A Core Language for Executable Models of Cyber-Physical Systems (Preliminary Report)

Walid Taha
Halmstad University, Halmstad, Sweden
Rice University, Houston, TX, USA
Walid.Taha@hh.se

Paul Brauner, Yingfu Zeng, and Robert Cartwright
Rice University, Houston, TX, USA
{polux2000, yingfuzeng, corky.cartwright}@gmail.com

Veronica Gaspes
Halmstad University, Halmstad, Sweden
Veronica.Gaspes@hh.se

Aaron Ames
University of Texas A&M, College Station, TX, USA
aames@tamu.edu

Alexandre Chapoutot
ENSTA ParisTech, Paris, France
alexandre.chapoutot@ensta-paristech.fr

***Abstract*— Recently we showed that an expressive class of mathematical equations can be automatically translated into simulation codes. By focusing on the expressivity of equations formed from continuous functions, this work did not accommodate a wide range of discrete behaviors or a dynamic collection of components. However, the interaction between continuous and hybrid components in many cyber-physical domains is highly coupled, and such systems are often highly dynamic in both respects. This paper gives an overview of a proposed core language for capturing executable hybrid models of highly dynamic cyber-physical systems.**

Keywords- Modeling, Simulation, Cyber-Physical Systems.

I. INTRODUCTION

Systems that evolve over a dense notion of time interact in complex ways that can confound both intuition and analytical methods [11, 12]. This problem is acute for non-linear differential systems, which include virtually all three-dimensional mechanical systems, and for which solutions rarely have closed form descriptions [13]. As a result, successful analysis and design of novel cyber-physical systems invariably includes an extensive experimental component that relies either on physical prototypes or on simulation. Today, both types of experimentation can be prohibitively costly and slow. Physical experiments incur material costs and pose challenges in control and reproducibility, measurement and instrumentation, and safety. Simulation has the potential to reduce these problems and to significantly accelerate innovation. But current methods either raise questions about fidelity or are extremely labor intensive. Using mainstream tools means depending on proprietary, black-box codes that come with a fixed set of component models and that offer only limited support for building custom models. Writing simulation codes by hand requires both effort and specialized expertise in mapping the high-level analytical models to executable codes, software implementation (including debugging and testing), and dealing with issues of floating point numerical precision. These difficulties can be debilitating for designers of cyber-physical systems, especially novel and creative designs.

To reduce the cost of building simulations, we recently developed and presented an automated, scalable mapping from an expressive class of mathematical equations to code, showing that this natural mathematical formalism can be viewed as an executable language for modeling mechanical systems [2]. While the examples used to illustrate the method focused on the mechanics, the formalism, called Acumen, can be used for other physical domains such as electrical, hydraulic, or heat transfer systems.

Originally, Acumen was developed as an extension of event-driven formalisms [5, 6, 7] that have a similar flavor to synchronous languages [14]. Acumen added systems of equations on functions on dense time for the purpose of describing the "physical environment" surrounding the "purely cyber controllers" that were already describable in synchronous formalisms. Although this approach seems intuitive at first, it makes an unnecessary association between physical and continuous and cyber and discrete. In reality, physical environments often exhibit both continuous and discrete behaviors. For example, the two legs of a walking robot continually and discretely change modality and role with each step forward. Dually, the use of a purely discrete model for controllers or embedded systems is overly restrictive. In early stages of design one may use purely continuous controller models for simplicity. In later stages of design, it may be important to capture physical aspects of a digital implementation, such as dense-time behavior, delays, energy consumption, or heat emissions. Thus, the expressivity needed to model both the physical and cyber components calls for tight integration between continuous and discrete behaviors.

This paper describes the key features of a new, more uniform design for Acumen. The proposed design allows fine-grained coupling between continuous and discrete behaviors in a unified notion of a hybrid object. Such objects have time-varying state, carry hybrid laws that specify their behavior, can be dynamically created and terminated, and can include and dynamically coordinate "child" objects.

In the rest of the paper, we review closely related work (Section II), summarize the proposed design (Section III), describe a case study aimed at testing the expressivity of the language (Section IV), and describe how it is simulated (Section V).[1]

II. Related Work

Acumen [3, 2, 1] is a modeling language being developed with the goal of bridging the gap between several important efforts in modeling and simulation, hybrid systems verification, and synchronous languages. In what follows we describe how the proposed design relates to other efforts.

The purely discrete event-driven predecessors [5, 6, 7] of Acumen have their roots in Functional Reactive Programming (FRP) [4], which itself supports both continuous and discrete behaviors in a purely functional setting. In formulating the predecessors of Acumen, we narrowed the functional framework to purely discrete systems to focus on the real-time properties of embedded controllers.

Modelica's support for equation-based (or relational) modeling [8] provided the initial inspiration for Acumen's equations on functions of dense time. Going beyond Modelica and other equation-based languages, the full Acumen language supports partial derivatives that can be used to specify systems using Euler-Lagrange equations, which still can be symbolically eliminated by translation to time derivatives.

Like CHARON [9], Acumen is a hybrid systems simulation language inspired by hybrid automata [11,12] and hybrid logics [10]. Acumen differs from CHARON in being untyped, deterministic, and built on a single, dynamic notion of object. We present a more detailed comparison after Core Acumen has been introduced (Section III).

Dynamic differential logic [10] encouraged us to explore a more "imperative" style of describing an object's state, and which is reflected in design presented in this paper. A key difference between hybrid logics and languages aimed at simulation (such as FRP, Modelica, and Acumen) is the treatment of non-determinism. Non-determinism is advantageous in formalisms used for automated reasoning, because it can be used to weaken assumptions and thus strengthen the established properties. But non-determinism is highly problematic for simulation formalisms, because it may require exploring a vast number of options for bounded domains, and is simply not possible for unbounded domains.

Allowing discrete computations to be repeated arbitrarily until there are no more additional changes is a standard way for computing a fixed point. Synchronous languages [14] such as Lustre or Esterel use a similar strategy for converging on the result of a synchronous system. In the proposed design we compute the fixed point for the state of the whole model being simulated. By the Bekic theorem, this produces the same result as computing the fixed point for all components of the system independently [17].

[1] An earlier version of this paper appeared Work in Progress (WIP) session of ICCPS 2011. Section IV is entirely new.

III. Core Acumen

Acumen's semantics is defined by a series of translations from a large source language into progressively smaller subsets of the language. The purpose of the core language is to serve as the minimal subset needed to express all the features of the full source language. In this section, the proposed design for Acumen's core language is illustrated by a series of small examples.

Acumen is implemented as free software and is available along with a hands-on tutorial from the Acumen website [1]. All examples can be simulated and visualized using the online distribution, version 10.12.13. The implementation and the tutorial include more examples than we provide in this description of the core language. The tutorial also presents the grammar (BNF), explains class parameters, how to define local notions of time, how to use the graphical user interface, and describes other features of the language that natural extend the subset presented here.

A. Objects and Hybrid Laws

Acumen objects are introduced by defining a class for each kind of object, and then by creating instances of these classes at a particular point in model/simulation time. As an example, consider a device consisting of a battery and discrete controller that decides whether the battery should charge or drive a load. When charging, the voltage on the battery increases at a constant rate until it reaches its full capacity. Then, the battery stops charging and starts driving the load until the voltage is too low. When that happens, the battery switches back to charging. In Core Acumen, the device is modeled as follows:

```
class Contraption ()
  private v = 0; v' = 0; mode = 0 end
  switch mode
    case 0 // Charge (until high)
      if (v < 0.8)  v' [=] 1/2
      else  mode = 1   end
    case 1 // Drive (until low)
      if (v > 0.2) v' [=] -v
      else  mode = 0  end
  end
end
```

All variables (such as `v`, `v'`, and `mode` in the first class) are implicitly functions of time [4]. Variables introduced in the `private` section of the class are the state of any object of this class. The value assigned to each variable is the initial value it has at the instant the object is created. A prime (`'`) following a variable name denotes its derivative with respect to time. Thus, given an initial value for the variable `v`, if `v'` is defined, then the value of `v` will be automatically

determined for future values as well. The `switch` statement allows different sets of rules to govern the behavior of the system under different conditions. The `case` that applies is determined by the value of the variable `mode`. When the value of `mode` is `0`, then the first `if` statement is active. When the value is `1`, then the second `if` statement is active. The two `if` statements follow a parallel pattern: their true branch contains a *continuous assignment*, and the false branch contains a *discrete assignment*. The main difference between discrete and continuous assignments is that discrete assignments block the progress of logical time, in the sense that simulation cannot advance beyond a particular point in time until all discrete assignments have been performed. Continuous assignments, in contrast, happen continuously, and pose no particular constraints on how the simulator advances time.

B. Class `Main` and the `simulator` parameter

In any Acumen model there must be a declaration for a class called `Main`. This class always represents the entire world being modeled.

Even though the goal of Acumen is to automate building simulation codes, there are fundamental computability limitations that dictate that not all implementation details can be hidden from the user. For example, there is no single, universal method for solving a system of equations, be it linear, non-linear, time-varying, or differential. Yet bridges and airplanes need to be built, and they will be, whether or not we help engineers write their simulation codes. Thus, a pragmatic decision is made in Acumen to allow the user to include in models additional details needed to perform the simulation. To support this, each `Main` object is required to have a parameter (by convention called the `simulator`) that allows the user to express how the model should be simulated. Continuing the above example, we can write:

```
class Main (simulator)
  private mode = "Init" end
  switch mode
    case "Init"
      simulator.timeStep = 0.001;
      simulator.endTime  = 5.0;
      create Contraption ();
      mode = "Persist"
    case "Persist"
  end
end
```

The default parameters for simulation start time, end time, and step size are 0, 10, and 0.01, respectively. The rest of the definition for this class uses a variable called mode to distinguish between two different states, one for initializing (or creating) the model and the other for letting the model run.

C. Object Life Cycle, Migration, and Regulation

When a `create` command is encountered at a certain point in model/simulation time, an object is introduced to the model. Similarly, objects can be terminated. Initially, any new object is considered the child of the object that created it. To allow objects to regulate the behavior of their children, Acumen allows iteration over children. It is also possible to move objects dynamically from one parent object to another, thereby changing the set of "external laws" that apply to this object.

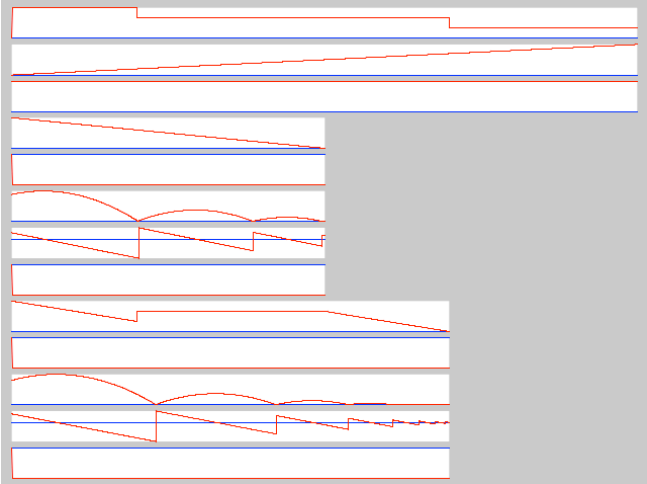
To illustrate these concepts, we will consider an artificial example that allows us to showcase all of these facilities concisely. In this example, we introduce a class for "fancy balls". The basic functionality of these balls is to bounce. In addition, they have a limited life span that is specified by the parent at the time they are created. Additionally, each fancy ball ensures that each child has a lifespan that is at least two seconds longer than that of its parent. The `Main` class will specify a world where two such objects are created at time 0, and then, at time 2, the second ball is moved from the top level world object to be a child object of the first ball. The model capturing this behavior is as follows:

```
class FancyBall (t, x,x',x'')
 private t'=0; end
 x'' [=] -9.8;
 if x<=0 x' = -0.6*x'; x  = -x end;
 t' [=] -1;
 if t<=0 terminate self end;
 for c = self.children
   c.t [=] 2
 end
end

class Main (simulator)
  private
   mode ="Init"; n = 0;  t=0; t'=1;
   a = create FancyBall (5,10,5,0);
   b = create FancyBall (3,10,7,0);
  end
   t' [=] 1;
   n [=] sum 1 for i in self.children
         if true;
  switch mode
     case "Init"
       if t>2
         move b a;
         mode = "Persist" end
     case "Persist"
  end
end
```

The code includes an additional private variable `n` which keeps track of the number of children in the top level world, and illustrates one iteration construct in Acumen (summation). By default, simulating a model in Acumen produces a series of normalized plots. The are normalized based on the minimum and maximum values of the variable during the simulation. No scales are explicitly displayed, but the user can determine the value of any point on a graph by selecting it with the mouse. The series shows the objects and their variables presented in the order that they are created, and for the duration that they exist, and with vertical scales

normalized by the value range. For the model above, the resulting series of plots is as follows:



The first band plots the number of children at the top level. As expected, the number of top-level children drops at time 2, because the second ball has been moved to be a child of the first ball. The nineth band is the variable `t` of the second object stops decreasing linearly and keeps a fixed value (2) as dictated by fancy ball's rule for its children. The number of top-level children does not change when the first object dies because the default behavior is that the grand parent inherits any grandchild that survives a terminated parent.

This example also displays *Zeno behavior*, where an infinite number of discrete transitions occur in a finite amount of time (see [15] for more on formally detecting Zeno behavior and [16] for the semantics of this behavior). The finite time interval for the simulation is evidence of the existence of Zeno behavior in this example, and points to the strong need to consider simulation semantics that account for multiple discrete computations at a single time instance.

D. *Comparison with CHARON*

To illustrate the points made about the relation with CHARON in Related Work (Section II), we consider a simple model of thermostat coming from CHARON's user manual. The temperature `x` of a room is controlled to keep it in the target range of *68-82* degrees Fahrenheit. The temperature can evolve continuously over time. The heater is activated if the value of `x` is less than *70* and the evolution of `x` follows the equation `x' = -x + 100`. If the value of `x` is greater than *80* the heater is off and the temperature follows the rule `x' = -x`.

A CHARON program is made of a set of *agents* that are executed concurrently. Agents may be aggregated to form a more complex agent. Each agent is made of a set of *modes* each representing a state of hybrid automaton system in which only one mode can be activated at a time. A mode may also be made of sub-modes. A set of local or shared *variables* may be associated with agents or with modes. These variables are the main communication technique in CHARON. Each variable is declared with a type (e.g. `int` or `real`), a kind (e.g. `analog` or `discrete`) to express if it is a continuous-time or a discrete-time variable, and access restriction. A special operator `d` is used to represent derivatives (see mode `onOff`). CHARON explicitly declares guarded transitions between modes. An action may be associated in case of the transition is taken. Finally, an invariant property may be associated with each mode. If the invariant is false, then the automata must transition to another mode. If it does not, it is considered *blocked*. The implementation of the example described above is as follows:

```
agent thermostat(){
  mode top = thermostatTop()
}
mode thermostatTop(){
  private analog real x;
  mode on =
   onOff(-10000000000.0, 82.0, 100.0);
  mode off =
   onOff(68.0, 10000000000.0, 0.0);
  trans toSubMode from default
        to on when true do {x= 73}
  trans fromOnToOff from on
        to off when x > 80.0 do {}
  trans fromOffToOn from off
        to on when x < 70.0 do {}
}
mode onOff(real a, real b, real c){
  readWrite analog real x;
  inv invOnOff {x > a and x < b}
  diff dOnOff {d(x) == -x + c}
}
```

Deterministic CHARON programs can be expressed in Acumen as follows. Agents are mapped to classes. Local variables are mapped to local variables, and additional variables are introduced as needed for derivatives by looking at the rest of the agent definition. Modes are mapped to strings that can be assigned to a mode variable for that class. A switch statement is then used to capture the rules for that mode. To deal with invariants, an extra mode called Blocked is introduced, and the invariants are test at the end of the case for each switch value (mode).

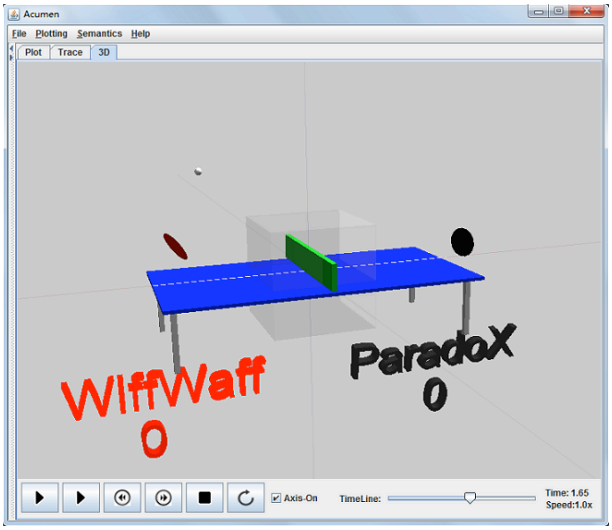
```
class Thermostat(mode)
 private x = 0; x' = 0 end
 switch mode
   case "Top" x = 73; mode = "On"
   case "On"  x' [=] -x + 100;
     // Transition from on to off
     if x > 80 mode = "Off" end;
     // Invariant invOnOff
     if not (x>-10000000000.0 && x < 82.0)
      mode = "Blocked" end;
   case "Off" x' [=] -x;
     // Transition from off to on
     if x < 70 mode = "On" end;
     // Invariant invOnOff
     if not (x>68.0 && x < 10000000000.0)
        mode = "Blocked" end;
 case "Blocked"
end
end
```

Thus, Core Acumen is a smaller language that can still naturally express deterministic CHARON programs. As a hierarchical agent language, CHARON supports tree-structured models of the world. However, it is not clear that CHARON supports a notion of agent mobility similar to the one supported by core Acumen.

IV. Case Study: Ping Pong

As a larger case study into the expressivity of this core language, we developed a simplified model of a Ping Pong game. This game was used in the first edition of a CPS course at Halmstad University in the third period of the 2011/2012 academic year. The version we report on was the one used for the second tournament for that course's project work.

The model we describe here consisted of nine classes spanning 618 lines of code. A few of these lines contained spaces, comments, and some declarations to generate a 3D animation to help users visualize the results of the simulation. The following is a snapshot from the animation of a game between a team called WiffWaff and another called Paradox.



The game model includes a class representing a 3D bouncing ball with a simple air resistance model. Core Acumen also supports vectors and basic vector operations, allowing us to express the essential behavior of the model as follows:

```
switch mode
 case "Fly"
   if dot(p,[0,0,1]) < 0
   && dot(p',[0,0,1])< 0
     mode = "Bounce";
   else
     p'' [=]   -k2 * norm(p') * p'
             + [0,0,-9.8];
   end;
 case "Bounce"
   p' = p' .* k_z;
   mode = "Fly";
end
```

Another class represents a mechanical player. This is the class that students developed during the course of the class project. Early on in the project, the player was allowed to send a velocity signal directly to the bat. Later on, the player could only send acceleration signals, and students had to write a controller that computes an acceleration signal `a` based on the velocity signal `v` they had already figured out how to compute. The controller for one of the players was expressed as follows:

```
private
   v   = [0,0,0]; // Bat's speed
   a   = [0,0,0]; // Bat's acceleration
   estimatedBatV  = [0,0,0]; // Predicted
   estimatedBatV' = [0,0,0]; // Computed
   desiredBatP  = [1.6*(-1)^n,0,0.2];
   desiredBatP' = [0,0,0];
 end

desiredBatP' [=] v;

estimatedBatV' [=] a;

a [=]   13*(desiredBatP' - estimatedBatV)
      + 50*(desiredBatP - batp);
```

Here, `desiredBatP` and its derivative are computed based on the `v` signal computed by a motion planning component. In addition, an estimate of the anticipated result of the controllers output is computed by the equation for `estimatedBatV'`. The acceleration `a` itself is computed by the last line, which constitutes a second order linear controller based on both measured and predicted components of the state of the bat as well as on the desired stated.

Two other major classes in the model are `Game` and `Referee`. The game class connects all components together, which is done using continues assignments. It is also responsible for resetting the state when discrete changes occur in the state of the game. The referee class observes the game to keep the score and enforce the rules of the game.

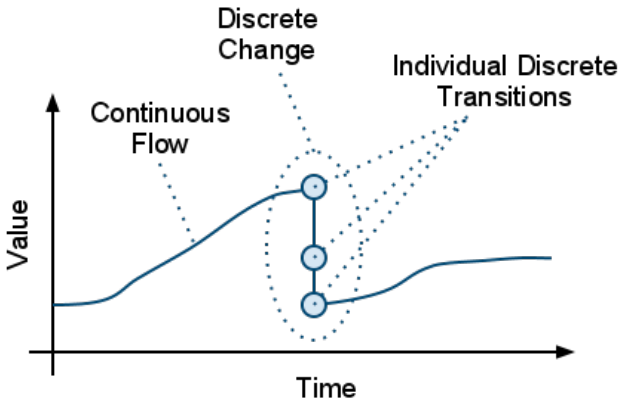
Two regulate the flow of information between components we introduce two styles of classes that we often refer to as *observers* or *actuators*. Two actuators are used in the model, one for the bat and one for the ball. On observer is used for the ball. The actuators make sure that the flow of information out of the player is regulated to enforce realistic constraints. The observer is used to limit the player's ability to observe the state of the ball precisely, and uses discrete sampling to model a common constraint in a cyber-physical system.

The case study increased our confidence in the expressivity of the core language. At the same time, it revealed several

specific concrete issues where the language can be improved. These issues are in areas that include expressivity, semantics, performance, numerical precision, user interface, and concepts that could make modeling physical systems more convenient. We hope to report on and address these issues in an extended version of this paper.

V. Simulation Semantics

Acumen models are simulated by a fine interleaving of a sequences that can consist of multiple discrete computations followed by a single computation updating the values that should evolve continuously. Conceptually, we can think of any instances in time as follows:



In the discrete phase of each sequence, all actions that require discrete change are performed. In the continuous phase, any range of numerical methods and tools for approximating continuous behavior can be used. Thus, simulating what is happening at any single instance in time consists of zero or more discrete steps followed by a single continuous step. The discrete steps capture sudden changes in state such as the impact of two objects, and consist of evaluating all active discrete assignments in the program until the whole system is stabilized. A system is stabilized when no more discrete steps are required. The following example illustrates how discrete assignments are handled:

```
class Main (simulator)
  private x  = 0; y  = 1; z  = 1; end
  if x<5 x = x+1 end;
end
```

The entire model is repeatedly evaluated until the condition in this statement is false. Simulation time (or logical time) is not advanced during these iterations. Acumen considers such changes to all be happening in the same instant. Using this type of global fixed point semantics allows Acumen to realize, among other things, what is sometimes called the "synchrony hypothesis," whereby the author of the model assume that certain discrete or digital events can happen "fast enough" so that we can view them in the rest of the model as happening instantaneously. In the example above, because the initial value of `x` is zero, the iteration will end when x has the value 5.

Once all discrete actions have taken place, the system moves on to performing all adjustments to the continuous state of the system. The continuous step performs all updates in parallel, meaning that all updates are based on the state that results after the sequence of discrete steps, rather than some later state that resulted from other continuous updates.

References

[1] Acumen Website, `http://www.acumen-language.org`.

[2] Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Marcia O'Malley, Robert Cartwright, Aaron Ames, and Raktim Bhattacharya. 2010. Mathematical equations as executable models of mechanical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems* (ICCPS '10).

[3] Yun Zhu, Jun Inoue, Marisa L. Peralta, Walid Taha, Marcia K. O'Malley, Dane Powell. 2009. Implementing Haptic Feedback Environments from High-level Descriptions, *International Conference on Embedded Software and Systems* (SHOES-09).

[4] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles, *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (PLDI'00).

[5] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-Driven FRP, *International Symposium on Practical Aspects of Declarative Languages* (PADL '02).

[6] Zhanyong Wan, Walid Taha, and Paul Hudak. 2001. Real-time FRP. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (ICFP '01).

[7] Roumen Kaiabachev, Walid Taha, and Angela Zhu. 2007. E-FRP with priorities, *ACM & IEEE international conference on Embedded software* (EMSOFT '07).

[8] David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. PhD thesis. Department of Computer and Information Science, Linköping University, Sweden, 2010.

[9] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. 2000. Modular Specification of Hybrid Systems in CHARON. *International Workshop on Hybrid Systems: Computation and Control* (HSCC '00)

[10] Andre Platzer and Jan-David Quesel. 2009. European Train Control System: A Case Study in Formal Verification. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering* (ICFEM '09)

[11] Thomas A. Henzinger, The Theory of Hybrid Automata, 1996. *IEEE Conference on Logic in Computer Science* (LICS'96).

[12] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. 1995. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.* 138, 1 (February 1995), 3-34.

[13] Wassim M. Haddad and VijaySekhar Chellaboina, *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press, 2008.

[14] Nicolas Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers., 1993.

[15] Andrew Lamperski and Aaron D. Ames. 2008. On the Existence of Zeno Behavior in Hybrid Systems with Non-Isolated Zeno Equilibria. 47th IEEE Conference on Decision and Control.

[16] Haiyang Zheng, Edward A. Lee and Aaron D. Ames. 2006. Beyond Zeno: Get on with it! Hybrid Systems: Computation and Control, Volume 3927 of Lecture Notes in Computer Science, 568-582, Springer Verlag.

[17] Glynn Winskel. The Formal Semantics of Programming Languages. Foundation of Computing Series. The MIT Press, 1993.