluid expanded temperature is defined via a simple proportionality to the difference between the fluid compressed temperature and the external temperature.

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 steady state operating conditions. Consequently θ_H, θ_L and θ_X are assumed constant, so do not require their own dynamical equations, in line with the 'optional' declarations in red square brackets on lines 3 and 4.

For simplicity we assume that θ_H is independent of other quantities, and that θ_L is lower than θ_H by an amount related to $\theta_{H0} - \theta_{X0}$. We also assume that when operating, the AC system cools the room air according to a simple linear law $\mathcal{D}\,\theta_R = -K_R(\theta_R - \theta_L)$, with K_R constant.

1 $ACsystem =$
2 $runAC : \mathbb{B} = \textbf{false}$; $tempUp : \mathbb{B} = \textbf{false}$; $tempDown : \mathbb{B} = \textbf{false}$;
3 $(User \mid\mid ACapparatus)$

FIGURE 3 The complete AC system in Pr1.

Putting the *User* and the *ACapparatus* together gives us the complete system, shown in Fig. 3. We see that it includes a number of declarations omitted earlier. Note that in Fig. 3, while *runAC* works as a toggle, *tempUp* and *tempDown* are reset by the apparatus. Finally, we recognise that for a sensibly behaved system, even as crude and simple a system as we have described, we would need a considerable number of relations to hold between all the constants that implicitly define the static structure of the system.

4 | WHAT'S WRONG WITH PR0? — 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.

[A] Firstly, we know that continuous variables must be considered as functions of real time; physics demands this (in the classical physics sphere). For discrete variables, we have a number of possibilities, since the main fact about them is that they take on a sequence of values during a run of the system. We thus have the option of mapping the sequence of values to a function of time which is piecewise constant, with each value in the sequence holding for the period of time that elapses between the relevant pair of update events. Allowing all variables of interest to be considered as functions of time yields a convenient uniformity between the isolated updates of the discrete variables and the continuous updates of the continuous variables. Treating the two kinds of variable and/or the two kinds of update in different ways, can lead, at the very least, to a certain amount of technical awkwardness, not to exclude mathematically sound but physically unacceptable characteristics, all of which are best avoided.

[B] When all variables correspond to functions of time, values at individual points in time have no physical significance. Only values aggregated over an interval of time make sense physically, because all physical law is defined using DEs and their integrals — even impulsive physics, such as the description of collisions etc., is consistent with this. For values aggregated over an interval of time to be well defined, the functions of time in question have to be sufficiently well behaved. They have to satisfy some appropriate regularity property, e.g. they can be continuous (from one side, or the other). Strictly speaking, the property of being integrable would actually be sufficient.

[C] In dealing with CPS systems we must take into account the consequences of using differential equations. We have already broached this topic in our discussion of the semantics of our language Pr1 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 these, as already suggested in Section 2 is to impose strict syntactic restrictions on the form of the permitted DEs, e.g. by insisting that they are linear.

[D] Physics is relentlessly *eager*, in that physical law is always valid and must be obeyed at all moments of time. 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. This is made possible by the fact that aside from moments of state change, in discrete system formalisms, the state remains constant until the next state change, so the next state change can usually be delayed (in real world time) without harm. As soon as physical behaviour enters the scene though, this choice and the range of possibilities it offers disappears. If one physical behaviour stops (within a given physical system), another must take over immediately (in the same physical system), as the universe does not 'go on hold' until some new favourable state of affairs arises. A formalism that is adequate for the description of systems with a physical component must take the eagerness of physics into account.

[E] Point **[D]** 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, such languages might not be fit for purpose physically. Thus, even though they may be perfectly consistent mathematically, unless they take due consideration of the requirements of the physical world, they become irrelevant for the description of real world systems. One simple consequence of this is that for any putative parallel construct in such a language, both branches must terminate together for the overall parallel construct to terminate in a well defined way — it is inconceivable that one part of the universe simply stops and 'nothing happens there' until some other part of the universe catches up.

[F] Points **[D]** and **[E]** boil down to a requirement that descriptions of physical behaviour must be guaranteed to be *total* as a function over the time period which is of interest in the system model. Languages intended for CPS and critical systems should not permit gaps in time during which the behaviour of some physical component is undefined.

Decl ::= [x : T [= x_0] ;]*

Db ::= x := e | {xs := es} | @b | #r

Pr0 ::= Db | {Decl ; Pr0} | Pr0 ; Pr0
| **if** b **then** Pr0 **else** Pr0 **fi** | **while** b **do** Pr0 **od** | Pr0 || Pr0

CbE ::= [iv] $\mathcal{D}\mathbf{x} = \mathsf{F}(\mathbf{x}, \mathbf{y}, \tau)$ **until** g | **obey** Rstr **until** g

Pr2 ::= CbE | Pr2 ; Pr2
| **if** b **then** Pr2 **else** Pr2 **fi** | **while** b **do** Pr2 **od** | Pr2 || Pr2

PrSys ::= Name | [Name =] {Decl ; PrSys} | Pr0 | Pr2 | PrSys || PrSys

FIGURE 4 The CDPPP syntax.

[G] 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. In the present context, it is not so much 'programming errors' that we are referring to when we say bugs, but semantic consequences of the chosen way of interworking, that while being mathematically sound enough, are unacceptable on physical grounds, for instance grounds such as the ones we have been discussing above.

[H] 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.

[I] 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. This is seldom an issue *per se* for practical languages, which must include facilities for division, hence for runtime occurrences of division by zero. Nevertheless, it is 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, CDPPP

In this section, we redesign the syntax of our earlier language, and we name the redesigned language CDPPP (**C**ombined **D**iscrete and **P**hysical **P**rograms in **P**arallel).

5.1 | The CDPPP Language

The CDPPP syntax is given in Fig. 4. Note that in contrast to the earlier language, the occurrences of declarations in program contexts now appear within braces (clauses Pr0 and PrSys). This is to make the scopes of the declarations unambiguous, which makes life easier when we consider the formal semantics in Section 7.

In the CDPPP grammar, the healthiness considerations of Section 4 that can be addressed via the syntax have been incorporated in the structure of the language. 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 from the earlier Pr1.

Specifically, there are now no facilities for Pr2 processes to wait, avoiding the loophole of designing descriptions of physical behaviour that just stop dead at some point. Furthermore, physical behaviour can only be combined with discrete processes at top level, precluding the sudden appearance of physical behaviour part way through a system run. This also means that physical behaviour must be declared at top level, which is also reflected in the design of the PrSys syntax.

Note the additional **obey** Rstr clause for physical behaviour, where Rstr is a predicate in the variables (and constants, etc.). It simply demands that the (physical) variables engage in any behaviour that satisfies Rstr, until such a time as the preemption guard g becomes true. This permits relatively loosely defined behaviour to be specified in cases where more prescriptive behaviour via a DE is not desired, or is impossible due to lack of knowledge, or for some other reason. This replaces the use of waiting clauses in the earlier grammar. Note that DE behaviour and **obey** behaviour are the *only* permitted ways of describing continuous behaviour at the bottom level. Any physical process must be executing a DE behaviour or an **obey** behaviour at all moments of an execution.

Although, through the redesign of the earlier Pr1 clauses to the Pr2 clauses of CDPPP, 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 **obey** cases when one or other of their syntactic components fails. Taking the DE case first, if iv does not evaluate to **true**,[4] then the whole top level PrSys process must **abort**, that is to say, execution terminates abruptly in a failing state. Note that this contrasts sharply with the behaviour described in Section 2 for the iv = **false** case, where we permitted the DE construct to wait. If F fails to satisfy the conditions for existence of a DE solution, then the top level PrSys process **abort**s. 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 **abort**s. Turning to the **obey** case, if Rstr does not evaluate to **true** in a left closed right open time interval starting from the moment the **obey** construct is encountered (or amounts to **skip** at that moment), then the 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 **abort**s.

5.2 | Healthiness Issues Revisited

Taking on board the discussion in Section 4 and having defined the syntax of the improved language CDPPP, we can check how it addresses the healthiness conditions described earlier.

Regarding **[A]**, there is nothing in the syntactic definition of CDPPP that prevents the interpretation of discrete variables being made in terms of piecewise constant functions of time, along with the interpretation of physical variables being piecewise continuous functions of time. So our syntactic definition does not impede what was considered desirable in Section 4 concerning these points.

Regarding **[B]**, the absence of observable effects that depend on an isolated value of a function of time, this is also implicit in the structure of CDPPP, but there are aspects of this issue that are more subtle, that thus bear closer examination. Concerning this, we indicated at the end of Section 2 that the semantic universe of our language(s) is based on functions that are piecewise continuous (or constant) on left-closed right-open intervals. Every point in the domain of such a function has a neighbourhood extending to the right (even if not necessarily extending to the left) on which the function is continuous. Thus there are no functions in the semantics which exhibit isolated values. This is consistent with the fact that CDPPP has updates that take place instantaneously at moments of time which are isolated from one another. Thus it is impossible to have an update to a variable at a certain moment of time, and immediately follow it with another update at 'the next moment of time' since: (a) the interpretation of updates given in Section 2 forbids it, and (b) there is no such thing as 'the next moment of time' since time is dense and the expressions we are able to write down can only refer to isolated time values anyway. All this is made clear in the formal semantics of CDPPP in Section 7.

Regarding **[C]**, concerning DEs, we already made an extensive detour around the issues that arise when we consider DEs in Section 2, and made further comments in Section 4. We consider these to be sufficient.

Regarding **[D]**, concerned with ensuring eagerness in the description of physical behaviour, we have designed the syntax of CDPPP to prohibit explicit lazy behaviour by disallowing the explicit wait-for-condition clauses that were present in Pr1.

This same justification extends to point **[E]**, which is concerned with disallowing descriptions of physical behaviour which are unphysical. The fact that DE behaviour and **obey** behaviour are the only permitted ways of describing physical behaviour at the bottom level, eliminates the temptation to mix inappropriate constructs into the physical description.

Additionally, completion of the execution of one physical behaviour needs to be followed immediately by the execution of the next physical behaviour, guaranteeing the totality over time of the description of physical behaviour that point **[F]** highlights as important, *provided* the behaviour described by the syntax is well defined semantically. The details of this aspect are somewhat implicit in the discussion above, and will emerge more directly in the formal semantics of CDPPP in Section 7.

Point **[G]** recommends a well thought out framework for the interworking of the discrete and physical parts of a system. The design of CDPPP contains separately designed syntactic subsystems and integrates them into a consistent whole, thus adequately satisfying this requirement. Point **[H]** stresses the inescapable semantic issues that must be faced in the design of such a language system. These will be addressed in detail in Section 7.

[4] That is to say, it evaluates to **false**, or fails to evaluate to a well defined value at all.

Point **[I]** indicates the necessity of having runtime aborts in the semantics. Although we remarked earlier that constructing languages in which it is possible to claim statically that runtime errors can never occur *a priori* is possible, such languages tend not to meet the requirements of practical systems, so are of little general interest. This is made the more so by the presence of physical behaviour, whose eagerness can potentially make such errors inevitable from innocent looking precursor system configurations. Again, this is a point to be explored in detail in the formal semantics of Section 7.

5.3 | Semantics and Verification

The heavy dependence on semantics of the preceding discussion raises the question of how we can be sure that any system that is written down defines a sensible behaviour — since *in extremis*, it is only determinable at 'runtime' whether this is the case or not. In purely discrete languages, there is a well trodden route from the syntactic structure of a system description, through semantic considerations, to verification condition schemas that are sound with respect to the semantics and that assure correct behaviour if satisfied for a given system model, to the instantiations of the verification condition schemas using the data of a given system model, which when then proved to hold, 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 then 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 CDPPP language (it has, after all, been constructed specifically for illustrative purposes), we can give an indication of how a couple of them would go. More such conditions can be inferred from the formal semantics in Section 7.

To take one example: if the flow of control reaches an DE construct $[\mathsf{iv}]\mathcal{D}\mathbf{x} = \mathsf{F}(\mathbf{x},\mathbf{y},\tau)\ \mathbf{until}\ \mathsf{g}$ 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. It is easy enough to generate a verification condition to enforce this, for example:

$$HYP \vdash \mathsf{iv}_1 \vee \mathsf{iv}_2 \vee \ldots \vee \mathsf{iv}_n$$

where $\mathsf{iv}_1, \mathsf{iv}_2, \ldots, \mathsf{iv}_n$ are the iv's of the various cases of the DE case analysis, and HYP includes information about values of variables, etc., that can be deduced statically from the context of the case analysis (and whose strength would depend on the incisiveness of the static analysis of the program).

For another example: once a DE construct has been preempted by its preemption guard g 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. Again, verification conditions to enforce this are easy to define, for example:

$$HYP \vdash \mathsf{g} \Rightarrow \mathsf{iv}$$

$$HYP \vdash \mathsf{g} \Rightarrow \mathsf{Rstr}$$

where, in the first case, iv is the entry condition to a sequentially following DE construct, and in the second case, Rstr is the condition of a sequentially following $\mathbf{obey}$ construct; and HYP is as previously. Some additional detail would be needed to preclude the use of such verification conditions at points where the execution of the program was expected to terminate.

For an easy last example, in the discrete part of the language, the success of an $\mathbf{if}$ statement can be assured, provided there is an $\mathbf{else}$ clause to capture exceptional cases. This is straightforward to enforce.

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 correctly denote (e.g. *in extremis* by not having division, or similar 'risky' constructs, in the language). Such languages *may* help in the verified design of critical systems, depending on the application.

```
1   ACcontroller =
2     while true
3     do @(runAC = true) ;
4       while runAC = true ∧ θR > θS
5       do @(tempUp = true ∨ tempDown = true ∨ runAC = false) ;
6         if tempUp = true
7         then {tempUp, θS := false, min(θS + 1, SH)}
8         elsif tempDown = true
9         then {tempDown, θS := false, max(θS − 1, SL)}
10        fi
11      od
12    od
13  ACfluid =
14    while true
15    do obey θR ∈ [RL . . . RH] until runAC = true ;
16      if θR ≥ θS + 1
17      then [ θR ∈ [RL . . . RH] ] D θR = −KR(θR − θRL)
18           until (θR = θS ∨ runAC = false)
19      else obey θR ∈ [RL . . . RH] until θR ≥ θS + 1 ∨ runAC = false
20      fi
21    od
22  ACsystem =
23    runAC : B = false ; tempUp : B = false ; tempDown : B = false ;
24    ( User || ( θS : N ∩ [SL . . . SH] = θS0 ; θR : R ∩ [RL . . . RH] = θR0 ;
25              ACcontroller || ACfluid ) )
```

FIGURE 5 The redesigned $ACcontroller$ and $ACfluid$ definitions, and the redesigned $ACsystem$.

6 | THE AIR CONDITIONING SYSTEM IN CDPPP

In the light of the preceding discussions, we return to our air conditioning running example and restructure it for the CDPPP 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 CDPPP. 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 $\mathcal{D}\,\theta_{\mathsf{R}} = -\mathsf{K}_{\mathsf{R}}(\theta_{\mathsf{R}} - \theta_{\mathsf{RL}})$, 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 $\mathbf{obey}$ 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. Putting all the three components together gives us the complete revised system, shown in Fig. 5.

7 | FORMAL SEMANTICS OF CDPPP

Although our presentation of CDPPP has been relatively informal thus far, in this section we show that we can make it quite precise by giving a formal operational semantics for it. We keep certain aspects relatively straightforward by, for instance, insisting that a runtime error aborts the whole semantic construction, as well as other simplifications. On the other hand, by allowing scopes to be defined fairly arbitrarily (by comparison with some other formalisms), allowing very general coupling between variable, we see that a certain degree of unavoidable complexity arises too.

7.1 | Initial Considerations

We start by listing some simplifying assumptions about the syntactic form of the CDPPP programs whose semantics we give. Thus, let $\mathcal{P}$ be a CDPPP program.

1. Every occurrence of a variable declaration in $\mathcal{P}$ declares a different variable.
2. There are no occurrences in $\mathcal{P}$ of undeclared variables, or of variables outside the scope of their declaration (the scope being defined via the declaration's immediately enclosing braces).
3. There are no program Name occurrences in $\mathcal{P}$.

Programs satisfying these conditions are called **rectified**. An arbitrary program can be manipulated relatively easily into rectified form, provided it contains no undeclared, multiply declared, or circularly declared Names, or variables that are undeclared or multiply declared, or variables that are used outside their scope (and we use the usual scope nesting rules to disambiguate variable occurrences in nested scopes). From now on we assume that all CDPPP programs are rectified, without further comment. We make some comments on non-rectifiable CDPPP programs at the end of this section.

A syntax for rectified CDPPP programs, simplified from Fig. 4, appears in Fig. 6. It contains labels for various node types in the syntax tree, shown in a smaller font (and in red), e.g. [PrSys]. This is for use in the semantics. More details appear below.

The starting point of the semantics is the definition of (the semantics of) time, which is the non-negative semi-infinite interval of the reals: $\mathbb{T} \equiv [0 \ldots \infty) \subseteq \mathbb{R}$. Moments in time will be referred to using the variable t.

Suppose a CDPPP program $\mathcal{P}$ has a set of variables[5] $\mathcal{V}$, with each $v \in \mathcal{V}$ taking values in a set $\mathcal{S}_v$, where each $\mathcal{S}_v$ contains an unassigned value $\mathbf{unass}_v \in \mathcal{S}_v$. The semantics of $\mathcal{P}$ is given by a *system trace*, which can be seen as a higher order map:

$$\mathrm{Sem}(\mathcal{P}) : \{\bot_{\mathcal{P}}\} \uplus \mathcal{V} \to \mathbb{T} \to \biguplus_{v \in \mathcal{V}} \mathcal{S}_v$$

where the following restriction holds:

$$\mathrm{Sem}(\mathcal{P}) = \bot_{\mathcal{P}} \ \oplus \ (\forall v \in \mathcal{V}, t \in \mathbb{T} \bullet \mathrm{Sem}(\mathcal{P})(v)(t) \in \mathcal{S}_v)$$

So, either $\mathcal{P}$ $\mathbf{aborts}$ ($\mathrm{Sem}(\mathcal{P}) = \bot_{\mathcal{P}}$), or, at all times, each variable takes a value in its appropriate semantic space.

On the face of it, there are at least two ways of constructing the semantics of a program $\mathcal{P}$. We can follow the traditional syntax-directed route, or we can follow an iterative strategy. In the syntax-directed approach, to every syntactic clause of Fig. 4, e.g. CbE ::= [iv] $\mathcal{D}\mathbf{x} = \mathsf{F}(\mathbf{x}, \mathbf{y}, \tau)$ **until** g, we would have one or more rules such as:

$$\frac{\begin{array}{c}\mathrm{dom}\ \mathrm{Sem}(\mathcal{P})(\mathbf{x}) = [0 \ldots t_A] \quad , \quad \mathsf{P} : \text{[DE]} \quad , \quad \mathsf{iv}(\mathbf{v}(t_A)) \quad , \\ \mathbf{f} : [t_A \ldots t_B] \to \mathcal{S}_{\mathbf{v}} \quad , \quad \mathbf{f}(t_A) = \mathbf{x}(t_A) \quad , \quad \mathbf{f} \models \mathcal{D}\mathbf{x} = \mathsf{F}(\mathbf{x}, \mathbf{y}, \tau) \quad , \\ (\forall t_A \leq t < t_B \bullet \neg \mathsf{g}(\mathbf{v}[\mathbf{f}\backslash\mathbf{x}](t))) \quad , \quad \mathsf{g}(\mathbf{v}[\mathbf{f}\backslash\mathbf{x}](t_B))\end{array}}{\mathrm{Sem}(\mathcal{P})(\mathbf{x})|_{[0 \ldots t_B]} = \mathrm{Sem}(\mathcal{P})(\mathbf{x})|_{[0 \ldots t_A]} \cup \mathbf{f}}$$

The above is an example of a typical 'big step' semantic rule, which aligns with our aim to develop the semantics as a set of functions from time to the semantic domains of variables. What this rule says in symbols is:

> **IF** we assume that: the semantics of **x** is defined as far as $t = t_A$; and process P is at the entry point of the relevant DE clause of the syntax tree of $\mathcal{P}$; and the entry conditions to the DE clause, iv($\mathbf{v}(t_A)$), hold; and the vector function **f** is defined over an interval $[t_A \ldots t_B]$ with range in the semantic space of **x**, i.e. $\mathcal{S}_{\mathbf{v}}$; and at $t = t_A$, **f** and **x** agree; and **f** solves the DE; and from $t = t_A$ up to, but not including $t = t_B$, g, with **f** substituted for **x** in **v** as needed, does not hold; and at $t = t_B$, g, with **f** substituted for **x** in **v**, holds; **THEN f** provides a suitable extension to the semantics of **x** on $[t_A \ldots t_B]$.

[5] Although rectified CDPPP has loops, it has no recursion, so that the set of variables can be read off from the declarations contained in $\mathcal{P}$. Similarly for processes. The maximum nesting of compound constructs determines the number of concurrent processes.

[Decl]Decl ::= [x : T [[Ini] = x_0] ;]*

[Db]Db ::= [As]x := e | [AsP]{xs := es} | [Wb]@b | [Wr]#r

[Pr0]Pr0 ::= Db | [DPr0]{Decl ; Pr0} | [Pr0Sq]Pr0 ; Pr0
| [Pr0I]**if** b **then** Pr0 **else** Pr0 **fi** | [Pr0Wh]**while** b **do** Pr0 **od** | [Pr0Par]Pr0 || Pr0

[CbE]CbE ::= [DE][iv] $\mathcal{D}\mathbf{x} = \mathsf{F}(\mathbf{x},\mathbf{y},\tau)$ **until** g | [OB]**obey** Rstr **until** g

[Pr2]Pr2 ::= CbE | [Pr2Sq]Pr2 ; Pr2
| [Pr2I]**if** b **then** Pr2 **else** Pr2 **fi** | [Pr2Wh]**while** b **do** Pr2 **od** | [Pr2Par]Pr2 || Pr2

[PrSys]PrSys ::= [DPrSys]{Decl ; PrSys} | Pr0 | Pr2 | [PrSysPar]PrSys || PrSys

FIGURE 6 The decorated syntax for rectified CDPPP.

Quite aside from the fact of how difficult it is to conveniently write such a rule in the conventional $\frac{\text{ANTECEDENTS}}{\text{CONSEQUENT}}$ format, what is interesting about this approach is, less what it says, so much as what it fails to say. Among such omissions we can mention the following.

- In common with almost all uses of this kind of rule based semantic definition, the rule implicitly assumes that facts about **x** can be deduced independently from other aspects of the semantics of $\mathcal{P}$, thus building the semantics in a 'divide and conquer' manner. Many of the points that follow demonstrate that this is not appropriate in a hybrid/cyberphysical context.
- While the evolution of **x** in the time interval following $\mathsf{t} = \mathsf{t}_\mathsf{A}$ generally depends on **x**, it may often depend on other variables e.g. **y** too, whose value may be defined in another process, as seen in $\mathcal{D}\mathbf{x} = \mathsf{F}(\mathbf{x},\mathbf{y},\tau)$. Not only this, but the evolution of **y** may itself depend on the value of **x**, etc. Thus the syntactic structure may disregard the true semantic dependencies of $\mathcal{P}$.
- The value of the RHS of the DE $\mathsf{F}(\mathbf{x},\mathbf{y},\tau)$ may be affected by discontinuities arising indirectly (or even directly, in a slightly altered language syntax) from activities in processes running concurrently. The syntax directed approach does not help to deal with such issues.
- The evolving values of the variables in the time interval following $\mathsf{t} = \mathsf{t}_\mathsf{A}$ affects the value of g, which is needed to determine the value of t_B, needed for the duration of the interval itself. Again, the syntax directed approach does not help to deal with such issues.

The remarks above confirm that that syntactic divide and conquer is unhelpful in determining the semantics of $\mathcal{P}$, which emerges in a manner which relies much more on interdependencies between concerns that cut across the syntactic structure. Moreover, many similar considerations arise when we consider other rules generated from the syntax. Accordingly, we adopt a much more imperative operational approach to the semantics.

7.2 | Operational Semantics

Broadly speaking, the absence of 'program names' and of recursion in a rectified CDPPP program $\mathcal{P}$, implies, as noted already in footnote 5, that there is no dynamic process creation, and no dynamic variable creation. This brings a welcome degree of structural simplification since we can navigate through the static syntax tree of $\mathcal{P}$ to locate the next construct to execute in each process. Once we have found it, we can then execute it, which is the approach we take to the semantics.

We base the semantics of a rectified CDPPP program $\mathcal{P}$ on the following assumptions, definitions, and other details, listed below. Much of the impact of these definitions can be summarised in the state transition diagram of Fig. 7, which outlines the process of navigating the syntax tree of $\mathcal{P}$ in order to locate the required leaf nodes, in which the job of extending the $\mathrm{Sem}(\mathcal{P})$-so-far is actually done, and which is described in detail later.

- We have available the syntax tree $\mathcal{T}$ of $\mathcal{P}$. Its nodes are labelled using the labels shown in Fig. 6.
- A **compound** node of $\mathcal{T}$ is a node labelled by one of: [Db], [Pr0], [DPr0], [Pr0Sq], [Pr0I], [Pr0Wh], [CbE], [Pr0Par], [Pr2], [Pr2Sq], [Pr2I], [Pr2Wh], [Pr2Par], [PrSys], [DPrSys], [PrSysPar].
- A **basic** node of $\mathcal{T}$ is a node labelled by one of: [As], [AsP], [Wb], [Wr], [DE], [OB]. Clearly, basic nodes occur at the leaves of $\mathcal{T}$.
- A **process** is (identified with) a node which is either the root node of $\mathcal{T}$, or is a child node of a node labelled by one of: [PrSysPar], [Pr0Par], [Pr2Par]. The name of a process node is the path π in $\mathcal{T}$ to the node. N.B. Processes conceptually correspond with the agents that execute $\mathcal{P}$, in the evident way. The process named by the empty path $\langle\rangle$ is the root process (corresponding to the root node of $\mathcal{T}$). Each node of $\mathcal{T}$ is executed by the process corresponding to its nearest process node ancestor in $\mathcal{T}$.

- At runtime (i.e. during the execution of $\mathcal{P}$) each process has a **state**, which is one of: **suspended**, **activated**, **leaf**, **terminated**. N.B. The former two states are concerned with executing **compound** nodes, i.e. navigating through $\mathcal{T}$, while **leaf** is concerned with executing **basic** nodes. The last state speaks for itself.
- Variables are divided into **discrete** variables $\mathsf{V_D}$ and **physical** variables $\mathsf{V_P}$: $\mathcal{V} = \mathsf{V_D} \uplus \mathsf{V_P}$.
- **initialassign unass**$_\mathsf{v}$ to v means: make an initial definition of $\mathrm{Sem}(\mathcal{P})(\mathsf{v})(\mathsf{t})$ to **unass**$_\mathsf{v}$ for all $\mathsf{t} \in \mathbb{T}$.
- **abort** means: abandon the construction of the semantics and set $\mathrm{Sem}(\mathcal{P}) = \bot_{\mathcal{P}}$.
- **evaluate** d means: attempt to evaluate d. If d does not evaluate, **abort**. Otherwise obtain $\mathrm{val}(\mathsf{d})$.
- **assign** d to z means: **evaluate** d; assign $\mathrm{val}(\mathsf{d})$ to program variable z. Time may or may not occur as a parameter in an assignment. If it does, in order to be non-aborting, the assignment must be non-aborting for all the relevant values of time.
- The construction of $\mathrm{Sem}(\mathcal{P})$ requires a further data structure, the task **pool**. Its entries are triples [*proc*, *task*, *stage*] where: *proc* is the name of a process: *task* corresponds to the syntactic content of a basic node, i.e. the command to be executed; *stage* identifies the current position in the execution of the *task* (the *stage*s are detailed below). In speaking generically, we refer to *task* entries using their node types, i.e. one of: [As], [AsP], [Wb], [Wr], [DE], [OB].
- $\mathrm{Sem}(\mathcal{P})$ is constructed step by step, extending the functions of time in $\mathrm{Sem}(\mathcal{P})$ in stages. Parts of such extensions may be contingent, for technical convenience, and may get overridden subsequently. The parts of the functions in $\mathrm{Sem}(\mathcal{P})$ that are definitive are the intial segments with domain up to, but not including, the time mentioned in the most recently executed **commit** command.
- A **commit** e command is, in effect, equivalent to **assign** e to t, where t is the time variable. Thus, updates to t correspond to successive definitions of the interval of time within which the semantics has been made definitive.
- $\mathrm{Sem}(\mathcal{P})$ is constructed step by step, the steps being guided by the structure of $\mathcal{T}$. There are **progress** steps, **work** steps, and **return** steps.
- **progress** steps are steps executed by processes in the **activated** state, and in these, the locus of control descends deeper into $\mathcal{T}$, or moves to sibling child nodes of appropriate compound nodes, when attempting to locate the next basic node to execute. When **progress**ing from the following compound node types, the associated actions are performed:
  - [PrSys], [Pr0], [Pr2], [Db], [CbE]: descend to child;
  - [DPrSys], [DPr0]: **assign** any [Ini]tialisations in the Declarations; proceed to subprocess;
  - [PrSysPar], [Pr0Par], [Pr2Par]: set state of current process to **suspended**; set state of child processes to **activated**, fork locus of control, and descend to child processes;
  - [Pr0Sq], [Pr2Sq]: descend to first child; upon return, **progress** once more, descending to second child;
  - [Pr0**I**f], [Pr2**I**f]: **evaluate** condition b; if true, descend to first child; otherwise descend to second child;
  - [Pr0Wh], [Pr2Wh]: **evaluate** condition b; if false then **return**; otherwise descend to child.
- When a **progress** step results in a process arriving at a basic node, the process state becomes **leaf**, and [π, [ty]:*cmd*, **st**] is added to the task **pool**, where π is the name of the current process, [ty]:*cmd* is the command to be executed (i.e. the content of the basic node, together with its type [ty]), and **st** denotes the start stage.
- At a basic node, $\mathrm{Sem}(\mathcal{P})$ is constructed step by step, using **work** steps, described below. These use the task **pool**.
- Upon completion of the execution of a basic node, the relevant task **pool** entry becomes [π, *cmd*, **return**], and the process state becomes **activated** once more.
- **return** steps are steps executed by processes in the **activated** state, and in these the process ascends higher into $\mathcal{T}$, when attempting to locate the next basic node to execute. When **return**ing from the following node types, the **pool** entry causing the **return** is removed and the associated actions are performed:

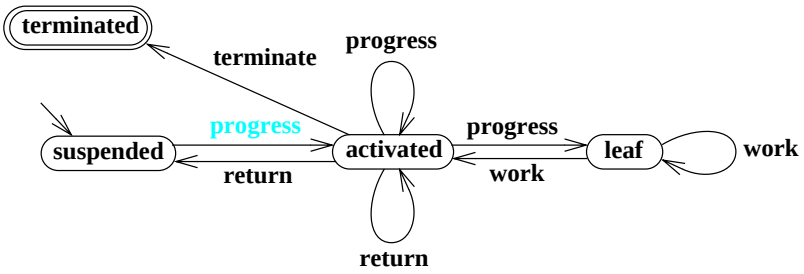

FIGURE 7 The transition diagram describing the construction of the semantics for a process in a rectified CDPPP program. The activity of extending the system trace takes place in the **leaf** state during **work** steps. The **progress** step from **suspended** to **activated** is shown faint since it is an action in a different process that causes the transition (in the process of interest).

- [PrSys]: if there is no parent then the (root) process state becomes **terminated** (and the semantics is completed); otherwise ascend to parent;
- [Pr0], [Pr2], [Db], [CbE], [DPrSys], [DPr0]: ascend to parent;
- [PrSysPar], [Pr0Par], [Pr2Par]: unless the child processes have completed at exactly the same moment in time then **abort**; otherwise, set state of child processes to **suspended**; set state of parent process to **activated**; merge locus of control, and ascend to parent;
- [Pr0Sq], [Pr2Sq], [Pr0f], [Pr2f]: ascend to parent;
- [Pr0Wh], [Pr2Wh]: **evaluate** condition b; if true then **progress**; otherwise ascend to parent.

With the above structural considerations in place, we turn to the construction of the semantics of a rectified CDPPP program $\mathcal{P}$, which now becomes more focused on the execution of the basic constructs, and the actual changes of state which they implement. The operational semantics is given by the pseudocode below.[6] This combines the **progress** and **return** steps that navigate the syntax tree $\mathcal{T}$, with the **work** steps needed to build $\mathrm{Sem}(\mathcal{P})$.

Fig. 8 shows the lower level transition diagrams that break up the activity of constructing $\mathrm{Sem}(\mathcal{P})$ during **work** steps into the smaller steps needed. This breakup is needed because of the fact that we deal with concurrent steps, some of which have a non-zero and statically unknown duration, all of which additionally take place over real time. Finegrained scheduling is therefore required to construct the semantics in a way that conforms to all the real world requirements and healthiness considerations we set out earlier.

The various cases in Fig. 8 can be outlined as follows. Case [OB] progresses a behaviour conforming to Rstr. Since this occupies a nonempty interval of time, it is regularly interrupted by the instantaneous events that may update the state discontinuously. The [DE] case is similar, but needs an instantaneous step to check [iv] before proceeding to progressing the solution of the DE system. The [Wr] case starts by calculating the waiting interval. If it is non-positive the construct can complete immediately. Otherwise the process waits, until the required elapsed time has passed. The [Wb] case is similar, but no calculation as such is needed. The most complicated cases are the [As] and the [AsP] cases, because we decided earlier that they would execute in isolation from other events, necessitating the introduction of suitable waiting intervals. So they first must fix on a non-clashing waiting interval; then, when that expires, the assignment(s) can be done. Then, another waiting interval must be found, after which the task can complete.

In outline, the pseudocode does the following. After initialisation, the outer main loop performs all available **progress** tasks, and then enters the main inner loop. On entry to this, the main inner loop attends to a collection of tasks that take no time, on the assumption that the scheduler is executing the pseudocode at a suitably selected moment. These tasks include: **abort**ing [iv]-failing DE constructs; dealing with expired assignment construct waits; dealing with wait-on-condition constructs with true conditions; dealing with wait-for-expression constructs with nonpositive or expired wait intervals. If any of these cases arise, they are dealt with, and the main execution loop is restarted in case there are more cases to deal with, or there are consequences that affect the subsequent evolution. If not, the main loop proceeds to evolving the functions of time in $\mathrm{Sem}(\mathcal{P})$.

Evolving the functions of time is a global task, which must take all the active DE and **obey** constructs and must solve them simultaneously. Obviously, this is one point in the semantics at which the problem of being able to calculate what is required must remain open, because of the very limited decidability properties of such systems. A consequence of this is that although we have no generic technique for constructing such a global solution for an arbitrary CDPPP program,[7] it is nevertheless the case that the mathematical framework within which we have cast the

[6] In the pseudocode, we use ;; as the meta-level sequential separator. Other constructs are self-explanatory.

[7] Of course, the absence of a completely generic technique does not preclude the ability to use the special cases familiar from applied mathematics if the problem form fits the appropriate syntactic pattern.

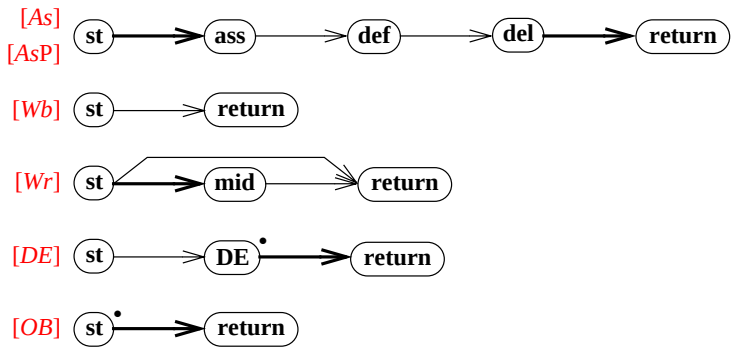

FIGURE 8 The lower level transition diagrams for the **work** steps that implement the activity of extending the system trace, for the various kinds of **leaf** nodes of $\mathcal{T}$. Thicker arrows denote transitions that require the passage of time to complete, whereas thin arrows denote transitions that can complete immediately when their guard condition holds. The blobs that decorate two states denote self-loops, indicating that the transitions emerging from them may be subdivided into a number of substeps.

semantics guarantees the semantics *exists*, i.e., either there is a solution (even if we may not know what it is), or there is no solution, in which case the attempt to **assign** it will cause an **abort**.

Assuming though, that a solution has been arrived at, the remaining job for the main loop is to determine the next interruption point. Again, this is an element of the semantics at which the ability to calculate what is required must remain open, for the same reasons as before. But, again assuming that the required information has been obtained somehow, the knowledge of the soonest needed interruption point allows the times of any pending assignment to be chosen strictly earlier, thus ensuring it doesn't clash with any other event. Taking all this into account, the new interruption point is finally determined, and the interval up to the new interruption point can be committed. This done, the main inner loop restarts. When there are no further **work** steps to perform, the main inner loop exits, and any available **return** tasks are executed. Then the outer main loop restarts.

initialisation
  assign 0 to t ;; **commit** t ;;
  forall v ∈ $\mathcal{V}$ do **initialassign unass**$_v$ to v od ;;
  forall processes P do **assign suspended** to **state**(P) od ;;
  assign activated to **state**(⟨⟩) ;;
  assign ∅ to **pool** ;;
end ;;
while **state**(⟨⟩) ≠ **terminated**
do
  while there exists an **activated** process π with an available **progress** step
  do
    choose such a π esooch ;;
    progress π /* as described above */
  od ;;
  /* all non-**suspended** processes are in state **leaf** */ ;;
  while there exists a **leaf** process with an available **work** step
      (i.e. **pool** contains entries whose *stage* is not **return**)
  do
    assign false to **escNow** ;;
    forall entries [π, [DE]:[iv] $\mathcal{D}$**x** = F(**x**, **y**, τ) **until** g, **st**] in **pool**
    do
      if (**evaluate** iv ≠ **true**)
      then
        abort

```
    else
      replace entry [π, [DE]:[iv] 𝒟 x = F(x, y, τ) until g, st] in pool with
        [π, [DE]:𝒟 x = F(x, y, τ) until g, DE]
    fi
  od ;;
  if (there is an entry [π, [As]:x := e, del:t^rr] or an entry [π, [AsP]:{xs := es}, del:t^rr] in pool
    and (t = t^rr))
  then   /* there will be exactly one such entry; '(s)' etc. as needed */
    replace entry [π, [As(P)]:{x(s) := e(s)}, del:t^rr] in pool with
      [π, [As(P)]:{x(s) := e(s)}, return] ;;
    assign true to escNow
  fi ;;
  if (there is an entry [π, [As]:x := e, ass:t^rr] or an entry [π, [AsP]:{xs := es}, ass:t^rr] in pool
    and (t = t^rr))
  then   /* there will be exactly one such entry; '(s)' etc. as needed */
    forall times u ∈ [t ... ∞) do assign e(s)(t) to x(s)(u) od ;;
    replace entry [π, [As(P)]:{x(s) := e(s)}, ass:t^rr] in pool with
      [π, [As(P)]:{x(s) := e(s)}, def] ;;
    assign true to escNow
  fi ;;
  forall entries [π, [Wb]:@b, st] in pool
  do
    if (evaluate b = true)
    then
      replace entry [π, [Wb]:@b, st] in pool with [π, [Wb]:@b, return] ;;
      assign true to escNow
    fi ;;
  od ;;
  forall entries [π, [Wr]:#r, st] in pool
  do
    assign (r + t) to t^rr ;;
    if (t ≥ t^rr)
    then
      replace entry [π, [Wr]:#r, st] in pool with [π, [Wr]:#r, return] ;;
      assign true to escNow
    else
      replace entry [π, [Wr]:#r, st] in pool with [π, [Wr]:#r, mid:t^rr]
    fi ;;
  od ;;
  forall entries [π, [Wr]:#r, mid:t^rr] in pool
  do
    if (t = t^rr)
    then
      replace entry [π, [Wr]:#r, mid:t^rr] in pool with [π, [Wr]:#r, return] ;;
      assign true to escNow
    fi ;;
  od ;;
  if (escNow = true) then commit t ;; break-to ▶ fi ;;
  forall entries
    [π, [DE]:𝒟 x = F(x, y, τ) until g, DE] or
    [π, [OB]:obey Rstr until g, st] in pool
```

```
    do
      choose a maximal t_max and functions x̃(u), ỹ(u) ... for u ∈ [t ... t_max)
      such that
        x̃(u), ỹ(u) ... solves all the 𝒟x = F(x, y, τ) and Rstr constraints for u ∈ [t ... t_max)
      esooch ;;
      forall times u ∈ [t ... t_max) do assign x̃(u), ỹ(u) ... to x(u), y(u) ... od ;;
    od ;;
    forall entries
      [π, [Wb]:@b, st] or
      [π, [Wr]:#r, mid:t^rr] or
      [π, [DE]:𝒟x = F(x, y, τ) until g_DE, DE] or
      [π, [OB]:obey Rstr until g_OB, st] or
      in pool
    do
      assign the minimum value (or, if necessary, the infimum value) of time at which
      any b becomes true or
      the time is any t^rr or
      any g_DE becomes true or
      any g_OB becomes true
      to t_min
    od ;;
    if (there is an entry
      [π, [As]:x := e, st] or [π, [AsP]:{xs := es}, st] or
      [π, [As]:x := e, def] or [π, [AsP]:{xs := es}, def]
      in pool)
    then
      choose exactly one such entry esooch ;;
      choose t_ass such that t < t_ass < t_min esooch ;; /* (t_ass − t) is chosen small */
      replace the chosen entry
      [π, [As(P)]:{x(s) := e(s)}, st] or [π, [As(P)]:{x(s) := e(s)}, def] with
      [π, [As(P)]:{x(s) := e(s)}, ass:t_ass] or [π, [As(P)]:{x(s) := e(s)}, del:t_ass] respectively
      in pool
    else
      assign t_min to t_ass
    fi ;;
    commit t_ass ;;
▶ od ;;
  while there exists an activated process π with an available return step
  do
    choose such a π esooch ;;
    if π has an entry [π, cmd, return] in pool then remove the entry from pool fi ;;
    return π /* as described above */
  od ;;
od
```

Unlike the rule based approach to the semantics suggested in Section 7.1, the above definition, although quite intricate, lays bare the true complexities of the scheduling required by having both real time and shared variable concurrent processes in the same formalism. As we see, the pseudocode expresses a real time scheduler at the level of abstraction needed for a sufficiently precise definition. The design of our CDPPP language was deliberately chosen to bring into sharp focus the consequences of having these two features simultaneously present in a fairly unrestricted way.

7.3 | Followup Comments

In this section we make some additional comments regarding issues not fully explored earlier.

Firstly, we return to non-rectified CDPPP programs. If a CDPPP program is non-rectified because it contains Names, the obvious technique to use is to substitute a Name's definition for occurrences of the Name. This works unless there are cyclic dependencies among the Name definitions, in which case the back substitution process will never end. Provided each dependency cycle contains at least one link which starts with something other than a reference to a Name, then recursive runtime stack techniques could handle such complications. The same comments apply to variables occurring in declarations contained in such cycles. The semantic details would add considerable complexity to our already complicated definition, so, because the techniques are by now very well known in operational semantics, we did not incorporate these possibilities in this paper.

Secondly, and somewhat under the bonnet for traditional semantic approaches, is the precise nature of the continuous mathematics entities that figure in a framework such as we have outlined. Arbitrary functions of time do not have properties that can be directly reconciled with the kind of properties we need for calculus and other 'engineering mathematics' purposes. The field of mathematical analysis has been active for a couple of centuries or so, engaged in investigating logically sound formulations of the kinds of property that are needed. Although, done properly, such logically sound formulations would be consistent with the kind of approach typical of model based formal methods, the bad news is that 'done raw', it would require each property to be formulated with so many interleavings of quantifiers that practical use would be rendered impossible. In practice, we would need to restrict quite heavily to 'calculationally feasible' fragments of engineering mathematics in order to build actual systems. Thus, as well as the 'physically based' healthiness conditions we discussed at length in this paper, there lies underneath, a further layer of lower level 'mathematical healthiness' considerations, pertaining to the continuous world, that we have largely ignored. Although it would take us too far outside the scope of this paper to delve in any depth into this material, some hint of the technical details involved is evident in the use of the 'infimum' concept in the pseudocode of the formal semantics, and in the discussion in Appendix B of the closedness (or otherwise) of the sets of states defined by guard constructs of our languages, which potentially intrudes at user level.

We observe thirdly that the semantics of our language permits executions of finite duration, and Zeno phenomena. Executions of finite duration are not excluded and are non-aborting provided all parallel processes come to an end at exactly the same moment in time. Zeno phenomena arise when a countably infinite sequence of execution fragments has an overall finite duration. The semantics takes no precautions to attempt to continue beyond a Zeno point for the sake of expository simplicity.

8 | RELATED APPROACHES

Hybrid systems have been identified as being of high importance for many years now. Some of the earliest work that we can cite includes papers like [9,10,11,12]. This initial effort rapidly led to other work such as [13,14,15,16,17,18,19,20]. Slightly later formulations include [21,22,23,24,25,26,27,28]. The more than decade old survey [29] covers a large number of formulations such as these, and in particular, the tools that support them, such as HyTech [30], d/dt [31], PHaVer [32] and others.

Modern developments along these lines continue apace. The fruits of much of the work that has been done has appeared in the annual *International Conference on Hybrid Systems: Computation and Control*, which acts as the focus of a major annual exposition of relevant work.

One is struck, in the early work indicated, by the typically low expressivity in the continuous sphere of many of the systems discussed. This is motivated, of course, by the desire for decidability of the resulting languages and systems. For decidability reasons, most such formalisms are based on variations of the early hybrid automaton concept [23,10,11]. In fact, for simple time linear behaviours —which are very frequently at the heart of such systems, e.g. $\mathcal{D}\,x = K$, with K constant, as is often found— there is very little difference between using a DE of the form just quoted on the one hand, and on the other hand, using an expression $x' = x + K\Delta T$ where ΔT is the duration of the behaviour. The consequences of the latter can be elicited by simple algebraic calculation, making this approach highly attractive for tool based approaches. Ironically, this self same low level of expressivity helps enormously in avoiding some of the problems we were concerned with in Section 4.

Despite the wealth of knowledge about differential equations and how to solve them [33], the proportion of equations that can be solved analytically is vanishingly small, and the need to tackle practical applications often forces the use of equations for which the only approach is numerical [34,35]. In this realm too, the aim is to reduce the approximate solution of a differential equation to a family of calculations based on the simple linear form $x' = x + K\Delta T$; with suitably chosen K and ΔT for each individual instance, according to the strategy dictated by the numerical scheme being followed. Thus, despite coming from very different starting points, many efficient approaches target the linear goal, and one sees a confluence of computation concerns on the one hand with similar concerns from numerical mathematics.

A major consumer of knowledge about hybrid behaviour is, of course, the cyber-physical systems (CPS) field, e.g. [1,2,36,37,38,39,40,41,42,43,44], as well as the wealth of current work presented at the annual *CPS Week* gatherings in recent years.

In this context we can point to the extensive survey [45], which covers a wide spectrum of research into cyber-physical systems, the tools and techniques used in that domain, as well as the applications that are tackled in the practical sphere. As we might expect, despite the relative

newness of the cyber-physical systems area, formal approaches are somewhat overshadowed by more traditional and simulation based techniques, especially when it comes to practical applications.

This can be traced back to a number of causes. One is the wide breadth of mathematics that is needed to properly integrate the various aspects of cyber-physical systems in a sound way, some of which was the focus of discussion in the preceding sections (and much of which (it has to be said) is unfamiliar to a large part of the computer science community). Another, very cogent one in the context of commercial applications, is the simple pressure to get to market rapidly in what has become a very competitive sector — in such a situation it is much easier, and quicker, to fall back on traditional approaches, rather than to invest in potentially more conceptually challenging ones.

One consequence of the wide variety of approaches seen in this arena is the propensity to combine the different formalisms for describing behaviour in the different contributing disciplines of CPS in a relatively *ad hoc* manner. This leads to the risk of 'bugs between formalisms' that we noted earlier, when multiple formalisms need to be combined.

The fact that it is often not possible to solve a hybrid/cyber-physical system exactly is not the insuperable obstacle it might, at first, seem to be. Often it is sufficient to know that a system will stay in a safe region of the state space indefinitely, without knowing exactly what the system dynamics will be. Terminology differs here. Some authors speak of an 'unsafe region' which is to be avoided. Others speak of a 'safe region' which the system is to be confined to. Other work speaks of an 'invariant' expression concerning the state space which defines the safe region. It is sufficient to regard the safe and unsafe regions as complementary in the set of values of the collection of variables of the system.[8]

When it is sufficient for the system to stay in a target region of the state space, various kinds of 'helper functions' may be employed to gain assurance that the system behaves well.

Variant functions are familiar from the classical discrete programming world [46,47,48]. To help control the behaviour of recursions and unbounded iterations, a variant function (of the state) is required to be decreased by each iteration's state change, the idea being that it is easier to ascertain this than to argue about the iterative behaviour directly. When the variant function takes its values in a well founded set, this gives a guarantee of termination, the 'safe region' aimed for being the states in which the iterative behaviour is not enabled.

Liapunov functions are well known from continuous control theory [49,50,51]. To help establish stability, the flow defined by the dynamics is required to decrease the Liapunov function (of the state), this being easier to ascertain than to argue about the flow itself, since it can be checked directly from the differential equation defining the flow. The Liapunov function has an easily identified minimum, which coincides with a stable fixed point of the dynamics, the state at which this occurs being the 'safe region' aimed for.

Barrier functions have become a familiar technique for establishing safety in the hybrid systems world [52,53,54]. They are required to have one sign (positive say) in the unsafe region, and to have the other sign (negative) in the set of initial states. Provided the barrier function is decreased by the flow defined by the continuous dynamics and is also decreased by each discrete state change, the unsafe region can never be reached. Barrier functions thus combine the basic ideas behind both variant functions and Liapunov functions. Clearly, the rich structure of the hybrid systems paradigm gives rise to many opportunities for fusing ideas from these two precursor worlds.

When a formalism for hybrid or cyber-physical systems is defined precisely enough, rigorous formal reasoning about invariants and barrier functions becomes possible.[9] This includes recent, less restrictive variants of hybrid automata, for example.

Closer to the style of this paper are language systems purposely conceived for proof regarding hybrid systems. One example is Hybrid CSP together with the tools that support it [12,55,56]. Another is the dynamic logic approach of Platzer [57,58], supported by the KeyMaera verification tool [59] which supports the kind of modelling exemplified in this paper. The original formulation of action systems for discrete systems [60] was extended to the hybrid sphere in [61].

Action systems provided much of the inspiration for the Event-B formalism [62], which builds on the earlier classical B-Method [63], (which is still actively used in critical applications in the urban rail sector, e.g. [64]). The mathematical flexibility of the Event-B formalism and the open architecture of its Rodin tool [65,66] lent themselves to supporting verification of hybrid and cyber-physical systems in various ways [67,68,69]. And an *ab initio* reappraisal and extension of the Event-B formalism, not tied to the techniques of the earlier discrete tool, is the Hybrid Event-B formalism [70,71].

While these systems are defined with complete precision, most exhibit some features that, in the light of the arguments of Section 4, we might view as undesirable. Most, for instance, make no distinction between discrete 'program style' variables and 'continuous physical' variables to which 'physics rules' ought to apply (aside from the immediate consequences of their different types). Many do not prevent the mixing of physical and discrete behaviours in the same thread of control, permitting (at least in principle) gaps in which physical behaviour is incomplete. Some do not monitor continuous behaviour closely enough to show safety continuously, unless specific precautions are taken. Some do not temporally isolate discrete (i.e. discontinuous) state changes —which, while comprehensible for program variables, where having several state changes at the same moment of time idealises close-to-instantaneous program execution— is unphysical when applied to genuine physical variables.

[8] Although it is not possible to have states that are both safe and unsafe, it is possible to imagine states that are neither. In such a situation, the nominated safe region may exclude states which are not hazardous (i.e. it is overly conservative), and/or, the nominated unsafe region may exclude states which are not immediately unsafe but which may become so (i.e. it is overly optimistic). We need not delve into these possibilities.

[9] It has to be said that many systems in the literature are not defined with sufficient precision to do this, and they avoid trouble by restricting attention to cases that are 'obviously well behaved'.

The one system that we are aware of that has been expressly designed to avoid the kind of traps regarding unphysical continuous behaviour and verification that we discussed earlier in this paper is Hybrid Event-B [70,71], mentioned above. The cited work contains a much more thorough discussion of these issues in the Hybrid Event-B context.

It has to be said that while we are not aware of any serious consequences of the features we have highlighted in existing systems, a residual risk of inappropriate system modelling nevertheless remains if designers are not sufficiently aware of the implications of physical systems.

Thus far our discussion has avoided mentioning noise or randomness. This is legitimate when the physical considerations imply that it is negligible, which, in fact, is very much the case in many realistic situations. 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. The inclusion of probabilistic techniques in computing formalisms is a large and active area. A brief indication of the issues that can arise and that are most closely related to the concerns of this paper can be found in [72,73,74,75,76].

If we look at the above as it applies an industrial context, a large percentage of the hybrid and cyber-physical systems work one sees, concerns systems that have an identifiable critical element. For such systems an increasing number of standards have been published that aim to confirm that system production practices in the relevant sphere meet a minimum level of proficiency. Among the newer standards in key fields, we can mention for example DO-178C (for avionics [77]), ISO 26262 (for automotive systems [78]), IEC 62304 (for medical devices [79]), or CENELEC EN 50128 (for railway systems [80]). It is apparent though, that in these instances the methods proposed are still heavily weighted in favour of mandating such things as thoroughly documented processes, specific testing strategies, and other practices heavily rooted in traditional development approaches — all rather far from the formally verified approaches discussed just above. Thus it is fair to say that the critical systems industry is rather conservative. This is understandable to a degree, since prematurely advocating radical new ways of doing things, such as processes heavily reliant on unfamiliar formal techniques that may not enjoy the highest levels of trust (from those responsible for decision making in the industrial sphere), risks major disasters in the field (from the point of view of such traditional managements).

Thus the entry of formal techniques into the standardised critical systems development portfolio is rather cautious, despite the strong evidence in niche quarters, and in the academic field, about the dependability that can be gained by appropriate use of formal development, when it is suitably integrated into the wider system engineering process.

One can conjecture that this is as much because entrenched industrial practice cannot move as nimbly as one might ideally wish, even when the evidence for attempting to do so is relatively strong. And the reasons for that range from simple human reluctance to change familiar ways that people do things, to genuine concern about the integrity, reliability, and even the legal defensibility of a novel production process, particularly if it is intended for a heavily regulated sphere.

9 | CONCLUSION

Motivated by the current dramatic proliferation in critical and cyber-physical 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 cyber-physical 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 sufficiently into account 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, resulting in our CDPPP language. The remaining issues need to be addressed within the semantics of CDPPP.

We illustrated our particular solution with a simplified air conditioning system. We first developed it in the relatively naïve style of our starting language, showing how the core steady state behaviour could be captured. Having improved the language, we then redeveloped the air conditioning system in CDPPP, showing how the restructured language leads to a better partition of concerns within the syntactic and semantic description.

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 differ in the low level detail — some of these variations were touched on in the discussion of the formal semantics of CDPPP. 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, CDPPP) is to be particularly recommended for actual critical cyber-physical system development. Not least this is because we chose to rely on the *existence* of solutions to the calculationally difficult fragments of the language, rather than pursuing criteria that made achieving a solution computable (which is the goal of a very large proportion of work on the formal aspects

of cyber-physical systems). But, 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 in practice.

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 then superseded by higher level languages, which are nowadays superseded by sophisticated IDEs, etc. In each case the desire was to raise the level of abstraction in such a way as to preclude as many perceived 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 cyber-physical systems are contemplated in the future.

How to cite this article: R. Banach, H. Zhu (9999), Language Evolution and Healthiness for Critical Cyber-physical Systems, *Journal of Software: Evolution and Process*, *9999;12:345-678*.

APPENDIX

A TECHNICALITIES REGARDING DIFFERENTIAL EQUATIONS

If we write a general first order differential equation as $\Phi(\mathbf{v}, \mathcal{D}\mathbf{v}, t) = 0$, where $\mathbf{v}$ is some tuple of real variables, $\mathcal{D}\mathbf{v}$ is a corresponding tuple of real variables intended to denote the derivatives of $\mathbf{v}$, 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.[8], 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.[10] The best known such class covers first order families that can be written in the form:

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

Here, the left hand form refers to a closed system of variables $\mathbf{x}$, whereas the right hand form also permits the presence of additional external controls $\mathbf{y}$. As well this syntactic shape, conditions have to be demanded on the vector of funtions F and on the entry conditions of the behaviour-to-be that we are trying to define using these techniques.

For simplicity, we assume that the vector of functions F is defined on a real, closed rectangular region, where for each $\mathbf{x}$ component index i we have a Cartesian component $x_i \in [x_{iL} \ldots x_{iU}] \subseteq \mathbb{R}$, and for each $\mathbf{y}$ component index j we have a Cartesian component $y_j \in [y_{jL} \ldots y_{jU}] \subseteq \mathbb{R}$, and where the time dependence of F has been normalised to a clock $\tau \in [0 \ldots \tau_f]$, 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 $\mathbf{x}, \mathbf{y}$ components, and T refers to clock time. We write X for just the $\mathbf{x}$ components and Y for just the $\mathbf{y}$ components, so that $XY = X \times Y$.

To guarantee existence of a solution it is enough to demand that F must satisfy a Lipschitz condition:

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

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[81,82]). Additionally, we require that F is continuous in time for all $(\mathbf{x}, \mathbf{y}) \in XY$.

With the above in place, if $\mathbf{x}_0$ is an initial value for $\mathbf{x}$ such that $\mathbf{x}_0 \in X$, then the standard theory for existence and uniqueness of solutions to DE systems guarantees us a solution $\mathbf{x}(\tau)$ for $\tau \in [0 \ldots \tau_{\mathbf{x}_0}]$, where $\tau_{\mathbf{x}_0} \leq \tau_f$, with $\mathbf{x}(\tau)$ differentiable in the interval $[0 \ldots \tau_{\mathbf{x}_0}]$ and satisfying the DE system, and such that we have $\forall \tau \bullet \tau \in [0 \ldots \tau_{\mathbf{x}_0}] \Rightarrow \mathbf{x}(\tau) \in X$. See[8] for details.

For convenience below, let us abbreviate the syntactic construct $[\mathrm{iv}]\,\mathcal{D}\,\mathbf{x} = F(\mathbf{x}, \mathbf{y}, \tau)$ **until** g, to $[\mathrm{iv}]\,\mathcal{DE}$ **until** g. We now observe that for soundness, on the one hand we need to assume that all the properties discussed above regarding F hold. But on the other hand, in the syntax of a practical language, it is impractical to include all the data needed to establish this suite of properties. Compounding the problem is the fact that even if we included all the required data syntactically in the language, it would not absolve us from the obligation of *proving* all the required properties on the basis of the given data — which is not necessarily trivial by any means in the general case.

[10] In fact there are many possibilities for this, familiar in the DE world. We just pick on a familiar and straightforward case.

On pragmatic grounds we therefore take the view that the presence of F in the $\mathcal{DE}$ language construct is accompanied, behind the scenes, by the needed machinery for doing all that is required. What this boils down to in practice, is a syntactic check that F matches one or other specific pattern for which it is known that the required generic properties hold, and therefore, that no explicit proving has to take place. This is the standard pragmatic approach to DEs.

B CLOSURE PROPERTIES OF GUARDS AND THEIR IMPACT

Our various languages feature guard constructs of various kinds. Some of them are discrete, such as $@\mathsf{b}$ and $\#\mathsf{r}$. Of these $@\mathsf{b}$ has the characteristic that it changes from uncompleted (i.e. with more time to run) to completed, in a single step, as the discrete variables making up b change value in a discrete manner. In the case of $\#\mathsf{r}$, if r is an expression of discrete variables and time progresses in discrete time units, a similar state of affairs applies. In these cases there is no ambiguity about when, in the course of time, these constructs complete.

Other guard constructs have a continuous nature, for example, $@\mathsf{g}$, the guard g in $[\mathsf{iv}]\,\mathcal{DE}\ \mathbf{until}\ \mathsf{g}$ and $[\mathsf{iv}]$ itself in the language Pr1 (though *not* in Pr2, where it either $\mathbf{skip}$s or $\mathbf{abort}$s). Since the expressions defining these contain continuous variables, and time also moves continuously in all the relevant languages, they may define sets of states which are entered (thus discharging the guard) at imprecisely defined points in time.

An example will illustrate. Suppose the guard $@\mathsf{g}$ is $@(\mathsf{x} > 1)$, and that at $\mathsf{t} = 0$, $\mathsf{x}(0) = 0$ and that x increases at unit rate. At what moment is the guard $@(\mathsf{x} > 1)$ first satisfied? The obvious answer, namely $\mathsf{t} = 1$ is not strictly speaking correct, since at $\mathsf{t} = 1$ we have $\mathsf{x}(1) = 1$, which is a value which is not strictly greater than 1. In fact there is *no* time at which $@(\mathsf{x} > 1)$ is first satisfied, since if $\mathsf{t_f}$ were to be the first time, then on the one hand $\mathsf{t_f} > 1$, and on the other $(1 + \mathsf{t_f})/2$ is a strictly earlier time at which $@(\mathsf{x} > 1)$ is satisfied. Contradiction. But the semantics requires a specific time at which the transition in behaviour prompted by the guard takes place.

There are two evident approaches to resolving this impasse. In the first, we simply insist that the sets of states defined by such guards are (topologically) **closed**. In this case $@(\mathsf{x} > 1)$ is forbidden, whereas $@(\mathsf{x} \geq 1)$ is permitted. The semantics of Section 7 assumes this implicitly, and we took this line in this paper for expository simplicity. In the other, a guard construct like $@(\mathsf{x} > 1)$ is simply *interpreted* via its topological closure, yielding $@(\mathsf{x} \geq 1)$. Since the topological closure is a generic concept for the kind of $\mathbb{R}$-valued sets of states that we are interested in, this approach leads to a well defined time, and we have a way of interpreting arbitrary guards. Note though, that this is quite a subtle point.

What we have just been discussing impacts the semantics of the $[\mathsf{iv}]\,\mathcal{DE}\ \mathbf{until}\ \mathsf{g}$ construct. Along with the properties of F, we need to know that on entry to $\mathcal{DE}$, the iv properties hold. This means that $[\mathbf{x}_0 \in \mathsf{X} \wedge \mathsf{P}(\mathbf{x}_0)]$ has to hold, where $\mathsf{P}(\mathbf{x}_0)$ denotes any properties needed to establish iv that go beyond the simple domain requirement $\mathbf{x}_0 \in \mathsf{X}$. The semantics of iv is then as for any other guard depending on which language we have in mind. In both Pr1 and Pr2, if iv holds, then the guard succeeds immediately, and execution of $\mathcal{DE}$ commences. If iv fails, then in Pr1 the process pauses, a clock is started, and it runs until the program state makes iv true, at which point the guard succeeds. For the latter to be well defined in our semantics, the true-set of iv has to be closed, as noted. If iv fails and we are considering Pr2, then the whole execution $\mathbf{abort}$s, also as noted.

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 $\mathsf{F}(\mathbf{x}, \mathbf{y}, \tau)$. We are guaranteed that the solution exists for some period of time.[11]

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

References

1. Alur R. *Principles of Cyberphysical Systems*. MIT Press. 2015.
2. Lee E, Shesha S. *Introduction to Embedded Systems: A Cyberphysical Systems Approach*. LeeShesha.org. 2nd. ed. 2015.
3. Hoare T, , He J. *Unifying Theories of Programming*. Prentice-Hall. 1998.
4. Louden K, Lambert K. *Programming Languages Principles and Practices*. CENGAGE. 2011.
5. Scott M. *Programming Language Pragmatics*. Morgan Kaufmann. 2016.

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

6. Banach R, Zhu H. Shared-Variable Concurrency, Continuous Behaviour and Healthiness for Critical Cyberphysical Systems. In: Proc. FTSCS-16, Vol. 694. Springer, CCIS; 2017: 109-125.

7. Zhou C, Hoare T, Ravn A. A Calculus of Durations. *Inf. Proc. Lett.* 1991; 40: 269-276.

8. Walter W. *Ordinary Differential Equations*. Springer. 1998.

9. Manna Z, Pnueli A. Verifying Hybrid Systems. In: . Hybrid Systems, Vol. 736. Springer, LNCS; 1993: 4-35.

10. Alur R, Courcoubetis C, Henzinger T, Ho PH. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In: Proc. Workshop on Theory of Hybrid Systems, Vol. 736. Springer, LNCS; 1993: 209-229.

11. Alur R, Dill D. A Theory of Timed Automata. *Theor. Comp. Sci.* 1994; 126: 183-235.

12. He J. From CSP to Hybrid Systems. In: Roscoe ed. *A Classical Mind, Essays in Honour of C.A.R. Hoare* Prentice-Hall; 1994: 171-189.

13. Lynch N, Segala R, Vaandrager F, Weinberg H. *Hybrid I/O Automata*. Springer. 1996.

14. Friesen V, Nordwig A, Weber M. Object-Oriented Specification of Hybrid Systems using UML, h and ZimOO. In: Proc. ZUM-98, Vol. 1493. Springer, LNCS; 1998: 328-346.

15. Friesen V, Nordwig A, Weber M. Toward an Object-Oriented Design Methodology for Hybrid Systems. *Object-Oriented Technology and Computing Systems Re-Engineering* 1999: 1.

16. Zhou C, Wang J, Ravn A. A Formal Description of Hybrid Systems. In: Proc. HS-95, Vol. 1066. Springer, LNCS; 1995: 511-530.

17. Grosu, R. and Stauner, T. and Broy, M. A Modular Visual Model for Hybrid Systems. In: Proc. FTRTFT-98, Vol. 1486. Springer, LNCS; 1998: 75-91.

18. Stauner T, Rumpe B, Scholz P. Hybrid System Model. 1999. Technische Universitat Munchen, TUM-I9903.

19. Deshpande A, Göllü A, Varaiya P. SHIFT: A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata. In: Proc. Hybrid Systems IV, Vol. 1273. Springer, LNCS; 1997: 113-133.

20. Back RJ, Petre L, Porres I. Continuous Action Systems as a Model for Hybrid Systems. *Nordic J. Comp.* 2001; 8: 2-21. Extended version of FTRTFT-00, LNCS Vol. 1926, 202-213.

21. Lynch N, Segala R, Vaandrager F. Hybrid I/O Automata. *Information and Computation* 2003; 185: 105-157. MIT Technical Report MIT-LCS-TR-827d.

22. Bender K, Broy M, Peter I, Pretschner A, Stauner T. Model Based Development of Hybrid Systems: Specification, Simulation, Test Case Generation. In: Modelling, Analysis, and Design of Hybrid Systems, Vol. 279, Springer, LNCS. 2002.

23. 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.

24. Kesten Y, Manna Z, Pnueli A. Verification of Clocked and Hybrid Systems. *Acta Informatica* 2000; 36: 837-912.

25. Clarke E, Fehnker A, Han Z, Krogh B, Stursberg O, Theobald M. Verification of Hybrid Systems Based on Counterexample-Guided Abstraction Refinement. In: Proc. TACAS-03, Vol. 2619. Springer, LNCS; 2003: 192-207.

26. Audemard G, Bozzano M, Cimatti A, Sebastiani R. Verifying Industrial Hybrid Systems with MathSAT. In: Proc. BMC-04, Vol. 119. ENTCS; 2005: 17-32.

27. Cimatti A, Roveri M. Requirements Validation for Hybrid Systems. In: Proc. CAV-09, Vol. 5643. Springer, LNCS; 2009: 188-203.

28. Frehse G, Le Guernic C, Donzé A, et al. SpaceEx: Scalable Verification of Hybrid Systems. In: Proc. CAV-11, Vol. 6806. Springer, LNCS; 2011: 379-395.

29. Carloni L, Passerone R, Pinto A, Sangiovanni-Vincentelli A. Languages and Tools for Hybrid Systems Design. *Foundations and Trends in Electronic Design Automation* 2006; 1: 1-193.

30. Henzinger T, Ho PH, Wong-Toi H. HyTech: A Model Checker for Hybrid Systems. *Int. J. Tools Tech. Trans.* 1997; 1: 110-122.

31. Asarin E, Dang T, Maler O. The d/dt Tool for Verification of Hybrid Systems. In: Proc. CAV-02, Vol. 2404. Springer, LNCS; 2002: 365-370.

32. Frehse G. PHaVer: Algorithmic Verification for Hybrid Systems past HyTech. *Int. J. Tools Tech. Trans.* 2008; 10: 263-279.

33. Polyanin A, Zaitsev V. *Handbook of Ordinary Differential Equations: Exact Solutions, Methods, and Problems*. C.R.C. Press. 2018.

34. Hairer E, Norsett S, Wanner G. *Solving Ordinary Differential Equations I Nonstiff Problems*. Springer. 1993.

35. Hairer E, Wanner G. *Solving Ordinary Differential Equations II Stiff and Differential-Algebraic Problems*. Springer. 1996.

36. Ptolemaeus C. (ed.) *System Design, Modeling, and Simulation Using Ptolemy II*. Ptolemy.org. 2014.

37. Wolf W. Cyber-Physical Systems. *IEEE Computer* 2009; 43: 88-89.

38. Sztipanovits J. Composition of Cyber-Physical Systems. In: Proc. 14th. Int. Conf. Eng. Comp.-Based Sys., IEEE; 2007: 3-6.

39. Sztipanovits J. Model Integration and Cyber Physical Systems: A Semantics Perspective. In: Proc. FM-11, Vol. 6664. Springer, LNCS; 2011: 1. Invited talk.

40. Willems J. Open Dynamical Systems: Their Aims and their Origins. Ruberti Lecture, Rome. 2007. http://homes.esat.kuleuven.be/~jwillems/Lectures/2007/Rubertilecture.pdf.

41. 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.

42. Ying T, Vuran M, Goddard S, Yue Y, Miao S, Ren S. A Concept Lattice-based Event Model for Cyber-Physical Systems. In: Proc. Conf. Cyber-Phys. Sys., ACM; 2010: 50-60.

43. Thacker R, Jones K, Myers C, Hao Z. Automatic Abstraction for Verification of Cyber-Physical Systems. In: Proc. Conf. Cyber-Phys. Sys., ACM; 2010: 12-21.

44. Egyed A. A Roadmap for Engineering Safe and Secure Cyber-Physical Systems. In: Proc. MEDI-18 Workshops, Vol. 929. Springer, CCIS; 2018: 113-114. Invited talk.

45. Geisberger E, Broy M. (eds.) 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.

46. Hoare C. An Axiomatic Basis for Computer Programming. *Comm. A.C.M.* 1969; 12: 576-580.

47. Dijkstra E. *A Discipline of Programming*. Prentice-Hall. 1976.

48. Apt K. Ten Years of Hoare's Logic: A Survey Part I. *A.C.M. Trans. Prog. Lang. Sys.* 1981; 3: 431-483.

49. Haddad W, Chellaboina V. *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press. 2008.

50. Sontag E. *Mathematical Control Theory*. Springer. 1998.

51. Hinrichsen D, Pritchard A. *Mathematical Systems Theory I*. Springer. 2005.

52. Prajna S, Jadbabaie A. Safety Verification of Hybrid Systems Using Barrier Certificates. In: Proc. HSCC-04, Vol. 2289. Springer, LNCS; 2004: 477-492.

53. Kong H, He F, Song X, Hung W, Gu M. Exponential-Condition-Based Barrier Certificate Generation for Safety Verification of Hybrid Systems. In: Proc. CAV-13, Vol. 8044. Springer, LNCS; 2002: 242-257.

54. Dai L, Gan T, Xia B, Zhan N. Barrier Certificates Revisited. *J. Symb. Comp.* 2017; 80: 62-86.

55. Liu J, Lv J, Quan Z, et al. A Calculus for Hybrid CSP. In: Proc. APLAS-10, Vol. 6461. Springer, LNCS; 2010: 1-15.

56. Zhan N, Wang S, Zhao H. Hybrid CSP. In: Formal Verification of Simulink/Stateflow Diagrams: A Deductive Approach, Springer; 2017: 71-90.
57. Platzer A. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer. 2010.
58. Platzer A. *Logical Foundations of Hybrid Systems*. Springer. 2018.
59. Symbolaris. http://www.symbolaris.org.
60. Back RJ, Wright vJ. *Refinement Calculus: A Systematic Introduction*. Springer. 1998.
61. Back RJ, Pete L, Porres I. Generalising Action Systems to Hybrid Systems. In: Proc. FTRTFT-00, Vol. 1926. Springer, LNCS; 2000: 202-213.
62. Abrial JR. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press. 2010.
63. Abrial JR. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press. 1996.
64. Clearsy. http://www.clearsy.com/en/.
65. RODIN Tool. http://www.event-b.org/.
66. Abrial JR, Butler M, Hallerstede S, Hoang T, Mehta F, Voisin L. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Int. J. Soft. Tools Tech. Trans.* 2010; 12: 447-466.
67. Su W, Abrial JR, Zhu H. Formalising Hybrid Systems with Event-B and the Rodin Platform. *Sci. Comp. Prog.* 2014; 94: 164-202.
68. Su W, Abrial JR. Aircraft Landing System: Approaches with Event-B to the Modelling of an Industrial System. *Int. J. Soft. Tools Tech. Trans.* 2017; 19: 141-166.
69. Butler M, Abrial JR, Banach R. Modelling and Refining Hybrid Systems in Event-B and Rodin. In: From Action System to Distributed Systems: The Refinement Approach, CRC Press; 2016: 29-42.
70. Banach R, Butler M, Qin S, Verma N, Zhu H. Core Hybrid Event-B I: Single Hybrid Event-B Machines. *Sci. Comp. Prog.* 2015; 105: 92-123.
71. Banach R, Butler M, Qin S, Zhu H. Core Hybrid Event-B II: Multiple Cooperating Hybrid Event-B Machines. *Sci. Comp. Prog.* 2017; 139: 1-35.
72. Zhu H, Qin S, He J, Bowen J. PTSC: Probability, Time and Shared-Variable Concurrency. *Innov. Syst. Softw. Eng.* 2009; 5: 271-284.
73. 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.* 2012; 81: 2-25.
74. Banach R. Contemplating the Addition of Stochastic Behaviour into Hybrid Event-B. In: Proc. TASE-14, IEEE; 2014: 42-49.
75. Banach R. Stochastic Analogues of Invariants: Martingales in Stochastic Event-B. In: Proc. ENASE-15, INSTICC; 2015: 238-243.
76. Peng Y, Wang S, Zhan N, Zhang L. Extending Hybrid CSP with Probability and Stochasticity. In: . Proc. SETTA-15, 9409. Springer, LNCS; 2015: 87-102.
77. DO-178C. http://www.rtca.org.
78. ISO 26262. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=54591.
79. IEC 62304. https://webstore.iec.ch/publication/22794.
80. CENELEC EN 50128. https://www.cenelec.eu/dyn/www/f?p=104:105.
81. Horn R, Johnson C. *Matrix Analysis*. Cambridge University Press. 1985.
82. Horn R, Johnson C. *Topics in Matrix Analysis*. Cambridge University Press. 1991.